Raspberry Pi Pico C/C++ SDK
Libraries and tools for
C/C++ development on
RP2040 microcontrollers
Colophon

Copyright © 2020-2024 Raspberry Pi Ltd (formerly 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: 2024-05-02
build-version: 576cee3-clean

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. Source code included in the documentation is Copyright © 2020-2023 Raspberry Pi Ltd (formerly Raspberry Pi (Trading) Ltd.) and licensed under the 3-Clause BSD license.

Legal disclaimer notice

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

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

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

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

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

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

Colophon .................................................................................................................. 1
Legal disclaimer notice ............................................................................................... 1
1. About the SDK ........................................................................................................ 7
   1.1. Introduction ........................................................................................................ 7
   1.2. Anatomy of a SDK Application ......................................................................... 7
2. SDK architecture ..................................................................................................... 10
   2.1. The Build System ............................................................................................. 10
   2.2. Every Library is an INTERFACE ..................................................................... 11
   2.3. SDK Library Structure ..................................................................................... 12
      2.3.1. Higher-level Libraries ............................................................................... 12
      2.3.2. Runtime Support (pico_runtime, pico_standard_link) ............................... 12
      2.3.3. Hardware Support Libraries ..................................................................... 13
      2.3.4. Hardware Structs Library .......................................................................... 14
      2.3.5. Hardware Registers Library ...................................................................... 15
      2.3.6. TinyUSB Port ............................................................................................ 16
      2.3.7. Wi-Fi on Pico W ....................................................................................... 16
      2.3.8. Bluetooth on Pico W .............................................................................. 17
   2.4. Directory Structure .......................................................................................... 18
      2.4.1. Locations of Files ..................................................................................... 18
   2.5. Conventions for Library Functions ................................................................... 19
      2.5.1. Function Naming Conventions .................................................................. 20
      2.5.2. Return Codes and Error Handling ............................................................. 20
      2.5.3. Use of Inline Functions ............................................................................. 21
      2.5.4. Builder Pattern for Hardware Configuration APIs ...................................... 22
   2.6. Customisation and Configuration Using Preprocessor variables ................. 22
      2.6.1. Preprocessor Variables via Board Configuration File ................................. 23
      2.6.2. Preprocessor Variables Per Binary or Library via CMake .......................... 23
   2.7. SDK Runtime .................................................................................................... 24
      2.7.1. Standard Input/Output (stdio) Support ....................................................... 24
      2.7.2. Floating-point Support ............................................................................. 24
      2.7.3. Hardware Divider ..................................................................................... 27
   2.8. Multi-core support ............................................................................................. 28
   2.9. Using C++ ......................................................................................................... 28
   2.10. Next Steps ....................................................................................................... 29
3. Using programmable I/O (PIO) ............................................................................. 30
   3.1. What is Programmable I/O (PIO)? .................................................................. 30
      3.1.1. Background ............................................................................................... 30
      3.1.2. I/O Using dedicated hardware on your PC ............................................... 30
      3.1.3. I/O Using dedicated hardware on your Raspberry Pi or microcontroller .... 30
      3.1.4. I/O Using software control of GPIOs ("bit-banging") ................................. 31
      3.1.5. Programmable I/O Hardware using FPGAs and CPLDs ......................... 32
      3.1.6. Programmable I/O Hardware using PIO ............................................... 32
   3.2. Getting started with PIO ................................................................................... 33
      3.2.1. A First PIO Application .......................................................................... 33
      3.2.2. A Real Example: WS2812 LEDs ............................................................... 37
      3.2.3. PIO and DMA (A Logic Analyser) ............................................................. 44
      3.2.4. Further examples ...................................................................................... 49
   3.3. Using PIOASM, the PIO Assembler ................................................................. 49
      3.3.1. Usage ........................................................................................................ 50
      3.3.2. Directives ................................................................................................. 50
      3.3.3. Values ....................................................................................................... 51
      3.3.4. Expressions .............................................................................................. 52
      3.3.5. Comments ............................................................................................... 52
      3.3.6. Labels ...................................................................................................... 52
      3.3.7. Instructions .............................................................................................. 52
<table>
<thead>
<tr>
<th>3.3.8. Pseudoinstructions</th>
<th>53</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.9. Output pass through</td>
<td>53</td>
</tr>
<tr>
<td>3.3.10. Language generators</td>
<td>54</td>
</tr>
</tbody>
</table>

### 3.4. PIO Instruction Set Reference

<table>
<thead>
<tr>
<th>3.4.1. Summary</th>
<th>59</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.2. JMP</td>
<td>60</td>
</tr>
<tr>
<td>3.4.3. WAIT</td>
<td>61</td>
</tr>
<tr>
<td>3.4.4. IN</td>
<td>62</td>
</tr>
<tr>
<td>3.4.5. OUT</td>
<td>63</td>
</tr>
<tr>
<td>3.4.6. PUSH</td>
<td>64</td>
</tr>
<tr>
<td>3.4.7. PULL</td>
<td>64</td>
</tr>
<tr>
<td>3.4.8. MOV</td>
<td>65</td>
</tr>
<tr>
<td>3.4.9. IRQ</td>
<td>67</td>
</tr>
<tr>
<td>3.4.10. SET</td>
<td>68</td>
</tr>
</tbody>
</table>

### 4. Library documentation

<table>
<thead>
<tr>
<th>4.1. Hardware APIs</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.1. hardware_adc</td>
<td>71</td>
</tr>
<tr>
<td>4.1.2. hardware_base</td>
<td>76</td>
</tr>
<tr>
<td>4.1.3. hardware_claim</td>
<td>78</td>
</tr>
<tr>
<td>4.1.4. hardware_clocks</td>
<td>80</td>
</tr>
<tr>
<td>4.1.5. hardware_divider</td>
<td>87</td>
</tr>
<tr>
<td>4.1.6. hardware_dma</td>
<td>96</td>
</tr>
<tr>
<td>4.1.7. hardware_exception</td>
<td>114</td>
</tr>
<tr>
<td>4.1.8. hardware_flash</td>
<td>116</td>
</tr>
<tr>
<td>4.1.9. hardware_gpio</td>
<td>119</td>
</tr>
<tr>
<td>4.1.10. hardware_i2c</td>
<td>139</td>
</tr>
<tr>
<td>4.1.11. hardware_interp</td>
<td>147</td>
</tr>
<tr>
<td>4.1.12. hardware_irq</td>
<td>156</td>
</tr>
<tr>
<td>4.1.13. hardware_pio</td>
<td>164</td>
</tr>
<tr>
<td>4.1.14. hardware_pll</td>
<td>198</td>
</tr>
<tr>
<td>4.1.15. hardware_pwm</td>
<td>199</td>
</tr>
<tr>
<td>4.1.16. hardware_resets</td>
<td>209</td>
</tr>
<tr>
<td>4.1.17. hardware_rtc</td>
<td>212</td>
</tr>
<tr>
<td>4.1.18. hardware_spi</td>
<td>215</td>
</tr>
<tr>
<td>4.1.19. hardware_sync</td>
<td>222</td>
</tr>
<tr>
<td>4.1.20. hardware_timer</td>
<td>229</td>
</tr>
<tr>
<td>4.1.21. hardware_uart</td>
<td>235</td>
</tr>
<tr>
<td>4.1.22. hardware_vreg</td>
<td>243</td>
</tr>
<tr>
<td>4.1.23. hardware_watchdog</td>
<td>243</td>
</tr>
<tr>
<td>4.1.24. hardware_xosc</td>
<td>247</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4.2. High Level APIs</th>
<th>247</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1. pico_async_context</td>
<td>248</td>
</tr>
<tr>
<td>4.2.2. pico_flash</td>
<td>260</td>
</tr>
<tr>
<td>4.2.3. pico_i2c_slave</td>
<td>262</td>
</tr>
<tr>
<td>4.2.4. pico_multicore</td>
<td>264</td>
</tr>
<tr>
<td>4.2.5. pico_rand</td>
<td>272</td>
</tr>
<tr>
<td>4.2.6. pico_stdlib</td>
<td>274</td>
</tr>
<tr>
<td>4.2.7. pico_sync</td>
<td>276</td>
</tr>
<tr>
<td>4.2.8. pico_time</td>
<td>290</td>
</tr>
<tr>
<td>4.2.9. pico_unique_id</td>
<td>309</td>
</tr>
<tr>
<td>4.2.10. pico_util</td>
<td>310</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4.3. Third-party Libraries</th>
<th>315</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.1. tinyusb_device</td>
<td>315</td>
</tr>
<tr>
<td>4.3.2. tinyusb_host</td>
<td>315</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4.4. Networking Libraries</th>
<th>315</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.1. pico_btstack</td>
<td>316</td>
</tr>
<tr>
<td>4.4.2. pico_lwip</td>
<td>317</td>
</tr>
<tr>
<td>4.4.3. pico_cyw43_driver</td>
<td>320</td>
</tr>
<tr>
<td>4.4.4. pico_cyw43_arch</td>
<td>321</td>
</tr>
</tbody>
</table>

| 4.5. Runtime Infrastructure | 359 |
Table of contents

4.5.1. boot_stage2 .......................................................... 360
4.5.2. pico_base ............................................................. 360
4.5.3. pico_binary_info ..................................................... 361
4.5.4. pico_bit_ops ......................................................... 361
4.5.5. pico_bootrom ........................................................ 362
4.5.6. pico_bootsel_via_double_reset .................................. 365
4.5.7. pico_cxx_options .................................................... 365
4.5.8. pico_divider ......................................................... 365
4.5.9. pico_double .......................................................... 373
4.5.10. pico_float ........................................................... 373
4.5.11. pico_int64_ops ...................................................... 374
4.5.12. pico_malloc ........................................................ 374
4.5.13. pico_mem_ops ....................................................... 374
4.5.14. pico_platform ....................................................... 374
4.5.15. pico_printf .......................................................... 382
4.5.16. pico_runtime ....................................................... 383
4.5.17. pico_stdio .......................................................... 383
4.5.18. pico_standard_link ................................................ 388

4.6. External API Headers .................................................. 388
4.6.1. boot_picoboot ....................................................... 388
4.6.2. boot_UF2 ............................................................. 388

4.7. pico_usb_reset_interface ............................................ 389

Appendix A: App Notes ....................................................... 390
Attaching a 7 segment LED via GPIO ........................................ 390
Wiring information .......................................................... 390
List of Files ....................................................................... 390
Bill of Materials .............................................................. 392
DHT-11, DHT-22, and AM2302 Sensors ...................................... 393
Wiring information .......................................................... 393
List of Files ....................................................................... 394
Bill of Materials .............................................................. 396
Attaching a 16x2 LCD via TTL .............................................. 396
Wiring information .......................................................... 397
List of Files ....................................................................... 397
Bill of Materials .............................................................. 400
Attaching a microphone using the ADC .................................. 401
Wiring information .......................................................... 401
List of Files ....................................................................... 402
Bill of Materials .............................................................. 403
Attaching a BME280 temperature/humidity/pressure sensor via SPI 404
Wiring information .......................................................... 404
List of Files ....................................................................... 404
Bill of Materials .............................................................. 409
Attaching a MPU9250 accelerometer/gyroscope via SPI .............. 409
Wiring information .......................................................... 409
List of Files ....................................................................... 410
Bill of Materials .............................................................. 413
Attaching a MPU6050 accelerometer/gyroscope via I2C .............. 413
Wiring information .......................................................... 413
List of Files ....................................................................... 414
Bill of Materials .............................................................. 416
Attaching a 16x2 LCD via I2C .............................................. 416
Wiring information .......................................................... 417
List of Files ....................................................................... 417
Bill of Materials .............................................................. 420
Attaching a BMP280 temp/pressure sensor via I2C ..................... 421
Wiring information .......................................................... 421
List of Files ....................................................................... 421
Bill of Materials .............................................................. 426
Attaching a LIS3DH Nano Accelerometer via I2C ...................... 426
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wiring information</td>
<td>427</td>
</tr>
<tr>
<td>List of Files</td>
<td>427</td>
</tr>
<tr>
<td>Bill of Materials</td>
<td>430</td>
</tr>
<tr>
<td>Attaching a MCP9808 digital temperature sensor via I2C</td>
<td>430</td>
</tr>
<tr>
<td>Wiring information</td>
<td>430</td>
</tr>
<tr>
<td>List of Files</td>
<td>431</td>
</tr>
<tr>
<td>Bill of Materials</td>
<td>433</td>
</tr>
<tr>
<td>Attaching a MMA8451 3-axis digital accelerometer via I2C</td>
<td>434</td>
</tr>
<tr>
<td>Wiring information</td>
<td>434</td>
</tr>
<tr>
<td>List of Files</td>
<td>434</td>
</tr>
<tr>
<td>Bill of Materials</td>
<td>437</td>
</tr>
<tr>
<td>Attaching an MPL3115A2 altimeter via I2C</td>
<td>437</td>
</tr>
<tr>
<td>Wiring information</td>
<td>438</td>
</tr>
<tr>
<td>List of Files</td>
<td>438</td>
</tr>
<tr>
<td>Bill of Materials</td>
<td>442</td>
</tr>
<tr>
<td>Attaching an OLED display via I2C</td>
<td>442</td>
</tr>
<tr>
<td>Wiring information</td>
<td>443</td>
</tr>
<tr>
<td>List of Files</td>
<td>444</td>
</tr>
<tr>
<td>Bill of Materials</td>
<td>454</td>
</tr>
<tr>
<td>Attaching a PA10100 Mini GPS module via I2C</td>
<td>455</td>
</tr>
<tr>
<td>Wiring information</td>
<td>455</td>
</tr>
<tr>
<td>List of Files</td>
<td>455</td>
</tr>
<tr>
<td>Bill of Materials</td>
<td>458</td>
</tr>
<tr>
<td>Attaching a PCF8523 Real Time Clock via I2C</td>
<td>459</td>
</tr>
<tr>
<td>Wiring information</td>
<td>459</td>
</tr>
<tr>
<td>List of Files</td>
<td>459</td>
</tr>
<tr>
<td>Bill of Materials</td>
<td>462</td>
</tr>
<tr>
<td>Interfacing 1-Wire devices to the Pico</td>
<td>463</td>
</tr>
<tr>
<td>Wiring information</td>
<td>463</td>
</tr>
<tr>
<td>Bill of materials</td>
<td>464</td>
</tr>
<tr>
<td>List of files</td>
<td>464</td>
</tr>
<tr>
<td>Communicating as master and slave via SPI</td>
<td>471</td>
</tr>
<tr>
<td>Wiring information</td>
<td>471</td>
</tr>
<tr>
<td>Outputs</td>
<td>472</td>
</tr>
<tr>
<td>List of Files</td>
<td>473</td>
</tr>
<tr>
<td>Bill of Materials</td>
<td>477</td>
</tr>
<tr>
<td>Appendix B: SDK configuration</td>
<td>478</td>
</tr>
<tr>
<td>Configuration Parameters</td>
<td>479</td>
</tr>
<tr>
<td>Appendix C: CMake build configuration</td>
<td>489</td>
</tr>
<tr>
<td>Configuration</td>
<td>489</td>
</tr>
<tr>
<td>Control of binary type produced (advanced)</td>
<td>490</td>
</tr>
<tr>
<td>Appendix D: Board configuration</td>
<td>492</td>
</tr>
<tr>
<td>Board Configuration</td>
<td>492</td>
</tr>
<tr>
<td>The Configuration files</td>
<td>492</td>
</tr>
<tr>
<td>Building applications with a custom board configuration</td>
<td>494</td>
</tr>
<tr>
<td>Available configuration parameters</td>
<td>494</td>
</tr>
<tr>
<td>Appendix E: Building the SDK API documentation</td>
<td>495</td>
</tr>
<tr>
<td>Appendix F: SDK release history</td>
<td>496</td>
</tr>
<tr>
<td>Release 1.0.0 (20/Jan/2021)</td>
<td>496</td>
</tr>
<tr>
<td>Release 1.0.1 (01/Feb/2021)</td>
<td>496</td>
</tr>
<tr>
<td>Boot Stage 2</td>
<td>496</td>
</tr>
<tr>
<td>Release 1.1.0 (05/Mar/2021)</td>
<td>496</td>
</tr>
<tr>
<td>Backwards incompatibility</td>
<td>497</td>
</tr>
<tr>
<td>Release 1.1.1 (01/Apr/2021)</td>
<td>497</td>
</tr>
<tr>
<td>Release 1.1.2 (07/Apr/2021)</td>
<td>497</td>
</tr>
<tr>
<td>Release 1.2.0 (03/Jun/2021)</td>
<td>497</td>
</tr>
<tr>
<td>New/improved Board headers</td>
<td>497</td>
</tr>
<tr>
<td>Updated TinyUSB to 0.10.1</td>
<td>497</td>
</tr>
<tr>
<td>Added CMSIS core headers</td>
<td>498</td>
</tr>
<tr>
<td>API improvements</td>
<td>498</td>
</tr>
<tr>
<td>Release</td>
<td>Date</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>1.3.0</td>
<td>02/Nov/2021</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3.1</td>
<td>18/May/2022</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4.0</td>
<td>30/Jun/2022</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5.0</td>
<td>11/Feb/2023</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5.1</td>
<td>14/June/2023</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Appendix G: Documentation release history
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/blob/master/blink/blink.c
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.

```
sleep_ms(250);
#endif
```

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/blob/master/blink/CMakeLists.txt

1 add_executable(blink
2     blink.c
3 )
4
5 # pull in common dependencies
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/blob/master/blink/CMakeLists.txt

```cpp
1 add_executable(blink
2   blink.c
3 )
4
5 # pull in common dependencies
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/blob/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 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.
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).

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 High Level APIs. These may be libraries that have cross-cutting concerns between multiple pieces of hardware (for example the sleep functions in pico_time need to concern themselves both with 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.

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 Runtime Infrastructure.

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 Hardware APIs.

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

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

rather than:

```c
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 don’t want to use the rest of the SDK libraries and runtime.
void pio_sm_set_wrap(PIO pio, uint sm, uint bottom, uint top) {} is actually implemented as a static inline function in [https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_pio/include/hardware/pio.h](https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_pio/include/hardware/pio.h) directly as shown above.

Using static inline functions is common in SDK header files because such methods are often called with parameters that have fixed known values at compile time. In such cases, the compiler is often able to fold the code down to a single register write (or in this case a read, AND with a constant value, OR with a constant value, and a write) with no function call overhead. This tends to produce much smaller and faster binaries.

### 2.3.3.1. Hardware Claiming

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

```c
*(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):

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

The structures and associated pointers to memory mapped register blocks hide the complexity and potential error-prone-ness of dealing with individual memory locations, pointer casts and volatile access. As a bonus, the structs tend to produce better code with older compilers, as they encourage the reuse of a base pointer with offset load/stores, instead of producing a 32 bit literal for every register accessed.

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

```c
typedef struct {
    _REG_ (PLL_CS_OFFSET) // PLL_CS
    // Control and Status
    // 0x80000000 [31] : LOCK (0): PLL is locked
    // 0x00000100 [8]  : BYPASS (0): Passes the reference clock to the output instead of the divided VCO
    // 0x0000003f [5:0] : REFDIV (1): Divides the PLL input reference clock
```

SDK: [https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/pll.h](https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2040/hardware_structs/include/hardware/structs/pll.h) Lines 24 - 53
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:

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

Or, equivalently

```c
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`
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.

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

**IMPORTANT**

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

### 2.3.7. Wi-Fi on Pico W

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

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

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

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

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

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

2.3.8. Bluetooth on Pico W

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

IMPORTANT

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

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

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

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

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

The following additional libraries are optional:

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

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

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

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

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

And in your source code you would include the DMA hardware library header as such:

```c
#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>
<thead>
<tr>
<th>Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>src/rp2040/</code></td>
<td>This contains the <code>hardware_regs</code> and <code>hardware_structs</code> libraries mentioned earlier, which are specific to RP2040.</td>
</tr>
</tbody>
</table>
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_`. Sometimes 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>
<thead>
<tr>
<th>Suffix</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(none)</td>
<td></td>
<td>The method is non-blocking, i.e. it does not wait on any external condition that could potentially take a long time.</td>
</tr>
<tr>
<td>_blocking</td>
<td></td>
<td>The method is blocking, and may potentially block indefinitely until some specific condition is met.</td>
</tr>
<tr>
<td>_blocking_until</td>
<td><code>absolute_time_t</code></td>
<td>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 <code>until</code> time is reached.</td>
</tr>
<tr>
<td>_timeout_ms</td>
<td><code>uint32_t</code></td>
<td>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.</td>
</tr>
<tr>
<td>_timeout_us</td>
<td><code>uint64_t</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

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 \texttt{bool} indicating success or not, or an integer return code from the
\texttt{PICO\_ERROR\_} family; non-error returns are $\geq 0$.

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 \texttt{void}) and must cause some sort of exceptional event.

As of right now the exceptional event is a C \texttt{assert}, so these checks are always disabled in release builds by
default. Additionally most of the calls to \texttt{assert} are disabled by default for code/size performance (even in debug
builds); You can set \texttt{PARAM\_ASSERTIONS\_ENABLE\_ALL=1} or \texttt{PARAM\_ASSERTIONS\_DISABLE\_ALL=1} in your build to change the
default across the entire SDK, or say \texttt{PARAM\_ASSERTIONS\_ENABLED\_I2C=0/1} to explicitly specify the behaviour for the
\texttt{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.

4. Finally some code may choose to "panic" directly if it detects an invalid state. A "panic" involves writing a message
to standard output and then halting (by executing a breakpoint instruction). Panicking is a good response when it
is undesirable to even attempt to continue given the current situation.

### 2.5.3. Use of Inline Functions

SDK libraries often contain a mixture of \texttt{static inline} functions in header files, and non-static functions in C source files.
In particular, the \texttt{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 \texttt{static inline} definitions gives the compiler more freedom to
do this.

One reason this is \textit{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 \texttt{hardware\_} functions as compared with using
preprocessor macros with the \texttt{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:

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

The value of `dma_channel` is known at compile time, so the compiler can replace `dma_channel` with `3` when generating code (constant folding). The `dma_` methods are static inline methods (from https://github.com/raspberrypi/pico-sdk/blob/master/src/rp2_common/hardware_dma/include/hardware/dma.h) meaning the implementations can be folded into your code by the compiler and, consequently, your constant parameters (like `DREQ_SPI0_RX`) are propagated through 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:

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

### 2.6.1. Preprocessor Variables via Board Configuration File

Many of the common configuration settings are actually related to the particular 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/blob/master/src/boards/include/boards/pico.h specifies

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

The header `my_board_name.h` is included by all other SDK headers as a result of setting `PICO_BOARD=my_board_name`. You 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/blob/master/hello_world/CMakeLists.txt with `target_compile_definitions` to specify an alternate set of UART pins to use.

```
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)
```

---

Raspberry Pi Pico C/C++ SDK

2.6. Customisation and Configuration Using Preprocessor variables
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.

```c
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`. 

---

**Raspberry Pi Pico C/C++ SDK**

2.7. SDK Runtime
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

<table>
<thead>
<tr>
<th>Function</th>
<th>ROM/SDK (μs)</th>
<th>GCC 9 (μs)</th>
<th>Performance Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>__aeabi_fadd</td>
<td>72.4</td>
<td>99.8</td>
<td>138%</td>
</tr>
<tr>
<td>__aeabi_fsub</td>
<td>86.7</td>
<td>133.6</td>
<td>154%</td>
</tr>
<tr>
<td>__aeabi_frsup</td>
<td>89.8</td>
<td>140.6</td>
<td>157%</td>
</tr>
<tr>
<td>__aeabi_fmul</td>
<td>61.5</td>
<td>145</td>
<td>236%</td>
</tr>
<tr>
<td>__aeabi_fdiv</td>
<td>74.7</td>
<td>437.5</td>
<td>586%</td>
</tr>
<tr>
<td>__aeabi_fcmpeq</td>
<td>39</td>
<td>61.1</td>
<td>157%</td>
</tr>
<tr>
<td>__aeabi_fcmpne</td>
<td>40.5</td>
<td>61.1</td>
<td>151%</td>
</tr>
<tr>
<td>__aeabi_fcmpge</td>
<td>40.5</td>
<td>61.2</td>
<td>151%</td>
</tr>
<tr>
<td>__aeabi_fcmpgt</td>
<td>41</td>
<td>61.2</td>
<td>149%</td>
</tr>
<tr>
<td>__aeabi_fcmpeq</td>
<td>40</td>
<td>41.5</td>
<td>104%</td>
</tr>
<tr>
<td>__aeabi_dadd</td>
<td>99.4</td>
<td>142.5</td>
<td>143%</td>
</tr>
<tr>
<td>__aeabi_dsub</td>
<td>114.2</td>
<td>182</td>
<td>159%</td>
</tr>
<tr>
<td>__aeabi_drsup</td>
<td>108</td>
<td>181.2</td>
<td>168%</td>
</tr>
<tr>
<td>__aeabi_dmul</td>
<td>168.2</td>
<td>338</td>
<td>201%</td>
</tr>
<tr>
<td>__aeabi_ddiv</td>
<td>197.1</td>
<td>412.2</td>
<td>209%</td>
</tr>
</tbody>
</table>

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.
### 2.7.2.3. Configuration and Alternate Implementations

There are three different floating point implementations provided:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>The default; equivalent to pico</td>
</tr>
<tr>
<td>pico</td>
<td>Use the fast/compact SDK/bootrom implementations</td>
</tr>
<tr>
<td>compiler</td>
<td>Use the standard compiler provided soft floating point implementations</td>
</tr>
<tr>
<td>none</td>
<td>Map all functions to a runtime assertion. You can use this when you know you don’t want any floating point support to make sure it isn’t accidentally pulled in by some library.</td>
</tr>
</tbody>
</table>

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

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

**TIP**

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

### 2.7.2.3.1. NaN Propagation

The SDK implementation by default treats input NaNs 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 `/` 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.
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) is halted, however you can start it executing some function in parallel from your main application thread.

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

Care should be taken with calling C library functions from both cores simultaneously as they are generally not designed to be thread safe. You can use the `mutex_API` provided by the SDK in the `pico_sync` library (https://github.com/raspberrypi/pico-sdk/blob/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.txt` file under either the `add_executable()` entry, or a separate `target_sources()` entry to append them to your target.
To save space, exception handling is disabled by default; this can be overridden with the CMake environment variable `PICO_CXX_ENABLE_EXCEPTIONS=1`. There are a handful of other C++ related `PICO_CXX` vars listed in 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.

 بكال طب

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

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

Above certain speeds — say a factor of 1000 below the processor clock speed — IRQs become impractical, in part due to the timing uncertainty of actually entering an interrupt handler. The alternative when "bit-banging" is to sit the processor in a carefully timed loop, often painstakingly written in assembly, trying to make sure the GPIO reading and writing happens on the exact cycle required. This is really really hard work if indeed possible at all. Many heroic hours and likely thousands of GitHub repositories are dedicated to the task of doing such things (a large proportion of them for LED strings).

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

Whilst dealing with something like an LED string is possible using "bit-banging", once your hardware protocol gets faster to the point that it is of similar order of magnitude to your system clock speed, there is really not much you can hope to do. The main case where software GPIO access is the best choice is LEDs and push buttons.

Therefore you're back to custom hardware for the protocols you know up front you are going to want (or more accurately, the chip designer thinks you might need).

### 3.1.5. Programmable I/O Hardware using FPGAs and CPLDs

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

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

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

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

### 3.1.6. Programmable I/O Hardware using PIO

The PIO subsystem on 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 trade-off is that a PIO state machine is not remotely capable of running general purpose software. As we shall see though, programming a PIO state machine is quite familiar for anyone who has written assembly code before, and the small instruction set should be fairly quick to pick up for those who haven’t.

For simple hardware protocols - such as PWM or duplex SPI - a single PIO state machine can handle the task of implementing the hardware interface all on its own. For more involved protocols such as SDIO or DPI video you may end up using two or three.

**TIP**

If you are ever tempted to “bit-bang” a protocol on 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
3.2.1.1. PIO Program

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


```
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 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.

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:

```
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
25     pio_gpio_init(pio, pin);
```

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.

Our `.pio` file also contains a helper function to set up a PIO state machine for correct execution of this program:
// Set the pin direction to output at the PIO
pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);

// Load our configuration, and jump to the start of the program
pio_sm_init(pio, sm, offset, &c);

// Set the state machine running
pio_sm_set_enabled(pio, sm, true);
}

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

1. The state machine needs to be told which GPIO or GPIOs to output to. There are four different pin groups which are used by different instructions in different situations; here we are using the out pin group, because we are just using an out instruction.

2. The GPIO also needs to be told that PIO is in control of it (GPIO function select)

3. If we are using the pin for output only, we need to make sure that PIO is driving the output enable line high. PIO can drive this line up and down programmatically using e.g. an out pindirs instruction, but here we are setting it up before starting the program.

3.2.1.2. C Program

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

Raspberry Pi Pico C/C++ SDK
3.2. Getting started with PIO

---

```c

#include "pico/stdlib.h"
#include "hardware/pio.h"
#include "hello.pio.h"

int main() {
    #ifndef PICO_DEFAULT_LED_PIN
    #warning pio/hello_pio example requires a board with a regular LED
    #else
    // Choose which PIO instance to use (there are two instances)
    PIO pio = pio0;

    // Our assembled program needs to be loaded into this PIO's instruction
    // memory. This SDK function will find a location (offset) in the
    // instruction memory where there is enough space for our program. We need
    // to remember this location!
    uint offset = pio_add_program(pio, &hello_program);

    // Find a free state machine on our chosen PIO (erroring if there are
    // none). Configure it to run our program, and start it, using the
    // helper function we included in our .pio file.
    uint sm = pio_claim_unused_sm(pio, true);
    hello_program_init(pio, sm, offset, PICO_DEFAULT_LED_PIN);

    // The state machine is now running. Any value we push to its TX FIFO will
```
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.


```cmake
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](image)

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.
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/blob/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 l 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 # 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 $T_3 - 1$ cycles after the instruction (i.e. the whole thing takes $T_3$ 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 $T_1 - 1$ 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.

The pin has been low for time $T_3$, and high for time $T_1$. 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 $T_2 - 1$ 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 $T_3$, high for time $T_1$, then high for an additional time $T_2$
```

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 $T_2$ 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.](image)

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.](image)

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.


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

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
% 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_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);` Sets the PIO pin direction of 1 pin starting at pin number `pin` to out

• `pio_sm_config c = ws2812_program_default_config(offset);` Get the default configuration using the generated function for this program (this includes things like the .wrap and .side_set configurations from the program). We’ll modify this configuration before loading it into the state machine.

• `sm_config_set_sideset_pins(&c, pin);` Sets the side-set to write to pins starting at pin `pin` (we say starting at because if you had .side_set 3, then it would be outputting values on numbers `pin`, `pin+1`, `pin+2`)

• `sm_config_set_out_shift(&c, false, true, rgbw ? 32 : 24);` False for shift_to_right (i.e. we want to shift out MSB first). True for autopull. 32 or 24 for the number of bits for the autopull threshold, i.e. the point at which the state machine triggers a refill of the OSR, depending on whether the LEDs are RGB or RGBW.

• `int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;` This is the total number of execution cycles to output a single bit. Here we see the benefit of .define public; we can use the T1 - T3 values in our code.

• `float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);` Slow the state machine’s execution down, based on the system clock speed and the number of execution cycles required per WS2812 data bit, so that we achieve the correct bit rate.

• `pio_sm_init(pio, sm, offset, &c);` Load our configuration into the state machine, and go to the start address (`offset`)

• `pio_sm_set_enabled(pio, sm, true);` And make it go now!

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

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

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

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

### 3.2.2.3. C Program

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


```
25 static inline void put_pixel(uint32_t pixel_grb) {
26     pio_sm_put_blocking(pio0, 0, pixel_grb << 8u);
27 }
```

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/ws2812/ws2812.c Lines 29 - 34

```
29 static inline uint32_t urgb_u32(uint8_t r, uint8_t g, uint8_t b) {
30     return ((uint32_t) (r) << 8) |
31     ((uint32_t) (g) << 16) |
32     ((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:


```c
50 void pattern_random(uint len, uint t) {
51     if (t % 8)
52         return;
53     for (int i = 0; i < len; ++i)
54         put_pixel(rand());
55 }
```

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.


```c
84 int main() {
85     //set_sys_clock_48();
86     stdio_init_all();
87     printf("WS2812 Smoke Test, using pin %d", WS2812_PIN);
88     // todo get free sm
89     PIO pio = pio0;
90     int sm = 0;
91     uint offset = pio_add_program(pio, &ws2812_program);
92     ws2812_program_init(pio, sm, offset, WS2812_PIN, 800000, IS_RGBW);
93     int t = 0;
94     while (1) {
95         int pat = rand() % count_of(pattern_table);
96         int dir = (rand() >> 30) & 1 ? 1 : -1;
97         puts(pattern_table[pat].name);
98         puts(dir == 1 ? "(forward)" : "(backward)");
99         for (int i = 0; i < 1000; ++i) {
100             pattern_table[pat].pat(NUM_PIXELS, t);
101             sleep_ms(10);
102             t += dir;
103         }
104     }
105 }
```

### 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/blob/master/pio/logic_analyser/logic_analyser.c

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

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 125MHz.

\texttt{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.

\texttt{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 \texttt{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 \texttt{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/blob/master/pio/logic_analyser/logic_analyser.c Lines 65 - 87

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

3.2. Getting started with PIO
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 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 $5x_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:
### 3.2. Getting started with PIO

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 15MHz on two GPIOs, and attaching our brand spanking new logic analyser to those same two GPIOs.

```c
int main() {
    stdio_init_all();
    printf("PIO logic analyser example\n");

    // We're going to capture into a u32 buffer, for best DMA efficiency. Need
    // to be careful of rounding in case the number of pins being sampled
    // isn't a power of 2.
    uint total_sample_bits = CAPTURE_N_SAMPLES * CAPTURE_PIN_COUNT;
    total_sample_bits += bits_packed_per_word(CAPTURE_PIN_COUNT) - 1;
    uint buf_size_words = total_sample_bits / bits_packed_per_word(CAPTURE_PIN_COUNT);
    uint32_t *capture_buf = malloc(buf_size_words * sizeof(uint32_t));
    hard_assert(capture_buf);

   // Grant high bus priority to the DMA, so it can shove the processors out
    // of the way. This should only be needed if you are pushing things up to
    // >16bits/clk here, i.e. if you need to saturate the bus completely.
    bus_ctrl_hw->priority = BUSCTRL_BUS_PRIORITY_DMA_W_BITS |
                   BUSCTRL_BUS_PRIORITY_DMA_R_BITS;
    PIO pio = pio0;
    uint sm = 0;
    uint dma_chan = 0;
    logic_analyser_init(pio, sm, CAPTURE_PIN_BASE, CAPTURE_PIN_COUNT, 1.f);
    printf("Arming trigger\n");
    logic_analyser_arm(pio, sm, dma_chan, capture_buf, buf_size_words, CAPTURE_PIN_BASE, true);
    printf("Starting PWM example\n");
}
```
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 command-line options, we can obtain it as follows (assuming the SDK is extracted at `$PICO_SDK_PATH`):
$ mkdir pioasm_build
$ cd pioasm_build
$ cmake $PICO_SDK_PATH/tools/pioasm
$ make

And then invoke as:

$ ./pioasm

### 3.3.1. Usage

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

$ pioasm -?

giving:

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

Assemble file of PIO program(s) for use in applications.

<input>             the input filename
<output>            the output filename (or filename prefix if the output
                    format produces multiple outputs).
                    if not specified, a single output will be written to stdout

options:
  -o <output_format>   select output_format (default 'c-sdk'); available options are:
                        c-sdk
                        C header suitable for use with the Raspberry Pi Pico SDK
                        python
                        Python file suitable for use with MicroPython
                        hex
                        Raw hex output (only valid for single program inputs)
  -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:
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:

```c
#define <program_name>_<symbol> value
```
for program symbols or

```c
#define <symbol> value
```
for global symbols.

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.

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.

If this directive is present, `<count>` indicates the number of side-set bits to be used. Additionally `opt` may be specified to indicate that a `side <value>` is optional for instructions (note this requires stealing an extra bit — in addition to the `<count>` bits — from those available for the instruction delay). Finally, `pindirs` may be specified to indicate that the side set values should be applied to the PINDIRs and not the PINs. This directive is only valid within a program before the first instruction.

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.

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.

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.

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>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>An integer value e.g. 3 or -7</td>
</tr>
<tr>
<td>hex</td>
<td>A hexadecimal value e.g. 0xf</td>
</tr>
<tr>
<td>binary</td>
<td>A binary value e.g. 0b1001</td>
</tr>
<tr>
<td>symbol</td>
<td>A value defined by a .define (see [pioasm_define])</td>
</tr>
<tr>
<td>&lt;label&gt;</td>
<td>The instruction offset of the label within the program. This makes most sense when used with a JMP instruction (see Section 3.4.2)</td>
</tr>
<tr>
<td>(&lt;expression&gt; )</td>
<td>An expression to be evaluated; see expressions. Note that the parentheses are necessary.</td>
</tr>
</tbody>
</table>
3.3.4. Expressions

Expressions may be freely used within pioasm values.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;expression&gt; + &lt;expression&gt;</code></td>
<td>The sum of two expressions</td>
</tr>
<tr>
<td><code>&lt;expression&gt; - &lt;expression&gt;</code></td>
<td>The difference of two expressions</td>
</tr>
<tr>
<td><code>&lt;expression&gt; * &lt;expression&gt;</code></td>
<td>The multiplication of two expressions</td>
</tr>
<tr>
<td><code>&lt;expression&gt; / &lt;expression&gt;</code></td>
<td>The integer division of two expressions</td>
</tr>
<tr>
<td><code>- &lt;expression&gt;</code></td>
<td>The negation of another expression</td>
</tr>
<tr>
<td><code>. &lt;expression&gt;</code></td>
<td>The bit reverse of another expression</td>
</tr>
<tr>
<td><code>&lt;value&gt;</code></td>
<td>Any value (see Section 3.3.3)</td>
</tr>
</tbody>
</table>

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) to apply to the side_set pins at the start of the instruction. Note that the rules for a side-set value via `side <side_set_value>` are dependent on the `.side_set` (see `pioasm_side_set`) directive for the program. If no `.side_set` is specified then the side `<side_set_value>` is invalid, if an optional number of sideset pins is specified then `side <side_set_value>` may be present, and if a non-optional number of sideset pins is specified, then `side <side_set_value>` is required. The `<side_set_value>` must fit within the number of side-set bits specified in the `.side_set` directive.
<delay_value> Specifies the number of cycles to delay after the instruction completes. The delay_value is specified as a value (see Section 3.3.3), and in general is between 0 and 31 inclusive (a 5-bit value), however the number of bits is reduced when sideset is enabled via the .side_set (see [pioasm_side_set]) directive. If the <delay_value> is not present, then the instruction has no delay.

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

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

3.3.8. Pseudoinstructions

Currently pioasm provides one pseudoinstruction, as a convenience:

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

3.3.9. Output pass through

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

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

```%
c-sdk {

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

The general format is

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

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

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

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

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

```assembly
    .program ws2812
    .side_set 1
10 .define public T1 2
11 .define public T2 5
12 .define public T3 3
13 .lang_opt python sideset_init = pico.PIO.OUT_HIGH
14 .lang_opt python out_init = pico.PIO.OUT_HIGH
15 .lang_opt python out_shiftdir = 1
16 .wrap_target
17 
wr_bitloop:
18     out x, 1       side 0 [T3 - 1]; Side-set still takes place when instruction stalls
19     jmp !x do_zero side 1 [T1 - 1]; Branch on the bit we shifted out. Positive pulse
20     do_one:
21     jmp bitloop side 1 [T2 - 1]; Continue driving high, for a long pulse
22     do_zero:
23     nop side 0 [T2 - 1]; Or drive low, for a short pulse
24 
```

```c
static inline void ws2812_program_init(PIO pio, uint sm, uint offset, uint pin, float freq, bool rgbw) {
    pio_gpio_init(pio, pin);
    pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
    pio_sm_config c = ws2812_program_get_default_config(offset);
    sm_config_set_sideset_pins(&c, pin);
    sm_config_set_out_shift(&c, false, true, rgbw ? 32 : 24);
    sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
    int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;
    float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
    sm_config_set_clkdiv(&c, div);
    pio_sm_init(pio, sm, offset, &c);
    pio_sm_set_enabled(pio, sm, true);
}
```

3.3. Using PIOASM, the PIO Assembler
3.3.10.1. c-sdk

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

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

TIP

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

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

```c
#pragma once

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

// ------ //
// ws2812 //
// ------ //
```
#define ws2812_wrap_target 0
#define ws2812_wrap 3
#define ws2812_T1 2
#define ws2812_T2 5
#define ws2812_T3 3

static const uint16_t ws2812_program_instructions[] = {
    0x6221,  // .wrap_target
    0x1123,  // 0: out    x, 1     side @ [2]
    0x1400,  // 1: jmp    !x, 3    side 1 [1]
    0xa442,  // 2: jmp    0        side 1 [4]
    0x0         // 3: nop                    side 0 [4]
};

#if !PICO_NO_HARDWARE
static const struct pio_program ws2812_program = {
    .instructions = ws2812_program_instructions,
    .length = 4,
    .origin = -1,
};

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

#include "hardware/clocks.h"

static inline void ws2812_program_init(PIO pio, uint sm, uint offset, uint pin, float freq, bool rgbw) {
    pio_gpio_init(pio, pin);
    pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
    pio_sm_config c = ws2812_program_get_default_config(offset);
    sm_config_set_sideset_pins(&c, pin);
    sm_config_set_out_shift(&c, false, true, rgbw ? 32 : 24);
    sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
    int cycles_per_bit = ws2812_T1 + ws2812_T2 + ws2812_T3;
    float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
    sm_config_set_clkdiv(&c, div);
    pio_sm_init(pio, sm, offset, &c);
    pio_sm_set_enabled(pio, sm, true);
}

#endif

// --------------- //
// ws2812_parallel //
// --------------- //
#define ws2812_parallel_wrap_target 0
#define ws2812_parallel_wrap 3
#define ws2812_parallel_T1 2
#define ws2812_parallel_T2 5
#define ws2812_parallel_T3 3

static const uint16_t ws2812_parallel_program_instructions[] = {
    0x6020,  // .wrap_target
    0xa10b,  // 0: out    x, 32
    0xa08b,  // 1: mov    pins, !null [1]
    0xa481,  // 2: mov    pins, x  [4]
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.
import rp2
from machine import Pin

ws2812_T1 = 2
ws2812_T2 = 5
ws2812_T3 = 3

@rp2.asm_pio(
sideset_init=pico.PIO.OUT_HIGH,
out_init=pico.PIO.OUT_HIGH,
out_shiftdir=1)
def ws2812():
  wrap_target()
  label("0")
  out(x, 1) .side(0) [2] # 0
  jmp(not_x, "3") .side(1) [1] # 1
  jmp("0") .side(1) [4] # 2
  label("3")
  nop() .side(0) [4] # 3
  wrap()

@rp2.asm_pio()
def ws2812_parallel():
  wrap_target()
  out(x, 32) # 0
  mov(pins, invert(null)) [1] # 1
  mov(pins, x) [4] # 2
  mov(pins, null) [1] # 3
  wrap()

3.3.10.3. hex

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

Given:
```c
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:


```
1 e081
2 e101
3 e000
4 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>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WAIT</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PULL</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRQ</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bit:</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Delay/side-set</td>
<td>Condition</td>
<td>Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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, prior to decrement
  - 011: !Y: scratch Y zero
  - 100: Y--: scratch Y non-zero, prior to decrement
  - 101: X!=Y: scratch X not equal scratch Y
  - 110: PIN: branch on input pin
  - 111: !OSRE: output shift register not empty

- **Address:** Instruction address to jump to. In the instruction encoding this is an absolute address within the PIO instruction memory.

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

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

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

3.4.2.3. 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

<table>
<thead>
<tr>
<th>Bit:</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. In other words, the pin is selected by adding Index to the PINCTRL_IN_BASE configuration, modulo 32.
  - 10: IRQ: PIO IRQ flag selected by Index
  - 11: 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 calculated by replacing the low two bits of the irq number (irq_num) with the low two bits of the sum (irq_num + sm_num) where sm_num is the state machine number.

3.4.4. IN

3.4.4.1. Encoding

<table>
<thead>
<tr>
<th>Bit:</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Delay/side-set</td>
<td>Source</td>
<td>Bit count</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 PINS, 3 will take the values of pins 5, 6 and 7, and shift these into the ISR. First the ISR is shifted to the left or right to make room for the new input data, then the input data is copied into the gap this leaves. The bit order of the input data is not dependent on the shift direction.

NULL can be used for shifting the ISR’s contents. For example, UARTs receive the LSB first, so must shift to the right. After 8 IN PINS, 1 instructions, the input serial data will occupy bits 31…24 of the ISR. An IN NULL, 24 instruction will shift in 24 zero bits, aligning the input data at ISR bits 7…0. Alternatively, the processor or DMA could perform a byte read from FIFO address + 3, which would take bits 31…24 of the FIFO contents.
3.4.4.3. 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

<table>
<thead>
<tr>
<th>Bit:</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Delay/side-set</td>
<td>Destination</td>
<td>Bit count</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\texttt{out <destination>, <bit\_count>}

where:

- \texttt{<destination>} is one of the destinations specified above.
- \texttt{<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

<table>
<thead>
<tr>
<th>Bit:</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Iff</td>
<td>Blk</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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-zeros.

- **Iffull**: If 1, do nothing unless the total input shift count has reached its threshold, \texttt{SHIFTCTRL\_PUSH\_THRESH} (the same as for autopush).
- **Block**: If 1, stall execution if RX FIFO is full.

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

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

3.4.6.3. Assembler Syntax

\texttt{push (iffull)}

\texttt{push (iffull) block}

\texttt{push (iffull) noblock}

where:

- \texttt{iffull} is equivalent to \texttt{Iffull == 1} above. i.e. the default if this is not specified is \texttt{Iffull == 0}
- \texttt{block} is equivalent to \texttt{Block == 1} above. This is the default if neither \texttt{block} nor \texttt{noblock} are specified
- \texttt{noblock} is equivalent to \texttt{Block == 0} above.

3.4.7. PULL
3.4.7.1. Encoding

<table>
<thead>
<tr>
<th>Bit:</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pull</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>IFE</td>
<td>Blk</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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, $\text{SHIFTCTRL\_PULL\_THRESH}$ (the same as for autopull).
- **Block**: If 1, stall if TX FIFO is empty. If 0, pulling from an empty FIFO copies scratch X to OSR.

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

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

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

**NOTE**

When autopull is enabled, any **Pull** instruction is a no-op when the OSR is full, so that the **Pull** instruction behaves as a barrier. **OUT NULL, 32** can be used to explicitly discard the OSR contents. See the **RP2040 Datasheet** for more detail on autopull.

3.4.7.3. Assembler Syntax

```
pull (ifempty)  
pull (ifempty) block  
pull (ifempty) noblock  
```

where:
- **ifempty**  
  Is equivalent to $\text{IfEmpty} = 1$ above. i.e. the default if this is not specified is $\text{IfEmpty} = 0$
- **block**  
  Is equivalent to $\text{Block} = 1$ above. This is the default if neither **block** nor **noblock** are specified
- **noblock**  
  Is equivalent to $\text{Block} = 0$ above.

3.4.8. MOV

3.4.8.1. Encoding

<table>
<thead>
<tr>
<th>Bit:</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mov</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

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

3.4.8.3. 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

<table>
<thead>
<tr>
<th>Bit:</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Delay/side-set</td>
<td>0</td>
<td>Clr</td>
<td>Wait</td>
<td>Index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 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

```assembly
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 calculated 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

<table>
<thead>
<tr>
<th>Bit:</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delay/side-set</td>
<td>Destination</td>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

**NOTE**

You can also build the API documentation locally, see Appendix E.
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.

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hardware_adc</td>
<td></td>
</tr>
<tr>
<td>hardware_base</td>
<td></td>
</tr>
<tr>
<td>hardware_claim</td>
<td></td>
</tr>
<tr>
<td>hardware_clocks</td>
<td></td>
</tr>
<tr>
<td>hardware_divider</td>
<td></td>
</tr>
<tr>
<td>hardware_dma</td>
<td></td>
</tr>
<tr>
<td>channel_config</td>
<td>DMA channel configuration</td>
</tr>
<tr>
<td>hardware_exception</td>
<td></td>
</tr>
<tr>
<td>hardware_flash</td>
<td></td>
</tr>
<tr>
<td>hardware_gpio</td>
<td></td>
</tr>
<tr>
<td>hardware_i2c</td>
<td></td>
</tr>
<tr>
<td>hardware_interp</td>
<td></td>
</tr>
<tr>
<td>interp_config</td>
<td>Interpolator configuration</td>
</tr>
<tr>
<td>hardware_irq</td>
<td></td>
</tr>
<tr>
<td>hardware_pio</td>
<td></td>
</tr>
<tr>
<td>sm_config</td>
<td>PIO state machine configuration</td>
</tr>
<tr>
<td>pio_instructions</td>
<td>PIO instruction encoding</td>
</tr>
<tr>
<td>hardware_pll</td>
<td></td>
</tr>
<tr>
<td>hardware_pwm</td>
<td></td>
</tr>
<tr>
<td>hardware_resets</td>
<td></td>
</tr>
<tr>
<td>hardware_rtc</td>
<td></td>
</tr>
<tr>
<td>hardware_spi</td>
<td></td>
</tr>
<tr>
<td>hardware_sync</td>
<td></td>
</tr>
<tr>
<td>hardware_timer</td>
<td></td>
</tr>
<tr>
<td>hardware_uart</td>
<td></td>
</tr>
<tr>
<td>hardware_vreg</td>
<td></td>
</tr>
<tr>
<td>hardware_watchdog</td>
<td></td>
</tr>
<tr>
<td>hardware_xosc</td>
<td></td>
</tr>
</tbody>
</table>

4.1.1. hardware_adc

4.1.1.1. Detailed Description

Analog to Digital Converter (ADC) API
The RP2040 has an internal analogue-digital converter (ADC) with the following features:

- SAR ADC
- 500 kS/s (Using an independent 48MHz clock)
- 12 bit (8.7 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_set_round_robin()`), the ADC 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:

\[ T = 27 - \frac{(ADC\_Voltage - 0.706)}{0.001721} \]

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

Example

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

int main() {
    stdio_init_all();
    printf("ADC Example, measuring GPIO26\n");
    adc_init();
    adc_gpio_init(26);
    adc_select_input(0);
    while (1) {
        // 12-bit conversion, assume max value == ADC_VREF == 3.3 V
        const float conversion_factor = 3.3f / (1 << 12);
        uint16_t result = adc_read();
        printf("Raw value: 0x%03x, voltage: %f V\n", result, result * conversion_factor);
        sleep_ms(500);
    }
}
```

### 4.1.1.2. Functions

**void adc_init (void)**

Initialise the ADC HW.

**static void adc_gpio_init (uint gpio)**

Initialise the gpio for use as an ADC pin.
static void adc_select_input (uint input)
    ADC input select.

static uint adc_get_selected_input (void)
    Get the currently selected ADC input channel.

static void adc_set_round_robin (uint input_mask)
    Round Robin sampling selector.

static void adc_set_temp_sensor_enabled (bool enable)
    Enable the onboard temperature sensor.

static uint16_t adc_read (void)
    Perform a single conversion.

static void adc_run (bool run)
    Enable or disable free-running sampling mode.

static void adc_set_clkdiv (float clkdiv)
    Set the ADC Clock divisor.

static void adc_fifo_setup (bool en, bool dreq_en, uint16_t dreq_thresh, bool err_in_fifo, bool byte_shift)
    Setup the ADC FIFO.

static bool adc_fifo_is_empty (void)
    Check FIFO empty state.

static uint8_t adc_fifo_get_level (void)
    Get number of entries in the ADC FIFO.

static uint16_t adc_fifo_get (void)
    Get ADC result from FIFO.

static uint16_t adc_fifo_get_blocking (void)
    Wait for the ADC FIFO to have data.

static void adc_fifo_drain (void)
    Drain the ADC FIFO.

static void adc_irq_set_enabled (bool enabled)
    Enable/Disable ADC interrupts.

---

### 4.1.1.3. Function Documentation

#### 4.1.1.3.1. adc_fifo_drain

static void adc_fifo_drain (void) [inline], [static]
Drain the ADC FIFO.
Will wait for any conversion to complete then drain the FIFO, discarding any results.

#### 4.1.1.3.2. adc_fifo_get

static uint16_t adc_fifo_get (void) [inline], [static]
Get ADC result from FIFO.
Pops the latest result from the ADC FIFO.
4.1.1.3.3. adc_fifo_get_blocking

static uint16_t adc_fifo_get_blocking (void) [inline], [static]
Wait for the ADC FIFO to have data.
Blocks until data is present in the FIFO

4.1.1.3.4. adc_fifo_get_level

static uint8_t adc_fifo_get_level (void) [inline], [static]
Get number of entries in the ADC FIFO.
The ADC FIFO is 4 entries long. This function will return how many samples are currently present.

4.1.1.3.5. adc_fifo_is_empty

static bool adc_fifo_is_empty (void) [inline], [static]
Check FIFO empty state.
Returns
Returns true if the FIFO is empty

4.1.1.3.6. adc_fifo_setup

static void adc_fifo_setup (bool en, bool dreq_en, uint16_t dreq_thresh, bool err_in_fifo, bool byte_shift) [inline], [static]
Setup the ADC FIFO.
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.

4.1.1.3.7. adc_get_selected_input

static uint adc_get_selected_input (void) [inline], [static]
Get the currently selected ADC input channel.
Returns
The currently selected input channel. 0…3 are GPIOs 26…29 respectively. Input 4 is the onboard temperature sensor.

4.1.1.3.8. adc_gpio_init

static void adc_gpio_init (uint gpio) [inline], [static]
Initialise the gpio for use as an ADC pin.
Prepare a GPIO for use with ADC by disabling all digital functions.

**Parameters**

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

### 4.1.1.3.9. adc_init

```c
void adc_init (void)
```

Initialise the ADC HW.

### 4.1.1.3.10. adc_irq_set_enabled

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

Enable/Disable ADC interrupts.

**Parameters**

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

### 4.1.1.3.11. adc_read

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

Perform a single conversion.

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

**Returns**

Result of the conversion.

### 4.1.1.3.12. adc_run

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

Enable or disable free-running sampling mode.

**Parameters**

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

### 4.1.1.3.13. adc_select_input

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

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.

### 4.1.1.3.14. adc_set_clkdiv

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

Set the ADC Clock divisor.
Period of samples will be \((1 + \text{div})\) cycles on average. Note it takes 96 cycles to perform a conversion, so any period less than that will be clamped to 96.

**Parameters**

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

### 4.1.1.3.15. adc_set_round_robin

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

Round Robin sampling selector.

This function sets which inputs are to be run through in round robin mode. 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.

### 4.1.1.3.16. adc_set_temp_sensor_enabled

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

Enable the onboard temperature sensor.

**Parameters**

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

### 4.1.2. hardware_base

#### 4.1.2.1. Detailed Description

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

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>io_</td>
<td></td>
<td></td>
<td>A Memory mapped IO register</td>
</tr>
<tr>
<td>ro_</td>
<td></td>
<td></td>
<td>read-only access</td>
</tr>
<tr>
<td>rw_</td>
<td></td>
<td></td>
<td>read-write access</td>
</tr>
<tr>
<td>wo_</td>
<td></td>
<td></td>
<td>write-only access (can't actually be enforced via C API)</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>8-bit wide access</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>16-bit wide access</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td>32-bit wide access</td>
</tr>
</tbody>
</table>
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.

4.1.2.2. Functions

```c
static __force_inline void hw_set_bits (io_rw_32 *addr, uint32_t mask)  
  Atomically set the specified bits to 1 in a HW register.

static __force_inline void hw_clear_bits (io_rw_32 *addr, uint32_t mask)  
  Atomically clear the specified bits to 0 in a HW register.

static __force_inline void hw_xor_bits (io_rw_32 *addr, uint32_t mask)  
  Atomically flip the specified bits in a HW register.

static __force_inline void hw_write_masked (io_rw_32 *addr, uint32_t values, uint32_t write_mask)  
  Set new values for a sub-set of the bits in a HW register.
```

4.1.2.3. Function Documentation

4.1.2.3.1. hw_clear_bits

```c
static __force_inline void hw_clear_bits (io_rw_32 *addr, uint32_t mask) [static]  
  Atomically clear the specified bits to 0 in a HW register.

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

4.1.2.3.2. hw_set_bits

```c
static __force_inline void hw_set_bits (io_rw_32 *addr, uint32_t mask) [static]  
  Atomically set the specified bits to 1 in a HW register.

Parameters  
  addr Address of writable register  
  mask Bit-mask specifying bits to set
```
4.1.2.3.3. hw_write_masked

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

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

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

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

**Parameters**

<table>
<thead>
<tr>
<th>addr</th>
<th>Address of writable register</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>Bits values</td>
</tr>
<tr>
<td>write_mask</td>
<td>Mask of bits to change</td>
</tr>
</tbody>
</table>

4.1.2.3.4. hw_xor_bits

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

Atomically flip the specified bits in a HW register.

**Parameters**

<table>
<thead>
<tr>
<th>addr</th>
<th>Address of writable register</th>
</tr>
</thead>
<tbody>
<tr>
<td>mask</td>
<td>Bit-mask specifying bits to invert</td>
</tr>
</tbody>
</table>

4.1.3. hardware_claim

4.1.3.1. Detailed Description

Lightweight hardware resource management

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

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

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

Facilities are providing for:

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

4.1.3.2. Functions

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

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

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

Atomically claim one resource out of a range of resources, optionally asserting if none are free.
bool hw_is_claimed (const uint8_t *bits, uint bit_index)
        Determine if a resource is claimed at the time of the call.

void hw_claim_clear (uint8_t *bits, uint bit_index)
        Atomically unclaim a resource.

uint32_t hw_claim_lock (void)
        Acquire the runtime mutual exclusion lock provided by the hardware_claim library.

void hw_claim_unlock (uint32_t token)
        Release the runtime mutual exclusion lock provided by the hardware_claim library.

### 4.1.3.3. Function Documentation

#### 4.1.3.3.1. hw_claim_clear

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

Atomically unclaim a resource.

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

**Parameters**

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

#### 4.1.3.3.2. hw_claim_lock

```c
uint32_t hw_claim_lock (void)
```

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

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

**NOTE**

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

**Returns**

a token to pass to `hw_claim_unlock()`

#### 4.1.3.3.3. hw_claim_or_assert

```c
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)
4.1.3.3.4. hw_claim_unlock

```c
void hwclaim_unlock (uint32_t token)
```

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

**NOTE**

This method MUST be called from the same core that call `hw_claim_lock()`

**Parameters**

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

4.1.3.3.5. hw_claim_unused_from_range

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

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

**Parameters**

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

**Returns**

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

4.1.3.3.6. hw_is_claimed

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

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

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

**Parameters**

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

**Returns**

true if the resource is claimed

4.1.4. hardware_clocks
4.1.4.1. Detailed Description

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

<table>
<thead>
<tr>
<th>Source</th>
<th>Reference Clock</th>
<th>System Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROSC</td>
<td>CLOCKSYSCLKREF_CTRL_SRC_VALUE_ROSCCLKSRC_PH</td>
<td>CLOCKSYSCLKSYS_CTRL_SRC_VALUE_CLKSRC_SYS_AUX</td>
</tr>
<tr>
<td>Auxiliary</td>
<td>CLOCKSYSCLKREF_CTRL_SRC_VALUE_CLKSRC_CLKREF_AUX</td>
<td></td>
</tr>
<tr>
<td>XOSC</td>
<td>CLOCKSYSCLKREF_CTRL_SRC_VALUE_XOSCCLKSRC</td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td></td>
<td>CLOCKSYSCLKSYS_CTRL_SRC_VALUE_CLKREF</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Aux Source</th>
<th>clk_gpout[x]</th>
<th>clk_ref</th>
<th>clk_sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>System PLL</td>
<td>CLOCKSYSCLKGPOUTX_CTRL_L_AUXSRC_VALUE_CLKSRC_PLL_SYS</td>
<td>CLOCKSYSCLKREF_CTRL_AUXSRC_VALUE_CLKSRC_GP_IN0</td>
<td>CLOCKSYSCLKSYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS</td>
</tr>
<tr>
<td>GPIO in 0</td>
<td>CLOCKSYSCLKGPOUTX_CTRL_L_AUXSRC_VALUE_CLKSRC_GPIN0</td>
<td>CLOCKSYSCLKREF_CTRL_AUXSRC_VALUE_CLKSRC_GPIN0</td>
<td>CLOCKSYSCLKSYS_CTRL_AUXSRC_VALUE_CLKSRC_GPIN0</td>
</tr>
<tr>
<td>GPIO in 1</td>
<td>CLOCKSYSCLKGPOUTX_CTRL_L_AUXSRC_VALUE_CLKSRC_GPIN1</td>
<td>CLOCKSYSCLKREF_CTRL_AUXSRC_VALUE_CLKSRC_GPIN1</td>
<td>CLOCKSYSCLKSYS_CTRL_AUXSRC_VALUE_CLKSRC_GPIN1</td>
</tr>
<tr>
<td>USB PLL</td>
<td>CLOCKSYSCLKGPOUTX_CTRL_L_AUXSRC_VALUE_CLKSRC_PLL_USB</td>
<td>CLOCKSYSCLKREF_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB</td>
<td>CLOCKSYSCLKSYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB</td>
</tr>
<tr>
<td>ROSC</td>
<td>CLOCKSYSCLKGPOUTX_CTRL_L_AUXSRC_VALUE_ROSC_CLKSRC</td>
<td></td>
<td>CLOCKSYSCLKSYS_CTRL_AUXSRC_VALUE_ROSC_CLKSRC</td>
</tr>
<tr>
<td>XOSC</td>
<td>CLOCKSYSCLKGPOUTX_CTRL_L_AUXSRC_VALUE_XOSC_CLKSRC</td>
<td></td>
<td>CLOCKSYSCLKSYS_CTRL_AUXSRC_VALUE_ROSC_CLKSRC</td>
</tr>
</tbody>
</table>
### AUX Source

<table>
<thead>
<tr>
<th>AUX Source</th>
<th>clk_gpout[x]</th>
<th>clk_ref</th>
<th>clk_sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>System clock</td>
<td>CLOCKS_CLK_GPOUTX_CTR</td>
<td>CLKS_CLK_GPOUTX_CTR</td>
<td>CLKS_CLK_GPOUTX_CTR</td>
</tr>
<tr>
<td>USB Clock</td>
<td>CLOCKS_CLK_GPOUTX_CTR</td>
<td>CLKS_CLK_GPOUTX_CTR</td>
<td>CLKS_CLK_GPOUTX_CTR</td>
</tr>
<tr>
<td>ADC clock</td>
<td>CLOCKS_CLK_GPOUTX_CTR</td>
<td>CLKS_CLK_GPOUTX_CTR</td>
<td>CLKS_CLK_GPOUTX_CTR</td>
</tr>
<tr>
<td>RTC Clock</td>
<td>CLOCKS_CLK_GPOUTX_CTR</td>
<td>CLKS_CLK_GPOUTX_CTR</td>
<td>CLKS_CLK_GPOUTX_CTR</td>
</tr>
<tr>
<td>Ref clock</td>
<td>CLOCKS_CLK_GPOUTX_CTR</td>
<td>CLKS_CLK_GPOUTX_CTR</td>
<td>CLKS_CLK_GPOUTX_CTR</td>
</tr>
</tbody>
</table>

### AUX Source

<table>
<thead>
<tr>
<th>AUX Source</th>
<th>clk_peri</th>
<th>clk_usb</th>
<th>clk_adc</th>
</tr>
</thead>
<tbody>
<tr>
<td>System PLL</td>
<td>CLOCKS_CLK_PERI_CTRL_A</td>
<td>CLOCKS_CLK_USB_CTRL_A</td>
<td>CLOCKS_CLK_ADC_CTRL_A</td>
</tr>
<tr>
<td>GPIO in 0</td>
<td>CLOCKS_CLK_PERI_CTRL_A</td>
<td>CLOCKS_CLK_USB_CTRL_A</td>
<td>CLOCKS_CLK_ADC_CTRL_A</td>
</tr>
<tr>
<td>GPIO in 1</td>
<td>CLOCKS_CLK_PERI_CTRL_A</td>
<td>CLOCKS_CLK_USB_CTRL_A</td>
<td>CLOCKS_CLK_ADC_CTRL_A</td>
</tr>
<tr>
<td>USB PLL</td>
<td>CLOCKS_CLK_PERI_CTRL_A</td>
<td>CLOCKS_CLK_USB_CTRL_A</td>
<td>CLOCKS_CLK_ADC_CTRL_A</td>
</tr>
<tr>
<td>ROSC</td>
<td>CLOCKS_CLK_PERI_CTRL_A</td>
<td>CLOCKS_CLK_USB_CTRL_A</td>
<td>CLOCKS_CLK_ADC_CTRL_A</td>
</tr>
<tr>
<td>XOSC</td>
<td>CLOCKS_CLK_PERI_CTRL_A</td>
<td>CLOCKS_CLK_USB_CTRL_A</td>
<td>CLOCKS_CLK_ADC_CTRL_A</td>
</tr>
<tr>
<td>System clock</td>
<td>CLOCKS_CLK_PERI_CTRL_A</td>
<td>CLOCKS_CLK_USB_CTRL_A</td>
<td>CLOCKS_CLK_ADC_CTRL_A</td>
</tr>
</tbody>
</table>

### AUX Source

<table>
<thead>
<tr>
<th>AUX Source</th>
<th>clk_rtc</th>
</tr>
</thead>
<tbody>
<tr>
<td>System PLL</td>
<td>CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS</td>
</tr>
<tr>
<td>GPIO in 0</td>
<td>CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_CLKSRC_GP1N0</td>
</tr>
<tr>
<td>GPIO in 1</td>
<td>CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_CLKSRC_GP1N1</td>
</tr>
</tbody>
</table>

---

**4.1. Hardware APIs**

---
Example

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

void measure_freqs(void) {
    uint f_pll_sys = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_PLL_SYS_CLKSRC_PRIMARY);
    uint f_pll_usb = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_PLL_USB_CLKSRC_PRIMARY);
    uint f_rosc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_ROSC_CLKSRC);
    uint f_clk_sys = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_SYS);
    uint f_clk_peri = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_PERI);
    uint f_clk_usb = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_USB);
    uint f_clk_adc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_ADC);
    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_CLKSYS_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,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
4.1.4.2. Typedefs

typedef void (*resus_callback_t)(void)

Resus callback function type.

4.1.4.3. 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.

4.1.4.4. Functions

void clocks_init (void)

Initialise the clock hardware.

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.

void clock_stop (enum clock_index clk_index)

Stop the specified clock.

uint32_t clock_get_hz (enum clock_index clk_index)

Get the current frequency of the specified clock.

uint32_t frequency_count_khz (uint src)

Measure a clocks frequency using the Frequency counter.

void clock_set_reported_hz (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.

void clocks_enable_resus (resus_callback_t resus_callback)

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

void clock_gpio_init_int_frac (uint gpio, uint src, uint32_t div_int, uint8_t div_frac)

Output an optionally divided clock to the specified gpio pin.

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

Output an optionally divided clock to the specified gpio pin.

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

Configure a clock to come from a gpio input.
4.1.4.5. Typedef Documentation

4.1.4.5.1. resus_callback_t

typedef void(* resus_callback_t) (void)

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

4.1.4.6. Enumeration Type Documentation

4.1.4.6.1. clock_index

enum clock_index

Enumeration identifying a hardware clock.

<table>
<thead>
<tr>
<th>Enumerators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk_gpout0</td>
<td>GPIO Muxing 0.</td>
</tr>
<tr>
<td>clk_gpout1</td>
<td>GPIO Muxing 1.</td>
</tr>
<tr>
<td>clk_gpout2</td>
<td>GPIO Muxing 2.</td>
</tr>
<tr>
<td>clk_gpout3</td>
<td>GPIO Muxing 3.</td>
</tr>
<tr>
<td>clk_ref</td>
<td>Watchdog and timers reference clock.</td>
</tr>
<tr>
<td>clk_sys</td>
<td>Processors, bus fabric, memory, memory mapped registers.</td>
</tr>
<tr>
<td>clk_peri</td>
<td>Peripheral clock for UART and SPI.</td>
</tr>
<tr>
<td>clk_usb</td>
<td>USB clock.</td>
</tr>
<tr>
<td>clk_adc</td>
<td>ADC clock.</td>
</tr>
<tr>
<td>clk_rtc</td>
<td>Real time clock.</td>
</tr>
</tbody>
</table>

4.1.4.7. Function Documentation

4.1.4.7.1. 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
4.1.4.7.2. clock_configure_gpin

```c
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

4.1.4.7.3. clock_get_hz

```c
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

4.1.4.7.4. clock_gpio_init

```c
static void clock_gpio_init (uint gpio, uint src, float div) [inline], [static]
```
Output an optionally divided clock to the specified gpio pin.

**Parameters**
- `gpio`: The GPIO pin to output the clock to. Valid GPIOs are: 21, 23, 24, 25. These GPIOs are connected to the GPOUT0-3 clock generators.
- `src`: The source clock. See the register field CLOCKS_CLK_GPOUT0_CTRL_AUXSRC for a full list. The list is the same for each GPOUT clock generator.
- `div`: The float amount to divide the source clock by. This is useful to not overwhelm the GPIO pin with a fast clock.

4.1.4.7.5. clock_gpio_init_int_frac

```c
void clock_gpio_init_int_frac (uint gpio, uint src, uint32_t div_int, uint8_t div_frac)
```
Output an optionally divided clock to the specified gpio pin.

**Parameters**
- `gpio`: The GPIO pin to output the clock to. Valid GPIOs are: 21, 23, 24, 25. These GPIOs are connected to the GPOUT0-3 clock generators.
- `src`: The source clock. See the register field CLOCKS_CLK_GPOUT0_CTRL_AUXSRC for a full list. The list is the same for each GPOUT clock generator.
- `div_int`: The integer part of the value to divide the source clock by. This is useful to not overwhelm the GPIO pin with a fast clock. this is in range of 1..2^24-1.
- `div_frac`: The fractional part of the value to divide the source clock by. This is in range of 0..255 (/256).
4.1.4.7.6. 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()

4.1.4.7.7. clock_stop

void clock_stop (enum clock_index clk_index)

Stop the specified clock.

Parameters

  clk_index  The clock to stop

4.1.4.7.8. clocks_enable_resus

void clocks_enable_resus (resus_callback_t resus_callback)

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

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

Parameters

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

4.1.4.7.9. clocks_init

void clocks_init (void)

Initialise the clock hardware.

Must be called before any other clock function.

4.1.4.7.10. 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

4.1.5.1. Detailed Description

Low-level hardware-divider access

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

This header provides low level macros and inline functions for accessing the hardware dividers directly, and perhaps most usefully performing asynchronous divides. These functions however do not follow the regular 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:

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

int main() {
  stdio_init_all();
  printf("Hello, divider!\n");
  // This is the basic hardware divider function
  int32_t dividend = 123456;
  int32_t divisor = -321;
  divmod_result_t result = hw_divider_divmod_s32(dividend, divisor);
  printf("%d/%d = %d remainder %d\n", dividend, divisor, to_quotient_s32(result), to_remainder_s32(result));
  // Is it right?
  printf("Working backwards! Result %d should equal %d!\n", to_quotient_s32(result) * divisor + to_remainder_s32(result), dividend);
  // This is the recommended unsigned fast divider for general use.
  int32_t udividend = 123456;
  int32_t udivisor = 321;
  divmod_result_t uresult = hw_divider_divmod_u32(udividend, udivisor);
  printf("%d/%d = %d remainder %d\n", udividend, udivisor, to_quotient_u32(result), to_remainder_u32(result));
  // Is it right?
  printf("Working backwards! Result %d should equal %d!\n", to_quotient_u32(result) * divisor + to_remainder_u32(result), dividend);
  // You can also do divides asynchronously. Divides will be complete after 8 cycles.
  hw_divider_divmod_s32_start(dividend, divisor);
  // Do something for 8 cycles!
  // In this example, our results function will wait for completion.
  // Use hw_divider_result_nowait() if you don't want to wait, but are sure you have delayed at least 8 cycles
  result = hw_divider_result_wait();
  printf("Async result %d/%d = %d remainder %d\n", dividend, divisor, to_quotient_s32
```

4.1. Hardware APIs
4.1.5.2. Functions

static void hw_divider_divmod_s32_start (int32_t a, int32_t b)
Start a signed asynchronous divide.

static void hw_divider_divmod_u32_start (uint32_t a, uint32_t b)
Start an unsigned asynchronous divide.

static void hw_divider_wait_ready (void)
Wait for a divide to complete.

static divmod_result_t hw_divider_result_nowait (void)
Return result of HW divide, nowait.

static divmod_result_t hw_divider_result_wait (void)
Return result of last asynchronous HW divide.

static uint32_t hw_divider_u32_quotient_wait (void)
Return result of last asynchronous HW divide, unsigned quotient only.

static int32_t hw_divider_s32_quotient_wait (void)
Return result of last asynchronous HW divide, signed quotient only.

static uint32_t hw_divider_u32_remainder_wait (void)
Return result of last asynchronous HW divide, unsigned remainder only.

static int32_t hw_divider_s32_remainder_wait (void)
Return result of last asynchronous HW divide, signed remainder only.
divmod_result_t hw_divider_divmod_s32 (int32_t a, int32_t b)

Do a signed HW divide and wait for result.

divmod_result_t hw_divider_divmod_u32 (uint32_t a, uint32_t b)

Do an unsigned HW divide and wait for result.

static uint32_t to_quotient_u32 (divmod_result_t r)

Efficient extraction of unsigned quotient from 32p32 fixed point.

static int32_t to_quotient_s32 (divmod_result_t r)

Efficient extraction of signed quotient from 32p32 fixed point.

static uint32_t to_remainder_u32 (divmod_result_t r)

Efficient extraction of unsigned remainder from 32p32 fixed point.

static int32_t to_remainder_s32 (divmod_result_t r)

Efficient extraction of signed remainder from 32p32 fixed point.

static uint32_t hw_divider_u32_quotient (uint32_t a, uint32_t b)

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

static uint32_t hw_divider_u32_remainder (uint32_t a, uint32_t b)

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

static int32_t hw_divider_quotient_s32 (int32_t a, int32_t b)

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

static int32_t hw_divider_remainder_s32 (int32_t a, int32_t b)

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

static void hw_divider_pause (void)

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

static uint32_t hw_divider_u32_quotient_inlined (uint32_t a, uint32_t b)

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

static uint32_t hw_divider_u32_remainder_inlined (uint32_t a, uint32_t b)

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

static int32_t hw_divider_s32_quotient_inlined (int32_t a, int32_t b)

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

static int32_t hw_divider_s32_remainder_inlined (int32_t a, int32_t b)

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

void hw_divider_save_state (hw_divider_state_t *dest)

Save the calling cores hardware divider state.

void hw_divider_restore_state (hw_divider_state_t *src)

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

### 4.1.5.3. Function Documentation

#### 4.1.5.3.1. hw_divider_divmod_s32

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 pair of 32-bit quotient/remainder values.

Parameters
a  The dividend
b  The divisor

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

4.1.5.3.2. hw_divider_divmod_s32_start

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

Start a signed asynchronous divide.

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

Parameters
a  The dividend
b  The divisor

4.1.5.3.3. hw_divider_divmod_u32

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 pair of 32-bit quotient/remainder values.

Parameters
a  The dividend
b  The divisor

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

4.1.5.3.4. hw_divider_divmod_u32_start

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

Start an unsigned asynchronous divide.

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

Parameters
a  The dividend
b  The divisor

4.1.5.3.5. hw_divider_pause

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

Pause for exact amount of time needed for a asynchronous divide to complete.
4.1.5.3.6. hw_divider_quotient_s32

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

Do a signed HW divide, wait for result, return quotient.
Divide \( a \) by \( b \), wait for calculation to complete, return quotient.

Parameters

- \( a \) The dividend
- \( b \) The divisor

Returns

Quotient results of the divide

4.1.5.3.7. hw_divider_remainder_s32

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

Do a signed HW divide, wait for result, return remainder.
Divide \( a \) by \( b \), wait for calculation to complete, return remainder.

Parameters

- \( a \) The dividend
- \( b \) The divisor

Returns

Remainder results of the divide

4.1.5.3.8. hw_divider_restore_state

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

4.1.5.3.9. hw_divider_result_nowait

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

Return result of HW divide, nowait.

NOTE

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

Returns

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

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

Return result of last asynchronous HW divide.

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

Returns

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

4.1.5.3.11. hw_divider_s32_quotient_inlined

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

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

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

Parameters

a The dividend
b The divisor

Returns

Quotient result of the divide

4.1.5.3.12. hw_divider_s32_quotient_wait

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

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

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

Returns

Current signed quotient result.

4.1.5.3.13. hw_divider_s32_remainder_inlined

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

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

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

Parameters

a The dividend
b The divisor

Returns

Remainder result of the divide

4.1.5.3.14. hw_divider_s32_remainder_wait

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

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

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

Returns

Current remainder results.

4.1.5.3.15. hw_divider_save_state

```c
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

4.1.5.3.16. hw_divider_u32_quotient

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

Do an unsigned HW divide, wait for result, return quotient. Divide \(a\) by \(b\), wait for calculation to complete, return quotient.

Parameters

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

Returns

Quotient results of the divide

4.1.5.3.17. hw_divider_u32_quotient_inlined

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

Do a hardware unsigned HW divide, wait for result, return quotient. Divide \(a\) by \(b\), wait for calculation to complete, return quotient.

Parameters

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

Returns

Quotient result of the divide

4.1.5.3.18. hw_divider_u32_quotient_wait

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

Return result of last asynchronous HW divide, unsigned quotient only. This function waits for the result to be ready by calling `hw_divider_wait_ready()`.

Returns
Current unsigned quotient result.

4.1.5.3.19. hw_divider_u32_remainder

static uint32_t hw_divider_u32_remainder (uint32_t a, uint32_t b) [inline], [static]
Do an unsigned HW divide, wait for result, return remainder.
Divide \(a\) by \(b\), wait for calculation to complete, return remainder.

Parameters

- \(a\) The dividend
- \(b\) The divisor

Returns

Remainder results of the divide

4.1.5.3.20. hw_divider_u32_remainder_inlined

static uint32_t hw_divider_u32_remainder_inlined (uint32_t a, uint32_t b) [inline], [static]
Do a hardware unsigned HW divide, wait for result, return remainder.
Divide \(a\) by \(b\), wait for calculation to complete, return remainder.

Parameters

- \(a\) The dividend
- \(b\) The divisor

Returns

Remainder result of the divide

4.1.5.3.21. hw_divider_u32_remainder_wait

static uint32_t hw_divider_u32_remainder_wait (void) [inline], [static]
Return result of last asynchronous HW divide, unsigned remainder only.
This function waits for the result to be ready by calling hw_divider_wait_ready().

Returns

Current unsigned remainder result.

4.1.5.3.22. hw_divider_wait_ready

static void hw_divider_wait_ready (void) [inline], [static]
Wait for a divide to complete.
Wait for a divide to complete

4.1.5.3.23. to_quotient_s32

static int32_t to_quotient_s32 (divmod_result_t r) [inline], [static]
Efficient extraction of signed quotient from 32p32 fixed point.
Parameters

\( r \) A pair of 32-bit quotient/remainder values.

Returns

Unsigned quotient

4.1.5.3.24. to_quotient_u32

static uint32_t to_quotient_u32 (divmod_result_t r) [inline], [static]
Efficient extraction of unsigned quotient from 32p32 fixed point.

Parameters

\( r \) A pair of 32-bit quotient/remainder values.

Returns

Unsigned quotient

4.1.5.3.25. to_remainder_s32

static int32_t to_remainder_s32 (divmod_result_t r) [inline], [static]
Efficient extraction of signed remainder from 32p32 fixed point.

Parameters

\( r \) A pair of 32-bit quotient/remainder values.

Returns

Signed remainder

⚠️ NOTE

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

4.1.5.3.26. to_remainder_u32

static uint32_t to_remainder_u32 (divmod_result_t r) [inline], [static]
Efficient extraction of unsigned remainder from 32p32 fixed point.

Parameters

\( r \) A pair of 32-bit quotient/remainder values.

Returns

Unsigned remainder

⚠️ NOTE

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

4.1.6. hardware_dma
4.1.6.1. Detailed Description

DMA Controller API

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

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

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

4.1.6.2. Modules

channel_config

DMA channel configuration.

4.1.6.3. 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.

4.1.6.4. Functions

```
void dma_channel_claim (uint channel)
  Mark a dma channel as used.

void dma_claim_mask (uint32_t channel_mask)
  Mark multiple dma channels as used.

void dma_channel_unclaim (uint channel)
  Mark a dma channel as no longer used.

void dma_unclaim_mask (uint32_t channel_mask)
  Mark multiple dma channels as no longer used.

int dma_claim_unused_channel (bool required)
  Claim a free dma channel.

bool dma_channel_is_claimed (uint channel)
  Determine if a dma channel is claimed.

static void dma_channel_set_config (uint channel, const dma_channel_config *config, bool trigger)
  Set a channel configuration.

static void dma_channel_set_read_addr (uint channel, const volatile void *read_addr, bool trigger)
  Set the DMA initial read address.

static void dma_channel_set_write_addr (uint channel, volatile void *write_addr, bool trigger)
  Set the DMA initial write address.

static void dma_channel_set_trans_count (uint channel, uint32_t trans_count, bool trigger)
  Set the number of bus transfers the channel will do.
```
static void dma_channel_configure (uint channel, const dma_channel_config *config, volatile void *write_addr, const volatile void *read_addr, uint transfer_count, bool trigger)

Configure all DMA parameters and optionally start transfer.

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

Start a DMA transfer from a buffer immediately.

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

Start a DMA transfer to a buffer immediately.

static void dma_start_channel_mask (uint32_t chan_mask)

Start one or more channels simultaneously.

static void dma_channel_start (uint channel)

Start a single DMA channel.

static void dma_channel_abort (uint channel)

Stop a DMA transfer.

static void dma_channel_set_irq0_enabled (uint channel, bool enabled)

Enable single DMA channel's interrupt via DMA_IRQ_0.

static void dma_set_irq0_channel_mask_enabled (uint32_t channel_mask, bool enabled)

Enable multiple DMA channels' interrupts via DMA_IRQ_0.

static void dma_channel_set_irq1_enabled (uint channel, bool enabled)

Enable single DMA channel's interrupt via DMA_IRQ_1.

static void dma_set_irq1_channel_mask_enabled (uint32_t channel_mask, bool enabled)

Enable multiple DMA channels' interrupts via DMA_IRQ_1.

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

Enable single DMA channel interrupt on either DMA_IRQ_0 or DMA_IRQ_1.

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

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

static bool dma_channel_get_irq0_status (uint channel)

Determine if a particular channel is a cause of DMA_IRQ_0.

static bool dma_channel_get_irq1_status (uint channel)

Determine if a particular channel is a cause of DMA_IRQ_1.

static bool dma_irqn_get_channel_status (uint irq_index, uint channel)

Determine if a particular channel is a cause of DMA_IRQ_N.

static void dma_channel_acknowledge_irq0 (uint channel)

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_0.

static void dma_channel_acknowledge_irq1 (uint channel)

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_1.

static void dma_irqn_acknowledge_channel (uint irq_index, uint channel)

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_N.

static bool dma_channel_is_busy (uint channel)

Check if DMA channel is busy.

static void dma_channel_wait_for_finish_blocking (uint channel)

Wait for a DMA channel transfer to complete.
static void dma_sniffer_enable (uint channel, uint mode, bool force_channel_enable)
    Enable the DMA sniffing targeting the specified channel.
static void dma_sniffer_set_byte_swap_enabled (bool swap)
    Enable the Sniffer byte swap function.
static void dma_sniffer_set_output_invert_enabled (bool invert)
    Enable the Sniffer output invert function.
static void dma_sniffer_set_output_reverse_enabled (bool reverse)
    Enable the Sniffer output bit reversal function.
static void dma_sniffer_disable (void)
    Disable the DMA sniffer.
static void dma_sniffer_set_data_accumulator (uint32_t seed_value)
    Set the sniffer's data accumulator with initial value.
static uint32_t dma_sniffer_get_data_accumulator (void)
    Get the sniffer's data accumulator value.
void dma_timer_claim (uint timer)
    Mark a dma timer as used.
void dma_timer_unclaim (uint timer)
    Mark a dma timer as no longer used.
int dma_claim_unused_timer (bool required)
    Claim a free dma timer.
bool dma_timer_is_claimed (uint timer)
    Determine if a dma timer is claimed.
static void dma_timer_set_fraction (uint timer, uint16_t numerator, uint16_t denominator)
    Set the divider for the given DMA timer.
static uint dma_get_timer_dreq (uint timer_num)
    Return the DREQ number for a given DMA timer.
void dma_channel_cleanup (uint channel)
    Performs DMA channel cleanup after use.

4.1.6.5. Enumeration Type Documentation

4.1.6.5.1. dma_channel_transfer_size

enum dma_channel_transfer_size
    Enumeration of available DMA channel transfer sizes.
    Names indicate the number of bits.
### Table 10. Enumerator

<table>
<thead>
<tr>
<th>DMA_SIZE_8</th>
<th>Byte transfer (8 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_SIZE_16</td>
<td>Half word transfer (16 bits)</td>
</tr>
<tr>
<td>DMA_SIZE_32</td>
<td>Word transfer (32 bits)</td>
</tr>
</tbody>
</table>

### 4.1.6.6. Function Documentation

#### 4.1.6.6.1. dma_channel_abort

```c
static void dma_channel_abort (uint channel) [inline], [static]
```

Stop a DMA transfer.

Function will only return once the DMA has stopped.

Note that due to errata RP2040-E13, aborting a channel which has transfers in-flight (i.e. an individual read has taken place but the corresponding write has not), the ABORT status bit will clear prematurely, and subsequently the in-flight transfers will trigger a completion interrupt once they complete.

The effect of this is that you may see a spurious completion interrupt on the channel as a result of calling this method.

The calling code should be sure to ignore a completion IRQ as a result of this method. This may not require any additional work, as aborting a channel which may be about to complete, when you have a completion IRQ handler registered, is inherently race-prone, and so code is likely needed to disambiguate the two occurrences.

If that is not the case, but you do have a channel completion IRQ handler registered, you can simply disable/re-enable the IRQ around the call to this method as shown by this code fragment (using DMA IRQ0).

```c
// disable the channel on IRQ0
dma_channel_set_irq0_enabled(channel, false);
// abort the channel
dma_channel_abort(channel);
// clear the spurious IRQ (if there was one)
dma_channel_acknowledge_irq0(channel);
// re-enable the channel on IRQ0
dma_channel_set_irq0_enabled(channel, true);
```

**Parameters**

- `channel` DMA channel

#### 4.1.6.6.2. dma_channel_acknowledge_irq0

```c
static void dma_channel_acknowledge_irq0 (uint channel) [inline], [static]
```

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_0.

**Parameters**

- `channel` DMA channel

#### 4.1.6.6.3. dma_channel_acknowledge_irq1

```c
static void dma_channel_acknowledge_irq1 (uint channel) [inline], [static]
```

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_1.

**Parameters**

- `channel` DMA channel
channel DMA channel

4.1.6.6.4. dma_channel_claim

void dma_channel_claim (uint channel)
Mark a dma channel as used.
Method for cooperative claiming of hardware. Will cause a panic if the channel is already claimed. Use of this method
by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters
channel the dma channel

4.1.6.6.5. dma_channel_cleanup

void dma_channel_cleanup (uint channel)
Performs DMA channel cleanup after use.
This can be used to cleanup dma channels when they're no longer needed, such that they are in a clean state for reuse.
IRQ's for the channel are disabled, any in flight-transfer is aborted and any outstanding interrupts are cleared. The
channel is then clear to be reused for other purposes.

```c
if (dma_channel >= 0) {
    dma_channel_cleanup(dma_channel);
    dma_channel_unclaim(dma_channel);
    dma_channel = -1;
}
```

Parameters
channel DMA channel

4.1.6.6.6. dma_channel_configure

static void dma_channel_configure (uint channel, const dma_channel_config * config, volatile void * write_addr, const
volatile void * read_addr, uint transfer_count, bool trigger) [inline], [static]
Configure all DMA parameters and optionally start transfer.

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

4.1.6.6.7. dma_channel_get_irq0_status

static bool dma_channel_get_irq0_status (uint channel) [inline], [static]
Determine if a particular channel is a cause of DMA_IRQ_0.

**Parameters**
- `channel`: DMA channel

**Returns**
- true if the channel is a cause of DMA_IRQ_0, false otherwise

### 4.1.6.6.8. dma_channel_get_irq1_status

```c
static bool dma_channel_get_irq1_status (uint channel) [inline], [static]
```

Determine if a particular channel is a cause of DMA_IRQ_1.

**Parameters**
- `channel`: DMA channel

**Returns**
- true if the channel is a cause of DMA_IRQ_1, false otherwise

### 4.1.6.6.9. dma_channel_is_busy

```c
static bool dma_channel_is_busy (uint channel) [inline], [static]
```

Check if DMA channel is busy.

**Parameters**
- `channel`: DMA channel

**Returns**
- true if the channel is currently busy

### 4.1.6.6.10. dma_channel_is_claimed

```c
bool dma_channel_is_claimed (uint channel)
```

Determine if a dma channel is claimed.

**Parameters**
- `channel`: the dma channel

**Returns**
- true if the channel is claimed, false otherwise

See also
- `dma_channel_claim`
- `dma_channel_claim_mask`

### 4.1.6.6.11. dma_channel_set_config

```c
static void dma_channel_set_config (uint channel, const dma_channel_config * config, bool trigger) [inline], [static]
```

Set a channel configuration.

**Parameters**
4.1.6.6.12. dma_channel_set_irq0_enabled

static void dma_channel_set_irq0_enabled (uint channel, bool enabled) [inline], [static]

Enable single DMA channel's interrupt via DMA_IRQ_0.

Parameters

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

4.1.6.6.13. dma_channel_set_irq1_enabled

static void dma_channel_set_irq1_enabled (uint channel, bool enabled) [inline], [static]

Enable single DMA channel's interrupt via DMA_IRQ_1.

Parameters

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

4.1.6.6.14. dma_channel_set_read_addr

static void dma_channel_set_read_addr (uint channel, const volatile void * read_addr, bool trigger) [inline], [static]

Set the DMA initial read address.

Parameters

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

4.1.6.6.15. dma_channel_set_trans_count

static void dma_channel_set_trans_count (uint channel, uint32_t trans_count, bool trigger) [inline], [static]

Set the number of bus transfers the channel will do.

Parameters

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

4.1.6.6.16. dma_channel_set_write_addr

static void dma_channel_set_write_addr (uint channel, volatile void * write_addr, bool trigger) [inline], [static]

Set the DMA initial write address.
Parameters

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

4.1.6.6.17. dma_channel_start

static void dma_channel_start (uint channel) [inline], [static]
Start a single DMA channel.

Parameters

channel: DMA channel

4.1.6.6.18. dma_channel_transfer_from_buffer_now

static void dma_channel_transfer_from_buffer_now (uint channel, const volatile void * read_addr, uint32_t transfer_count) [inline], [static]
Start a DMA transfer from a buffer immediately.

Parameters

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

4.1.6.6.19. dma_channel_transfer_to_buffer_now

static void dma_channel_transfer_to_buffer_now (uint channel, volatile void * write_addr, uint32_t transfer_count) [inline], [static]
Start a DMA transfer to a buffer immediately.

Parameters

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

4.1.6.6.20. dma_channel_unclaim

void dma_channel_unclaim (uint channel)
Mark a dma channel as no longer used.

Parameters

channel: the dma channel to release
4.1.6.6.21. dma_channel_wait_for_finish_blocking

static void dma_channel_wait_for_finish_blocking (uint channel) [inline], [static]

Wait for a DMA channel transfer to complete.

Parameters

channel DMA channel

4.1.6.6.22. dma_claim_mask

void dma_claim_mask (uint32_t channel_mask)

Mark multiple dma channels as used.

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

Parameters

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

4.1.6.6.23. dma_claim_unused_channel

int dma_claim_unused_channel (bool required)

Claim a free_dma channel.

Parameters

required if true the function will panic if none are available

Returns

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

4.1.6.6.24. dma_claim_unused_timer

int dma_claim_unused_timer (bool required)

Claim a free_dma timer.

Parameters

required if true the function will panic if none are available

Returns

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

4.1.6.6.25. dma_get_timer_dreq

static uint dma_get_timer_dreq (uint timer_num) [inline], [static]

Return the DREQ number for a given DMA timer.

Parameters

timer_num DMA timer number 0-3
4.1.6.26. dma_irqn_acknowledge_channel

```c
static void dma_irqn_acknowledge_channel (uint irq_index, uint channel) [inline], [static]
```

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_N.

**Parameters**

- `irq_index` - the IRQ index; either 0 or 1 for DMA_IRQ_0 or DMA_IRQ_1
- `channel` - DMA channel

4.1.6.27. dma_irqn_get_channel_status

```c
static bool dma_irqn_get_channel_status (uint irq_index, uint channel) [inline], [static]
```

Determine if a particular channel is a cause of DMA_IRQ_N.

**Parameters**

- `irq_index` - the IRQ index; either 0 or 1 for DMA_IRQ_0 or DMA_IRQ_1
- `channel` - DMA channel

**Returns**

true if the channel is a cause of the DMA_IRQ_N, false otherwise

4.1.6.28. dma_irqn_set_channel_enabled

```c
static void dma_irqn_set_channel_enabled (uint irq_index, uint channel, bool enabled) [inline], [static]
```

Enable single DMA channel interrupt on either DMA_IRQ_0 or DMA_IRQ_1.

**Parameters**

- `irq_index` - the IRQ index; either 0 or 1 for DMA_IRQ_0 or DMA_IRQ_1
- `channel` - DMA channel
- `enabled` - true to enable interrupt via irq_index for specified channel, false to disable.

4.1.6.29. dma_irqn_set_channel_mask_enabled

```c
static void dma_irqn_set_channel_mask_enabled (uint irq_index, uint32_t channel_mask, bool enabled) [inline], [static]
```

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

**Parameters**

- `irq_index` - the IRQ index; either 0 or 1 for DMA_IRQ_0 or DMA_IRQ_1
- `channel_mask` - Bitmask of all the channels to enable/disable. Channel 0 = bit 0, channel 1 = bit 1 etc.
- `enabled` - true to enable all the interrupts specified in the mask, false to disable all the interrupts specified in the mask.

4.1.6.30. dma_set_irq0_channel_mask_enabled

```c
static void dma_set_irq0_channel_mask_enabled (uint32_t channel_mask, bool enabled) [inline], [static]
```

Enable multiple DMA channels’ interrupts via DMA_IRQ_0.

**Parameters**
### 4.1.6.6.31. `dma_set_irq1_channel_mask_enabled`

```c
static void dma_set_irq1_channel_mask_enabled (uint32_t channel_mask, bool enabled) [inline], [static]
```

Enable multiple DMA channels' interrupts via DMA_IRQ_1.

**Parameters**

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

### 4.1.6.6.32. `dma_sniffer_disable`

```c
static void dma_sniffer_disable (void) [inline], [static]
```

Disable the DMA sniffer.

### 4.1.6.6.33. `dma_sniffer_enable`

```c
static void dma_sniffer_enable (uint channel, uint mode, bool force_channel_enable) [inline], [static]
```

Enable the DMA sniffing targeting the specified channel.

The mode can be one of the following:

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

**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).
4.1.6.6.34. dma_sniffer_get_data_accumulator

static uint32_t dma_sniffer_get_data_accumulator (void) [inline], [static]

Get the sniffer's data accumulator value.
Read value calculated by the hardware from sniffing the DMA stream

4.1.6.6.35. dma_sniffer_set_byte_swap_enabled

static void dma_sniffer_set_byte_swap_enabled (bool swap) [inline], [static]

Enable the Sniffer byte swap function.
Locally perform a byte reverse on the sniffed data, before feeding into checksum.
Note that the sniff hardware is downstream of the DMA channel byteswap performed in the read master: if
channel_config_set_bswap() and dma_sniffer_set_byte_swap_enabled() are both enabled, their effects cancel from the
sniffer's point of view.

Parameters

    swap    Set true to enable byte swapping

4.1.6.6.36. dma_sniffer_set_data_accumulator

static void dma_sniffer_set_data_accumulator (uint32_t seed_value) [inline], [static]

Set the sniffer's data accumulator with initial value.
Generally, CRC algorithms are used with the data accumulator initially seeded with 0xFFFF or 0xFFFFFFFF (for crc16
and crc32 algorithms)

Parameters

    seed_value    value to set data accumulator

4.1.6.6.37. dma_sniffer_set_output_invert_enabled

static void dma_sniffer_set_output_invert_enabled (bool invert) [inline], [static]

Enable the Sniffer output invert function.
If enabled, the sniff data result appears bit-inverted when read. This does not affect the way the checksum is calculated.

Parameters

    invert    Set true to enable output bit inversion

4.1.6.6.38. dma_sniffer_set_output_reverse_enabled

static void dma_sniffer_set_output_reverse_enabled (bool reverse) [inline], [static]

Enable the Sniffer output bit reversal function.
If enabled, the sniff data result appears bit-reversed when read. This does not affect the way the checksum is calculated.

Parameters

    reverse    Set true to enable output bit reversal
4.1.6.39. **dma_start_channel_mask**

```c
static void dma_start_channel_mask (uint32_t chan_mask) [inline], [static]
```

Start one or more channels simultaneously.

**Parameters**

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

4.1.6.40. **dma_timer_claim**

```c
void dma_timer_claim (uint timer)
```

Mark a dma timer as used.

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

**Parameters**

- `timer` the dma timer

4.1.6.41. **dma_timer_is_claimed**

```c
bool dma_timer_is_claimed (uint timer)
```

Determine if a dma timer is claimed.

**Parameters**

- `timer` the dma timer

**Returns**

true if the timer is claimed, false otherwise

See also

dma_timer_claim

4.1.6.42. **dma_timer_set_fraction**

```c
static void dma_timer_set_fraction (uint timer, uint16_t numerator, uint16_t denominator) [inline], [static]
```

Set the divider for the given DMA timer.

The timer will run at the system_clock_freq * numerator / denominator, so this is the speed that data elements will be transferred at via a DMA channel using this timer as a DREQ.

**Parameters**

- `timer` the dma timer
- `numerator` the fraction's numerator
- `denominator` the fraction's denominator

4.1.6.43. **dma_timer_unclaim**

```c
void dma_timer_unclaim (uint timer)
```

Mark a dma timer as no longer used.

Method for cooperative claiming of hardware.
Parameters

timer the dma timer to release

4.1.6.6.44. dma_unclaim_mask

void dma_unclaim_mask (uint32_t channel_mask)
Mark multiple dma channels as no longer used.

Parameters

c-channel_mask Bitfield of all channels to unclaim (bit 0 == channel 0, bit 1 == channel 1 etc)

4.1.6.7. channel_config

DMA channel configuration.

4.1.6.7.1. Detailed Description

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

4.1.6.7.2. Functions

static void channel_config_set_read_increment (dma_channel_config *c, bool incr)
Set DMA channel read increment in a channel configuration object.

static void channel_config_set_write_increment (dma_channel_config *c, bool incr)
Set DMA channel write increment in a channel configuration object.

static void channel_config_set_dreq (dma_channel_config *c, uint dreq)
Select a transfer request signal in a channel configuration object.

static void channel_config_set_chain_to (dma_channel_config *c, uint chain_to)
Set DMA channel chain_to channel in a channel configuration object.

static void channel_config_set_transfer_data_size (dma_channel_config *c, enum dma_channel_transfer_size size)
Set the size of each DMA bus transfer in a channel configuration object.

static void channel_config_set_ring (dma_channel_config *c, bool write, uint size_bits)
Set address wrapping parameters in a channel configuration object.

static void channel_config_set_bswap (dma_channel_config *c, bool bswap)
Set DMA byte swapping config in a channel configuration object.

static void channel_config_set_irq_quiet (dma_channel_config *c, bool irq_quiet)
Set IRQ quiet mode in a channel configuration object.

static void channel_config_set_high_priority (dma_channel_config *c, bool high_priority)
Set the channel priority in a channel configuration object.

static void channel_config_set_enable (dma_channel_config *c, bool enable)
Enable/Disable the DMA channel in a channel configuration object.

static void channel_config_set_sniff_enable (dma_channel_config *c, bool sniff_enable)
Enable access to channel by sniff hardware in a channel configuration object.
static dma_channel_config dma_channel_get_default_config(uint channel)
    Get the default channel configuration for a given channel.

static dma_channel_config dma_get_channel_config(uint channel)
    Get the current configuration for the specified channel.

static uint32_t channel_config_get_ctrl_value(const dma_channel_config *config)
    Get the raw configuration register from a channel configuration.

4.1.6.7.3. Function Documentation

channel_config_get_ctrl_value

static uint32_t channel_config_get_ctrl_value(const dma_channel_config *config) [inline], [static]
    Get the raw configuration register from a channel configuration.

Parameters

cfg Pointer to a config structure.

Returns

Register content

channel_config_set_bswap

static void channel_config_set_bswap (dma_channel_config * c, bool bswap) [inline], [static]
    Set DMA byte swapping config in a channel configuration object.

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

Parameters

c c Pointer to channel configuration object
bswap bswap True to enable byte swapping

channel_config_set_chain_to

static void channel_config_set_chain_to (dma_channel_config * c, uint chain_to) [inline], [static]
    Set DMA channel chain_to channel in a channel configuration object.

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

Parameters

c c Pointer to channel configuration object
chain_to chain_to Channel to trigger when this channel completes.

channel_config_set_dreq

static void channel_config_set_dreq (dma_channel_config * c, uint dreq) [inline], [static]
    Select a transfer request signal in a channel configuration object.

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

Parameters
Pointer to channel configuration data
dreq Source (see description)

c

channel_config_set_enable

static void channel_config_set_enable (dma_channel_config * c, bool enable) [inline], [static]

Enable/Disable the DMA channel in a channel configuration object.

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

Parameters

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

channel_config_set_high_priority

static void channel_config_set_high_priority (dma_channel_config * c, bool high_priority) [inline], [static]

Set the channel priority in a channel configuration object.

When true, gives a channel preferential treatment in issue scheduling: in each scheduling round, all high priority channels are considered first, and then only a single low priority channel, before returning to the high priority channels.

This only affects the order in which the DMA schedules channels. The DMA’s bus priority is not changed. If the DMA is not saturated then a low priority channel will see no loss of throughput.

Parameters

c Pointer to channel configuration object
high_priority True to enable high priority

channel_config_set_irq_quiet

static void channel_config_set_irq_quiet (dma_channel_config * c, bool irq_quiet) [inline], [static]

Set IRQ quiet mode in a channel configuration object.

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

Parameters

c Pointer to channel configuration object
irq_quiet True to enable quiet mode, false to disable.

channel_config_set_read_increment

static void channel_config_set_read_increment (dma_channel_config * c, bool incr) [inline], [static]

Set DMA channel read increment in a channel configuration object.

Parameters

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

channel_config_set_ring

static void channel_config_set_ring (dma_channel_config * c, bool write, uint size_bits) [inline], [static]

Set address wrapping parameters in a channel configuration object.

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

\(0x0 \rightarrow \text{No wrapping.}\)

**Parameters**

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

channel_config_set_sniff_enable

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

Enable access to channel by sniff hardware in a channel configuration object.

Sniff HW must be enabled and have this channel selected.

**Parameters**

- **c**  
  Pointer to channel configuration object
- **sniff_enable**  
  True to enable the Sniff HW access to this DMA channel.

channel_config_set_transfer_data_size

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

Set the size of each DMA bus transfer in a channel configuration object.

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

**Parameters**

- **c**  
  Pointer to channel configuration object
- **size**  
  See enum for possible values.

channel_config_set_write_increment

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

Set DMA channel write increment in a channel configuration object.

**Parameters**

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

**dma_channel_get_default_config**

```c
static dma_channel_config dma_channel_get_default_config (uint channel) [inline], [static]
```

Get the default channel configuration for a given channel.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Increment</td>
<td>true</td>
</tr>
<tr>
<td>Write Increment</td>
<td>false</td>
</tr>
<tr>
<td>DReq</td>
<td>DREQ_FORCE</td>
</tr>
<tr>
<td>Chain to</td>
<td>self</td>
</tr>
<tr>
<td>Setting</td>
<td>Default</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>Data size</td>
<td>DMA_SIZE_32</td>
</tr>
<tr>
<td>Ring</td>
<td>write=false, size=0 (i.e. off)</td>
</tr>
<tr>
<td>Byte Swap</td>
<td>false</td>
</tr>
<tr>
<td>Quiet IRQs</td>
<td>false</td>
</tr>
<tr>
<td>High Priority</td>
<td>false</td>
</tr>
<tr>
<td>Channel Enable</td>
<td>true</td>
</tr>
<tr>
<td>Sniff Enable</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>DMA channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>the default configuration which can then be modified.</td>
</tr>
</tbody>
</table>

`dma_get_channel_config`

```c
static dma_channel_config dma_get_channel_config (uint channel) [inline], [static]
```

Get the current configuration for the specified channel.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>DMA channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns</td>
<td>The current configuration as read from the HW register (not cached)</td>
</tr>
</tbody>
</table>

### 4.1.7. hardware_exception

#### 4.1.7.1. Detailed Description

Methods for setting processor exception handlers

Exceptions are identified by a `exception_number` which is a number from -15 to -1; these are the numbers relative to the index of the first IRQ vector in the vector table. (i.e. vector table index is exception_num plus 16)

There is one set of exception handlers per core, so the exception handlers for each core as set by these methods are independent.

⚠️ **NOTE**

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

#### 4.1.7.2. Typedefs

```c
typedef void(* exception_handler_t)(void)
```

Exception handler function type.
4.1.7.3. Enumerations

```c
enum exception_number { NMI_EXCEPTION = -14, HARDFAULT_EXCEPTION = -13, SVCALL_EXCEPTION = -5, PENDSV_EXCEPTION = -2, SYSTICK_EXCEPTION = -1 };
```

Exception number definitions.

4.1.7.4. Functions

```c
exception_handler_t exception_set_exclusive_handler (enum exception_number num, exception_handler_t handler)
```

Set the exception handler for an exception on the executing core.

```c
void exception_restore_handler (enum exception_number num, exception_handler_t original_handler)
```

Restore the original exception handler for an exception on this core.

```c
exception_handler_t exception_get_vtable_handler (enum exception_number num)
```

Get the current exception handler for the specified exception from the currently installed vector table of the execution core.

4.1.7.5. Typedef Documentation

4.1.7.5.1. exception_handler_t

```c
typedef void(* exception_handler_t) (void)
```

Exception handler function type.

All exception handlers should be of this type, and follow normal ARM EABI register saving conventions.

4.1.7.6. Enumeration Type Documentation

4.1.7.6.1. exception_number

```c
enum exception_number
```

Exception number definitions.

Note for consistency with irq numbers, these numbers are defined to be negative. The VTABLE index is the number here plus 16.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMI_EXCEPTION</td>
<td>-14</td>
<td>Non Maskable Interrupt</td>
</tr>
<tr>
<td>HARDFAULT_EXCEPTION</td>
<td>-13</td>
<td>HardFault</td>
</tr>
<tr>
<td>SVCALL_EXCEPTION</td>
<td>-5</td>
<td>SV Call</td>
</tr>
<tr>
<td>PENDSV_EXCEPTION</td>
<td>-2</td>
<td>Pend SV</td>
</tr>
<tr>
<td>SYSTICK_EXCEPTION</td>
<td>-1</td>
<td>System Tick</td>
</tr>
</tbody>
</table>

4.1.7.7. Function Documentation
4.1.7.7.1. exception_get_vtable_handler

(exception_handler_t exception_get_vtable_handler (enum exception_number num))

Get the current exception handler for the specified exception from the currently installed vector table of the execution core.

**Parameters**

 num Exception number

**Returns**

the address stored in the VTABLE for the given exception number

4.1.7.7.2. exception_restore_handler

(void exception_restore_handler (enum exception_number num, exception_handler_t original_handler))

Restore the original exception handler for an exception on this core.

This method may be used to restore the exception handler for an exception on this core to the state prior to the call to exception_set_exclusive_handler(), so that exception_set_exclusive_handler() may be called again in the future.

**Parameters**

 num Exception number exception_number

original_handler The original handler returned from exception_set_exclusive_handler

**See also**

exception_set_exclusive_handler()

4.1.7.7.3. exception_set_exclusive_handler

(exception_handler_t exception_set_exclusive_handler (enum exception_number num, exception_handler_t handler))

Set the exception handler for an exception on the executing core.

This method will assert if an exception handler has been set for this exception number on this core via this method, without an intervening restore via exception_restore_handler.

**NOTE**

this method may not be used to override an exception handler that was specified at link time by providing a strong replacement for the weakly defined stub exception handlers. It will assert in this case too.

**Parameters**

 num Exception number

handler The handler to set

**See also**

exception_number

4.1.8. hardware_flash
4.1.8.1. Detailed Description

Low level flash programming and erase API

Note these functions are unsafe if you are using both cores, and the other is executing from flash concurrently with the operation. In this could be the case, you must perform your own synchronisation to make sure that no XIP accesses take place during flash programming. One option is to use the lockout functions.

Likewise they are unsafe if you have interrupt handlers or an interrupt vector table in flash, so you must disable interrupts before calling in this case.

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

Example

```c
#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");
    flash_range_erase(FLASH_TARGET_OFFSET, FLASH_SECTOR_SIZE);
    printf("Done. Read back target region:\n");
    print_buf(flash_target_contents, FLASH_PAGE_SIZE);

    printf("\nProgramming target region...\n");
    flash_range_program(FLASH_TARGET_OFFSET, random_data, FLASH_PAGE_SIZE);
    printf("Done. Read back target region:\n");
    print_buf(flash_target_contents, FLASH_PAGE_SIZE);
    bool mismatch = false;
    for (int i = 0; i < FLASH_PAGE_SIZE; ++i) {
        if (random_data[i] != flash_target_contents[i])
            mismatch = true;
    }
    return mismatch;
}
```
```c
if (mismatch)
    printf("Programming failed!\n");
else
    printf("Programming successful!\n");
```

### 4.1.8.2. Functions

- `void flash_range_erase(uint32_t flash_offs, size_t count)`
  Erase areas of flash.

- `void flash_range_program(uint32_t flash_offs, const uint8_t *data, size_t count)`
  Program flash.

- `void flash_get_unique_id(uint8_t *id_out)`
  Get flash unique 64 bit identifier.

- `void flash_do_cmd(const uint8_t *txbuf, uint8_t *rxbuf, size_t count)`
  Execute bidirectional flash command.

### 4.1.8.3. Function Documentation

#### 4.1.8.3.1. flash_do_cmd

- `void flash_do_cmd(const uint8_t *txbuf, uint8_t *rxbuf, size_t count)`
  Execute bidirectional flash command.

  Low-level function to execute a serial command on a flash device attached to the QSPI interface. Bytes are simultaneously transmitted and received from `txbuf` and to `rxbuf`. Therefore, both buffers must be the same length, `count`, which is the length of the overall transaction. This is useful for reading metadata from the flash chip, such as device ID or SFDP parameters.

  The XIP cache is flushed following each command, in case flash state has been modified. Like other `hardware_flash` functions, the flash is not accessible for execute-in-place transfers whilst the command is in progress, so entering a flash-resident interrupt handler or executing flash code on the second core concurrently will be fatal. To avoid these pitfalls it is recommended that this function only be used to extract flash metadata during startup, before the main application begins to run; see the implementation of `pico_get_unique_id()` for an example of this.

  **Parameters**

  - `txbuf` Pointer to a byte buffer which will be transmitted to the flash
  - `rxbuf` Pointer to a byte buffer where data received from the flash will be written. `txbuf` and `rxbuf` may be the same buffer.
  - `count` Length in bytes of `txbuf` and of `rxbuf`

#### 4.1.8.3.2. flash_get_unique_id

- `void flash_get_unique_id(uint8_t *id_out)`
  Get flash unique 64 bit identifier.

  Use a standard 4Bh RUID instruction to retrieve the 64 bit unique identifier from a flash device attached to the QSPI interface. Since there is a 1:1 association between the MCU and this flash, this also serves as a unique identifier for the
Parameters

id_out  Pointer to an 8-byte buffer to which the ID will be written

4.1.8.3.3. flash_range_erase

void flash_range_erase (uint32_t flash_offs, size_t count)

Erase areas of flash.

Parameters

flash_offs  Offset into flash, in bytes, to start the erase. Must be aligned to a 4096-byte flash sector.

count  Number of bytes to be erased. Must be a multiple of 4096 bytes (one sector).

4.1.8.3.4. flash_range_program

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

Program flash.

Parameters

flash_offs  Flash address of the first byte to be programmed. Must be aligned to a 256-byte flash page.

data  Pointer to the data to program into flash

count  Number of bytes to program. Must be a multiple of 256 bytes (one page).

4.1.9. hardware_gpio

4.1.9.1. Detailed Description

General Purpose Input/Output (GPIO) API

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

The function allocated to each GPIO is selected by calling the gpio_set_function function.

NOTE

Not all functions are available on all pins.

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

Table 11. Function Select Table

<table>
<thead>
<tr>
<th>GPIO</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
<th>F9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SPI0 RX</td>
<td>UART0 TX</td>
<td>I2C0 SDA</td>
<td>PWM0 A</td>
<td>SIO</td>
<td>PIO0</td>
<td>PIO1</td>
<td>USB OVCUR DET</td>
<td></td>
</tr>
<tr>
<td>GPIO</td>
<td>F1</td>
<td>F2</td>
<td>F3</td>
<td>F4</td>
<td>F5</td>
<td>F6</td>
<td>F7</td>
<td>F8</td>
<td>F9</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>1</td>
<td>SPI0 CSn</td>
<td>UART0 RX</td>
<td>I2C0 SCL</td>
<td>PWM0 B</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB VBUS DET</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SPI0 SCK</td>
<td>UART0 CTS</td>
<td>I2C1 SDA</td>
<td>PWM1 A</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB VBUS EN</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SPI0 TX</td>
<td>UART0 RTS</td>
<td>I2C1 SCL</td>
<td>PWM1 B</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB OVCUR DET</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SPI0 RX</td>
<td>UART1 TX</td>
<td>I2C0 SDA</td>
<td>PWM2 A</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB VBUS DET</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SPI1 CSn</td>
<td>UART1 RX</td>
<td>I2C0 SCL</td>
<td>PWM2 B</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB VBUS EN</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>SPI1 SCK</td>
<td>UART1 CTS</td>
<td>I2C1 SDA</td>
<td>PWM3 A</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB OVCUR DET</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>SPI1 TX</td>
<td>UART1 RTS</td>
<td>I2C1 SCL</td>
<td>PWM3 B</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB VBUS DET</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>SPI1 RX</td>
<td>UART1 TX</td>
<td>I2C0 SDA</td>
<td>PWM4 A</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB VBUS EN</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>SPI1 CSn</td>
<td>UART1 RX</td>
<td>I2C0 SCL</td>
<td>PWM4 B</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB OVCUR DET</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>SPI1 SCK</td>
<td>UART1 CTS</td>
<td>I2C1 SDA</td>
<td>PWM5 A</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB VBUS DET</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>SPI1 TX</td>
<td>UART1 RTS</td>
<td>I2C1 SCL</td>
<td>PWM5 B</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB VBUS EN</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>SPI1 RX</td>
<td>UART0 TX</td>
<td>I2C0 SDA</td>
<td>PWM6 A</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB OVCUR DET</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>SPI1 CSn</td>
<td>UART0 RX</td>
<td>I2C0 SCL</td>
<td>PWM6 B</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB VBUS DET</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>SPI1 SCK</td>
<td>UART0 CTS</td>
<td>I2C1 SDA</td>
<td>PWM7 A</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB VBUS EN</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>SPI1 TX</td>
<td>UART0 RTS</td>
<td>I2C1 SCL</td>
<td>PWM7 B</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB OVCUR DET</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>SPI0 RX</td>
<td>UART0 TX</td>
<td>I2C0 SDA</td>
<td>PWM0 A</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB VBUS DET</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>SPI0 CSn</td>
<td>UART0 RX</td>
<td>I2C0 SCL</td>
<td>PWM0 B</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB VBUS EN</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>SPI0 SCK</td>
<td>UART0 CTS</td>
<td>I2C1 SDA</td>
<td>PWM1 A</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB OVCUR DET</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>SPI0 TX</td>
<td>UART0 RTS</td>
<td>I2C1 SCL</td>
<td>PWM1 B</td>
<td>SIO</td>
<td>PI00</td>
<td>PI01</td>
<td>USB VBUS DET</td>
<td></td>
</tr>
</tbody>
</table>
### 4.1.9.2. Typedefs

typedef void(* gpio_irq_callback_t)(uint gpio, uint32_t event_mask)

### 4.1.9.3. 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 = 0x1f }

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 (GPIO events)

enum gpio_slew_rate { GPIO_SLEW_RATE_SLOW = 0, GPIO_SLEW_RATE_FAST = 1 }

Slew rate limiting levels for GPIO outputs.

enum gpio_drive_strength { GPIO_DRIVE_STRENGTH_2MA = 0, GPIO_DRIVE_STRENGTH_4MA = 1, GPIO_DRIVE_STRENGTH_8MA = 2, GPIO_DRIVE_STRENGTH_12MA = 3 }

Drive strength levels for GPIO outputs.
4.1.9.4. Functions

```c
void gpio_set_function (uint gpio, enum gpio_function fn)
    // Select GPIO function.

enum gpio_function gpio_get_function (uint gpio)
    // Determine current GPIO function.

void gpio_set_pulls (uint gpio, bool up, bool down)
    // Select up and down pulls on specific GPIO.

static void gpio_pull_up (uint gpio)
    // Set specified GPIO to be pulled up.

static bool gpio_is_pulled_up (uint gpio)
    // Determine if the specified GPIO is pulled up.

static void gpio_pull_down (uint gpio)
    // Set specified GPIO to be pulled down.

static bool gpio_is_pulled_down (uint gpio)
    // Determine if the specified GPIO is pulled down.

static void gpio_disable_pulls (uint gpio)
    // Disable pulls on specified GPIO.

void gpio_set_irqover (uint gpio, uint value)
    // Set GPIO IRQ override.

void gpio_set_outover (uint gpio, uint value)
    // Set GPIO output override.

void gpio_set_inover (uint gpio, uint value)
    // Select GPIO input override.

void gpio_set_oeover (uint gpio, uint value)
    // Select GPIO output enable override.

void gpio_set_input_enabled (uint gpio, bool enabled)
    // Enable GPIO input.

void gpio_set_input_hysteresis_enabled (uint gpio, bool enabled)
    // Enable/disable GPIO input hysteresis (Schmitt trigger)

bool gpio_is_input_hysteresis_enabled (uint gpio)
    // Determine whether input hysteresis is enabled on a specified GPIO.

void gpio_set_slew_rate (uint gpio, enum gpio_slew_rate slew)
    // Set slew rate for a specified GPIO.

enum gpio_slew_rate gpio_get_slew_rate (uint gpio)
    // Determine current slew rate for a specified GPIO.

void gpio_set_drive_strength (uint gpio, enum gpio_drive_strength drive)
    // Set drive strength for a specified GPIO.

enum gpio_drive_strength gpio_get_drive_strength (uint gpio)
    // Determine current drive strength for a specified GPIO.

void gpio_set_irq_enabled (uint gpio, uint32_t event_mask, bool enabled)
    // Enable or disable specific interrupt events for specified GPIO.
```
void gpio_set_irq_callback (gpio_irq_callback_t callback)
    Set the generic callback used for GPIO IRQ events for the current core.

void gpio_set_irq_enabled_with_callback (uint gpio, uint32_t event_mask, bool enabled, gpio_irq_callback_t callback)
    Convenience function which performs multiple GPIO IRQ related initializations.

void gpio_set_dormant_irq_enabled (uint gpio, uint32_t event_mask, bool enabled)
    Enable dormant wake up interrupt for specified GPIO and events.

static uint32_t gpio_get_irq_event_mask (uint gpio)
    Return the current interrupt status (pending events) for the given GPIO.

void gpio_acknowledge_irq (uint gpio, uint32_t event_mask)
    Acknowledge a GPIO interrupt for the specified events on the calling core.

void gpio_add_raw_irq_handler_with_order_priority_masked (uint gpio_mask, irq_handler_t handler, uint8_t order_priority)
    Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

static void gpio_add_raw_irq_handler_with_order_priority (uint gpio, irq_handler_t handler, uint8_t order_priority)
    Adds a raw GPIO IRQ handler for a specific GPIO on the current core.

void gpio_add_raw_irq_handler_masked (uint gpio_mask, irq_handler_t handler)
    Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

static void gpio_add_raw_irq_handler (uint gpio, irq_handler_t handler)
    Adds a raw GPIO IRQ handler for a specific GPIO on the current core.

void gpio_remove_raw_irq_handler_masked (uint gpio_mask, irq_handler_t handler)
    Removes a raw GPIO IRQ handler for the specified GPIOs on the current core.

static void gpio_remove_raw_irq_handler (uint gpio, irq_handler_t handler)
    Removes a raw GPIO IRQ handler for the specified GPIO on the current core.

void gpio_init (uint gpio)
    Initialise a GPIO for (enabled I/O and set func to GPIO_FUNC_SIO)

void gpio_deinit (uint gpio)
    Resets a GPIO back to the NULL function, i.e. disables it.

void gpio_init_mask (uint gpio_mask)
    Initialise multiple GPIOs (enabled I/O and set func to GPIO_FUNC_SIO)

static bool gpio_get (uint gpio)
    Get state of a single specified GPIO.

static uint32_t gpio_get_all (void)
    Get raw value of all GPIOs.

static void gpio_set_mask (uint32_t mask)
    Drive high every GPIO appearing in mask.

static void gpio_clr_mask (uint32_t mask)
    Drive low every GPIO appearing in mask.

static void gpio_xor_mask (uint32_t mask)
    Toggle every GPIO appearing in mask.

static void gpio_put_masked (uint32_t mask, uint32_t value)
    Drive GPIO high/low depending on parameters.
static void gpio_put_all (uint32_t value)
  Drive all pins simultaneously.

static void gpio_put (uint gpio, bool value)
  Drive a single GPIO high/low.

static bool gpio_get_out_level (uint gpio)
  Determine whether a GPIO is currently driven high or low.

static void gpio_set_dir_out_masked (uint32_t mask)
  Set a number of GPIOs to output.

static void gpio_set_dir_in_masked (uint32_t mask)
  Set a number of GPIOs to input.

static void gpio_set_dir_masked (uint32_t mask, uint32_t value)
  Set multiple GPIO directions.

static void gpio_set_dir_all_bits (uint32_t values)
  Set direction of all pins simultaneously.

static void gpio_set_dir (uint gpio, bool out)
  Set a single GPIO direction.

static bool gpio_is_dir_out (uint gpio)
  Check if a specific GPIO direction is OUT.

static uint gpio_get_dir (uint gpio)
  Get a specific GPIO direction.

4.1.9.5. Typedef Documentation

4.1.9.5.1. gpio_irq_callback_t

typedef void(* gpio_irq_callback_t) (uint gpio, uint32_t event_mask)
  Callback function type for GPIO events

Parameters
  gpio   Which GPIO caused this interrupt
  event_mask Which events caused this interrupt. See gpio_irq_level for details.

See also
  gpio_set_irq_enabled_with_callback()
  gpio_set_irq_callback()

4.1.9.6. Enumeration Type Documentation

4.1.9.6.1. gpio_function

enum gpio_function
  GPIO function definitions for use with function select.
  GPIO function selectors
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 selection.

### 4.1.9.6.2. gpio_irq_level

enum gpio_irq_level

GPIO Interrupt level definitions (GPIO events)

GPIO Interrupt levels

An interrupt can be generated for every GPIO pin in 4 scenarios:

- **Level High**: the GPIO pin is a logical 1
- **Level Low**: the GPIO pin is a logical 0
- **Edge High**: the GPIO has transitioned from a logical 0 to a logical 1
- **Edge Low**: the GPIO has transitioned from a logical 1 to a logical 0

The level interrupts are not latched. This means that if the pin is a logical 1 and the level high interrupt is active, it will become inactive as soon as the pin changes to a logical 0. The edge interrupts are stored in the INTR register and can be cleared by writing to the INTR register.

### 4.1.9.6.3. gpio_slew_rate

enum gpio_slew_rate

Slew rate limiting levels for GPIO outputs.

Slew rate limiting increases the minimum rise/fall time when a GPIO output is lightly loaded, which can help to reduce electromagnetic emissions.

**See also**

`gpio_set_slew_rate`

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO_SLEW_RATE_SLOW</td>
<td>Slew rate limiting enabled.</td>
</tr>
<tr>
<td>GPIO_SLEW_RATE_FAST</td>
<td>Slew rate limiting disabled.</td>
</tr>
</tbody>
</table>

### 4.1.9.6.4. gpio_drive_strength

enum gpio_drive_strength

Drive strength levels for GPIO outputs.

**See also**

`gpio_set_drive_strength`

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO_DRIVE_STRENGTH_2MA</td>
<td>2 mA nominal drive strength</td>
</tr>
<tr>
<td>GPIO_DRIVE_STRENGTH_4MA</td>
<td>4 mA nominal drive strength</td>
</tr>
<tr>
<td>GPIO_DRIVE_STRENGTH_8MA</td>
<td>8 mA nominal drive strength</td>
</tr>
<tr>
<td>GPIO_DRIVE_STRENGTH_12MA</td>
<td>12 mA nominal drive strength</td>
</tr>
</tbody>
</table>
4.1.9.7. Function Documentation

4.1.9.7.1. gpio_acknowledge_irq

```c
void gpio_acknowledge_irq (uint gpio, uint32_t event_mask)
```

Acknowledge a GPIO interrupt for the specified events on the calling core.

**NOTE**

This may be called with a mask of any of valid bits specified in `gpio_irq_level`, however it has no effect on level sensitive interrupts which remain pending while the GPIO is at the specified level. When handling level sensitive interrupts, you should generally disable the interrupt (see `gpio_set_irq_enabled`) and then set it up again later once the GPIO level has changed (or to catch the opposite level).

**Parameters**

- `gpio` GPIO number

**NOTE**

For callbacks set with `gpio_set_irq_enabled_with_callback`, or `gpio_set_irq_callback`, this function is called automatically.

**Parameters**

- `event_mask` Bitmask of events to clear. See `gpio_irq_level` for details.

4.1.9.7.2. gpio_add_raw_irq_handler

```c
static void gpio_add_raw_irq_handler (uint gpio, irq_handler_t handler) [inline], [static]
```

Adds a raw GPIO IRQ handler for a specific GPIO on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see `gpio_set_irq_callback`), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method adds such a callback, and disables the "default" callback for the specified GPIO.

**NOTE**

Multiple raw handlers should not be added for the same GPIO, and this method will assert if you attempt to.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```c
void my_irq_handler(void) {
    if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
        gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
        // handle the IRQ
    }
}
```

**Parameters**

- `gpio` the GPIO number that will no longer be passed to the default callback for this core
- `handler` the handler to add to the list of GPIO IRQ handlers for this core
4.1.9.7.3. gpio_add_raw_irq_handler_masked

```c
void gpio_add_raw_irq_handler_masked (uint gpio_mask, irq_handler_t handler)
```

Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see `gpio_set_irq_callback`), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method adds such a callback, and disables the "default" callback for the specified GPIOs.

**NOTE**

Multiple raw handlers should not be added for the same GPIOs, and this method will assert if you attempt to.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```c
void my_irq_handler(void) {
  if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
    gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
    // handle the IRQ
  }
  if (gpio_get_irq_event_mask(my_gpio_num2) & my_gpio_event_mask2) {
    gpio_acknowledge_irq(my_gpio_num2, my_gpio_event_mask2);
    // handle the IRQ
  }
}
```

**Parameters**

- `gpio_mask`  
  a bit mask of the GPIO numbers that will no longer be passed to the default callback for this core

- `handler`  
  the handler to add to the list of GPIO IRQ handlers for this core

4.1.9.7.4. gpio_add_raw_irq_handler_with_order_priority

```c
static void gpio_add_raw_irq_handler_with_order_priority (uint gpio, irq_handler_t handler, uint8_t order_priority) [inline], [static]
```

Adds a raw GPIO IRQ handler for a specific GPIO on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see `gpio_set_irq_callback`), it is possible to add explicit GPIO IRQ handlers which are called independent of the default callback. The order relative to the default callback can be controlled via the `order_priority` parameter (the default callback has the priority `GPIO_IRQ_CALLBACK_ORDER_PRIORITY` which defaults to the lowest priority with the intention of it running last).

This method adds such a callback, and disables the "default" callback for the specified GPIO.

**NOTE**

Multiple raw handlers should not be added for the same GPIO, and this method will assert if you attempt to.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```c
void my_irq_handler(void) {
  if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
    gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
```

4.1. Hardware APIs
4.1.9.7.5. `gpio_add_raw_irq_handler_with_order_priority_masked`

```c
void gpio_add_raw_irq_handler_with_order_priority_masked(uint gpio_mask, irq_handler_t handler, uint8_t order_priority)
```

Adds a raw GPIO IRQ handler for the specified GPIOs on the current core. In addition to the default mechanism of a single GPIO IRQ event callback per core (see `gpio_set_irq_callback`), it is possible to add explicit GPIO IRQ handlers which are called independent of the default callback. The order relative to the default callback can be controlled via the `order_priority` parameter (the default callback has the priority `GPIO_IRQ_CALLBACK_ORDER_PRIORITY` which defaults to the lowest priority with the intention of it running last).

This method adds such an explicit GPIO IRQ handler, and disables the "default" callback for the specified GPIOs.

NOTE

Multiple raw handlers should not be added for the same GPIOs, and this method will assert if you attempt to.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```c
void my_irq_handler(void) {
    if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
        gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
        // handle the IRQ
    }
    if (gpio_get_irq_event_mask(my_gpio_num2) & my_gpio_event_mask2) {
        gpio_acknowledge_irq(my_gpio_num2, my_gpio_event_mask2);
        // handle the IRQ
    }
}
```

4.1.9.7.6. `gpio_clr_mask`

```c
static void gpio_clr_mask (uint32_t mask) [inline], [static]
```

Drive low every GPIO appearing in mask.

Parameters

- `gpio_mask` a bit mask of the GPIO numbers that will no longer be passed to the default callback for this core
- `handler` the handler to add to the list of GPIO IRQ handlers for this core
- `order_priority` the priority order to determine the relative position of the handler in the list of GPIO IRQ handlers for this core.
Parameters

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

4.1.9.7.7. gpio_deinit

void gpio_deinit (uint gpio)

Resets a GPIO back to the NULL function, i.e. disables it.

Parameters

gpio GPIO number

4.1.9.7.8. gpio_disable_pulls

static void gpio_disable_pulls (uint gpio) [inline], [static]

Disable pulls on specified GPIO.

Parameters

gpio GPIO number

4.1.9.7.9. gpio_get

static bool gpio_get (uint gpio) [inline], [static]

Get state of a single specified GPIO.

Parameters

gpio GPIO number

Returns

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

4.1.9.7.10. gpio_get_all

static uint32_t gpio_get_all (void) [inline], [static]

Get raw value of all GPIOs.

Returns

Bitmask of raw GPIO values, as bits 0-29

4.1.9.7.11. gpio_get_dir

static uint gpio_get_dir (uint gpio) [inline], [static]

Get a specific GPIO direction.

Parameters

gpio GPIO number

Returns

1 for out, 0 for in
4.1.9.7.12. gpio_get_drive_strength

```c
enum gpio_drive_strength gpio_get_drive_strength (uint gpio)
```

Determine current slew rate for a specified GPIO.

*See also*

`gpio_set_drive_strength`

**Parameters**

- `gpio`: GPIO number

**Returns**

Current drive strength of that GPIO

4.1.9.7.13. gpio_get_function

```c
enum gpio_function gpio_get_function (uint gpio)
```

Determine current GPIO function.

**Parameters**

- `gpio`: GPIO number

**Returns**

Which GPIO function is currently selected from list `gpio_function`

4.1.9.7.14. gpio_get_irq_event_mask

```c
static uint32_t gpio_get_irq_event_mask (uint gpio) [inline], [static]
```

Return the current interrupt status (pending events) for the given GPIO.

**Parameters**

- `gpio`: GPIO number

**Returns**

Bitmask of events that are currently pending for the GPIO. See `gpio_irq_level` for details.

*See also*

`gpio_acknowledge_irq`

4.1.9.7.15. gpio_get_out_level

```c
static bool gpio_get_out_level (uint gpio) [inline], [static]
```

Determine whether a GPIO is currently driven high or low.

This function returns the high/low output level most recently assigned to a GPIO via `gpio_put()` or similar. This is the value that is presented outward to the IO muxing, not the input level back from the pad (which can be read using `gpio_get()`).

To avoid races, this function must not be used for read-modify-write sequences when driving GPIOs – instead functions like `gpio_put()` should be used to atomically update GPIOs. This accessor is intended for debug use only.

**Parameters**

- `gpio`: GPIO number
Returns
true if the GPIO output level is high, false if low.

4.1.9.7.16. gpio_get_slew_rate

determine current slew rate for a specified GPIO.

See also
gpio_set_slew_rate

Parameters
   gpio  GPIO number

Returns
Current slew rate of that GPIO

4.1.9.7.17. gpio_init

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

4.1.9.7.18. gpio_init_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

4.1.9.7.19. gpio_is_dir_out

check if a specific GPIO direction is OUT.

Parameters
   gpio  GPIO number

Returns
true if the direction for the pin is OUT
4.1.9.7.20. gpio_is_input_hysteresis_enabled

```c
bool gpio_is_input_hysteresis_enabled (uint gpio)
```

Determine whether input hysteresis is enabled on a specified GPIO.

See also

`gpio_set_input_hysteresis_enabled`

**Parameters**

- `gpio` GPIO number

4.1.9.7.21. gpio_is_pulled_down

```c
static bool gpio_is_pulled_down (uint gpio) [inline], [static]
```

Determine if the specified GPIO is pulled down.

**Parameters**

- `gpio` GPIO number

**Returns**

true if the GPIO is pulled down

4.1.9.7.22. gpio_is_pulled_up

```c
static bool gpio_is_pulled_up (uint gpio) [inline], [static]
```

Determine if the specified GPIO is pulled up.

**Parameters**

- `gpio` GPIO number

**Returns**

true if the GPIO is pulled up

4.1.9.7.23. gpio_pull_down

```c
static void gpio_pull_down (uint gpio) [inline], [static]
```

Set specified GPIO to be pulled down.

**Parameters**

- `gpio` GPIO number

4.1.9.7.24. gpio_pull_up

```c
static void gpio_pull_up (uint gpio) [inline], [static]
```

Set specified GPIO to be pulled up.

**Parameters**

- `gpio` GPIO number
4.1.9.7.25. gpio_put

static void gpio_put (uint gpio, bool value) [inline], [static]

Drive a single GPIO high/low.

Parameters

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

4.1.9.7.26. gpio_put_all

static void gpio_put_all (uint32_t value) [inline], [static]

Drive all pins simultaneously.

Parameters

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

4.1.9.7.27. gpio_put_masked

static void gpio_put_masked (uint32_t mask, uint32_t value) [inline], [static]

Drive GPIO high/low depending on parameters.

Parameters

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

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

4.1.9.7.28. gpio_remove_raw_irq_handler

static void gpio_remove_raw_irq_handler (uint gpio, irq_handler_t handler) [inline], [static]

Removes a raw GPIO IRQ handler for the specified GPIO on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method removes such a callback, and enables the "default" callback for the specified GPIO.

Parameters

  gpio the GPIO number that will now be passed to the default callback for this core
  handler the handler to remove from the list of GPIO IRQ handlers for this core

4.1.9.7.29. gpio_remove_raw_irq_handler_masked

void gpio_remove_raw_irq_handler_masked (uint gpio_mask, irq_handler_t handler)

Removes a raw GPIO IRQ handler for the specified GPIOs on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method removes such a callback, and enables the "default" callback for the specified GPIOs.
Parameters

gpio_mask  a bit mask of the GPIO numbers that will now be passed to the default callback for this core
handler  the handler to remove from the list of GPIO IRQ handlers for this core

4.1.9.7.30. gpio_set_dir

static void gpio_set_dir (uint gpio, bool out) [inline], [static]
Set a single GPIO direction.

Parameters

gpio  GPIO number
out  true for out, false for in

4.1.9.7.31. gpio_set_dir_all_bits

static void gpio_set_dir_all_bits (uint32_t values) [inline], [static]
Set direction of all pins simultaneously.

Parameters

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

4.1.9.7.32. gpio_set_dir_in_masked

static void gpio_set_dir_in_masked (uint32_t mask) [inline], [static]
Set a number of GPIOs to input.

Parameters

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

4.1.9.7.33. gpio_set_dir_masked

static void gpio_set_dir_masked (uint32_t mask, uint32_t value) [inline], [static]
Set multiple GPIO directions.

Parameters

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

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

4.1.9.7.34. gpio_set_dir_out_masked

static void gpio_set_dir_out_masked (uint32_t mask) [inline], [static]
Set a number of GPIOs to output.

Switch all GPIOs in 'mask' to output

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

4.1.9.7.35. **gpio_set_dormant_irq_enabled**

```c
void gpio_set_dormant_irq_enabled (uint gpio, uint32_t event_mask, bool enabled)
```

Enable dormant wake up interrupt for specified GPIO and events.
This configures IQRs to restart the XOSC or ROSC when they are disabled in dormant mode

**Parameters**

- **gpio**  GPIO number
- **event_mask**  Which events will cause an interrupt. See **gpio_irq_level** for details.
- **enabled**  Enable/disable flag

4.1.9.7.36. **gpio_set_drive_strength**

```c
void gpio_set_drive_strength (uint gpio, enum gpio_drive_strength drive)
```

Set drive strength for a specified GPIO.

**See also**

**gpio_get_drive_strength**

**Parameters**

- **gpio**  GPIO number
- **drive**  GPIO output drive strength

4.1.9.7.37. **gpio_set_function**

```c
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**

4.1.9.7.38. **gpio_set_inover**

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

Select GPIO input override.

**Parameters**

- **gpio**  GPIO number
- **value**  See **gpio_override**

4.1.9.7.39. **gpio_set_input_enabled**

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

Enable GPIO input.
4.1.9.7.40. gpio_set_input_hysteresis_enabled

void gpio_set_input_hysteresis_enabled (uint gpio, bool enabled)

Enable/disable GPIO input hysteresis (Schmitt trigger)

Enable or disable the Schmitt trigger hysteresis on a given GPIO. This is enabled on all GPIOs by default. Disabling input hysteresis can lead to inconsistent readings when the input signal has very long rise or fall times, but slightly reduces the GPIO’s input delay.

See also

gpio_is_input_hysteresis_enabled

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpio</td>
<td>GPIO number</td>
</tr>
<tr>
<td>enabled</td>
<td>true to enable input on specified GPIO</td>
</tr>
</tbody>
</table>

4.1.9.7.41. gpio_set_irq_callback

void gpio_set_irq_callback (gpio_irq_callback_t callback)

Set the generic callback used for GPIO IRQ events for the current core.

This function sets the callback used for all GPIO IRQs on the current core that are not explicitly hooked via gpio_add_raw_irq_handler or other gpio_add_raw_irq_handler_ functions.

This function is called with the GPIO number and event mask for each of the (not explicitly hooked) GPIOs that have events enabled and that are pending (see gpio_get_irq_event_mask).

**NOTE**

The IO IRQs are independent per-processor. This function affects the processor that calls the function.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>callback</td>
<td>default user function to call on GPIO irq. Note only one of these can be set per processor.</td>
</tr>
</tbody>
</table>

4.1.9.7.42. gpio_set_irq_enabled

void gpio_set_irq_enabled (uint gpio, uint32_t event_mask, bool enabled)

Enable or disable specific interrupt events for specified GPIO.

This function sets which GPIO events cause a GPIO interrupt on the calling core. See gpio_set_irq_callback, gpio_set_irq_enabled_with_callback and gpio_add_raw_irq_handler to set up a GPIO interrupt handler to handle the events.
NOTE

The IO IRQs are independent per-processor. This configures the interrupt events for the processor that calls the function.

Parameters

- gpio
- event_mask
- enabled
- callback

Events is a bitmask of the following gpio_irq_level values:

<table>
<thead>
<tr>
<th>bit</th>
<th>constant</th>
<th>interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>GPIO_IRQ_LEVEL_LOW</td>
<td>Continuously while level is low</td>
</tr>
<tr>
<td>1</td>
<td>GPIO_IRQ_LEVEL_HIGH</td>
<td>Continuously while level is high</td>
</tr>
<tr>
<td>2</td>
<td>GPIO_IRQ_EDGE_FALL</td>
<td>On each transition from high to low</td>
</tr>
<tr>
<td>3</td>
<td>GPIO_IRQ_EDGE_RISE</td>
<td>On each transition from low to high</td>
</tr>
</tbody>
</table>

which are specified in gpio_irq_level

4.1.9.743. gpio_set_irq_enabled_with_callback

```c
void gpio_set_irq_enabled_with_callback (uint gpio, uint32_t event_mask, bool enabled, gpio_irq_callback_t callback);
```

Convenience function which performs multiple GPIO IRQ related initializations.

This method is a slightly eclectic mix of initialization, that:

- Updates whether the specified events for the specified GPIO causes an interrupt on the calling core based on the enable flag.
- Sets the callback handler for the calling core to callback (or clears the handler if the callback is NULL).
- Enables GPIO IRQs on the current core if enabled is true.

This method is commonly used to perform a one time setup, and following that any additional IRQs/events are enabled via gpio_set_irq_enabled. All GPIOs/events added in this way on the same core share the same callback; for multiple independent handlers for different GPIOs you should use gpio_add_raw_irq_handler and related functions.

This method is equivalent to:

```c
1 gpio_set_irq_enabled(gpio, event_mask, enabled);
2 gpio_set_irq_callback(callback);
3 if (enabled) irq_set_enabled(IO_IRQ_BANK0, true);
```

NOTE

The IO IRQs are independent per-processor. This method affects only the processor that calls the function.

Parameters

- gpio
- event_mask
- enabled
- callback

user function to call on GPIO irq, if NULL, the callback is removed
4.1.9.7.44. gpio_set_irqover

```c
void gpio_set_irqover (uint gpio, uint value)
```

Set GPIO IRQ override. Optionally invert a GPIO IRQ signal, or drive it high or low

**Parameters**

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

4.1.9.7.45. gpio_set_mask

```c
static void gpio_set_mask (uint32_t mask) [inline], [static]
```

Drive high every GPIO appearing in mask.

**Parameters**

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

4.1.9.7.46. gpio_set_oeover

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

Select GPIO output enable override.

**Parameters**

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

4.1.9.7.47. gpio_set_outover

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

Set GPIO output override.

**Parameters**

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

4.1.9.7.48. gpio_set_pulls

```c
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
4.1.9.7.49. gpio_set_slew_rate

```c
void gpio_set_slew_rate (uint gpio, enum gpio_slew_rate slew)
```

Set slew rate for a specified GPIO.

See also

gpio_get_slew_rate

Parameters

- **gpio**  
  GPIO number
- **slew**  
  GPIO output slew rate

4.1.9.7.50. gpio_xor_mask

```c
static void gpio_xor_mask (uint32_t mask) [inline], [static]
```

Toggle every GPIO appearing in mask.

Parameters

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

4.1.10. hardware_i2c

4.1.10.1. Detailed Description

I2C Controller API

The I2C bus is a two-wire serial interface, consisting of a serial data line SDA and a serial clock SCL. These wires carry information between the devices connected to the bus. Each device is recognized by a unique 7-bit address and can operate as either a “transmitter” or “receiver”, depending on the function of the device. Devices can also be considered as masters or slaves when performing data transfers. A master is a device that initiates a data transfer on the bus and generates the clock signals to permit that transfer. The first byte in the data transfer always contains the 7-bit address and a read/write bit in the LSB position. This API takes care of toggling the read/write bit. After this, any device addressed is considered a slave.

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

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

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

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

Example

```c
1 // Sweep through all 7-bit I2C addresses, to see if any slaves are present on
2 // 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 "pico/binary_info.h"
#include "hardware/i2c.h"

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

int main() {
    // Enable UART so we can print status output
    stdio_init_all();
    #if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN)
        #warning i2c/bus_scan example requires a board with I2C pins
        puts("Default I2C pins were not defined");
    #else
        // This example will use I2C0 on the default SDA and SCL pins (GP4, GP5 on a Pico)
        i2c_init(i2c_default, 100 * 1000);
        gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
        gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
        gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
        gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
        // Make the I2C pins available to picotool
        bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));

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

        for (int addr = 0; addr < (1 << 7); ++addr) {
            if (addr % 16 == 0) {
                printf("%02x  ", addr);
            }
            int ret;
            uint8_t rxdata;
            if (reserved_addr(addr))
                ret = PICO_ERROR_GENERIC;
            else
                ret = i2c_read_blocking(i2c_default, addr, &rxdata, 1, false);
        }
        // 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.
    #endif
    return 0;
}
4.1.10.2. Functions

uint i2c_init (i2c_inst_t *i2c, uint baudrate)
    Initialise the I2C HW block.

void i2c_deinit (i2c_inst_t *i2c)
    Disable the I2C HW block.

uint i2c_set_baudrate (i2c_inst_t *i2c, uint baudrate)
    Set I2C baudrate.

void i2c_set_slave_mode (i2c_inst_t *i2c, bool slave, uint8_t addr)
    Set I2C port to slave mode.

static uint i2c_hw_index (i2c_inst_t *i2c)
    Convert I2C instance to hardware instance number.

int i2c_write_blocking_until (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop, absolute_time_t until)
    Attempt to write specified number of bytes to address, blocking until the specified absolute time is reached.

int i2c_read_blocking_until (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len, bool nostop, absolute_time_t until)
    Attempt to read specified number of bytes from address, blocking until the specified absolute time is reached.

static int i2c_write_timeout_us (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop, uint timeout_us)
    Attempt to write specified number of bytes to address, with timeout.

static int i2c_read_timeout_us (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len, bool nostop, uint timeout_us)
    Attempt to read specified number of bytes from address, with timeout.

int i2c_write_blocking (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop)
    Attempt to write specified number of bytes to address, blocking.

int i2c_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.

static size_t i2c_get_write_available (i2c_inst_t *i2c)
    Determine non-blocking write space available.

static size_t i2c_get_read_available (i2c_inst_t *i2c)
    Determine number of bytes received.

static void i2c_write_raw_blocking (i2c_inst_t *i2c, const uint8_t *src, size_t len)
    Write direct to TX FIFO.

static void i2c_read_raw_blocking (i2c_inst_t *i2c, uint8_t *dst, size_t len)
    Read direct from RX FIFO.

static uint8_t i2c_read_byte_raw (i2c_inst_t *i2c)
    Pop a byte from I2C Rx FIFO.
static void i2c_write_byte_raw (i2c_inst_t *i2c, uint8_t value)
Push a byte into I2C Tx FIFO.

static uint i2c_get_dreq (i2c_inst_t *i2c, bool is_tx)
Return the DREQ to use for pacing transfers to/from a particular I2C instance.

4.1.10.2.1. i2c0_inst
i2c_inst_t i2c0_inst
The I2C identifiers for use in I2C functions.
e.g. i2c_init(i2c0, 48000)

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.
Parameters
   i2c       Either i2c0 or i2c1
Disable the I2C again if it is no longer used. Must be reinitialised before being used again.

4.1.10.3.2. i2c_get_dreq
static uint i2c_get_dreq (i2c_inst_t * i2c, bool is_tx) [inline], [static]
Return the DREQ to use for pacing transfers to/from a particular I2C instance.
Parameters
   i2c       Either i2c0 or i2c1
   is_tx     true for sending data to the I2C instance, false for receiving data from the I2C instance

4.1.10.3.3. i2c_get_read_available
static size_t i2c_get_read_available (i2c_inst_t * i2c) [inline], [static]
Determine number of bytes received.
Parameters
   i2c       Either i2c0 or i2c1
Returns
0 if no data available, if return is nonzero at least that many bytes can be read without blocking.

4.1.10.3.4. i2c_get_write_available
static size_t i2c_get_write_available (i2c_inst_t * i2c) [inline], [static]
Determine non-blocking write space available.
Parameters

**i2c**

Either `i2c0` or `i2c1`

Returns

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

### 4.1.10.3.5. i2c_hw_index

```c
static uint i2c_hw_index (i2c_inst_t * i2c) [inline], [static]
```

Convert I2C instance to hardware instance number.

Parameters

**i2c**

I2C instance

Returns

Number of I2C, 0 or 1.

### 4.1.10.3.6. i2c_init

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

Initialise the I2C HW block.

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

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

Parameters

**i2c**

Either `i2c0` or `i2c1`

**baudrate**

Baudrate in Hz (e.g. 100kHz is 100000)

Returns

Actual set baudrate

### 4.1.10.3.7. i2c_read_blocking

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

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

Parameters

**i2c**

Either `i2c0` or `i2c1`

**addr**

7-bit address of device to read from

**dst**

Pointer to buffer to receive data

**len**

Length of data in bytes to receive

**nostop**

If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.

Returns

Number of bytes read, or PICO_ERROR_GENERIC if address not acknowledged or no device present.
4.1.10.3.8. i2c_read_blocking_until

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

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

**Parameters**

- `i2c`: Either `i2c0` or `i2c1`
- `addr`: 7-bit address of device to read from
- `dst`: Pointer to buffer to receive data
- `len`: Length of data in bytes to receive
- `nostop`: If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.
- `until`: The absolute time that the block will wait until the entire transaction is complete.

**Returns**

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

4.1.10.3.9. i2c_read_byte_raw

```c
static uint8_t i2c_read_byte_raw (i2c_inst_t * i2c) [inline], [static]
```

Pop a byte from I2C Rx FIFO.

This function is non-blocking and assumes the Rx FIFO isn’t empty.

**Parameters**

- `i2c`: I2C instance.

**Returns**

`uint8_t` Byte value.

4.1.10.3.10. i2c_read_raw_blocking

```c
static void i2c_read_raw_blocking (i2c_inst_t * i2c, uint8_t * dst, size_t len) [inline], [static]
```

Read direct from RX FIFO.

**Parameters**

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

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

4.1.10.3.11. i2c_read_timeout_us

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

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

**Parameters**
Either i2c0 or i2c1
7-bit address of device to read from
Pointer to buffer to receive data
Length of data in bytes to receive
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.
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.

4.1.10.3.12. 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

4.1.10.3.13. 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

4.1.10.3.14. i2c_write_blocking

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

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

Parameters
i2c Either i2c0 or i2c1
addr 7-bit address of device to write to
src Pointer to data to send
len Length of data in bytes to send
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.

### 4.1.10.3.15. i2c_write_blocking_until

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

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

**Parameters**

- **i2c**: Either i2c0 or i2c1
- **addr**: 7-bit address of device to write to
- **src**: Pointer to data to send
- **len**: Length of data in bytes to send
- **nostop**: If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.
- **until**: The absolute time that the block will wait until the entire transaction is complete. Note, an individual timeout of this value divided by the length of data is applied for each byte transfer, so if the first or subsequent bytes fails to transfer within that sub timeout, the function will return with an error.

**Returns**

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

### 4.1.10.3.16. i2c_write_byte_raw

```c
static void i2c_write_byte_raw (i2c_inst_t * i2c, uint8_t value) [inline], [static]
```

Push a byte into I2C Tx FIFO.

This function is non-blocking and assumes the Tx FIFO isn’t full.

**Parameters**

- **i2c**: I2C instance.
- **value**: Byte value.

### 4.1.10.3.17. i2c_write_raw_blocking

```c
static void i2c_write_raw_blocking (i2c_inst_t * i2c, const uint8_t * src, size_t len) [inline], [static]
```

Write direct to TX FIFO.

**Parameters**

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

Writes directly to the I2C TX FIFO which is mainly useful for slave-mode operation.
4.1.10.3.18. i2c_write_timeout_us

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

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

Parameters

- **i2c**: Either i2c0 or i2c1
- **addr**: 7-bit address of device to write to
- **src**: Pointer to data to send
- **len**: Length of data in bytes to send
- **nostop**: If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.
- **timeout_us**: The time that the function will wait for the entire transaction to complete. Note, an individual timeout of this value divided by the length of data is applied for each byte transfer, so if the first or subsequent bytes fails to transfer within that sub timeout, the function will return with an error.

Returns

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

4.1.11. hardware_interp

4.1.11.1. Detailed Description

Hardware Interpolator API

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

The interpolators are used heavily to accelerate audio operations within the 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.

4.1.11.2. Modules

- **interp_config**
  Interpolator configuration.

4.1.11.3. Functions

- **void interp_claim_lane (interp_hw_t *interp, uint lane)**
  Claim the interpolator lane specified.

- **void interp_claim_lane_mask (interp_hw_t *interp, uint lane_mask)**
  Claim the interpolator lanes specified in the mask.
void interp_unclaim_lane (interp_hw_t *interp, uint lane)
    Release a previously claimed interpolator lane.

bool interp_lane_is_claimed (interp_hw_t *interp, uint lane)
    Determine if an interpolator lane is claimed.

void interp_unclaim_lane_mask (interp_hw_t *interp, uint lane_mask)
    Release previously claimed interpolator lanes.

static void interp_set_force_bits (interp_hw_t *interp, uint lane, uint bits)
    Directly set the force bits on a specified lane.

void interp_save (interp_hw_t *interp, interp_hw_save_t *saver)
    Save the specified interpolator state.

void interp_restore (interp_hw_t *interp, interp_hw_save_t *saver)
    Restore an interpolator state.

static void interp_set_base (interp_hw_t *interp, uint lane, uint32_t val)
    Sets the interpolator base register by lane.

static uint32_t interp_get_base (interp_hw_t *interp, uint lane)
    Gets the content of interpolator base register by lane.

static void interp_set_base_both (interp_hw_t *interp, uint32_t val)
    Sets the interpolator base registers simultaneously.

static void interp_set_accumulator (interp_hw_t *interp, uint lane, uint32_t val)
    Sets the interpolator accumulator register by lane.

static uint32_t interp_get_accumulator (interp_hw_t *interp, uint lane)
    Gets the content of the interpolator accumulator register by lane.

static uint32_t interp_pop_lane_result (interp_hw_t *interp, uint lane)
    Read lane result, and write lane results to both accumulators to update the interpolator.

static uint32_t interp_peek_lane_result (interp_hw_t *interp, uint lane)
    Read lane result.

static uint32_t interp_pop_full_result (interp_hw_t *interp)
    Read lane result, and write lane results to both accumulators to update the interpolator.

static uint32_t interp_peek_full_result (interp_hw_t *interp)
    Read lane result.

static void interp_add_accumulator (interp_hw_t *interp, uint lane, uint32_t val)
    Add to accumulator.

static uint32_t interp_get_raw (interp_hw_t *interp, uint lane)
    Get raw lane value.

4.1.11.4. Function Documentation

4.1.11.4.1. interp_add_accumulator

static void interp_add_accumulator (interp_hw_t * interp, uint lane, uint32_t val) [inline], [static]
Add to accumulator.
Atomically add the specified value to the accumulator on the specified lane

Parameters

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

4.1.11.4.2. interp_claim_lane

```c
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.

4.1.11.4.3. interp_claim_lane_mask

```c
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)

4.1.11.4.4. interp_get_accumulator

```c
static uint32_t interp_get_accumulator (interp_hw_t * interp, uint lane) [inline], [static]
```

Gets the content of the interpolator accumulator register by lane.

Parameters

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

Returns

The current content of the register

4.1.11.4.5. interp_get_base

```c
static uint32_t interp_get_base (interp_hw_t * interp, uint lane) [inline], [static]
```

Gets the content of interpolator base register by lane.

Parameters

- interp: Interpolator instance, interp0 or interp1.
- lane: The lane number, 0 or 1 or 2
Returns
The current content of the lane base register

4.1.11.4.6. interp_get_raw

static uint32_t interp_get_raw (interp_hw_t * interp, uint lane) [inline], [static]
Get raw lane value.
Returns the raw shift and mask value from the specified lane, BASE0 is NOT added

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

Returns
The raw shift/mask value

4.1.11.4.7. interp_lane_is_claimed

bool interp_lane_is_claimed (interp_hw_t * interp, uint lane)
Determine if an interpolator lane is claimed.

Parameters
interp Interpolator whose lane to check
lane The lane number, 0 or 1

Returns
ture if claimed, false otherwise

See also
interp_claim_lane
interp_claim_lane_mask

4.1.11.4.8. interp_peek_full_result

static uint32_t interp_peek_full_result (interp_hw_t * interp) [inline], [static]
Read lane result.

Parameters
interp Interpolator instance, interp0 or interp1.

Returns
The content of the FULL register

4.1.11.4.9. interp_peek_lane_result

static uint32_t interp_peek_lane_result (interp_hw_t * interp, uint lane) [inline], [static]
Read lane result.

Parameters
4.1.11.4.10. interp_pop_full_result

```
static uint32_t interp_pop_full_result (interp_hw_t * interp) [inline], [static]
```

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

**Parameters**

- `interp`: Interpolator instance, interp0 or interp1.

**Returns**

The content of the FULL register

4.1.11.4.11. interp_pop_lane_result

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

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

**Parameters**

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

**Returns**

The content of the lane result register

4.1.11.4.12. interp_restore

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

Restore an interpolator state.

**Parameters**

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

4.1.11.4.13. interp_save

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

Save the specified interpolator state.

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

**Parameters**

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

```c
static void interp_set_accumulator (interp_hw_t * interp, uint lane, uint32_t val) [inline], [static]
```

Sets the interpolator accumulator register by lane.

**Parameters**

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

4.1.11.4.15. interp_set_base

```c
static void interp_set_base (interp_hw_t * interp, uint lane, uint32_t val) [inline], [static]
```

Sets the interpolator base register by lane.

**Parameters**

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

4.1.11.4.16. interp_set_base_both

```c
static void interp_set_base_both (interp_hw_t * interp, uint32_t val) [inline], [static]
```

Sets the interpolator base registers simultaneously.

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

**Parameters**

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

4.1.11.4.17. interp_set_force_bits

```c
static void interp_set_force_bits (interp_hw_t * interp, uint lane, uint bits) [inline], [static]
```

Directly set the force bits on a specified lane.

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

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

**Parameters**

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

```c
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

4.1.11.4.19. interp_unclaim_lane_mask

```c
void interp_unclaim_lane_mask (interp_hw_t * interp, uint lane_mask)
```

Release previously claimed interpolator lanes.

**See also**
- `interp_claim_lane_mask`

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

4.1.11.5. interp_config

Interpolator configuration.

4.1.11.5.1. Detailed Description

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

4.1.11.5.2. Functions

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

Set the interpolator shift value.

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

Set the interpolator mask range.

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

Enable cross input.

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

Enable cross results.

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

Set sign extension.

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

Set raw add option.

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

Set blend mode.
static void interp_config_set_clamp (interp_config *c, bool clamp)
    Set interpolator clamp mode (Interpolator 1 only)
static void interp_config_set_force_bits (interp_config *c, uint bits)
    Set interpolator Force bits.
static interp_config interp_default_config (void)
    Get a default configuration.
static void interp_set_config (interp_hw_t *interp, uint lane, interp_config *config)
    Send configuration to a lane.

4.1.11.5.3. Function Documentation

interp_config_set_add_raw

static void interp_config_set_add_raw (interp_config * c, bool add_raw) [inline], [static]
Set raw add option.
When enabled, mask + shift is bypassed for LANE0 result. This does not affect the FULL result.

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

interp_config_set_blend

static void interp_config_set_blend (interp_config * c, bool blend) [inline], [static]
Set blend mode.
If enabled, LANE1 result is a linear interpolation between BASE0 and BASE1, controlled by the 8 LSBs of lane 1 shift and
mask value (a fractional number between 0 and 255/256ths)
LANE0 result does not have BASE0 added (yields only the 8 LSBs of lane 1 shift+mask value)
FULL result does not have lane 1 shift+mask value added (BASE2 + lane 0 shift+mask)
LANE1 SIGNED flag controls whether the interpolation is signed or unsig

Parameters
    c Pointer to interpolation config
    blend Set true to enable blend mode.

interp_config_set_clamp

static void interp_config_set_clamp (interp_config * c, bool clamp) [inline], [static]
Set interpolator clamp mode (Interpolator 1 only)
Only present on INTERP1 on each core. If CLAMP mode is enabled:
    • LANE0 result is a shifted and masked ACCUM0, clamped by a lower bound of BASE0 and an upper bound of
    BASE1.
    • Signedness of these comparisons is determined by LANE0_CTRL_SIGNED

Parameters
    c Pointer to interpolation config
    clamp Set true to enable clamp mode

interp_config_set_cross_input
static void interp_config_set_cross_input (interp_config * c, bool cross_input) [inline], [static]

Enable cross input.

Allows feeding of the accumulator content from the other lane back into this lane's shift+mask hardware. This will take effect even if the interp_config_set_add_raw option is set as the cross input mux is before the shift+mask bypass.

Parameters

c Pointer to interpolation config

cross_input If true, enable the cross input.

interp_config_set_cross_result

static void interp_config_set_cross_result (interp_config * c, bool cross_result) [inline], [static]

Enable cross results.

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

Parameters

c Pointer to interpolation config

cross_result If true, enables the cross result

interp_config_set_force_bits

static void interp_config_set_force_bits (interp_config * c, uint bits) [inline], [static]

Set interpolator Force bits.

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

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

Parameters

c Pointer to interpolation config

bits Sets the force bits to that specified. Range 0-3 (two bits)

interp_config_set_mask

static void interp_config_set_mask (interp_config * c, uint mask_lsb, uint mask_msb) [inline], [static]

Set the interpolator mask range.

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

Parameters

c Pointer to interpolation config

mask_lsb The least significant bit allowed to pass

mask_msb The most significant bit allowed to pass

interp_config_set_shift

static void interp_config_set_shift (interp_config * c, uint shift) [inline], [static]

Set the interpolator shift value.

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

Parameters

c Pointer to an interpolator config

shift Number of bits

interp_config_set_signed
static void interp_config_set_signed (interp_config * c, bool _signed) [inline], [static]

Set sign extension.

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

Parameters

c Pointer to interpolation config

_signed If true, enables sign extension

interp_default_config

static interp_config interp_default_config (void) [inline], [static]

Get a default configuration.

Returns

A default interpolation configuration

interp_set_config

static void interp_set_config (interp_hw_t * interp, uint lane, interp_config * config) [inline], [static]

Send configuration to a lane.

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

Parameters

interp Interpolator instance, interp0 or interp1.

lane The lane to set

config Pointer to interpolation config

4.1.12. hardware_irq

4.1.12.1. Detailed Description

Hardware interrupt handling

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

Interrupts are identified by a number from 0 to 31.

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

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

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).

NOTE

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

Interrupt Numbers

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

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

### 4.1.12.2. Typedefs

```c
typedef void(* irq_handler_t)(void)
```

Interrupt handler function type.

### 4.1.12.3. Functions

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

Set specified interrupt’s priority.

```c
uint irq_get_priority (uint num)
```

Get specified interrupt’s priority.

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

Enable or disable a specific interrupt on the executing core.

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

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

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

Enable/disable multiple interrupts on the executing core.

```c
void irq_set-exclusive_handler (uint num, irq_handler_t handler)
```

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

```c
irq_handler_t irq_get-exclusive_handler (uint num)
```

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

```c
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.

```c
void irq_remove-handler (uint num, irq_handler_t handler)
```

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

```c
bool irq_has-shared_handler (uint num)
```

Determine if the current handler for the given number is shared.

```c
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.

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

Clear a specific interrupt on the executing core.
void irq_set_pending (uint num)
    Force an interrupt to be pending on the executing core.

void user_irq_claim (uint irq_num)
    Claim ownership of a user IRQ on the calling core.

void user_irq_unclaim (uint irq_num)
    Mark a user IRQ as no longer used on the calling core.

int user_irq_claim_unused (bool required)
    Claim ownership of a free user IRQ on the calling core.

4.1.12.4. Typedef Documentation

4.1.12.4.1. irq_handler_t

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

All interrupts handlers should be of this type, and follow normal ARM EABI register saving conventions

4.1.12.5. Function Documentation

4.1.12.5.1. 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
  - Interrupt Numbers

- handler
  - The handler to set. See irq_handler_t

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

The order_priority uses higher values for higher priorities which is the opposite of the CPU interrupt priorities passed to irq_set_priority() which use lower values for higher priorities.

See also

irq_set_exclusive_handler()

4.1.12.5.2. irq_clear

static void irq_clear (uint int_num) [inline], [static]

Clear a specific interrupt on the executing core.

This method is only useful for "software" IRQs that are not connected to hardware (i.e. IRQs 26-31) as the NVIC always reflects the current state of the IRQ state of the hardware for hardware IRQs, and clearing of the IRQ state of the hardware is performed via the hardware’s registers instead.

Parameters

int_num Interrupt number

4.1.12.5.3. 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

See also

irq_set_exclusive_handler()

Returns

handler The handler if an exclusive handler is set for the IRQ, NULL if no handler is set or shared/shareable handlers are installed

4.1.12.5.4. irq_get_priority

uint irq_get_priority (uint num)

Get specified interrupt's priority.

Numerically-lower values indicate a higher priority. Hardware priorities range from 0 (highest priority) to 255 (lowest priority) 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

Parameters

num Interrupt number

Returns

the IRQ priority
4.1.12.5.1. irq_get_vtable_handler

```c
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

4.1.12.5.6. irq_has_shared_handler

```c
bool irq_has_shared_handler (uint num)
```

Determine if the current handler for the given number is shared.

**Parameters**

- `num` : Interrupt number [Interrupt Numbers]

**Returns**

true if the specified IRQ has a shared handler

4.1.12.5.7. irq_is_enabled

```c
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

4.1.12.5.8. irq_remove_handler

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

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

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

⚠ **NOTE**

This method may only be called from user (non IRQ code) or from within the handler itself (i.e. an IRQ handler may remove itself as part of handling the IRQ). Attempts to call from another IRQ will cause an assertion.

**Parameters**

- `num` : Interrupt number [Interrupt Numbers]
- `handler` : The handler to removed.

See also
irq_set_exclusive_handler()
irq_add_shared_handler()

4.1.12.5.9. 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
- enabled: true to enable the interrupt, false to disable

4.1.12.5.10. 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
- handler: The handler to set. See irq_handler_t

See also
irq_add_shared_handler()

4.1.12.5.11. 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

4.1.12.5.12. irq_set_pending

void irq_set_pending (uint num)
Force an interrupt to be pending on the executing core.

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

Parameters
- num: Interrupt number
4.1.12.5.13. irq_set_priority

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

Set specified interrupt's priority.

**Parameters**

- `num`: Interrupt number
- `hardware_priority`: Priority to set. Numerically-lower values indicate a higher priority. Hardware priorities range from 0 (highest priority) to 255 (lowest priority) 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

4.1.12.5.14. user_irq_claim

```c
void user_irq_claim (uint irq_num)
```

Claim ownership of a user IRQ on the calling core.

User IRQs are numbered 26-31 and are not connected to any hardware, but can be triggered by `irq_set_pending`.

**NOTE**

User IRQs are a core local feature; they cannot be used to communicate between cores. Therefore all functions dealing with User IRQs affect only the calling core.

This method explicitly claims ownership of a user IRQ, so other code can know it is being used.

**Parameters**

- `irq_num`: the user IRQ to claim

4.1.12.5.15. user_irq_claim_unused

```c
int user_irq_claim_unused (bool required)
```

Claim ownership of a free user IRQ on the calling core.

User IRQs are numbered 26-31 and are not connected to any hardware, but can be triggered by `irq_set_pending`.

**NOTE**

User IRQs are a core local feature; they cannot be used to communicate between cores. Therefore all functions dealing with User IRQs affect only the calling core.

This method explicitly claims ownership of an unused user IRQ if there is one, so other code can know it is being used.

**Parameters**

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

**Returns**

the user IRQ number or -1 if required was false, and none were free

4.1.12.5.16. user_irq_unclaim

```c
void user_irq_unclaim (uint irq_num)
```
Mark a user IRQ as no longer used on the calling core.

User IRQs are numbered 26-31 and are not connected to any hardware, but can be triggered by `irq_set_pending`.

**NOTE**

User IRQs are a core local feature; they cannot be used to communicate between cores. Therefore all functions dealing with User IRQs affect only the calling core.

This method explicitly releases ownership of a user IRQ, so other code can know it is free to use.

**NOTE**

It is customary to have disabled the irq and removed the handler prior to calling this method.

**Parameters**

- `irq_num` the irq irq_num to unclaim

### 4.1.13. hardware_pio

#### 4.1.13.1. Detailed Description

Programmable I/O (PIO) API

A programmable input/output block (PIO) is a versatile hardware interface which can support a number of different I/O 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 I/O, 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.

#### 4.1.13.2. Modules

- `sm_config` PIO state machine configuration.
- `pio_instructions` PIO instruction encoding.
4.1.13.3. Macros

- `#define pio0 pio0_hw`
- `#define pio1 pio1_hw`

4.1.13.4. Enumerations

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

```c
enum pio_mov_status_type {
    STATUS_TX_LESSTHAN = 0,
    STATUS_RX_LESSTHAN = 1
}  // MOV status types.
```

```c
enum pio_interrupt_source {
    pis_interrupt0 = PIO_INTR_SM0_LSB,
    pis_interrupt1 = PIO_INTR_SM1_LSB,
    pis_interrupt2 = PIO_INTR_SM2_LSB,
    pis_interrupt3 = PIO_INTR_SM3_LSB,
    pis_sm0_tx_fifo_not_full = PIO_INTR_SM0_TXNFULL_LSB,
    pis_sm1_tx_fifo_not_full = PIO_INTR_SM1_TXNFULL_LSB,
    pis_sm2_tx_fifo_not_full = PIO_INTR_SM2_TXNFULL_LSB,
    pis_sm3_tx_fifo_not_full = PIO_INTR_SM3_TXNFULL_LSB,
    pis_sm0_rx_fifo_not_empty = PIO_INTR_SM0_RXNEMPTY_LSB,
    pis_sm1_rx_fifo_not_empty = PIO_INTR_SM1_RXNEMPTY_LSB,
    pis_sm2_rx_fifo_not_empty = PIO_INTR_SM2_RXNEMPTY_LSB,
    pis_sm3_rx_fifo_not_empty = PIO_INTR_SM3_RXNEMPTY_LSB
}  // PIO interrupt source numbers for pio related IRQs.
```

4.1.13.5. Functions

```c
static void pio_sm_set_config (PIO pio, uint sm, const pio_sm_config *config)
    // Apply a state machine configuration to a state machine.
```

```c
static uint pio_get_index (PIO pio)
    // Return the instance number of a PIO instance.
```

```c
static void pio_gpio_init (PIO pio, uint pin)
    // Setup the function select for a GPIO to use output from the given PIO instance.
```

```c
static uint pio_get_dreq (PIO pio, uint sm, bool is_tx)
    // Return the DREQ to use for pacing transfers to/from a particular state machine FIFO.
```

```c
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.
```

```c
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.
```

```c
uint pio_add_program (PIO pio, const pio_program_t *program)
    // Attempt to load the program, panicking if not possible.
```

```c
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.
```

```c
void pio_remove_program (PIO pio, const pio_program_t *program, uint loaded_offset)
    // Remove a program from a PIO instance's instruction memory.
```

```c
void pio_clear_instruction_memory (PIO pio)
    // Clears all of a PIO instance's instruction memory.
```

```c
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.
```
static void pio_sm_set_enabled (PIO pio, uint sm, bool enabled)
   Enable or disable a PIO state machine.
static void pio_set_sm_mask_enabled (PIO pio, uint32_t mask, bool enabled)
   Enable or disable multiple PIO state machines.
static void pio_sm_restart (PIO pio, uint sm)
   Restart a state machine with a known state.
static void pio_restart_sm_mask (PIO pio, uint32_t mask)
   Restart multiple state machine with a known state.
static void pio_sm_clkdiv_restart (PIO pio, uint sm)
   Restart a state machine's clock divider from a phase of 0.
static void pio_clkdiv_restart_sm_mask (PIO pio, uint32_t mask)
   Restart multiple state machines' clock dividers from a phase of 0.
static void pio_enable_sm_mask_in_sync (PIO pio, uint32_t mask)
   Enable multiple PIO state machines synchronizing their clock dividers.
static void pio_set_irq0_source_enabled (PIO pio, enum pio_interrupt_source source, bool enabled)
   Enable/Disable a single source on a PIO's IRQ 0.
static void pio_set_irq1_source_enabled (PIO pio, enum pio_interrupt_source source, bool enabled)
   Enable/Disable a single source on a PIO's IRQ 1.
static void pio_set_irq0_source_mask_enabled (PIO pio, uint32_t source_mask, bool enabled)
   Enable/Disable multiple sources on a PIO's IRQ 0.
static void pio_set_irq1_source_mask_enabled (PIO pio, uint32_t source_mask, bool enabled)
   Enable/Disable multiple sources on a PIO's IRQ 1.
static void pio_set_irqn_source_enabled (PIO pio, uint irq_index, enum pio_interrupt_source source, bool enabled)
   Enable/Disable a single source on a PIO's specified (0/1) IRQ index.
static void pio_set_irqn_source_mask_enabled (PIO pio, uint irq_index, uint32_t source_mask, bool enabled)
   Enable/Disable multiple sources on a PIO's specified (0/1) IRQ index.
static bool pio_interrupt_get (PIO pio, uint pio_interrupt_num)
   Determine if a particular PIO interrupt is set.
static void pio_interrupt_clear (PIO pio, uint pio_interrupt_num)
   Clear a particular PIO interrupt.
static uint8_t pio_sm_get_pc (PIO pio, uint sm)
   Return the current program counter for a state machine.
static void pio_sm_exec (PIO pio, uint sm, uint instr)
   Immediately execute an instruction on a state machine.
static bool pio_sm_is_exec_stalled (PIO pio, uint sm)
   Determine if an instruction set by pio_sm_exec() is stalled executing.
static void pio_sm_exec_wait_blocking (PIO pio, uint sm, uint instr)
   Immediately execute an instruction on a state machine and wait for it to complete.
static void pio_sm_set_wrap (PIO pio, uint sm, uint wrap_target, uint wrap)
   Set the current wrap configuration for a state machine.
static void pio_sm_set_out_pins (PIO pio, uint sm, uint out_base, uint out_count)
    Set the current 'out' pins for a state machine.
static void pio_sm_set_set_pins (PIO pio, uint sm, uint set_base, uint set_count)
    Set the current 'set' pins for a state machine.
static void pio_sm_set_in_pins (PIO pio, uint sm, uint in_base)
    Set the current 'in' pins for a state machine.
static void pio_sm_set_sideset_pins (PIO pio, uint sm, uint sideset_base)
    Set the current 'sideset' pins for a state machine.
static void pio_sm_put (PIO pio, uint sm, uint32_t data)
    Write a word of data to a state machine's TX FIFO.
static uint32_t pio_sm_get (PIO pio, uint sm)
    Read a word of data from a state machine's RX FIFO.
static bool pio_sm_is_rx_fifo_full (PIO pio, uint sm)
    Determine if a state machine's RX FIFO is full.
static bool pio_sm_is_rx_fifo_empty (PIO pio, uint sm)
    Determine if a state machine's RX FIFO is empty.
static uint pio_sm_get_rx_fifo_level (PIO pio, uint sm)
    Return the number of elements currently in a state machine's RX FIFO.
static bool pio_sm_is_tx_fifo_full (PIO pio, uint sm)
    Determine if a state machine's TX FIFO is full.
static bool pio_sm_is_tx_fifo_empty (PIO pio, uint sm)
    Determine if a state machine's TX FIFO is empty.
static uint pio_sm_get_tx_fifo_level (PIO pio, uint sm)
    Return the number of elements currently in a state machine's TX FIFO.
static void pio_sm_put_blocking (PIO pio, uint sm, uint32_t data)
    Write a word of data to a state machine's TX FIFO, blocking if the FIFO is full.
static uint32_t pio_sm_get_blocking (PIO pio, uint sm)
    Read a word of data from a state machine's RX FIFO, blocking if the FIFO is empty.
void pio_sm_drain_tx_fifo (PIO pio, uint sm)
    Empty out a state machine's TX FIFO.
static void pio_sm_set_clkdiv_int_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
static void pio_sm_set_clkdiv (PIO pio, uint sm, float div)
    Set the current clock divider for a state machine
static void pio_sm_clear_fifos (PIO pio, uint sm)
    Clear a state machine's TX and RX FIFOs.
void pio_sm_set_pins (PIO pio, uint sm, uint32_t pin_values)
    Use a state machine to set a value on all pins for the PIO instance.
void pio_sm_set_pins_with_mask (PIO pio, uint sm, uint32_t pin_values, uint32_t pin_mask)
    Use a state machine to set a value on multiple pins for the PIO instance.
void pio_sm_set_pindirs_with_mask (PIO pio, uint sm, uint32_t pin_dirs, uint32_t pin_mask)

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

void pio_sm_set_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.

void pio_sm_claim (PIO pio, uint sm)

Mark a state machine as used.

void pio_claim_sm_mask (PIO pio, uint sm_mask)

Mark multiple state machines as used.

void pio_sm_unclaim (PIO pio, uint sm)

Mark a state machine as no longer used.

int pio_claim_unused_sm (PIO pio, bool required)

Claim a free state machine on a PIO instance.

bool pio_sm_is_claimed (PIO pio, uint sm)

Determine if a PIO state machine is claimed.

4.1.13.6. Macro Definition Documentation

4.1.13.6.1. pio0

#define pio0 pio0_hw

Identifier for the first (PIO 0) hardware PIO instance (for use in PIO functions).

e.g. pio_gpio_init(pio0, 5)

4.1.13.6.2. pio1

#define pio1 pio1_hw

Identifier for the second (PIO 1) hardware PIO instance (for use in PIO functions).

e.g. pio_gpio_init(pio1, 5)

4.1.13.7. Enumeration Type Documentation

4.1.13.7.1. pio_fifo_join

enum pio_fifo_join

FIFO join states.

4.1.13.7.2. pio_mov_status_type

enum pio_mov_status_type

MOV status types.
4.1.13.7.3. pio_interrupt_source

enum pio_interrupt_source
PIO interrupt source numbers for pio related IRQs.

4.1.13.8. Function Documentation

4.1.13.8.1. pio_add_program

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

See also
pio_can_add_program() if you need to check whether the program can be loaded

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

Returns
the instruction memory offset the program is loaded at

4.1.13.8.2. 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.

See also
pio_can_add_program_at_offset() if you need to check whether the program can be loaded

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

4.1.13.8.3. 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
4.1.13.8.4. pio_can_add_program_at_offset

```c
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

4.1.13.8.5. pio_claim_sm_mask

```c
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

4.1.13.8.6. pio_claim_unused_sm

```c
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

4.1.13.8.7. pio_clear_instruction_memory

```c
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

4.1.13.8.8. pio_clkdiv_restart_sm_mask

```c
static void pio_clkdiv_restart_sm_mask (PIO pio, uint32_t mask) [inline], [static]
```

Restart multiple state machines' clock dividers from a phase of 0.
Each state machine’s clock divider is a free-running piece of hardware, that generates a pattern of clock enable pulses for the state machine, based only on the configured integer/fractional divisor. The pattern of running/halted cycles slows the state machine’s execution to some controlled rate.

This function simultaneously clears the integer and fractional phase accumulators of multiple state machines’ clock dividers. If these state machines all have the same integer and fractional divisors configured, their clock dividers will run in precise deterministic lockstep from this point.

With their execution clocks synchronised in this way, it is then safe to e.g. have multiple state machines performing a ‘wait irq’ on the same flag, and all clear it on the same cycle.

Also note that this function can be called whilst state machines are running (e.g. if you have just changed the clock divisors of some state machines and wish to resynchronise them), and that disabling a state machine does not halt its clock divider: that is, if multiple state machines have their clocks synchronised, you can safely disable and reenable one of the state machines without losing synchronisation.

Parameters

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

### 4.1.13.8.9. pio_enable_sm_mask_in_sync

```c
static void pio_enable_sm_mask_in_sync (PIO pio, uint32_t mask) [inline], [static]
```

Enable multiple PIO state machines synchronizing their clock dividers.

This is equivalent to calling both `pio_set_sm_mask_enabled()` and `pio_clkdiv_restart_sm_mask()` on the same clock cycle. All state machines specified by ‘mask’ are started simultaneously and, assuming they have the same clock divisors, their divided clocks will stay precisely synchronised.

Parameters

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

### 4.1.13.8.10. pio_get_dreq

```c
static uint pio_get_dreq (PIO pio, uint sm, bool is_tx) [inline], [static]
```

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

Parameters

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

### 4.1.13.8.11. pio_get_index

```c
static uint pio_get_index (PIO pio) [inline], [static]
```

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)
4.1.13.8.12. pio_gpio_init

```c
static void pio_gpio_init (PIO pio, uint pin) [inline], [static]
```

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

PIO appears as an alternate function in the GPIO muxing, just like an SPI or UART. This function configures that multiplexing to connect a given PIO instance to a GPIO. Note that this is not necessary for a state machine to be able to read the input value from a GPIO, but only for it to set the output value or output enable.

Parameters

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

4.1.13.8.13. pio_interrupt_clear

```c
static void pio_interrupt_clear (PIO pio, uint pio_interrupt_num) [inline], [static]
```

Clear a particular PIO interrupt.

Parameters

- **pio**  The PIO instance; either pio0 or pio1
- **pio_interrupt_num**  the PIO interrupt number 0-7

4.1.13.8.14. pio_interrupt_get

```c
static bool pio_interrupt_get (PIO pio, uint pio_interrupt_num) [inline], [static]
```

Determine if a particular PIO interrupt is set.

Parameters

- **pio**  The PIO instance; either pio0 or pio1
- **pio_interrupt_num**  the PIO interrupt number 0-7

Returns

true if corresponding PIO interrupt is currently set

4.1.13.8.15. pio_remove_program

```c
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

4.1.13.8.16. pio_restart_sm_mask

```c
static void pio_restart_sm_mask (PIO pio, uint32_t mask) [inline], [static]
```

Restart multiple state machine with a known state.

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

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

4.1.13.8.17. pio_set_irq0_source_enabled

static void pio_set_irq0_source_enabled (PIO pio, enum pio_interrupt_source source, bool enabled) [inline], [static]

Enable/Disable a single source on a PIO’s IRQ 0.

Parameters
- pio  The PIO instance; either pio0 or pio1
- source the source number (see pio_interrupt_source)
- enabled true to enable IRQ 0 for the source, false to disable.

4.1.13.8.18. pio_set_irq0_source_mask_enabled

static void pio_set_irq0_source_mask_enabled (PIO pio, uint32_t source_mask, bool enabled) [inline], [static]

Enable/Disable multiple sources on a PIO’s IRQ 0.

Parameters
- pio  The PIO instance; either pio0 or pio1
- source_mask Mask of bits, one for each source number (see pio_interrupt_source) to affect
- enabled true to enable all the sources specified in the mask on IRQ 0, false to disable all the sources specified in the mask on IRQ 0

4.1.13.8.19. pio_set_irq1_source_enabled

static void pio_set_irq1_source_enabled (PIO pio, enum pio_interrupt_source source, bool enabled) [inline], [static]

Enable/Disable a single source on a PIO’s IRQ 1.

Parameters
- pio  The PIO instance; either pio0 or pio1
- source the source number (see pio_interrupt_source)
- enabled true to enable IRQ 0 for the source, false to disable.

4.1.13.8.20. pio_set_irq1_source_mask_enabled

static void pio_set_irq1_source_mask_enabled (PIO pio, uint32_t source_mask, bool enabled) [inline], [static]

Enable/Disable multiple sources on a PIO’s IRQ 1.

Parameters
- pio  The PIO instance; either pio0 or pio1
- source_mask Mask of bits, one for each source number (see pio_interrupt_source) to affect
- enabled true to enable all the sources specified in the mask on IRQ 1, false to disable all the source specified in the mask on IRQ 1
4.1.13.8.21. pio_set_irqn_source_enabled

static void pio_set_irqn_source_enabled (PIO pio, uint irq_index, enum pio_interrupt_source source, bool enabled) [inline], [static]

Enable/Disable a single source on a PIO's specified (0/1) IRQ index.

Parameters
- pio: The PIO instance; either pio0 or pio1
- irq_index: the IRQ index; either 0 or 1
- source: the source number (see pio_interrupt_source)
- enabled: true to enable the source on the specified IRQ, false to disable.

4.1.13.8.22. pio_set_irqn_source_mask_enabled

static void pio_set_irqn_source_mask_enabled (PIO pio, uint irq_index, uint32_t source_mask, bool enabled) [inline], [static]

Enable/Disable multiple sources on a PIO's specified (0/1) IRQ index.

Parameters
- pio: The PIO instance; either pio0 or pio1
- irq_index: the IRQ index; either 0 or 1
- source_mask: Mask of bits, one for each source number (see pio_interrupt_source) to affect
- enabled: true to enable all the sources specified in the mask on the specified IRQ, false to disable all the sources specified in the mask on the specified IRQ

4.1.13.8.23. pio_set_sm_mask_enabled

static void pio_set_sm_mask_enabled (PIO pio, uint32_t mask, bool enabled) [inline], [static]

Enable or disable multiple PIO state machines.

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

See also
pio_enable_sm_mask_in_sync() if you wish to enable multiple state machines and ensure their clock dividers are in sync.

Parameters
- pio: The PIO instance; 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

4.1.13.8.24. 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)

4.1.13.8.25. pio_sm_clear_fifos

static void pio_sm_clear_fifos (PIO pio, uint sm) [inline], [static]
Clear a state machine's TX and RX FIFOs.
Parameters

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

4.1.13.8.26. pio_sm_clkdiv_restart

static void pio_sm_clkdiv_restart (PIO pio, uint sm) [inline], [static]
Restart a state machine's clock divider from a phase of 0.
Each state machine's clock divider is a free-running piece of hardware, that generates a pattern of clock enable pulses for the state machine, based only on the configured integer/fractional divisor. The pattern of running/halted cycles slows the state machine's execution to some controlled rate.
This function clears the divider's integer and fractional phase accumulators so that it restarts this pattern from the beginning. It is called automatically by pio_sm_init() but can also be called at a later time, when you enable the state machine, to ensure precisely consistent timing each time you load and run a given PIO program.
More commonly this hardware mechanism is used to synchronise the execution clocks of multiple state machines – see pio_clkdiv_restart_sm_mask().
Parameters

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

4.1.13.8.27. pio_sm_drain_tx_fifo

void pio_sm_drain_tx_fifo (PIO pio, uint sm)
Empty out a state machine's TX FIFO.
This method executes pull instructions on the state machine until the TX FIFO is empty. This disturbs the contents of the OSR, so see also pio_sm_clear_fifos() which clears both FIFOs but leaves the state machine's internal state undisturbed.
Parameters

pio  The PIO instance; either pio0 or pio1
sm  State machine index (0..3)
See also
pio_sm_clear_fifos()
4.1.13.8.28. pio_sm_exec

```c
static void pio_sm_exec (PIO pio, uint sm, uint instr) [inline], [static]
```

Immediately execute an instruction on a state machine.

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

**See also**

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

**Parameters**

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

4.1.13.8.29. pio_sm_exec_wait_blocking

```c
static void pio_sm_exec_wait_blocking (PIO pio, uint sm, uint instr) [inline], [static]
```

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

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

**See also**

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

**Parameters**

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

4.1.13.8.30. pio_sm_get

```c
static uint32_t pio_sm_get (PIO pio, uint sm) [inline], [static]
```

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

This is a raw FIFO access that does not check for emptiness. If the FIFO is empty, the hardware ignores the attempt to read from the FIFO (the FIFO remains in an empty state following the read) and the sticky RXUNDER flag for this FIFO is set in FDEBUG to indicate that the system tried to read from this FIFO when empty. The data returned by this function is undefined when the FIFO is empty.

**Parameters**

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

**See also**

`pio_sm_get_blocking()`
4.1.13.8.31. pio_sm_get_blocking

static uint32_t pio_sm_get_blocking (PIO pio, uint sm) [inline], [static]
Read a word of data from a state machine's RX FIFO, blocking if the FIFO is empty.

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

4.1.13.8.32. pio_sm_get_pc

static uint8_t pio_sm_get_pc (PIO pio, uint sm) [inline], [static]
Return the current program counter for a state machine.

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

Returns
the program counter

4.1.13.8.33. pio_sm_get_rx_fifo_level

static uint pio_sm_get_rx_fifo_level (PIO pio, uint sm) [inline], [static]
Return the number of elements currently in a state machine's RX FIFO.

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

Returns
the number of elements in the RX FIFO

4.1.13.8.34. pio_sm_get_tx_fifo_level

static uint pio_sm_get_tx_fifo_level (PIO pio, uint sm) [inline], [static]
Return the number of elements currently in a state machine's TX FIFO.

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

Returns
the number of elements in the TX FIFO

4.1.13.8.35. 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:

- Disables the state machine (if running)
- Clears the FIFOs
- Applies the configuration specified by 'config'
- Resets any internal state e.g. shift counters
- Jumps to the initial program location given by 'initial_pc'

The state machine is left 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)

4.1.13.8.36. pio_sm_is_claimed

```cpp
bool pio_sm_is_claimed (PIO pio, uint sm)
```

Determine if a PIO state machine is claimed.

Parameters

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

Returns

true if claimed, false otherwise

See also

- `pio_sm_claim`
- `pio_claim_sm_mask`

4.1.13.8.37. pio_sm_is_exec_stalled

```cpp
static bool pio_sm_is_exec_stalled (PIO pio, uint sm) [inline], [static]
```

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

Parameters

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

Returns

true if the executed instruction is still running (stalled)

4.1.13.8.38. pio_sm_is_rx_fifo_empty

```cpp
static bool pio_sm_is_rx_fifo_empty (PIO pio, uint sm) [inline], [static]
```

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

Parameters
The PIO instance; either pio0 or pio1

sm  State machine index (0..3)

Returns
true if the RX FIFO is empty

4.1.13.8.39. pio_sm_is_rx_fifo_full

static bool pio_sm_is_rx_fifo_full (PIO pio, uint sm) [inline], [static]
Determine if a state machine’s RX FIFO is full.

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

Returns
true if the RX FIFO is full

4.1.13.8.40. pio_sm_is_tx_fifo_empty

static bool pio_sm_is_tx_fifo_empty (PIO pio, uint sm) [inline], [static]
Determine if a state machine’s TX FIFO is empty.

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

Returns
true if the TX FIFO is empty

4.1.13.8.41. pio_sm_is_tx_fifo_full

static bool pio_sm_is_tx_fifo_full (PIO pio, uint sm) [inline], [static]
Determine if a state machine’s TX FIFO is full.

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

Returns
true if the TX FIFO is full

4.1.13.8.42. pio_sm_put

static void pio_sm_put (PIO pio, uint sm, uint32_t data) [inline], [static]
Write a word of data to a state machine’s TX FIFO.

This is a raw FIFO access that does not check for fullness. If the FIFO is full, the FIFO contents and state are not affected by the write attempt. Hardware sets the TXOVER sticky flag for this FIFO in FDEBUG, to indicate that the system attempted to write to a full FIFO.
Parameters

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

See also

`pio_sm_put_blocking()`

### 4.1.13.8.43. pio_sm_put_blocking

```c
static void pio_sm_put_blocking (PIO pio, uint sm, uint32_t data) [inline], [static]
```

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

Parameters

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

### 4.1.13.8.44. pio_sm_restart

```c
static void pio_sm_restart (PIO pio, uint sm) [inline], [static]
```

Restart a state machine with a known state.

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

Parameters

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

### 4.1.13.8.45. pio_sm_set_clkdiv

```c
static void pio_sm_set_clkdiv (PIO pio, uint sm, float div) [inline], [static]
```

set the current clock divider for a state machine

Parameters

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

### 4.1.13.8.46. pio_sm_set_clkdiv_int_frac

```c
static void pio_sm_set_clkdiv_int_frac (PIO pio, uint sm, uint16_t div_int, uint8_t div_frac) [inline], [static]
```

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

Parameters

- **pio**: The PIO instance; either **pio0** or **pio1**
smState machine index (0..3)
div_intthe integer part of the clock divider
div_fracthe fractional part of the clock divider in 1/256s

4.1.13.8.47. pio_sm_set_config

static void pio_sm_set_config (PIO pio, uint sm, const pio_sm_config * config) [inline], [static]

Apply a state machine configuration to a state machine.

Parameters
pioHandle to PIO instance; either pio0 or pio1
smState machine index (0..3)
cfgthe configuration to apply

4.1.13.8.48. 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
pioThe PIO instance; either pio0 or pio1
smState 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

4.1.13.8.49. pio_sm_set_enabled

static void pio_sm_set_enabled (PIO pio, uint sm, bool enabled) [inline], [static]

Enable or disable a PIO state machine.

Parameters
pioThe PIO instance; either pio0 or pio1
smState machine index (0..3)
enabledtrue to enable the state machine; false to disable

4.1.13.8.50. pio_sm_set_in_pins

static void pio_sm_set_in_pins (PIO pio, uint sm, uint in_base) [inline], [static]

Set the current ‘in’ pins for a state machine.

Can overlap with the ‘out’, ‘set’ and ‘sideset’ pins
4.1.13.8.51. pio_sm_set_out_pins

static void pio_sm_set_out_pins (PIO pio, uint sm, uint out_base, uint out_count) [inline], [static]
Set the current 'out' pins for a state machine.
Can overlap with the 'in', '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 use as input
out_count 0-32 Number of pins to set.

4.1.13.8.52. 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.

4.1.13.8.53. 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

4.1.13.8.54. 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.

4.1.13.8.55. pio_sm_set_set_pins

static void pio_sm_set_set_pins (PIO pio, uint sm, uint set_base, uint set_count) [inline], [static]

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

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.

4.1.13.8.56. pio_sm_set_sideset_pins

static void pio_sm_set_sideset_pins (PIO pio, uint sm, uint sideset_base) [inline], [static]

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

Can overlap with the 'in', 'out' and 'sideset' pins

Parameters
pio  The PIO instance; either pio0 or pio1
sm  State machine index (0..3)
sideset_base  0-31 base pin for 'side set'

4.1.13.8.57. pio_sm_set_wrap

static void pio_sm_set_wrap (PIO pio, uint sm, uint wrap_target, uint wrap) [inline], [static]

Set the current wrap configuration for a state machine.

Parameters
pio

The PIO instance; 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

4.1.13.8.58. 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)

4.1.13.9. sm_config

PIO state machine configuration.

4.1.13.9.1. Detailed Description

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

4.1.13.9.2. Functions

static void sm_config_set_out_pins (pio_sm_config *c, uint out_base, uint out_count)

Set the 'out' pins in a state machine configuration.

static void sm_config_set_set_pins (pio_sm_config *c, uint set_base, uint set_count)

Set the 'set' pins in a state machine configuration.

static void sm_config_set_in_pins (pio_sm_config *c, uint in_base)

Set the 'in' pins in a state machine configuration.

static void sm_config_set_sideset_pins (pio_sm_config *c, uint sideset_base)

Set the 'sideset' pins in a state machine configuration.

static void sm_config_set_sideset (pio_sm_config *c, uint bit_count, bool optional, bool pindirs)

Set the 'sideset' options in a state machine configuration.

static void sm_config_set_clkdiv_int_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.

static void sm_config_set_clkdiv (pio_sm_config *c, float div)

Set the state machine clock divider (from a floating point value) in a state machine configuration.

static void sm_config_set_wrap (pio_sm_config *c, uint wrap_target, uint wrap)

Set the wrap addresses in a state machine configuration.
static void sm_config_set_jmp_pin (pio_sm_config *c, uint pin)
    Set the 'jmp' pin in a state machine configuration.

static void sm_config_set_in_shift (pio_sm_config *c, bool shift_right, bool autopush, uint push_threshold)
    Setup 'in' shifting parameters in a state machine configuration.

static void sm_config_set_out_shift (pio_sm_config *c, bool shift_right, bool autopull, uint pull_threshold)
    Setup 'out' shifting parameters in a state machine configuration.

static void sm_config_set_fifo_join (pio_sm_config *c, enum pio_fifo_join join)
    Setup the FIFO joining in a state machine configuration.

static void sm_config_set_out_special (pio_sm_config *c, bool sticky, bool has_enable_pin, uint enable_pin_index)
    Set special 'out' operations in a state machine configuration.

static void sm_config_set_mov_status (pio_sm_config *c, enum pio_mov_status_type status_sel, uint status_n)
    Set source for 'mov status' in a state machine configuration.

static pio_sm_config pio_get_default_sm_config (void)
    Get the default state machine configuration.

### 4.1.13.9.3. Function Documentation

**pio_get_default_sm_config**

```c
static pio_sm_config pio_get_default_sm_config (void) [inline], [static]
```

Get the default state machine configuration.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out Pins</td>
<td>32 starting at 0</td>
</tr>
<tr>
<td>Set Pins</td>
<td>0 starting at 0</td>
</tr>
<tr>
<td>In Pins (base)</td>
<td>0</td>
</tr>
<tr>
<td>Side Set Pins (base)</td>
<td>0</td>
</tr>
<tr>
<td>Side Set</td>
<td>disabled</td>
</tr>
<tr>
<td>Wrap</td>
<td>wrap=31, wrap_to=0</td>
</tr>
<tr>
<td>In Shift</td>
<td>shift_direction=right, autopush=false, push_threshold=32</td>
</tr>
<tr>
<td>Out Shift</td>
<td>shift_direction=right, autopull=false, pull_threshold=32</td>
</tr>
<tr>
<td>Jmp Pin</td>
<td>0</td>
</tr>
<tr>
<td>Out Special</td>
<td>sticky=false, has_enable_pin=false, enable_pin_index=0</td>
</tr>
<tr>
<td>Mov Status</td>
<td>status_sel=STATUS_TX_LESSTHAN, n=0</td>
</tr>
</tbody>
</table>

**Returns**

the default state machine configuration which can then be modified.

**sm_config_set_clkdiv**

```c
static void sm_config_set_clkdiv (pio_sm_config *c, float div) [inline], [static]
```

Set the state machine clock divider (from a floating point value) in a state machine configuration.

The clock divider slows the state machine’s execution by masking the system clock on some cycles, in a repeating pattern, so that the state machine does not advance. Effectively this produces a slower clock for the state machine to run from, which can be used to generate e.g. a particular UART baud rate. See the datasheet for further detail.
Parameters

c Pointer to the configuration structure to modify

div The fractional divisor to be set. 1 for full speed. An integer clock divisor of n will cause the state machine to run 1 cycle in every n. Note that for small n, the jitter introduced by a fractional divider (e.g. 2.5) may be unacceptable although it will depend on the use case.

sm_config_set_clkdiv_int_frac

static void sm_config_set_clkdiv_int_frac (pio_sm_config * c, uint16_t div_int, uint8_t div_frac) [inline], [static]
Set the state machine clock divider (from integer and fractional parts - 16:8) in a state machine configuration.
The clock divider can slow the state machine's execution to some rate below the system clock frequency, by enabling the state machine on some cycles but not on others, in a regular pattern. This can be used to generate e.g. a given UART baud rate. See the datasheet for further detail.

Parameters

c Pointer to the configuration structure to modify

div_int Integer part of the divisor

div_frac Fractional part in 1/256ths

See also

sm_config_set_clkdiv()

sm_config_set_fifo_join

static void sm_config_set_fifo_join (pio_sm_config * c, enum pio_fifo_join join) [inline], [static]
Setup the FIFO joining in a state machine configuration.

Parameters

c Pointer to the configuration structure to modify

join Specifies the join type.

See also

enum pio_fifo_join

sm_config_set_in_pins

static void sm_config_set_in_pins (pio_sm_config * c, uint in_base) [inline], [static]
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 use as input

sm_config_set_in_shift

static void sm_config_set_in_shift (pio_sm_config * c, bool shift_right, bool autopush, uint push_threshold) [inline], [static]
Setup 'in' shifting parameters in a state machine configuration.

Parameters

c Pointer to the configuration structure to modify

shift_right true to shift ISR to right, false to shift ISR to left

4.1. Hardware APIs
Whether autopush is enabled

Threshold in bits to shift in before auto/conditional re-pushing of the ISR

**sm_config_set_jmp_pin**

```c
static void sm_config_set_jmp_pin (pio_sm_config * c, uint pin) [inline], [static]
```

Set the 'jmp' pin in a state machine configuration.

**Parameters**

- **c** Pointer to the configuration structure to modify
- **pin** The raw GPIO pin number to use as the source for a jmp pin instruction

**sm_config_set_mov_status**

```c
static void sm_config_set_mov_status (pio_sm_config * c, enum pio_mov_status_type status_sel, uint status_n) [inline], [static]
```

Set source for 'mov status' in a state machine configuration.

**Parameters**

- **c** Pointer to the configuration structure to modify
- **status_sel** the status operation selector.
- **status_n** parameter for the mov status operation (currently a bit count)

**sm_config_set_out_pins**

```c
static void sm_config_set_out_pins (pio_sm_config * c, uint out_base, uint out_count) [inline], [static]
```

Set the 'out' pins in a state machine configuration.

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.

**sm_config_set_out_shift**

```c
static void sm_config_set_out_shift (pio_sm_config * c, bool shift_right, bool autopull, uint pull_threshold) [inline], [static]
```

Setup 'out' shifting parameters in a state machine configuration.

**Parameters**

- **c** Pointer to the configuration structure to modify
- **shift_right** true to shift OSR to right, false to shift OSR to left
- **autopull** whether autopull is enabled
- **pull_threshold** threshold in bits to shift out before auto/conditional re-pulling of the OSR

**sm_config_set_out_special**

```c
static void sm_config_set_out_special (pio_sm_config * c, bool sticky, bool has_enable_pin, uint enable_pin_index)
```

Raspberry Pi Pico C/C++ SDK

4.1. Hardware APIs
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

**sm_config_set_set_pins**

```
static void sm_config_set_set_pins (pio_sm_config * c, uint set_base, uint set_count) [inline], [static]
```

Set the 'set' pins in a state machine configuration.

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.

**sm_config_set_sideset**

```
static void sm_config_set_sideset (pio_sm_config * c, uint bit_count, bool optional, bool pindirs) [inline], [static]
```

Set the 'sideset' options in a state machine configuration.

**Parameters**

- **c** Pointer to the configuration structure to modify
- **bit_count** Number of bits to steal from delay field in the instruction for use of side set (max 5)
- **optional** True if the topmost side set bit is used as a flag for whether to apply side set on that instruction
- **pindirs** True if the side set affects pin directions rather than values

**sm_config_set_sideset_pins**

```
static void sm_config_set_sideset_pins (pio_sm_config * c, uint sideset_base) [inline], [static]
```

Set the 'sideset' pins in a state machine configuration.

Can overlap with the 'in', 'out' and 'set' pins

**Parameters**

- **c** Pointer to the configuration structure to modify
- **sideset_base** 0-31 base pin for 'side set'

**sm_config_set_wrap**

```
static void sm_config_set_wrap (pio_sm_config * c, uint wrap_target, uint wrap) [inline], [static]
```

Set the wrap addresses in a state machine configuration.

**Parameters**

- **c** Pointer to the configuration structure to modify
- **wrap_target** the instruction memory address to wrap to
- **wrap** the instruction memory address after which to set the program counter to wrap_target if the instruction does not itself update the program_counter
4.1.13.9.4. pio_instructions

PIO instruction encoding.

Detailed Description

Functions for generating PIO instruction encodings programmatically. In debug builds
PARAM_ASSERTIONS_ENABLEDPIO_INSTRUCTIONS can be set to 1 to enable validation of encoding function parameters.

For fuller descriptions of the instructions in question see the “RP2040 Datasheet”

Enumerations

```c
enum pio_src_dest { pio_pins = 0u, pio_x = 1u, pio_y = 2u, pio_null = 3u | 0x20u | 0x80u, pio_pindirs = 4u | 0x08u | 0x40u | 0x80u, pio_exec_mov = 4u | 0x08u | 0x10u | 0x20u | 0x40u, pio_status = 5u | 0x08u | 0x10u | 0x20u | 0x80u, pio_pc = 5u | 0x08u | 0x20u | 0x40u, pio_isr = 6u | 0x20u, pio_osr = 7u | 0x10u | 0x20u, pio_exec_out = 7u | 0x08u | 0x20u | 0x40u | 0x80u }
```

Enumeration of values to pass for source/destination args for instruction encoding functions.

Functions

```c
static uint pio_encode_delay (uint cycles)
    Encode just the delay slot bits of an instruction.
static uint pio_encode_sideset (uint sideset_bit_count, uint value)
    Encode just the side set bits of an instruction (in non optional side set mode)
static uint pio_encode_sideset_opt (uint sideset_bit_count, uint value)
    Encode just the side set bits of an instruction (in optional -opt side set mode)
static uint pio_encode_jmp (uint addr)
    Encode an unconditional JMP instruction.
static uint pio_encode_jmp_not_x (uint addr)
    Encode a conditional JMP if scratch X zero instruction.
static uint pio_encode_jmp_x_dec (uint addr)
    Encode a conditional JMP if scratch X non-zero (and post-decrement X) instruction.
static uint pio_encode_jmp_not_y (uint addr)
    Encode a conditional JMP if scratch Y zero instruction.
static uint pio_encode_jmp_y_dec (uint addr)
    Encode a conditional JMP if scratch Y non-zero (and post-decrement Y) instruction.
static uint pio_encode_jmp_x_ne_y (uint addr)
    Encode a conditional JMP if scratch X not equal scratch Y instruction.
static uint pio_encode_jmp_pin (uint addr)
    Encode a conditional JMP if input pin high instruction.
static uint pio_encode_jmp_not_osre (uint addr)
    Encode a conditional JMP if output shift register not empty instruction.
static uint pio_encode_wait_gpio (bool polarity, uint gpio)
    Encode a WAIT for GPIO pin instruction.
static uint pio_encode_wait_pin (bool polarity, uint pin)
    Encode a WAIT for pin instruction.
static uint pio_encode_wait_irq (bool polarity, bool relative, uint irq)
    Encode a WAIT for IRQ instruction.
```
static uint pio_encode_in (enum pio_src_dest src, uint count)
    Encode an IN instruction.

static uint pio_encode_out (enum pio_src_dest dest, uint count)
    Encode an OUT instruction.

static uint pio_encode_push (bool if_full, bool block)
    Encode a PUSH instruction.

static uint pio_encode_pull (bool if_empty, bool block)
    Encode a PULL instruction.

static uint pio_encode_mov (enum pio_src_dest dest, enum pio_src_dest src)
    Encode a MOV instruction.

static uint pio_encode_mov_not (enum pio_src_dest dest, enum pio_src_dest src)
    Encode a MOV instruction with bit invert.

static uint pio_encode_mov_reverse (enum pio_src_dest dest, enum pio_src_dest src)
    Encode a MOV instruction with bit reverse.

static uint pio_encode_irq_set (bool relative, uint irq)
    Encode a IRQ SET instruction.

static uint pio_encode_irq_wait (bool relative, uint irq)
    Encode a IRQ WAIT instruction.

static uint pio_encode_irq_clear (bool relative, uint irq)
    Encode a IRQ CLEAR instruction.

static uint pio_encode_set (enum pio_src_dest dest, uint value)
    Encode a SET instruction.

static uint pio_encode_nop (void)
    Encode a NOP instruction.

Enumeration Type Documentation

pio_src_dest

enum pio_src_dest

Enumeration of values to pass for source/destination args for instruction encoding functions.

︎ NOTE

Not all values are suitable for all functions. Validity is only checked in debug mode when PARAM_ASSERTIONS_ENABLEDPIO_INSTRUCTIONS is 1

Function Documentation

pio_encode_delay

static uint pio_encode_delay (uint cycles) [inline], [static]

Encode just the delay slot bits of an instruction.
NOTE

This function does not return a valid instruction encoding; instead it returns an encoding of the delay slot suitable for `OR`ing with the result of an encoding function for an actual instruction. Care should be taken when combining the results of this function with the results of `pio_encode_sideset` and `pio_encode_sideset_opt` as they share the same bits within the instruction encoding.

Parameters

cycles the number of cycles 0-31 (or less if side set is being used)

Returns
the delay slot bits to be ORed with an instruction encoding

pio_encode_in

static uint pio_encode_in (enum pio_src_dest src, uint count) [inline], [static]

Encode an IN instruction.

Parameters

src The source to take data from
count The number of bits 1-32

Returns
The instruction encoding with 0 delay and no side set value

See also
`pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt`

pio_encode_irq_clear

static uint pio_encode_irq_clear (bool relative, uint irq) [inline], [static]

Encode an IRQ CLEAR instruction.

Parameters

relative true for a IRQ CLEAR <irq> REL, false for regular IRQ CLEAR <irq>
irq the irq number 0-7

Returns
The instruction encoding with 0 delay and no side set value

See also
`pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt`

pio_encode_irq_set

static uint pio_encode_irq_set (bool relative, uint irq) [inline], [static]

Encode an IRQ SET instruction.

Parameters

relative true for a IRQ SET <irq> REL, false for regular IRQ SET <irq>
irq the irq number 0-7
Returns
The instruction encoding with 0 delay and no side set value

See also
pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_irq_wait
static uint pio_encode_irq_wait (bool relative, uint irq) [inline], [static]
Encode a IRQ WAIT instruction.
This is the equivalent of IRQ WAIT <irq> <relative>

Parameters
  relative  true for a IRQ WAIT <irq> REL, false for regular IRQ WAIT <irq>
  irq      the irq number 0-7

Returns
The instruction encoding with 0 delay and no side set value

See also
pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_jmp
static uint pio_encode_jmp (uint addr) [inline], [static]
Encode an unconditional JMP instruction.
This is the equivalent of JMP <addr>

Parameters
  addr     The target address 0-31 (an absolute address within the PIO instruction memory)

Returns
The instruction encoding with 0 delay and no side set value

See also
pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_jmp_not_osre
static uint pio_encode_jmp_not_osre (uint addr) [inline], [static]
Encode a conditional JMP if output shift register not empty instruction.
This is the equivalent of JMP !OSRE <addr>

Parameters
  addr     The target address 0-31 (an absolute address within the PIO instruction memory)

Returns
The instruction encoding with 0 delay and no side set value

See also
pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

pio_encode_jmp_not_x
static uint pio_encode_jmp_not_x (uint addr) [inline], [static]
Encode a conditional JMP if scratch X zero instruction.
This is the equivalent of `JMP IX <addr>`

**Parameters**

- `addr` The target address 0-31 (an absolute address within the PIO instruction memory)

**Returns**

The instruction encoding with 0 delay and no side set value

**See also**

`pio_encode_delay`, `pio_encode_sideset`, `pio_encode_sideset_opt`

`pio_encode_jmp_not_y`

```
static uint pio_encode_jmp_not_y (uint addr) [inline], [static]
```

Encode a conditional JMP if scratch Y zero instruction.

This is the equivalent of `JMP !Y <addr>`

**Parameters**

- `addr` The target address 0-31 (an absolute address within the PIO instruction memory)

**Returns**

The instruction encoding with 0 delay and no side set value

**See also**

`pio_encode_delay`, `pio_encode_sideset`, `pio_encode_sideset_opt`

`pio_encode_jmp_pin`

```
static uint pio_encode_jmp_pin (uint addr) [inline], [static]
```

Encode a conditional JMP if input pin high instruction.

This is the equivalent of `JMP PIN <addr>`

**Parameters**

- `addr` The target address 0-31 (an absolute address within the PIO instruction memory)

**Returns**

The instruction encoding with 0 delay and no side set value

**See also**

`pio_encode_delay`, `pio_encode_sideset`, `pio_encode_sideset_opt`

`pio_encode_jmp_x_dec`

```
static uint pio_encode_jmp_x_dec (uint addr) [inline], [static]
```

Encode a conditional JMP if scratch X non-zero (and post-decrement X) instruction.

This is the equivalent of `JMP X-- <addr>`

**Parameters**

- `addr` The target address 0-31 (an absolute address within the PIO instruction memory)

**Returns**

The instruction encoding with 0 delay and no side set value

**See also**

`pio_encode_delay`, `pio_encode_sideset`, `pio_encode_sideset_opt`

`pio_encode_jmp_x_ne_y`

Raspberry Pi Pico C++ SDK

4.1. Hardware APIs
static uint pio_encode_jmp_x_ne_y (uint addr) [inline], [static]

Encode a conditional JMP if scratch X not equal scratch Y instruction.

This is the equivalent of \texttt{JMP X!=Y <addr>}

Parameters

\begin{itemize}
\item \texttt{addr} \hspace{1cm} The target address 0-31 (an absolute address within the PIO instruction memory)
\end{itemize}

Returns

The instruction encoding with 0 delay and no side set value

See also

\texttt{pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt}

\texttt{pio_encode_jmp_y_dec}

static uint pio_encode_jmp_y_dec (uint addr) [inline], [static]

Encode a conditional JMP if scratch Y non-zero (and post-decrement Y) instruction.

This is the equivalent of \texttt{JMP Y-- <addr>}

Parameters

\begin{itemize}
\item \texttt{addr} \hspace{1cm} The target address 0-31 (an absolute address within the PIO instruction memory)
\end{itemize}

Returns

The instruction encoding with 0 delay and no side set value

See also

\texttt{pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt}

\texttt{pio_encode_mov}

static uint pio_encode_mov (enum pio_src_dest dest, enum pio_src_dest src) [inline], [static]

Encode a MOV instruction.

This is the equivalent of \texttt{MOV <dest>, <src>}

Parameters

\begin{itemize}
\item \texttt{dest} \hspace{1cm} The destination to write data to
\item \texttt{src} \hspace{1cm} The source to take data from
\end{itemize}

Returns

The instruction encoding with 0 delay and no side set value

See also

\texttt{pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt}

\texttt{pio_encode_mov_not}

static uint pio_encode_mov_not (enum pio_src_dest dest, enum pio_src_dest src) [inline], [static]

Encode a MOV instruction with bit invert.

This is the equivalent of \texttt{MOV <dest>, ~<src>}

Parameters

\begin{itemize}
\item \texttt{dest} \hspace{1cm} The destination to write inverted data to
\item \texttt{src} \hspace{1cm} The source to take data from
\end{itemize}

Returns
The instruction encoding with 0 delay and no side set value

**See also**

`pio_encode_delay`, `pio_encode_sideset`, `pio_encode_sideset_opt`

**pio_encode_mov_reverse**

```c
static uint pio_encode_mov_reverse (enum pio_src_dest dest, enum pio_src_dest src) [inline], [static]
```

Encode a MOV instruction with bit reverse.

This is the equivalent of `MOV <dest>, ::<src>`

**Parameters**

- **dest**  
  The destination to write bit reversed data to

- **src**  
  The source to take data from

**Returns**

The instruction encoding with 0 delay and no side set value

**See also**

`pio_encode_delay`, `pio_encode_sideset`, `pio_encode_sideset_opt`

**pio_encode_nop**

```c
static uint pio_encode_nop (void) [inline], [static]
```

Encode a NOP instruction.

This is the equivalent of `NOP` which is itself encoded as `MOV y, y`

**Returns**

The instruction encoding with 0 delay and no side set value

**See also**

`pio_encode_delay`, `pio_encode_sideset`, `pio_encode_sideset_opt`

**pio_encode_out**

```c
static uint pio_encode_out (enum pio_src_dest dest, uint count) [inline], [static]
```

Encode an OUT instruction.

This is the equivalent of `OUT <src>, <count>`

**Parameters**

- **dest**  
  The destination to write data to

- **count**  
  The number of bits 1-32

**Returns**

The instruction encoding with 0 delay and no side set value

**See also**

`pio_encode_delay`, `pio_encode_sideset`, `pio_encode_sideset_opt`

**pio_encode_pull**

```c
static uint pio_encode_pull (bool if_empty, bool block) [inline], [static]
```

Encode a PULL instruction.

This is the equivalent of `PULL <if_empty>, <block>`

**Parameters**

4.1. Hardware APIs
### pio_encode_push

`static uint pio_encode_push (bool if_full, bool block) [inline], [static]`

Encode a PUSH instruction. This is the equivalent of `PUSH <if_full>, <block>`

**Parameters**

- `if_full` true for `PUSH IF_FULL ...`, false for `PUSH ...`
- `block` true for `PUSH ... BLOCK`, false for `PUSH ...`

**Returns**

The instruction encoding with 0 delay and no side set value

**See also**

- `pio_encode_delay`
- `pio_encode_sideset`
- `pio_encode_sideset_opt`

### pio_encode_set

`static uint pio_encode_set (enum pio_src_dest dest, uint value) [inline], [static]`

Encode a SET instruction. This is the equivalent of `SET <dest>, <value>`

**Parameters**

- `dest` The destination to apply the value to
- `value` The value 0-31

**Returns**

The instruction encoding with 0 delay and no side set value

**See also**

- `pio_encode_delay`
- `pio_encode_sideset`
- `pio_encode_sideset_opt`

### pio_encode_sideset

`static uint pio_encode_sideset (uint sideset_bit_count, uint value) [inline], [static]`

Encode just the side set bits of an instruction (in non optional side set mode)
### NOTE

This function does not return a valid instruction encoding; instead it returns an encoding of the side set bits suitable for `OR`ing with the result of an encoding function for an actual instruction. Care should be taken when combining the results of this function with the results of `pio_encode_delay` as they share the same bits within the instruction encoding.

**Parameters**

- `sideset_bit_count`: number of side set bits as would be specified via `.sideset` in pioasm
- `value`: the value to sideset on the pins

**Returns**

the side set bits to be ORed with an instruction encoding

### `pio_encode_sideset_opt`

```c
static uint pio_encode_sideset_opt (uint sideset_bit_count, uint value) [inline], [static]
```

Encode just the side set bits of an instruction (in optional `opt` side set mode)

---

### NOTE

This function does not return a valid instruction encoding; instead it returns an encoding of the side set bits suitable for `OR`ing with the result of an encoding function for an actual instruction. Care should be taken when combining the results of this function with the results of `pio_encode_delay` as they share the same bits within the instruction encoding.

**Parameters**

- `sideset_bit_count`: number of side set bits as would be specified via `.sideset <n> opt` in pioasm
- `value`: the value to sideset on the pins

**Returns**

the side set bits to be ORed with an instruction encoding

### `pio_encode_wait_gpio`

```c
static uint pio_encode_wait_gpio (bool polarity, uint gpio) [inline], [static]
```

Encode a WAIT for GPIO pin instruction.

This is the equivalent of `WAIT <polarity> GPIO <gpio>`

**Parameters**

- `polarity`: true for `WAIT 1`, false for `WAIT 0`
- `gpio`: The real GPIO number 0-31

**Returns**

The instruction encoding with 0 delay and no side set value

**See also**

- `pio_encode_delay`, `pio_encode_sideset`, `pio_encode_sideset_opt`

### `pio_encode_wait_irq`

```c
static uint pio_encode_wait_irq (bool polarity, bool relative, uint irq) [inline], [static]
```

Encode a WAIT for IRQ instruction.

This is the equivalent of `WAIT <polarity> IRQ <irq> <relative>`
Parameters

- **polarity** : true for `WAIT 1`, false for `WAIT 0`
- **relative** : true for a `WAIT IRQ <irq> REL`, false for regular `WAIT IRQ <irq>`
- **irq** : the irq number 0-7

Returns

The instruction encoding with 0 delay and no side set value

See also

- `pio_encode_delay`, `pio_encode_sideset`, `pio_encode_sideset_opt`

```c
static uint pio_encode_wait_pin (bool polarity, uint pin) [inline], [static]
```

Encode a WAIT for pin instruction.

This is the equivalent of `WAIT <polarity> PIN <pin>`

Parameters

- **polarity** : true for `WAIT 1`, false for `WAIT 0`
- **pin** : The pin number 0-31 relative to the executing SM’s input pin mapping

Returns

The instruction encoding with 0 delay and no side set value

See also

- `pio_encode_delay`, `pio_encode_sideset`, `pio_encode_sideset_opt`

### 4.1.14. hardware_pll

#### 4.1.14.1. Detailed Description

Phase Locked Loop control APIs

There are two PLLs in RP2040. They are:

- **pll_sys** - Used to generate up to a 133MHz system clock
- **pll_usb** - Used to generate a 48MHz USB reference clock

For details on how the PLLs are calculated, please refer to the RP2040 datasheet.

#### 4.1.14.2. Functions

```c
void pll_init (PLL pll, uint ref_div, uint vco_freq, uint post_div1, uint post_div2)
```

Initialise specified PLL.

```c
void pll_deinit (PLL pll)
```

Release/uninitialise specified PLL.

#### 4.1.14.3. Function Documentation
4.1.14.3.1. pll_deinit

```c
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

4.1.14.3.2. pll_init

```c
void pll_init (PLL pll, uint ref_div, uint vco_freq, uint post_div1, uint post_div2)
```

Initialise specified PLL.

**Parameters**

- `pll`: PLL_sys or PLL_usb
- `ref_div`: Input clock divider.
- `vco_freq`: Requested output from the VCO (voltage controlled oscillator)
- `post_div1`: Post Divider 1 - range 1-7. Must be >= post_div2
- `post_div2`: Post Divider 2 - range 1-7

4.1.15. hardware_pwm

4.1.15.1. Detailed Description

Hardware Pulse Width Modulation (PWM) API

The RP2040 PWM block has 8 identical slices. Each slice can drive two PWM output signals, or measure the frequency or duty cycle of an input signal. This gives a total of up to 16 controllable PWM outputs. All 30 GPIOs can be driven by the PWM block.

The PWM hardware functions by continuously comparing the input value to a free-running counter. This produces a toggling output where the amount of time spent at the high output level is proportional to the input value. The fraction of time spent at the high signal level is known as the duty cycle of the signal.

The default behaviour of a PWM slice is to count upward until the wrap value (`pwm_config_set_wrap`) is reached, and then immediately wrap to 0. PWM slices also offer a phase-correct mode, where the counter starts to count downward after reaching TOP, until it reaches 0 again.

**Example**

```c
#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);

    // Output PWM signals on pins 0 and 1
    ...
```

Raspberry Pi Pico C/C++ SDK 4.1. Hardware APIs
// 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.

4.1.15.2. Enumerations

enum pwm_clkdiv_mode { PWM_DIV_FREE_RUNNING = 0, PWM_DIV_B_HIGH = 1, PWM_DIV_B_RISING = 2, PWM_DIV_B_FALLING = 3 }

PWM Divider mode settings.

4.1.15.3. Functions

static uint pwm_gpio_to_slice_num (uint gpio)
    Determine the PWM slice that is attached to the specified GPIO.

static uint pwm_gpio_to_channel (uint gpio)
    Determine the PWM channel that is attached to the specified GPIO.

static void pwm_config_set_phase_correct (pwm_config *c, bool phase_correct)
    Set phase correction in a PWM configuration.

static void pwm_config_set_clkdiv (pwm_config *c, float div)
    Set PWM clock divider in a PWM configuration.

static void pwm_config_set_clkdiv_int_frac (pwm_config *c, uint32_t integer, uint32_t fract)
    Set PWM clock divider in a PWM configuration using an 8:4 fractional value.

static void pwm_config_set_clkdiv_int (pwm_config *c, uint div)
    Set PWM clock divider in a PWM configuration.

static void pwm_config_set_clkdiv_mode (pwm_config *c, enum pwm_clkdiv_mode mode)
    Set PWM counting mode in a PWM configuration.

static void pwm_config_set_output_polarity (pwm_config *c, bool a, bool b)
    Set output polarity in a PWM configuration.

static void pwm_config_set_wrap (pwm_config *c, uint16_t wrap)
    Set PWM counter wrap value in a PWM configuration.

static void pwm_init (uint slice_num, pwm_config *c, bool start)
    Initialise a PWM with settings from a configuration object.

static pwm_config pwm_get_default_config (void)
    Get a set of default values for PWM configuration.
static void pwm_set_wrap (uint slice_num, uint16_t wrap)
    
    Set the current PWM counter wrap value.

static void pwm_set_chan_level (uint slice_num, uint chan, uint16_t level)
    
    Set the current PWM counter compare value for one channel.

static void pwm_set_both_levels (uint slice_num, uint16_t level_a, uint16_t level_b)
    
    Set PWM counter compare values.

static void pwm_set_gpio_level (uint gpio, uint16_t level)
    
    Helper function to set the PWM level for the slice and channel associated with a GPIO.

static uint16_t pwm_get_counter (uint slice_num)
    
    Get PWM counter.

static void pwm_set_counter (uint slice_num, uint16_t c)
    
    Set PWM counter.

static void pwm_advance_count (uint slice_num)
    
    Advance PWM count.

static void pwm_retard_count (uint slice_num)
    
    Retard PWM count.

static void pwm_set_clkdiv_int_frac (uint slice_num, uint8_t integer, uint8_t fract)
    
    Set PWM clock divider using an 8:4 fractional value.

static void pwm_set_clkdiv (uint slice_num, float divider)
    
    Set PWM clock divider.

static void pwm_set_output_polarity (uint slice_num, bool a, bool b)
    
    Set PWM output polarity.

static void pwm_set_clkdiv_mode (uint slice_num, enum pwm_clkdiv_mode mode)
    
    Set PWM divider mode.

static void pwm_set_phase_correct (uint slice_num, bool phase_correct)
    
    Set PWM phase correct on/off.

static void pwm_set_enabled (uint slice_num, bool enabled)
    
    Enable/Disable PWM.

static void pwm_set_mask_enabled (uint32_t mask)
    
    Enable/Disable multiple PWM slices simultaneously.

static void pwm_set_irq_enabled (uint slice_num, bool enabled)
    
    Enable PWM instance interrupt.

static void pwm_set_irq_mask_enabled (uint32_t slice_mask, bool enabled)
    
    Enable multiple PWM instance interrupts.

static void pwm_clear_irq (uint slice_num)
    
    Clear a single PWM channel interrupt.

static uint32_t pwm_get_irq_status_mask (void)
    
    Get PWM interrupt status, raw.

static void pwm_force_irq (uint slice_num)
    
    Force PWM interrupt.
static uint pwm_get_dreq (uint slice_num)
    Return the DREQ to use for pacing transfers to a particular PWM slice.

4.1.15.4. Enumeration Type Documentation

4.1.15.4.1. pwm_clkdiv_mode

enum pwm_clkdiv_mode
    PWM Divider mode settings.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWM_DIV_FREE_RUNNING</td>
<td>Free-running counting at rate dictated by fractional divider.</td>
</tr>
<tr>
<td>PWM_DIV_B_HIGH</td>
<td>Fractional divider is gated by the PWM B pin.</td>
</tr>
<tr>
<td>PWM_DIV_B_RISING</td>
<td>Fractional divider advances with each rising edge of the PWM B pin.</td>
</tr>
<tr>
<td>PWM_DIV_B_FALLING</td>
<td>Fractional divider advances with each falling edge of the PWM B pin.</td>
</tr>
</tbody>
</table>

4.1.15.5. Function Documentation

4.1.15.5.1. pwm_advance_count

static void pwm_advance_count (uint slice_num) [inline], [static]
    Advance PWM count.
    Advance the phase of a running the counter by 1 count.
    This function will return once the increment is complete.

Parameters
    slice_num  PWM slice number

4.1.15.5.2. pwm_clear_irq

static void pwm_clear_irq (uint slice_num) [inline], [static]
    Clear a single PWM channel interrupt.

Parameters
    slice_num  PWM slice number

4.1.15.5.3. pwm_config_set_clkdiv

static void pwm_config_set_clkdiv (pwm_config * c, float div) [inline], [static]
    Set PWM clock divider in a PWM configuration.

Parameters
    c        PWM configuration struct to modify
    div      Value to divide counting rate by. Must be greater than or equal to 1.
If the divide mode is free-running, the PWM counter runs at clk_sys / div. Otherwise, the divider reduces the rate of events seen on the B pin input (level or edge) before passing them on to the PWM counter.

4.1.15.5.4. pwm_config_set_clkdiv_int

static void pwm_config_set_clkdiv_int (pwm_config * c, uint div) [inline], [static]
Set PWM clock divider in a PWM configuration.

Parameters
- c PWM configuration struct to modify
- div Integer value to reduce counting rate by. Must be greater than or equal to 1.

If the divide mode is free-running, the PWM counter runs at clk_sys / div. Otherwise, the divider reduces the rate of events seen on the B pin input (level or edge) before passing them on to the PWM counter.

4.1.15.5.5. pwm_config_set_clkdiv_int_frac

static void pwm_config_set_clkdiv_int_frac (pwm_config * c, uint8_t integer, uint8_t fract) [inline], [static]
Set PWM clock divider in a PWM configuration using an 8:4 fractional value.

Parameters
- c PWM configuration struct to modify
- integer 8 bit integer part of the clock divider. Must be greater than or equal to 1.
- fract 4 bit fractional part of the clock divider

If the divide mode is free-running, the PWM counter runs at clk_sys / div. Otherwise, the divider reduces the rate of events seen on the B pin input (level or edge) before passing them on to the PWM counter.

4.1.15.5.6. pwm_config_set_clkdiv_mode

static void pwm_config_set_clkdiv_mode (pwm_config * c, enum pwm_clkdiv_mode mode) [inline], [static]
Set PWM counting mode in a PWM configuration.

Parameters
- c PWM configuration struct to modify
- mode PWM divide/count mode

Configure which event gates the operation of the fractional divider. The default is always-on (free-running PWM). Can also be configured to count on high level, rising edge or falling edge of the B pin input.

4.1.15.5.7. pwm_config_set_output_polarity

static void pwm_config_set_output_polarity (pwm_config * c, bool a, bool b) [inline], [static]
Set output polarity in a PWM configuration.

Parameters
- c PWM configuration struct to modify
- a true to invert output A
- b true to invert output B
4.1.15.5.8. pwm_config_set_phase_correct

```
static void pwm_config_set_phase_correct (pwm_config * c, bool phase_correct) [inline], [static]
```

Set phase correction in a PWM configuration.

**Parameters**

- `c` : PWM configuration struct to modify
- `phase_correct` : true to set phase correct modulation, false to set trailing edge

Setting phase control to true means that instead of wrapping back to zero when the wrap point is reached, the PWM starts counting back down. The output frequency is halved when phase-correct mode is enabled.

4.1.15.5.9. pwm_config_set_wrap

```
static void pwm_config_set_wrap (pwm_config * c, uint16_t wrap) [inline], [static]
```

Set PWM counter wrap value in a PWM configuration.

Set the highest value the counter will reach before returning to 0. Also known as TOP.

**Parameters**

- `c` : PWM configuration struct to modify
- `wrap` : Value to set wrap to

4.1.15.5.10. pwm_force_irq

```
static void pwm_force_irq (uint slice_num) [inline], [static]
```

Force PWM interrupt.

**Parameters**

- `slice_num` : PWM slice number

4.1.15.5.11. pwm_get_counter

```
static uint16_t pwm_get_counter (uint slice_num) [inline], [static]
```

Get PWM counter.

Get current value of PWM counter.

**Parameters**

- `slice_num` : PWM slice number

**Returns**

Current value of the PWM counter

4.1.15.5.12. pwm_get_default_config

```
static pwm_config pwm_get_default_config (void) [inline], [static]
```

Get a set of default values for PWM configuration.

PWM config is free-running at system clock speed, no phase correction, wrapping at 0xffff, with standard polarities for channels A and B.

**Returns**
Set of default values.

4.1.15.5.13. pwm_get_dreq

static uint pwm_get_dreq (uint slice_num) [inline], [static]
Return the DREQ to use for pacing transfers to a particular PWM slice.

Parameters
    slice_num  PWM slice number

4.1.15.5.14. pwm_get_irq_status_mask

static uint32_t pwm_get_irq_status_mask (void) [inline], [static]
Get PWM interrupt status, raw.

Returns
Bitmask of all PWM interrupts currently set

4.1.15.5.15. pwm_gpio_to_channel

static uint pwm_gpio_to_channel (uint gpio) [inline], [static]
Determine the PWM channel that is attached to the specified GPIO.
Each slice 0 to 7 has two channels, A and B.

Returns
The PWM channel that controls the specified GPIO.

4.1.15.5.16. pwm_gpio_to_slice_num

static uint pwm_gpio_to_slice_num (uint gpio) [inline], [static]
Determine the PWM slice that is attached to the specified GPIO.

Returns
The PWM slice number that controls the specified GPIO.

4.1.15.5.17. pwm_init

static void pwm_init (uint slice_num, pwm_config * c, bool start) [inline], [static]
Initialise a PWM with settings from a configuration object.
Use the pwm_get_default_config() function to initialise a config structure, make changes as needed using the
pwm_config_* functions, then call this function to set up the PWM.

Parameters
    slice_num       PWM slice number
    c               The configuration to use
    start           If true the PWM will be started running once configured. If false you will need to start manually using pwm_set_enabled() or pwm_set_mask_enabled()
4.1.15.5.18. pwm_retard_count

static void pwm_retard_count (uint slice_num) [inline], [static]
Retard PWM count.
Retard the phase of a running counter by 1 count
This function will return once the retardation is complete.

Parameters
slice_num PWM slice number

4.1.15.5.19. pwm_set_both_levels

static void pwm_set_both_levels (uint slice_num, uint16_t level_a, uint16_t level_b) [inline], [static]
Set PWM counter compare values.
Set the value of the PWM counter compare values, A and B.
The counter compare register is double-buffered in hardware. This means that, when the PWM is running, a write to the
counter compare values does not take effect until the next time the PWM slice wraps (or, in phase-correct mode, the
next time the slice reaches 0). If the PWM is not running, the write is latched in immediately.

Parameters
slice_num PWM slice number
level_a Value to set compare A to. When the counter reaches this value the A output is deasserted
level_b Value to set compare B to. When the counter reaches this value the B output is deasserted

4.1.15.5.20. pwm_set_chan_level

static void pwm_set_chan_level (uint slice_num, uint chan, uint16_t level) [inline], [static]
Set the current PWM counter compare value for one channel.
Set the value of the PWM counter compare value, for either channel A or channel B.
The counter compare register is double-buffered in hardware. This means that, when the PWM is running, a write to the
counter compare values does not take effect until the next time the PWM slice wraps (or, in phase-correct mode, the
next time the slice reaches 0). If the PWM is not running, the write is latched in immediately.

Parameters
slice_num PWM slice number
chan Which channel to update. 0 for A, 1 for B.
level new level for the selected output

4.1.15.5.21. pwm_set_clkdiv

static void pwm_set_clkdiv (uint slice_num, float divider) [inline], [static]
Set PWM clock divider.
Set the clock divider. Counter increment will be on sysclock divided by this value, taking into account the gating.

Parameters
slice_num PWM slice number
4.1.15.5.22. `pwm_set_clkdiv_int_frac`

```c
static void pwm_set_clkdiv_int_frac (uint slice_num, uint8_t integer, uint8_t fract) [inline], [static]
```

Set PWM clock divider using an 8:4 fractional value.

Set the clock divider. Counter increment will be on sysclock divided by this value, taking into account the gating.

**Parameters**

- `slice_num` - PWM slice number
- `integer` - 8 bit integer part of the clock divider
- `fract` - 4 bit fractional part of the clock divider

---

4.1.15.5.23. `pwm_set_clkdiv_mode`

```c
static void pwm_set_clkdiv_mode (uint slice_num, enum pwm_clkdiv_mode mode) [inline], [static]
```

Set PWM divider mode.

**Parameters**

- `slice_num` - PWM slice number
- `mode` - Required divider mode

---

4.1.15.5.24. `pwm_set_counter`

```c
static void pwm_set_counter (uint slice_num, uint16_t c) [inline], [static]
```

Set PWM counter.

Set the value of the PWM counter

**Parameters**

- `slice_num` - PWM slice number
- `c` - Value to set the PWM counter to

---

4.1.15.5.25. `pwm_set_enabled`

```c
static void pwm_set_enabled (uint slice_num, bool enabled) [inline], [static]
```

Enable/Disable PWM.

When a PWM is disabled, it halts its counter, and the output pins are left high or low depending on exactly when the counter is halted. When re-enabled the PWM resumes immediately from where it left off.

If the PWM's output pins need to be low when halted:

- The counter compare can be set to zero whilst the PWM is enabled, and then the PWM disabled once both pins are seen to be low
- The GPIO output overrides can be used to force the actual pins low
- The PWM can be run for one cycle (i.e. enabled then immediately disabled) with a TOP of 0, count of 0 and counter compare of 0, to force the pins low when the PWM has already been halted. The same method can be used with a counter compare value of 1 to force a pin high.

Note that, when disabled, the PWM can still be advanced one count at a time by pulsing the PH_ADV bit in its CSR.
output pins transition as though the PWM were enabled.

**Parameters**

- `slice_num` - PWM slice number
- `enabled` - true to enable the specified PWM, false to disable.

### 4.1.15.5.26. `pwm_set_gpio_level`

```c
static void pwm_set_gpio_level (uint gpio, uint16_t level) [inline], [static]
```

Helper function to set the PWM level for the slice and channel associated with a GPIO.

Look up the correct slice (0 to 7) and channel (A or B) for a given GPIO, and update the corresponding counter compare field.

This PWM slice should already have been configured and set running. Also be careful of multiple GPIOs mapping to the same slice and channel (if GPIOs have a difference of 16).

The counter compare register is double-buffered in hardware. This means that, when the PWM is running, a write to the counter compare values does not take effect until the next time the PWM slice wraps (or, in phase-correct mode, the next time the slice reaches 0). If the PWM is not running, the write is latched in immediately.

**Parameters**

- `gpio` - GPIO to set level of
- `level` - PWM level for this GPIO

### 4.1.15.5.27. `pwm_set_irq_enabled`

```c
static void pwm_set_irq_enabled (uint slice_num, bool enabled) [inline], [static]
```

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

### 4.1.15.5.28. `pwm_set_irq_mask_enabled`

```c
static void pwm_set_irq_mask_enabled (uint32_t slice_mask, bool enabled) [inline], [static]
```

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

### 4.1.15.5.29. `pwm_set_mask_enabled`

```c
static void pwm_set_mask_enabled (uint32_t mask) [inline], [static]
```

Enable/Disable multiple PWM slices simultaneously.
Parameters

mask Bitmap of PWMs to enable/disable. Bits 0 to 7 enable slices 0-7 respectively

4.1.15.5.30. pwm_set_output_polarity

static void pwm_set_output_polarity (uint slice_num, bool a, bool b) [inline], [static]

Set PWM output polarity.

Parameters

slice_num PWM slice number
a true to invert output A
b true to invert output B

4.1.15.5.31. pwm_set_phase_correct

static void pwm_set_phase_correct (uint slice_num, bool phase_correct) [inline], [static]

Set PWM phase correct on/off.

Parameters

slice_num PWM slice number
phase_correct true to set phase correct modulation, false to set trailing edge

Setting phase control to true means that instead of wrapping back to zero when the wrap point is reached, the PWM starts counting back down. The output frequency is halved when phase-correct mode is enabled.

4.1.15.5.32. pwm_set_wrap

static void pwm_set_wrap (uint slice_num, uint16_t wrap) [inline], [static]

Set the current PWM counter wrap value.

Set the highest value the counter will reach before returning to 0. Also known as TOP.

The counter wrap value is double-buffered in hardware. This means that, when the PWM is running, a write to the counter wrap value does not take effect until after the next time the PWM slice wraps (or, in phase-correct mode, the next time the slice reaches 0). If the PWM is not running, the write is latched in immediately.

Parameters

slice_num PWM slice number
wrap Value to set wrap to

4.1.16. hardware_resets

4.1.16.1. Detailed Description

Hardware Reset API

The reset controller allows software control of the resets to all of the peripherals that are not critical to boot the processor in the RP2040.

reset_bitmask
Multiple blocks are referred to using a bitmask as follows:

<table>
<thead>
<tr>
<th>Block to reset</th>
<th>Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB</td>
<td>24</td>
</tr>
<tr>
<td>UART 1</td>
<td>23</td>
</tr>
<tr>
<td>UART 0</td>
<td>22</td>
</tr>
<tr>
<td>Timer</td>
<td>21</td>
</tr>
<tr>
<td>TB Manager</td>
<td>20</td>
</tr>
<tr>
<td>Sysinfo</td>
<td>19</td>
</tr>
<tr>
<td>System Config</td>
<td>18</td>
</tr>
<tr>
<td>SPI 1</td>
<td>17</td>
</tr>
<tr>
<td>SPI 0</td>
<td>16</td>
</tr>
<tr>
<td>RTC</td>
<td>15</td>
</tr>
<tr>
<td>PWM</td>
<td>14</td>
</tr>
<tr>
<td>PLL USB</td>
<td>13</td>
</tr>
<tr>
<td>PLL System</td>
<td>12</td>
</tr>
<tr>
<td>PIO 1</td>
<td>11</td>
</tr>
<tr>
<td>PIO 0</td>
<td>10</td>
</tr>
<tr>
<td>Pads - QSPI</td>
<td>9</td>
</tr>
<tr>
<td>Pads - bank 0</td>
<td>8</td>
</tr>
<tr>
<td>JTAG</td>
<td>7</td>
</tr>
<tr>
<td>IO Bank 1</td>
<td>6</td>
</tr>
<tr>
<td>IO Bank 0</td>
<td>5</td>
</tr>
<tr>
<td>I2C 1</td>
<td>4</td>
</tr>
<tr>
<td>I2C 0</td>
<td>3</td>
</tr>
<tr>
<td>DMA</td>
<td>2</td>
</tr>
<tr>
<td>Bus Control</td>
<td>1</td>
</tr>
<tr>
<td>ADC 0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example

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

int main()
{
    stdio_init_all();
    printf("Hello, reset!
");
    // Put the PWM block into reset
    reset_block(RESETS_RESET_PWM_BITS);
    // And bring it out
}"
```
4.1.16.2. Functions

**static void reset_block (uint32_t bits)**
Reset the specified HW blocks.

**static void unreset_block (uint32_t bits)**
bring specified HW blocks out of reset

**static void unreset_block_wait (uint32_t bits)**
Bring specified HW blocks out of reset and wait for completion.

4.1.16.3. Function Documentation

4.1.16.3.1. reset_block

**static void reset_block (uint32_t bits) [inline], [static]**
Reset the specified HW blocks.

**Parameters**

- **bits** Bit pattern indicating blocks to reset. See reset_bitmask

4.1.16.3.2. unreset_block

**static void unreset_block (uint32_t bits) [inline], [static]**
bring specified HW blocks out of reset

**Parameters**

- **bits** Bit pattern indicating blocks to unreset. See reset_bitmask

4.1.16.3.3. unreset_block_wait

**static void unreset_block_wait (uint32_t bits) [inline], [static]**
Bring specified HW blocks out of reset and wait for completion.

**Parameters**

- **bits** Bit pattern indicating blocks to unreset. See reset_bitmask
4.1.17. hardware_rtc

4.1.17.1. Detailed Description

Hardware Real Time Clock API

The RTC keeps track of time in human readable format and generates events when the time is equal to a preset value. Think of a digital clock, not epoch time used by most computers. There are seven fields, one each for year (12 bit), month (4 bit), day (5 bit), day of the week (3 bit), hour (5 bit) minute (6 bit) and second (6 bit), storing the data in binary format.

See also
datetime_t

Example

```
#include <stdio.h>
#include "hardware/rtc.h"
#include "pico/stdlib.h"
#include "pico/util/datetime.h"

int main() {
    stdio_init_all();
    printf("Hello RTC!\n");

    char datetime_buf[256];
    char *datetime_str = &datetime_buf[0];

    // Start on Friday 5th of June 2020 15:45:00
    datetime_t t = {
        .year = 2020,
        .month = 06,
        .day = 05,
        .dotw = 5,     // 0 is Sunday, so 5 is Friday
        .hour = 15,
        .min = 45,
        .sec = 00
    };

    // Start the RTC
    rtc_init();
    rtc_set_datetime(&t);

    // clk_sys is >2000x faster than clk_rtc, so datetime is not updated immediately when rtc_get_datetime() is called.
    // the delay is up to 3 RTC clock cycles (which is 64us with the default clock settings)
    sleep_us(64);

    // Print the time
    while (true) {
        rtc_get_datetime(&t);
        datetime_to_str(datetime_str, sizeof(datetime_buf), &t);
        printf("\r\n", datetime_str);
        sleep_ms(100);
    }
```
4.1.17.2. Typedefs

typedef void(* rtc_callback_t)(void)

4.1.17.3. Functions

void rtc_init (void)
    Initialise the RTC system.

bool rtc_set_datetime (datetime_t *t)
    Set the RTC to the specified time.

bool rtc_get_datetime (datetime_t *t)
    Get the current time from the RTC.

bool rtc_running (void)
    Is the RTC running?

void rtc_set_alarm (datetime_t *t, rtc_callback_t user_callback)
    Set a time in the future for the RTC to call a user provided callback.

void rtc_enable_alarm (void)
    Enable the RTC alarm (if inactive)

void rtc_disable_alarm (void)
    Disable the RTC alarm (if active)

4.1.17.4. Typedef Documentation

4.1.17.4.1. rtc_callback_t

typedef void(* rtc_callback_t) (void)
    Callback function type for RTC alarms

See also
    rtc_set_alarm()

4.1.17.5. Function Documentation

4.1.17.5.1. rtc_disable_alarm

void rtc_disable_alarm (void)
    Disable the RTC alarm (if active)

4.1.17.5.2. rtc_enable_alarm

void rtc_enable_alarm (void)
    Enable the RTC alarm (if inactive)
4.1.17.5.3. rtc_get_datetime

bool rtc_get_datetime (datetime_t * t)

Get the current time from the RTC.

Parameters
   t   Pointer to a datetime_t structure to receive the current RTC time

Returns
true if datetime is valid, false if the RTC is not running.

4.1.17.5.4. rtc_init

void rtc_init (void)

Initialise the RTC system.

4.1.17.5.5. rtc_running

bool rtc_running (void)

Is the RTC running?

4.1.17.5.6. 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

4.1.17.5.7. rtc_set_datetime

bool rtc_set_datetime (datetime_t * t)

Set the RTC to the specified time.

NOTE
Note that after setting the RTC date and time, a subsequent read of the values (e.g. via rtc_get_datetime()) may not reflect the new setting until up to three cycles of the potentially-much-slower RTC clock domain have passed. This represents a period of 64 microseconds with the default RTC clock configuration.

Parameters
   t   Pointer to a datetime_t structure contains time to set

Returns
true if set, false if the passed in datetime was invalid.
4.1.18. hardware_spi

4.1.18.1. Detailed Description

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.

4.1.18.2. Macros

- ```c
#define spi0 ((spi_inst_t *)spi0_hw)
```
- ```c
#define spi1 ((spi_inst_t *)spi1_hw)
```

4.1.18.3. Enumerations

```c
enum spi_cpha_t { SPI_CPHA_0 = 0, SPI_CPHA_1 = 1 }
```

Enumeration of SPI CPHA (clock phase) values.

```c
enum spi_cpol_t { SPI_CPOL_0 = 0, SPI_CPOL_1 = 1 }
```

Enumeration of SPI CPOL (clock polarity) values.

```c
enum spi_order_t { SPI_LSB_FIRST = 0, SPI_MSB_FIRST = 1 }
```

Enumeration of SPI bit-order values.

4.1.18.4. Functions

```c
uint spi_init (spi_inst_t *spi, uint baudrate)
```
Initialise SPI instances

```c
void spi_deinit (spi_inst_t *spi):
```
Deinitialise SPI instances

Puts the SPI into a known state, and enable it. Must be called before other functions. void `spi_deinit (spi_inst_t *spi)`: Deinitialise SPI instances

```c
static uint spi_get_index (const spi_inst_t *spi):
```
Convert SPI instance to hardware instance number. static void `spi_set_format (spi_inst_t *spi, uint data_bits, spi_cpol_t cpol, spi_cpha_t cpha, __unused spi_order_t order)`: Configure SPI. static void `spi_set_slave (spi_inst_t *spi, bool slave)`: Set SPI master/slave. static bool `spi_is_writable (const spi_inst_t *spi)`: Check whether a write can be done on SPI device. static bool `spi_is_readable (const spi_inst_t *spi)`: Check whether a read can be done on SPI device. static bool `spi_is_busy (const spi_inst_t *spi)`: Check whether SPI is busy.

```c
int spi_write_read_blocking (spi_inst_t *spi, const uint8_t *src, uint8_t *dst, size_t len):
```
Write/Read to/from an SPI device. int `spi_write_blocking (spi_inst_t *spi, const uint8_t *src, size_t len)`: Write to an SPI device, blocking. int `spi_read_blocking (spi_inst_t *spi, uint8_t repeated_tx_data, uint8_t *dst, size_t len)`: Read from an SPI device. int `spi_write16_blocking (spi_inst_t *spi, const uint16_t *src, size_t len)`: Write to an SPI device. int `spi_read16_blocking (spi_inst_t *spi, uint16_t repeated_tx_data, uint16_t *dst, size_t len)`: Read from an SPI device. static `uint spi_get_dreq (spi_inst_t *spi, bool is_tx)`: Return the DREQ to use for pacing transfers to/from a particular SPI instance.
4.1.18.5. Macro Definition Documentation

4.1.18.5.1. spi0

#define spi0 ((spi_inst_t *)spi0_hw)
Identifier for the first (SPI 0) hardware SPI instance (for use in SPI functions).
e.g. spi_init(spi0, 48000)

4.1.18.5.2. spi1

#define spi1 ((spi_inst_t *)spi1_hw)
Identifier for the second (SPI 1) hardware SPI instance (for use in SPI functions).
e.g. spi_init(spi1, 48000)

4.1.18.6. Enumeration Type Documentation

4.1.18.6.1. spi_cpha_t

enum spi_cpha_t
Enumeration of SPI CPHA (clock phase) values.

4.1.18.6.2. spi_cpol_t

enum spi_cpol_t
Enumeration of SPI CPOL (clock polarity) values.

4.1.18.6.3. spi_order_t

enum spi_order_t
Enumeration of SPI bit-order values.

4.1.18.7. Function Documentation

4.1.18.7.1. spi_deinit

void spi_deinit (spi_inst_t * spi)
Deinitialise SPI instances
Puts the SPI into a disabled state. Init will need to be called to reenable the device functions.
Parameters

spi SPI instance specifier, either spi0 or spi1
### 4.1.18.7.2. spi_get_baudrate

```c
uint spi_get_baudrate (const spi_inst_t * spi)
```

Get SPI baudrate.

Get SPI baudrate which was set by

See also

`spi_set_baudrate`

**Parameters**

- `spi`  
  SPI instance specifier, either `spi0` or `spi1`

**Returns**

The actual baudrate set

### 4.1.18.7.3. spi_get_dreq

```c
static uint spi_get_dreq (spi_inst_t * spi, bool is_tx) [inline], [static]
```

Return the DREQ to use for pacing transfers to/from a particular SPI instance.

**Parameters**

- `spi`  
  SPI instance specifier, either `spi0` or `spi1`

- `is_tx`  
  true for sending data to the SPI instance, false for receiving data from the SPI instance

### 4.1.18.7.4. spi_get_index

```c
static uint spi_get_index (const spi_inst_t * spi) [inline], [static]
```

Convert SPI instance to hardware instance number.

**Parameters**

- `spi`  
  SPI instance

**Returns**

Number of SPI, 0 or 1.

### 4.1.18.7.5. spi_init

```c
uint spi_init (spi_inst_t * spi, uint baudrate)
```

Initialise SPI instances

Put the SPI into a known state, and enable it. Must be called before other functions.

**NOTE**

There is no guarantee that the baudrate requested can be achieved exactly; the nearest will be chosen and returned

**Parameters**

- `spi`  
  SPI instance specifier, either `spi0` or `spi1`

- `baudrate`  
  Baudrate requested in Hz

**Returns**

Raspberry Pi Pico C/C++ SDK

4.1. Hardware APIs
the actual baud rate set

4.1.18.7.6. spi_is_busy

static bool spi_is_busy (const spi_inst_t * spi) [inline], [static]
Check whether SPI is busy.

Parameters

spi SPI instance specifier, either spi0 or spi1

Returns

true if SPI is busy

4.1.18.7.7. spi_is_readable

static bool spi_is_readable (const spi_inst_t * spi) [inline], [static]
Check whether a read can be done on SPI device.

Parameters

spi SPI instance specifier, either spi0 or spi1

Returns

true if a read is possible i.e. data is present

4.1.18.7.8. spi_is_writable

static bool spi_is_writable (const spi_inst_t * spi) [inline], [static]
Check whether a write can be done on SPI device.

Parameters

spi SPI instance specifier, either spi0 or spi1

Returns

false if no space is available to write. True if a write is possible

4.1.18.7.9. spi_read16_blocking

int spi_read16_blocking (spi_inst_t * spi, uint16_t repeated_tx_data, uint16_t * dst, size_t len)
Read from an SPI device.

Read len halfwords from SPI to dst. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate. repeated_tx_data is output repeatedly on TX as data is read in from RX. Generally this can be 0, but some devices require a specific value here, e.g. SD cards expect 0xff
NOTE

SPI should be initialised with 16 data_bits using `spi_set_format` first, otherwise this function will only read 8 data_bits.

Parameters

- `spi`  
  SPI instance specifier, either `spi0` or `spi1`
- `repeated_tx_data`  
  Buffer of data to write
- `dst`  
  Buffer for read data
- `len`  
  Length of buffer `dst` in halfwords

Returns

Number of halfwords written/read

4.1.18.7.10. spi_read_blocking

```c
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

4.1.18.7.11. spi_set_baudrate

```c
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
4.1.18.7.12. spi_set_format

```c
static void spi_set_format (spi_inst_t * spi, uint data_bits, spi_cpol_t cpol, spi_cpha_t cpha, __unused spi_order_t order) [inline], [static]
```

Configure SPI.
Configure how the SPI serialises and deserialises data on the wire

**Parameters**

- `spi`: SPI instance specifier, either `spi0` or `spi1`
- `data_bits`: Number of data bits per transfer. Valid values 4..16.
- `cpol`: SSPCLKOUT polarity, applicable to Motorola SPI frame format only.
- `cpha`: SSPCLKOUT phase, applicable to Motorola SPI frame format only
- `order`: Must be SPI_MSB_FIRST, no other values supported on the PL022

4.1.18.7.13. spi_set_slave

```c
static void spi_set_slave (spi_inst_t * spi, bool slave) [inline], [static]
```

Set SPI master/slave.
Configure the SPI for master- or slave-mode operation. By default, `spi_init()` sets master-mode.

**Parameters**

- `spi`: SPI instance specifier, either `spi0` or `spi1`
- `slave`: true to set SPI device as a slave device, false for master.

4.1.18.7.14. spi_write16_blocking

```c
int spi_write16_blocking (spi_inst_t * spi, const uint16_t * src, size_t len)
```

Write to an SPI device.
Write `len` halfwords from `src` to SPI. Discard any data received back. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

> **NOTE**

SPI should be initialised with 16 data_bits using `spi_set_format` first, otherwise this function will only write 8 data_bits.

**Parameters**

- `spi`: SPI instance specifier, either `spi0` or `spi1`
- `src`: Buffer of data to write
- `len`: Length of buffers

**Returns**

Number of halfwords written/read

4.1.18.7.15. spi_write16_read16_blocking

```c
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 $\textit{len}$ halfwords from $\textit{src}$ to SPI. Simultaneously read $\textit{len}$ halfwords from SPI to $\textit{dst}$. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

**NOTE**

SPI should be initialised with 16 data_bits using `spi_set_format` first, otherwise this function will only read/write 8 data_bits.

**Parameters**

- $\textit{spi}$: SPI instance specifier, either `spi0` or `spi1`
- $\textit{src}$: Buffer of data to write
- $\textit{dst}$: Buffer for read data
- $\textit{len}$: Length of BOTH buffers in halfwords

**Returns**

Number of halfwords written/read

4.1.18.7.16. spi_write_blocking

```c
int spi_write_blocking (spi_inst_t * spi, const uint8_t * src, size_t len)
```

Write to an SPI device, blocking.

Write $\textit{len}$ bytes from $\textit{src}$ 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**

- $\textit{spi}$: SPI instance specifier, either `spi0` or `spi1`
- $\textit{src}$: Buffer of data to write
- $\textit{len}$: Length of $\textit{src}$

**Returns**

Number of bytes written/read

4.1.18.7.17. spi_write_read_blocking

```c
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 $\textit{len}$ bytes from $\textit{src}$ to SPI. Simultaneously read $\textit{len}$ bytes from SPI to $\textit{dst}$. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

**Parameters**

- $\textit{spi}$: SPI instance specifier, either `spi0` or `spi1`
- $\textit{src}$: Buffer of data to write
- $\textit{dst}$: Buffer for read data
- $\textit{len}$: Length of BOTH buffers

**Returns**

Number of bytes written/read
4.1.19. hardware_sync

4.1.19.1. Detailed Description

Low level hardware spin locks, barrier and processor event APIs

Spin Locks

The RP2040 provides 32 hardware spin locks, which can be used to manage mutually-exclusive access to shared software and hardware resources.

Generally each spin lock itself is a shared resource, i.e. the same hardware spin lock can be used by multiple higher level primitives (as long as the spin locks are neither held for long periods, nor held concurrently with other spin locks by the same core - which could lead to deadlock). A hardware spin lock that is exclusively owned can be used individually without more flexibility and without regard to other software. Note that no hardware spin lock may be acquired re-entrantly (i.e. hardware spin locks are not on their own safe for use by both thread code and IRQs) however the default spin lock related methods here (e.g. spin_lock_blocking) always disable interrupts while the lock is held as use by IRQ handlers and user code is common/desirable, and spin locks are only expected to be held for brief periods.

The SDK uses the following default spin lock assignments, classifying which spin locks are reserved for exclusive/special purposes vs those suitable for more general shared use:

<table>
<thead>
<tr>
<th>Number (ID)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-13</td>
<td>Currently reserved for exclusive use by the SDK and other libraries. If you use these spin locks, you risk breaking SDK or other library functionality. Each reserved spin lock used individually has its own PICO_SPINLOCK_ID so you can search for those.</td>
</tr>
<tr>
<td>14,15</td>
<td>(PICO_SPINLOCK_ID_OS1 and PICO_SPINLOCK_ID_OS2). Currently reserved for exclusive use by an operating system (or other system level software) co-existing with the SDK.</td>
</tr>
<tr>
<td>16-23</td>
<td>(PICO_SPINLOCK_ID_STRIPED_FIRST - PICO_SPINLOCK_ID_STRIPED_LAST). Spin locks from this range are assigned in a round-robin fashion via next_striped_spin_lock_num(). These spin locks are shared, but assigning numbers from a range reduces the probability that two higher level locking primitives using striped spin locks will actually be using the same spin lock.</td>
</tr>
<tr>
<td>24-31</td>
<td>(PICO_SPINLOCK_IDCLAIM_FREE_FIRST - PICO_SPINLOCK_IDCLAIM_FREE_LAST). These are reserved for exclusive use and are allocated on a first come first served basis at runtime via spin_lock_claim_unused().</td>
</tr>
</tbody>
</table>

4.1.19.2. Typedefs

typedef volatile uint32_t spin_lock_t

A spin lock identifier.
4.1.19.3. Functions

static __force_inline void __sev (void)
    Insert a SEV instruction in to the code path.

static __force_inline void __wfe (void)
    Insert a WFE instruction in to the code path.

static __force_inline void __wfi (void)
    Insert a WFI instruction in to the code path.

static __force_inline void __dmb (void)
    Insert a DMB instruction in to the code path.

static __force_inline void __dsb (void)
    Insert a DSB instruction in to the code path.

static __force_inline void __isb (void)
    Insert a ISB instruction in to the code path.

static __force_inline void __mem_fence_acquire (void)
    Acquire a memory fence.

static __force_inline void __mem_fence_release (void)
    Release a memory fence.

static __force_inline uint32_t save_and_disable_interrupts (void)
    Save and disable interrupts.

static __force_inline void restore_interrupts (uint32_t status)
    Restore interrupts to a specified state.

static __force_inline spin_lock_t * spin_lock_instance (uint lock_num)
    Get HW Spinlock instance from number.

static __force_inline uint spin_lock_get_num (spin_lock_t *lock)
    Get HW Spinlock number from instance.

static __force_inline void spin_lock_unsafe_blocking (spin_lock_t *lock)
    Acquire a spin lock without disabling interrupts (hence unsafe)

static __force_inline void spin_unlockUnsafe (spin_lock_t *lock)
    Release a spin lock without re-enabling interrupts.

static __force_inline uint32_t spin_lock_blocking (spin_lock_t *lock)
    Acquire a spin lock safely.

static bool is_spin_locked (spin_lock_t *lock)
    Check to see if a spinlock is currently acquired elsewhere.

static __force_inline void spin_unlock (spin_lock_t *lock, uint32_t saved_irq)
    Release a spin lock safely.

spin_lock_t * spin_lock_init (uint lock_num)
    Initialise a spin lock.

void spin_locks_reset (void)
    Release all spin locks.

uint next_striped_spin_lock_num (void)
    Return a spin lock number from the striped range.
void spin_lock_claim (uint lock_num)
    Mark a spin lock as used.

void spin_lock_claim_mask (uint32_t lock_num_mask)
    Mark multiple spin locks as used.

void spin_lock_unclaim (uint lock_num)
    Mark a spin lock as no longer used.

int spin_lock_claim_unused (bool required)
    Claim a free spin lock.

bool spin_lock_is_claimed (uint lock_num)
    Determine if a spin lock is claimed.

4.1.19.4. Typedef Documentation

4.1.19.4.1. spin_lock_t
typedef volatile uint32_t spin_lock_t
A spin lock identifier.

4.1.19.5. Function Documentation

4.1.19.5.1. __dmb
static __force_inline void __dmb (void) [static]
Insert a DMB instruction in to the code path.
The DMB (data memory barrier) acts as a memory barrier, all memory accesses prior to this instruction will be observed before any explicit access after the instruction.

4.1.19.5.2. __dsb
static __force_inline void __dsb (void) [static]
Insert a DSB instruction in to the code path.
The DSB (data synchronization barrier) acts as a special kind of data memory barrier (DMB). The DSB operation completes when all explicit memory accesses before this instruction complete.

4.1.19.5.3. __isb
static __force_inline void __isb (void) [static]
Insert a ISB instruction in to the code path.
ISB acts as an instruction synchronization barrier. It flushes the pipeline of the processor, so that all instructions following the ISB are fetched from cache or memory again, after the ISB instruction has been completed.
4.1.19.5.4. __mem_fence_acquire

static __force_inline void __mem_fence_acquire (void) [static]
Acquire a memory fence.

4.1.19.5.5. __mem_fence_release

static __force_inline void __mem_fence_release (void) [static]
Release a memory fence.

4.1.19.5.6. __sev

static __force_inline void __sev (void) [static]
Insert a SEV instruction in to the code path.
The SEV (send event) instruction sends an event to both cores.

4.1.19.5.7. __wfe

static __force_inline void __wfe (void) [static]
Insert a WFE instruction in to the code path.
The WFE (wait for event) instruction waits until one of a number of events occurs, including events signalled by the SEV instruction on either core.

4.1.19.5.8. __wfi

static __force_inline void __wfi (void) [static]
Insert a WFI instruction in to the code path.
The WFI (wait for interrupt) instruction waits for a interrupt to wake up the core.

4.1.19.5.9. is_spin_locked

static bool is_spin_locked (spin_lock_t * lock) [inline], [static]
Check to see if a spinlock is currently acquired elsewhere.

Parameters

  lock  Spinlock instance

4.1.19.5.10. next_striped_spin_lock_num

uint next_striped_spin_lock_num (void)
Return a spin lock number from the striped range.

Returns a spin lock number in the range PICO_SPINLOCK_ID_STRIPED_FIRST to PICO_SPINLOCK_ID_STRIPED_LAST in a round robin fashion. This does not grant the caller exclusive access to the spin lock, so the caller must:

1. Abide (with other callers) by the contract of only holding this spin lock briefly (and with IRQs disabled - the default via spin_lock_blocking()), and not whilst holding other spin locks.
2. Be OK with any contention caused by the - brief due to the above requirement - contention with other possible users of the spin lock.

**Returns**

lock_num a spin lock number the caller may use (non exclusively)

**See also**

PICO_SPINLOCK_ID_STRIPED_FIRST

PICO_SPINLOCK_ID_STRIPED_LAST

### 4.1.19.5.11. restore_interrupts

```c
static __forceinline void restore_interrupts (uint32_t status) [static]
```

Restore interrupts to a specified state.

**Parameters**

- **status** Previous interrupt status from `save_and_disable_interrupts()`

### 4.1.19.5.12. save_and_disable_interrupts

```c
static __forceinline uint32_t save_and_disable_interrupts (void) [static]
```

Save and disable interrupts.

**Returns**

The prior interrupt enable status for restoration later via `restore_interrupts()`

### 4.1.19.5.13. spin_lock_blocking

```c
static __forceinline uint32_t spin_lock_blocking (spin_lock_t * lock) [static]
```

Acquire a spin lock safely.

This function will disable interrupts prior to acquiring the spinlock

**Parameters**

- **lock** Spinlock instance

**Returns**

interrupt status to be used when unlocking, to restore to original state

### 4.1.19.5.14. spin_lock_claim

```c
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
4.1.19.5.15. spin_lock_claim_mask

```c
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)

4.1.19.5.16. spin_lock_claim_unused

```c
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

4.1.19.5.17. spin_lock_get_num

```c
static __force_inline uint spin_lock_get_num (spin_lock_t * lock) [static]
```

Get HW Spinlock number from instance.

**Parameters**

- `lock` The Spinlock instance

**Returns**

- The Spinlock ID

4.1.19.5.18. spin_lock_init

```c
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

4.1.19.5.19. spin_lock_instance

```c
static __force_inline spin_lock_t * spin_lock_instance (uint lock_num) [static]
```

Get HW Spinlock instance from number.

**Parameters**


lock_num Spinlock ID

Returns
The spinlock instance

4.1.19.5.20. spin_lock_is_claimed

bool spin_lock_is_claimed (uint lock_num)
Determine if a spin lock is claimed.
Parameters
lock_num the spin lock number
Returns
true if claimed, false otherwise
See also
spin_lock_claim
spin_lock_claim_mask

4.1.19.5.21. 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

4.1.19.5.22. spin_lock_unsafe_blocking

static __force_inline void spin_lock_unsafe_blocking (spin_lock_t * lock) [static]
Acquire a spin lock without disabling interrupts (hence unsafe)
Parameters
lock Spinlock instance

4.1.19.5.23. spin_locks_reset

void spin_locks_reset (void)
Release all spin locks.

4.1.19.5.24. spin_unlock

static __force_inline void spin_unlock (spin_lock_t * lock, uint32_t saved_irq) [static]
Release a spin lock safely.
This function will re-enable interrupts according to the parameters.
Parameters

4.1. Hardware APIs
lock Spinlock instance

saved_irq Return value from the spin_lock_blocking() function.

See also
spin_lock_blocking()

4.1.19.5.25. spin_unlock_unsafe

static __force_inline void spin_unlock_unsafe (spin_lock_t * lock) [static]
Release a spin lock without re-enabling interrupts.

Parameters
lock Spinlock instance

4.1.20. hardware_timer

4.1.20.1. Detailed Description

Low-level hardware timer API

This API provides medium level access to the timer HW. See also pico_time which provides higher levels functionality using the hardware timer.

The timer peripheral on RP2040 supports the following features:

• single 64-bit counter, incrementing once per microsecond
• Latching two-stage read of counter, for race-free read over 32 bit bus
• Four alarms: match on the lower 32 bits of counter, IRQ on match.

By default the timer uses a one microsecond reference that is generated in the Watchdog (see Section 4.8.2) which is derived from the clk_ref.

The timer has 4 alarms, and can output a separate interrupt for each alarm. The alarms match on the lower 32 bits of the 64 bit counter which means they can be fired a maximum of $2^{32}$ microseconds into the future. This is equivalent to:

• $2^{32} \div 10^6 \approx 4295$ seconds
• $4295 \div 60 \approx 72$ 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());
}
```
```c
15     return true;
16 }
17
18 int main() {
19     stdio_init_all();
20     printf("Hello Timer!\n");
21     // Call alarm_callback in 2 seconds
22     add_alarm_in_ms(2000, alarm_callback, NULL, false);
23     // Wait for alarm callback to set timer_fired
24     while (!timer_fired) {
25         tight_loop_contents();
26     }
27     // Create a repeating timer that calls repeating_timer_callback.
28     // If the delay is > 0 then this is the delay between the previous callback ending and the
29     // next starting.
30     // If the delay is negative (see below) then the next call to the callback will be exactly
31     // 500ms after the
32     // start of the call to the last callback
33     struct repeating_timer timer;
34     add_repeating_timer_ms(500, repeating_timer_callback, NULL, &timer);
35     sleep_ms(3000);
36     bool cancelled = cancel_repeating_timer(&timer);
37     printf("cancelled... %d\n", cancelled);
38     sleep_ms(2000);
39     // Negative delay so means we will call repeating_timer_callback, and call it again
40     // 500ms later regardless of how long the callback took to execute
41     add_repeating_timer_ms(-500, repeating_timer_callback, NULL, &timer);
42     sleep_ms(3000);
43     cancelled = cancel_repeating_timer(&timer);
44     printf("cancelled... %d\n", cancelled);
45     sleep_ms(2000);
46     printf("Done\n");
47     return 0;
48 }
```

See also

`pico_time`

### 4.1.20.2. Typedefs

typedef void(* hardware_alarm_callback_t)(uint alarm_num)

### 4.1.20.3. Functions

static uint32_t time_us_32 (void)

Return a 32 bit timestamp value in microseconds.

uint64_t time_us_64 (void)

Return the current 64 bit timestamp value in microseconds.

void busy_wait_us_32 (uint32_t delay_us)

Busy wait wasting cycles for the given (32 bit) number of microseconds.
void busy_wait_us (uint64_t delay_us)
    Busy wait wasting cycles for the given (64 bit) number of microseconds.

void busy_wait_ms (uint32_t delay_ms)
    Busy wait wasting cycles for the given number of milliseconds.

void busy_wait_until (absolute_time_t t)
    Busy wait wasting cycles until after the specified timestamp.

static bool time_reached (absolute_time_t t)
    Check if the specified timestamp has been reached.

void hardware_alarm_claim (uint alarm_num)
    cooperatively claim the use of this hardware alarm_num

int hardware_alarm_claim_unused (bool required)
    cooperatively claim the use of this hardware alarm_num

void hardware_alarm_unclaim (uint alarm_num)
    cooperatively release the claim on use of this hardware alarm_num

bool hardware_alarm_is_claimed (uint alarm_num)
    Determine if a hardware alarm has been claimed.

void hardware_alarm_set_callback (uint alarm_num, hardware_alarm_callback_t callback)
    Enable/Disable a callback for a hardware timer on this core.

bool hardware_alarm_set_target (uint alarm_num, absolute_time_t t)
    Set the current target for the specified hardware alarm.

void hardware_alarm_cancel (uint alarm_num)
    Cancel an existing target (if any) for a given hardware_alarm.

void hardware_alarm_force_irq (uint alarm_num)
    Force and IRQ for a specific hardware alarm.

4.1.20.4. Typedef Documentation

4.1.20.4.1. hardware_alarm_callback_t

typedef void(* hardware_alarm_callback_t) (uint alarm_num)
    Callback function type for hardware alarms

Parameters
    alarm_num        the hardware alarm number

See also
    hardware_alarm_set_callback()

4.1.20.5. Function Documentation

4.1.20.5.1. busy_wait_ms

void busy_wait_ms (uint32_t delay_ms)
Busy wait wasting cycles for the given number of milliseconds.

**Parameters**

- `delay_ms` delay amount in milliseconds

### 4.1.20.5.2. busy_wait_until

```c
void busy_wait_until (absolute_time_t t)
```

Busy wait wasting cycles until after the specified timestamp.

**Parameters**

- `t` Absolute time to wait until

### 4.1.20.5.3. busy_wait_us

```c
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 in microseconds

### 4.1.20.5.4. busy_wait_us_32

```c
void busy_wait_us_32 (uint32_t delay_us)
```

Busy wait wasting cycles for the given (32 bit) number of microseconds.

**Parameters**

- `delay_us` delay amount in microseconds

### 4.1.20.5.5. hardware_alarm_cancel

```c
void hardware_alarm_cancel (uint alarm_num)
```

Cancel an existing target (if any) for a given hardware_alarm.

**Parameters**

- `alarm_num` the hardware alarm number

### 4.1.20.5.6. hardware_alarm_claim

```c
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
4.1.20.5.7. hardware_alarm_claim_unused

```c
int hardware_alarm_claim_unused (bool required)
```

cooperatively claim the use of this hardware alarm_num

This method attempts to claim an unused hardware alarm

**Returns**

alarm_num the hardware alarm claimed or -1 if requires was false, and none are available

**See also**

hardware_claiming

4.1.20.5.8. hardware_alarm_force_irq

```c
void hardware_alarm_force_irq (uint alarm_num)
```

Force and IRQ for a specific hardware alarm.

This method will forcibly make sure the current alarm callback (if present) for the hardware alarm is called from an IRQ context after this call. If an actual callback is due at the same time then the callback may only be called once.

Calling this method does not otherwise interfere with regular callback operations.

**Parameters**

- `alarm_num` the hardware alarm number

4.1.20.5.9. hardware_alarm_is_claimed

```c
bool hardware_alarm_is_claimed (uint alarm_num)
```

Determine if a hardware alarm has been claimed.

**Parameters**

- `alarm_num` the hardware alarm number

**Returns**

true if claimed, false otherwise

**See also**

hardware_alarm_claim

4.1.20.5.10. hardware_alarm_set_callback

```c
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
**NOTE**

This will install the handler on the current core if the IRQ handler isn’t already set. Therefore the user has the opportunity to call this up from the core of their choice.

**Parameters**

- `alarm_num` the hardware alarm number
- `callback` the callback to install, or NULL to unset

**See also**

`hardware_alarm_set_target()`

### 4.1.20.5.11. hardware_alarm_set_target

```c
bool hardware_alarm_set_target (uint alarm_num, absolute_time_t t)
```

Set the current target for the specified hardware alarm. This will replace any existing target.

**Parameters**

- `alarm_num` the hardware alarm number
- `t` the target timestamp

**Returns**

true if the target was “missed”; i.e. it was in the past, or occurred before a future hardware timeout could be set

### 4.1.20.5.12. hardware_alarm_unclaim

```c
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`

### 4.1.20.5.13. time_reached

```c
static bool time_reached (absolute_time_t t) [inline], [static]
```

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
4.1.20.5.14. time_us_32

static uint32_t time_us_32 (void) [inline], [static]

Return a 32 bit timestamp value in microseconds.

Returns the low 32 bits of the hardware timer.

NOTE

This value wraps roughly every 1 hour 11 minutes and 35 seconds.

Returns

the 32 bit timestamp

4.1.20.5.15. time_us_64

uint64_t time_us_64 (void)

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).

Returns

the 64 bit timestamp

Return the current 64 bit timestamp value in microseconds.

4.1.21. hardware_uart

4.1.21.1. Detailed Description

Hardware UART API

RP2040 has 2 identical instances of a UART peripheral, based on the ARM PL011. Each UART can be connected to a number of GPIO pins as defined in the GPIO muxing.

Only the TX, RX, RTS, and CTS signals are connected, meaning that the modem mode and IrDA mode of the PL011 are not supported.

Example

```c
int main() {

    // Initialise UART 0
    uart_init(uart0, 115200);

    // Set the GPIO pin mux to the UART - 0 is TX, 1 is RX
    gpio_set_function(0, GPIO_FUNC_UART);
    gpio_set_function(1, GPIO_FUNC_UART);

    uart_puts(uart0, "Hello world!");
}
```
4.1.21.2. Enumerations

```c
enum uart_parity_t { UART_PARITY_NONE, UART_PARITY_EVEN, UART_PARITY_ODD }
```

UART Parity enumeration.

4.1.21.3. Functions

```c
static uint uart_get_index (uart_inst_t *uart)
    Convert UART instance to hardware instance number.

uint uart_init (uart_inst_t *uart, uint baudrate)
    Initialise a UART.

void uart_deinit (uart_inst_t *uart)
    Deinitialise a UART.

uint uart_set_baudrate (uart_inst_t *uart, uint baudrate)
    Set UART baud rate.

static void uart_set_hw_flow (uart_inst_t *uart, bool cts, bool rts)
    Set UART flow control CTS/RTS.

void uart_set_format (uart_inst_t *uart, uint data_bits, uint stop_bits, uart_parity_t parity)
    Set UART data format.

static void uart_set_irq_enables (uart_inst_t *uart, bool rx_has_data, bool tx_needs_data)
    Setup UART interrupts.

static bool uart_is_enabled (uart_inst_t *uart)
    Test if specific UART is enabled.

void uart_set_fifo_enabled (uart_inst_t *uart, bool enabled)
    Enable/Disable the FIFOs on specified UART.

static bool uart_is_writable (uart_inst_t *uart)
    Determine if space is available in the TX FIFO.

static void uart_tx_wait_blocking (uart_inst_t *uart)
    Wait for the UART TX fifo to be drained.

static bool uart_is_readable (uart_inst_t *uart)
    Determine whether data is waiting in the RX FIFO.

static void uart_write_blocking (uart_inst_t *uart, const uint8_t *src, size_t len)
    Write to the UART for transmission.

static void uart_read_blocking (uart_inst_t *uart, uint8_t *dst, size_t len)
    Read from the UART.

static void uart_putchar_raw (uart_inst_t *uart, char c)
    Write single character to UART for transmission.

static void uart_putchar (uart_inst_t *uart, char c)
    Write single character to UART for transmission, with optional CR/LF conversions.

static void uart_puts (uart_inst_t *uart, const char *s)
    Write string to UART for transmission, doing any CR/LF conversions.
```
static char uart_getc (uart_inst_t *uart)
    Read a single character from the UART.

void uart_set_break (uart_inst_t *uart, bool en)
    Assert a break condition on the UART transmission.

void uart_set_translate_crlf (uart_inst_t *uart, bool translate)
    Set CR/LF conversion on UART.

static void uart_default_tx_wait_blocking (void)
    Wait for the default UART’s TX FIFO to be drained.

bool uart_is_readable_within_us (uart_inst_t *uart, uint32_t us)
    Wait for up to a certain number of microseconds for the RX FIFO to be non empty.

static uint uart_get_dreq (uart_inst_t *uart, bool is_tx)
    Return the DREQ to use for pacing transfers to/from a particular UART instance.

4.1.21.3.1. uart0
#define uart0 ((uart_inst_t *)uart0_hw)
Identifier for UART instance 0.
The UART identifiers for use in UART functions.
e.g. uart_init(uart1, 48000)

4.1.21.3.2. uart1
#define uart1 ((uart_inst_t *)uart1_hw)
Identifier for UART instance 1.

4.1.21.4. Enumeration Type Documentation

4.1.21.4.1. uart_parity_t
enum uart_parity_t
    UART Parity enumeration.

4.1.21.5. Function Documentation

4.1.21.5.1. uart_default_tx_wait_blocking
static void uart_default_tx_wait_blocking (void) [inline], [static]
    Wait for the default UART’s TX FIFO to be drained.

4.1.21.5.2. 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`

### 4.1.21.5.3. uart_get_dreq

```c
static uint uart_get_dreq (uart_inst_t * uart, bool is_tx) [inline], [static]
```

Return the DREQ to use for pacing transfers to/from a particular UART instance.

**Parameters**

- `uart` UART instance. `uart0` or `uart1`
- `is_tx` true for sending data to the UART instance, false for receiving data from the UART instance

### 4.1.21.5.4. uart_get_index

```c
static uint uart_get_index (uart_inst_t * uart) [inline], [static]
```

Convert UART instance to hardware instance number.

**Parameters**

- `uart` UART instance

**Returns**

Number of UART, 0 or 1.

### 4.1.21.5.5. uart_getc

```c
static char uart_getc (uart_inst_t * uart) [inline], [static]
```

Read a single character from the UART.

This function will block until a character has been read.

**Parameters**

- `uart` UART instance. `uart0` or `uart1`

**Returns**

The character read.

### 4.1.21.5.6. uart_init

```c
uint uart_init (uart_inst_t * uart, uint baudrate)
```

Initialise a UART.

Put the UART into a known state, and enable it. Must be called before other functions.

This function always enables the FIFOs, and configures the UART for the following default line format:

- 8 data bits
- No parity bit
- One stop bit
NOTE

There is no guarantee that the baudrate requested will be possible, the nearest will be chosen, and this function will return the configured baud rate.

Parameters

- `uart`: UART instance. `uart0` or `uart1`
- `baudrate`: Baudrate of UART in Hz

Returns

Actual set baudrate

4.1.21.5.7. `uart_is_enabled`

```
static bool uart_is_enabled (uart_inst_t * uart) [inline], [static]
```

Test if specific UART is enabled.

Parameters

- `uart`: UART instance. `uart0` or `uart1`

Returns

true if the UART is enabled

4.1.21.5.8. `uart_is_readable`

```
static bool uart_is_readable (uart_inst_t * uart) [inline], [static]
```

Determine whether data is waiting in the RX FIFO.

Parameters

- `uart`: UART instance. `uart0` or `uart1`

Returns

true if the RX FIFO is not empty, otherwise false.

4.1.21.5.9. `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

4.1.21.5.10. `uart_is_writable`

```
static bool uart_is_writable (uart_inst_t * uart) [inline], [static]
```

Determine if space is available in the TX FIFO.
Parameters

- uart: UART instance. uart0 or uart1

Returns

false if no space available, true otherwise

---

### 4.1.21.5.11. uart_putchar

```c
static void uart_putchar (uart_inst_t * uart, char c) [inline], [static]
```

Write single character to UART for transmission, with optional CR/LF conversions.

This function will block until the character has been sent.

**Parameters**

- uart: UART instance. uart0 or uart1
- c: The character to send

---

### 4.1.21.5.12. uart_putchar_raw

```c
static void uart_putchar_raw (uart_inst_t * uart, char c) [inline], [static]
```

Write single character to UART for transmission.

This function will block until the entire character has been sent.

**Parameters**

- uart: UART instance. uart0 or uart1
- c: The character to send

---

### 4.1.21.5.13. uart_puts

```c
static void uart_puts (uart_inst_t * uart, const char * s) [inline], [static]
```

Write string to UART for transmission, doing any CR/LF conversions.

This function will block until the entire string has been sent.

**Parameters**

- uart: UART instance. uart0 or uart1
- s: The null terminated string to send

---

### 4.1.21.5.14. uart_read_blocking

```c
static void uart_read_blocking (uart_inst_t * uart, uint8_t * dst, size_t len) [inline], [static]
```

Read from the UART.

This function blocks until len characters have been read from the UART.

**Parameters**

- uart: UART instance. uart0 or uart1
- dst: Buffer to accept received bytes
- len: The number of bytes to receive.
4.1.21.5.15. uart_set_baudrate

```c
uint uart_set_baudrate (uart_inst_t * uart, uint baudrate)
```

Set UART baud rate.

Set baud rate as close as possible to requested, and return actual rate selected.

The UART is paused for around two character periods whilst the settings are changed. Data received during this time may be dropped by the UART.

Any characters still in the transmit buffer will be sent using the new updated baud rate. `uart_tx_wait_blocking()` can be called before this function to ensure all characters at the old baud rate have been sent before the rate is changed.

This function should not be called from an interrupt context, and the UART interrupt should be disabled before calling this function.

**Parameters**

- **uart**: UART instance. `uart0` or `uart1`
- **baudrate**: Baudrate in Hz

**Returns**

Actual set baudrate

4.1.21.5.16. uart_set_break

```c
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.

4.1.21.5.17. uart_set_fifo_enabled

```c
void uart_set_fifo_enabled (uart_inst_t * uart, bool enabled)
```

Enable/Disable the FIFOs on specified UART.

The UART is paused for around two character periods whilst the settings are changed. Data received during this time may be dropped by the UART.

Any characters still in the transmit FIFO will be lost if the FIFO is disabled. `uart_tx_wait_blocking()` can be called before this function to avoid this.

This function should not be called from an interrupt context, and the UART interrupt should be disabled when calling this function.

**Parameters**

- **uart**: UART instance. `uart0` or `uart1`
- **enabled**: true to enable FIFO (default), false to disable

4.1.21.5.18. uart_set_format

```c
void uart_set_format (uart_inst_t * uart, uint data_bits, uint stop_bits, uart_parity_t parity)
```

Set UART data format.
Configure the data format (bits etc) for the UART.

The UART is paused for around two character periods whilst the settings are changed. Data received during this time may be dropped by the UART.

Any characters still in the transmit buffer will be sent using the new updated data format. `uart_tx_wait_blocking()` can be called before this function to ensure all characters needing the old format have been sent before the format is changed.

This function should not be called from an interrupt context, and the UART interrupt should be disabled before calling this function.

**Parameters**

- `uart` UART instance. `uart0` or `uart1`
- `data_bits` Number of bits of data. 5..8
- `stop_bits` Number of stop bits 1..2
- `parity` Parity option.

### 4.1.21.5.19. `uart_set_hw_flow`

```c
static void uart_set_hw_flow (uart_inst_t * uart, bool cts, bool rts) [inline], [static]
```

Set UART flow control CTS/RTS.

**Parameters**

- `uart` UART instance. `uart0` or `uart1`
- `cts` If true enable flow control of TX by clear-to-send input
- `rts` If true enable assertion of request-to-send output by RX flow control

### 4.1.21.5.20. `uart_set_irq_enables`

```c
static void uart_set_irq_enables (uart_inst_t * uart, bool rx_has_data, bool tx_needs_data) [inline], [static]
```

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 contains data.
- `tx_needs_data` If true an interrupt will be fired when the TX FIFO needs data.

### 4.1.21.5.21. `uart_set_translate_crlf`

```c
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
4.1.21.5.22. uart_tx_wait_blocking

static void uart_tx_wait_blocking (uart_inst_t * uart) [inline], [static]
Wait for the UART TX fifo to be drained.

Parameters
- uart UART instance. uart0 or uart1

4.1.21.5.23. uart_write_blocking

static void uart_write_blocking (uart_inst_t * uart, const uint8_t * src, size_t len) [inline], [static]
Write to the UART for transmission.
This function will block until all the data has been sent to the UART

Parameters
- uart UART instance. uart0 or uart1
- src The bytes to send
- len The number of bytes to send

4.1.22. hardware_vreg

4.1.22.1. Detailed Description
Voltage Regulation API

4.1.22.2. Functions

void vreg_set_voltage (enum vreg_voltage voltage)
Set voltage.

4.1.22.3. Function Documentation

4.1.22.3.1. vreg_set_voltage

void vreg_set_voltage (enum vreg_voltage voltage)
Set voltage.

Parameters
- voltage The voltage (from enumeration vreg_voltage) to apply to the voltage regulator

4.1.23. hardware_watchdog
4.1.23.1. Detailed Description

Hardware Watchdog Timer API

Supporting functions for the Pico hardware watchdog timer.

The RP2040 has a built-in HW watchdog Timer. This is a countdown timer that can restart parts of the chip if it reaches zero. For example, this can be used to restart the processor if the software running on it gets stuck in an infinite loop or similar. The programmer has to periodically write a value to the watchdog to stop it reaching zero.

Example

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

int main() {
    stdio_init_all();
    if (watchdog_caused_reboot()) {
        printf("Rebooted by Watchdog!\n");
        return 0;
    } else {
        printf('Clean boot\n');
    }
    // Enable the watchdog, requiring the watchdog to be updated every 100ms or the chip will reboot
    // second arg is pause on debug which means the watchdog will pause when stepping through code
    watchdog_enable(100, 1);
    for (uint i = 0; i < 5; i++) {
        printf("Updating watchdog %d\n", i);
        watchdog_update();
    }
    // Wait in an infinite loop and don’t update the watchdog so it reboots us
    printf("Waiting to be rebooted by watchdog\n");
    while(1);
}
```

4.1.23.2. Functions

- `void watchdog_reboot (uint32_t pc, uint32_t sp, uint32_t delay_ms)`
  Define actions to perform at watchdog timeout.

- `void watchdog_start_tick (uint cycles)`
  Start the watchdog tick.

- `void watchdog_update (void)`
  Reload the watchdog counter with the amount of time set in watchdog_enable.

- `void watchdog_enable (uint32_t delay_ms, bool pause_on_debug)`
  Enable the watchdog.

- `bool watchdog_caused_reboot (void)`
  Did the watchdog cause the last reboot?
```c
bool watchdog_enable_caused_reboot (void)
Did watchdog_enable cause the last reboot?
```

```c
uint32_t watchdog_get_count (void)
Returns the number of microseconds before the watchdog will reboot the chip.
```

### 4.1.23.3. Function Documentation

#### 4.1.23.3.1. watchdog_caused_reboot

```c
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` if there has been no watchdog reboot since the last power on reset. A power on reset is typically caused by a power cycle or the run pin (reset button) being toggled.

#### 4.1.23.3.2. watchdog_enable

```c
void watchdog_enable (uint32_t delay_ms, bool pause_on_debug)
Enable the watchdog.
```

**NOTE**

If `watchdog_start_tick` value does not give a 1MHz clock to the watchdog system, then the `delay_ms` parameter will not be in microseconds. See the datasheet for more details.

By default the SDK assumes a 12MHz XOSC and sets the `watchdog_start_tick` appropriately.

This method sets a marker in the watchdog scratch register 4 that is checked by `watchdog_enable_caused_reboot`. If the device is subsequently reset via a call to `watchdog_reboot` (including for example by dragging a UF2 onto the RPI-RP2), then this value will be cleared, and so `watchdog_enable_caused_reboot` will return false.

**Parameters**

- `delay_ms` Number of milliseconds before watchdog will reboot without `watchdog_update` being called. Maximum of 0x7ffffff, which is approximately 8.3 seconds
- `pause_on_debug` If the watchdog should be paused when the debugger is stepping through code

#### 4.1.23.3.3. watchdog_enable_caused_reboot

```c
bool watchdog_enable_caused_reboot (void)
Did watchdog_enable cause the last reboot?
```

Perform additional checking along with `watchdog_caused_reboot` to determine if a watchdog timeout initiated by `watchdog_enable` caused the last reboot.

This method checks for a special value in watchdog scratch register 4 placed there by `watchdog_enable`. This would not be present if a watchdog reset is initiated by `watchdog_reboot` or by the RP2040 bootrom (e.g. dragging a UF2 onto the RPI-RP2 drive).
Returns
true If the watchdog timer or a watchdog force caused (see watchdog_caused_reboot) the last reboot and the watchdog reboot happened after watchdog_enable was called

Returns
false If there has been no watchdog reboot since the last power on reset, or the watchdog reboot was not caused by a watchdog timeout after watchdog_enable was called. A power on reset is typically caused by a power cycle or the run pin (reset button) being toggled.

4.1.23.3.4. 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.

4.1.23.3.5. watchdog_reboot

void watchdog_reboot (uint32_t pc, uint32_t sp, uint32_t delay_ms)
Define actions to perform at watchdog timeout.

NOTE
If watchdog_start_tick value does not give a 1MHz clock to the watchdog system, then the delay_ms parameter will not be in microseconds. See the datasheet for more details.

By default the SDK assumes a 12MHz XOSC and sets the watchdog_start_tick appropriately.

Parameters
pc If Zero, a standard boot will be performed, if non-zero this is the program counter to jump to on reset.
sp If pc is non-zero, this will be the stack pointer used.
delay_ms Initial load value. Maximum value 0x7fffffff, approximately 8.3s.

4.1.23.3.6. 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.

4.1.23.3.7. watchdog_update

void watchdog_update (void)
Reload the watchdog counter with the amount of time set in watchdog_enable.
4.1.24. hardware_xosc

4.1.24.1. Detailed Description

Crystal Oscillator (XOSC) API

4.1.24.2. Functions

```c
void xosc_init (void)
    Initialise the crystal oscillator system.
```

```c
void xosc_disable (void)
    Disable the Crystal oscillator.
```

```c
void xosc_dormant (void)
    Set the crystal oscillator system to dormant.
```

4.1.24.3. Function Documentation

4.1.24.3.1. xosc_disable

```c
void xosc_disable (void)
    Disable the Crystal oscillator.
```

Disables the crystal oscillator source, and waits for it to become unstable.

4.1.24.3.2. xosc_dormant

```c
void xosc_dormant (void)
    Set the crystal oscillator system to dormant.
```

Sets the crystal oscillator system to dormant. This function will block until the oscillator becomes stable after its wake up.

4.1.24.3.3. xosc_init

```c
void xosc_init (void)
    Initialise the crystal oscillator system.
```

Initialises 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_async_context
```
async_context_freertos
async_context_poll
async_context_threadsafe_background
pico_flash
pico_i2c_slave
pico_multicore
fifo Functions for the inter-core FIFOs.
lockout Functions to enable one core to force the other core to pause execution in a known state.
pico_rand
pico_stdlib
pico_sync
critical_section Critical Section API for short-lived mutual exclusion safe for IRQ and multi-core.
lock_core base synchronization/lock primitive support
mutex Mutex API for non IRQ mutual exclusion between cores.
sem Semaphore API for restricting access to a resource.
pico_time
timestamp Timestamp functions relating to points in time (including the current time)
sleep Sleep functions for delaying execution in a lower power state.
alarm Alarm functions for scheduling future execution.
repeating_timer Repeating Timer functions for simple scheduling of repeated execution.
pico_unique_id
pico_util Useful data structures and utility functions.
datetime Date/Time formatting.
pheap
queue

4.2.1. pico_async_context

4.2.1.1. Detailed Description

An async_context provides a logically single-threaded context for performing work, and responding to asynchronous events. Thus an async_context instance is suitable for servicing third-party libraries that are not re-entrant.

The "context" in async_context refers to the fact that when calling workers or timeouts within the async_context various pre-conditions hold:

1. That there is a single logical thread of execution; i.e. that the context does not call any worker functions concurrently.

2. That the context always calls workers from the same processor core, as most uses of async_context rely on
interaction with IRQs which are themselves core-specific.

The `async_context` provides two mechanisms for asynchronous work:

- **when_pending** workers, which are processed whenever they have work pending. See `async_context_add_when_pending_worker`, `async_context_remove_when_pending_worker`, and `async_context_set_work_pending`, the latter of which can be used from an interrupt handler to signal that servicing work is required to be performed by the worker from the regular `async_context`.

- **at_time** workers, that are executed after at a specific time.

Note: "when pending" workers with work pending are executed before "at time" workers.

The `async_context` provides locking mechanisms, see `async_context_acquire_lock_blocking`, `async_context_release_lock` and `async_context_lock_check` which can be used by external code to ensure execution of external code does not happen concurrently with worker code. Locked code runs on the calling core, however `async_context_execute_sync` is provided to synchronously run a function from the core of the `async_context`.

The SDK ships with the following default `async_contexts`:

- `async_context_poll` - this context is not thread-safe, and the user is responsible for calling `async_context_poll()` periodically, and can use `async_context_wait_for_work_until()` to sleep between calls until work is needed if the user has nothing else to do.

- `async_context_threadsafe_background` - in order to work in the background, a low priority IRQ is used to handle callbacks. Code is usually invoked from this IRQ context, but may be invoked after any other code that uses the async context in another (non-IRQ) context on the same core. Calling `async_context_poll()` is not required, and is a no-op. This context implements `async_context` locking and is thus safe to call from either core, according to the specific notes on each API.

- `async_context_freertos` - Work is performed from a separate "async_context" task, however once again, code may also be invoked after a direct use of the `async_context` on the same core that the `async_context` belongs to. Calling `async_context_poll()` is not required, and is a no-op. This context implements `async_context` locking and is thus safe to call from any task, and from either core, according to the specific notes on each API.

Each `async_context` provides bespoke methods of instantiation which are provided in the corresponding headers (e.g. `async_context_poll.h`, `async_context_threadsafe_background.h`, `async_context_freertos.h`). `async_contexts` are de-initialized by the common `async_context_deinit()` method.

Multiple `async_context` instances can be used by a single application, and they will operate independently.

### 4.2.1.2. Modules

- `async_context_freertos`

- `async_context_poll`

- `async_context_threadsafe_background`

### 4.2.1.3. Typedefs

- **typedef struct** `async_work_on_timeout async_at_time_worker_t`

  A "timeout" instance used by an `async_context`.

- **typedef struct** `async_when_pending_worker async_when_pending_worker_t`

  A "worker" instance used by an `async_context`.

---

4.2. High Level APIs 249
typedef struct async_context_type async_context_type_t

Implementation of an async_context type, providing methods common to that type.

4.2.1.4. Functions

static void async_context_acquire_lock_blocking (async_context_t *context)

Acquire the async_context lock.

static void async_context_release_lock (async_context_t *context)

Release the async_context lock.

static void async_context_lock_check (async_context_t *context)

Assert if the caller does not own the lock for the async_context.

static uint32_t async_context_execute_sync (async_context_t *context, uint32_t(*func)(void *param), void *param)

Execute work synchronously on the core the async_context belongs to.

static bool async_context_add_at_time_worker (async_context_t *context, async_at_time_worker_t *worker)

Add an "at time" worker to a context.

static bool async_context_add_at_time_worker_at (async_context_t *context, async_at_time_worker_t *worker, absolute_time_t at)

Add an "at time" worker to a context.

static bool async_context_add_at_time_worker_in_ms (async_context_t *context, async_at_time_worker_t *worker, uint32_t ms)

Add an "at time" worker to a context.

static bool async_context_remove_at_time_worker (async_context_t *context, async_at_time_worker_t *worker)

Remove an "at time" worker from a context.

static bool async_context_add_when_pending_worker (async_context_t *context, async_when_pending_worker_t *worker)

Add a "when pending" worker to a context.

static bool async_context_remove_when_pending_worker (async_context_t *context, async_when_pending_worker_t *worker)

Remove a "when pending" worker from a context.

static void async_context_set_work_pending (async_context_t *context, async_when_pending_worker_t *worker)

Mark a "when pending" worker as having work pending.

static void async_context_poll (async_context_t *context)

Perform any pending work for polling style async_context.

static void async_context_wait_until (async_context_t *context, absolute_time_t until)

Sleep until the specified time in an async_context callback safe way.

static void async_context_wait_for_work_until (async_context_t *context, absolute_time_t until)

Block until work needs to be done or the specified time has been reached.

static void async_context_wait_for_work_ms (async_context_t *context, uint32_t ms)

Block until work needs to be done or the specified number of milliseconds have passed.

static uint async_context_core_num (const async_context_t *context)

Return the processor core this async_context belongs to.

static void async_context_deinit (async_context_t *context)

End async_context processing, and free any resources.
4.2.1.5. Typedef Documentation

4.2.1.5.1. async_at_time_worker_t

typedef struct async_work_on_timeout async_at_time_worker_t

A "timeout" instance used by an async_context.

A "timeout" represents some future action that must be taken at a specific time. Its methods are called from the async_context under lock at the given time

See also
async_context_add_worker_at
async_context_add_worker_in_ms

4.2.1.5.2. async_when_pending_worker_t

typedef struct async_when_pending_worker async_when_pending_worker_t

A "worker" instance used by an async_context.

A "worker" represents some external entity that must do work in response to some external stimulus (usually an IRQ). Its methods are called from the async_context under lock at the given time

See also
async_context_add_worker_at
async_context_add_worker_in_ms

4.2.1.5.3. async_context_type_t

typedef struct async_context_type async_context_type_t

Implementation of an async_context type, providing methods common to that type.

4.2.1.6. Function Documentation

4.2.1.6.1. async_context_acquire_lock_blocking

static void async_context_acquire_lock_blocking (async_context_t * context) [inline], [static]

Acquire the async_context lock.

The owner of the async_context lock is the logic owner of the async_context and other work related to this async_context will not happen concurrently.

This method may be called in a nested fashion by the the lock owner.
NOTE
the async_context lock is nestable by the same caller, so an internal count is maintained
for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from
within any worker method called by the async_context or from any other non-IRQ context.

Parameters
context the async_context

See also
async_context_release_lock

4.2.1.6.2. async_context_add_at_time_worker
static bool async_context_add_at_time_worker (async_context_t * context, async_at_time_worker_t * worker) [inline],
[static]
Add an "at time" worker to a context.
An "at time" worker will run at or after a specific point in time, and is automatically when (just before) it runs.
The time to fire is specified in the next_time field of the worker.

NOTE
for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from
within any worker method called by the async_context or from any other non-IRQ context.

Parameters
context the async_context
worker the "at time" worker to add

Returns
ture if the worker was added, false if the worker was already present.

4.2.1.6.3. async_context_add_at_time_worker_at
static bool async_context_add_at_time_worker_at (async_context_t * context, async_at_time_worker_t * worker,
absolute_time_t at) [inline], [static]
Add an "at time" worker to a context.
An "at time" worker will run at or after a specific point in time, and is automatically when (just before) it runs.
The time to fire is specified by the at parameter.

NOTE
for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from
within any worker method called by the async_context or from any other non-IRQ context.

Parameters
context the async_context
worker the "at time" worker to add
at the time to fire at

Returns
true if the worker was added, false if the worker was already present.

4.2.1.6.4. async_context_add_at_time_worker_in_ms

```
static bool async_context_add_at_time_worker_in_ms (async_context_t * context, async_at_time_worker_t * worker, uint32_t ms) [inline], [static]
```

Add an "at time" worker to a context.

An "at time" worker will run at or after a specific point in time, and is automatically when (just before) it runs.

The time to fire is specified by a delay via the ms parameter

**NOTE**

for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from within any worker method called by the async_context or from any other non-IRQ context.

Parameters
- **context** the async_context
- **worker** the "at time" worker to add
- **ms** the number of milliseconds from now to fire after

Returns
true if the worker was added, false if the worker was already present.

4.2.1.6.5. async_context_add_when_pending_worker

```
static bool async_context_add_when_pending_worker (async_context_t * context, async_when_pending_worker_t * worker) [inline], [static]
```

Add a "when pending" worker to a context.

An "when pending" worker will run when it is pending (can be set via async_context_set_work_pending), and is NOT automatically removed when it runs.

The time to fire is specified by a delay via the ms parameter

**NOTE**

for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from within any worker method called by the async_context or from any other non-IRQ context.

Parameters
- **context** the async_context
- **worker** the "when pending" worker to add

Returns
true if the worker was added, false if the worker was already present.
4.2.1.6.6. async_context_core_num

static uint async_context_core_num (const async_context_t * context) [inline], [static]

Return the processor core this async_context belongs to.

Parameters
   context  the async_context

Returns
   the physical core number

4.2.1.6.7. async_context_deinit

static void async_context_deinit (async_context_t * context) [inline], [static]

End async_context processing, and free any resources.

Note the user should clean up any resources associated with workers in the async_context themselves.

Asynchronous (non-polled) async_contexts guarantee that no callback is being called once this method returns.

Parameters
   context  the async_context

4.2.1.6.8. async_context_execute_sync

static uint32_t async_context_execute_sync (async_context_t * context, uint32_t(*)(void * param) func, void * param) [inline], [static]

Execute work synchronously on the core the async_context belongs to.

This method is intended for code external to the async_context (e.g. another thread/task) to execute a function with the same guarantees (single core, logical thread of execution) that async_context workers are called with.

NOTE

you should NOT call this method while holding the async_context's lock

Parameters
   context  the async_context
   func     the function to call
   param    the parameter to pass to the function

Returns
   the return value from func

4.2.1.6.9. async_context_lock_check

static void async_context_lock_check (async_context_t * context) [inline], [static]

Assert if the caller does not own the lock for the async_context.
NOTE

this method is thread-safe

Parameters

context the async_context

4.2.1.6.10. async_context_poll

static void async_context_poll (async_context_t * context) [inline], [static]

Perform any pending work for polling style async_context.

For a polled async_context (e.g. async_context_poll) the user is responsible for calling this method periodically to
perform any required work.

This method may immediately perform outstanding work on other context types, but is not required to.

Parameters

context the async_context

4.2.1.6.11. async_context_release_lock

static void async_context_release_lock (async_context_t * context) [inline], [static]

Release the async_context lock.

NOTE

the async_context lock may be called in a nested fashion, so an internal count is maintained. On the outermost
release, when the outermost lock is released, a check is made for work which might have been skipped while the
lock was held, and any such work may be performed during this call if the call is made from the same core that the
async_context belongs to.

for async_contexts that provide locking (not async_context_poll), this method is thread-safe and may be called from
within any worker method called by the async_context or from any other non-IRQ context.

Parameters

context the async_context

See also
async_context_acquire_lock_blocking

4.2.1.6.12. async_context_remove_at_time_worker

static bool async_context_remove_at_time_worker (async_context_t * context, async_at_time_worker_t * worker) [inline],
[static]

Remove an "at time" worker from a context.
NOTE

for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from within any worker method called by the async_context or from any other non-IRQ context.

Parameters

- `context` the async_context
- `worker` the "at time" worker to remove

Returns

true if the worker was removed, false if the instance not present.

4.2.1.6.13. async_context_remove_when_pending_worker

```c
static bool async_context_remove_when_pending_worker (async_context_t * context, async_when_pending_worker_t * worker) [inline], [static]
```

Remove a "when pending" worker from a context.

NOTE

for async_contexts that provide locking (not async_context_poll), this method is threadsafe. and may be called from within any worker method called by the async_context or from any other non-IRQ context.

Parameters

- `context` the async_context
- `worker` the "when pending" worker to remove

Returns

true if the worker was removed, false if the instance not present.

4.2.1.6.14. async_context_set_work_pending

```c
static void async_context_set_work_pending (async_context_t * context, async_when_pending_worker_t * worker) [inline], [static]
```

Mark a "when pending" worker as having work pending.

The worker will be run from the async_context at a later time.

NOTE

this method may be called from any context including IRQS

Parameters

- `context` the async_context
- `worker` the "when pending" worker to mark as pending.

4.2.1.6.15. async_context_wait_for_work_ms

```c
static void async_context_wait_for_work_ms (async_context_t * context, uint32_t ms) [inline], [static]
```

Block until work needs to be done or the specified number of milliseconds have passed.
NOTE

this method should not be called from a worker callback

Parameters

- `context` the `async_context`
- `ms` the number of milliseconds to return after if no work is required

4.2.1.6.16. async_context_wait_for_work_until

```c
static void async_context_wait_for_work_until (async_context_t * context, absolute_time_t until) [inline], [static]
```

Block until work needs to be done or the specified time has been reached.

NOTE

this method should not be called from a worker callback

Parameters

- `context` the `async_context`
- `until` the time to return if no work is required

4.2.1.6.17. async_context_wait_until

```c
static void async_context_wait_until (async_context_t * context, absolute_time_t until) [inline], [static]
```

Sleep until the specified time in an `async_context` callback safe way.

NOTE

for async_contexts that provide locking (not `async_context_poll`), this method is threadsafe. and may be called from within any worker method called by the `async_context` or from any other non-IRQ context.

Parameters

- `context` the `async_context`
- `until` the time to sleep until

4.2.1.7. async_context_freertos

4.2.1.7.1. Detailed Description

async_context_freertos provides an implementation of `async_context` that handles asynchronous work in a separate FreeRTOS task.

4.2.1.7.2. Functions

```c
bool async_context_freertos_init (async_context_freertos_t *self, async_context_freertos_config_t *config)
```

Initialize an `async_context_freertos` instance using the specified configuration.
static async_context_freertos_config_t async_context_freertos_default_config(void)

Return a copy of the default configuration object used by async_context_freertos_init_with_defaults()

static bool async_context_freertos_init_with_defaults(async_context_freertos_t *self)

Initialize an async_context_freertos instance with default values.

4.2.1.7.3. Function Documentation

async_context_freertos_default_config

static async_context_freertos_config_t async_context_freertos_default_config(void) [inline], [static]

Return a copy of the default configuration object used by async_context_freertos_init_with_defaults()

The caller can then modify just the settings it cares about, and call async_context_freertos_init()

Returns

the default configuration object

async_context_freertos_init

bool async_context_freertos_init(async_context_freertos_t *self, async_context_freertos_config_t *config)

Initialize an async_context_freertos instance using the specified configuration.

If this method succeeds (returns true), then the async_context is available for use and can be de-initialized by calling async_context_deinit().

Parameters

self a pointer to async_context_freertos structure to initialize
config the configuration object specifying characteristics for the async_context

Returns

true if initialization is successful, false otherwise

async_context_freertos_init_with_defaults

static bool async_context_freertos_init_with_defaults(async_context_freertos_t *self) [inline], [static]

Initialize an async_context_freertos instance with default values.

If this method succeeds (returns true), then the async_context is available for use and can be de-initialized by calling async_context_deinit().

Parameters

self a pointer to async_context_freertos structure to initialize

Returns

true if initialization is successful, false otherwise

4.2.1.7.4. async_context_poll

Detailed Description

async_context_poll provides an implementation of async_context that is intended for use with a simple polling loop on one core. It is not thread safe.

The async_context_poll() method must be called periodically to handle asynchronous work that may now be pending. async_context_wait_for_work_until() may be used to block a polling loop until there is work to do, and prevent tight spinning.
Functions

```c
bool async_context_poll_init_with_defaults (async_context_poll_t *self)
```

Initialize an `async_context_poll` instance with default values.

Function Documentation

async_context_poll_init_with_defaults

```c
bool async_context_poll_init_with_defaults (async_context_poll_t * self)
```

Initialize an `async_context_poll` instance with default values.

If this method succeeds (returns true), then the `async_context` is available for use and can be de-initialized by calling `async_context_deinit()`.

Parameters

- `self` a pointer to `async_context_poll` structure to initialize

Returns

true if initialization is successful, false otherwise

async_context_threadsafe_background

Detailed Description

`async_context_threadsafe_background` provides an implementation of `async_context` that handles asynchronous work in a low priority IRQ, and there is no need for the user to poll for work.

⚠️ **NOTE**

The workers used with this `async_context` MUST be safe to call from an IRQ.

Functions

```c
bool async_context_threadsafe_background_init (async_context_threadsafe_background_t *self, async_context_threadsafe_background_config_t *config)
```

Initialize an `async_context_threadsafe_background` instance using the specified configuration.

```c
async_context_threadsafe_background_config_t async_context_threadsafe_background_default_config (void)
```

Return a copy of the default configuration object used by `async_context_threadsafe_background_init_with_defaults()`.

```c
static bool async_context_threadsafe_background_init_with_defaults (async_context_threadsafe_background_t *self)
```

Initialize an `async_context_threadsafe_background` instance with default values.

Function Documentation

async_context_threadsafe_background_default_config

```c
async_context_threadsafe_background_config_t async_context_threadsafe_background_default_config (void)
```

Return a copy of the default configuration object used by `async_context_threadsafe_background_init_with_defaults()`.

The caller can then modify just the settings it cares about, and call `async_context_threadsafe_background_init()`.

Returns

the default configuration object

async_context_threadsafe_background_init

```c
bool async_context_threadsafe_background_init (async_context_threadsafe_background_t * self, async_context_threadsafe_background_config_t * config)
```

Initialize an `async_context_threadsafe_background` instance using the specified configuration.
If this method succeeds (returns true), then the async_context is available for use and can be de-initialized by calling async_context_deinit().

Parameters

- `self` a pointer to async_context_threadsafe_background structure to initialize
- `config` the configuration object specifying characteristics for the async_context

Returns

ture if initialization is successful, false otherwise

```c
static bool async_context_threadsafe_background_init_with_defaults (async_context_threadsafe_background_t * self)
```

Initialize an async_context_threadsafe_background instance with default values.

If this method succeeds (returns true), then the async_context is available for use and can be de-initialized by calling async_context_deinit().

Parameters

- `self` a pointer to async_context_threadsafe_background structure to initialize

Returns

ture if initialization is successful, false otherwise

### 4.2.2. pico_flash

#### 4.2.2.1. Detailed Description

High level flash API

Flash cannot be erased or written to when in XIP mode. However the system cannot directly access memory in the flash address space when not in XIP mode.

It is therefore critical that no code or data is being read from flash while flash is been written or erased.

If only one core is being used, then the problem is simple - just disable interrupts; however if code is running on the other core, then it has to be asked, nicely, to avoid flash for a bit. This is hard to do if you don't have complete control of the code running on that core at all times.

This library provides a flash_safe_execute method which calls a function back having sucessfully gotten into a state where interrupts are disabled, and the other core is not executing or reading from flash.

How it does this is dependent on the supported environment (Free RTOS SMP or pico_multicore). Additionally the user can provide their own mechanism by providing a strong definition of get_flash_safety_helper().

Using the default settings, flash_safe_execute will only call the callback function if the state is safe otherwise returning an error (or an assert depending on PICO_FLASH_ASSERT_ON_UNSAFE).

There are conditions where safety would not be guaranteed:

1. FreeRTOS smp with `configNUM_CORES=1` - FreeRTOS still uses pico_multicore in this case, so flash_safe_execute cannot know what the other core is doing, and there is no way to force code execution between a FreeRTOS core and a non FreeRTOS core.
2. FreeRTOS non SMP with pico_multicore - Again, there is no way to force code execution between a FreeRTOS core and a non FreeRTOS core.
3. pico_multicore without flash_safe_execute_core_init() having been called on the other core - The flash_safe_execute method does not know if code is executing on the other core, so it has to assume it is. Either
way, it is not able to intervene if `flash_safe_execute_core_init()` has not been called on the other core. Fortunately, all is not lost in this situation, you may:

- Set PICO_FLASH_ASSUME_CORE0_SAFE=1 to explicitly say that core 0 is never using flash.
- Set PICO_FLASH_ASSUME_CORE1_SAFE=1 to explicitly say that core 1 is never using flash.

### 4.2.2.2. Functions

```c
bool flash_safe_execute_core_init (void)

bool flash_safe_execute_core_deinit (void)

int flash_safe_execute (void(*)(void *) func, void * param, uint32_t enter_exit_timeout_ms)

flash_safety_helper_t * get_flash_safety_helper (void)
```

#### 4.2.2.3. Function Documentation

##### 4.2.2.3.1. flash_safe_execute

```c
int flash_safe_execute (void(*)(void *) func, void * param, uint32_t enter_exit_timeout_ms)
```

Execute a function with IRQs disabled and with the other core also not executing/reading flash

**Parameters**

- `func` the function to call
- `param` the parameter to pass to the function
- `enter_exit_timeout_ms` the timeout for each of the enter/exit phases when coordinating with the other core

**Returns**

PICO_OK on success (the function will have been called). PICO_TIMEOUT on timeout (the function may have been called). PICO_ERROR_NOT_PERMITTED if safe execution is not possible (the function will not have been called). PICO_ERROR_INSUFFICIENT_RESOURCES if the method fails due to dynamic resource exhaustion (the function will not have been called).

**NOTE**

if PICO_FLASH_ASSERT_ON_UNSAFE is 1, this function will assert in debug mode vs returning PICO_ERROR_NOT_PERMITTED

##### 4.2.2.3.2. flash_safe_execute_core_deinit

```c
bool flash_safe_execute_core_deinit (void)
```

De-initialize work done by `flash_safe_execute_core_init`

**Returns**

true on success
4.2.2.3.3. flash_safe_execute_core_init

```c
bool flash_safe_execute_core_init (void)
```

Initialize a core such that the other core can lock it out during flash_safe_execute.

**NOTE**

This is not necessary for FreeRTOS SMP, but should be used when launching via multicore_launch_core1

Returns

true on success; there is no need to call flash_safe_execute_core_deinit() on failure.

4.2.2.3.4. get_flash_safety_helper

```c
flash_safety_helper_t * get_flash_safety_helper (void)
```

Internal method to return the flash safety helper implementation.

Advanced users can provide their own implementation of this function to perform different inter-core coordination before disabling XIP mode.

Returns

the flash_safety_helper_t

4.2.3. pico_i2c_slave

4.2.3.1. Detailed Description

Functions providing an interrupt driven I2C slave interface.

This I2C slave helper library configures slave mode and hooks the relevant I2C IRQ so that a user supplied handler is called with enumerated I2C events.

An example application slave_mem_i2c, which makes use of this library, can be found in pico_examples.

4.2.3.2. Typedefs

```c
typedef enum i2c_slave_event_t i2c_slave_event_t
```

I2C slave event types.

```c
typedef void(* i2c_slave_handler_t)(i2c_inst_t *i2c, i2c_slave_event_t event)
```

I2C slave event handler.

4.2.3.3. Enumerations

```c
enum i2c_slave_event_t { I2C_SLAVE_RECEIVE, I2C_SLAVE_REQUEST, I2C_SLAVE_FINISH }
```

I2C slave event types.

4.2.3.4. Functions
4.2.3.5. Typedef Documentation

4.2.3.5.1. i2c_slave_event_t

typedef enum i2c_slave_event_t i2c_slave_event_t

I2C slave event types.

4.2.3.5.2. i2c_slave_handler_t

typedef void(* i2c_slave_handler_t) (i2c_inst_t *i2c, i2c_slave_event_t event)

I2C slave event handler.

The event handler will run from the I2C ISR, so it should return quickly (under 25us at 400 kb/s). Avoid blocking inside the handler and split large data transfers across multiple calls for best results. When sending data to master, up to i2c_get_write_available() bytes can be written without blocking. When receiving data from master, up to i2c_get_read_available() bytes can be read without blocking.

Parameters

- **i2c**: Either i2c0 or i2c1
- **event**: Event type.

4.2.3.6. Enumeration Type Documentation

4.2.3.6.1. i2c_slave_event_t

enum i2c_slave_event_t

I2C slave event types.

4.2.3.7. Function Documentation

4.2.3.7.1. i2c_slave_deinit

void i2c_slave_deinit (i2c_inst_t *i2c)

Restore an I2C instance to master mode.

Parameters

- **i2c**: Either i2c0 or i2c1
4.2.3.7.2. i2c_slave_init

```c
void i2c_slave_init (i2c_inst_t * i2c, uint8_t address, i2c_slave_handler_t handler)
```

Configure an I2C instance for slave mode.

**Parameters**
- `i2c` I2C instance.
- `address` 7-bit slave address.
- `handler` Callback for events from I2C master. It will run from the I2C ISR, on the CPU core where the slave was initialised.

### 4.2.4. pico_multicore

#### 4.2.4.1. Detailed Description

Adds support for running code on the second processor core (core 1)

**Example**

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "pico/multicore.h"

#define FLAG_VALUE 123

void core1_entry () {
    multicore_fifo_push_blocking(FLAG_VALUE);
    uint32_t g = multicore_fifo_pop_blocking();
    if (g != FLAG_VALUE)
        printf("Hmm, that's not right on core 1!!
    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!");
    }
```
4.2.4.2. Modules

**fifo**
Functions for the inter-core FIFOs.

**lockout**
Functions to enable one core to force the other core to pause execution in a known state.

4.2.4.3. Functions

**void multicore_reset_core1 (void)**
Reset core 1.

**void multicore_launch_core1 (void(*)(void) entry)**
Run code on core 1.

**void multicore_launch_core1_with_stack (void(*)(void) entry, uint32_t *stack_bottom, size_t stack_size_bytes)**
Launch code on core 1 with stack.

**void multicore_launch_core1_raw (void(*)(void) entry, uint32_t *sp, uint32_t vector_table)**
Launch code on core 1 with no stack protection.

4.2.4.4. Function Documentation

4.2.4.4.1. multicore_launch_core1

**void multicore_launch_core1 (void(*)(void) entry)**
Run code on core 1.

Wake up (a previously reset) core 1 and enter the given function on core 1 using the default core 1 stack (below core 0 stack).

Core 1 must previously have been reset either as a result of a system reset or by calling multicore_reset_core1

Core 1 will use the same vector table as core 0

**Parameters**

- **entry** Function entry point

**See also**

multicore_reset_core1

4.2.4.4.2. multicore_launch_core1_raw

**void multicore_launch_core1_raw (void(*)(void) entry, uint32_t *sp, uint32_t vector_table)**
Launch code on core 1 with no stack protection.

Wake up (a previously reset) core 1 and start it executing with a specific entry point, stack pointer and vector table.

This is a low level function that does not provide a stack guard even if USE_STACK_GUARDS is defined
core 1 must previously have been reset either as a result of a system reset or by calling `multicore_reset_core1`

**Parameters**

- **entry** Function entry point
- **sp** Pointer to the top of the core 1 stack
- **vector_table** address of the vector table to use for core 1

**See also**

`multicore_reset_core1`

### 4.2.4.4.3. multicore_launch_core1_with_stack

```c
void multicore_launch_core1_with_stack (void(*)(void) entry, uint32_t * stack_bottom, size_t stack_size_bytes)
```

Launch code on core 1 with stack.

Wake up (a previously reset) core 1 and enter the given function on core 1 using the passed stack for core 1

core 1 must previously have been reset either as a result of a system reset or by calling `multicore_reset_core1`

core 1 will use the same vector table as core 0

**Parameters**

- **entry** Function entry point
- **stack_bottom** The bottom (lowest address) of the stack
- **stack_size_bytes** The size of the stack in bytes (must be a multiple of 4)

**See also**

`multicore_reset_core1`

### 4.2.4.4.4. multicore_reset_core1

```c
void multicore_reset_core1 (void)
```

Reset core 1.

This function can be used to reset core 1 into its initial state (ready for launching code against via `multicore_launch_core1` and similar methods)

**NOTE**

this function should only be called from core 0

### 4.2.4.5. fifo

Functions for the inter-core FIFOs.

#### 4.2.4.5.1. Detailed Description

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.
NOTE

The inter-core FIFOs are a very precious resource and are frequently used for SDK functionality (e.g. during core 1 launch or by the lockout functions). Additionally they are often required for the exclusive use of an RTOS (e.g. FreeRTOS SMP). For these reasons it is suggested that you do not use the FIFO for your own purposes unless none of the above concerns apply; the majority of cases for transferring data between cores can be equally well handled by using a queue.

4.2.4.5.2. Functions

static bool multicore_fifo_rvalid (void)
    Check the read FIFO to see if there is data available (sent by the other core)

static bool multicore_fifo_wready (void)
    Check the write FIFO to see if it has space for more data.

void multicore_fifo_push_blocking (uint32_t data)
    Push data on to the write FIFO (data to the other core).

boot multicore_fifo_push_timeout_us (uint32_t data, uint64_t timeout_us)
    Push data on to the write FIFO (data to the other core) with timeout.

uint32_t multicore_fifo_pop_blocking (void)
    Pop data from the read FIFO (data from the other core).

bool multicore_fifo_pop_timeout_us (uint64_t timeout_us, uint32_t *out)
    Pop data from the read FIFO (data from the other core) with timeout.

static void multicore_fifo_drain (void)
    Discard any data in the read FIFO.

static void multicore_fifo_clear_irq (void)
    Clear FIFO interrupt.

static uint32_t multicore_fifo_get_status (void)
    Get FIFO statuses.

4.2.4.5.3. Function Documentation

multicore_fifo_clear_irq

static void multicore_fifo_clear_irq (void) [inline], [static]
    Clear FIFO interrupt.

Note that this only clears an interrupt that was caused by the ROE or WOF flags. To clear the VLD flag you need to use one of the ‘pop’ or ‘drain’ functions.

See the note in the fifo section for considerations regarding use of the inter-core FIFOs.

See also

multicore_fifo_get_status

multicore_fifo_drain

static void multicore_fifo_drain (void) [inline], [static]
    Discard any data in the read FIFO.

See the note in the fifo section for considerations regarding use of the inter-core FIFOs.
**multicore_fifo_get_status**

static uint32_t multicore_fifo_get_status (void) [inline], [static]

Get FIFO statuses.

**Returns**

The statuses as a bitfield

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Sticky flag indicating the RX FIFO was read when empty (ROE). This read was ignored by the FIFO.</td>
</tr>
<tr>
<td>2</td>
<td>Sticky flag indicating the TX FIFO was written when full (WOF). This write was ignored by the FIFO.</td>
</tr>
<tr>
<td>1</td>
<td>Value is 1 if this core’s TX FIFO is not full (i.e. if FIFO_WR is ready for more data)</td>
</tr>
<tr>
<td>0</td>
<td>Value is 1 if this core’s RX FIFO is not empty (i.e. if FIFO_RD is valid)</td>
</tr>
</tbody>
</table>

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

**multicore_fifo_pop_blocking**

uint32_t multicore_fifo_pop_blocking (void)

Pop data from the read FIFO (data from the other core).

This function will block until there is data ready to be read Use multicore_fifo_rvalid() to check if data is ready to be read if you don’t want to block.

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

**Returns**

32 bit data from the read FIFO.

**multicore_fifo_pop_timeout_us**

bool multicore_fifo_pop_timeout_us (uint64_t timeout_us, uint32_t * out)

Pop data from the read FIFO (data from the other core) with timeout.

This function will block until there is data ready to be read or the timeout is reached

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

**Parameters**

- `timeout_us` the timeout in microseconds
- `out` the location to store the popped data if available

**Returns**

true if the data was popped and a value copied into out, false if the timeout occurred before data could be popped

**multicore_fifo_push_blocking**

void multicore_fifo_push_blocking (uint32_t data)

Push data on to the write FIFO (data to the other core).

This function will block until there is space for the data to be sent. Use multicore_fifo_wready() to check if it is possible to write to the FIFO if you don’t want to block.

See the note in the fifo section for considerations regarding use of the inter-core FIFOs
Parameters

**data**  A 32 bit value to push on to the FIFO

**multicore_fifo_push_timeout_us**

```c
bool multicore_fifo_push_timeout_us (uint32_t data, uint64_t timeout_us)
```

Push data on to the write FIFO (data to the other core) with timeout.

This function will block until there is space for the data to be sent or the timeout is reached.

**Parameters**

- **data**  A 32 bit value to push on to the FIFO
- **timeout_us**  the timeout in microseconds

**Returns**

true if the data was pushed, false if the timeout occurred before data could be pushed

**multicore_fifo_rvalid**

```c
static bool multicore_fifo_rvalid (void) [inline], [static]
```

Check the read FIFO to see if there is data available (sent by the other core).

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

**Returns**

true if the FIFO has data in it, false otherwise

**multicore_fifo_wready**

```c
static bool multicore_fifo_wready (void) [inline], [static]
```

Check the write FIFO to see if it has space for more data.

See the note in the fifo section for considerations regarding use of the inter-core FIFOs

**Returns**

true if the FIFO has room for more data, false otherwise

---

**4.2.4.5.4. lockout**

Functions to enable one core to force the other core to pause execution in a known state.

**Detailed Description**

Sometimes it is useful to enter a critical section on both cores at once. On a single core system a critical section can trivially be entered by disabling interrupts, however on a multi-core system that is not sufficient, and unless the other core is polling in some way, then it will need to be interrupted in order to cooperatively enter a blocked state.

These "lockout" functions use the inter core FIFOs to cause an interrupt on one core from the other, and manage waiting for the other core to enter the "locked out" state.

The usage is that the "victim" core ... i.e the core that can be 'locked out' by the other core calls **multicore_lockout_victim_init** to hook the FIFO interrupt. Note that either or both cores may do this.
NOTE

When 'locked out' the victim core is paused (it is actually executing a tight loop with code in RAM) and has interrupts disabled. This makes the lockout functions suitable for use by code that wants to write to flash (at which point no code may be executing from flash).

The core which wishes to lockout the other core calls `multicore_lockout_start_blocking` or `multicore_lockout_start_timeout_us` to interrupt the other "victim" core and wait for it to be in a "locked out" state. Once the lockout is no longer needed it calls `multicore_lockout_end_blocking` or `multicore_lockout_end_timeout_us` to release the lockout and wait for confirmation.

NOTE

Because multicore lockout uses the intercore FIFOs, the FIFOs cannot be used for any other purpose.

Functions

`void multicore_lockout_victim_init (void)`

Initialize the current core such that it can be a "victim" of lockout (i.e. forced to pause in a known state by the other core).

`bool multicore_lockout_victim_is_initialized (uint core_num)`

Determine if `multicore_victim_init()` has been called on the specified core.

`void multicore_lockout_start_blocking (void)`

Request the other core to pause in a known state and wait for it to do so.

`bool multicore_lockout_start_timeout_us (uint64_t timeout_us)`

Request the other core to pause in a known state and wait up to a time limit for it to do so.

`void multicore_lockout_end_blocking (void)`

Release the other core from a locked out state and wait for it to acknowledge.

`bool multicore_lockout_end_timeout_us (uint64_t timeout_us)`

Release the other core from a locked out state and wait up to a time limit for it to acknowledge.

Function Documentation

`multicore_lockout_end_blocking`

`void multicore_lockout_end_blocking (void)`

Release the other core from a locked out state and wait for it to acknowledge.

NOTE

The other core must previously have been "locked out" by calling a `multicore_lockout_start_` function from this core.

`multicore_lockout_end_timeout_us`

`bool multicore_lockout_end_timeout_us (uint64_t timeout_us)`

Release the other core from a locked out state and wait up to a time limit for it to acknowledge.

The other core must previously have been "locked out" by calling a `multicore_lockout_start_` function from this core.
NOTE

be very careful using small timeout values, as a timeout here will leave the 'lockout' functionality in a bad state. It is probably preferable to use `multicore_lockout_end_blocking` anyway as if you have already waited for the victim core to enter the lockout state, then the victim core will be ready to exit the lockout state very quickly.

Parameters

- `timeout_us` the timeout in microseconds

Returns

ture if the other core successfully exited locked out state within the timeout, false otherwise

`multicore_lockout_start_blocking`

```c
void multicore_lockout_start_blocking (void)
```

Request the other core to pause in a known state and wait for it to do so.

The other (victim) core must have previously executed `multicore_lockout_victim_init()`

NOTE

`multicore_lockout_start_` functions are not nestable, and must be paired with a call to a corresponding `multicore_lockout_end_blocking`

`multicore_lockout_start_timeout_us`

```c
bool multicore_lockout_start_timeout_us (uint64_t timeout_us)
```

Request the other core to pause in a known state and wait up to a time limit for it to do so.

The other core must have previously executed `multicore_lockout_victim_init()`

NOTE

`multicore_lockout_start_` functions are not nestable, and must be paired with a call to a corresponding `multicore_lockout_end_blocking`

Parameters

- `timeout_us` the timeout in microseconds

Returns

true if the other core entered the locked out state within the timeout, false otherwise

`multicore_lockout_victim_init`

```c
void multicore_lockout_victim_init (void)
```

Initialize the current core such that it can be a 'victim' of lockout (i.e. forced to pause in a known state by the other core)

This code hooks the intercore FIFO IRQ, and the FIFO may not be used for any other purpose after this.

`multicore_lockout_victim_is_initialized`

```c
bool multicore_lockout_victim_is_initialized (uint core_num)
```

Determine if `multicore_victim_init()` has been called on the specified core.
4.2 High Level APIs

4.2.5 pico_rand

4.2.5.1 Detailed Description

Random Number Generator API

This module generates random numbers at runtime utilizing a number of possible entropy sources and uses those sources to modify the state of a 128-bit ‘Pseudo Random Number Generator’ implemented in software.

The random numbers (32 to 128 bit) to be supplied are read from the PRNG which is used to help provide a large number space.

The following (multiple) sources of entropy are available (of varying quality), each enabled by a #define:

- The Ring Oscillator (ROSC) (PICO_RAND_ENTROPY_SRC_ROSC == 1): PICO_RAND_ROSC_BIT_SAMPLE_COUNT bits are gathered from the ring oscillator "random bit" and mixed in each time. This should not be used if the ROSC is off, or the processor is running from the ROSC.

  **NOTE**

  the maximum throughput of ROSC bit sampling is controlled by PICO_RAND_MIN_ROSC_BIT_SAMPLE_TIME_US which defaults to 10us, i.e. 100,000 bits per second.

- Time (PICO_RAND_ENTROPY_SRC_TIME == 1): The 64-bit microsecond timer is mixed in each time.

- Bus Performance Counter (PICO_RAND_ENTROPY_SRC_BUS_PERF_COUNTER == 1): One of the bus fabric's performance counters is mixed in each time.

**NOTE**

All entropy sources are hashed before application to the PRNG state machine.

The first time a random number is requested, the 128-bit PRNG state must be seeded. Multiple entropy sources are also available for the seeding operation:

- The Ring Oscillator (ROSC) (PICO_RAND_SEED_ENTROPY_SRC_ROSC == 1): 64 bits are gathered from the ring oscillator "random bit" and mixed into the seed.

- Time (PICO_RAND_SEED_ENTROPY_SRC_TIME == 1): The 64-bit microsecond timer is mixed into the seed.

- Board Identifier (PICO_RAND_SEED_ENTROPY_SRC_BOARD_ID == 1): The board id via pico_get_unique_board_id is mixed into the seed.

- RAM hash (PICO_RAND_SEED_ENTROPY_SRC_RAM_HASH (PICO_RAND_SEED_ENTROPY_SRC_RAM_HASH): The hashed contents of a subset of RAM are mixed in. Initial RAM contents are undefined on power up, so provide a reasonable source of entropy. By default the last 1K of RAM (which usually contains the core 0 stack) is hashed, which may also provide for differences after each warm reset.
With default settings, the seed generation takes approximately 1 millisecond while subsequent random numbers generally take between 10 and 20 microseconds to generate.

pico_rand methods may be safely called from either core or from an IRQ, but be careful in the latter case as the calls may block for a number of microseconds waiting on more entropy.

### 4.2.5.2. Functions

```c
void get_rand_128 (rng_128_t * rand128)
```

Get 128-bit random number.

```c
uint64_t get_rand_64 (void)
```

Get 64-bit random number.

```c
uint32_t get_rand_32 (void)
```

Get 32-bit random number.

### 4.2.5.3. Function Documentation

#### 4.2.5.3.1. get_rand_128

```c
void get_rand_128 (rng_128_t * rand128)
```

Get 128-bit random number.

This method may be safely called from either core or from an IRQ, but be careful in the latter case as the call may block for a number of microseconds waiting on more entropy.

**Parameters**

- `rand128` Pointer to storage to accept a 128-bit random number

#### 4.2.5.3.2. get_rand_32

```c
uint32_t get_rand_32 (void)
```

Get 32-bit random number.

This method may be safely called from either core or from an IRQ, but be careful in the latter case as the call may block for a number of microseconds waiting on more entropy.

**Returns**

32-bit random number

#### 4.2.5.3.3. get_rand_64

```c
uint64_t get_rand_64 (void)
```

Get 64-bit random number.

This method may be safely called from either core or from an IRQ, but be careful in the latter case as the call may block for a number of microseconds waiting on more entropy.

**Returns**

64-bit random number
4.2.6. pico_stdlib

4.2.6.1. Detailed Description

Aggregation of a core subset of Raspberry Pi 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.

4.2.6.2. Functions

```c
void setup_default_uart (void)
```
Set up the default UART and assign it to the default GPIOs.

```c
void set_sys_clock_48mhz (void)
```
Initialise the system clock to 48MHz.

```c
void set_sys_clock_pll (uint32_t vco_freq, uint post_div1, uint post_div2)
```
Initialise the system clock.

```c
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.

```c
static bool set_sys_clock_khz (uint32_t freq_khz, bool required)
```
Attempt to set a system clock frequency in khz.

4.2.6.3. Function Documentation

4.2.6.3.1. check_sys_clock_khz

```c
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
On success, the voltage controlled oscillator frequency to be used by the SYS PLL.

On success, The first post divider for the SYS PLL.

On success, The second post divider for the SYS PLL.

Returns
true if the frequency is possible and the output parameters have been written.

4.2.6.3.2. set_sys_clock_48mhz

```c
void set_sys_clock_48mhz (void)
```
Initialise the system clock to 48MHz.
Set the system clock to 48MHz, and set the peripheral clock to match.

4.2.6.3.3. set_sys_clock_khz

```c
static bool set_sys_clock_khz (uint32_t freq_khz, bool required) [inline], [static]
```
Attempt to set a system clock frequency in khz.

Note that not all clock frequencies are possible; it is preferred that you use src/rp2_common/hardware_clocks/scripts/vcocalc.py to calculate the parameters for use with set_sys_clock_pll

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

Returns
true if the clock was configured

4.2.6.3.4. set_sys_clock_pll

```c
void set_sys_clock_pll (uint32_t vco_freq, uint post_div1, uint post_div2)
```
Initialise the system clock.

Parameters
- `vco_freq` The voltage controller oscillator frequency to be used by the SYS PLL
- `post_div1` The first post divider for the SYS PLL
- `post_div2` The second post divider for the SYS PLL.

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

4.2.6.3.5. setup_default_uart

```c
void setup_default_uart (void)
```
Set up the default UART and assign it to the default GPIOs.
By default this will use UART 0, with TX to pin GPIO 0, RX to pin GPIO 1, and the baudrate to 115200
Calling this method also initializes stdin/stdout over UART if the pico_stdio_uart library is linked.

Defaults can be changed using configuration defines, PICO_DEFAULT_UART_INSTANCE, PICO_DEFAULT_UART_BAUD_RATE PICO_DEFAULT_UART_TX_PIN PICO_DEFAULT_UART_RX_PIN
4.2.7. pico_sync

4.2.7.1. Detailed Description

Synchronization primitives and mutual exclusion

4.2.7.2. Modules

**critical_section**
- Critical Section API for short-lived mutual exclusion safe for IRQ and multi-core.

**lock_core**
- base synchronization/lock primitive support

**mutex**
- Mutex API for non IRQ mutual exclusion between cores.

**sem**
- Semaphore API for restricting access to a resource.

4.2.7.3. critical_section

Critical Section API for short-lived mutual exclusion safe for IRQ and multi-core.

4.2.7.3.1. Detailed Description

A critical section is non-reentrant, and provides mutual exclusion using a spin-lock to prevent access from the other core, and from (higher priority) interrupts on the same core. It does the former using a spin lock and the latter by disabling interrupts on the calling core.

Because interrupts are disabled when a `critical_section` is owned, uses of the `critical_section` should be as short as possible.

4.2.7.3.2. Functions

```c
void critical_section_init (critical_section_t *crit_sec)
    Initialise a `critical_section` structure allowing the system to assign a spin lock number.

void critical_section_init_with_lock_num (critical_section_t *crit_sec, uint lock_num)
    Initialise a `critical_section` structure assigning a specific spin lock number.

static void critical_section_enter_blocking (critical_section_t *crit_sec)
    Enter a `critical_section`.

static void critical_section_exit (critical_section_t *crit_sec)
    Release a `critical_section`.

void critical_section_deinit (critical_section_t *crit_sec)
    De-Initialise a `critical_section` created by the critical_section_init method.
```
4.2.7.3.3. Function Documentation

**critical_section_deinit**

```c
void critical_section_deinit (critical_section_t * crit_sec)
```

De-initialise a `critical_section` created by the `critical_section_init` method.

This method is only used to free the associated spin lock allocated via the `critical_section_init` method (it should not be used to de-initialize a spin lock created via `critical_section_init_with_lock_num`). After this call, the critical section is invalid.

**Parameters**
- `crit_sec` Pointer to `critical_section` structure

**critical_section_enter_blocking**

```c
static void critical_section_enter_blocking (critical_section_t * crit_sec) [inline], [static]
```

Enter a `critical_section`.

If the spin lock associated with this critical section is in use, then this method will block until it is released.

**Parameters**
- `crit_sec` Pointer to `critical_section` structure

**critical_section_exit**

```c
static void critical_section_exit (critical_section_t * crit_sec) [inline], [static]
```

Release a `critical_section`.

**Parameters**
- `crit_sec` Pointer to `critical_section` structure

**critical_section_init**

```c
void critical_section_init (critical_section_t * crit_sec)
```

Initialise a `critical_section` structure allowing the system to assign a spin lock number.

The critical section is initialized ready for use, and will use a (possibly shared) spin lock number assigned by the system. Note that in general it is unlikely that you would be nesting critical sections, however if you do so you must use `critical_section_init_with_lock_num` to ensure that the spin locks used are different.

**Parameters**
- `crit_sec` Pointer to `critical_section` structure

**critical_section_init_with_lock_num**

```c
void critical_section_init_with_lock_num (critical_section_t * crit_sec, uint lock_num)
```

Initialise a `critical_section` structure assigning a specific spin lock number.

**Parameters**
- `crit_sec` Pointer to `critical_section` structure
- `lock_num` the specific spin lock number to use

4.2.7.3.4. lock_core

base synchronization/lock primitive support

**Detailed Description**

Most of the pico_sync locking primitives contain a `lock_core_t` structure member. This currently just holds a spin lock
which is used only to protect the contents of the rest of the structure as part of implementing the synchronization primitive. As such, the spin_lock member of lock_core is never still held on return from any function for the primitive.

critical_section is an exceptional case in that it does not have a lock_core_t and simply wraps a spin lock, providing methods to lock and unlock said spin lock.

lock_core based structures work by locking the spin lock, checking state, and then deciding whether they additionally need to block or notify when the spin lock is released. In the blocking case, they will wake up again in the future, and try the process again.

By default the SDK just uses the processors' events via SEV and WEV for notification and blocking as these are sufficient for cross core, and notification from interrupt handlers. However macros are defined in this file that abstract the wait and notify mechanisms to allow the SDK locking functions to effectively be used within an RTOS or other environment.

When implementing an RTOS, it is desirable for the SDK synchronization primitives that wait, to block the calling task (and immediately yield), and those that notify, to wake a blocked task which isn't on processor. At least the wait macro implementation needs to be atomic with the protecting spin_lock unlock from the callers point of view; i.e. the task should unlock the spin lock when it starts its wait. Such implementation is up to the RTOS integration, however the macros are defined such that such operations are always combined into a single call (so they can be performed atomically) even though the default implementation does not need this, as a WFE which starts following the corresponding SEV is not missed.

Macros

- `#define lock_owner_id_t int8_t`
- `#define LOCK_INVALID_OWNER_ID ((lock_owner_id_t)-1)`
- `#define lock_get_caller_owner_id() ((lock_owner_id_t)get_core_num())`
- `#define lock_internal_spin_unlock_with_wait(lock, save) spin_unlock((lock)->spin_lock, save), __wfe()`
- `#define lock_internal_spin_unlock_with_notify(lock, save) spin_unlock((lock)->spin_lock, save), __sev()`
- `#define lock_internal_spin_unlock_with_best_effort_wait_or_timeout(lock, save, until)`
- `#define sync_internal_yield_until_before(until) ((void)0)`

Functions

`void lock_init(lock_core_t *core, uint lock_num)`

Initialise a lock structure.

Macro Definition Documentation

lock_owner_id_t

`#define lock_owner_id_t int8_t`

type to use to store the 'owner' of a lock.

By default this is int8_t as it only needs to store the core number or -1, however it may be overridden if a larger type is required (e.g. for an RTOS task id)

LOCK_INVALID_OWNER_ID

`#define LOCK_INVALID_OWNER_ID ((lock_owner_id_t)-1)`

marker value to use for a lock_owner_id_t which does not refer to any valid owner

lock_get_caller_owner_id

`#define lock_get_caller_owner_id() ((lock_owner_id_t)get_core_num())`

return the owner id for the caller

By default this returns the calling core number, but may be overridden (e.g. to return an RTOS task id)
lock_internal_spin_unlock_with_wait

#define lock_internal_spin_unlock_with_wait(lock, save) spin_unlock((lock)->spin_lock, save), __wfe()

Atomically unlock the lock's spin lock, and wait for a notification.

Atomic here refers to the fact that it should not be possible for a concurrent lock_internal_spin_unlock_with_notify to insert itself between the spin unlock and this wait in a way that the wait does not see the notification (i.e. causing a missed notification). In other words this method should always wake up in response to a lock_internal_spin_unlock_with_notify for the same lock, which completes after this call starts.

In an ideal implementation, this method would return exactly after the corresponding lock_internal_spin_unlock_with_notify has subsequently been called on the same lock instance, however this method is free to return at any point before that; this macro is always used in a loop which locks the spin lock, checks the internal locking primitive state and then waits again if the calling thread should not proceed.

By default this macro simply unlocks the spin lock, and then performs a WFE, but may be overridden (e.g. to actually block the RTOS task).

Parameters

lock the lock_core for the primitive which needs to block

save the uint32_t value that should be passed to spin_unlock when the spin lock is unlocked. (i.e. the PRIMASK state when the spin lock was acquire)

lock_internal_spin_unlock_with_notify

#define lock_internal_spin_unlock_with_notify(lock, save) spin_unlock((lock)->spin_lock, save), __sev()

Atomically unlock the lock's spin lock, and send a notification.

Atomic here refers to the fact that it should not be possible for this notification to happen during a lock_internal_spin_unlock_with_wait in a way that that wait does not see the notification (i.e. causing a missed notification). In other words this method should always wake up any lock_internal_spin_unlock_with_wait which started before this call completes.

In an ideal implementation, this method would wake up only the corresponding lock_internal_spin_unlock_with_wait that has been called on the same lock instance, however it is free to wake up any of them, as they will check their condition and then re-wait if necessary.

By default this macro simply unlocks the spin lock, and then performs a SEV, but may be overridden (e.g. to actually unblock RTOS task(s)).

Parameters

lock the lock_core for the primitive which needs to block

save the uint32_t value that should be passed to spin_unlock when the spin lock is unlocked. (i.e. the PRIMASK state when the spin lock was acquire)

lock_internal_spin_unlock_with_best_effort_wait_or_timeout

#define lock_internal_spin_unlock_with_best_effort_wait_or_timeout(lock, save, until) ({
    spin_unlock((lock)->spin_lock, save);
    best_effort_wfe_or_timeout(until);
})

Atomically unlock the lock's spin lock, and wait for a notification or a timeout.

Atomic here refers to the fact that it should not be possible for a concurrent lock_internal_spin_unlock_with_notify to insert itself between the spin unlock and this wait in a way that the wait does not see the notification (i.e. causing a missed notification). In other words this method should always wake up in response to a lock_internal_spin_unlock_with_notify for the same lock, which completes after this call starts.
In an ideal implementation, this method would return exactly after the corresponding
lock_internal_spin_unlock_with_notify has subsequently been called on the same lock instance or the timeout has been
reached, however this method is free to return at any point before that; this macro is always used in a loop which locks
the spin lock, checks the internal locking primitive state and then waits again if the calling thread should not proceed.

By default this simply unlocks the spin lock, and then calls best_effort_wfe_or_timeout but may be overridden (e.g. to
actually block the RTOS task with a timeout).

**Parameters**

- `lock` the lock_core for the primitive which needs to block
- `save` the uint32_t value that should be passed to spin_unlock when the spin lock is unlocked. (i.e. the
  PRIMASK state when the spin lock was acquire)
- `until` the absolute_time_t value

**Returns**

true if the timeout has been reached

```
#define sync_internal_yield_until_before(until) ((void)0)
```

This method is provided for cases where the caller has no useful work to do until the specified time.

By default this method does nothing, however it can be overridden (for example by an RTOS which is able to block the
current task until the scheduler tick before the given time)

**Parameters**

- `until` the absolute_time_t value

**Function Documentation**

**lock_init**

```
void lock_init (lock_core_t * core, uint lock_num)
```

Initialise a lock structure.

**Parameters**

- `core` Pointer to the lock_core to initialize
- `lock_num` Spin lock number to use for the lock. As the spin lock is only used internally to the locking primitive
  method implementations, this does not need to be globally unique, however could suffer contention

**mutex**

Mutex API for non IRQ mutual exclusion between cores.

**Detailed Description**

Mutexes are application level locks usually used protecting data structures that might be used by multiple threads of
execution. Unlike critical sections, the mutex protected code is not necessarily required/expected to complete quickly,
as no other system wide locks are held on account of an acquired mutex.

When acquired, the mutex has an owner (see lock_get_caller_owner_id) which with the plain SDK is just the acquiring
core, but in an RTOS it could be a task, or an IRQ handler context.

Two variants of mutex are provided; mutex_t (and associated mutex_ functions) is a regular mutex that cannot be
acquired recursively by the same owner (a deadlock will occur if you try). recursive_mutex_t (and associated
recursive_mutex_ functions) is a recursive mutex that can be recursively obtained by the same caller, at the expense of
some more overhead when acquiring and releasing.
It is generally a bad idea to call blocking mutex_ or recursive_mutex_ functions from within an IRQ handler. It is valid to call mutex_try_enter or recursive_mutex_try_enter from within an IRQ handler, if the operation that would be conducted under lock can be skipped if the mutex is locked (at least by the same owner).

**NOTE**

For backwards compatibility with version 1.2.0 of the SDK, if the define PICO_MUTEX_ENABLE_SDK120_COMPATIBILITY is set to 1, then the the regular mutex_ functions may also be used for recursive mutexes. This flag will be removed in a future version of the SDK.

See critical_section.h for protecting access between multiple cores AND IRQ handlers.

### Macros

- #define auto_init_mutex(name) static __attribute__((section(".mutex_array"))) mutex_t name
- #define auto_init_recursive_mutex(name) static __attribute__((section(".mutex_array"))) recursive_mutex_t name = {
  .core = { .spin_lock = (spin_lock_t *)1 /* marker for runtime_init */ }, .owner = 0, .enter_count = 0 }

### Typedefs

typedef struct __packed_aligned recursive_mutex_t
  recursive mutex instance

typedef struct __packed_aligned mutex mutex_t
  regular (non recursive) mutex instance

### Functions

static bool critical_section_is_initialized (critical_section_t *crit_sec)
  Test whether a critical_section has been initialized.
void mutex_init (mutex_t *mtx)
  Initialise a mutex structure.
void recursive_mutex_init (recursive_mutex_t *mtx)
  Initialise a recursive mutex structure.
void mutex_enter_blocking (mutex_t *mtx)
  Take ownership of a mutex.
void recursive_mutex_enter_blocking (recursive_mutex_t *mtx)
  Take ownership of a recursive mutex.
bool mutex_try_enter (mutex_t *mtx, uint32_t *owner_out)
  Attempt to take ownership of a mutex.
bool mutex_try_enter_block_until (mutex_t *mtx, absolute_time_t until)
  Attempt to take ownership of a mutex until the specified time.
bool recursive_mutex_try_enter (recursive_mutex_t *mtx, uint32_t *owner_out)
  Attempt to take ownership of a recursive mutex.
bool mutex_enter_timeout_ms (mutex_t *mtx, uint32_t timeout_ms)
  Wait for mutex with timeout.
bool recursive_mutex_enter_timeout_ms (recursive_mutex_t *mtx, uint32_t timeout_ms)
  Wait for recursive mutex with timeout.
bool mutex_enter_timeout_us (mutex_t *mtx, uint32_t timeout_us)
  Wait for mutex with timeout.
bool recursive_mutex_enter_timeout_us (recursive_mutex_t *mtx, uint32_t timeout_us)

Wait for recursive mutex with timeout.

bool mutex_enter_block_until (mutex_t *mtx, absolute_time_t until)

Wait for mutex until a specific time.

bool recursive_mutex_enter_block_until (recursive_mutex_t *mtx, absolute_time_t until)

Wait for mutex until a specific time.

void mutex_exit (mutex_t *mtx)

Release ownership of a mutex.

void recursive_mutex_exit (recursive_mutex_t *mtx)

Release ownership of a recursive mutex.

static bool mutex_is_initialized (mutex_t *mtx)

Test for mutex initialized state.

static bool recursive_mutex_is_initialized (recursive_mutex_t *mtx)

Test for recursive mutex initialized state.

**Macro Definition Documentation**

**auto_init_mutex**

#define auto_init_mutex(name) static __attribute__((section(".mutex_array"))) mutex_t name

Helper macro for static definition of mutexes.

A mutex defined as follows:

```c
1 auto_init_mutex(my_mutex);
```

Is equivalent to doing

```c
1 static mutex_t my_mutex;
2 3 void my_init_function() {
4 5 mutex_init(&my_mutex);
6 }
```

But the initialization of the mutex is performed automatically during runtime initialization

**auto_init_recursive_mutex**

#define auto_init_recursive_mutex(name) static __attribute__((section(".mutex_array"))) recursive_mutex_t name = { .core = { .spin_lock = (spin_lock_t *)1 /* marker for runtime_init */ }, .owner = 0, .enter_count = 0 }

Helper macro for static definition of recursive mutexes.

A recursive mutex defined as follows:

```c
1 auto_init_recursive_mutex(my_recursive_mutex);
```

Is equivalent to doing

```c
1 auto_init_recursive_mutex(my_recursive_mutex);
```


```c
static recursive_mutex_t my_recursive_mutex;

void my_init_function()
{
    recursive_mutex_init(&my_recursive_mutex);
}
```

But the initialization of the mutex is performed automatically during runtime initialization

**Typedef Documentation**

`recursive_mutex_t`

typedef struct __packed_aligned recursive_mutex_t
recursive mutex instance

`mutex_t`

typedef struct __packed_aligned mutex mutex_t
regular (non recursive) mutex instance

**Function Documentation**

`critical_section_is_initialized`

static bool critical_section_is_initialized (critical_section_t * crit_sec) [inline], [static]
Test whether a `critical_section` has been initialized.

**Parameters**

- `crit_sec` Pointer to `critical_section` structure

**Returns**

true if the critical section is initialized, false otherwise

`mutex_enter_block_until`

bool mutex_enter_block_until (mutex_t * mtx, absolute_time_t until)
Wait for mutex until a specific time.

Wait until the specific time to take ownership of the mutex. If the caller can be granted ownership of the mutex before the timeout expires, then true will be returned and the caller will own the mutex, otherwise false will be returned and the caller will NOT own the mutex.

**Parameters**

- `mtx` Pointer to mutex structure
- `until` The time after which to return if the caller cannot be granted ownership of the mutex

**Returns**

true if mutex now owned, false if timeout occurred before ownership could be granted

`mutex_enter_blocking`

void mutex_enter_blocking (mutex_t * mtx)
Take ownership of a mutex.

This function will block until the caller can be granted ownership of the mutex. On return the caller owns the mutex

**Parameters**

- `mtx` Pointer to mutex structure

4.2. High Level APIs
mutex_enter_timeout_ms

bool mutex_enter_timeout_ms (mutex_t * mtx, uint32_t timeout_ms)

Wait for mutex with timeout.

Wait for up to the specific time to take ownership of the mutex. If the caller can be granted ownership of the mutex before the timeout expires, then true will be returned and the caller will own the mutex, otherwise false will be returned and the caller will NOT own the mutex.

Parameters

mtx Pointer to mutex structure

timeout_ms The timeout in milliseconds.

Returns

ture if mutex now owned, false if timeout occurred before ownership could be granted

mutex_enter_timeout_us

bool mutex_enter_timeout_us (mutex_t * mtx, uint32_t timeout_us)

Wait for mutex with timeout.

Wait for up to the specific time to take ownership of the mutex. If the caller can be granted ownership of the mutex before the timeout expires, then true will be returned and the caller will own the mutex, otherwise false will be returned and the caller will NOT own the mutex.

Parameters

mtx Pointer to mutex structure

timeout_us The timeout in microseconds.

Returns

ture if mutex now owned, false if timeout occurred before ownership could be granted

mutex_exit

void mutex_exit (mutex_t * mtx)

Release ownership of a mutex.

Parameters

mutex Pointer to mutex structure

mutex_init

void mutex_init (mutex_t * mtx)

Initialise a mutex structure.

Parameters

mutex Pointer to mutex structure

mutex_is_initialized

static bool mutex_is_initialized (mutex_t * mtx) [inline], [static]

Test for mutex initialized state.

Parameters

mutex Pointer to mutex structure

Returns

ture if the mutex is initialized, false otherwise
mutex_try_enter

bool mutex_try_enter (mutex_t * mtx, uint32_t * owner_out)

Attempt to take ownership of a mutex.

If the mutex wasn't owned, this will claim the mutex for the caller and return true. Otherwise (if the mutex was already owned) this will return false and the caller will NOT own the mutex.

Parameters

mtx - Pointer to mutex structure
owner_out - If mutex was already owned, and this pointer is non-zero, it will be filled in with the owner id of the current owner of the mutex

Returns

ture if mutex now owned, false otherwise

mutex_try_enter_block_until

bool mutex_try_enter_block_until (mutex_t * mtx, absolute_time_t until)

Attempt to take ownership of a mutex until the specified time.

If the mutex wasn't owned, this method will immediately claim the mutex for the caller and return true. If the mutex is owned by the caller, this method will immediately return false. If the mutex is owned by someone else, this method will try to claim it until the specified time, returning true if it succeeds, or false on timeout

Parameters

mtx - Pointer to mutex structure
until - The time after which to return if the caller cannot be granted ownership of the mutex

Returns

ture if mutex now owned, false otherwise

recursive_mutex_enter_block_until

bool recursive_mutex_enter_block_until (recursive_mutex_t * mtx, absolute_time_t until)

Wait for mutex until a specific time.

Wait until the specific time to take ownership of the mutex. If the caller already has ownership of the mutex or can be granted ownership of the mutex before the timeout expires, then true will be returned and the caller will own the mutex, otherwise false will be returned and the caller will NOT own the mutex.

Parameters

mtx - Pointer to recursive mutex structure
until - The time after which to return if the caller cannot be granted ownership of the mutex

Returns

ture if the recursive mutex (now) owned, false if timeout occurred before ownership could be granted

recursive_mutex_enter_blocking

void recursive_mutex_enter_blocking (recursive_mutex_t * mtx)

Take ownership of a recursive mutex.

This function will block until the caller can be granted ownership of the mutex. On return the caller owns the mutex

Parameters

mtx - Pointer to recursive mutex structure

recursive_mutex_enter_timeout_ms
bool recursive_mutex_enter_timeout_ms (recursive_mutex_t * mtx, uint32_t timeout_ms)

Wait for recursive mutex with timeout.

Wait for up to the specific time to take ownership of the recursive mutex. If the caller already has ownership of the mutex or can be granted ownership of the mutex before the timeout expires, then true will be returned and the caller will own the mutex, otherwise false will be returned and the caller will NOT own the mutex.

Parameters

- **mtx**  
  Pointer to recursive mutex structure
- **timeout_ms**  
  The timeout in milliseconds.

Returns

true if the recursive mutex (now) owned, false if timeout occurred before ownership could be granted

bool recursive_mutex_enter_timeout_us (recursive_mutex_t * mtx, uint32_t timeout_us)

Wait for recursive mutex with timeout.

Wait for up to the specific time to take ownership of the recursive mutex. If the caller already has ownership of the mutex or can be granted ownership of the mutex before the timeout expires, then true will be returned and the caller will own the mutex, otherwise false will be returned and the caller will NOT own the mutex.

Parameters

- **mtx**  
  Pointer to mutex structure
- **timeout_us**  
  The timeout in microseconds.

Returns

true if the recursive mutex (now) owned, false if timeout occurred before ownership could be granted

void recursive_mutex_exit (recursive_mutex_t * mtx)

Release ownership of a recursive mutex.

Parameters

- **mtx**  
  Pointer to recursive mutex structure

recursive_mutex_init

void recursive_mutex_init (recursive_mutex_t * mtx)

Initialise a recursive mutex structure.

A recursive mutex may be entered in a nested fashion by the same owner.

Parameters

- **mtx**  
  Pointer to recursive mutex structure

recursive_mutex_is_initialized

static bool recursive_mutex_is_initialized (recursive_mutex_t * mtx) [inline], [static]

Test for recursive mutex initialized state.

Parameters

- **mtx**  
  Pointer to recursive mutex structure

Returns

true if the recursive mutex is initialized, false otherwise
**recursive_mutex_try_enter**

```c
bool recursive_mutex_try_enter (recursive_mutex_t * mtx, uint32_t * owner_out)
```

Attempt to take ownership of a recursive mutex.

If the mutex wasn't owned or was owned by the caller, this will claim the mutex and return true. Otherwise (if the mutex was already owned by another owner) this will return false and the caller will NOT own the mutex.

**Parameters**

- **mtx**: Pointer to recursive mutex structure
- **owner_out**: If mutex was already owned by another owner, and this pointer is non-zero, it will be filled in with the owner id of the current owner of the mutex

**Returns**

true if the recursive mutex (now) owned, false otherwise

**sem**

Semaphore API for restricting access to a resource.

**Detailed Description**

A semaphore holds a number of available permits. `sem_acquire` methods will acquire a permit if available (reducing the available count by 1) or block if the number of available permits is 0. `sem_release()` increases the number of available permits by one potentially unblocking a `sem_acquire` method.

Note that `sem_release()` may be called an arbitrary number of times, however the number of available permits is capped to the max_permits 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)

**Functions**

```c
void sem_init (semaphore_t *sem, int16_t initial_permits, int16_t max_permits)
```

Initialise a semaphore structure.

```c
int sem_available (semaphore_t *sem)
```

Return number of available permits on the semaphore.

```c
bool sem_release (semaphore_t *sem)
```

Release a permit on a semaphore.

```c
void sem_reset (semaphore_t *sem, int16_t permits)
```

Reset semaphore to a specific number of available permits.

```c
void sem_acquire_blocking (semaphore_t *sem)
```

Acquire a permit from the semaphore.

```c
bool sem_acquire_timeout_ms (semaphore_t *sem, uint32_t timeout_ms)
```

Acquire a permit from a semaphore, with timeout.

```c
bool sem_acquire_timeout_us (semaphore_t *sem, uint32_t timeout_us)
```

Acquire a permit from a semaphore, with timeout.

```c
bool sem_acquire_block_until (semaphore_t *sem, absolute_time_t until)
```

Wait to acquire a permit from a semaphore until a specific time.

```c
bool sem_try_acquire (semaphore_t *sem)
```

Attempt to acquire a permit from a semaphore without blocking.

**Function Documentation**
sem_acquire_block_until

bool sem_acquire_block_until (semaphore_t * sem, absolute_time_t until)

Wait to acquire a permit from a semaphore until a specific time.
This function will block and wait if no permits are available, until the specified timeout time. If the timeout is reached the function will return false, otherwise it will return true.

Parameters

- **sem** Pointer to semaphore structure
- **until** The time after which to return if the sem is not available.

Returns

true if permit was acquired, false if the until time was reached before acquiring.

sem_acquire_blocking

void sem_acquire_blocking (semaphore_t * sem)

Acquire a permit from the semaphore.
This function will block and wait if no permits are available.

Parameters

- **sem** Pointer to semaphore structure

sem_acquire_timeout_ms

bool sem_acquire_timeout_ms (semaphore_t * sem, uint32_t timeout_ms)

Acquire a permit from a semaphore, with timeout.
This function will block and wait if no permits are available, until the defined timeout has been reached. If the timeout is reached the function will return false, otherwise it will return true.

Parameters

- **sem** Pointer to semaphore structure
- **timeout_ms** Time to wait to acquire the semaphore, in milliseconds.

Returns

false if timeout reached, true if permit was acquired.

sem_acquire_timeout_us

bool sem_acquire_timeout_us (semaphore_t * sem, uint32_t timeout_us)

Acquire a permit from a semaphore, with timeout.
This function will block and wait if no permits are available, until the defined timeout has been reached. If the timeout is reached the function will return false, otherwise it will return true.

Parameters

- **sem** Pointer to semaphore structure
- **timeout_us** Time to wait to acquire the semaphore, in microseconds.

Returns

false if timeout reached, true if permit was acquired.

sem_available

int sem_available (semaphore_t * sem)

Return number of available permits on the semaphore.
Parameters

sem  Pointer to semaphore structure

Returns

The number of permits available on the semaphore.

sem_init

void sem_init (semaphore_t * sem, int16_t initial_permits, int16_t max_permits)

Initialise a semaphore structure.

Parameters

sem  Pointer to semaphore structure

initial_permits  How many permits are initially acquired

max_permits  Total number of permits allowed for this semaphore

sem_release

bool sem_release (semaphore_t * sem)

Release a permit on a semaphore.

Increases the number of permits by one (unless the number of permits is already at the maximum). A blocked
sem_acquire will be released if the number of permits is increased.

Parameters

sem  Pointer to semaphore structure

Returns

true if the number of permits available was increased.

sem_reset

void sem_reset (semaphore_t * sem, int16_t permits)

Reset semaphore to a specific number of available permits.

Reset value should be from 0 to the max_permits specified in the init function.

Parameters

sem  Pointer to semaphore structure

permits  the new number of available permits

sem_try_acquire

bool sem_try_acquire (semaphore_t * sem)

Attempt to acquire a permit from a semaphore without blocking.

This function will return false without blocking if no permits are available, otherwise it will acquire a permit and return true.

Parameters

sem  Pointer to semaphore structure

Returns

true if permit was acquired.
4.2.8. pico_time

4.2.8.1. Detailed Description

API for accurate timestamps, sleeping, and time based callbacks

**NOTE**

The functions defined here provide a much more powerful and user friendly wrapping around the low level hardware timer functionality. For these functions (and any other SDK functionality e.g. timeouts, that relies on them) to work correctly, the hardware timer should not be modified. i.e. it is expected to be monotonically increasing once per microsecond. Fortunately there is no need to modify the hardware timer as any functionality you can think of that isn’t already covered here can easily be modelled by adding or subtracting a constant value from the unmodified hardware timer.

See also

hardware_timer

4.2.8.2. Modules

**timestamp**

Timestamp functions relating to points in time (including the current time)

**sleep**

Sleep functions for delaying execution in a lower power state.

**alarm**

Alarm functions for scheduling future execution.

**repeating_timer**

Repeating Timer functions for simple scheduling of repeated execution.

4.2.8.3. timestamp

Timestamp functions relating to points in time (including the current time)

4.2.8.3.1. Detailed Description

These are functions for dealing with timestamps (i.e. instants in time) represented by the type `absolute_time_t`. This opaque type is provided to help prevent accidental mixing of timestamps and relative time values.

4.2.8.3.2. Functions

```c
static uint64_t to_us_since_boot (absolute_time_t t)
    convert an absolute_time_t into a number of microseconds since boot.

static void update_us_since_boot (absolute_time_t *t, uint64_t us_since_boot)
    update an absolute_time_t value to represent a given number of microseconds since boot

static absolute_time_t from_us_since_boot (uint64_t us_since_boot)
    convert a number of microseconds since boot to an absolute_time_t
```
static absolute_time_t get_absolute_time (void)
    Return a representation of the current time.

static uint32_t to_ms_since_boot (absolute_time_t t)
    Convert a timestamp into a number of milliseconds since boot.

static absolute_time_t delayed_by_us (const absolute_time_t t, uint64_t us)
    Return a timestamp value obtained by adding a number of microseconds to another timestamp.

static absolute_time_t delayed_by_ms (const absolute_time_t t, uint32_t ms)
    Return a timestamp value obtained by adding a number of milliseconds to another timestamp.

static absolute_time_t make_timeout_time_us (uint64_t us)
    Convenience method to get the timestamp a number of microseconds from the current time.

static absolute_time_t make_timeout_time_ms (uint32_t ms)
    Convenience method to get the timestamp a number of milliseconds from the current time.

static int64_t absolute_time_diff_us (absolute_time_t from, absolute_time_t to) [inline], [static]
    Return the difference in microseconds between two timestamps.

static absolute_time_t absolute_time_min (absolute_time_t a, absolute_time_t b)
    Return the earlier of two timestamps.

static bool is_at_the_end_of_time (absolute_time_t t)
    Determine if the given timestamp is "at_the_end_of_time".

static bool is_nil_time (absolute_time_t t)
    Determine if the given timestamp is nil.

4.2.8.3.3. Variables

const absolute_time_t at_the_end_of_time
    The timestamp representing the end of time; this is actually not the maximum possible timestamp, but is set to 0x7fffffff_ffffffff microseconds to avoid sign overflows with time arithmetic. This is almost 300,000 years, so should be sufficient.

const absolute_time_t nil_time
    The timestamp representing a null timestamp.

4.2.8.3.4. Function Documentation

absolute_time_diff_us

static int64_t absolute_time_diff_us (absolute_time_t from, absolute_time_t to) [inline], [static]
    Return the difference in microseconds between two timestamps.

NOTE
    be careful when diffing against large timestamps (e.g. at_the_end_of_time) as the signed integer may overflow.

Parameters

from    the first timestamp

to      the second timestamp

Returns
the number of microseconds between the two timestamps (positive if \( \text{to} \) is after \( \text{from} \) except in case of overflow)

**absolute_time_min**

```c
static absolute_time_t absolute_time_min (absolute_time_t a, absolute_time_t b) [inline], [static]
```

Return the earlier of two timestamps.

**Parameters**

- `a` the first timestamp
- `b` the second timestamp

**Returns**

the earlier of the two timestamps

**delayed_by_ms**

```c
static absolute_time_t delayed_by_ms (const absolute_time_t t, uint32_t ms) [inline], [static]
```

Return a timestamp value obtained by adding a number of milliseconds to another timestamp.

**Parameters**

- `t` the base timestamp
- `ms` the number of milliseconds to add

**Returns**

the timestamp representing the resulting time

**delayed_by_us**

```c
static absolute_time_t delayed_by_us (const absolute_time_t t, uint64_t us) [inline], [static]
```

Return a timestamp value obtained by adding a number of microseconds to another timestamp.

**Parameters**

- `t` the base timestamp
- `us` the number of microseconds to add

**Returns**

the timestamp representing the resulting time

**from_us_since_boot**

```c
static absolute_time_t from_us_since_boot (uint64_t us_since_boot) [inline], [static]
```

convert a number of microseconds since boot to an `absolute_time_t`

**fn from_us_since_boot**

**get_absolute_time**

```c
static absolute_time_t get_absolute_time (void) [inline], [static]
```

Return a representation of the current time.

Returns an opaque high fidelity representation of the current time sampled during the call.

**Returns**
the absolute time (now) of the hardware timer

See also

absolute_time_t
sleep_until()
time_us_64()

is_at_the_end_of_time

static bool is_at_the_end_of_time (absolute_time_t t) [inline], [static]

Determine if the given timestamp is "at_the_end_of_time".

Parameters

  t    the timestamp

Returns

true if the timestamp is at_the_end_of_time

See also

at_the_end_of_time

is_nil_time

static bool is_nil_time (absolute_time_t t) [inline], [static]

Determine if the given timestamp is nil.

Parameters

  t    the timestamp

Returns

true if the timestamp is nil

See also

nil_time

make_timeout_time_ms

static absolute_time_t make_timeout_time_ms (uint32_t ms) [inline], [static]

Convenience method to get the timestamp a number of milliseconds from the current time.

Parameters

  ms    the number of milliseconds to add to the current timestamp

Returns

the future timestamp

make_timeout_time_us

static absolute_time_t make_timeout_time_us (uint64_t us) [inline], [static]

Convenience method to get the timestamp a number of microseconds from the current time.

Parameters

  us    the number of microseconds to add to the current timestamp

Returns

the future timestamp

to_ms_since_boot
static uint32_t to_ms_since_boot (absolute_time_t t) [inline], [static]

Convert a timestamp into a number of milliseconds since boot.

fn to_ms_since_boot

Parameters

  t an absolute_time_t value to convert

Returns

the number of milliseconds since boot represented by t

See also

to_us_since_boot()

to_us_since_boot

static uint64_t to_us_since_boot (absolute_time_t t) [inline], [static]

convert an absolute_time_t into a number of microseconds since boot.

fn to_us_since_boot

Parameters

  t the absolute time to convert

Returns

a number of microseconds since boot, equivalent to t

update_us_since_boot

static void update_us_since_boot (absolute_time_t * t, uint64_t us_since_boot) [inline], [static]

update an absolute_time_t value to represent a given number of microseconds since boot

fn update_us_since_boot

Parameters

  t the absolute time value to update

  us_since_boot the number of microseconds since boot to represent. Note this should be representable as a signed 64 bit integer

4.2.8.3.5. Variable Documentation

at_the_end_of_time

const absolute_time_t at_the_end_of_time

The timestamp representing the end of time; this is actually not the maximum possible timestamp, but is set to 0x7fffffff_ffffffff microseconds to avoid sign overflows with time arithmetic. This is almost 300,000 years, so should be sufficient.

nil_time

const absolute_time_t nil_time

The timestamp representing a null timestamp.

4.2.8.3.6. sleep

Sleep functions for delaying execution in a lower power state.
Detailed Description

These functions allow the calling core to sleep. This is a lower powered sleep; waking and re-checking time on every processor event (WFE)

NOTE

These functions should not be called from an IRQ handler.

Lower powered sleep requires use of the default alarm pool which may be disabled by the PICO_TIME_DEFAULT_ALARM_POOL_DISABLED #define or currently full in which case these functions become busy waits instead.

Whilst sleep_ functions are preferable to busy_wait functions from a power perspective, the busy_wait equivalent function may return slightly sooner after the target is reached.

See also

busy_wait_until()
busy_wait_us()
busy_wait_us_32()

Functions

void sleep_until (absolute_time_t target)
Wait until after the given timestamp to return.

void sleep_us (uint64_t us)
Wait for the given number of microseconds before returning.

void sleep_ms (uint32_t ms)
Wait for the given number of milliseconds before returning.

bool best_effort_wfe_or_timeout (absolute_time_t timeout_timestamp)
Helper method for blocking on a timeout.

Function Documentation

best_effort_wfe_or_timeout

bool best_effort_wfe_or_timeout (absolute_time_t timeout_timestamp)
Helper method for blocking on a timeout.

This method will return in response to an event (as per __wfe) or when the target time is reached, or at any point before.

This method can be used to implement a lower power polling loop waiting on some condition signalled by an event (__sev()).

This is called best_effort because under certain circumstances (notably the default timer pool being disabled or full) the best effort is simply to return immediately without a __wfe, thus turning the calling code into a busy wait.

Example usage:

```c
bool my_function_with_timeout_us(uint64_t timeout_us) {
    absolute_time_t timeout_time = make_timeout_time_us(timeout_us);
    do {
        // each time round the loop, we check to see if the condition
        // we are waiting on has happened
        if (my_check_done()) {
            // do something
            return true;
        }
    }
    return false;
}
```
10    /* will try to sleep until timeout or the next processor event */
11    while (!best_effort_wfe_or_timeout(timeout_time));
12    return false; // timed out
13 }

Parameters

  timeout_timestamp  the timeout time

Returns

true if the target time is reached, false otherwise

sleep_ms

void sleep_ms (uint32_t ms)

Wait for the given number of milliseconds before returning.

NOTE

This method attempts to perform a lower power sleep (using WFE) as much as possible.

Parameters

  ms  the number of milliseconds to sleep

sleep_until

void sleep_until (absolute_time_t target)

Wait until after the given timestamp to return.

NOTE

This method attempts to perform a lower power (WFE) sleep

Parameters

  target  the time after which to return

See also

sleep_us()  
busy_wait_until()

sleep_us

void sleep_us (uint64_t us)

Wait for the given number of microseconds before returning.

NOTE

This method attempts to perform a lower power (WFE) sleep

Parameters

  us  the number of microseconds to sleep

See also

busy_wait_us()  
alarm
Alarm functions for scheduling future execution.

**Detailed Description**

Alarms are added to alarm pools, which may hold a certain fixed number of active alarms. Each alarm pool utilizes one of four underlying 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

**Macros**

- #define PICO_TIME_DEFAULT_ALARM_POOL_DISABLED 0
- #define PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM 3
- #define PICO_TIME_DEFAULT_ALARM_POOL_MAX_TIMERS 16

**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.

**Functions**

void alarm_pool_init_default (void)

Create the default alarm pool (if not already created or disabled)

alarm_pool_t * alarm_pool_get_default (void)

The default alarm pool used when alarms are added without specifying an alarm pool, and also used by the SDK to support lower power sleeps and timeouts.

alarm_pool_t * alarm_pool_create (uint hardware_alarm_num, uint max_timers)

Create an alarm pool.

alarm_pool_t * alarm_pool_create_with_unused_hardware_alarm (uint max_timers)

Create an alarm pool, claiming an used hardware alarm to back it.

uint alarm_pool_hardware_alarm_num (alarm_pool_t *pool)

Return the hardware alarm used by an alarm pool.

uint alarm_pool_core_num (alarm_pool_t *pool)

Return the core number the alarm pool was initialized on (and hence callbacks are called on)

void alarm_pool_destroy (alarm_pool_t *pool)

Destroy the alarm pool, cancelling all alarms and freeing up the underlying hardware alarm.

alarm_id_t alarm_pool_add_alarm_at (alarm_pool_t *pool, absolute_time_t time, alarm_callback_t callback, void *user_data, bool fire_if_past)

Add an alarm callback to be called at a specific time.

alarm_id_t alarm_pool_add_alarm_at_force_in_context (alarm_pool_t *pool, absolute_time_t time, alarm_callback_t callback, void *user_data)

Add an alarm callback to be called at or after a specific time.
static alarm_id_t alarm_pool_add_alarm_in_us (alarm_pool_t *pool, uint64_t us, alarm_callback_t callback, void *user_data, bool fire_if_past)

Add an alarm callback to be called after a delay specified in microseconds.

static alarm_id_t alarm_pool_add_alarm_in_ms (alarm_pool_t *pool, uint32_t ms, alarm_callback_t callback, void *user_data, bool fire_if_past)

Add an alarm callback to be called after a delay specified in milliseconds.

bool alarm_pool_cancel_alarm (alarm_pool_t *pool, alarm_id_t alarm_id)

Cancel an alarm.

static alarm_id_t add_alarm_at (absolute_time_t time, alarm_callback_t callback, void *user_data, bool fire_if_past)

Add an alarm callback to be called at a specific time.

static alarm_id_t add_alarm_in_us (uint64_t us, alarm_callback_t callback, void *user_data, bool fire_if_past)

Add an alarm callback to be called after a delay specified in microseconds.

static alarm_id_t add_alarm_in_ms (uint32_t ms, alarm_callback_t callback, void *user_data, bool fire_if_past)

Add an alarm callback to be called after a delay specified in milliseconds.

static bool cancel_alarm (alarm_id_t alarm_id)

Cancel an alarm from the default alarm pool.

Macro Definition Documentation

PICO_TIME_DEFAULT_ALARM_POOL_DISABLED

#define PICO_TIME_DEFAULT_ALARM_POOL_DISABLED 0

If 1 then the default alarm pool is disabled (so no hardware alarm is claimed for the pool)

NOTE

Setting to 1 may cause some code not to compile as default timer pool related methods are removed

When the default alarm pool is disabled, _sleep methods and timeouts are no longer lower powered (they become _busy_wait)

See also

PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM

alarm_pool_get_default()

PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM

#define PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM 3

Selects which hardware alarm is used for the default alarm pool.

See also

alarm_pool_get_default()

PICO_TIME_DEFAULT_ALARM_POOL_MAX_TIMERS

#define PICO_TIME_DEFAULT_ALARM_POOL_MAX_TIMERS 16

Selects the maximum number of concurrent timers in the default alarm pool.
NOTE
For implementation reasons this is limited to PICO_PHEAP_MAX_ENTRIES which defaults to 255

See also
PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM

alarm_pool_get_default()

TypeDef Documentation

alarm_id_t
typedef int32_t alarm_id_t
The identifier for an alarm.

NOTE
this identifier is signed because -1 is used as an error condition when creating alarms
alarm ids may be reused, however for convenience the implementation makes an attempt to defer reusing as long as possible. You should certainly expect it to be hundreds of ids before one is reused, although in most cases it is more. Nonetheless care must still be taken when cancelling alarms or other functionality based on alarms when the alarm may have expired, as eventually the alarm id may be reused for another alarm.

alarm_callback_t
typedef int64_t(* alarm_callback_t) (alarm_id_t id, void *user_data)
User alarm callback.

Parameters
id the alarm_id as returned when the alarm was added
user_data the user data passed when the alarm was added

Returns
<0 to reschedule the same alarm this many us from the time the alarm was previously scheduled to fire
Return
>0 to reschedule the same alarm this many us from the time this method returns
Return
0 to not reschedule the alarm

Function Documentation

add_alarm_at
static alarm_id_t add_alarm_at (absolute_time_t time, alarm_callback_t callback, void *user_data, bool fire_if_past) [inline], [static]
Add an alarm callback to be called at a specific time.
Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.
NOTE
It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters
- **time**: the timestamp when (after which) the callback should fire
- **callback**: the callback function
- **user_data**: user data to pass to the callback function
- **fire_if_past**: if true, and the alarm time falls before or during this call before the alarm can be set, then the callback should be called during (by) this function instead

Returns
- >0 the alarm id

0 if the alarm time passed before or during the call AND there is no active alarm to return the id of. The latter can either happen because fire_if_past was false (i.e. no timer was ever created), or if the callback was called during this method but the callback cancelled itself by returning 0

Returns
- -1 if there were no alarm slots available

add_alarm_in_ms

```c
static alarm_id_t add_alarm_in_ms (uint32_t ms, alarm_callback_t callback, void * user_data, bool fire_if_past) [inline], [static]
```

Add an alarm callback to be called after a delay specified in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

NOTE
It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters
- **ms**: the delay (from now) in milliseconds when (after which) the callback should fire
- **callback**: the callback function
- **user_data**: user data to pass to the callback function
- **fire_if_past**: if true, and the alarm time falls during this call before the alarm can be set, then the callback should be called during (by) this function instead

Returns
- >0 the alarm id

Returns
- 0 if the alarm time passed before or during the call AND there is no active alarm to return the id of. The latter can either happen because fire_if_past was false (i.e. no timer was ever created), or if the callback was called during this method but the callback cancelled itself by returning 0

Returns
-1 if there were no alarm slots available

**add_alarm_in_us**

```cpp
static alarm_id_t add_alarm_in_us (uint64_t us, alarm_callback_t callback, void * user_data, bool fire_if_past) [inline],
[static]
```

Add an alarm callback to be called after a delay specified in microseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

**NOTE**

It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

**Parameters**

- **us**: the delay (from now) in microseconds when (after which) the callback should fire
- **callback**: the callback function
- **user_data**: user data to pass to the callback function
- **fire_if_past**: if true, and the alarm time falls during this call before the alarm can be set, then the callback should be called during (by) this function instead

**Returns**

- >0 the alarm id

**alarm_pool_add_alarm_at**

```cpp
alarm_id_t alarm_pool_add_alarm_at (alarm_pool_t * pool, absolute_time_t time, alarm_callback_t callback, void * user_data, bool fire_if_past)
```

Add an alarm callback to be called at a specific time.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

**NOTE**

It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

**Parameters**

- **pool**: the alarm pool to use for scheduling the callback (this determines which hardware alarm is used, and which core calls the callback)
- **time**: the timestamp when (after which) the callback should fire
- **callback**: the callback function
alarm_id_t alarm_pool_add_alarm_at_force_in_context (alarm_pool_t * pool, absolute_time_t time, alarm_callback_t callback, void * user_data)

Add an alarm callback to be called at or after a specific time.

The callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. Unlike alarm_pool_add_alarm_at, this method guarantees to call the callback from that core even if the time is during this method call or in the past.

**NOTE**

It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

**Parameters**

- **pool**: the alarm pool to use for scheduling the callback (this determines which 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

**Returns**

- >0 the alarm id for an active (at the time of return) alarm
- Returns -1 if there were no alarm slots available

alarm_id_t alarm_pool_add_alarm_in_ms (alarm_pool_t * pool, uint32_t ms, alarm_callback_t callback, void * user_data, bool fire_if_past) [inline], [static]

Add an alarm callback to be called after a delay specified in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.
It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

- pool: the alarm pool to use for scheduling the callback (this determines which 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, and the alarm time falls before or during this call before the alarm can be set, then the callback should be called during (by) this function instead

Returns

>0 the alarm id

0 if the alarm time passed before or during the call AND there is no active alarm to return the id of. The latter can either happen because fire_if_past was false (i.e. no timer was ever created), or if the callback was called during this method but the callback cancelled itself by returning 0

Returns

-1 if there were no alarm slots available

alarm_pool_add_alarm_in_us

static alarm_id_t alarm_pool_add_alarm_in_us (alarm_pool_t * pool, uint64_t us, alarm_callback_t callback, void * user_data, bool fire_if_past) [inline], [static]

Add an alarm callback to be called after a delay specified in microseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

- pool: the alarm pool to use for scheduling the callback (this determines which 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, and the alarm time falls during this call before the alarm can be set, then the callback should be called during (by) this function instead

Returns

>0 the alarm id

0 if the alarm time passed before or during the call AND there is no active alarm to return the id of. The latter can either
happen because fire_if_past was false (i.e. no timer was ever created), or if the callback was called during this method but the callback cancelled itself by returning 0

Returns
-1 if there were no alarm slots available

alarm_pool_cancel_alarm

bool alarm_pool_cancel_alarm (alarm_pool_t * pool, alarm_id_t alarm_id)

Cancel an alarm.

Parameters
  pool the alarm_pool containing the alarm
  alarm_id the alarm

Returns
true if the alarm was cancelled, false if it didn't exist

See also
alarm_id_t for a note on reuse of IDs

alarm_pool_core_num

uint alarm_pool_core_num (alarm_pool_t * pool)

Return the core number the alarm pool was initialized on (and hence callbacks are called on)

Parameters
  pool the pool

Returns
the core used by the pool

alarm_pool_create

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)

**NOTE**

This method will hard assert if the hardware alarm is already claimed.

Parameters

  hardware_alarm_num the hardware alarm to use to back this pool
  max_timers the maximum number of timers
NOTE

For implementation reasons this is limited to PICO_PHEAP_MAX_ENTRIES which defaults to 255

See also

alarm_pool_get_default()
hardware_claiming

alarm_pool_create_with_unused_hardware_alarm

alarm_pool_t * alarm_pool_create_with_unused_hardware_alarm (uint max_timers)
Create an alarm pool, claiming an unused hardware alarm to back it.

The alarm pool will call callbacks from an alarm IRQ Handler on the core of this function is called from.

In many situations there is never any need for anything other than the default alarm pool, however you might want to
create another if you want alarm callbacks on core 1 or require alarm pools of different priority (IRQ priority based
preemption of callbacks)

NOTE

This method will hard assert if there is no free hardware to claim.

Parameters

max_timers  the maximum number of timers

NOTE

For implementation reasons this is limited to PICO_PHEAP_MAX_ENTRIES which defaults to 255

See also

alarm_pool_get_default()
hardware_claiming

alarm_pool_destroy

void alarm_pool_destroy (alarm_pool_t * pool)
Destroy the alarm pool, cancelling all alarms and freeing up the underlying hardware alarm.

Parameters

pool  the pool

alarm_pool_get_default

alarm_pool_t * alarm_pool_get_default (void)
The default alarm pool used when alarms are added without specifying an alarm pool, and also used by the SDK to
support lower power sleeps and timeouts.

See also

PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM

alarm_pool_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

alarm_pool_init_default

void alarm_pool_init_default (void)
Create the default alarm pool (if not already created or disabled)

cancel_alarm

static bool cancel_alarm (alarm_id_t alarm_id) [inline], [static]
Cancel an alarm from the default alarm pool.

Parameters
alarm_id    the alarm

Returns
true if the alarm was cancelled, false if it didn’t exist

See also
alarm_id_t for a note on reuse of IDs

repeating_timer

Repeating Timer functions for simple scheduling of repeated execution.

Detailed Description

NOTE

The regular alarm_ functionality can be used to make repeating alarms (by return non zero from the callback), however these methods abstract that further (at the cost of a user structure to store the repeat delay in (which the alarm framework does not have space for)).

Typedefs

typedef bool(* repeating_timer_callback_t)(repeating_timer_t *rt)
    Callback for a repeating timer.

Functions

bool alarm_pool_add_repeating_timer_us (alarm_pool_t *pool, int64_t delay_us, repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)
    Add a repeating timer that is called repeatedly at the specified interval in microseconds.

static bool alarm_pool_add_repeating_timer_ms (alarm_pool_t *pool, int32_t delay_ms, repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)
    Add a repeating timer that is called repeatedly at the specified interval in milliseconds.

static bool add_repeating_timer_us (int64_t delay_us, repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)
    Add a repeating timer that is called repeatedly at the specified interval in microseconds.

static bool add_repeating_timer_ms (int32_t delay_ms, repeating_timer_callback_t callback, void *user_data, repeating_timer_t *out)
    Add a repeating timer that is called repeatedly at the specified interval in milliseconds.
bool cancel_repeating_timer (repeating_timer_t *timer)
  
  Cancel a repeating timer.

Typedef Documentation

repeating_timer_callback_t
typedef bool(* repeating_timer_callback_t) (repeating_timer_t *rt)

Callback for a repeating timer.

Parameters

  rt        repeating time structure containing information about the repeating time. user_data is of primary important to the user

Returns

true to continue repeating, false to stop.

Function Documentation

add_repeating_timer_ms

static bool add_repeating_timer_ms (int32_t delay_ms, repeating_timer_callback_t callback, void * user_data, repeating_timer_t * out) [inline], [static]

Add a repeating timer that is called repeatedly at the specified interval in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

NOTE

It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

  delay_ms  the repeat delay in milliseconds; if >0 then this is the delay between one callback ending and the next starting; if <0 then this is the negative of the time between the starts of the callbacks. The value of 0 is treated as 1 microsecond
  callback  the repeating timer callback function
  user_data user data to pass to store in the repeating_timer structure for use by the callback.
  out       the pointer to the user owned structure to store the repeating timer info in. BEWARE this storage location must outlive the repeating timer, so be careful of using stack space

Returns

false if there were no alarm slots available to create the timer, true otherwise.

add_repeating_timer_us

static bool add_repeating_timer_us (int64_t delay_us, repeating_timer_callback_t callback, void * user_data, repeating_timer_t * out) [inline], [static]

Add a repeating timer that is called repeatedly at the specified interval in microseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core of the default alarm pool (generally core 0). If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.
NOTE

It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

- **delay_us**: the repeat delay in microseconds; if >0 then this is the delay between one callback ending and the next starting; if <0 then this is the negative of the time between the starts of the callbacks. The value of 0 is treated as 1 microsecond.
- **callback**: the repeating timer callback function.
- **user_data**: user data to pass to store in the `repeating_timer` structure for use by the callback.
- **out**: the pointer to the user owned structure to store the repeating timer info in. BEWARE this storage location must outlive the repeating timer, so be careful of using stack space.

Returns

false if there were no alarm slots available to create the timer, true otherwise.

### `alarm_pool_add_repeating_timer_ms`

```c
static bool alarm_pool_add_repeating_timer_ms (alarm_pool_t * pool, int32_t delay_ms, repeating_timer_callback_t callback, void * user_data, repeating_timer_t * out) [inline], [static]
```

Add a repeating timer that is called repeatedly at the specified interval in milliseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.

NOTE

It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

- **pool**: the alarm pool to use for scheduling the repeating timer (this determines which 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.

### `alarm_pool_add_repeating_timer_us`

```c
bool alarm_pool_add_repeating_timer_us (alarm_pool_t * pool, int64_t delay_us, repeating_timer_callback_t callback, void * user_data, repeating_timer_t * out)
```

Add a repeating timer that is called repeatedly at the specified interval in microseconds.

Generally the callback is called as soon as possible after the time specified from an IRQ handler on the core the alarm pool was created on. If the callback is in the past or happens before the alarm setup could be completed, then this method will optionally call the callback itself and then return a return code to indicate that the target time has passed.
NOTE

It is safe to call this method from an IRQ handler (including alarm callbacks), and from either core.

Parameters

- **pool**: the alarm pool to use for scheduling the repeating timer (this determines which 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.

**cancel_repeating_timer**

```c
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 for a note on reuse of IDs

### 4.2.9. pico_unique_id

#### 4.2.9.1. Detailed Description

Unique device ID access API

RP2040 does not have an on-board unique identifier (all instances of RP2040 silicon are identical and have no persistent state). However, RP2040 boots from serial NOR flash devices which have a 64-bit unique ID as a standard feature, and there is a 1:1 association between RP2040 and flash, so this is suitable for use as a unique identifier for an RP2040-based board.

This library injects a call to the flash_get_unique_id function from the hardware_flash library, to run before main, and stores the result in a static location which can safely be accessed at any time via pico_get_unique_id().

This avoids some pitfalls of the hardware_flash API, which requires any flash-resident interrupt routines to be disabled when called into.
4.2.9.2. Functions

void pico_get_unique_board_id (pico_unique_board_id_t *id_out)
   Get unique ID.

void pico_get_unique_board_id_string (char *id_out, uint len)
   Get unique ID in string format.

4.2.9.3. Function Documentation

4.2.9.3.1. pico_get_unique_board_id

void pico_get_unique_board_id (pico_unique_board_id_t *id_out)
   Get unique ID.
   Get the unique 64-bit device identifier which was retrieved from the external NOR flash device at boot.
   On PICO_NO_FLASH builds the unique identifier is set to all 0xEE.

Parameters
   id_out  a pointer to a pico_unique_board_id_t struct, to which the identifier will be written

4.2.9.3.2. pico_get_unique_board_id_string

void pico_get_unique_board_id_string (char *id_out, uint len)
   Get unique ID in string format.
   Get the unique 64-bit device identifier which was retrieved from the external NOR flash device at boot, formatted as an ASCII hex string. Will always 0-terminate.
   On PICO_NO_FLASH builds the unique identifier is set to all 0xEE.

Parameters
   id_out  a pointer to a char buffer of size len, to which the identifier will be written
   len     the size of id_out. For full serial, len >= 2 * PICO_UNIQUE_BOARD_ID_SIZE_BYTES + 1

4.2.10. pico_util

Useful data structures and utility functions.

4.2.10.1. Modules

datetime
   Date/Time formatting.

pheap

queue
4.2.10.2. datetime

Date/Time formatting.

4.2.10.2.1. Functions

```c
void datetime_to_str (char * buf, uint buf_size, const datetime_t * t)
```
Convert a `datetime_t` structure to a string.

4.2.10.2.2. Function Documentation
datetime_to_str

```c
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.

4.2.10.2.3. pheap

Detailed Description

Pairing Heap Implementation

`pheap` defines a simple pairing heap. The implementation simply tracks array indexes, it is up to the user to provide storage for heap entries and a comparison function.

**NOTE**

This class is not safe for concurrent usage. It should be externally protected. Furthermore if used concurrently, the caller needs to protect around their use of the returned id. For example, `ph_remove_and_free_head` returns the id of an element that is no longer in the heap. The user can still use this to look at the data in their companion array, however obviously further operations on the heap may cause them to overwrite that data as the id may be reused on subsequent operations.

queue

Detailed Description

Multi-core and IRQ safe queue implementation.

Note that this queue stores values of a specified size, and pushed values are copied into the queue.

**Functions**

```c
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.

```c
static void queue_init (queue_t *q, uint element_size, uint element_count)
```
Initialise a queue, allocating a (possibly shared) spinlock.

```c
void queue_free (queue_t *q)
```
Destroy the specified queue.
static uint queue_get_level_unsafe (queue_t *q)
    Unsafe check of level of the specified queue.

static uint queue_get_level (queue_t *q)
    Check of level of the specified queue.

static bool queue_is_empty (queue_t *q)
    Check if queue is empty.

static bool queue_is_full (queue_t *q)
    Check if queue is full.

bool queue_try_add (queue_t *q, const void *data)
    Non-blocking add value queue if not full.

bool queue_try_remove (queue_t *q, void *data)
    Non-blocking removal of entry from the queue if non empty.

bool queue_try_peek (queue_t *q, void *data)
    Non-blocking peek at the next item to be removed from the queue.

void queue_add_blocking (queue_t *q, const void *data)
    Blocking add of value to queue.

void queue_remove_blocking (queue_t *q, void *data)
    Blocking remove entry from queue.

void queue_peek_blocking (queue_t *q, void *data)
    Blocking peek at next value to be removed from queue.

Function Documentation

queue_add_blocking

void queue_add_blocking (queue_t * q, const void * data)
Blocking add of value to queue.

Parameters

    q Pointer to a queue_t structure, used as a handle
    data Pointer to value to be copied into the queue

If the queue is full this function will block, until a removal happens on the queue

queue_free

void queue_free (queue_t * q)
Destroy the specified queue.

Parameters

    q Pointer to a queue_t structure, used as a handle

Does not deallocate the queue_t structure itself.

queue_get_level

static uint queue_get_level (queue_t * q) [inline], [static]
Check of level of the specified queue.

Parameters

    q Pointer to a queue_t structure, used as a handle
Returns
Number of entries in the queue

queue_get_level_unsafe
static uint queue_get_level_unsafe (queue_t * q) [inline], [static]
Unsafe check of level of the specified queue.
Parameters
q Pointer to a queue_t structure, used as a handle

Returns
Number of entries in the queue
This does not use the spinlock, so may return incorrect results if the spin lock is not externally locked

queue_init
static void queue_init (queue_t * q, uint element_size, uint element_count) [inline], [static]
Initialise a queue, allocating a (possibly shared) spinlock.
Parameters
q Pointer to a queue_t structure, used as a handle
element_size Size of each value in the queue
element_count Maximum number of entries in the queue

queue_init_with_spinlock
void queue_init_with_spinlock (queue_t * q, uint element_size, uint element_count, uint spinlock_num)
Initialise a queue with a specific spinlock for concurrency protection.
Parameters
q Pointer to a queue_t structure, used as a handle
element_size Size of each value in the queue
element_count Maximum number of entries in the queue
spinlock_num The spin ID used to protect the queue

queue_is_empty
static bool queue_is_empty (queue_t * q) [inline], [static]
Check if queue is empty.
Parameters
q Pointer to a queue_t structure, used as a handle

Returns
true if queue is empty, false otherwise
This function is interrupt and multicore safe.

queue_is_full
static bool queue_is_full (queue_t * q) [inline], [static]
Check if queue is full.
Parameters
q Pointer to a queue_t structure, used as a handle
Returns
true if queue is full, false otherwise
This function is interrupt and multicore safe.

queue_peek_blocking

```c
void queue_peek_blocking (queue_t * q, void * data)
```
Blocking peek at next value to be removed from queue.

Parameters
- `q` Pointer to a `queue_t` structure, used as a handle
- `data` Pointer to the location to receive the peeked value
If the queue is empty function will block until a value is added

queue_remove_blocking

```c
void queue_remove_blocking (queue_t * q, void * data)
```
Blocking remove entry from queue.

Parameters
- `q` Pointer to a `queue_t` structure, used as a handle
- `data` Pointer to the location to receive the removed value
If the queue is empty this function will block until a value is added.

queue_try_add

```c
bool queue_try_add (queue_t * q, const void * data)
```
Non-blocking add value queue if not full.

Parameters
- `q` Pointer to a `queue_t` structure, used as a handle
- `data` Pointer to value to be copied into the queue

Returns
true if the value was added
If the queue is full this function will return immediately with false, otherwise the data is copied into a new value added to the queue, and this function will return true.

queue_try_peek

```c
bool queue_try_peek (queue_t * q, void * data)
```
Non-blocking peek at the next item to be removed from the queue.

Parameters
- `q` Pointer to a `queue_t` structure, used as a handle
- `data` Pointer to the location to receive the peeked value

Returns
true if there was a value to peek
If the queue is not empty this function will return immediately with true with the peeked entry copied into the location specified by the data parameter, otherwise the function will return false.

queue_try_remove
```c
bool queue_try_remove (queue_t * q, void * data)
```

Non-blocking removal of entry from the queue if non empty.

**Parameters**
- `q`: Pointer to a `queue_t` structure, used as a handle
- `data`: Pointer to the location to receive the removed value

**Returns**
- `true` if a value was removed

If the queue is not empty function will copy the removed value into the location provided and return immediately with `true`, otherwise the function will return immediately with `false`.

### 4.3. Third-party Libraries

Third party libraries for implementing high level functionality.

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tinyusb_device</td>
<td>TinyUSB Device-mode support for the RP2040. The TinyUSB documentation site can be found <a href="#">here</a>.</td>
</tr>
<tr>
<td>tinyusb_host</td>
<td>TinyUSB Host-mode support for the RP2040.</td>
</tr>
</tbody>
</table>

#### 4.3.1. tinyusb_device

**4.3.1.1. Detailed Description**

TinyUSB Device-mode support for the RP2040. The TinyUSB documentation site can be found [here](#).

#### 4.3.2. tinyusb_host

**4.3.2.1. Detailed Description**

TinyUSB Host-mode support for the RP2040.

### 4.4. Networking Libraries

Functions for implementing networking.

<table>
<thead>
<tr>
<th>Library</th>
<th>Integration/wrapper libraries for BTstack the documentation for which is <a href="#">here</a>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pico_btstack</td>
<td>Integration/wrapper libraries for lwIP the documentation for which is <a href="#">here</a>.</td>
</tr>
<tr>
<td>pico_lwip</td>
<td>lwIP compiler adapters. This is not included by default in <code>pico_lwip</code> in case you wish to implement your own.</td>
</tr>
<tr>
<td>pico_lwip_arch</td>
<td>lwIP for NO_SYS=0 mode with the SDK. Simple <code>init</code> and <code>deinit</code> are all that is required to hook up lwIP (with full blocking API support) via an <code>async_context</code> instance.</td>
</tr>
<tr>
<td>pico_lwip_freertos</td>
<td>Glue library for integration lwIP in NO_SYS=0 mode with the SDK. Simple <code>init</code> and <code>deinit</code> are all that is required to hook up lwIP via an <code>async_context</code> instance.</td>
</tr>
<tr>
<td>pico_lwip_nosys</td>
<td>Glue library for integration lwIP in NO_SYS=1 mode with the SDK. Simple <code>init</code> and <code>deinit</code> are all that is required to hook up lwIP via an <code>async_context</code> instance.</td>
</tr>
</tbody>
</table>
4.4.1. pico_btstack

Integration/wrapper libraries for BTstack the documentation for which is here.

4.4.1.1. Detailed Description

A supplemental license for BTstack (in addition to the stock BTstack licensing terms) is provided here.

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

To use BTstack you need to provide a btstack_config.h file in your source tree and add its location to your include path. The BTstack configuration macros ENABLE_CLASSIC and ENABLE_BLE are defined for you when you link the pico_btstack_classic and pico_btstack_ble libraries respectively, so you should not define them yourself.

For more details, see How to configure BTstack and the relevant pico-examples.

The follow libraries are provided for you to link.

- pico_btstack_ble - Adds Bluetooth Low Energy (LE) support.
- pico_btstack_classic - Adds Bluetooth Classic support.
- pico_btstack_sbc_encoder - Adds Bluetooth Sub Band Coding (SBC) encoder support.
- pico_btstack_sbc_decoder - Adds Bluetooth Sub Band Coding (SBC) decoder support.
- pico_btstack_bnep_lwip - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP.
- pico_btstack_bnep_lwip_sys_freertos - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP with FreeRTOS for NO_SYS=0.

NOTE

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

See also

pico_btstack_cyw43 in pico_cyw43_driver, which adds the cyw43 driver support needed for BTstack including BTstack run loop support.

4.4.1.2. Functions

const hal_flash_bank_t * pico_flash_bank_instance (void)

Return the singleton BTstack HAL flash instance, used for non-volatile storage.

const btstack_run_loop_t * btstack_run_loop_async_context_get_instance (async_context_t *context)

Initialize and return the singleton BTstack run loop instance that integrates with the async_context API.
const btstack_chipset_t * btstack_chipset_cyw43_instance (void)
Return the singleton BTstack chipset CY43 API instance.

4.4.1.3. Function Documentation

4.4.1.3.1. btstack_chipset_cyw43_instance
const btstack_chipset_t * btstack_chipset_cyw43_instance (void)
Return the singleton BTstack chipset CY43 API instance.

4.4.1.3.2. btstack_run_loop_async_context_get_instance
const btstack_run_loop_t * btstack_run_loop_async_context_get_instance (async_context_t * context)
Initialize and return the singleton BTstack run loop instance that integrates with the async_context API.
Parameters
  context the async_context instance that provides the abstraction for handling asynchronous work.
Returns
the BTstack run loop instance

4.4.1.3.3. pico_flash_bank_instance
const hal_flash_bank_t * pico_flash_bank_instance (void)
Return the singleton BTstack HAL flash instance, used for non-volatile storage.

NOTE
By default two sectors at the end of flash are used (see PICO_FLASH_BANK_STORAGE_OFFSET and PICO_FLASH_BANK_TOTAL_SIZE)

4.4.2. pico_lwip
Integration/wrapper libraries for lwIP the documentation for which is here.

4.4.2.1. Detailed Description
The main pico_lwip library itself aggregates the lwIP RAW API: pico_lwip_core, pico_lwip_core4, pico_lwip_core6, pico_lwip_api, pico_lwip_netif, pico_lwip_sixlowpan and pico_lwip_ppp.
If you wish to run in NO_SYS=1 mode, then you can link pico_lwip along with pico_lwip_nosys.
If you wish to run in NO_SYS=0 mode, then you can link pico_lwip with (for instance) pico_lwip_freertos, and also link in pico_lwip_api for the additional blocking/thread-safe APIs.
Additionally you must link in pico_lwip_arch unless you provide your own compiler bindings for lwIP.
Additional individual pieces of lwIP functionality are available à la cart, by linking any of the libraries below.
The following libraries are provided that contain exactly the equivalent lwIP functionality groups:
  * pico_lwip_core -
The following libraries are provided that contain exactly the equivalent lwIP application support:

- `pico_lwip_snmp`
- `pico_lwip_http`
- `pico_lwip_makefsdata`
- `pico_lwip_iperf`
- `pico_lwip_smtp`
- `pico_lwip_sntp`
- `pico_lwip_mdns`
- `pico_lwip_netbios`
- `pico_lwip_tftp`
- `pico_lwip_mbedtls`
- `pico_lwip_mqtt`

### 4.4.2.2. Modules

**pico_lwip_arch**

lwIP compiler adapters. This is not included by default in `pico_lwip` in case you wish to implement your own.

**pico_lwip_freertos**

Glue library for integration lwIP in NO_SYS=0 mode with the SDK. Simple `init` and `deinit` are all that is required to hook up lwIP (with full blocking API support) via an `async_context` instance.

**pico_lwip_nosys**

Glue library for integration lwIP in NO_SYS=1 mode with the SDK. Simple `init` and `deinit` are all that is required to hook up lwIP via an `async_context` instance.

### 4.4.2.3. pico_lwip_arch

lwIP compiler adapters. This is not included by default in `pico_lwip` in case you wish to implement your own.

#### 4.4.2.3.1. pico_lwip_freertos

Glue library for integration lwIP in NO_SYS=0 mode with the SDK. Simple `init` and `deinit` are all that is required to hook up lwIP (with full blocking API support) via an `async_context` instance.

**Functions**

```c
bool lwip_freertos_init (async_context_t *context)
```

Initializes lwIP (NO_SYS=0 mode) support support for FreeRTOS using the provided `async_context`.

```c
void lwip_freertos_deinit (async_context_t *context)
```

De-initialize lwIP (NO_SYS=0 mode) support for FreeRTOS.
Function Documentation

lwip_freertos_deinit

```c
void lwip_freertos_deinit (async_context_t * context)
```

De-initialize lwIP (NO_SYS=0 mode) support for FreeRTOS.

Note that since lwIP may only be initialized once, and doesn't itself provide a shutdown mechanism, lwIP itself may still consume resources.

It is however safe to call `lwip_freertos_init` again later.

**Parameters**

- `context` the async_context the lwip_freertos support was added to via `lwip_freertos_init`

lwip_freertos_init

```c
bool lwip_freertos_init (async_context_t * context)
```

Initializes lwIP (NO_SYS=0 mode) support support for FreeRTOS using the provided async_context.

If the initialization succeeds, `lwip_freertos_deinit()` can be called to shutdown lwIP support.

**Parameters**

- `context` the async_context instance that provides the abstraction for handling asynchronous work. Note in general this would be an async_context_freertos instance, though it doesn't have to be.

**Returns**

true if the initialization succeeded

pico_lwip_nosys

Glue library for integration lwIP in NO_SYS=1 mode with the SDK. Simple init and deinit are all that is required to hook up lwIP via an async_context instance.

**Functions**

```c
bool lwip_nosys_init (async_context_t * context)
```

Initializes lwIP (NO_SYS=1 mode) support using the provided async_context.

```c
void lwip_nosys_deinit (async_context_t * context)
```

De-initialize lwIP (NO_SYS=1 mode) support.

Function Documentation

lwip_nosys_deinit

```c
void lwip_nosys_deinit (async_context_t * context)
```

De-initialize lwIP (NO_SYS=1 mode) support.

Note that since lwIP may only be initialized once, and doesn't itself provide a shutdown mechanism, lwIP itself may still consume resources.

It is however safe to call `lwip_nosys_init` again later.

**Parameters**

- `context` the async_context the lwip_nosys support was added to via `lwip_nosys_init`

lwip_nosys_init

```c
bool lwip_nosys_init (async_context_t * context)
```

Initializes lwIP (NO_SYS=1 mode) support using the provided async_context.

If the initialization succeeds, `lwip_nosys_deinit()` can be called to shutdown lwIP support.
Parameters

- context: the async_context instance that provides the abstraction for handling asynchronous work.

Returns

ture if the initialization succeeded

### 4.4.3. pico_cyw43_driver

#### 4.4.3.1. Detailed Description

A wrapper around the lower level cyw43_driver, that integrates it with pico_async_context for handling background work.

#### 4.4.3.2. Modules

- **pico_btstack_cyw43**
  
  Low-level Bluetooth HCI support.

#### 4.4.3.3. Functions

- **const hci_transport_t**
  
  Get the Bluetooth HCI transport instance for cyw43.

- **bool cyw43_driver_init (async_context_t *context)**
  
  Initializes the lower level cyw43_driver and integrates it with the provided async_context.

- **void cyw43_driver_deinit (async_context_t *context)**
  
  De-initialize the lower level cyw43_driver and unhooks it from the async_context.

#### 4.4.3.4. Function Documentation

##### 4.4.3.4.1. cyw43_driver_deinit

```c
void cyw43_driver_deinit (async_context_t *context)
```

De-initialize the lower level cyw43_driver and unhooks it from the async_context.

Parameters

- context: the async_context the cyw43_driver support was added to via cyw43_driver_init

##### 4.4.3.4.2. cyw43_driver_init

```c
bool cyw43_driver_init (async_context_t *context)
```

Initializes the lower level cyw43_driver and integrates it with the provided async_context.

If the initialization succeeds, lwip_nosys_deinit() can be called to shutdown lwIP support.

Parameters

- context: the async_context instance that provides the abstraction for handling asynchronous work.
4.4.3.4.3. hci_transport_cyw43_instance

const hci_transport_t * hci_transport_cyw43_instance (void)

Get the Bluetooth HCI transport instance for cyw43.

Returns
An instantiation of the hci_transport_t interface for the cyw43 chipset

4.4.3.5. pico_btstack_cyw43

Low-level Bluetooth HCI support.

4.4.3.5.1. Detailed Description

This library provides utility functions to initialise and de-initialise BTstack for CYW43.

4.4.4. pico_cyw43_arch

4.4.4.1. Detailed Description

Architecture for integrating the CYW43 driver (for the wireless on Pico W) and lwIP (for TCP/IP stack) into the SDK. It is also necessary for accessing the on-board LED on Pico W.

Both the low level cyw43_driver and the lwIP stack require periodic servicing, and have limitations on whether they can be called from multiple cores/threads.

pico_cyw43_arch attempts to abstract these complications into several behavioral groups:

- ‘poll’ - This not multi-core/IRQ safe, and requires the user to call cyw43_arch_poll periodically from their main loop
- ‘thread_safe_background’ - This is multi-core/thread/task safe, and maintenance of the driver and TCP/IP stack is handled automatically in the background
- ‘freertos’ - This is multi-core/thread/task safe, and uses a separate FreeRTOS task to handle lwIP and and driver work.

As of right now, lwIP is the only supported TCP/IP stack, however the use of pico_cyw43_arch is intended to be independent of the particular TCP/IP stack used (and possibly Bluetooth stack used) in the future. For this reason, the integration of lwIP is handled in the base (pico_cyw43_arch) library based on the #define CYW43_LWIP used by the cyw43_driver.
As of version 1.5.0 of the Raspberry Pi SDK, the `pico_cyw43_arch` library no longer directly implements the distinct behavioral abstractions. This is now handled by the more general `pico_async_context` library. The user facing behavior of `pico_cyw43_arch` has not changed as a result of this implementation detail, however `pico_cyw43_arch` is now just a thin wrapper which creates an appropriate `async_context` and makes a simple call to add lwIP or cyw43_driver support as appropriate. You are free to perform this context creation and adding of lwIP, cyw43_driver or indeed any other additional future protocol/driver support to your `async_context`, however for now `pico_cyw43_arch` still provides a few cyw43_specific (i.e. Pico W) APIs for connection management, locking and GPIO interaction.

The connection management APIs at least may be moved to a more generic library in a future release. The locking methods are now backed by their `async_context` equivalents, and those methods may be used interchangeably (see `cyw43_arch_lwip_begin`, `cyw43_arch_lwip_end` and `cyw43_arch_lwip_check` for more details).

For examples of creating of your own `async_context` and addition of cyw43_driver and lwIP support, please refer to the specific source files `cyw43_arch_poll.c`, `cyw43_arch_threadsafe_background.c` and `cyw43_arch_freertos.c`.

Whilst you can use the `pico_cyw43_arch` library directly and specify CYW43_LWIP (and other defines) yourself, several other libraries are made available to the build which aggregate the defines and other dependencies for you:

- `pico_cyw43_arch_lwip_poll` - For using the RAW lwIP API (in NO_SYS=1 mode) without any background processing or multi-core/thread safety.
  
  The user must call `cyw43_arch_poll` periodically from their main loop.

  This wrapper library:
  
  - Sets CYW43_LWIP=1 to enable lwIP support in `pico_cyw43_arch` and `cyw43_driver`.
  - Sets PICO_CYW43_ARCH_POLL=1 to select the polling behavior.
  - Adds the `pico_lwip` as a dependency to pull in lwIP.

- `pico_cyw43_arch_lwip_threadsafe_background` - For using the RAW lwIP API (in NO_SYS=1 mode) with multi-core/thread safety, and automatic servicing of the `cyw43_driver` and lwIP in background.

  Calls into the `cyw43_driver` high level API (`cyw43.h`) may be made from either core or from lwIP callbacks, however calls into lwIP (which is not thread-safe) other than those made from lwIP callbacks, must be bracketed with `cyw43_arch_lwip_begin` and `cyw43_arch_lwip_end`. It is fine to bracket calls made from within lwIP callbacks too; you just don't have to.

  lwIP callbacks happen in a (low priority) IRQ context (similar to an alarm callback), so care should be taken when interacting with other code.

  This wrapper library:
  
  - Sets CYW43_LWIP=1 to enable lwIP support in `pico_cyw43_arch` and `cyw43_driver`.
  - Sets PICO_CYW43_ARCH_THREADSAFE_BACKGROUND=1 to select the thread-safe/non-polling behavior.
  - Adds the `pico_lwip` as a dependency to pull in lwIP.

  This library can also be used under the RP2040 port of FreeRTOS with lwIP in NO_SYS=1 mode (allowing you to call `cyw43_driver` APIs from any task, and to call lwIP from lwIP callbacks, or from any task if you bracket the calls with `cyw43_arch_lwip_begin` and `cyw43_arch_lwip_end`). Again, you should be careful about what you do in lwIP callbacks, as you cannot call most FreeRTOS APIs from within an IRQ context. Unless you have good reason, you should probably use the full FreeRTOS integration (with NO_SYS=0) provided by `pico_cyw43_arch_lwip_sys_freertos`.

- `pico_cyw43_arch_lwip_sys_freertos` - For using the full lwIP API including blocking sockets in OS (NO_SYS=0) mode, along with with multi-core/task/thread safety, and automatic servicing of the `cyw43_driver` and the lwIP stack.

  This wrapper library:
- Sets CYW43_LWIP=1 to enable lwIP support in pico_cyw43_arch and cyw43_driver.
- Sets PICO_CYW43_ARCH_FREERTOS=1 to select the NO_SYS=0 lwip/FreeRTOS integration.
- Sets LWIP_PROVIDE_ERRNO=1 to provide error numbers needed for compilation without an OS.
- Adds the pico_lwip as a dependency to pull in lwIP.
- Adds the lwIP/FreeRTOS code from lwip-contrib (in the contrib directory of lwIP).

Calls into the cyw43_driver high level API (cyw43.h) may be made from any task or from lwIP callbacks, but not from IRQs. Calls into the lwIP RAW API (which is not thread safe) must be bracketed with cyw43_arch_lwip_begin and cyw43_arch_lwip_end. It is fine to bracket calls made from within lwIP callbacks too; you just don't have to.

**NOTE**

this wrapper library requires you to link FreeRTOS functionality with your application yourself.

- **pico_cyw43_arch_none** - If you do not need the TCP/IP stack but wish to use the on-board LED.

  This wrapper library:
  - Sets CYW43_LWIP=0 to disable lwIP support in pico_cyw43_arch and cyw43_driver.

### 4.4.4.2. Modules

**cyw43_driver**

Driver used for Pico W wireless.

### 4.4.4.3. Functions

```c
int cyw43_arch_init (void)
```

Initialize the CYW43 architecture.

```c
int cyw43_arch_init_with_country (uint32_t country)
```

Initialize the CYW43 architecture for use in a specific country.

```c
void cyw43_arch_deinit (void)
```

De-initialize the CYW43 architecture.

```c
async_context_t * cyw43_arch_async_context (void)
```

Return the current async_context currently in use by the cyw43_arch code.

```c
void cyw43_arch_set_async_context (async_context_t *context)
```

Set the async_context to be used by the cyw43_arch_init.

```c
async_context_t * cyw43_arch_init_default_async_context (void)
```

Initialize the default async_context for the current cyw43_arch type.

```c
void cyw43_arch_poll (void)
```

Perform any processing required by the cyw43_driver or the TCP/IP stack.

```c
void cyw43_arch_wait_for_work_until (absolute_time_t until)
```

Sleep until there is cyw43_driver work to be done.

```c
uint32_t cyw43_arch_get_country_code (void)
```

Return the country code used to initialize cyw43_arch.

```c
void cyw43_arch_enable_sta_mode (void)
```

Enables Wi-Fi STA (Station) mode.
void cyw43_arch_disable_standalone_mode(void)
    Enables Wi-Fi STA (Station) mode.

void cyw43_arch_enable_ap_mode(const char *ssid, const char *password, uint32_t auth)
    Enables Wi-Fi AP (Access point) mode.

void cyw43_arch_disable_ap_mode(void)
    Enables Wi-Fi AP (Access point) mode.

int cyw43_arch_wifi_connect_blocking(const char *ssid, const char *pw, uint32_t auth)
    Attempt to connect to a wireless access point, blocking until the network is joined or a failure is detected.

int cyw43_arch_wifi_connect_bssid_blocking(const char *ssid, const uint8_t *bssid, const char *pw, uint32_t auth)
    Attempt to connect to a wireless access point specified by SSID and BSSID, blocking until the network is joined or a failure is detected.

int cyw43_arch_wifi_connect_timeout_ms(const char *ssid, const char *pw, uint32_t auth, uint32_t timeout)
    Attempt to connect to a wireless access point, blocking until the network is joined, a failure is detected or a timeout occurs.

int cyw43_arch_wifi_connect_bssid_timeout_ms(const char *ssid, const uint8_t *bssid, const char *pw, uint32_t auth, uint32_t timeout)
    Attempt to connect to a wireless access point specified by SSID and BSSID, blocking until the network is joined, a failure is detected or a timeout occurs.

int cyw43_arch_wifi_connect_async(const char *ssid, const char *pw, uint32_t auth)
    Start attempting to connect to a wireless access point.

int cyw43_arch_wifi_connect_bssid_async(const char *ssid, const uint8_t *bssid, const char *pw, uint32_t auth)
    Start attempting to connect to a wireless access point specified by SSID and BSSID.

void cyw43_arch_gpio_put(uint wl_gpio, bool value)
    Set a GPIO pin on the wireless chip to a given value.

bool cyw43_arch_gpio_get(uint wl_gpio)
    Read the value of a GPIO pin on the wireless chip.

### 4.4.4.4. Function Documentation

#### 4.4.4.4.1. cyw43_arch_async_context

async_context_t * cyw43_arch_async_context(void)

Return the current async_context currently in use by the cyw43_arch code.

Returns
   the async_context.

#### 4.4.4.4.2. cyw43_arch_deinit

void cyw43_arch_deinit(void)

De-initialize the CYW43 architecture.

This method de-initializes the cyw43_driver code and de-initializes the lwIP stack (if it was enabled at build time). Note this method should always be called from the same core (or RTOS task, depending on the environment) as cyw43_arch_init.
Additionally if the cyw43_arch is using its own `async_context` instance, then that instance is de-initialized.

### 4.4.4.4.3. cyw43_arch_disable_ap_mode

```c
void cyw43_arch_disable_ap_mode (void)
```

Disables Wi-Fi AP (Access point) mode.

This Disables the Wi-Fi in Access Point mode.

### 4.4.4.4.4. cyw43_arch_disable_sta_mode

```c
void cyw43_arch_disable_sta_mode (void)
```

Disables Wi-Fi STA (Station) mode.

This disables the Wi-Fi in Station mode, disconnecting any active connection. You should subsequently check the status by calling `cyw43_wifi_link_status`.

### 4.4.4.4.5. cyw43_arch_enable_ap_mode

```c
void cyw43_arch_enable_ap_mode (const char * ssid, const char * password, uint32_t auth)
```

Enables Wi-Fi AP (Access point) mode.

This enables the Wi-Fi in Access Point mode such that connections can be made to the device by other Wi-Fi clients.

**Parameters**

- `ssid` the name for the access point
- `password` the password to use or NULL for no password.
- `auth` the authorization type to use when the password is enabled. Values are `CYW43_AUTH_WPA_TKIP_PSK`, `CYW43_AUTH_WPA2_AES_PSK`, or `CYW43_AUTH_WPA2_MIXED_PSK` (see `CYW43_AUTH`).

### 4.4.4.4.6. cyw43_arch_enable_sta_mode

```c
void cyw43_arch_enable_sta_mode (void)
```

Enables Wi-Fi STA (Station) mode.

This enables the Wi-Fi in Station mode such that connections can be made to other Wi-Fi Access Points.

### 4.4.4.4.7. cyw43_arch_get_country_code

```c
uint32_t cyw43_arch_get_country_code (void)
```

Return the country code used to initialize cyw43_arch.

**Returns**

the country code (see `CYW43_COUNTRY_`)

### 4.4.4.4.8. cyw43_arch_gpio_get

```c
bool cyw43_arch_gpio_get (uint wl_gpio)
```

Read the value of a GPIO pin on the wireless chip.
NOTE

this method does not check for errors setting the GPIO. You can use the lower level cyw43_gpio_get instead if you wish to check for errors.

Parameters

wl_gpio    the GPIO number on the wireless chip

Returns

true if the GPIO is high, false otherwise

4.4.4.9. cyw43_arch_gpio_put

void cyw43_arch_gpio_put (uint wl_gpio, bool value)

Set a GPIO pin on the wireless chip to a given value.

NOTE

this method does not check for errors setting the GPIO. You can use the lower level cyw43_gpio_set instead if you wish to check for errors.

Parameters

wl_gpio    the GPIO number on the wireless chip

value    true to set the GPIO, false to clear it.

4.4.4.10. cyw43_arch_init

int cyw43_arch_init (void)

Initialize the CYW43 architecture.

This method initializes the cyw43_driver code and initializes the lwIP stack (if it was enabled at build time). This method must be called prior to using any other pico_cyw43_arch, cyw43_driver or lwIP functions.

NOTE

this method initializes wireless with a country code of PICO_CYW43_ARCH_DEFAULT_COUNTRY_CODE which defaults to CYW43_COUNTRY_WORLDWIDE. Worldwide settings may not give the best performance; consider setting PICO_CYW43_ARCH_DEFAULT_COUNTRY_CODE to a different value or calling cyw43_arch_init_with_country

By default this method initializes the cyw43_arch code’s own async_context by calling cyw43_arch_init_default_async_context, however the user can specify use of their own async_context by calling cyw43_arch_set_async_context() before calling this method

Returns

0 if the initialization is successful, an error code otherwise

See also

pico_error_codes
4.4.4.4.11. cyw43_arch_init_default_async_context

```c
async_context_t * cyw43_arch_init_default_async_context (void)
```

Initialize the default `async_context` for the current cyw43_arch type.

This method initializes and returns a pointer to the static `async_context` associated with cyw43_arch. This method is called by `cyw43_arch_init` automatically if a different `async_context` has not been set by `cyw43_arch_set_async_context`

**Returns**

the context or NULL if initialization failed.

4.4.4.4.12. cyw43_arch_init_with_country

```c
int cyw43_arch_init_with_country (uint32_t country)
```

Initialize the CYW43 architecture for use in a specific country.

This method initializes the `cyw43_driver` code and initializes the lwIP stack (if it was enabled at build time). This method must be called prior to using any other `pico_cyw43_arch`, `cyw43_driver` or lwIP functions.

By default this method initializes the cyw43_arch code’s own `async_context` by calling `cyw43_arch_init_default_async_context`, however the user can specify use of their own `async_context` by calling `cyw43_arch_set_async_context()` before calling this method

**Parameters**

- `country` the country code to use (see CYW43_COUNTRY_)

**Returns**

0 if the initialization is successful, an error code otherwise

**See also**

pico_error_codes

4.4.4.4.13. cyw43_arch_poll

```c
void cyw43_arch_poll (void)
```

Perform any processing required by the `cyw43_driver` or the TCP/IP stack.

This method must be called periodically from the main loop when using a polling style `pico_cyw43_arch` (e.g. `pico_cyw43_arch_lwip_poll`). It may be called in other styles, but it is unnecessary to do so.

4.4.4.4.14. cyw43_arch_set_async_context

```c
void cyw43_arch_set_async_context (async_context_t * context)
```

Set the `async_context` to be used by the cyw43_arch_init.

**NOTE**

This method must be called before calling cyw43_arch_init or cyw43_arch_init_with_country if you wish to use a custom `async_context` instance.

**Parameters**

- `context` the `async_context` to be used
4.4.4.15. cyw43_arch_wait_for_work_until

```c
void cyw43_arch_wait_for_work_until (absolute_time_t until)
```

Sleep until there is cyw43_driver work to be done. This method may be called by code that is waiting for an event to come from the cyw43_driver, and has no work to do, but would like to sleep without blocking any background work associated with the cyw43_driver.

**Parameters**

- `until` the time to wait until if there is no work to do.

4.4.4.16. cyw43_arch_wifi_connect_async

```c
int cyw43_arch_wifi_connect_async (const char * ssid, const char * pw, uint32_t auth)
```

Start attempting to connect to a wireless access point. This method tells the CYW43 driver to start connecting to an access point. You should subsequently check the status by calling `cyw43_wifi_link_status`.

**Parameters**

- `ssid` the network name to connect to
- `pw` the network password or NULL if there is no password required
- `auth` the authorization type to use when the password is enabled. Values are `CYW43_AUTH_WPA_TKIP_PSK`, `CYW43_AUTH_WPA2_AES_PSK` or `CYW43_AUTH_WPA2_MIXED_PSK` (see `CYW43_AUTH`).

**Returns**

- 0 if the scan was started successfully, an error code otherwise

**See also**

`pico_error_codes`

4.4.4.17. cyw43_arch_wifi_connect_blocking

```c
int cyw43_arch_wifi_connect_blocking (const char * ssid, const char * pw, uint32_t auth)
```

Attempt to connect to a wireless access point, blocking until the network is joined or a failure is detected.

**Parameters**

- `ssid` the network name to connect to
- `pw` the network password or NULL if there is no password required
- `auth` the authorization type to use when the password is enabled. Values are `CYW43_AUTH_WPA_TKIP_PSK`, `CYW43_AUTH_WPA2_AES_PSK`, or `CYW43_AUTH_WPA2_MIXED_PSK` (see `CYW43_AUTH`).

**Returns**

- 0 if the initialization is successful, an error code otherwise

**See also**

`pico_error_codes`

4.4.4.18. cyw43_arch_wifi_connect_bssid_async

```c
int cyw43_arch_wifi_connect_bssid_async (const char * ssid, const uint8_t * bssid, const char * pw, uint32_t auth)
```

Start attempting to connect to a wireless access point specified by SSID and BSSID.
This method tells the CYW43 driver to start connecting to an access point. You should subsequently check the status by calling `cyw43_wifi_link_status`.

**Parameters**
- **ssid**: the network name to connect to
- **bssid**: the network BSSID to connect to or NULL if ignored
- **pw**: the network password or NULL if there is no password required
- **auth**: the authorization type to use when the password is enabled. Values are `CYW43_AUTH_WPA_TKIP_PSK`, `CYW43_AUTH_WPA2_AES_PSK`, or `CYW43_AUTH_WPA2_MIXED_PSK` (see `CYW43_AUTH_`)

**Returns**
- 0 if the scan was started successfully, an error code otherwise

**See also**
- pico_error_codes

### 4.4.4.19. `cyw43_arch_wifi_connect_bssid_blocking`

```c
int cyw43_arch_wifi_connect_bssid_blocking (const char * ssid, const uint8_t * bssid, const char * pw, uint32_t auth)
```

Attempt to connect to a wireless access point specified by SSID and BSSID, blocking until the network is joined or a failure is detected.

**Parameters**
- **ssid**: the network name to connect to
- **bssid**: the network BSSID to connect to or NULL if ignored
- **pw**: the network password or NULL if there is no password required
- **auth**: the authorization type to use when the password is enabled. Values are `CYW43_AUTH_WPA_TKIP_PSK`, `CYW43_AUTH_WPA2_AES_PSK`, or `CYW43_AUTH_WPA2_MIXED_PSK` (see `CYW43_AUTH_`)

**Returns**
- 0 if the initialization is successful, an error code otherwise

**See also**
- pico_error_codes

### 4.4.4.20. `cyw43_arch_wifi_connect_bssid_timeout_ms`

```c
int cyw43_arch_wifi_connect_bssid_timeout_ms (const char * ssid, const uint8_t * bssid, const char * pw, uint32_t auth, uint32_t timeout)
```

Attempt to connect to a wireless access point specified by SSID and BSSID, blocking until the network is joined, a failure is detected or a timeout occurs.

**Parameters**
- **ssid**: the network name to connect to
- **bssid**: the network BSSID to connect to or NULL if ignored
- **pw**: the network password or NULL if there is no password required
- **auth**: the authorization type to use when the password is enabled. Values are `CYW43_AUTH_WPA_TKIP_PSK`, `CYW43_AUTH_WPA2_AES_PSK`, or `CYW43_AUTH_WPA2_MIXED_PSK` (see `CYW43_AUTH_`)
timeout how long to wait in milliseconds for a connection to succeed before giving up

Returns
0 if the initialization is successful, an error code otherwise

See also
pico_error_codes

4.4.4.4.21. cyw43_arch_wifi_connect_timeout_ms

int cyw43_arch_wifi_connect_timeout_ms (const char * ssid, const char * pw, uint32_t auth, uint32_t timeout)

Attempt to connect to a wireless access point, blocking until the network is joined, a failure is detected or a timeout occurs.

Parameters
ssid the network name to connect to
pw the network password or NULL if there is no password required
auth the authorization type to use when the password is enabled. Values are CYW43_AUTH_WPA_TKIP_PSK, CYW43_AUTH_WPA2_AES_PSK, or CYW43_AUTH_WPA2_MIXED_PSK (see CYW43_AUTH)
timeout how long to wait in milliseconds for a connection to succeed before giving up

Returns
0 if the initialization is successful, an error code otherwise

See also
pico_error_codes

4.4.4.5. cyw43_driver

Driver used for Pico W wireless.

4.4.4.5.1. Modules
cyw43_ll
Low Level CYW43 driver interface.

4.4.4.5.2. Macros

• #define CYW43_DEFAULT_PM cyw43_pm_value(CYW43_PM2_POWERSAVE_MODE, 200, 1, 1, 10)
• #define CYW43_AGGRESSIVE_PM cyw43_pm_value(CYW43_PM2_POWERSAVE_MODE, 2000, 1, 1, 10)
• #define CYW43_PERFORMANCE_PM cyw43_pm_value(CYW43_PM2_POWERSAVE_MODE, 20, 1, 1, 1)
• #define CYW43_COUNTRY(A, B, REV) ((unsigned char)(A) | ((unsigned char)(B) << 8) | ((REV) << 16))

4.4.4.5.3. Typedefs
typedef struct _cyw43_t cyw43_t
4.4.4.5.4. Functions

```c
void cyw43_init (cyw43_t *self)
    Initialize the driver.

void cyw43_deinit (cyw43_t *self)
    Shut the driver down.

int cyw43_ioctl (cyw43_t *self, uint32_t cmd, size_t len, uint8_t *buf, uint32_t iface)
    Send an ioctl command to cyw43.

int cyw43_send_ethernet (cyw43_t *self, int itf, size_t len, const void *buf, bool is_pbuf)
    Send a raw ethernet packet.

int cyw43_wifi_pm (cyw43_t *self, uint32_t pm)
    Set the wifi power management mode.

int cyw43_wifi_get_pm (cyw43_t *self, uint32_t *pm)
    Get the wifi power management mode.

int cyw43_wifi_link_status (cyw43_t *self, int itf)
    Get the wifi link status.

void cyw43_wifi_set_up (cyw43_t *self, int itf, bool up, uint32_t country)
    Set up and initialise wifi.

int cyw43_wifi_get_mac (cyw43_t *self, int itf, uint8_t mac[6])
    Get the mac address of the device.

int cyw43_wifi_update_multicast_filter (cyw43_t *self, uint8_t *addr, bool add)
    Add/remove multicast group address.

int cyw43_wifi_scan (cyw43_t *self, cyw43_wifi_scan_options_t *opts, void *env, int(*result_cb)(void *, const cyw43_ev_scan_result_t *))
    Perform a wifi scan for wifi networks.

static bool cyw43_wifi_scan_active (cyw43_t *self)
    Determine if a wifi scan is in progress.

int cyw43_wifi_join (cyw43_t *self, size_t ssid_len, const uint8_t *ssid, size_t key_len, const uint8_t *key, uint32_t auth_type, const uint8_t *bssid, uint32_t channel)
    Connect or join a wifi network.

int cyw43_wifi_leave (cyw43_t *self, int itf)
    Disassociate from a wifi network.

int cyw43_wifi_get_rssi (cyw43_t *self, int32_t *rssi)
    Get the signal strength (RSSI) of the wifi network.

int cyw43_wifi_get_bssid (cyw43_t *self, uint8_t bssid[6])
    Get the BSSID of the connected wifi network.

static void cyw43_wifi_ap_get_ssid (cyw43_t *self, size_t *len, const uint8_t **buf)
    Get the ssid for the access point.

static uint32_t cyw43_wifi_ap_get_auth (cyw43_t *self)
    Get the security authorisation used in AP mode.

static void cyw43_wifi_ap_set_channel (cyw43_t *self, uint32_t channel)
    Set the the channel for the access point.
```
static void cyw43_wifi_ap_set_ssid (cyw43_t *self, size_t len, const uint8_t *buf)
    Set the ssid for the access point.
static void cyw43_wifi_ap_set_password (cyw43_t *self, size_t len, const uint8_t *buf)
    Set the password for the wifi access point.
static void cyw43_wifi_ap_set_auth (cyw43_t *self, uint32_t auth)
    Set the security authorisation used in AP mode.
void cyw43_wifi_ap_get_max_stas (cyw43_t *self, int *max_stas)
    Get the maximum number of devices (STAs) that can be associated with the wifi access point.
void cyw43_wifi_ap_get_stas (cyw43_t *self, int *num_stas, uint8_t *macs)
    Get the number of devices (STAs) associated with the wifi access point.
static bool cyw43_is_initialized (cyw43_t *self)
    Determines if the cyw43 driver been initialised.
void cyw43_cb_tcpip_init (cyw43_t *self, int itf)
    Initialise the IP stack.
void cyw43_cb_tcpip_deinit (cyw43_t *self, int itf)
    Deinitialise the IP stack.
void cyw43_cb_tcpip_set_link_up (cyw43_t *self, int itf)
    Notify the IP stack that the link is up.
void cyw43_cb_tcpip_set_link_down (cyw43_t *self, int itf)
    Notify the IP stack that the link is down.
int cyw43_tcpip_link_status (cyw43_t *self, int itf)
    Get the link status.
static uint32_t cyw43_pm_value (uint8_t pm_mode, uint16_t pm2_sleep_ret_ms, uint8_t li_beacon_period, uint8_t li_dtim_period, uint8_t li_assoc)
    Return a power management value to pass to cyw43_wifi_pm.

4.4.4.5.5. Variables

cyw43_t cyw43_state

void(* cyw43_poll

uint32_t cyw43_sleep

4.4.4.5.6. CYW43 driver version as components

Current version of the CYW43 driver as major/minor/micro components

CYW43_VERSION_MAJOR
#define CYW43_VERSION_MAJOR 0

CYW43_VERSION_MINOR
#define CYW43_VERSION_MINOR 9

4.4. Networking Libraries
4.4.4.5.7. CYW43 driver version

Combined CYW43 driver version as a 32-bit number

\[
\text{CYW43\_VERSION} = \text{CYW43\_VERSION\_MAJOR} \ll 16 \mid \text{CYW43\_VERSION\_MINOR} \ll 8 \mid \text{CYW43\_VERSION\_MICRO}
\]

4.4.4.5.8. Trace flags

\[
\text{CYW43\_TRACE\_ASYNC\_EV} = 0x0001
\]

\[
\text{CYW43\_TRACE\_ETH\_TX} = 0x0002
\]

\[
\text{CYW43\_TRACE\_ETH\_RX} = 0x0004
\]

\[
\text{CYW43\_TRACE\_ETH\_FULL} = 0x0008
\]

\[
\text{CYW43\_TRACE\_MAC} = 0x0010
\]

4.4.4.5.9. Link status

See also

\text{status\_name()} to get a user readable name of the status for debug

\text{cyw43\_wifi\_link\_status()} to get the wifi status

\text{cyw43\_tcpip\_link\_status()} to get the overall link status

\[
\text{CYW43\_LINK\_DOWN} = 0
\]

\text{link is down}

\[
\text{CYW43\_LINK\_JOIN} = 1
\]

\text{Connected to wifi.}

\[
\text{CYW43\_LINK\_NOIP} = 2
\]

\text{Connected to wifi, but no IP address.}

\[
\text{CYW43\_LINK\_UP} = 3
\]

\text{Connect to wifi with an IP address.}
#define CYW43_LINK_FAIL (-1)
Connection failed.

#define CYW43_LINK_NONET (-2)
No matching SSID found (could be out of range, or down)

#define CYW43_LINK_BADAUTH (-3)
Authentication failure

4.4.4.5.10. Country codes

define CYW43_COUNTRY_WORLDWIDE CYW43_COUNTRY('X', 'X', 0)

#define CYW43_COUNTRY_AUSTRALIA CYW43_COUNTRY('A', 'U', 0)

#define CYW43_COUNTRY_AUSTRIA CYW43_COUNTRY('A', 'T', 0)

#define CYW43_COUNTRY_BELGIUM CYW43_COUNTRY('B', 'E', 0)

#define CYW43_COUNTRY_BRAZIL CYW43_COUNTRY('B', 'R', 0)

#define CYW43_COUNTRY_CANADA CYW43_COUNTRY('C', 'A', 0)

#define CYW43_COUNTRY_CHILE CYW43_COUNTRY('C', 'L', 0)

#define CYW43_COUNTRY_CHINA CYW43_COUNTRY('C', 'N', 0)

#define CYW43_COUNTRY_COLOMBIA CYW43_COUNTRY('C', 'O', 0)

#define CYW43_COUNTRY_CZECH_REPUBLIC CYW43_COUNTRY('C', 'Z', 0)

#define CYW43_COUNTRY_DENMARK CYW43_COUNTRY('D', 'K', 0)

#define CYW43_COUNTRY_ESTONIA CYW43_COUNTRY('E', 'E', 0)

#define CYW43_COUNTRY_FINLAND CYW43_COUNTRY('F', 'I', 0)
#define CYW43_COUNTRY_FRANCE CYW43_COUNTRY('F', 'R', 0)

#define CYW43_COUNTRY_GERMANY CYW43_COUNTRY('D', 'E', 0)

#define CYW43_COUNTRY_GREECE CYW43_COUNTRY('G', 'R', 0)

#define CYW43_COUNTRY_HONG_KONG CYW43_COUNTRY('H', 'K', 0)

#define CYW43_COUNTRY_HUNGARY CYW43_COUNTRY('H', 'U', 0)

#define CYW43_COUNTRY_ICELAND CYW43_COUNTRY('I', 'S', 0)

#define CYW43_COUNTRY_INDIA CYW43_COUNTRY('I', 'N', 0)

#define CYW43_COUNTRY_ISRAEL CYW43_COUNTRY('I', 'L', 0)

#define CYW43_COUNTRY_ITALY CYW43_COUNTRY('I', 'T', 0)

#define CYW43_COUNTRY_JAPAN CYW43_COUNTRY('J', 'P', 0)

#define CYW43_COUNTRY_KENYA CYW43_COUNTRY('K', 'E', 0)

#define CYW43_COUNTRY_LATVIA CYW43_COUNTRY('L', 'V', 0)

#define CYW43_COUNTRY_LIECHTENSTEIN CYW43_COUNTRY('L', 'I', 0)

#define CYW43_COUNTRY_LITHUANIA CYW43_COUNTRY('L', 'T', 0)

#define CYW43_COUNTRY_LUXEMBOURG CYW43_COUNTRY('L', 'U', 0)

#define CYW43_COUNTRY_MALAYSIA CYW43_COUNTRY('M', 'Y', 0)

#define CYW43_COUNTRY_MALTA CYW43_COUNTRY('M', 'T', 0)

#define CYW43_COUNTRY_MEXICO CYW43_COUNTRY('M', 'X', 0)

#define CYW43_COUNTRY_NETHERLANDS
```c
#define CYW43_COUNTRY_NETHERLANDS CYW43_COUNTRY('N', 'L', 0)
CYW43_COUNTRY_NEW_ZEALAND
#define CYW43_COUNTRY_NEW_ZEALAND CYW43_COUNTRY('N', 'Z', 0)
CYW43_COUNTRY_NIGERIA
#define CYW43_COUNTRY_NIGERIA CYW43_COUNTRY('N', 'G', 0)
CYW43_COUNTRY_NORWAY
#define CYW43_COUNTRY_NORWAY CYW43_COUNTRY('N', 'O', 0)
CYW43_COUNTRY_PERU
#define CYW43_COUNTRY_PERU CYW43_COUNTRY('P', 'E', 0)
CYW43_COUNTRY_PHILIPPINES
#define CYW43_COUNTRY_PHILIPPINES CYW43_COUNTRY('P', 'H', 0)
CYW43_COUNTRY_POLAND
#define CYW43_COUNTRY_POLAND CYW43_COUNTRY('P', 'L', 0)
CYW43_COUNTRY_PORTUGAL
#define CYW43_COUNTRY_PORTUGAL CYW43_COUNTRY('P', 'T', 0)
CYW43_COUNTRY_SINGAPORE
#define CYW43_COUNTRY_SINGAPORE CYW43_COUNTRY('S', 'G', 0)
CYW43_COUNTRY_SLOVAKIA
#define CYW43_COUNTRY_SLOVAKIA CYW43_COUNTRY('S', 'K', 0)
CYW43_COUNTRY_SLOVENIA
#define CYW43_COUNTRY_SLOVENIA CYW43_COUNTRY('S', 'I', 0)
CYW43_COUNTRY_SOUTH_AFRICA
#define CYW43_COUNTRY_SOUTH_AFRICA CYW43_COUNTRY('Z', 'A', 0)
CYW43_COUNTRY_SOUTH_KOREA
#define CYW43_COUNTRY_SOUTH_KOREA CYW43_COUNTRY('K', 'R', 0)
CYW43_COUNTRY_SPAIN
#define CYW43_COUNTRY_SPAIN CYW43_COUNTRY('E', 'S', 0)
CYW43_COUNTRY_SWEDEN
#define CYW43_COUNTRY_SWEDEN CYW43_COUNTRY('S', 'E', 0)
CYW43_COUNTRY_SWITZERLAND
#define CYW43_COUNTRY_SWITZERLAND CYW43_COUNTRY('C', 'H', 0)
CYW43_COUNTRY_TAIWAN
#define CYW43_COUNTRY_TAIWAN CYW43_COUNTRY('T', 'W', 0)
CYW43_COUNTRY_THAILAND
#define CYW43_COUNTRY_THAILAND CYW43_COUNTRY('T', 'H', 0)
CYW43_COUNTRY_TURKEY
#define CYW43_COUNTRY_TURKEY CYW43_COUNTRY('T', 'R', 0)
```
4.4.4.5.11. Macro Definition Documentation

**CYW43_DEFAULT_PM**

```c
#define CYW43_DEFAULT_PM cyw43_pm_value(CYW43_PM2_POWERSAVE_MODE, 200, 1, 1, 10)
```

Default power management mode.

**CYW43_AGGRESSIVE_PM**

```c
#define CYW43_AGGRESSIVE_PM cyw43_pm_value(CYW43_PM2_POWERSAVE_MODE, 2000, 1, 1, 10)
```

Aggressive power management mode for optimal power usage at the cost of performance.

**CYW43_PERFORMANCE_PM**

```c
#define CYW43_PERFORMANCE_PM cyw43_pm_value(CYW43_PM2_POWERSAVE_MODE, 20, 1, 1, 1)
```

Performance power management mode where more power is used to increase performance.

**CYW43_COUNTRY**

```c
#define CYW43_COUNTRY(A, B, REV) ((unsigned char)(A) | ((unsigned char)(B) << 8) | ((REV) << 16))
```

create a country code from the two character country and revision number

4.4.4.5.12. Typedef Documentation

**cyw43_t**

```c
typedef struct _cyw43_t cyw43_t
```

4.4.4.5.13. Function Documentation

**cyw43_cb_tcpip_deinit**

```c
void cyw43_cb_tcpip_deinit (cyw43_t * self, int itf)
```

Deinitialise the IP stack.

This method must be provided by the network stack interface It is called to close the IP stack and free resources.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td>the driver state object. This should always be <code>&amp;cyw43_state</code></td>
</tr>
<tr>
<td>itf</td>
<td>the interface used, either CYW43_ITF_STA or CYW43_ITF_AP</td>
</tr>
</tbody>
</table>

**cyw43_cb_tcpip_init**

```c
void cyw43_cb_tcpip_init (cyw43_t * self, int itf)
```

Initialise the IP stack.

This method must be provided by the network stack interface It is called to initialise the IP stack.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td>the driver state object. This should always be <code>&amp;cyw43_state</code></td>
</tr>
<tr>
<td>itf</td>
<td>the interface used, either CYW43_ITF_STA or CYW43_ITF_AP</td>
</tr>
</tbody>
</table>
**cyw43_cb_tcpip_set_link_down**

`void cyw43_cb_tcpip_set_link_down (cyw43_t * self, int itf)`

Notify the IP stack that the link is down.

**Parameters**

- `self`: the driver state object. This should always be `&cyw43_state`
- `itf`: the interface used, either CYW43_ITF_STA or CYW43_ITF_AP

**cyw43_cb_tcpip_set_link_up**

`void cyw43_cb_tcpip_set_link_up (cyw43_t * self, int itf)`

Notify the IP stack that the link is up.

**Parameters**

- `self`: the driver state object. This should always be `&cyw43_state`
- `itf`: the interface used, either CYW43_ITF_STA or CYW43_ITF_AP

**cyw43_deinit**

`void cyw43_deinit (cyw43_t * self)`

Shut the driver down.

This method will close the network interfaces, and free up resources

**Parameters**

- `self`: the driver state object. This should always be `&cyw43_state`

**cyw43_init**

`void cyw43_init (cyw43_t * self)`

Initialize the driver.

This method must be called before using the driver

**Parameters**

- `self`: the driver state object. This should always be `&cyw43_state`

**cyw43_ioctl**

`int cyw43_ioctl (cyw43_t * self, uint32_t cmd, size_t len, uint8_t * buf, uint32_t iface)`

Send an ioctl command to cyw43.

This method sends a command to cyw43.

**Parameters**

- `self`: the driver state object. This should always be `&cyw43_state`
- `cmd`: the command to send
- `len`: the amount of data to send with the command
- `buf`: a buffer containing the data to send
**cyw43_is_initialized**

```c
static bool cyw43_is_initialized (cyw43_t * self) [inline], [static]
```

Determines if the cyw43 driver been initialised.

**Parameters**

- `self` the driver state object. This should always be &cyw43_state

**Returns**

True if the cyw43 driver has been initialised

---

**cyw43_pm_value**

```c
static uint32_t cyw43_pm_value (uint8_t pm_mode, uint16_t pm2_sleep_ret_ms, uint8_t li_beacon_period, uint8_t li_dtim_period, uint8_t li_assoc) [inline], [static]
```

Generate the power management value to pass to cyw43_wifi_pm.

<table>
<thead>
<tr>
<th>pm_mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYW43_NO POWERSAVE_MODE</td>
<td>No power saving</td>
</tr>
<tr>
<td>CYW43_PM1 POWERSAVE_MODE</td>
<td>Aggressive power saving which reduces wifi throughp</td>
</tr>
<tr>
<td>CYW43_PM2 POWERSAVE_MODE</td>
<td>Power saving with High throughput (preferred). Saves</td>
</tr>
<tr>
<td></td>
<td>power when there is no wifi activity for some time.</td>
</tr>
</tbody>
</table>

See also

- CYW43_DEFAULT_PM
- CYW43_AGGRESSIVE_PM
- CYW43_PERFORMANCE_PM

**Parameters**

- `pm_mode` Power management mode
- `pm2_sleep_ret_ms` The maximum time to wait before going back to sleep for CYW43_PM2 POWERSAVE_MODE mode. Value measured in milliseconds and must be between 10 and 2000ms and divisible by 10
- `li_beacon_period` Wake period is measured in beacon periods
- `li_dtim_period` Wake interval measured in DTIMs. If this is set to 0, the wake interval is measured in beacon periods
- `li_assoc` Wake interval sent to the access point

---

**cyw43_send_ethernet**

```c
int cyw43_send_ethernet (cyw43_t * self, int itf, size_t len, const void * buf, bool is_pbuf)
```

Send a raw ethernet packet.

This method sends a raw ethernet packet.

---

4.4. Networking Libraries
self the driver state object. This should always be &cyw43_state
itf interface to use, either CYW43_ITF_STA or CYW43_ITF_AP
len the amount of data to send
buf the data to send
is_pbuf true if buf points to an lwip struct pbuf

Returns
0 on success

cyw43_tcpip_link_status

int cyw43_tcpip_link_status (cyw43_t * self, int itf)

Get the link status.

Returns the status of the link which is a superset of the wifi link status returned by cyw43_wifi_link_status

NOTE
If the link status is negative it indicates an error

<table>
<thead>
<tr>
<th>link status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYW43_LINK_DOWN</td>
<td>Wifi down</td>
</tr>
<tr>
<td>CYW43_LINK_JOIN</td>
<td>Connected to wifi</td>
</tr>
<tr>
<td>CYW43_LINK_NOIP</td>
<td>Connected to wifi, but no IP address</td>
</tr>
<tr>
<td>CYW43_LINK_UP</td>
<td>Connect to wifi with an IP address</td>
</tr>
<tr>
<td>CYW43_LINK_FAIL</td>
<td>Connection failed</td>
</tr>
<tr>
<td>CYW43_LINK_NONET</td>
<td>No matching SSID found (could be out of range, or down)</td>
</tr>
<tr>
<td>CYW43_LINK_BADAUTH</td>
<td>Authentication failure</td>
</tr>
</tbody>
</table>

Parameters

self the driver state object. This should always be &cyw43_state
itf the interface for which to return the link status, should be CYW43_ITF_STA or CYW43_ITF_AP

Returns

A value representing the link status

cyw43_wifi_ap_get_auth

static uint32_t cyw43_wifi_ap_get_auth (cyw43_t * self) [inline], [static]

Get the security authorisation used in AP mode.

For access point (AP) mode, this method can be used to get the security authorisation mode.

Parameters

self the driver state object. This should always be &cyw43_state

Returns

the current security authorisation mode for the access point

cyw43_wifi_ap_get_max_stas

void cyw43_wifi_ap_get_max_stas (cyw43_t * self, int * max_stas)
Get the maximum number of devices (STAs) that can be associated with the wifi access point.

For access point (AP) mode, this method can be used to get the maximum number of devices that can be connected to the wifi access point.

**Parameters**

- `self`: the driver state object. This should always be `cyw43_state`
- `max_stas`: Returns the maximum number of devices (STAs) that can be connected to the access point

### cyw43_wifi_ap_get_ssid

```
static void cyw43_wifi_ap_get_ssid (cyw43_t * self, size_t * len, const uint8_t ** buf) [inline], [static]
```

Get the ssid for the access point.

For access point (AP) mode, this method can be used to get the SSID name of the wifi access point.

**Parameters**

- `self`: the driver state object. This should always be `cyw43_state`
- `len`: Returns the length of the AP SSID name
- `buf`: Returns a pointer to an internal buffer containing the AP SSID name

### cyw43_wifi_ap_get_stas

```
void cyw43_wifi_ap_get_stas (cyw43_t * self, int * num_stas, uint8_t * macs)
```

Get the number of devices (STAs) associated with the wifi access point.

For access point (AP) mode, this method can be used to get the number of devices and mac addresses of devices connected to the wifi access point.

**Parameters**

- `self`: the driver state object. This should always be `cyw43_state`
- `num_stas`: Returns the number of devices (STA) connected to the access point
- `macs`: Returns the mac addresses of devices (STA) connected to the access point. The supplied buffer should have enough room for 6 bytes per mac address. Call `cyw43_wifi_ap_get_max_stas` to determine how many mac addresses can be returned.

### cyw43_wifi_ap_set_auth

```
static void cyw43_wifi_ap_set_auth (cyw43_t * self, uint32_t auth) [inline], [static]
```

Set the security authorisation used in AP mode.

For access point (AP) mode, this method can be used to set how access to the access point is authorised.

**Auth mode**

<table>
<thead>
<tr>
<th>Auth mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYW43_AUTH_OPEN</td>
<td>Use an open access point with no authorisation required</td>
</tr>
<tr>
<td>CYW43_AUTH_WPA_TKIP_PSK</td>
<td>Use WPA authorisation</td>
</tr>
<tr>
<td>CYW43_AUTH_WPA2_AES_PSK</td>
<td>Use WPA2 (preferred)</td>
</tr>
<tr>
<td>CYW43_AUTH_WPA2_MIXED_PSK</td>
<td>Use WPA2/WPA mixed (currently treated the same as CYW43_AUTH_WPA2_AES_PSK)</td>
</tr>
</tbody>
</table>

**Parameters**

- `self`: the driver state object. This should always be `cyw43_state`
- `auth`: Auth mode for the access point

### cyw43_wifi_ap_set_channel
static void cyw43_wifi_ap_set_channel (cyw43_t * self, uint32_t channel) [inline], [static]

Set the channel for the access point.

For access point (AP) mode, this method can be used to set the channel used for the wifi access point.

Parameters

- `self`: the driver state object. This should always be &cyw43_state
- `channel`: Wifi channel to use for the wifi access point

cyw43_wifi_ap_set_password

static void cyw43_wifi_ap_set_password (cyw43_t * self, size_t len, const uint8_t * buf) [inline], [static]

Set the password for the wifi access point.

For access point (AP) mode, this method can be used to set the password for the wifi access point.

Parameters

- `self`: the driver state object. This should always be &cyw43_state
- `len`: The length of the AP password
- `buf`: A buffer containing the AP password

cyw43_wifi_ap_set_ssid

static void cyw43_wifi_ap_set_ssid (cyw43_t * self, size_t len, const uint8_t * buf) [inline], [static]

Set the ssid for the access point.

For access point (AP) mode, this method can be used to set the SSID name of the wifi access point.

Parameters

- `self`: the driver state object. This should always be &cyw43_state
- `len`: The length of the AP SSID name
- `buf`: A buffer containing the AP SSID name

cyw43_wifi_get_bssid

int cyw43_wifi_get_bssid (cyw43_t * self, uint8_t bssid)

Get the BSSID of the connected wifi network.

Parameters

- `self`: the driver state object. This should always be &cyw43_state
- `bssid`: a buffer to receive the BSSID

Returns

0 on success

cyw43_wifi_get_mac

int cyw43_wifi_get_mac (cyw43_t * self, int itf, uint8_t mac)

Get the mac address of the device.

This method returns the mac address of the interface.

Parameters

- `self`: the driver state object. This should always be &cyw43_state
- `itf`: the interface to use, either CYW43_ITF_STA or CYW43_ITF_AP
- `mac`: a buffer to receive the mac address
Returns

0 on success

cyw43_wifi_get_pm

int cyw43_wifi_get_pm (cyw43_t * self, uint32_t * pm)

Get the wifi power management mode.

This method gets the power management mode used by cyw43. The value is expressed as an unsigned integer 0x00adbrrm where, m = pm_mode Power management mode rr = pm2_sleep_ret (in units of 10ms) b = li_beacon_period d = li_dtim_period a = li_assoc

See also

cyw43_pm_value for an explanation of these values This should be called after cyw43_wifi_set_up

Parameters

self the driver state object. This should always be &cyw43_state

pm Power management value

Returns

0 on success

cyw43_wifi_get_rssi

int cyw43_wifi_get_rssi (cyw43_t * self, int32_t * rssi)

Get the signal strength (RSSI) of the wifi network.

For STA (client) mode, returns the signal strength or RSSI of the wifi network. An RSSI value of zero is returned if you call this function before a network is connected.

Parameters

self the driver state object. This should always be &cyw43_state

rssi a pointer to which the returned RSSI value is stored.

Returns

0 on success

cyw43_wifi_join

int cyw43_wifi_join (cyw43_t * self, size_t ssid_len, const uint8_t * ssid, size_t key_len, const uint8_t * key, uint32_t auth_type, const uint8_t * bssid, uint32_t channel)

Connect or join a wifi network.

Connect to a wifi network in STA (client) mode After success is returned, periodically call cyw43_wifi_link_status or cyw43_tcpip_link_status, to query the status of the link. It can take a many seconds to connect to fully join a network.

NOTE

Call cyw43_wifi_leave to disassociate from a wifi network.

Parameters

self the driver state object. This should always be &cyw43_state

ssid_len the length of the wifi network name

ssid A buffer containing the wifi network name

key_len The length of the wifi password
key A buffer containing the wifi password
auth_type Auth type,

See also
CYW43_AUTH_

Parameters
bssid the mac address of the access point to connect to. This can be NULL.
channel Used to set the band of the connection. This is only used if bssid is non NULL.

Returns
0 on success
cyw43_wifi_leave

int cyw43_wifi_leave (cyw43_t * self, int itf)
Disassociate from a wifi network.
This method disassociates from a wifi network.

Parameters
self the driver state object. This should always be &cyw43_state
itf The interface to disconnect, either CYW43_ITF_STA or CYW43_ITF_AP

Returns
0 on success
cyw43_wifi_link_status

int cyw43_wifi_link_status (cyw43_t * self, int itf)
Get the wifi link status.

Returns the status of the wifi link.

<table>
<thead>
<tr>
<th>link status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYW43_LINK_DOWN</td>
<td>Wifi down</td>
</tr>
<tr>
<td>CYW43_LINK_JOIN</td>
<td>Connected to wifi</td>
</tr>
<tr>
<td>CYW43_LINK_FAIL</td>
<td>Connection failed</td>
</tr>
<tr>
<td>CYW43_LINK_NONET</td>
<td>No matching SSID found (could be out of range, or down)</td>
</tr>
<tr>
<td>CYW43_LINK_BADAUTH</td>
<td>Authentication failure</td>
</tr>
</tbody>
</table>

NOTE
If the link status is negative it indicates an error The wifi link status for the interface CYW43_ITF_AP is always CYW43_LINK_DOWN

Parameters
self the driver state object. This should always be &cyw43_state
itf the interface to use, should be CYW43_ITF_STA or CYW43_ITF_AP

Returns
A integer value representing the link status
cyw43_wifi_pm

```c
int cyw43_wifi_pm (cyw43_t * self, uint32_t pm)
```

Set the wifi power management mode.
This method sets the power management mode used by cyw43. This should be called after cyw43_wifi_set_up

See also
cyw43_pm_value
CYW43_DEFAULT_PM
CYW43_AGGRESSIVE_PM
CYW43_PERFORMANCE_PM

Parameters

self the driver state object. This should always be &cyw43_state
pm Power management value

Returns

0 on success

cyw43_wifi_scan

```c
int cyw43_wifi_scan (cyw43_t * self, cyw43_wifi_scan_options_t * opts, void * env, int(*)(void *, const cyw43_ev_scan_result_t *) result_cb)
```

Perform a wifi scan for wifi networks.
Start a scan for wifi networks. Results are returned via the callback.

NOTE

The scan is complete when cyw43_wifi_scan_active return false

Parameters

self the driver state object. This should always be &cyw43_state
opts An instance of cyw43_wifi_scan_options_t. Values in here are currently ignored.
env Pointer passed back in the callback
result_cb Callback for wifi scan results, see cyw43_ev_scan_result_t

Returns

0 on success

cyw43_wifi_scan_active

```c
static bool cyw43_wifi_scan_active (cyw43_t * self) [inline], [static]
```

Determine if a wifi scan is in progress.
This method tells you if the scan is still in progress

Parameters

self the driver state object. This should always be &cyw43_state

Returns

true if a wifi scan is in progress

cyw43_wifi_set_up
void cyw43_wifi_set_up (cyw43.t * self, int itf, bool up, uint32_t country)

Set up and initialise wifi.

This method turns on wifi and sets the country for regulation purposes. The power management mode is initialised to CYW43_DEFAULT_PM For CYW43_ITF_AP, the access point is enabled. For CYW43_ITF_STA, the TCP/IP stack is reinitialised.

Parameters

self the driver state object. This should always be &cyw43_state
itf the interface to use either CYW43_ITF_STA or CYW43_ITF_AP
up true to enable the link. Set to false to disable AP mode. Setting the up parameter to false for CYW43_ITF_STA is ignored.
country the country code, see CYW43_COUNTRY_

cyw43_wifi_update_multicast_filter

int cyw43_wifi_update_multicast_filter (cyw43.t * self, uint8_t * addr, bool add)

Add/remove multicast group address.

This method adds/removes an address from the multicast filter, allowing frames sent to this group to be received.

Parameters

self the driver state object. This should always be &cyw43_state
addr a buffer containing a group mac address
add true to add the address, false to remove it

Returns

0 on success

4.4.4.5.14. Variable Documentation

cyw43_state
cyw43.t cyw43_state

cyw43_poll
void(* cyw43_poll) (void)

cyw43_sleep
uint32_t cyw43_sleep

4.4.4.5.15. cyw43_ll

Low Level CYW43 driver interface.

Macros

• #define CYW43_IOCTL_GET_SSID (0x32)
• #define CYW43_IOCTL_GET_CHANNEL (0x3a)
• #define CYW43_IOCTL_SET_DISASSOC (0x69)
• #define CYW43_IOCTL_GET_ANTDIV (0x7e)
• #define CYW43_IOCTL_SET_ANTDIV (0x81)
• #define CYW43_IOCTL_SET_MONITOR (0xd9)
• #define CYW43_IOCTL_GET_RSSI (0xfe)
• #define CYW43_IOCTL_GET_VAR (0x20c)
• #define CYW43_IOCTL_SET_VAR (0x20f)
• #define CYW43_EV_SET_SSID (0)
• #define CYW43_EV_JOIN (1)
• #define CYW43_EV_AUTH (3)
• #define CYW43_EV_DEAUTH (5)
• #define CYW43_EV_DEAUTH_IND (6)
• #define CYW43_EV_ASSOC (7)
• #define CYW43_EV_DISASSOC (11)
• #define CYW43_EV_DISASSOC_IND (12)
• #define CYW43_EV_LINK (16)
• #define CYW43_EV_PRUNE (23)
• #define CYW43_EV_PSK_SUP (46)
• #define CYW43_EV_ESCAN_RESULT (69)
• #define CYW43_EV_CSA_COMPLETE_IND (80)
• #define CYW43_EV_ASSOC_REQ_IE (87)
• #define CYW43_EV_ASSOC_RESP_IE (88)
• #define CYW43_STATUS_SUCCESS (0)
• #define CYW43_STATUS_FAIL (1)
• #define CYW43_STATUS_TIMEOUT (2)
• #define CYW43_STATUS_NO_NETWORKS (3)
• #define CYW43_STATUS_ABORT (4)
• #define CYW43_STATUS_NO_ACK (5)
• #define CYW43_STATUS_UNSOLICITED (6)
• #define CYW43_STATUS_ATTEMPT (7)
• #define CYW43_STATUS_PARTIAL (8)
• #define CYW43_STATUS_NEWSCAN (9)
• #define CYW43_STATUS_NEWASSOC (10)
• #define CYW43_SUP_DISCONNECTED (0)
• #define CYW43_SUP_CONNECTING (1)
• #define CYW43_SUP_IDREQUIRED (2)
• #define CYW43_SUP_AUTHENTICATING (3)
• #define CYW43_SUP_AUTHENTICATED (4)
• #define CYW43_SUP_KEYXCHANGE (5)
• #define CYW43_SUP_KEYED (6)
- #define CYW43_SUP_TIMEOUT (7)
- #define CYW43_SUP_LAST_BASIC_STATE (8)
- #define CYW43_SUP_KEYXCHANGE_WAIT_M1 CYW43_SUP_AUTHENTICATED
- #define CYW43_SUP_KEYXCHANGE_PREP_M2 CYW43_SUP_KEYXCHANGE
- #define CYW43_SUP_KEYXCHANGE_WAIT_M3 CYW43_SUP_LAST_BASIC_STATE
- #define CYW43_SUP_KEYXCHANGE_PREP_M4 (9)
- #define CYW43_SUP_KEYXCHANGE_WAIT_G1 (10)
- #define CYW43_SUP_KEYXCHANGE_PREP_G2 (11)
- #define CYW43_REASON_INITIAL_ASSOC (0)
- #define CYW43_REASON_LOW_RSSI (1)
- #define CYW43_REASON_DEAUTH (2)
- #define CYW43_REASON_DISASSOC (3)
- #define CYW43_REASON_BCNS_LOST (4)
- #define CYW43_REASON_FAST_ROAM_FAILED (5)
- #define CYW43_REASON_DIRECTED_ROAM (6)
- #define CYW43_REASON_TSPEC_REJECTED (7)
- #define CYW43_REASON_BETTER_AP (8)
- #define CYW43_REASON_PRUNE_ENCR_MISMATCH (1)
- #define CYW43_REASON_PRUNE_BCAST_BSSID (2)
- #define CYW43_REASON_PRUNE_MAC_DENY (3)
- #define CYW43_REASON_PRUNE_MAC_NA (4)
- #define CYW43_REASON_PRUNE_REG_PASSV (5)
- #define CYW43_REASON_PRUNE_SPCT_MGMT (6)
- #define CYW43_REASON_PRUNE_RADAR (7)
- #define CYW43_REASON_PRUNE_RSN_MISMATCH (8)
- #define CYW43_REASON_PRUNE_NO_COMMON_RATES (9)
- #define CYW43_REASON_PRUNE_basic_RATES (10)
- #define CYW43_REASON_PRUNE_CCXFAST_PREVAP (11)
- #define CYW43_REASON_PRUNE_CIPHER_NA (12)
- #define CYW43_REASON_PRUNE_KNOWN_STA (13)
- #define CYW43_REASON_PRUNE_CCXFAST_DROAM (14)
- #define CYW43_REASON_PRUNE_MDS_PEER (15)
- #define CYW43_REASON_PRUNE_QBSS_LOAD (16)
- #define CYW43_REASON_PRUNE_HOME_AP (17)
- #define CYW43_REASON_PRUNE_AP_BLOCKED (18)
- #define CYW43_REASON_PRUNE_NO_DIAG_SUPPORT (19)
- #define CYW43_REASON_SUP_OTHER (0)
• #define CYW43_REASON_SUP_DECRYPT_KEY_DATA (1)
• #define CYW43_REASON_SUP_BAD_UCAST_WEP128 (2)
• #define CYW43_REASON_SUP_BAD_UCAST_WEP40 (3)
• #define CYW43_REASON_SUP_UNSUP_KEY_LEN (4)
• #define CYW43_REASON_SUP_PW_KEY_CIPHER (5)
• #define CYW43_REASON_SUP_MSG3_TOO_MANY_IE (6)
• #define CYW43_REASON_SUP_MSG3_IE_MISMATCH (7)
• #define CYW43_REASON_SUP_NO_INSTALL_FLAG (8)
• #define CYW43_REASON_SUP_MSG3_NO_GTK (9)
• #define CYW43_REASON_SUP_GRP_KEY_CIPHER (10)
• #define CYW43_REASON_SUP_GRP_MSG1_NO_GTK (11)
• #define CYW43_REASON_SUP_GTK_DECRYPT_FAIL (12)
• #define CYW43_REASON_SUP_SEND_FAIL (13)
• #define CYW43_REASON_SUP_DEAUTH (14)
• #define CYW43_REASON_SUP_WPA_PSK_TMO (15)
• #define CYW43_WPA_AUTH_PSK (0x0004)
• #define CYW43_WPA2_AUTH_PSK (0x0080)
• #define CYW43_NO_POWERSAVE_MODE (0)
• #define CYW43_PM1_POWERSAVE_MODE (1)
• #define CYW43_PM2_POWERSAVE_MODE (2)
• #define CYW43_BUS_MAX_BLOCK_SIZE 16384
• #define CYW43_BACKPLANE_READ_PAD_LEN_BYTES 0
• #define CYW43_LL_STATE_SIZE_WORDS 526 + 5
• #define CYW43_CHANNEL_NONE (0xffffffff)

**Typedefs**

```c
typedef struct _cyw43_async_event_t cyw43_async_event_t

typedef struct _cyw43_ll_t cyw43_ll_t
```

**Functions**

```c
void cyw43_ll_init (cyw43_ll_t *self, void *cb_data)

void cyw43_ll_deinit (cyw43_ll_t *self)

int cyw43_ll_bus_init (cyw43_ll_t *self, const uint8_t *mac)

void cyw43_ll_bus_sleep (cyw43_ll_t *self, bool can_sleep)
```

4.4. Networking Libraries
void cyw43_ll_process_packets (cyw43_ll_t *self)

int cyw43_ll_ioctl (cyw43_ll_t *self, uint32_t cmd, size_t len, uint8_t *buf, uint32_t iface)

int cyw43_ll_send_ethernet (cyw43_ll_t *self, int itf, size_t len, const void *buf, bool is_pbuf)

int cyw43_ll_wifi_on (cyw43_ll_t *self, uint32_t country)

int cyw43_ll_wifi_pm (cyw43_ll_t *self, uint32_t pm, uint32_t pm_sleep_ret, uint32_t li_bcn, uint32_t li_dtim, uint32_t li_assoc)

int cyw43_ll_wifi_get_pm (cyw43_ll_t *self, uint32_t *pm, uint32_t *pm_sleep_ret, uint32_t *li_bcn, uint32_t *li_dtim, uint32_t *li_assoc)

int cyw43_ll_wifi_scan (cyw43_ll_t *self, cyw43_wifi_scan_options_t *opts)

int cyw43_ll_wifi_join (cyw43_ll_t *self, size_t ssid_len, const uint8_t *ssid, size_t key_len, const uint8_t *key, uint32_t auth_type, const uint8_t *bssid, uint32_t channel)

void cyw43_ll_wifi_set_wpa_auth (cyw43_ll_t *self)

void cyw43_ll_wifi_rejoin (cyw43_ll_t *self)

int cyw43_ll_wifi_get_bssid (cyw43_ll_t *self, uint8_t *bssid)

int cyw43_ll_wifi_ap_init (cyw43_ll_t *self, size_t ssid_len, const uint8_t *ssid, uint32_t auth, size_t key_len, const uint8_t *key, uint32_t channel)

int cyw43_ll_wifi_ap_set_up (cyw43_ll_t *self, bool up)

int cyw43_ll_wifi_ap_get_stas (cyw43_ll_t *self, int *num_stas, uint8_t *macs)

int cyw43_ll_wifi_get_mac (cyw43_ll_t *self, uint8_t *addr)

int cyw43_ll_wifi_update_multicast_filter (cyw43_ll_t *self, uint8_t *addr, bool add)

bool cyw43_ll_has_work (cyw43_ll_t *self)

bool cyw43_ll_bt_has_work (cyw43_ll_t *self)

int cyw43_cb_read_host_interrupt_pin (void *cb_data)
void cyw43_cb_ensure_awake (void *cb_data)

void cyw43_cb_process_async_event (void *cb_data, const cyw43_async_event_t *ev)

void cyw43_cb_process_ethernet (void *cb_data, int itf, size_t len, const uint8_t *buf)

void cyw43_ll_write_backplane_reg (cyw43_ll_t *self_in, uint32_t addr, uint32_t val)

uint32_t cyw43_ll_read_backplane_reg (cyw43_ll_t *self_in, uint32_t addr)

int cyw43_ll_write_backplane_mem (cyw43_ll_t *self_in, uint32_t addr, uint32_t len, const uint8_t *buf)

int cyw43_ll_read_backplane_mem (cyw43_ll_t *self_in, uint32_t addr, uint32_t len, uint8_t *buf)

anonymous enum

Network interface types.

<table>
<thead>
<tr>
<th>Enumerator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYW43_ITF_STA</td>
<td>Client interface STA mode.</td>
</tr>
<tr>
<td>CYW43_ITF_AP</td>
<td>Access point (AP) interface mode.</td>
</tr>
</tbody>
</table>

 typedef struct _cyw43_ev_scan_result_t cyw43_ev_scan_result_t
 Structure to return wifi scan results.

define struct _cyw43_wifi_scan_options_t cyw43_wifi_scan_options_t
 wifi scan options passed to cyw43_wifi_scan

Authorization types

Used when setting up an access point, or connecting to an access point

CYW43_AUTH_OPEN

#define CYW43_AUTH_OPEN (0)

No authorisation required (open)

CYW43_AUTH_WPA_TKIP_PSK

#define CYW43_AUTH_WPA_TKIP_PSK (0x00200002)

WPA authorisation.

CYW43_AUTH_WPA2_AES_PSK

#define CYW43_AUTH_WPA2_AES_PSK (0x00400004)

WPA2 authorisation (preferred)

CYW43_AUTH_WPA2_MIXED_PSK

#define CYW43_AUTH_WPA2_MIXED_PSK (0x00400006)
WPA2/WPA mixed authorisation

Macro Definition Documentation

CYW43_IOCTL_GET_SSID
#define CYW43_IOCTL_GET_SSID (0x32)

CYW43_IOCTL_GET_CHANNEL
#define CYW43_IOCTL_GET_CHANNEL (0x3a)

CYW43_IOCTL_SET_DISASSOC
#define CYW43_IOCTL_SET_DISASSOC (0x69)

CYW43_IOCTL_GET_ANTDIV
#define CYW43_IOCTL_GET_ANTDIV (0x7e)

CYW43_IOCTL_SET_ANTDIV
#define CYW43_IOCTL_SET_ANTDIV (0x81)

CYW43_IOCTL_SET_MONITOR
#define CYW43_IOCTL_SET_MONITOR (0xd9)

CYW43_IOCTL_GET_RSSI
#define CYW43_IOCTL_GET_RSSI (0xfe)

CYW43_IOCTL_GET_VAR
#define CYW43_IOCTL_GET_VAR (0x20c)

CYW43_IOCTL_SET_VAR
#define CYW43_IOCTL_SET_VAR (0x20f)

CYW43_EV_SET_SSID
#define CYW43_EV_SET_SSID (0)

CYW43_EV_JOIN
#define CYW43_EV_JOIN (1)

CYW43_EV_AUTH
#define CYW43_EV_AUTH (3)

CYW43_EV_DEAUTH
#define CYW43_EV_DEAUTH (5)

CYW43_EV_DEAUTH_IND
#define CYW43_EV_DEAUTH_IND (6)

CYW43_EV_ASSOC
#define CYW43_EV_ASSOC (7)

CYW43_EV_DISASSOC
#define CYW43_EV_DISASSOC (11)

CYW43_EV_DISASSOC_IND
#define CYW43_EV_DISASSOC_IND (12)
CYW43_SUP_CONNECTING
#define CYW43_SUP_CONNECTING (1)

CYW43_SUP_IDREQUIRED
#define CYW43_SUP_IDREQUIRED (2)

CYW43_SUP_AUTHENTICATING
#define CYW43_SUP_AUTHENTICATING (3)

CYW43_SUP_AUTHENTICATED
#define CYW43_SUP_AUTHENTICATED (4)

CYW43_SUP_KEYXCHANGE
#define CYW43_SUP_KEYXCHANGE (5)

CYW43_SUP_KEYED
#define CYW43_SUP_KEYED (6)

CYW43_SUP_TIMEOUT
#define CYW43_SUP_TIMEOUT (7)

CYW43_SUP_LAST_BASIC_STATE
#define CYW43_SUP_LAST_BASIC_STATE (8)

CYW43_SUP_KEYXCHANGE_WAIT_M1
#define CYW43_SUP_KEYXCHANGE_WAIT_M1 CYW43_SUP_AUTHENTICATED

CYW43_SUP_KEYXCHANGE_PREP_M2
#define CYW43_SUP_KEYXCHANGE_PREP_M2 CYW43_SUP_KEYXCHANGE

CYW43_SUP_KEYXCHANGE_WAIT_M3
#define CYW43_SUP_KEYXCHANGE_WAIT_M3 CYW43_SUP_LAST_BASIC_STATE

CYW43_SUP_KEYXCHANGE_PREP_M4
#define CYW43_SUP_KEYXCHANGE_PREP_M4 (9)

CYW43_SUP_KEYXCHANGE_WAIT_G1
#define CYW43_SUP_KEYXCHANGE_WAIT_G1 (10)

CYW43_SUP_KEYXCHANGE_PREP_G2
#define CYW43_SUP_KEYXCHANGE_PREP_G2 (11)

CYW43_REASON_INITIAL_ASSOC
#define CYW43_REASON_INITIAL_ASSOC (0)

CYW43_REASON_LOW_RSSI
#define CYW43_REASON_LOW_RSSI (1)

CYW43_REASON_DEAUTH
#define CYW43_REASON_DEAUTH (2)

CYW43_REASON_DISASSOC
#define CYW43_REASON_DISASSOC (3)

CYW43_REASON_BCNS_LOST
#define CYW43_REASON_BCNS_LOST (4)

CYW43_REASON_FAST_ROAM_FAILED

#define CYW43_REASON_FAST_ROAM_FAILED (5)

CYW43_REASON_DIRECTED_ROAM

#define CYW43_REASON_DIRECTED_ROAM (6)

CYW43_REASON_TSPEC_REJECTED

#define CYW43_REASON_TSPEC_REJECTED (7)

CYW43_REASON_BETTER_AP

#define CYW43_REASON_BETTER_AP (8)

CYW43_REASON_PRUNE_ENCR_MISMATCH

#define CYW43_REASON_PRUNE_ENCR_MISMATCH (1)

CYW43_REASON_PRUNE_BCAST_BSSID

#define CYW43_REASON_PRUNE_BCAST_BSSID (2)

CYW43_REASON_PRUNE_MAC_DENY

#define CYW43_REASON_PRUNE_MAC_DENY (3)

CYW43_REASON_PRUNE_MAC_NA

#define CYW43_REASON_PRUNE_MAC_NA (4)

CYW43_REASON_PRUNE_REG_PASSV

#define CYW43_REASON_PRUNE_REG_PASSV (5)

CYW43_REASON_PRUNE_SPCT_MGMT

#define CYW43_REASON_PRUNE_SPCT_MGMT (6)

CYW43_REASON_PRUNE_RADAR

#define CYW43_REASON_PRUNE_RADAR (7)

CYW43_REASON_RSN_MISMATCH

#define CYW43_REASON_RSN_MISMATCH (8)

CYW43_REASON_PRUNE_NO_COMMON_RATES

#define CYW43_REASON_PRUNE_NO_COMMON_RATES (9)

CYW43_REASON_PRUNE_BASIC_RATES

#define CYW43_REASON_PRUNE_BASIC_RATES (10)

CYW43_REASON_PRUNE_CCXFAST_PREVAP

#define CYW43_REASON_PRUNE_CCXFAST_PREVAP (11)

CYW43_REASON_PRUNE_CIPHER_NA

#define CYW43_REASON_PRUNE_CIPHER_NA (12)

CYW43_REASON_PRUNE_KNOWN_STA

#define CYW43_REASON_PRUNE_KNOWN_STA (13)

CYW43_REASON_PRUNE_CCXFAST_DROAM

#define CYW43_REASON_PRUNE_CCXFAST_DROAM (14)
#define CYW43_REASON_PRUNE_WDS_PEER (15)

#define CYW43_REASON_PRUNE_QBSS_LOAD (16)

#define CYW43_REASON_PRUNE_HOME_AP (17)

#define CYW43_REASON_PRUNE_AP_BLOCKED (18)

#define CYW43_REASON_PRUNE_NO_DIAG_SUPPORT (19)

#define CYW43_REASON_SUP_OTHER (0)

#define CYW43_REASON_SUP_DECRYPT_KEY_DATA (1)

#define CYW43_REASON_SUP_BAD_UCAST_WEP128 (2)

#define CYW43_REASON_SUP_BAD_UCAST_WEP40 (3)

#define CYW43_REASON_SUP_UNSUP_KEY_LEN (4)

#define CYW43_REASON_SUP_PW_KEY_CIPHER (5)

#define CYW43_REASON_SUP_MSG3_TOO_MANY_IE (6)

#define CYW43_REASON_SUP_MSG3_IE_MISMATCH (7)

#define CYW43_REASON_SUP_NO_INSTALL_FLAG (8)

#define CYW43_REASON_SUP_MSG3_NO_GTK (9)

#define CYW43_REASON_SUP_GRP_KEY_CIPHER (10)

#define CYW43_REASON_SUP_GRP_MSG1_NO_GTK (11)

#define CYW43_REASON_SUP_GTK_DECRYPT_FAIL (12)

#define CYW43_REASON_SUP_SEND_FAIL (13)
```c
#define CYW43_REASON_SUP_SEND_FAIL (13)

CYW43_REASON_SUP_DEAUTH
#define CYW43_REASON_SUP_DEAUTH (14)

CYW43_REASON_SUP_WPA_PSK_TMO
#define CYW43_REASON_SUP_WPA_PSK_TMO (15)

CYW43_WPA_AUTH_PSK
#define CYW43_WPA_AUTH_PSK (0x0004)

CYW43_WPA2_AUTH_PSK
#define CYW43_WPA2_AUTH_PSK (0x0080)

CYW43_NO_POWERSAVE_MODE
#define CYW43_NO_POWERSAVE_MODE (0)

Power save mode parameter passed to cyw43_ll_wifi_pm.
No Powersave mode

CYW43_PM1_POWERSAVE_MODE
#define CYW43_PM1_POWERSAVE_MODE (1)

Powersave mode on specified interface without regard for throughput reduction.

CYW43_PM2_POWERSAVE_MODE
#define CYW43_PM2_POWERSAVE_MODE (2)

Powersave mode on specified interface with High throughput.

CYW43_BUS_MAX_BLOCK_SIZE
#define CYW43_BUS_MAX_BLOCK_SIZE 16384

CYW43_BACKPLANE_READ_PAD_LEN_BYTES
#define CYW43_BACKPLANE_READ_PAD_LEN_BYTES 0

CYW43_LL_STATE_SIZE_WORDS
#define CYW43_LL_STATE_SIZE_WORDS 526 + 5

CYW43_CHANNEL_NONE
#define CYW43_CHANNEL_NONE (0xffffffff)

To indicate no specific channel when calling cyw43_ll_wifi_join with bssid specified.
No Channel specified (use the AP’s channel)

Typedef Documentation
cyw43_async_event_t
typedef struct _cyw43_async_event_t cyw43_async_event_t
cyw43_ll_t
typedef struct _cyw43_ll_t cyw43_ll_t

Function Documentation
cyw43_cb_ensure_awake
void cyw43_cb_ensure_awake (void * cb_data)
```
cyw43_cb_process_async_event

void cyw43_cb_process_async_event (void * cb_data, const cyw43_async_event_t * ev)

cyw43_cb_process_ethernet

void cyw43_cb_process_ethernet (void * cb_data, int itf, size_t len, const uint8_t * buf)

cyw43_cb_read_host_interrupt_pin

int cyw43_cb_read_host_interrupt_pin (void * cb_data)

cyw43_ll_bt_has_work

bool cyw43_ll_bt_has_work (cyw43_ll_t * self)

cyw43_ll_bus_init

int cyw43_ll_bus_init (cyw43_ll_t * self, const uint8_t * mac)

cyw43_ll_bus_sleep

void cyw43_ll_bus_sleep (cyw43_ll_t * self, bool can_sleep)

cyw43_ll_deinit

void cyw43_ll_deinit (cyw43_ll_t * self)

cyw43_ll_has_work

bool cyw43_ll_has_work (cyw43_ll_t * self)

cyw43_ll_init

void cyw43_ll_init (cyw43_ll_t * self, void * cb_data)

cyw43_ll_ioctl

int cyw43_ll_ioctl (cyw43_ll_t * self, uint32_t cmd, size_t len, uint8_t * buf, uint32_t iface)

cyw43_ll_process_packets

void cyw43_ll_process_packets (cyw43_ll_t * self)

cyw43_ll_read_backplane_mem

int cyw43_ll_read_backplane_mem (cyw43_ll_t * self_in, uint32_t addr, uint32_t len, uint8_t * buf)

cyw43_ll_read_backplane_reg

uint32_t cyw43_ll_read_backplane_reg (cyw43_ll_t * self_in, uint32_t addr)

cyw43_ll_send_ethernet

int cyw43_ll_send_ethernet (cyw43_ll_t * self, int itf, size_t len, const void * buf, bool is_pbuf)

cyw43_ll_wifi_ap_get_stas

int cyw43_ll_wifi_ap_get_stas (cyw43_ll_t * self, int * num_stas, uint8_t * macs)

cyw43_ll_wifi_ap_init

int cyw43_ll_wifi_ap_init (cyw43_ll_t * self, size_t ssid_len, const uint8_t * ssid, uint32_t auth, size_t key_len, const uint8_t * key, uint32_t channel)

cyw43_ll_wifi_ap_set_up

int cyw43_ll_wifi_ap_set_up (cyw43_ll_t * self, bool up)

cyw43_ll_wifi_get_bssid

int cyw43_ll_wifi_get_bssid (cyw43_ll_t * self_in, uint8_t * bssid)

cyw43_ll_wifi_get_mac
4.5. Runtime Infrastructure

Libraries that are used to provide efficient implementation of certain language level and C library functions, as well as CMake INTERFACE libraries abstracting the compilation and link steps in the SDK

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boot_stage2</td>
<td>Second stage boot loaders responsible for setting up external flash.</td>
</tr>
<tr>
<td>pico_base</td>
<td></td>
</tr>
<tr>
<td>pico_binary_info</td>
<td></td>
</tr>
<tr>
<td>pico_bit_ops</td>
<td></td>
</tr>
<tr>
<td>pico_bootrom</td>
<td></td>
</tr>
<tr>
<td>pico_bootsel_via_double_reset</td>
<td></td>
</tr>
<tr>
<td>pico_cxx_options</td>
<td>non-code library controlling C++ related compile options</td>
</tr>
<tr>
<td>pico_divider</td>
<td></td>
</tr>
<tr>
<td>pico_double</td>
<td></td>
</tr>
</tbody>
</table>
### 4.5.1. boot_stage2

Second stage boot loaders responsible for setting up external flash.

### 4.5.2. pico_base

#### 4.5.2.1. Detailed Description

Core types and macros for the Raspberry Pi Pico SDK. This header is intended to be included by all source code as it includes configuration headers and overrides in the correct order.

This header may be included by assembly code.

#### 4.5.2.2. Enumerations

```c
enum pico_error_codes {
    PICO_OK = 0, PICO_ERROR_NONE = 0, PICO_ERROR_TIMEOUT = -1, PICO_ERROR_GENERIC = -2,
    PICO_ERROR_NO_DATA = -3, PICO_ERROR_NOT_PERMITTED = -4, PICO_ERROR_INVALID_ARG = -5,
    PICO_ERROR_IO = -6, PICO_ERROR_BADAUTH = -7, PICO_ERROR_CONNECT_FAILED = -8,
    PICO_ERROR_INSUFFICIENT_RESOURCES = -9
}
```

Common return codes from pico_sdk methods that return a status.

#### 4.5.2.3. Enumeration Type Documentation

##### 4.5.2.3.1. pico_error_codes

```c
enum pico_error_codes
```

Common return codes from pico_sdk methods that return a status.
4.5.3. pico_binary_info

4.5.3.1. Detailed Description

Binary info is intended for embedding machine readable information with the binary in FLASH.

Example uses include:

- Program identification / information
- Pin layouts
- Included features
- Identifying flash regions used as block devices/storage

4.5.3.2. Macros

- #define bi_decl (_decl) __bi_mark_enclosure _decl; __bi_decl(__bi_ptr_lineno_var_name, &__bi_lineno_var_name.core, ".binary_info.keep.", __used);

- #define bi_decl_if_func_used (_decl) ({__bi_mark_enclosure _decl; __bi_decl(__bi_ptr_lineno_var_name, &__bi_lineno_var_name.core, ".binary_info.", ); *(const volatile uint8_t *)&__bi_ptr_lineno_var_name;});

4.5.3.3. Macro Definition Documentation

4.5.3.3.1. bi_decl

#define bi_decl (_decl) __bi_mark_enclosure _decl; __bi_decl(__bi_ptr_lineno_var_name, &__bi_lineno_var_name.core, ".binary_info.keep.", __used);

Declare some binary information that will be included if the contain source file/line is compiled into the binary

4.5.3.3.2. bi_decl_if_func_used

#define bi_decl_if_func_used (_decl) ({__bi_mark_enclosure _decl; __bi_decl(__bi_ptr_lineno_var_name, &__bi_lineno_var_name.core, ".binary_info.", ); *(const volatile uint8_t *)&__bi_ptr_lineno_var_name;});

Declare some binary information that will be included if the function containing the decl is linked into the binary. The SDK uses –gc-sections, so functions that are never called will be removed by the linker, and any associated binary information declared this way will also be stripped

4.5.4. pico_bit_ops

4.5.4.1. Detailed Description

Optimized bit manipulation functions. Additionally provides replacement implementations of the compiler built-ins
__builtin_popcount, __builtin_clz and __builtin_ctz

4.5.4.2. Functions

uint32_t __rev (uint32_t bits)

Reverse the bits in a 32 bit word.
uint64_t __revll (uint64_t bits)

Reverse the bits in a 64 bit double word.

4.5.4.3. Function Documentation

4.5.4.3.1. __rev

uint32_t __rev (uint32_t bits)

Reverse the bits in a 32 bit word.

Parameters

bits  32 bit input

Returns

the 32 input bits reversed

4.5.4.3.2. __revll

uint64_t __revll (uint64_t bits)

Reverse the bits in a 64 bit double word.

Parameters

bits  64 bit input

Returns

the 64 input bits reversed

4.5.5. pico_bootrom

4.5.5.1. Detailed Description

Access to functions and data in the RP2040 bootrom

This header may be included by assembly code

4.5.5.2. Macros

- #define ROM_TABLE_CODE(c1, c2) ((c1) | ((c2) << 8))

4.5.5.3. Functions

static uint32_t rom_table_code (uint8_t c1, uint8_t c2)

Return a bootrom lookup code based on two ASCII characters.

void * rom_func_lookup (uint32_t code)

Lookup a bootrom function by code.

void * rom_data_lookup (uint32_t code)

Lookup a bootrom address by code.
bool rom_funcs_lookup (uint32_t *table, unsigned int count)

    Helper function to lookup the addresses of multiple bootrom functions.

static __force_inline void * rom_func_lookup_inline (uint32_t code)

    Lookup a bootrom function by code. This method is forcibly inlined into the caller for FLASH/RAM sensitive code usage.

static void reset_usb_boot (uint32_t usb_activity_gpio_pin_mask, uint32_t disable_interface_mask)

    Reboot the device into BOOTSEL mode.

4.5.5.4. Macro Definition Documentation

4.5.5.4.1. ROM_TABLE_CODE

#define ROM_TABLE_CODE(c1, c2) ((c1) | ((c2) << 8))

    Return a bootrom lookup code based on two ASCII characters.
    These codes are uses to lookup data or function addresses in the bootrom

Parameters
    c1     the first character
    c2     the second character

Returns
    the 'code' to use in rom_func_lookup() or rom_data_lookup()

4.5.5.5. Function Documentation

4.5.5.5.1. reset_usb_boot

static void reset_usb_boot (uint32_t usb_activity_gpio_pin_mask, uint32_t disable_interface_mask) [inline], [static]

    Reboot the device into BOOTSEL mode.

    This function reboots the device into the BOOTSEL mode (‘usb boot’). Facilities are provided to enable an ‘activity light’ via GPIO attached LED for the USB Mass Storage Device, and to limit the USB interfaces exposed.

Parameters
    usb_activity_gpio_pin_mask   0 No pins are used as per a cold boot. Otherwise a single bit set indicating which GPIO pin should be set to output and raised whenever there is mass storage activity from the host.
    disable_interface_mask      value to control exposed interfaces
        • 0 To enable both interfaces (as per a cold boot)
        • 1 To disable the USB Mass Storage Interface
        • 2 To disable the USB PICOBOOT Interface

4.5.5.5.2. 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

4.5.5.5.3. 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

4.5.5.5.4. rom_func_lookup_inline

static __force_inline void * rom_func_lookup_inline (uint32_t code) [static]

Lookup a bootrom function by code. This method is forcibly inlined into the caller for FLASH/RAM sensitive code usage.

Parameters

code  the code

Returns

a pointer to the function, or NULL if the code does not match any bootrom function

4.5.5.5.5. rom_funcs_lookup

bool rom_funcs_lookup (uint32_t * table, unsigned int count)

Helper function to lookup the addresses of multiple bootrom functions.

This method looks up the 'codes' in the table, and convert each table entry to the looked up function pointer, if there is a function for that code in the bootrom.

Parameters

table  an IN/OUT array, elements are codes on input, function pointers on success.
count  the number of elements in the table

Returns

true if all the codes were found, and converted to function pointers, false otherwise

4.5.5.5.6. rom_table_code

static uint32_t rom_table_code (uint8_t c1, uint8_t c2) [inline], [static]

Return a bootrom lookup code based on two ASCII characters.

These codes are uses to lookup data or function addresses in the bootrom

Parameters
Returns
the 'code' to use in \texttt{rom\_func\_lookup()} or \texttt{rom\_data\_lookup()}. 

4.5.6. pico\_bootsel\_via\_double\_reset

4.5.6.1. Detailed Description

When the 'pico\_bootsel\_via\_double\_reset' library is linked, a function is injected before main() which will detect when the system has been reset twice in quick succession, and enter the USB ROM bootloader (BOOTSEL mode) when this happens. This allows a double tap of a reset button on a development board to be used to enter the ROM bootloader, provided this library is always linked.

4.5.7. pico\_cxx\_options

\texttt{non-code library controlling C++ related compile options}

4.5.8. pico\_divider

4.5.8.1. Detailed Description

Optimized 32 and 64 bit division functions accelerated by the RP2040 hardware divider. Additionally provides integration with the C / and \% operators

4.5.8.2. Functions

\begin{verbatim}
int32_t div_s32s32 (int32_t a, int32_t b)  
    Integer divide of two signed 32-bit values.

static int32_t divmod_s32s32_rem (int32_t a, int32_t b, int32_t *rem)  
    Integer divide of two signed 32-bit values, with remainder.

divmod_result_t divmod_s32s32 (int32_t a, int32_t b)  
    Integer divide of two signed 32-bit values.

uint32_t div_u32u32 (uint32_t a, uint32_t b)  
    Integer divide of two unsigned 32-bit values.

static uint32_t divmod_u32u32_rem (uint32_t a, uint32_t b, uint32_t *rem)  
    Integer divide of two unsigned 32-bit values, with remainder.

divmod_result_t divmod_u32u32 (uint32_t a, uint32_t b)  
    Integer divide of two unsigned 32-bit values.

int64_t div_s64s64 (int64_t a, int64_t b)  
    Integer divide of two signed 64-bit values.
\end{verbatim}
int64_t divmod_s64s64_rem (int64_t a, int64_t b, int64_t *rem)
    Integer divide of two signed 64-bit values, with remainder.

int64_t divmod_s64s64 (int64_t a, int64_t b)
    Integer divide of two signed 64-bit values.

uint64_t div_u64u64 (uint64_t a, uint64_t b)
    Integer divide of two unsigned 64-bit values.

uint64_t divmod_u64u64_rem (uint64_t a, uint64_t b, uint64_t *rem)
    Integer divide of two unsigned 64-bit values, with remainder.

uint64_t divmod_u64u64 (uint64_t a, uint64_t b)
    Integer divide of two signed 64-bit values.

int32_t div_s32s32_unsafe (int32_t a, int32_t b)
    Unsafe integer divide of two signed 32-bit values.

int32_t divmod_s32s32_rem_unsafe (int32_t a, int32_t b, int32_t *rem)
    Unsafe integer divide of two signed 32-bit values, with remainder.

int64_t divmod_s32s32Unsafe (int32_t a, int32_t b)
    Unsafe integer divide of two unsigned 32-bit values.

uint32_t div_u32u32_unsafe (uint32_t a, uint32_t b)
    Unsafe integer divide of two unsigned 32-bit values.

uint32_t divmod_u32u32_rem_unsafe (uint32_t a, uint32_t b, uint32_t *rem)
    Unsafe integer divide of two unsigned 32-bit values, with remainder.

uint64_t divmod_u32u32_unsafe (uint32_t a, uint32_t b)
    Unsafe integer divide of two signed 64-bit values.

int64_t divmod_s64s64_unsafe (int64_t a, int64_t b)
    Unsafe integer divide of two signed 64-bit values.

int64_t divmod_s64s64_rem_unsafe (int64_t a, int64_t b, int64_t *rem)
    Unsafe integer divide of two signed 64-bit values, with remainder.

int64_t divmod_s64s64_unsafe (int64_t a, int64_t b)
    Unsafe integer divide of two signed 64-bit values.

uint64_t div_u64u64_unsafe (uint64_t a, uint64_t b)
    Unsafe integer divide of two unsigned 64-bit values.

uint64_t divmod_u64u64_rem_unsafe (uint64_t a, uint64_t b, uint64_t *rem)
    Unsafe integer divide of two unsigned 64-bit values, with remainder.

uint64_t divmod_u64u64_unsafe (uint64_t a, uint64_t b)
    Unsafe integer divide of two signed 64-bit values.

4.5.8.3. Function Documentation

4.5.8.3.1. div_s32s32

int32_t div_s32s32 (int32_t a, int32_t b)
    Integer divide of two signed 32-bit values.
Parameters
a Dividend
b Divisor

Returns
quotient

4.5.8.3.2. div_s32s32_unsafe
int32_t div_s32s32_unsafe (int32_t a, int32_t b)
Unsafe integer divide of two signed 32-bit values.
Parameters
a Dividend
b Divisor
Returns
quotient
Do not use in interrupts

4.5.8.3.3. div_s64s64
int64_t div_s64s64 (int64_t a, int64_t b)
Integer divide of two signed 64-bit values.
Parameters
a Dividend
b Divisor
Returns
Quotient

4.5.8.3.4. div_s64s64_unsafe
int64_t div_s64s64_unsafe (int64_t a, int64_t b)
Unsafe integer divide of two signed 64-bit values.
Parameters
a Dividend
b Divisor
Returns
Quotient
Do not use in interrupts

4.5.8.3.5. div_u32u32
uint32_t div_u32u32 (uint32_t a, uint32_t b)
Integer divide of two unsigned 32-bit values.

**Parameters**
- a  Dividend
- b  Divisor

**Returns**
Quotient

### 4.5.8.3.6. div_u32u32_unsafe

```c
tuint32_t div_u32u32_unsafe (uint32_t a, uint32_t b)
```

Unsafe integer divide of two unsigned 32-bit values.

**Parameters**
- a  Dividend
- b  Divisor

**Returns**
Quotient
Do not use in interrupts

### 4.5.8.3.7. div_u64u64

```c
tuint64_t div_u64u64 (uint64_t a, uint64_t b)
```

Integer divide of two unsigned 64-bit values.

**Parameters**
- a  Dividend
- b  Divisor

**Returns**
Quotient

### 4.5.8.3.8. div_u64u64_unsafe

```c
tuint64_t div_u64u64_unsafe (uint64_t a, uint64_t b)
```

Unsafe integer divide of two unsigned 64-bit values.

**Parameters**
- a  Dividend
- b  Divisor

**Returns**
Quotient
Do not use in interrupts
4.5.8.3.9. divmod_s32s32

divmod_result_t divmod_s32s32 (int32_t a, int32_t b)

Integer divide of two signed 32-bit values.

**Parameters**
- `a` Dividend
- `b` Divisor

**Returns**
quotient in low word/r0, remainder in high word/r1

4.5.8.3.10. divmod_s32s32_rem

static int32_t divmod_s32s32_rem (int32_t a, int32_t b, int32_t * rem) [inline], [static]

Integer divide of two signed 32-bit values, with remainder.

**Parameters**
- `a` Dividend
- `b` Divisor
- `rem` The remainder of dividend/divisor

**Returns**
Quotient result of dividend/divisor

4.5.8.3.11. divmod_s32s32_rem_unsafe

int32_t divmod_s32s32_rem_unsafe (int32_t a, int32_t b, int32_t * rem)

Unsafe integer divide of two signed 32-bit values, with remainder.

**Parameters**
- `a` Dividend
- `b` Divisor
- `rem` The remainder of dividend/divisor

**Returns**
Quotient result of dividend/divisor
Do not use in interrupts

4.5.8.3.12. divmod_s32s32_unsafe

int64_t divmod_s32s32_unsafe (int32_t a, int32_t b)

Unsafe integer divide of two unsigned 32-bit values.

**Parameters**
- `a` Dividend
- `b` Divisor

**Returns**
quotient in low word/r0, remainder in high word/r1
Do not use in interrupts

4.5.8.3.13. divmod_s64s64

```c
int64_t divmod_s64s64 (int64_t a, int64_t b)
```
Integer divide of two signed 64-bit values.

**Parameters**
- `a` Dividend
- `b` Divisor

**Returns**
- Quotient in result (r0,r1), remainder in regs (r2, r3)

4.5.8.3.14. divmod_s64s64_rem

```c
int64_t divmod_s64s64_rem (int64_t a, int64_t b, int64_t * rem)
```
Integer divide of two signed 64-bit values, with remainder.

**Parameters**
- `a` Dividend
- `b` Divisor
- `rem` The remainder of dividend/divisor

**Returns**
- Quotient result of dividend/divisor

4.5.8.3.15. divmod_s64s64_rem_unsafe

```c
int64_t divmod_s64s64_rem_unsafe (int64_t a, int64_t b, int64_t * rem)
```
Unsafe integer divide of two signed 64-bit values, with remainder.

**Parameters**
- `a` Dividend
- `b` Divisor
- `rem` The remainder of dividend/divisor

**Returns**
- Quotient result of dividend/divisor
- Do not use in interrupts

4.5.8.3.16. divmod_s64s64_unsafe

```c
int64_t divmod_s64s64_unsafe (int64_t a, int64_t b)
```
Unsafe integer divide of two signed 64-bit values.

**Parameters**
**4.5.8.3.17. divmod_u32u32**

\[
divmod\_result\_t \ divmod\_u32u32 (uint32\_t a, uint32\_t b)
\]

Integer divide of two unsigned 32-bit values.

**Parameters**
- \(a\) Dividend
- \(b\) Divisor

**Returns**
- Quotient in low word/r0, remainder in high word/r1

**4.5.8.3.18. divmod_u32u32\_rem**

\[
static \ uint32\_t \ divmod\_u32u32\_rem (uint32\_t a, uint32\_t b, uint32\_t * \ rem) [inline], [static]
\]

Integer divide of two unsigned 32-bit values, with remainder.

**Parameters**
- \(a\) Dividend
- \(b\) Divisor
- \(\text{rem}\) The remainder of dividend/divisor

**Returns**
- Quotient result of dividend/divisor

**4.5.8.3.19. divmod_u32u32\_rem\_unsafe**

\[
uint32\_t \ divmod\_u32u32\_rem\_unsafe (uint32\_t a, uint32\_t b, uint32\_t * \ rem)
\]

Unsafe integer divide of two unsigned 32-bit values, with remainder.

**Parameters**
- \(a\) Dividend
- \(b\) Divisor
- \(\text{rem}\) The remainder of dividend/divisor

**Returns**
- Quotient result of dividend/divisor
- Do not use in interrupts
4.5.8.3.20. divmod_u32u32_unsafe

```c
uint64_t divmod_u32u32_unsafe (uint32_t a, uint32_t b)
```

Unsafe integer divide of two unsigned 32-bit values.

**Parameters**

- `a`: Dividend
- `b`: Divisor

**Returns**

Quotient in low word/r0, remainder in high word/r1

Do not use in interrupts

4.5.8.3.21. divmod_u64u64

```c
uint64_t divmod_u64u64 (uint64_t a, uint64_t b)
```

Integer divide of two signed 64-bit values.

**Parameters**

- `a`: Dividend
- `b`: Divisor

**Returns**

Quotient in result (r0,r1), remainder in regs (r2, r3)

4.5.8.3.22. divmod_u64u64_rem

```c
uint64_t divmod_u64u64_rem (uint64_t a, uint64_t b, uint64_t * rem)
```

Integer divide of two unsigned 64-bit values, with remainder.

**Parameters**

- `a`: Dividend
- `b`: Divisor
- `rem`: The remainder of dividend/divisor

**Returns**

Quotient result of dividend/divisor

4.5.8.3.23. divmod_u64u64_rem_unsafe

```c
uint64_t divmod_u64u64_rem_unsafe (uint64_t a, uint64_t b, uint64_t * rem)
```

Unsafe integer divide of two unsigned 64-bit values, with remainder.

**Parameters**

- `a`: Dividend
- `b`: Divisor
- `rem`: The remainder of dividend/divisor

**Returns**
Raspberry Pi Pico C/C++ SDK

Quotient result of dividend/divisor
Do not use in interrupts

4.5.8.3.24. divmod_u64u64_unsafe
uint64_t divmod_u64u64_unsafe (uint64_t a, uint64_t b)

Unsafe integer divide of two signed 64-bit values.
Parameters
a

Dividend

b

Divisor

Returns
quotient in result (r0,r1), remainder in regs (r2, r3)
Do not use in interrupts

4.5.9. pico_double
4.5.9.1. Detailed Description
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

4.5.10. pico_float
4.5.10.1. Detailed Description
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

4.5. Runtime Infrastructure

373


• powinf, 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

4.5.11. pico_int64_ops

4.5.11.1. Detailed Description
Optimized replacement implementations of the compiler built-in 64 bit multiplication
This library does not provide any additional functions

4.5.12. pico_malloc

4.5.12.1. Detailed Description
Multi-core safety for malloc, calloc and free
This library does not provide any additional functions

4.5.13. pico_mem_ops

4.5.13.1. Detailed Description
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

4.5.14. pico_platform

4.5.14.1. Detailed Description
Macros and definitions (and functions when included by non assembly code) for the RP2 family device / architecture to provide a common abstraction over low level compiler / platform specifics.
This header may be included by assembly code

4.5.14.2. Macros
• #define __isr
• #define __after_data(group) __attribute__((section(".after_data." group)))
• #define __not_in_flash(group) __attribute__((section(".time_critical." group)))
4.5.14.3. Functions

static void __breakpoint (void)
Execute a breakpoint instruction.

static __always_inline void __compiler_memory_barrier (void)
Ensure that the compiler does not move memory access across this method call.

void panic_unsupported (void)
Panics with the message "Unsupported".

void panic (const char *fmt,…)
Displays a panic message and halts execution.

uint8_t rp2040_chip_version (void)
Returns the RP2040 chip revision number.

static uint8_t rp2040_rom_version (void)
Returns the RP2040 rom version number.

static __always_inline void tight_loop_contents (void)
No-op function for the body of tight loops.

static __always_inline int32_t __mul_instruction (int32_t a, int32_t b)
Multiply two integers using an assembly MUL instruction.

static __always_inline uint __get_current_exception (void)
Get the current exception level on this core.

static void busy_wait_at_least_cycles (uint32_t minimum_cycles)
Helper method to busy-wait for at least the given number of cycles.

static __always_inline uint get_core_num (void)
Get the current core number.
4.5.14.4. Macro Definition Documentation

4.5.14.4.1. __isr

```c
#define __isr
Marker for an interrupt handler.
For example an IRQ handler function called my_interrupt_handler:

void __isr my_interrupt_handler(void) {
```

4.5.14.4.2. __after_data

```c
#define __after_data(group) __attribute__((section(".after_data." group)))
Section attribute macro for placement in RAM after the .data section.
For example a 400 element uint32_t array placed after the .data section:

uint32_t __after_data("my_group_name") a_big_array[400];
```

The section attribute is .after_data.<group>

Parameters

- **group** a string suffix to use in the section name to distinguish groups that can be linker garbage-collected independently

4.5.14.4.3. __not_in_flash

```c
#define __not_in_flash(group) __attribute__((section(".time_critical." group)))
Section attribute macro for placement not in flash (i.e. in RAM)
For example a 3 element uint32_t array placed in RAM (even though it is static const):

static const uint32_t __not_in_flash("my_group_name") an_array[3];
```

The section attribute is .time_critical.<group>

Parameters

- **group** a string suffix to use in the section name to distinguish groups that can be linker garbage-collected independently

4.5.14.4.4. __scratch_x

```c
#define __scratch_x(group) __attribute__((section(".scratch_x." group)))
Section attribute macro for placement in the SRAM bank 4 (known as "scratch X")
Scratch X is commonly used for critical data and functions accessed only by one core (when only one core is accessing the RAM bank, there is no opportunity for stalls)
```
For example a `uint32_t` variable placed in "scratch X"

```c
uint32_t __scratch_x("my_group_name") foo = 23;
```

The section attribute is `.scratch_x.<group>`

**Parameters**

- `group` a string suffix to use in the section name to distinguish groups that can be linker garbage-collected independently

### 4.5.14.4.5. __scratch_y

```c
#define __scratch_y(group) __attribute__((section(".scratch_y." group)))
```

Section attribute macro for placement in the SRAM bank 5 (known as "scratch Y")

Scratch Y is commonly used for critical data and functions accessed only by one core (when only one core is accessing the RAM bank, there is no opportunity for stalls)

For example a `uint32_t` variable placed in "scratch Y"

```c
uint32_t __scratch_y("my_group_name") foo = 23;
```

The section attribute is `.scratch_y.<group>`

**Parameters**

- `group` a string suffix to use in the section name to distinguish groups that can be linker garbage-collected independently

### 4.5.14.4.6. __uninitialized_ram

```c
#define __uninitialized_ram(group) __attribute__((section(".uninitialized_data." #group)))
```

Section attribute macro for data that is to be left uninitialized.

Data marked this way will retain its value across a reset (normally uninitialized data - in the .bss section) is initialized to zero during runtime initialization

For example a `uint32_t` `foo` that will retain its value if the program is restarted by reset.

```c
uint32_t __uninitialized_ram(foo);
```

The section attribute is `.uninitialized_data.<group>`

**Parameters**

- `group` a string suffix to use in the section name to distinguish groups that can be linker garbage-collected independently

### 4.5.14.4.7. __in_flash

```c
#define __in_flash(group) __attribute__((section(".flashdata." group)))
```

Section attribute macro for placement in flash even in a COPY_TO_RAM binary.
For example a `uint32_t` variable explicitly placed in flash (it will hard fault if you attempt to write it!)

```c
uint32_t __in_flash("my_group_name") foo = 23;
```

The section attribute is `.flashdata.<group>`

**Parameters**

- `group` a string suffix to use in the section name to distinguish groups that can be linker garbage-collected independently

### 4.5.14.4.8. __not_in_flash_func

```c
#define __not_in_flash_func(func_name) __not_in_flash(__STRING(func_name)) func_name
```

Indicates a function should not be stored in flash.

Decorates a function name, such that the function will execute from RAM (assuming it is not inlined into a flash function by the compiler)

For example a function called `my_func` taking an int parameter:

```c
void __not_in_flash_func(my_func)(int some_arg) {
```

The function is placed in the `.time_critical.<func_name>` linker section

**See also**

- `__no_inline_not_in_flash_func`

### 4.5.14.4.9. __time_critical_func

```c
#define __time_critical_func(func_name) __not_in_flash_func(func_name)
```

Indicates a function is time/latency critical and should not run from flash.

Decorates a function name, such that the function will execute from RAM (assuming it is not inlined into a flash function by the compiler) to avoid possible flash latency. Currently this macro is identical in implementation to `__not_in_flash_func`, however the semantics are distinct and a `__time_critical_func` may in the future be treated more specially to reduce the overhead when calling such function from a flash function.

For example a function called `my_func` taking an int parameter:

```c
void __time_critical(my_func)(int some_arg) {
```

The function is placed in the `.time_critical.<func_name>` linker section

**See also**

- `__not_in_flash_func`

### 4.5.14.4.10. __no_inline_not_in_flash_func

```c
#define __no_inline_not_in_flash_func(func_name) __noinline __not_in_flash_func(func_name)
```

Indicate a function should not be stored in flash and should not be inlined.
Decorates a function name, such that the function will execute from RAM, explicitly marking it as noinline to prevent it being inlined into a flash function by the compiler.

For example a function called `my_func` taking an int parameter:

```c
void __no_inline_not_in_flash_func(my_func)(int some_arg) {
```

The function is placed in the `.time_critical.<func_name>` linker section.

4.5.14.4.11. __force_inline

#define __force_inline __always_inline

Attribute to force inlining of a function regardless of optimization level.

For example `my_function` here will always be inlined:

```c
int __force_inline my_function(int x) {
```

4.5.14.4.12. count_of

#define count_of(a) (sizeof(a)/sizeof((a)[0]))

Macro to determine the number of elements in an array.

4.5.14.4.13. MAX

#define MAX(a, b) ((a)>(b)?(a):(b))

Macro to return the maximum of two comparable values.

4.5.14.4.14. MIN

#define MIN(a, b) ((b)>(a)?(a):(b))

Macro to return the minimum of two comparable values.

4.5.14.4.15. host_safe_hw_ptr

#define host_safe_hw_ptr(x) ((uintptr_t)(x))

Macro for converting memory addresses to 32 bit addresses suitable for DMA.

This is just a cast to `uintptr_t` on the RP2040, however you may want to use this when developing code that also runs in "host" mode. If the host mode is 64 bit and you are embedding data pointers in other data (e.g. DMA chaining), then there is a need in 'host' mode to convert a 64 bit native pointer to a 32 bit value for storage, which can be done using this macro.

4.5.14.4.16. __fast_mul
multiply two integer values using the fastest method possible

Efficiently multiplies value a by possibly constant value b.

If b is known to be constant and not zero or a power of 2, then a mul instruction is used rather than gcc’s default which is often a slow combination of shifts and adds. If b is a power of 2 then a single shift is of course preferable and will be used

Parameters

- a the first operand
- b the second operand

Returns

a * b

4.5.14.4.17. __check_type_compatible

#define __check_type_compatible(type_a, type_b) static_assert(__builtin_types_compatible_p(type_a, type_b), __STRING(type_a) " is not compatible with " __STRING(type_b));

Utility macro to assert two types are equivalent.

This macro can be useful in other macros along with typeof to assert that two parameters are of equivalent type (or that a single parameter is of an expected type)

4.5.14.5. Function Documentation

4.5.14.5.1. __breakpoint

static void __breakpoint (void) [inline], [static]

Execute a breakpoint instruction.

4.5.14.5.2. __compiler_memory_barrier

static __always_inline void __compiler_memory_barrier (void) [static]

Ensure that the compiler does not move memory access across this method call.

For example in the following code:

*some_memory_location = var_a;
__compiler_memory_barrier();
uint32_t var_b = *some_other_memory_location

The compiler will not move the load from some_other_memory_location above the memory barrier (which it otherwise might - even above the memory store!)
4.5.14.5.3. __get_current_exception

static __always_inline uint __get_current_exception (void) [static]
Get the current exception level on this core.

Returns
the exception number if the CPU is handling an exception, or 0 otherwise

4.5.14.5.4. __mul_instruction

static __always_inline int32_t __mul_instruction (int32_t a, int32_t b) [static]
Multiply two integers using an assembly MUL instruction.
This multiplies a by b using multiply instruction using the ARM mul instruction regardless of values (the compiler might otherwise choose to perform shifts/adds), i.e. this is a 1 cycle operation.

Parameters
  a  the first operand
  b  the second operand

Returns
a * b

4.5.14.5.5. busy_wait_at_least_cycles

static void busy_wait_at_least_cycles (uint32_t minimum_cycles) [inline], [static]
Helper method to busy-wait for at least the given number of cycles.
This method is useful for introducing very short delays.
This method busy-waits in a tight loop for the given number of system clock cycles. The total wait time is only accurate to within 2 cycles, and this method uses a loop counter rather than a hardware timer, so the method will always take longer than expected if an interrupt is handled on the calling core during the busy-wait; you can of course disable interrupts to prevent this.
You can use clock_get_hz(clk_sys) to determine the number of clock cycles per second if you want to convert an actual time duration to a number of cycles.

Parameters
  minimum_cycles  the minimum number of system clock cycles to delay for

4.5.14.5.6. get_core_num

static __always_inline uint get_core_num (void) [static]
Get the current core number.

Returns
The core number the call was made from

4.5.14.5.7. panic

void panic (const char * fmt, …)
Displays a panic message and halts execution.
An attempt is made to output the message to all registered STDOUT drivers after which this method executes a BKPT instruction.

Parameters

- `fmt` format string (printf-like)
- `...` printf-like arguments

4.5.14.5.8. panic_unsupported

```c
void panic_unsupported (void)
```

Panic with the message “Unsupported”.

See also

panic

4.5.14.5.9. rp2040_chip_version

```c
uint8_t rp2040_chip_version (void)
```

Returns the RP2040 chip revision number.

Returns

the RP2040 chip revision number (1 for B0/B1, 2 for B2)

4.5.14.5.10. rp2040_rom_version

```c
static uint8_t rp2040_rom_version (void) [inline], [static]
```

Returns the RP2040 rom version number.

Returns

the RP2040 rom version number (1 for RP2040-B0, 2 for RP2040-B1, 3 for RP2040-B2)

4.5.14.5.11. tight_loop_contents

```c
static __always_inline void tight_loop_contents (void) [static]
```

No-op function for the body of tight loops.

No-op function intended to be called by any tight hardware polling loop. Using this ubiquitously makes it much easier to find tight loops, but also in the future #ifdef-ed support for lockup debugging might be added

4.5.15. pico_printf

4.5.15.1. Detailed Description

Compact replacement for printf by Marco Paland (info@paland.com)
4.5.16. pico_runtime

4.5.16.1. Detailed Description

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

4.5.17. pico_stdio

4.5.17.1. Detailed Description

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

4.5.17.2. 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).

4.5.17.3. Functions

bool stdio_init_all (void)
Initialize all of the present standard stdio types that are linked into the binary.

void stdio_flush (void)
Flushes any buffered output.

int getchar_timeout_us (uint32_t timeout_us)
Return a character from stdin if there is one available within a timeout.

void stdio_set_driver_enabled (stdio_driver_t *driver, bool enabled)
Adds or removes a driver from the list of active drivers used for input/output.

void stdio_filter_driver (stdio_driver_t *driver)
Control limiting of output to a single driver.

void stdio_set_translate_crlf (stdio_driver_t *driver, bool translate)
control conversion of line feeds to carriage return on transmissions

int putchar_raw (int c)
putchar variant that skips any CR/LF conversion if enabled

int puts_raw (const char *s)
puts variant that skips any CR/LF conversion if enabled
void stdio_set_chars_available_callback (void(*fn)(void *), void *param)
get notified when there are input characters available

4.5.17.4. Function Documentation

4.5.17.4.1. 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

4.5.17.4.2. putchar_raw

int putchar_raw (int c)
putchar variant that skips any CR/LF conversion if enabled

4.5.17.4.3. puts_raw

int puts_raw (const char * s)
puts variant that skips any CR/LF conversion if enabled

4.5.17.4.4. stdio_filter_driver

void stdio_filter_driver (stdio_driver_t * driver)
Control limiting of output to a single driver.

NOTE this method should always be called on an initialized driver

Parameters

driver if non-null then output only that driver will be used for input/output (assuming it is in the list of enabled drivers), if NULL then all enabled drivers will be used

4.5.17.4.5. stdio_flush

void stdio_flush (void)
Flushes any buffered output.

4.5.17.4.6. stdio_init_all

bool stdio_init_all (void)
Initialize all of the present standard stdio types that are linked into the binary.

Call this method once you have set up your clocks to enable the stdio support for UART, USB and semihosting based on the presence of the respective libraries in the binary.

When stdio_usb is configured, this method can be optionally made to block, waiting for a connection via the variables specified in `stdio_usb_init` (i.e. `PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS`)

Returns

true if at least one output was successfully initialized, false otherwise.

See also

stdio_uart, stdio_usb, stdio_semihosting

### 4.5.17.4.7. `stdio_set_chars_available_callback`

```c
void stdio_set_chars_available_callback (void(*)(void *) fn, void * param)
```

get notified when there are input characters available

**Parameters**

- `fn` Callback function to be called when characters are available. Pass NULL to cancel any existing callback
- `param` Pointer to pass to the callback

### 4.5.17.4.8. `stdio_set_driver_enabled`

```c
void stdio_set_driver_enabled (stdio_driver_t * driver, bool enabled)
```

Adds or removes a driver from the list of active drivers used for input/output.

**NOTE**

this method should always be called on an initialized driver and is not re-entrant

**Parameters**

- `driver` the driver
- `enabled` true to add, false to remove

### 4.5.17.4.9. `stdio_set_translate_crlf`

```c
void stdio_set_translate_crlf (stdio_driver_t * driver, bool translate)
```

control conversion of line feeds to carriage return on transmissions

**NOTE**

this method should always be called on an initialized driver

**Parameters**

- `driver` the driver
- `translate` If true, convert line feeds to carriage return on transmissions
4.5.17.5. pico_stdio_semihosting

Experimental support for stdout using RAM semihosting.

4.5.17.5.1. Detailed Description

Linking this library or calling `pico_enable_stdio_semihosting(TARGET ENABLED)` in the CMake (which achieves the same thing) will add semihosting to the drivers used for standard output.

4.5.17.5.2. Functions

```c
void stdio_semihosting_init (void)
```

Explicitly initialize stdout over semihosting and add it to the current set of stdout targets.

4.5.17.5.3. Function Documentation

`stdio_semihosting_init`

```c
void stdio_semihosting_init (void)
```

Explicitly initialize stdout over semihosting and add it to the current set of stdout targets.

**NOTE**

This method is automatically called by `stdio_init_all()` if `pico_stdio_semihosting` is included in the build.

4.5.17.5.4. pico_stdio_uart

Support for stdin/stdout using UART.

**Detailed Description**

Linking this library or calling `pico_enable_stdio_uart(TARGET ENABLED)` in the CMake (which achieves the same thing) will add UART to the drivers used for standard input/output.

**Functions**

```c
void stdio_uart_init (void)
```

Explicitly initialize stdin/stdout over UART and add it to the current set of stdin/stdout drivers.

```c
void stdin_uart_init (void)
```

Explicitly initialize stdin only (no stdout) over UART and add it to the current set of stdin drivers.

```c
void stdout_uart_init (void)
```

Explicitly initialize stdout only (no stdin) over UART and add it to the current set of stdout drivers.

```c
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.

**Function Documentation**

`stdin_uart_init`

```c
void stdin_uart_init (void)
```

Explicitly initialize stdin only (no stdout) over UART and add it to the current set of stdin drivers.

This method sets up `PICO_DEFAULT_UART_RX_PIN` for UART input (if defined), and configures the baud rate as `PICO_DEFAULT_UART_BAUD_RATE`.
stdio_uart_init

```c
void stdio_uart_init (void)
```

Explicitly initialize stdin/stdout over UART and add it to the current set of stdin/stdout drivers.

This method sets up `PICO_DEFAULT_UART_TX_PIN` for UART output (if defined), `PICO_DEFAULT_UART_RX_PIN` for input (if defined) and configures the baud rate as `PICO_DEFAULT_UART_BAUD_RATE`.

**NOTE**

this method is automatically called by `stdio_init_all()` if `pico_stdio_uart` is included in the build

stdio_uart_init_full

```c
void stdio_uart_init_full (uart_inst_t * uart, uint baud_rate, int tx_pin, int rx_pin)
```

Perform custom initialization initialize stdin/stdout over UART and add it to the current set of stdin/stdout drivers.

**Parameters**

- `uart`: the uart instance to use, `uart0` or `uart1`
- `baud_rate`: the baud rate in Hz
- `tx_pin`: the UART pin to use for stdout (or -1 for no stdout)
- `rx_pin`: the UART pin to use for stdin (or -1 for no stdin)

stdout_uart_init

```c
void stdout_uart_init (void)
```

Explicitly initialize stdout only (no stdin) over UART and add it to the current set of stdout drivers.

This method sets up `PICO_DEFAULT_UART_TX_PIN` for UART output (if defined), and configures the baud rate as `PICO_DEFAULT_UART_BAUD_RATE`.

pico_stdio_usb

Support for stdin/stdout over USB serial (CDC).

**Detailed Description**

Linking this library or calling `pico_enable_stdio_usb(TARGET ENABLED)` in the CMake (which achieves the same thing) will add USB CDC to the drivers used for standard input/output.

Note this library is a developer convenience. It is not applicable in all cases; for one it takes full control of the USB device precluding your use of the USB in device or host mode. For this reason, this library will automatically disengage if you try to using it alongside `tinyusb_device` or `tinyusb_host`. It also takes control of a lower level IRQ and sets up a periodic background task.

This library also includes (by default) functionality to enable the RP2040 to be reset over the USB interface.

**Functions**

- `bool stdio_usb_init (void)`
  - Explicitly initialize USB stdio and add it to the current set of stdin drivers.
- `bool stdio_usb_connected (void)`
  - Check if there is an active stdio CDC connection to a host.

**Function Documentation**

stdio_usb_connected

```c
bool stdio_usb_connected (void)
```

Check if there is an active stdio CDC connection to a host.
Returns
true if stdio is connected over CDC

```c
stdio_usb_init
```

bool stdio_usb_init (void)

Explicitly initialize USB stdio and add it to the current set of stdin drivers.

PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS can be set to cause this method to wait for a CDC connection from
the host before returning, which is useful if you don't want any initial stdout output to be discarded before the
connection is established.

Returns
true if the USB CDC was initialized, false if an error occurred

Copyright (c) 2020 Raspberry Pi (Trading) Ltd.

SPDX-License-Identifier: BSD-3-Clause

4.5.18. pico_standard_link

Standard link step providing the basics for creating a runnable binary.

4.5.18.1. Detailed Description

This includes

- C runtime initialization
- Linker scripts for 'default', 'no_flash', 'blocked_ram' and 'copy_to_ram' binaries
- 'Binary Information' support
- Linker option control

4.6. External API Headers

Headers for interfaces that are shared with code outside of the SDK

<table>
<thead>
<tr>
<th>boot_picoboot</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>boot UF2</td>
<td></td>
</tr>
</tbody>
</table>

4.6.1. boot_picoboot

4.6.1.1. Detailed Description

Header file for the PICOBROWSER USB interface exposed by an RP2040 in BOOTSEL mode.

4.6.2. boot UF2
4.6.2.1. Detailed Description

Header file for the UF2 format supported by an RP2040 in BOOTSEL mode.

4.7. pico_usb_reset_interface

Definition for the reset interface that may be exposed by the pico_stdio_usb library.
Appendix A: App Notes

Attaching a 7 segment LED via GPIO

This example code shows how to interface the Raspberry Pi Pico to a generic 7 segment LED device. It uses the LED to count from 0 to 9 and then repeat. If the button is pressed, then the numbers will count down instead of up.

Wiring information

Our 7 Segment display has pins as follows.

```
--A--
 F  B
--G--
 E  C
--D--
```

By default we are allocating GPIO 2 to segment A, 3 to B etc. So, connect GPIO 2 to pin A on the 7 segment LED display and so on. You will need the appropriate resistors (68 ohm should be fine) for each segment. The LED device used here is common anode, so the anode pin is connected to the 3.3v supply, and the GPIOs need to pull low (to ground) to complete the circuit. The pull direction of the GPIOs is specified in the code itself.

Connect the switch to connect on pressing. One side should be connected to ground, the other to GPIO 9.

List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/gpio/hello_7segment/CMakeLists.txt

```
1  add_executable(hello_7segment
2      hello_7segment.c
3 )
4
5 # pull in common dependencies
6 target_link_libraries(hello_7segment pico_stdlib)
7
```
8 # create map/bin/hex file etc.
9 pico_add_extra_outputs(hello_7segment)
10
11 # add url via pico_set_program_url
12 example_auto_set_url(hello_7segment)

hello_7segment.c
The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/gpio/hello_7segment/hello_7segment.c

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 "pico/stdlib.h"
9 #include "hardware/gpio.h"
10
11 /*
12 Our 7 Segment display has pins as follows:
13
14 --A--
15 F B
16 --G--
17 E C
18 --D--
19
20 By default we are allocating GPIO 2 to segment A, 3 to B etc.
21 So, connect GPIO 2 to pin A on the 7 segment LED display etc. Don't forget
22 the appropriate resistors, best to use one for each segment!
23
24 Connect button so that pressing the switch connects the GPIO 9 (default) to
25 ground (pull down)
26 */
27
28 #define FIRST_GPIO 2
29 #define BUTTON_GPIO (FIRST_GPIO+7)
30
31 // This array converts a number 0-9 to a bit pattern to send to the GPIOs
32 int bits[10] = {
33 0xf3, // 0
34 0x06, // 1
35 0x5b, // 2
36 0x4f, // 3
37 0x66, // 4
38 0x6d, // 5
39 0x7d, // 6
40 0x07, // 7
41 0x7f, // 8
42 0x67 // 9
43 };
44
45 /// \tag::hello_gpio[]
46 int main() {
47 stdio_init_all();
48 printf("Hello, 7segment - press button to count down!\n");
49 // We could use gpio_set_dir_out_masked() here
50 for (int gpio = FIRST_GPIO; gpio < FIRST_GPIO + 7; gpio++) {

Attaching a 7 segment LED via GPIO
gpio_init(gpio);
gpio_set_dir(gpio, GPIO_OUT);

// Our bitmap above has a bit set where we need an LED on, BUT, we are pulling low to light
// so invert our output
gpio_set_outover(gpio, GPIO_OVERRIDE_INVERT);
}

gpio_init(BUTTON_GPIO);
gpio_set_dir(BUTTON_GPIO, GPIO_IN);

// We are using the button to pull down to 0V when pressed, so ensure that when
// unpressed, it uses internal pull ups. Otherwise when unpressed, the input will
// be floating.
gpio_pull_up(BUTTON_GPIO);

int val = 0;
while (true) {
    // Count upwards or downwards depending on button input
    // We are pulling down on switch active, so invert the get to make
    // a press count downwards
    if (!gpio_get(BUTTON_GPIO)) {
        if (val == 9) {
            val = 0;
        } else {
            val++;
        }
    } else if (val == 0) {
        val = 9;
    } else {
        val--;
    }
    // We are starting with GPIO 2, our bitmap starts at bit 0 so shift to start at 2.
    int32_t mask = bits[val] << FIRST_GPIO;
    // Set all our GPIOs in one go!
    // If something else is using GPIO, we might want to use gpio_put_masked()
    gpio_set_mask(mask);
    sleep_ms(250);
    gpio_clr_mask(mask);
}

/// \end::hello_gpio[]

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>7 segment LED module</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>68 ohm resistor</td>
<td>7</td>
<td>generic part</td>
</tr>
<tr>
<td>DIL push to make switch</td>
<td>1</td>
<td>generic switch</td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>10</td>
<td>generic part</td>
</tr>
</tbody>
</table>
DHT-11, DHT-22, and AM2302 Sensors

The DHT sensors are fairly well known hobbyist sensors for measuring relative humidity and temperature using a capacitive humidity sensor, and a thermistor. While they are slow, one reading every ~2 seconds, they are reliable and good for basic data logging. Communication is based on a custom protocol which uses a single wire for data.

NOTE

The DHT-11 and DHT-22 sensors are the most common. They use the same protocol but have different characteristics, the DHT-22 has better accuracy, and has a larger sensor range than the DHT-11. The sensor is available from a number of retailers.

Wiring information

See Figure 10 for wiring instructions.

NOTE

One of the pins (pin 3) on the DHT sensor will not be connected, it is not used.

You will want to place a 10 kΩ resistor between VCC and the data pin, to act as a medium-strength pull up on the data line.

Connecting UART0 of Pico to Raspberry Pi as in Figure 10 and you should see something similar to Figure 11 in minicom when connected to /dev/serial0 on the Raspberry Pi.
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 into the examples build tree.


```c
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.


```c
1 /**
2 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3 */
```
```c
#include <stdio.h>
#include <math.h>
#include "pico/stdlib.h"
#include "hardware/gpio.h"

#if defined PICO_DEFAULT_LED_PIN
#define LED_PIN PICO_DEFAULT_LED_PIN
#endif

const uint DHT_PIN = 15;
const uint MAX_TIMINGS = 85;

typedef struct {
  float humidity;
  float temp_celsius;
} dht_reading;

void read_from_dht(dht_reading *result);

int main() {
  stdio_init_all();
  gpio_init(DHT_PIN);
  #ifdef LED_PIN
  gpio_init(LED_PIN);
  gpio_set_dir(LED_PIN, GPIO_OUT);
  #endif
  while (1) {
    dht_reading reading;
    read_from_dht(&reading);
    float fahrenheit = (reading.temp_celsius * 9 / 5) + 32;
    printf("Humidity = %.1f\%, Temperature = %.1f\C (%.1ff)\n",
           reading.humidity, reading.temp_celsius, fahrenheit);
    sleep_ms(2000);
  }
}

void read_from_dht(dht_reading *result) {
  int data[5] = {0, 0, 0, 0, 0};
  uint last = 1;
  uint j = 0;
  gpio_set_dir(DHT_PIN, GPIO_OUT);
  gpio_put(DHT_PIN, 0);
  sleep_ms(20);
  gpio_set_dir(DHT_PIN, GPIO_IN);
  #ifdef LED_PIN
  gpio_put(LED_PIN, 1);
  #endif
  for (uint i = 0; i < MAX_TIMINGS; i++) {
    uint count = 0;
    while (gpio_get(DHT_PIN) == last) {
      count++;
      sleep_us(1);
      if (count == 255) break;
    }
    last = gpio_get(DHT_PIN);
    if (count == 255) break;
    if ((i >= 4) && (i % 2 == 0)) {
```

Raspberry Pi Pico C/C++ SDK

DHT-11, DHT-22, and AM2302 Sensors

395
```c
68     data[j / 8] <<= 1;
69     if (count > 16) data[j / 8] |= 1;
70     j++;
71 }
72 }
73 #ifdef LED_PIN
74     gpio_put(LED_PIN, 0);
75 #endif
77     result->humidity = ((float)((data[2] & 0x7F) << 8) + data[3]) / 10;
78     result->temp_celsius = data[2];
79 }
80 if (result->humidity > 100) {
81     result->humidity = data[0];
82 }
83 if (result->temp_celsius > 125) {
84     result->temp_celsius = -result->temp_celsius;
85 }
86 } else {
87     printf("Bad data\n");
88 }
89 }
90 }
91 }
```

**Bill of Materials**

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>10 kΩ resistor</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>4</td>
<td>generic part</td>
</tr>
<tr>
<td>DHT-22 sensor</td>
<td>1</td>
<td>generic part</td>
</tr>
</tbody>
</table>

**Attaching a 16x2 LCD via TTL**

This example code shows how to interface the Raspberry Pi Pico to one of the very common 16x2 LCD character displays. Due to the large number of pins these displays use, they are commonly used with extra drivers or backpacks. In this example, we will use an Adafruit LCD display backpack, which supports communication over USB or TTL. A monochrome display with an RGB backlight is also used, but the backpack is compatible with monochrome backlight displays too. There is another example that uses I2C to control a 16x2 display.

The backpack processes a set of commands that are documented here and preceded by the “special” byte 0xFE. The backpack does the ASCII character conversion and even supports custom character creation. In this example, we use the Pico’s primary UART (uart0) to read characters from our computer and send them via the other UART (uart1) to print them onto the LCD. We also define a special startup sequence and vary the display’s backlight color.
NOTE

You can change where stdio output goes (Pico’s USB, uart0 or both) with CMake directives. The CMakeLists.txt file shows how to enable both.

Wiring information

Wiring up the backpack to the Pico requires 3 jumpers, to connect VCC (3.3v), GND, TX. The example here uses both of the Pico’s UARTs, one (uart0) for stdio and the other (uart1) for communication with the backpack. Pin 8 is used as the TX pin. Power is supplied from the 3.3V pin. To connect the backpack to the display, it is common practice to solder it onto the back of the display, or during the prototyping stage to use the same parallel lanes on a breadboard.

NOTE

While this display will work at 3.3V, it will be quite dim. Using a 5V source will make it brighter.

List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

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

```
add_executable(lcd_uart
    lcd_uart.c
)

# pull in common dependencies and additional uart hardware support
target_link_libraries(lcd_uart pico_stdlib hardware_uart)

# enable usb output and uart output
# modify here as required
pico_enable_stdio_usb(lcd_uart 1)
pico_enable_stdio_uart(lcd_uart 1)

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

# add url via pico_set_program_url
```
### The example code.

**Pico Examples:** [https://github.com/raspberrypi/pico-examples/blob/master/uart/lcd_uart/lcd_uart.c](https://github.com/raspberrypi/pico-examples/blob/master/uart/lcd_uart/lcd_uart.c)

```c
/**
 * Copyright (c) 2021 Raspberry Pi (Trading) Ltd.
 * *
 * SPDX-License-Identifier: BSD-3-Clause
 */

/* Example code to drive a 16x2 LCD panel via an Adafruit TTL LCD "backpack"

Optionally, the backpack can be connected the VBUS (pin 40) at 5V if
the Pico in question is powered by USB for greater brightness.

If this is done, then no other connections should be made to the backpack apart
from those listed below as the backpack's logic levels will change.

Connections on Raspberry Pi Pico board, other boards may vary.

GPIO 8 (pin 11)-> RX on backpack
3.3v (pin 36) -> 3.3v on backpack
GND (pin 38) -> GND on backpack
*/

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

// leave uart0 free for stdio
#define UART_ID uart1
#define BAUD_RATE 9600
#define UART_TX_PIN 8
#define LCD_WIDTH 16
#define LCD_HEIGHT 2

// basic commands
#define LCD_DISPLAY_ON 0x42
#define LCD_DISPLAY_OFF 0x46
#define LCD_SET_BRIGHTNESS 0x99
#define LCD_SET_CONTRAST 0x50
#define LCD_AUTOSCROLL_ON 0x51
#define LCD_AUTOSCROLL_OFF 0x52
#define LCD_CLEAR_SCREEN 0x58
#define LCD_SET_SPLASH 0x40

// cursor commands
#define LCD_SET_CURSOR_POS 0x47
#define LCD_CURSOR_HOME 0x48
#define LCD_CURSOR_FORWARD 0x4D
#define LCD_UNDERLINE_CURSOR_ON 0x4A
#define LCD_BLOCK_CURSOR_ON 0x53
#define LCD_BLOCK_CURSOR_OFF 0x54

// rgb commands
```
```c
#define LCD_SET_BACKLIGHT_COLOR 0xD0
#define LCD_SET_DISPLAY_SIZE 0xD1

// change to 0 if display is not RGB capable
#define LCD_IS_RGB 1

void lcd_write(uint8_t cmd, uint8_t* buf, uint8_t buflen) {
    // all commands are prefixed with 0xFE
    const uint8_t pre = 0xFE;
    uart_write_blocking(UART_ID, &pre, 1);
    uart_write_blocking(UART_ID, &cmd, 1);
    uart_write_blocking(UART_ID, buf, buflen);
    sleep_ms(10); // give the display some time
}

void lcd_set_size(uint8_t w, uint8_t h) {
    // sets the dimensions of the display
    uint8_t buf[] = { w, h };
    lcd_write(LCD_SET_DISPLAY_SIZE, buf, 2);
}

void lcd_set_contrast(uint8_t contrast) {
    // sets the display contrast
    lcd_write(LCD_SET_CONTRAST, &contrast, 1);
}

void lcd_set_brightness(uint8_t brightness) {
    // sets the backlight brightness
    lcd_write(LCD_SET_BRIGHTNESS, &brightness, 1);
}

void lcd_set_cursor(bool is_on) {
    // set is_on to true if we want the blinking block and underline cursor to show
    if (is_on) {
        lcd_write(LCD_BLOCK_CURSOR_ON, NULL, 0);
        lcd_write(LCD_UNDERLINE_CURSOR_ON, NULL, 0);
    } else {
        lcd_write(LCD_BLOCK_CURSOR_OFF, NULL, 0);
        lcd_write(LCD_UNDERLINE_CURSOR_OFF, NULL, 0);
    }
}

void lcd_set_backlight(bool is_on) {
    // turn the backlight on (true) or off (false)
    if (is_on) {
        lcd_write(LCD_DISPLAY_ON, (uint8_t*) 0, 1);
    } else {
        lcd_write(LCD_DISPLAY_OFF, NULL, 0);
    }
}

void lcd_clear() {
    // clear the contents of the display
    lcd_write(LCD_CLEAR_SCREEN, NULL, 0);
}

void lcd_cursor_reset() {
    // reset the cursor to (1, 1)
    lcd_write(LCD_CURSOR_HOME, NULL, 0);
}

#if LCD_IS_RGB
void lcd_set_backlight_color(uint8_t r, uint8_t g, uint8_t b) {
    // only supported on RGB displays!
#endif
```

---

*Attaching a 16x2 LCD via TTL*
```c
120  uint8_t buf[] = { r, g, b; }
121  lcd_write(LCD_SET_BACKLIGHT_COLOR, buf, 3);
122 }
123 #endif
124
125 void lcd_init() {
126  lcd_set_backlight(true);
127  lcd_set_size(LCD_WIDTH, LCD_HEIGHT);
128  lcd_set_contrast(155);
129  lcd_set_brightness(255);
130  lcd_set_cursor(false);
131 }
132
133 int main() {
134  stdio_init_all();
135  uart_init(UART_ID, BAUD_RATE);
136  uart_set_translate_crlf(UART_ID, false);
137  gpio_set_function(UART_TX_PIN, GPIO_FUNC_UART);
138  bi_decl(bi_1pin_with_func(UART_TX_PIN, GPIO_FUNC_UART));
139  lcd_init();
140
141  // define startup sequence and save to EEPROM
142  // no more or less than 32 chars, if not enough, fill remaining ones with spaces
143  uint8_t splash_buf[] = "Hello LCD, from Pi Towers! ";
144  lcd_write(LCD_SET_SPLASH, splash_buf, LCD_WIDTH * LCD_HEIGHT);
145  lcd_cursor_reset();
146  lcd_clear();
147
148 #if LCD_IS_RGB
149  uint8_t i = 0; // it's ok if this overflows and wraps, we're using sin
150  const float frequency = 0.1f;
151  uint8_t red, green, blue;
152 #endif
153
154  while (1) {
155     // send any chars from stdio straight to the backpack
156     char c = getchar();
157     // any bytes not followed by 0xFE (the special command) are interpreted
158     // as text to be displayed on the backpack, so we just send the char
159     // down the UART byte pipe!
160     if (c < 128) uart_putchar(UART_ID, c); // skip extra non-ASCII chars
161     red = (uint8_t)(sin(frequency * i + 0) * 127 + 128);
162     green = (uint8_t)(sin(frequency * i + 2) * 127 + 128);
163     blue = (uint8_t)(sin(frequency * i + 4) * 127 + 128);
164     i++;
165 #if LCD_IS_RGB
166     lcd_set_backlight_color(red, green, blue);
167 #endif
168  }
169 }
170
171 #if LCD_IS_RGB
172
```

### Bill of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
</tbody>
</table>

Table 18. A list of materials required for the example
Attaching a microphone using the ADC

This example code shows how to interface the Raspberry Pi Pico with a standard analog microphone via the onboard analog to digital converter (ADC). In this example, we use an ICS-40180 breakout board by SparkFun but any analog microphone should be compatible with this tutorial. SparkFun have written a guide for this board that goes into more detail about the board and how it works.

**TIP**

An analog to digital converter (ADC) is responsible for reading continually varying input signals that may range from 0 to a specified reference voltage (in the Pico’s case this reference voltage is set by the supply voltage and can be measured on pin 35, ADC_VREF) and converting them into binary, i.e. a number that can be digitally stored.

The Pico has a 12-bit ADC (ENOB of 8.7-bit, see RP2040 datasheet section 4.9.3 for more details), meaning that a read operation will return a number ranging from 0 to 4095 ($2^{12} - 1$) for a total of 4096 possible values. Therefore, the resolution of the ADC is $3.3/4096$, so roughly steps of 0.8 millivolts. The SparkFun breakout uses an OPA344 operational amplifier to boost the signal coming from the microphone to voltage levels that can be easily read by the ADC. An important side effect is that a bias of $0.5\times \text{Vcc}$ is added to the signal, even when the microphone is not picking up any sound.

The ADC provides us with a raw voltage value but when dealing with sound, we’re more interested in the amplitude of the audio signal. This is defined as one half the peak-to-peak amplitude. Included with this example is a very simple Python script that will plot the voltage values it receives via the serial port. By tweaking the sampling rates, and various other parameters, the data from the microphone can be analysed in various ways, such as in a Fast Fourier Transform to see what frequencies make up the signal.

**Wiring information**

Wiring up the device requires 3 jumpers, to connect VCC (3.3v), GND, and AOUT. The example here uses ADC0, which is GP26. Power is supplied from the 3.3V pin.
WARNING

Most boards will take a range of VCC voltages from the Pico’s default 3.3V to the 5 volts commonly seen on other microcontrollers. Ensure your board doesn’t output an analogue signal greater than 3.3V as this may result in permanent damage to the Pico’s ADC.

List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

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

```c
1 add_executable(microphone_adc
2  microphone_adc.c
3 )
4
5 # pull in common dependencies and adc hardware support
6 target_link_libraries(microphone_adc pico_stdlib hardware_adc)
7
8 # create map/bin/hex file etc.
9 pico_add_extra_outputs(microphone_adc)
10
11 # add url via pico_set_program_url
12 example_auto_set_url(microphone_adc)
```

microphone_adc.c

The example code.

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

```c
1 /**
2 * Copyright (c) 2021 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX-License-Identifier: BSD-3-Clause
5 */
6```
```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/gpio.h"
#include "hardware/adc.h"
#include "hardware/uart.h"
#include "pico/binary_info.h"

/* Example code to extract analog values from a microphone using the ADC
with accompanying Python file to plot these values
Connections on Raspberry Pi Pico board, other boards may vary.
GPIO 26/ADC0 (pin 31)-> AOUT or AUD on microphone board
3.3v (pin 36) -> VCC on microphone board
GND (pin 38) -> GND on microphone board */

#define ADC_NUM 0
#define ADC_PIN (26 + ADC_NUM)
#define ADC_VREF 3.3
#define ADC_RANGE (1 << 12)
#define ADC_CONVERT (ADC_VREF / (ADC_RANGE - 1))

int main() {
    stdio_init_all();
    printf("Beep boop, listening...\n");
    bi_decl(bi_program_description("Analog microphone example for Raspberry Pi Pico")); // for picotool
    bi_decl(bi_1pin_with_name(ADC_PIN, "ADC input pin"));
    adc_init();
    adc_gpio_init(ADC_PIN);
    adc_select_input(ADC_NUM);
    uint adc_raw;
    while (1) {
        adc_raw = adc_read(); // raw voltage from ADC
        printf("%.2f\n", adc_raw * ADC_CONVERT);
        sleep_ms(10);
    }
}
```

## Bill of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>ICS-40180 microphone breakout board or similar</td>
<td>1</td>
<td>From SparkFun</td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>3</td>
<td>generic part</td>
</tr>
</tbody>
</table>
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/ICS 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.

List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Picoprojects: https://github.com/raspberrypi/picoprojects/blob/master/spi/bme280_spi/CMakeLists.txt

1 add_executable(bme280_spi)
2 bme280_spi.c
3 )
4
5 # pull in common dependencies and additional spi hardware support
6 target_link_libraries(bme280_spi pico_stdlib hardware_spi)
# create map/bin/hex file etc.
pico_add_extra_outputs(bme280_spi)

# add url via pico_set_program_url
eexample_auto_set_url(bme280_spi)

bme280_spi.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/spi/bme280_spi/bme280_spi.c

```c
#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
#include "pico/binary_info.h"
#include "hardware/spi.h"

#define READ_BIT 0x80

int32_t t_fine;
uint16_t dig_T1;
int16_t dig_T2, dig_T3;
uint16_t dig_P1;
int16_t dig_P2, dig_P3, dig_P4, dig_P5, dig_P6, dig_P7, dig_P8, dig_P9;
uint8_t dig_H1, dig_H3;
int8_t dig_H6;

/* Example code to talk to a bme280 humidity/temperature/pressure sensor.

NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico
GPIO (and therefore SPI) cannot be used at 5v.

You will need to use a level shifter on the SPI lines if you want to run the
board at 5v.

Connections on Raspberry Pi Pico board and a generic bme280 board, other
boards may vary.

GPIO 16 (pin 21) MISO/spi0_rx-> SDO/SDO on bme280 board
GPIO 17 (pin 22) Chip select -> CSB/!CS on bme280 board
GPIO 18 (pin 24) SCK/spi0_sclk -> SCL/SCK on bme280 board
GPIO 19 (pin 25) MOSI/spi0_tx -> SDA/SDI on bme280 board
3.3v (pin 36) -> VCC on bme280 board
GND (pin 38) -> GND on bme280 board

Note: SPI devices can have a number of different naming schemes for pins. See
for variations.

This code uses a bunch of register definitions, and some compensation code derived
from the Bosch datasheet which can be found here.
https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-
ds02.pdf
*/

#define READ_BIT 0x80

int32_t t_fine;
uint16_t dig_T1;
int16_t dig_T2, dig_T3;
uint16_t dig_P1;
int16_t dig_P2, dig_P3, dig_P4, dig_P5, dig_P6, dig_P7, dig_P8, dig_P9;
uint8_t dig_H1, dig_H3;
int8_t dig_H6;
```

Attaching a BME280 temperature/humidity/pressure sensor via SPI
/* The following compensation functions are required to convert from the raw ADC
data from the chip to something usable. Each chip has a different set of
compensation parameters stored on the chip at point of manufacture, which are
read from the chip at startup and used in these routines.
*/

uint32_t compensate_temp(int32_t adc_T) {
    int32_t var1, var2, T;
    var1 = (((adc_T >> 3) - (int32_t) dig_T1 << 1)) * ((int32_t) dig_T2) >> 11;
    var2 = (((adc_T >> 4) - (int32_t) dig_T1) * ((adc_T >> 4) - ((int32_t) dig_T1))) >> 12) * ((int32_t) dig_T3);
    t_fine = var1 + var2;
    T = (t_fine * 5 + 128) >> 8;
    return T;
}

uint32_t compensate_pressure(int32_t adc_P) {
    int32_t var1, var2;
    uint32_t p;
    var1 = ((int32_t) t_fine >> 1) - (int32_t) 64000;
    var2 = ((var1 >> 2) * (var1 >> 2) >> 11) * ((int32_t) dig_P6);
    var2 = var2 + ((var1 * ((int32_t) dig_P5)) << 1);
    var2 = (var2 >> 2) + (int32_t) dig_P4 << 16;
    var1 = (((int32_t) dig_P3 * ((var1 >> 2) * (var1 >> 2) >> 13)) >> 3) + (((int32_t) dig_P2) * var1) >> 11) >> 18;
    var1 = (((32768 + var1)) * ((int32_t) dig_P1)) >> 15;
    if (var1 == 0)
        return 0;
    p = (((uint32_t) ((int32_t) 1048576 - adc_P) - (var2 >> 12))) * 3125;
    if (p < 0x80000000)
        p = (p << 1) / (uint32_t) var1;
    else
        p = (p / (uint32_t) var1) * 2;
    var1 = ((int32_t) dig_P9) * ((int32_t) (((p >> 3) + (p >> 3)) >> 13)) >> 12;
    var2 = ((int32_t) (p >> 2)) * ((int32_t) dig_P8) >> 13;
    p = (uint32_t) (((int32_t) p + (var1 + var2 + dig_P7) >> 4));
    return p;
}

uint32_t compensate_humidity(int32_t adc_H) {
    int32_t v_x1_u32r;
    v_x1_u32r = (t_fine - (int32_t) 76800);
    v_x1_u32r = (((adc_H << 14) - (((int32_t) dig_H4) << 20) - (((int32_t) dig_H5) * v_x1_u32r)) +
                  (((int32_t) 16384)) >> 15) * (((((v_x1_u32r * ((int32_t) dig_H6)) >> 10) * (((v_x1_u32r * 
                      ((int32_t) dig_H3)) >> 11) + ((int32_t) 32768))) >> 10) + ((int32_t) 2097152)) *
                      ((int32_t) dig_H2) + 8192) >> 14));
    v_x1_u32r = (v_x1_u32r - (((v_x1_u32r >> 15) * (v_x1_u32r >> 15)) >> 7) * ((int32_t) dig_H1)) >> 4);
    v_x1_u32r = (v_x1_u32r < 0 ? 0 : v_x1_u32r);
    v_x1_u32r = (v_x1_u32r > 419430400 ? 419430400 : v_x1_u32r);
    return (uint32_t) (v_x1_u32r >> 12);
#ifndef PICO_DEFAULT_SPI_CSN_PIN
static inline void cs_select() {
    asm volatile("nop\n nop\n nop\n");
    gpio_put(PICO_DEFAULT_SPI_CSN_PIN, 0); // Active low
    asm volatile("nop\n nop\n nop\n");
}

static inline void cs_deselect() {
    asm volatile("nop\n nop\n nop\n");
    gpio_put(PICO_DEFAULT_SPI_CSN_PIN, 1);
    asm volatile("nop\n nop\n nop\n");
}
#endif

#define PICO_DEFAULT_SPI_CSN_PIN

static void write_register(uint8_t reg, uint8_t data) {
    uint8_t buf[2];
    buf[0] = reg & 0x7f; // remove read bit as this is a write
    buf[1] = data;
    cs_select();
    spi_write_blocking(spi_default, buf, 2);
    cs_deselect();
    sleep_ms(10);
}

static void read_registers(uint8_t reg, uint8_t *buf, uint16_t len) {
    // For this particular device, we send the device the register we want to read
    // first, then subsequently read from the device. The register is auto incrementing
    // so we don’t need to keep sending the register we want, just the first.
    reg |= READ_BIT;
    cs_select();
    spi_write_blocking(spi_default, &reg, 1);
    sleep_ms(10);
    spi_read_blocking(spi_default, 0, buf, len);
    cs_deselect();
    sleep_ms(10);
}

/* This function reads the manufacturing assigned compensation parameters from the device */
void read_compensation_parameters() {
    uint8_t buffer[26];
    read_registers(0x88, buffer, 24);
    dig_T1 = buffer[0] | (buffer[1] << 8);
    dig_T2 = buffer[2] | (buffer[3] << 8);
    dig_T3 = buffer[4] | (buffer[5] << 8);
    dig_P1 = buffer[6] | (buffer[7] << 8);
    dig_P2 = buffer[8] | (buffer[9] << 8);
    dig_P4 = buffer[12] | (buffer[13] << 8);
    dig_P5 = buffer[14] | (buffer[15] << 8);
    dig_P6 = buffer[16] | (buffer[17] << 8);
    dig_P7 = buffer[18] | (buffer[19] << 8);
    dig_P8 = buffer[20] | (buffer[21] << 8);
    dig_P9 = buffer[22] | (buffer[23] << 8);
    dig_H1 = buffer[25];
    read_registers(0xE1, buffer, 8);
    dig_H2 = buffer[0] | (buffer[1] << 8);
    dig_H3 = (int8_t)buffer[2];
```c
174    dig_H6 = (int8_t) buffer[7];
175 }
176
177 static void bme280_read_raw(int32_t *humidity, int32_t *pressure, int32_t *temperature) {
178    uint8_t buffer[8];
179    read_registers(0xF7, buffer, 8);
181        4);
183        >> 4);
184    *humidity = (uint32_t) buffer[6] << 8 | buffer[7];
185 }
186
187 int main() {
188    stdio_init_all();
189    #if defined(spi_default) || !defined(PICO_DEFAULT_SPI_SCK_PIN) ||
190        !defined(PICO_DEFAULT_SPI_CSN_PIN) || !defined(PICO_DEFAULT_SPI_TX_PIN) ||
191        !defined(PICO_DEFAULT_SPI_RX_PIN) ||
192        !defined(PICO_DEFAULT_SPI_CSN_PIN)
193    #warning spi/bme280_spi example requires a board with SPI pins
194    puts("Default SPI pins were not defined");
195    #else
196
197    // This example will use SPI0 at 0.5MHz.
198    spi_init(spi_default, 500 * 1000);
199    gpio_set_function(PICO_DEFAULT_SPI_RX_PIN, GPIO_FUNC_SPI);
200    gpio_set_function(PICO_DEFAULT_SPI_TX_PIN, GPIO_FUNC_SPI);
201    // Make the SPI pins available to picotool
202    bi_decl(bi_3pins_with_func(PICO_DEFAULT_SPI_RX_PIN, PICO_DEFAULT_SPI_TX_PIN,
203        PICO_DEFAULT_SPI_SCK_PIN, GPIO_FUNC_SPI));
204
205    // Chip select is active-low, so we’ll initialise it to a driven-high state
206    gpio_init(PICO_DEFAULT_SPI_SCK_PIN);
207    gpio_set_dir(PICO_DEFAULT_SPI_SCK_PIN, GPIO_OUT);
208    // Make the CS pin available to picotool
209    bi_decl(bi_1pin_with_name(PICO_DEFAULT_SPI_SCK_PIN, "SPI CS"));
210
211    // See if SPI is working - interrogate the device for its I2C ID number, should be 0x60
212    uint8_t id;
213    read_registers(0x0D, &id, 1);
214    printf("Chip ID is 0x\%x\n", id);
215
216    read Compensation parameters();
217
218    write_register(0xF2, 0x1); // Humidity oversampling register - going for x1
219    write_register(0xF4, 0x27); // Set rest of oversampling modes and run mode to normal
220
221    int32_t humidity, pressure, temperature;
222
223    while (1) {
224        bme280_read_raw(&humidity, &pressure, &temperature);
225    }
226
227    // These are the raw numbers from the chip, so we need to run through the
228    // compensations to get human understandable numbers
229    pressure = compensate_pressure(pressure);
230    temperature = compensate_temp(temperature);
231    humidity = compensate_humidity(humidity);
```
printf("Humidity = %.2f%%\n", humidity / 1024.0);
printf("Pressure = %dPa\n", pressure);
printf("Temp. = %.2fC\n", temperature / 100.0);
sleep_ms(1000);
#else
#endif

Bill of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>BME280 board</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>6</td>
<td>generic part</td>
</tr>
</tbody>
</table>

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.

List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/spi/mpu9250_spi/CMakeLists.txt

```c
1 add_executable(mpu9250_spi
2   mpu9250_spi.c
3 )
4
5 # pull in common dependencies and additional spi hardware support
6 target_link_libraries(mpu9250_spi pico_stdlib hardware_spi)
7 # create map/bin/hex file etc.
8 pico_add_extra_outputs(mpu9250_spi)
9
10 # add url via pico_set_program_url
11 example_auto_set_url(mpu9250_spi)
```

mpu9250_spi.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/spi/mpu9250_spi/mpu9250_spi.c

```c
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 #include "pico/binary_info.h"
12
13 /* Example code to talk to a MPU9250 MEMS accelerometer and gyroscope.
14 Ignores the magnetometer, that is left as a exercise for the reader.
15 */
```
This is taking too simple approach of simply reading registers. It’s perfectly possible to link up an interrupt line and set things up to read from the inbuilt FIFO to make it more useful.

NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico GPIO (and therefore SPI) cannot be used at 5v.

You will need to use a level shifter on the I2C lines if you want to run the board at 5v.

Connections on Raspberry Pi Pico board and a generic MPU9250 board, other boards may vary.

GPIO 4 (pin 6) MISO/spi0_rx -> ADO on MPU9250 board
GPIO 5 (pin 7) Chip select -> NCS on MPU9250 board
GPIO 6 (pin 9) SCK/spi0_sclk -> SCL on MPU9250 board
GPIO 7 (pin 10) MOSI/spi0_tx -> SDA on MPU9250 board
3.3v (pin 36) -> VCC on MPU9250 board
GND (pin 38)  -> GND on MPU9250 board

Note: SPI devices can have a number of different naming schemes for pins. See the Wikipedia page at https://en.wikipedia.org/wiki/Serial_Peripheral_Interface for variations.
The particular device used here uses the same pins for I2C and SPI, hence the using of I2C names
/*
#define PIN_MISO 4
#define PIN_CS   5
#define PIN_SCK  6
#define PIN_MOSI 7
#define SPI_PORT spi0
#define READ_BIT 0x80

static inline void cs_select() {
  asm volatile("nop \n nop \n nop");
  gpio_put(PIN_CS, 0); // Active low
  asm volatile("nop \n nop \n nop");
}

static inline void cs_deselect() {
  asm volatile("nop \n nop \n nop");
  gpio_put(PIN_CS, 1);
  asm volatile("nop \n nop \n nop");
}

static void mpu9250_reset() {
  // Two byte reset. First byte register, second byte data
  // There are a lot more options to set up the device in different ways that could be added here
  uint8_t buf[] = {0x6B, 0x00};
  cs_select();
  spi_write_blocking(SPI_PORT, buf, 2);
  cs_deselect();
}

static void read_registers(uint8_t reg, uint8_t *buf, uint16_t len) {
  // For this particular device, we send the device the register we want to read
  // first, then subsequently read from the device. The register is auto incrementing
  // so we don’t need to keep sending the register we want, just the first.
  reg |= READ_BIT;
79  cs_select();
80  spi_write_blocking(SPI_PORT, &reg, 1);
81  sleep_ms(10);
82  spi_read_blocking(SPI_PORT, 0, buf, len);
83  cs_deselect();
84  sleep_ms(10);
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  for (int i = 0; i < 3; i++) {
93    accel[i] = (buffer[i * 2] << 8 | buffer[(i * 2) + 1]);
94  }
95
96  // Now gyro data from reg 0x43 for 6 bytes
97  read_registers(0x43, buffer, 6);
98  for (int i = 0; i < 3; i++) {
99    gyro[i] = (buffer[i * 2] << 8 | buffer[(i * 2) + 1]);
100   }
101
102  // Now temperature from reg 0x41 for 2 bytes
103  read_registers(0x41, buffer, 2);
104  *temp = buffer[0] << 8 | buffer[1];
105 }
106
107 int main() {
108  stdio_init_all();
109
110  printf("Hello, MPU9250! Reading raw data from registers via SPI...\n");
111
112  // This example will use SPI0 at 0.5MHz.
113  spi_init(SPI_PORT, 500 * 1000);
114  gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);
115  gpio_set_function(PIN_SCK, GPIO_FUNC_SPI);
116  gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);
117  // Make the SPI pins available to picotool
118  bi_decl(bi_3pins_with_func(PIN_MISO, PIN_MOSI, PIN_SCK, GPIO_FUNC_SPI));
119
120  // Chip select is active-low, so we'll initialise it to a driven-high state
121  gpio_init(PIN_CS);
122  gpio_set_dir(PIN_CS, GPIO_OUT);
123  gpio_put(PIN_CS, 1);
124  // Make the CS pin available to picotool
125  bi_decl(bi_1pin_with_name(PIN_CS, "SPI CS"));
126
127  // See if SPI is working - interrogate the device for its I2C ID number, should be 0x71
128  uint8_t id;
129  read_registers(0x75, &id, 1);
130  printf("I2C address is 0x%x\n", id);
131
132  // start reading the raw data
133  while (1) {
134    mpu9250_read_raw(acceleration, gyro, &temp);
135  }
136}
These are the raw numbers from the chip, so will need tweaking to be really useful. See the datasheet for more information.

```c
printf("Acc. X = %d, Y = %d, Z = %d\n", acceleration[0], acceleration[1], acceleration[2]);
printf("Gyro. X = %d, Y = %d, Z = %d\n", gyro[0], gyro[1], gyro[2]);
// Temperature is simple so use the datasheet calculation to get deg C.
printf("Temp. = %f\n", (temp / 340.0) + 36.53);
```

Bill of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>MPU9250 board</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>6</td>
<td>generic part</td>
</tr>
</tbody>
</table>

Attaching a MPU6050 accelerometer/gyroscope via I2C

This example code shows how to interface the Raspberry Pi Pico to the MPU6050 accelerometer/gyroscope board. This device uses I2C for communications, and most MPU6050 parts are happy running at either 3.3 or 5v. The Raspberry Pi RP2040 GPIO's work at 3.3v so that is what the example uses.

**NOTE**

This is a very basic example, and only recovers raw data from the sensor. There are various calibration options available that should be used to ensure that the final results are accurate. It is also possible to wire up the interrupt pin to a GPIO and read data only when it is ready, rather than using the polling approach in the example.

Wiring information

Wiring up the device requires 4 jumpers, to connect VCC (3.3v), GND, SDA and SCL. The example here uses I2C port 0, which is assigned to GPIO 4 (SDA) and 5 (SCL) in software. Power is supplied from the 3.3V pin.

**NOTE**

There are many different manufacturers who sell boards with the MPU6050. Whilst they all appear slightly different, they all have, at least, the same 4 pins required to power and communicate. When wiring up a board that is different to the one in the diagram, ensure you connect up as described in the previous paragraph.
Figure 17. Wiring Diagram for MPU6050.

List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/mpu6050_i2c/CMakeLists.txt

```c
1 add_executable(mpu6050_i2c
2   mpu6050_i2c.c
3 )
4
5 # pull in common dependencies and additional i2c hardware support
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/blob/master/i2c/mpu6050_i2c/mpu6050_i2c.c

```c
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
12
13 /* Example code to talk to a MPU6050 MEMS accelerometer and gyroscope
14 This is taking to simple approach of simply reading registers. It's perfectly
15 possible to link up an interrupt line and set things up to read from the
16 inbuilt FIFO to make it more useful.
17 NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico
18 GPIO (and therefor I2C) cannot be used at 5v.
19 You will need to use a level shifter on the I2C lines if you want to run the
```
board at 5v.

Connections on Raspberry Pi Pico board, other boards may vary.

GPIO PICO_DEFAULT_I2C_SDA_PIN (On Pico this is GP4 (pin 6)) -> SDA on MPU6050 board
GPIO PICO_DEFAULT_I2C_SCL_PIN (On Pico this is GP5 (pin 7)) -> SCL on MPU6050 board
3.3v (pin 36) -> VCC on MPU6050 board
GND (pin 38) -> GND on MPU6050 board

// By default these devices are on bus address 0x68
static int addr = 0x68;

#ifdef i2c_default
void mpu6050_reset()
{
  // Two byte reset. First byte register, second byte data
  // There are a load more options to set up the device in different ways that could be added here
  uint8_t buf[] = {0x6B, 0x80};
  i2c_write_blocking(i2c_default, addr, buf, 2, false);
}

void mpu6050_read_raw(int16_t accel[3], int16_t gyro[3], int16_t *temp)
{
  // For this particular device, we send the device the register we want to read
  // first, then subsequently read from the device. The register is auto incrementing
  // so we don't need to keep sending the register we want, just the first.
  uint8_t buffer[6];

  // Start reading acceleration registers from register 0x3B for 6 bytes
  uint8_t val = 0x3B;
  i2c_write_blocking(i2c_default, addr, &val, 1, true); // true to keep master control of bus
  i2c_read_blocking(i2c_default, addr, buffer, 6, false);

  for (int i = 0; i < 3; i++)
  {
    accel[i] = (buffer[i * 2] << 8 | buffer[(i * 2) + 1]);
  }

  // Now gyro data from reg 0x43 for 6 bytes
  // The register is auto incrementing on each read
  val = 0x43;
  i2c_write_blocking(i2c_default, addr, &val, 1, true);
  i2c_read_blocking(i2c_default, addr, buffer, 6, false); // False - finished with bus

  for (int i = 0; i < 3; i++)
  {
    gyro[i] = (buffer[i * 2] << 8 | buffer[(i * 2) + 1]);
  }

  // Now temperature from reg 0x41 for 2 bytes
  // The register is auto incrementing on each read
  val = 0x41;
  i2c_write_blocking(i2c_default, addr, &val, 1, true);
  i2c_read_blocking(i2c_default, addr, buffer, 2, false); // False - finished with bus
  *temp = buffer[0] << 8 | buffer[1];
}
#endif

int main()
{
  stdio_init_all();
  if (!defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN))
    #warning i2c/mpu6050_i2c example requires a board with I2C pins
puts("Default I2C pins were not defined");
return 0;
#else
printf("Hello, MPU6050! Reading raw data from registers...\n");
// This example will use I2C0 on the default SDA and SCL pins (4, 5 on a Pico)
i2c_init(i2c_default, 400 * 1000);
gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
// Make the I2C pins available to picotool
bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));
mpu6050_reset();

int16_t acceleration[3], gyro[3], temp;
while (1) {
    mpu6050_read_raw(acceleration, gyro, &temp);
    // These are the raw numbers from the chip, so will need tweaking to be really useful.
    // See the datasheet for more information
    printf("Acc. X = %d, Y = %d, Z = %d\n", acceleration[0], acceleration[1], acceleration[2]);
    printf("Gyro. X = %d, Y = %d, Z = %d\n", gyro[0], gyro[1], gyro[2]);
    // Temperature is simple so use the datasheet calculation to get deg C.
    // Note this is chip temperature.
    printf("Temp. = %f\n", (temp / 340.0) + 36.53);
    sleep_ms(100);
} #endif

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>MPU6050 board</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>4</td>
<td>generic part</td>
</tr>
</tbody>
</table>

**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.

List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/lcd_1602_i2c/CMakeLists.txt

```c
1  add_executable(lcd_1602_i2c
2    lcd_1602_i2c.c
3  )
4
5  # pull in common dependencies and additional i2c hardware support
6  target_link_libraries(lcd_1602_i2c pico_stdlib hardware_i2c)
7  # create map/bin/hex file etc.
8  pico_add_extra_outputs(lcd_1602_i2c)
9  # add url via pico_set_program_url
10 example_auto_set_url(lcd_1602_i2c)
```

lcd_1602_i2c.c

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

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

/* Example code to drive a 16x2 LCD panel via a I2C bridge chip (e.g. PCF8574)

NOTE: The panel must be capable of being driven at 3.3v NOT 5v. The Pico
GPIO (and therefor I2C) cannot be used at 5v.
You will need to use a level shifter on the I2C lines if you want to run the
board at 5v.

Connections on Raspberry Pi Pico board, other boards may vary.
GPIO 4 (pin 6)-> SDA on LCD bridge board
GPIO 5 (pin 7)-> SCL on LCD bridge board
3.3v (pin 36) -> VCC on LCD bridge board
GND (pin 38)  -> GND on LCD bridge board
*/

// commands
const int LCD_CLEARDISPLAY = 0x01;
const int LCD_RETURNHOME = 0x02;
const int LCD_ENTRYMODESET = 0x04;
const int LCD_DISPLAYCONTROL = 0x08;
const int LCD_CURSORSHIFT = 0x10;
const int LCD_FUNCTIONSET = 0x20;
const int LCD_SETCGRAMADDR = 0x40;
const int LCD_SETDDRAMADDR = 0x80;

// flags for display entry mode
const int LCD_ENTRYSHIFTINCREMENT = 0x01;
const int LCD_ENTRYLEFT = 0x02;

// flags for display and cursor control
const int LCD_BLINKON = 0x01;
const int LCD_CURSORON = 0x02;
const int LCD_DISPLAYON = 0x04;

// flags for display and cursor shift
const int LCD_MOVERIGHT = 0x04;
const int LCD_DISPLAYMOVE = 0x08;

// flags for function set
const int LCD_5x10DOTS = 0x04;
const int LCD_2LINE = 0x08;
const int LCD_8BITMODE = 0x10;

// flag for backlight control
const int LCD_BACKLIGHT = 0x08;

const int LCD_ENABLE_BIT = 0x04;

// By default these LCD display drivers are on bus address 0x27
static int addr = 0x27;

// Modes for lcd_send_byte
#define LCD_CHARACTER 1
#define LCD_COMMAND 0
#define MAX_LINES 2
#define MAX_CHARS 16

/* Quick helper function for single byte transfers */
void i2c_write_byte(uint8_t val)
{
  #ifdef i2c_default
    i2c_write_blocking(i2c_default, addr, &val, 1, false);
  #endif
}

void lcd_toggle_enable(uint8_t val)
{
  // Toggle enable pin on LCD display
  // We cannot do this too quickly or things don’t work
  #define DELAY_US 600
  sleep_us(DELAY_US);
  i2c_write_byte(val | LCD_ENABLE_BIT);
  sleep_us(DELAY_US);
  i2c_write_byte(val & ~LCD_ENABLE_BIT);
  sleep_us(DELAY_US);
}

// The display is sent a byte as two separate nibble transfers
void lcd_send_byte(uint8_t val, int mode)
{
  uint8_t high = mode | (val & 0xF0) | LCD_BACKLIGHT;
  uint8_t low = mode | ((val << 4) & 0xF0) | LCD_BACKLIGHT;
  i2c_write_byte(high);
  lcd_toggle_enable(high);
  i2c_write_byte(low);
  lcd_toggle_enable(low);
}

void lcd_clear(void)
{
  lcd_send_byte(LCD_CLEARDISPLAY, LCD_COMMAND);
}

void lcd_set_cursor(int line, int position)
{
  int val = (line == 0) ? 0x80 + position : 0xB0 + position;
  lcd_send_byte(val, LCD_COMMAND);
}

void lcd_char(char val)
{
  lcd_send_byte(val, LCD_CHARACTER);
}

void lcd_string(const char *s)
{
  while (*s)
  {
    lcd_char(*s++);
  }
}

void lcd_init()
{
  lcd_send_byte(0x03, LCD_COMMAND);
  lcd_send_byte(0x03, LCD_COMMAND);
  lcd_send_byte(0x03, LCD_COMMAND);
  lcd_send_byte(0x02, LCD_COMMAND);
}
```c
126  lcd_send_byte(LCD_ENTRYMODESET | LCD_ENTRYLEFT, LCD_COMMAND);
127  lcd_send_byte(LCD_FUNCTIONSET | LCD_2LINE, LCD_COMMAND);
128  lcd_send_byte(LCD_DISPLAYCONTROL | LCD_DISPLAYON, LCD_COMMAND);
129  lcd_clear();
130 }
131
132 int main() {
133 #if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) ||
134 #defined(PICO_DEFAULT_I2C_SCL_PIN)
135 #warning i2c/lcd_1602_i2c example requires a board with I2C pins
136 #else
137 // This example will use I2C on the default SDA and SCL pins (4, 5 on a Pico)
138 i2c_init(i2c_default, 100 * 1000);
139 gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
140 gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
141 gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
142 gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
143 // Make the I2C pins available to picotool
144 bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));
145
146 lcd_init();
147
148 static char *message[] =
149 {
150 "RP2040 by", "Raspberry Pi",
151 "A brand new", "microcontroller",
152 "Twin core M0", "Full C SDK",
153 "More power in", "your product",
154 "More beans", "than Heinz!"
155 }
156 while (1) {
157 for (int m = 0; m < sizeof(message) / sizeof(message[0])); m += MAX_LINES) {
158 for (int line = 0; line < MAX_LINES; line++) {
159 lcd_set_cursor(line, (MAX_CHARS / 2) - strlen(message[m + line]) / 2);
160 lcd_string(message[m + line]);
161 }
162 sleep_ms(2000);
163 lcd_clear();
164 }
165 }
166 #endif
167 }
```

### Bill of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>1602A based LCD panel 3.3v</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>1602A to I2C bridge device 3.3v</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>4</td>
<td>generic part</td>
</tr>
</tbody>
</table>
Attaching a BMP280 temp/pressure sensor via I2C

This example code shows how to interface the Raspberry Pi Pico with the popular BMP280 temperature and air pressure sensor manufactured by Bosch. A similar variant, the BME280, exists that can also measure humidity. There is another example that uses the BME280 device but talks to it via SPI as opposed to I2C.

The code reads data from the sensor’s registers every 500 milliseconds and prints it via the onboard UART. This example operates the BMP280 in normal mode, meaning that the device continuously cycles between a measurement period and a standby period at a regular interval we can set. This has the advantage that subsequent reads do not require configuration register writes and is the recommended mode of operation to filter out short-term disturbances.

**TIP**

The BMP280 is highly configurable with 3 modes of operation, various oversampling levels, and 5 filter settings. Find the datasheet online ([https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmp280-ds001.pdf](https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmp280-ds001.pdf)) to explore all of its capabilities beyond the simple example given here.

Wiring information

Wiring up the device requires 4 jumpers, to connect VCC (3.3v), GND, SDA and SCL. The example here uses the default I2C port 0, which is assigned to GPIO 4 (SDA) and 5 (SCL) in software. Power is supplied from the 3.3V pin from the Pico.

**WARNING**

The BMP280 has a maximum supply voltage rating of 3.6V. Most breakout boards have voltage regulators that will allow a range of input voltages of 2-6V, but make sure to check beforehand.

List of Files

CMakeLists.txt

CMake file to incorporate the example into the examples build tree.
Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/bmp280_i2c/bmp280_i2c.c

```
1 add_executable(bmp280_i2c
2   bmp280_i2c.c
3 )
4
5 # pull in common dependencies and additional i2c hardware support
6 target_link_libraries(bmp280_i2c pico_stdlib hardware_i2c)
7
8 # create map/bin/hex file etc.
9 pico_add_extra_outputs(bmp280_i2c)
10
11 # add url via pico_set_program_url
12 example_auto_set_url(bmp280_i2c)
```

### bmp280_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/bmp280_i2c/bmp280_i2c.c

```
1 /**
2 * Copyright (c) 2021 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX-License-Identifier: BSD-3-Clause
5 **/
6
7 #include <stdio.h>
8
9 #include "hardware/i2c.h"
10 #include "pico/binary_info.h"
11 #include "pico/stdlib.h"
12
13 /* Example code to talk to a BMP280 temperature and pressure sensor
14
15 NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico
16 GPIO (and therefore 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 PICO_DEFAULT_I2C_SDA_PIN (on Pico this is GP4 (pin 6)) -> SDA on BMP280
24 board
25 GPIO PICO_DEFAULT_I2C_SCK_PIN (on Pico this is GP5 (pin 7)) -> SCL on
26 BMP280 board
27 3.3v (pin 36) -> VCC on BMP280 board
28 GND (pin 38) -> GND on BMP280 board
29 */
30
31 // device has default bus address of 0x76
32 #define ADDR _u(0x76)
33
34 // hardware registers
35 #define REG_CONFIG _u(0xF5)
36 #define REG_CTRL_MEAS _u(0xF4)
37 #define REG_RESET _u(0xE0)
38
39 #define REG_TEMP_XLSB _u(0xFC)
40 #define REG_TEMP_LSB _u(0xFB)
41 #define REG_TEMP_MSB _u(0xFA)
```
#define REG_PRESSURE_XLSB  _u(0xF9)
#define REG_PRESSURE_LSB   _u(0xF8)
#define REG_PRESSURE_MSB   _u(0xF7)

// calibration registers
#define REG_DIG_T1_LSB     _u(0x88)
#define REG_DIG_T1_MSB     _u(0x89)
#define REG_DIG_T2_LSB     _u(0x8A)
#define REG_DIG_T2_MSB     _u(0x8B)
#define REG_DIG_T3_LSB     _u(0x8C)
#define REG_DIG_T3_MSB     _u(0x8D)
#define REG_DIG_P1_LSB     _u(0x8E)
#define REG_DIG_P1_MSB     _u(0x8F)
#define REG_DIG_P2_LSB     _u(0x90)
#define REG_DIG_P2_MSB     _u(0x91)
#define REG_DIG_P3_LSB     _u(0x92)
#define REG_DIG_P3_MSB     _u(0x93)
#define REG_DIG_P4_LSB     _u(0x94)
#define REG_DIG_P4_MSB     _u(0x95)
#define REG_DIG_P5_LSB     _u(0x96)
#define REG_DIG_P5_MSB     _u(0x97)
#define REG_DIG_P6_LSB     _u(0x98)
#define REG_DIG_P6_MSB     _u(0x99)
#define REG_DIG_P7_LSB     _u(0x9A)
#define REG_DIG_P7_MSB     _u(0x9B)
#define REG_DIG_P8_LSB     _u(0x9C)
#define REG_DIG_P8_MSB     _u(0x9D)
#define REG_DIG_P9_LSB     _u(0x9E)
#define REG_DIG_P9_MSB     _u(0x9F)

// number of calibration registers to be read
#define NUM_CALIB_PARAMS 24

struct bmp280_calib_param {
    // temperature params
    uint16_t dig_t1;
    int16_t  dig_t2;
    int16_t  dig_t3;
    
    // pressure params
    uint16_t dig_p1;
    int16_t  dig_p2;
    int16_t  dig_p3;
    int16_t  dig_p4;
    int16_t  dig_p5;
    int16_t  dig_p6;
    int16_t  dig_p7;
    int16_t  dig_p8;
    int16_t  dig_p9;
};

#endif

void bmp280_init() {
    // use the "handheld device dynamic" optimal setting (see datasheet)
    uint8_t buf[2];
    
    // 500ms sampling time, x16 filter
    const uint8_t reg_config_val = ((0x04 << 5) | (0x05 << 2)) & 0xFC;
    
    // send register number followed by its corresponding value
    buf[0] = REG_CONFIG;
    buf[1] = reg_config_val;
    I2C_write_blocking(12c_default, ADDR, buf, 2, false);

// osrs_t x1, osrs_p x4, normal mode operation
const uint8_t reg_ctrl_meas_val = (0x01 << 5) | (0x03 << 2) | (0x03);

buf[0] = REG_CTRL_MEAS;
buf[1] = reg_ctrl_meas_val;

i2c_write_blocking(i2c_default, ADDOR, buf, 2, false);

void bmp280_read_raw(int32_t* temp, int32_t* pressure) {
  // BMP280 data registers are auto-incrementing and we have 3 temperature and
  // pressure registers each, so we start at 0xF7 and read 6 bytes to 0xFC
  // note: normal mode does not require further ctrl_meas and config register writes
  uint8_t buf[6];
  uint8_t reg = REG_PRESSURE_MSB;

  i2c_read_blocking(i2c_default, ADDOR, buf, 6, false); // true to keep master control of
  // bus

  i2c_read_blocking(i2c_default, ADDOR, buf, 6, false); // false - finished with bus

  // store the 28 bit read in a 32 bit signed integer for conversion
  *pressure = (buf[0] << 12) | (buf[1] << 4) | (buf[2] >> 4);
}

void bmp280_reset() {
  // reset the device with the power-on-reset procedure
  uint8_t buf[2] = { REG_RESET, 0xB6 };

  i2c_write_blocking(i2c_default, ADDOR, buf, 2, false);
}

// intermediate function that calculates the fine resolution temperature
// used for both pressure and temperature conversions
int32_t bmp280_convert(int32_t temp, struct bmp280_calib_param* params) {
  // use the 32-bit fixed point compensation implementation given in the
  // datasheet

  int32_t var1, var2;

  var1 = (((temp >> 3) - ((int32_t)params->dig_t1 << 1)) * ((int32_t)params->dig_t2)) >> 11;
  var2 = (((((temp >> 4) - ((int32_t)params->dig_t1)) * ((temp >> 4) - ((int32_t)params->
  - dig_t1))) >> 12) * ((int32_t)params->dig_t3)) >> 14;

  return var1 + var2;
}

int32_t bmp280_convert_temp(int32_t temp, struct bmp280_calib_param* params) {
  // uses the BMP280 calibration parameters to compensate the temperature value read from
  // its registers

  int32_t t_fine = bmp280_convert(temp, params);
  return (t_fine * 5 + 128) >> 8;
}

int32_t bmp280_convert_pressure(int32_t pressure, int32_t temp, struct bmp280_calib_param*
  params) {
  // uses the BMP280 calibration parameters to compensate the pressure value read from its
  // registers

  int32_t var1, var2;

  int32_t converted = 0.0;

  var1 = (((int32_t)t_fine) >> 1) - (int32_t)64000;
  var2 = (((var1 >> 2) * (var1 >> 2)) >> 11) * ((int32_t)params->dig_p6);
  var2 = ((var1 * ((int32_t)params->dig_p5) << 1));
  var2 = (var2 >> 2) + (((int32_t)params->dig_p4) // 16);
Attaching a BMP280 temp/pressure sensor via I2C

```c
164  var1 = (((params->dig_p3 * (var1 >> 2) * (var1 >> 2)) >> 13)) >> 3) + (((int32_t
165  params->dig_p2) * var1) >> 18);
166  var1 = (((32768 + var1)) * (int32_t)params->dig_p1)) >> 15);
167  if (var1 == 0) {
168      return 0; // avoid exception caused by division by zero
169  }
170  converted = (((int32_t)(((int32_t)1848576) - pressure) - (var2 >> 12))) * 3125;
171  if (converted < 0x80000000) {
172      converted = (converted << 1) / ((int32_t)var1);
173  } else {
174      converted = (converted / (int32_t)var1) * 2;
175  }
176  var1 = (((int32_t)params->dig_p9) * ((int32_t)((converted >> 3) * (converted >> 3)) >> 4)) >> 42;
177  var2 = (((int32_t)(converted >> 2)) * ((int32_t)params->dig_p8)) >> 13;
178  converted = (((int32_t)converted + ((var1 + var2 + params->dig_p7) >> 4)));
179  return converted;
180 }
181 void bmp280_get_calib_params(struct bmp280_calib_param* params) {
182  // raw temp and pressure values need to be calibrated according to
183  // parameters generated during the manufacturing of the sensor
184  // there are 3 temperature params, and 9 pressure params, each with a LSB
185  // and MSB register, so we read from 24 registers
186  uint8_t buf[NUM_CALIB_PARAMS] = { 0 };  
187  uint8_t reg = REG_DIG_T1_LSB;
188  i2c_write_blocking(i2c_default, ADDR, &reg, 1, true); // true to keep master control of
189  i2c_read_blocking(i2c_default, ADDR, buf, NUM_CALIB_PARAMS, false); // false, we’re
done reading
190  // store these in a struct for later use
191  params->dig_t1 = (uint16_t)(buf[1] << 8) | buf[0];
192  params->dig_t2 = (int16_t)(buf[3] << 8) | buf[2];
193  params->dig_t3 = (int16_t)(buf[5] << 8) | buf[4];
194  params->dig_p1 = (uint16_t)(buf[7] << 8) | buf[6];
195  params->dig_p2 = (int16_t)(buf[9] << 8) | buf[8];
197  params->dig_p4 = (int16_t)(buf[13] << 8) | buf[12];
198  params->dig_p5 = (int16_t)(buf[15] << 8) | buf[14];
199  params->dig_p6 = (int16_t)(buf[17] << 8) | buf[16];
200  params->dig_p7 = (int16_t)(buf[19] << 8) | buf[18];
201  params->dig_p8 = (int16_t)(buf[21] << 8) | buf[20];
202  params->dig_p9 = (int16_t)(buf[23] << 8) | buf[22];
203 } #endif
204
211  int main() {
212     stdio_init_all();
213     #if (defined(i2c_default) && !defined(PICO_DEFAULT_I2C_SDA_PIN)) ||  
214     !defined(PICO_DEFAULT_I2C_SCL_PIN)
215     #warning i2c / bmp280_i2c example requires a board with I2C pins
216     puts("Default I2C pins were not defined");
217     return 0;
218 #else
219     // useful information for picotool
220     bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN,  
221            GPIO_FUNC_I2C));
222     bi_decl(bi_program_description("BMP280 I2C example for the Raspberry Pi Pico"));
```
printf("Hello, BMP280! Reading temperature and pressure values from sensor...
\n");

// I2C is "open drain", pull ups to keep signal high when no data is being sent
i2c_init(i2c_default, 100 * 1000);

gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);

// configure BMP280
bmp280_init();

// retrieve fixed compensation params
struct bmp280_calib_param params;
bmp280_get_calib_params(&params);

int32_t raw_temperature;
int32_t raw_pressure;
sleep_ms(250); // sleep so that data polling and register update don’t collide

while (1) {
    bmp280_read_raw(&raw_temperature, &raw_pressure);
    int32_t temperature = bmp280_convert_temp(raw_temperature, &params);
    int32_t pressure = bmp280_convert_pressure(raw_pressure, raw_temperature, &params);
    printf("Pressure = %.3f kPa\n", pressure / 1000.0f);
    printf("Temp. = %.2f C\n", temperature / 100.0f);
    sleep_ms(500);
}

#endif

---

Table 24. A list of materials required for the example

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>BMP280-based breakout board</td>
<td>1</td>
<td>from Pimoroni</td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>4</td>
<td>generic part</td>
</tr>
</tbody>
</table>

---

### Attaching a LIS3DH Nano Accelerometer via i2c.

This example shows you how to interface the Raspberry Pi Pico to the LIS3DH accelerometer and temperature sensor.

The code reads and displays the acceleration values of the board in the 3 axes and the ambient temperature value. The datasheet for the sensor can be found at [https://www.st.com/resource/en/datasheet/cd00274221.pdf](https://www.st.com/resource/en/datasheet/cd00274221.pdf). The device is being operated on 'normal mode' and at a frequency of 1.344 kHz (this can be changed by editing the ODR bits of CTRL_REG4). The range of the data is controlled by the FS bit in CTRL_REG4 and is equal to ±2g in this example. The sensitivity depends on the operating mode and data range; exact values can be found on page 10 of the datasheet. In this case, the sensitivity value is 4mg (where g is the value of gravitational acceleration on the surface of Earth). In order to use the auxiliary ADC to read temperature, we must set the BDU bit to 1 in CTRL_REG4 and the ADC_EN bit to 1 in TEMP_CFG_REG. Temperature is communicated through ADC 3.
NOTE

The sensor doesn't have features to eliminate offsets in the data and these will need to be taken into account in the code.

Wiring information

Wiring up the device requires 4 jumpers, to connect VIN, GND, SDA and SCL. The example here uses I2C port 0, which is assigned to GPIO 4 (SDA) and 5 (SCL) in software. Power is supplied from the 3V pin.

List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/lis3dh_i2c/CMakeLists.txt

```plaintext
1  add_executable(lis3dh_i2c
2       lis3dh_i2c.c
3    )
4
5  # pull in common dependencies and additional i2c hardware support
6  target_link_libraries(lis3dh_i2c pico_stdlib hardware_i2c)
7  # create map/bin/hex file etc.
8  pico_add_extra_outputs(lis3dh_i2c)
9  # add url via pico_set_program_url
10 # example_auto_set_url(lis3dh_i2c)
```

lis3dh_i2c.c

The example code.
#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
#include "pico/binary_info.h"
#include "hardware/i2c.h"

/* Example code to talk to a LIS3DH Mini GPS module.

This example reads data from all 3 axes of the accelerometer and uses an auxiliary ADC to output temperature values.

Connections on Raspberry Pi Pico board, other boards may vary.
GPI0 PICO_DEFAULT_I2C_SDA_PIN (On Pico this is 4 (physical pin 6)) -> SDA on LIS3DH board
GPI0 PICO_DEFAULT_I2C_SCK_PIN (On Pico this is 5 (physical pin 7)) -> SCL on LIS3DH board
3.3v (physical pin 36) -> VIN on LIS3DH board
GND (physical pin 38) -> GND on LIS3DH board
*/

// By default this device is on bus address 0x18
const int ADDRESS = 0x18;
const uint8_t CTRL_REG_1 = 0x20;
const uint8_t CTRL_REG_4 = 0x23;
const uint8_t TEMP_CFG_REG = 0xC0;

#ifdef i2c_default

void lis3dh_init()
{
    uint8_t buf[2];

    // Turn normal mode and 1.344kHz data rate on
    buf[0] = CTRL_REG_1;
    buf[1] = 0x97;
    i2c_write_blocking(i2c_default, ADDRESS, buf, 2, false);

    // Turn block data update on (for temperature sensing)
    buf[0] = CTRL_REG_4;
    buf[1] = 0x80;
    i2c_write_blocking(i2c_default, ADDRESS, buf, 2, false);

    // Turn auxillary ADC on
    buf[0] = TEMP_CFG_REG;
    buf[1] = 0x00;
    i2c_write_blocking(i2c_default, ADDRESS, buf, 2, false);
}

void lis3dh_calc_value(uint16_t raw_value, float *final_value, bool isAccel)
{
    // Convert with respect to the value being temperature or acceleration reading
    float scaling;
    float sensitivity = 0.004f; // g per unit

    if (isAccel == true)
    {
        scaling = 64 / sensitivity;
    }
    else
    {
        // Convert to temperature
        // Calculate the temperature
        // Store the temperature
        // Return
    }
}

#endif

Raspberry Pi Pico C/C++ SDK

Attaching a LIS3DH Mini Accelerometer via i2c.
scaling = 64;

// raw_value is signed
*final_value = (float) ((int16_t) raw_value) / scaling;

void lis3dh_read_data(uint8_t reg, float *final_value, bool IsAccel) {
  // Read two bytes of data and store in a 16 bit data structure
  uint8_t lsb;
  uint8_t msb;
  uint16_t raw_accel;
  i2c_write_blocking(i2c_default, ADDRESS, &reg, 1, true);
  i2c_read_blocking(i2c_default, ADDRESS, &lsb, 1, false);
  reg |= 0x01;
  i2c_write_blocking(i2c_default, ADDRESS, &reg, 1, true);
  i2c_read_blocking(i2c_default, ADDRESS, &msb, 1, false);
  raw_accel = (msb << 8) | lsb;
  lis3dh_calc_value(raw_accel, final_value, IsAccel);
}

int main() {
  st dio_init_all();
  #if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN)
    #warning i2c/lis3dh_i2c example requires a board with I2C pins
    puts("Default I2C pins were not defined");
  #else
    printf("Hello, LIS3DH! Reading raw data from registers...\n");
    // This example will use I2C0 on the default SDA and SCL pins (4, 5 on a Pico)
    i2c_init(i2c_default, 400 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
    // Make the I2C pins available to picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));
    float x_accel, y_accel, z_accel, temp;
    lis3dh_init();
    while (1) {
      lis3dh_read_data(0x28, &x_accel, true);
      lis3dh_read_data(0x2A, &y_accel, true);
      lis3dh_read_data(0x2C, &z_accel, true);
      lis3dh_read_data(0x0C, &temp, false);
      // Display data
      printf("TEMPERATURE: %.3f°C\n", temp, 176);
      // Acceleration is read as a multiple of g (gravitational acceleration on the Earth's surface)
      printf("ACCELERATION VALUES: \n");
      printf("X acceleration: %.3fg", x_accel);
      printf("Y acceleration: %.3fg", y_accel);
      printf("Z acceleration: %.3fg", z_accel);
    }
  }
}
// Clear terminal
printf("\033[1;1H\033[2J");
#endif
}

Bill of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>LIS3DH board</td>
<td>1</td>
<td><a href="https://www.adafruit.com/product/2809">https://www.adafruit.com/product/2809</a></td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>4</td>
<td>generic part</td>
</tr>
</tbody>
</table>

Attaching a MCP9808 digital temperature sensor via I2C

This example code shows how to interface the Raspberry Pi Pico to the MCP9808 digital temperature sensor board.

This example reads the ambient temperature value each second from the sensor and sets upper, lower and critical limits for the temperature and checks if alerts need to be raised. The CONFIG register can also be used to check for an alert if the critical temperature is surpassed.

Wiring information

Wiring up the device requires 4 jumpers, to connect VDD, GND, SDA and SCL. The example here uses I2C port 0, which is assigned to GPIO 4 (SDA) and 5 (SCL) in software. Power is supplied from the VSYS pin.
List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/mcp9808_i2c/CMakeLists.txt

```cmake
add_executable(mcp9808_i2c mcp9808_i2c.c)

# pull in common dependencies and additional i2c hardware support
target_link_libraries(mcp9808_i2c pico_stdlib hardware_i2c)

# create map/bin/hex file etc.
pico_add_extra_outputs(mcp9808_i2c)

# add url via pico_set_program_url
example_auto_set_url(mcp9808_i2c)
```

mcp9808_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/mcp9808_i2c/mcp9808_i2c.c

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

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

/* Example code to talk to a MCP9808 ±0.5°C Digital temperature Sensor

This reads and writes to registers on the board.

Connections on Raspberry Pi Pico board, other boards may vary.

GPIO PICO_DEFAULT_I2C_SDA_PIN (On Pico this is GP4 (physical pin 6)) -> SDA on MCP9808 board
GPIO PICO_DEFAULT_I2C_SCK_PIN (On Pico this is GP5 (physical pin 7)) -> SCL on MCP9808 board
Vsys (physical pin 39) --> VDD on MCP9808 board
GND (physical pin 38) --> GND on MCP9808 board
*/

//The bus address is determined by the state of pins A0, A1 and A2 on the MCP9808 board
static uint8_t ADDRESS = 0x18;

//hardware registers
const uint8_t REG_POINTER = 0x00;
const uint8_t REG_CONFIG = 0x01;
const uint8_t REG_TEMP_UPPER = 0x02;
const uint8_t REG_TEMP_LOWER = 0x03;
```
const uint8_t REG_TEMP_CRIT = 0x04;
const uint8_t REG_TEMP_AMB = 0x05;
const uint8_t REG_RESOLUTION = 0x08;

void mcp9808_check_limits(uint8_t upper_byte) {
    // Check flags and raise alerts accordingly
    if ((upper_byte & 0x40) == 0x40) {
        // TA > TUPPER
        printf("Temperature is above the upper temperature limit.\n");
    }
    if ((upper_byte & 0x20) == 0x20) {
        // TA < TLOWER
        printf("Temperature is below the lower temperature limit.\n");
    }
    if ((upper_byte & 0x80) == 0x80) {
        // TA > TCRIT
        printf("Temperature is above the critical temperature limit.\n");
    }
}

float mcp9808_convert_temp(uint8_t upper_byte, uint8_t lower_byte) {
    float temperature;
    // Check if TA <= 0°C and convert to denary accordingly
    if ((upper_byte & 0x10) == 0x10) {
        upper_byte = upper_byte & 0x0F;
        temperature = 256 - (((float) upper_byte * 16) + ((float) lower_byte / 16));
    } else {
        temperature = (((float) upper_byte * 16) + ((float) lower_byte / 16));
    }
    return temperature;
}

#ifdef i2c_default
void mcp9808_set_limits() {
    // Set an upper limit of 30°C for the temperature
    uint8_t upper_temp_msb = 0x01;
    uint8_t upper_temp_lsb = 0xE0;
    // Set a lower limit of 20°C for the temperature
    uint8_t lower_temp_msb = 0x01;
    uint8_t lower_temp_lsb = 0x40;
    // Set a critical limit of 40°C for the temperature
    uint8_t crit_temp_msb = 0x02;
    uint8_t crit_temp_lsb = 0x80;

    uint8_t buf[3];
    buf[0] = REG_TEMP_UPPER;
    buf[1] = upper_temp_msb;
    buf[2] = upper_temp_lsb;
    i2c_write_blocking(i2c_default, ADDRESS, buf, 3, false);
    buf[0] = REG_TEMP_LOWER;
    buf[1] = lower_temp_msb;
    buf[2] = lower_temp_lsb;
    i2c_write_blocking(i2c_default, ADDRESS, buf, 3, false);
    buf[0] = REG_TEMP_CRIT;
    buf[1] = crit_temp_msb;
    buf[2] = crit_temp_lsb;
98     i2c_write_blocking(i2c_default, ADDRESS, buf, 3, false);
99 #endif
100
101 int main() {
102   stdio_init_all();
103
104 #if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) ||
105    !defined(PICO_DEFAULT_I2C_SCL_PIN)
106 #warning i2c/mcp9808_i2c example requires a board with I2C pins
107   puts("Default I2C pins were not defined");
108 #else
109   printf("Hello, MCP9808! Reading raw data from registers...
110
111 // This example will use I2C0 on the default SDA and SCL pins (4, 5 on a Pico)
112 i2c_init(i2c_default, 400 * 1000);
113 gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
114 gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
115 gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
116 gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
117 // Make the I2C pins available to picotool
118 bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));
119 mcp9808_set_limits();
120
121 uint8_t buf[2];
122 uint16_t upper_byte;
123 uint16_t lower_byte;
124
125 float temperature;
126
127 while (1) {
128   // Start reading ambient temperature register for 2 bytes
129   i2c_write_blocking(i2c_default, ADDRESS, &REG_TEMP_AMB, 1, true);
130   i2c_read_blocking(i2c_default, ADDRESS, buf, 2, false);
131
132 upper_byte = buf[0];
133 lower_byte = buf[1];
134
135 //isolates limit flags in upper byte
136 mcp9808_check_limits(upper_byte & 0xE0);
137 //clears flag bits in upper byte
138 temperature = mcp9808_convert_temp(upper_byte & 0x1F, lower_byte);
139 printf("Ambient temperature: %.4f°C\n", temperature);
140
141 sleep_ms(1000);
142 }
143 #endif
144 }
145

---

**Bill of Materials**

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
</tbody>
</table>
### Attaching a MMA8451 3-axis digital accelerometer via I2C

This example code shows how to interface the Raspberry Pi Pico to the MMA8451 digital accelerometer sensor board.

This example reads and displays the acceleration values of the board in the 3 axis. It also allows the user to set the trade-off between the range and precision based on the values they require. Values often have an offset which can be accounted for by writing to the offset correction registers. The datasheet for the sensor can be found at [https://cdn-shop.adafruit.com/datasheets/MMA8451Q-1.pdf](https://cdn-shop.adafruit.com/datasheets/MMA8451Q-1.pdf) for additional information.

### Wiring information

Wiring up the device requires 4 jumpers, to connect VIN, GND, SDA and SCL. The example here uses I2C port 0, which is assigned to GPIO 4 (SDA) and 5 (SCL) in software. Power is supplied from the VSYS pin.

![Wiring Diagram for MMA8451](image)

### List of Files

**CMakeLists.txt**

CMake file to incorporate the example in to the examples build tree.

**Pico Examples:** [https://github.com/raspberrypi/pico-examples/blob/master/i2c/mma8451_i2c/CMakeLists.txt](https://github.com/raspberrypi/pico-examples/blob/master/i2c/mma8451_i2c/CMakeLists.txt)

```cpp
1 add_executable(mma8451_i2c
2   mma8451_i2c.c
3 )
4 # pull in common dependencies and additional i2c hardware support
5 target_link_libraries(mma8451_i2c pico_stdlib hardware_i2c)
6 ```
# create map/bin/hex file etc.
pico_add_extra_outputs(mma8451_i2c)

# add url via pico_set_program_url
example_auto_set_url(mma8451_i2c)

mma8451_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/mma8451_i2c/mma8451_i2c.c

```
/* Example code to talk to a MMA8451 triple-axis accelerometer.

This reads and writes to registers on the board.

Connections on Raspberry Pi Pico board, other boards may vary.

GPIO PICO_DEFAULT_I2C_SDA_PIN (On Pico this is GP4 (physical pin 6)) -> SDA on MMA8451 board
GPIO PICO_DEFAULT_I2C_SCK_PIN (On Pico this is GP5 (physical pin 7)) -> SCL on MMA8451 board
VSYS (physical pin 39) -> VDD on MMA8451 board
GND (physical pin 38)  -> GND on MMA8451 board
*/

const uint8_t ADDRESS = 0x1D;

//hardware registers
const uint8_t REG_X_MSB = 0x01;
const uint8_t REG_X_LSB = 0x02;
const uint8_t REG_Y_MSB = 0x03;
const uint8_t REG_Y_LSB = 0x04;
const uint8_t REG_Z_MSB = 0x05;
const uint8_t REG_Z_LSB = 0x06;
const uint8_t REG_DATA_CFG = 0x0E;
const uint8_t REG_CTRL_REG1 = 0x2A;

// Set the range and precision for the data
const uint8_t range_config = 0x01; // 0x00 for ±2g, 0x01 for ±4g, 0x02 for ±8g
const float count = 2048; // 4096 for ±2g, 2048 for ±4g, 1024 for ±8g

uint8_t buf[2];

float mma8451_convert_accel(uint16_t raw_accel) {
  float acceleration;
  // Acceleration is read as a multiple of g (gravitational acceleration on the Earth’s surface)
  // Check if acceleration < 0 and convert to decimal accordingly
```
if ((raw_accel & 0x2000) == 0x2000) {
    raw_accel &= 0x1FFF;
    acceleration = (-8192 + (float) raw_accel) / count;
} else {
    acceleration = (float) raw_accel / count;
}
acceleration *= 9.81f;
return acceleration;

#ifdef i2c_default
void mma8451_set_state(uint8_t state) {
    buf[0] = REG_CTRL_REG1;
    buf[1] = state; // Set RST bit to 1
    i2c_write_blocking(i2c_default, ADDRESS, buf, 2, false);
}
#else
#endif

int main() {
    stdio_init_all();

#ifdef i2c_default || !defined(PICO_DEFAULT_I2C_SDA_PIN) ||
!defined(PICO_DEFAULT_I2C_SCL_PIN)
    puts("Default I2C pins were not defined");
#else
    printf("Hello, MMA8451! Reading raw data from registers...

    // This example will use I2C0 on the default SDA and SCL pins (4, 5 on a Pico)
    i2c_init(i2c_default, 400 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
    // Make the I2C pins available to picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN,
      GPIO_FUNC_I2C));

    float x_acceleration;
    float y_acceleration;
    float z_acceleration;

    // Enable standby mode
    mma8451_set_state(0x00);

    // Edit configuration while in standby mode
    buf[0] = REG_DATA_CFG;
    buf[1] = range_config;
    i2c_write_blocking(i2c_default, ADDRESS, buf, 2, false);

    // Enable active mode
    mma8451_set_state(0x01);

    while (1) {

    // Start reading acceleration registers for 2 bytes
    i2c_write_blocking(i2c_default, ADDRESS, &REG_X_MSB, 1, true);
    i2c_read_blocking(i2c_default, ADDRESS, buf, 2, false);
    x_acceleration = mma8451_convert_accel(buf[0] << 6 | buf[1] >> 2);

    i2c_write_blocking(i2c_default, ADDRESS, &REG_Y_MSB, 1, true);
    i2c_read_blocking(i2c_default, ADDRESS, buf, 2, false);
    y_acceleration = mma8451_convert_accel(buf[0] << 6 | buf[1] >> 2);
i2c_write_blocking(i2c_default, ADDRESS, &REG_Z_MSB, 1, true);

i2c_read_blocking(i2c_default, ADDRESS, buf, 2, false);

z_acceleration = mma8451_convert_accel(buf[0] << 6 | buf[1] >> 2);

// Display acceleration values
printf("ACCELERATION VALUES:\n\n");
printf("X acceleration: %.6fms^-2\n", x_acceleration);
printf("Y acceleration: %.6fms^-2\n", y_acceleration);
printf("Z acceleration: %.6fms^-2\n", z_acceleration);

sleep_ms(500);

// Clear terminal
printf("\033[1;1H\033[2J");

#ifdef
}
#endif

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>MMA8451 board</td>
<td>1</td>
<td><a href="https://www.adafruit.com/product/2019">https://www.adafruit.com/product/2019</a></td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>4</td>
<td>generic part</td>
</tr>
</tbody>
</table>

### Attaching an MPL3115A2 altimeter via I2C

This example code shows how to interface the Raspberry Pi Pico to an MPL3115A2 altimeter via I2C. The MPL3115A2 has onboard pressure and temperature sensors which are used to estimate the altitude. In comparison to the BMP-family of pressure and temperature sensors, the MPL3115A2 has two interrupt pins for ultra low power operation and takes care of the sensor reading compensation on the board! It also has multiple modes of operation and impressive operating conditions.

The board used in this example comes from Adafruit, but any MPL3115A2 breakouts should work similarly.

The MPL3115A2 makes available two ways of reading its temperature and pressure data. The first is known as polling, where the Pico will continuously read data out of a set of auto-incrementing registers which are refreshed with new data every so often. The second, which this example will demonstrate, uses a 160-byte first-in-first-out (FIFO) queue and configurable interrupts to tell the Pico when to read data. More information regarding when the interrupts can be triggered available in the datasheet. This example waits for the 32 sample FIFO to overflow, detects this via an interrupt pin, and then averages the 32 samples taken. The sensor is configured to take a sample every second.

Bit math is used to convert the temperature and altitude data from the raw bits collected in the registers. Take the temperature calculation as an example: it is a 12-bit signed number with 8 integer bits and 4 fractional bits. First, we read the 2 8-bit registers and store them in a buffer. Then, we concatenate them into one unsigned 16-bit integer starting with the OUT_T_MSB register, thus making sure that the last bit of this register is aligned with the MSB in our 16 bit unsigned integer so it is correctly interpreted as the signed bit when we later cast this to a signed 16-bit integer. Finally, the entire number is converted to a float implicitly when we multiply it by $1/2^8$ to shift it 8 bits to the right of the decimal point. Though only the last 4 bits of the OUT_T_LSB register hold data, this does not matter as the remaining 4
are held at zero and “disappear” when we shift the decimal point left by 8. Similar logic is applied to the altitude calculation.

💡 TIP

Choosing the right sensor for your project among so many choices can be hard! There are multiple factors you may have to consider in addition to any constraints imposed on you. Cost, operating temperature, sensor resolution, power consumption, ease of use, communication protocols and supply voltage are all but a few factors that can play a role in sensor choice. For most hobbyist purposes though, the majority of sensors out there will do just fine!

**Wiring information**

Wiring up the device requires 5 jumpers, to connect VCC (3.3v), GND, INT1, SDA and SCL. The example here uses I2C port 0, which is assigned to GPIO 4 (SDA) and GPIO 5 (SCL) by default. Power is supplied from the 3.3V pin.

⚠️ NOTE

The MPL3115A2 has a 1.6-3.6V voltage supply range. This means it can work with the Pico’s 3.3v pins out of the box but our Adafruit breakout has an onboard voltage regulator for good measure. This may not always be true of other sensors, though.

---

**List of Files**

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.

Pico Examples: [https://github.com/raspberrypi/pico-examples/blob/master/i2c/mpl3115a2_i2c/CMakeLists.txt](https://github.com/raspberrypi/pico-examples/blob/master/i2c/mpl3115a2_i2c/CMakeLists.txt)

```cpp
1 add_executable(mpl3115a2_i2c mpl3115a2_i2c.c )
2 # pull in common dependencies and additional i2c hardware support
3 target_link_libraries(mpl3115a2_i2c pico_stdlib hardware_i2c)
4 5
6
7```

---

*Attaching an MPL3115A2 altimeter via I2C*
mpl3115a2_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/mpl3115a2_i2c/mpl3115a2_i2c.c

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "pico/binary_info.h"
#include "hardware/gpio.h"
#include "hardware/i2c.h"

/* Example code to talk to an MPL3115A2 altimeter sensor via I2C

See accompanying documentation in README.adoc or the C++ SDK booklet.

Connections on Raspberry Pi Pico board, other boards may vary.

GPIO PICO_DEFAULT_I2C_SDA_PIN (On Pico this is 4 (pin 6)) -> SDA on MPL3115A2 board
GPIO PICO_DEFAULT_I2C_SCK_PIN (On Pico this is 5 (pin 7)) -> SCL on MPL3115A2 board
GPIO 16 -> INT1 on MPL3115A2 board
3.3V (pin 36) -> VCC on MPL3115A2 board
GND (pin 38) -> GND on MPL3115A2 board
*/

// 7-bit address
#define ADDR 0x60
#define INT1_PIN _u(16)

// following definitions only valid for F_MODE > 0 (ie. if FIFO enabled)
#define MPL3115A2_F_DATA _u(0x01)
#define MPL3115A2_F_STATUS _u(0x00)
#define MPL3115A2_F_SETUP _u(0x80)
#define MPL3115A2_F_INT_SOURCE _u(0x12)
#define MPL3115A2_CTRLREG1 _u(0x26)
#define MPL3115A2_CTRLREG2 _u(0x27)
#define MPL3115A2_CTRLREG3 _u(0x28)
#define MPL3115A2_CTRLREG4 _u(0x29)
#define MPL3115A2_CTRLREG5 _u(0x2A)
#define MPL3115A2_PT_DATA_CFG _u(0x13)
#define MPL3115A2_OFF_P _u(0x2B)
#define MPL3115A2_OFF_T _u(0x2C)
#define MPL3115A2_OFF_H _u(0x2D)

#define MPL3115A2_FIFO_DISABLED _u(0x00)
#define MPL3115A2_FIFO_STOP_ON_OVERFLOW _u(0x80)
#define MPL3115A2_FIFO_SIZE 32
#define MPL3115A2_DATA_BATCH_SIZE 5
#define MPL3115A2_ALTITUDE_NUM_REGS 3
#define MPL3115A2_ALTITUDE_INT_SIZE 20
#define MPL3115A2_TEMPERATURE_INT_SIZE 12
```
```c
#define MPL3115A2_NUM_FRAC_BITS 4
#define PARAM_ASSERTIONS_ENABLE_I2C 1
c volatile uint8_t fifo_data[MPL3115A2_FIFO_SIZE * MPL3115A2_DATA_BATCH_SIZE];
c volatile bool has_new_data = false;
c struct mpl3115a2_data_t {
   // Q8.4 fixed point
   float temperature;
   // Q16.4 fixed-point
   float altitude;
};

void copy_to_vbuf(uint8_t buf1[], volatile uint8_t buf2[], int buflen) {
   for (size_t i = 0; i < buflen; i++) {
      buf2[i] = buf1[i];
   }
}

#ifdef i2c_default
void mpl3115a2_read_fifo(volatile uint8_t fifo_buf[]) {
   // drains the 160 byte FIFO
   uint8_t reg = MPL3115A2_F_DATA;
   uint8_t buf[MPL3115A2_FIFO_SIZE * MPL3115A2_DATA_BATCH_SIZE];
   i2c_write_blocking(i2c_default, ADDR, &reg, 1, true);
   // burst read 160 bytes from fifo
   i2c_read_blocking(i2c_default, ADDR, buf, MPL3115A2_FIFO_SIZE *
                    MPL3115A2_DATA_BATCH_SIZE, false);
   copy_to_vbuf(buf, fifo_buf, MPL3115A2_FIFO_SIZE * MPL3115A2_DATA_BATCH_SIZE);
}

uint8_t mpl3115a2_read_reg(uint8_t reg) {
   uint8_t read;
   i2c_write_blocking(i2c_default, ADDR, &reg, 1, true); // keep control of bus
   i2c_read_blocking(i2c_default, ADDR, &read, 1, false);
   return read;
}

void mpl3115a2_init() {
   // set as altimeter with oversampling ratio of 128
   uint8_t buf[] = {MPL3115A2_CTRLREG1, 0xB8};
   i2c_write_blocking(i2c_default, ADDR, buf, 2, false);
   // set data refresh every 2 seconds, 0 next bits as we're not using those interrupts
   buf[0] = MPL3115A2_CTRLREG2, buf[1] = 0x00;
   i2c_write_blocking(i2c_default, ADDR, buf, 2, false);
   // set both interrupts pins to active low and enable internal pullups
   buf[0] = MPL3115A2_CTRLREG3, buf[1] = 0x01;
   i2c_write_blocking(i2c_default, ADDR, buf, 2, false);
   // enable FIFO interrupt
   buf[0] = MPL3115A2_CTRLREG4, buf[1] = 0x40;
   i2c_write_blocking(i2c_default, ADDR, buf, 2, false);
   // tie FIFO interrupt to pin INT!
   buf[0] = MPL3115A2_CTRLREG5, buf[1] = 0x40;
   i2c_write_blocking(i2c_default, ADDR, buf, 2, false);
   // set p, t and h offsets here if needed
   // eg. 2's complement number: 0xFF subtracts 1 meter
   //buf[0] = MPL3115A2_OFF_H, buf[1] = 0xFF;
```

Raspberry Pi Pico C/C++ SDK

Attaching an MPL3115A2 altimeter via I2C
// do not accept more data on FIFO overflow
buf[0] = MPL3115A2_F.Setup, buf[1] = MPL3115A2_FIFO_STOP_ON_OVERFLOW;
i2c_write_blocking(i2c_default, ADDR, buf, 2, false);

// set device active
buf[0] = MPL3115A2_CTRLREG1, buf[1] = 0xB9;
i2c_write_blocking(i2c_default, ADDR, buf, 2, false);

void gpio_callback(uint gpio, uint32_t events)
{
    // if we had enabled more than 2 interrupts on same pin, then we should read
    // INT_SOURCE reg to find out which interrupt triggered

    // we can filter by which GPIO was triggered
    if (gpio == INT1_PIN)
    {
        // FIFO overflow interrupt
        // watermark bits set to 0 in F_SETUP reg, so only possible event is an overflow
        printf("FIFO overflow!\n");
        // drain the fifo
        mpl3115a2_read_fifo(fifo_data);
        // read status register to clear interrupt bit
        mpl3115a2_read_reg(MPL3115A2_F_STATUS);
        has_new_data = true;
    }
}

void mpl3115a2_convert_fifo_batch(uint8_t start, volatile uint8_t buf[], struct mpl3115a2_data_t *data)
{
    // convert a batch of fifo data into temperature and altitude data

    // 3 altitude registers: MSB (8 bits), CSB (8 bits) and LSB (4 bits, starting from MSB)
    // first two are integer bits (2's complement) and LSB is fractional bits -> makes 20 bit signed integer
    int32_t h = (int32_t)buf[start] << 24;
    h |= (int32_t)buf[start + 1] << 16;
    h |= (int32_t)buf[start + 2] << 8;
    data->altitude = ((float)h) / 65536.f;

    // 2 temperature registers: MSB (8 bits) and LSB (4 bits, starting from MSB)
    // first 8 are integer bits with sign and LSB is fractional bits -> 12 bit signed integer
    int16_t t = (int16_t)buf[start + 3] << 8;
    t |= (int16_t)buf[start + 4];
    data->temperature = ((float)t) / 256.f;
}

int main()
{
    stdio_init_all();

    if (defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN))
        !defined(PICO_DEFAULT_I2C_SDA_PIN)
    #warning i2c / mpl3115a2_i2c example requires a board with I2C pins
    puts("Default I2C pins were not defined");
    return 0;
    #else

    printf("Hello, MPL3115A2. Waiting for something to interrupt me!...
");

    // use default I2C0 at 400kHz, I2C is active low
    i2c_init(i2c_default, 400 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
176  gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
177  gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
178
179  gpio_init(INT1_PIN);
180  gpio_pull_up(INT1_PIN); // pull it up even more!
181
182  // add program information for picotool
183  bi_decl(bi_program_name("Example in the pico-examples library for the MPL3115A2 altimeter"));
184  bi_decl(bi_1pin_with_name(16, "Interrupt pin 1"));
185  bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN,
186     GPIO_FUNC_I2C));
187
188  mpl3115a2_init();
189
190  gpio_set_irq_enabled_with_callback(INT1_PIN, GPIO_IRQ_LEVEL_LOW, true, &gpio_callback);
191
192  while (1) {
193      // as interrupt data comes in, let's print the 32 sample average
194      if (has_new_data) {
195          float tsum = 0, hsum = 0;
196          struct mpl3115a2_data_t data;
197          for (int i = 0; i < MPL3115A2_FIFO_SIZE; i++) {
198              mpl3115a2_convert_fifo_batch(i * MPL3115A2_DATA_BATCH_SIZE, fifo_data, &
199                  data);
200                  tsum += data.temperature;
201                  hsum += data.altitude;
202          }
203          printf("%d sample average -> t: %.4f C, h: %.4f m\n", MPL3115A2_FIFO_SIZE, tsum
204              / MPL3115A2_FIFO_SIZE,
205              hsum / MPL3115A2_FIFO_SIZE);
206          has_new_data = false;
207      }
208      sleep_ms(10);
209  }
210
Bill of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>MPL3115A2 altimeter</td>
<td>1</td>
<td>Adafruit</td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>5</td>
<td>generic part</td>
</tr>
</tbody>
</table>

Attaching an OLED display via I2C

This example code shows how to interface the Raspberry Pi Pico with an 128x32 OLED display board based on the SSD1306 display driver, datasheet [here](#).

The code displays a series of small demo graphics; tiny raspberries that scroll horizontally, some text, and some line
drawing, in the process showing you how to initialize the display, write to the entire display, write to only a portion of the display, configure scrolling, invert the display etc.

The SSD1306 is operated via a list of versatile commands (see datasheet) that allows the user to access all the capabilities of the driver. After sending a slave address, the data that follows can be either a command, flags to follow up a command or data to be written directly into the display’s RAM. A control byte is required for each write after the slave address so that the driver knows what type of data is being sent.

The example code supports displays of 32 pixel or 64 pixels high by 128 pixels wide by changing a define at the top of the code.

In the 32 vertical pixels case, the display is partitioned into 4 pages, each 8 pixels in height. In RAM, this looks roughly like:

```
<table>
<thead>
<tr>
<th>COL0</th>
<th>COL1</th>
<th>COL2</th>
<th>COL3</th>
<th>...</th>
<th>COL126</th>
<th>COL127</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGE 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAGE 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAGE 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAGE 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
--------------------------------------------------------------
```

Within each page, we have:

```
<table>
<thead>
<tr>
<th>COL0</th>
<th>COL1</th>
<th>COL2</th>
<th>COL3</th>
<th>...</th>
<th>COL126</th>
<th>COL127</th>
</tr>
</thead>
<tbody>
<tr>
<td>COM 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COM 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COM 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
-------------------------------------------------------------
```

**NOTE**

There is a difference between columns in RAM and the actual segment pads that connect the driver to the display. The RAM addresses COL0 · COL127 are mapped to these segment pins SEG0 · SEG127 by default. The distinction between these two is important as we can for example, easily mirror contents of RAM without rewriting a buffer.

The driver has 3 modes of transferring the pixels in RAM to the display (provided that the driver is set to use its RAM content to drive the display, i.e. command 0xA4 is sent). We choose horizontal addressing mode which, after setting the column address and page address registers to our desired start positions, will increment the column address register until the OLED display width is reached (127 in our case) after which the column address register will reset to its starting value and the page address is incremented. Once the page register reaches the end, it will wrap around as well. Effectively, this scans across the display from top to bottom, left to right in blocks that are 8 pixels high. When a byte is sent to be written into RAM, it sets all the rows for the current position of the column address register. So, if we send 10101010, and we are on PAGE 0 and COL1, COM0 is set to 1, COM1 is set to 0, COM2 is set to 1, and so on. Effectively, the byte is “transposed” to fill a single page’s column. The datasheet has further information on this and the two other modes.

Horizontal addressing mode has the key advantage that we can keep one single 512 byte buffer (128 columns x 4 pages and each byte fills a page’s rows) and write this in one go to the RAM (column address auto increments on writes as well as reads) instead of working with 2D matrices of pixels and adding more overhead.

**Wiring information**

Wiring up the device requires 4 jumpers, to connect VCC (3.3v), GND, SDA and SCL and optionally a 5th jumper for the driver RESET pin. The example here uses the default I2C port 0, which is assigned to GPIO 4 (SDA) and 5 (SCL) in...
software. Power is supplied from the 3.3V pin from the Pico.

List of Files

CMakeLists.txt

CMake file to incorporate the example into the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/ssd1306_i2c/CMakeLists.txt

```c
1 add_executable(ssd1306_i2c
2     ssd1306_i2c.c
3 )
4
5 # pull in common dependencies and additional i2c hardware support
6 target_link_libraries(ssd1306_i2c pico_stdlib hardware_i2c)
7
8 # create map/bin/hex file etc.
9 pico_add_extra_outputs(ssd1306_i2c)
10
11 # add url via pico_set_program_url
12 example_auto_set_url(ssd1306_i2c)
```

ssd1306_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/ssd1306_i2c/ssd1306_i2c.c

```c
1 /**
2 * Copyright (c) 2021 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX-License-Identifier: BSD-3-Clause
5 */
6
7 #include <stdio.h>
8 #include <string.h>
9 #include <stdlib.h>
```
/* Example code to talk to an SSD1306-based OLED display

The SSD1306 is an OLED/PLED driver chip, capable of driving displays up to
128x64 pixels.

NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico
GPIO (and therefore I2C) cannot be used at 5v.

You will need to use a level shifter on the I2C lines if you want to run the
board at 5v.

Connections on Raspberry Pi Pico board, other boards may vary.

GPIO PICO_DEFAULT_I2C_SDA_PIN (on Pico this is GP4 (pin 6)) -> SDA on display
board
GPIO PICO_DEFAULT_I2C_SCL_PIN (on Pico this is GP5 (pin 7)) -> SCL on
display board
3.3v (pin 36) -> VCC on display board
GND (pin 38) -> GND on display board
*/

// Define the size of the display we have attached. This can vary, make sure you
// have the right size defined or the output will look rather odd!
// Code has been tested on 128x32 and 128x64 OLED displays
#define SSD1306_HEIGHT 32
#define SSD1306_WIDTH 128
#define SSD1306_I2C_ADDR _u(0x3C)
#define SSD1306_I2C_CLK 400

// commands (see datasheet)
#define SSD1306_SET_MEM_MODE _u(0x20)
#define SSD1306_SET_COL_ADDR _u(0x21)
#define SSD1306_SET_PAGE_ADDR _u(0x22)
#define SSD1306_SET_HORIZ_SCROLL _u(0x26)
#define SSD1306_SET_SCROLL _u(0x2E)
#define SSD1306_SET_DISP_START_LINE _u(0x40)
#define SSD1306_SET_CONTRAST _u(0x81)
#define SSD1306_SET_CHARGE_PUMP _u(0xA8)
#define SSD1306_SET_SEG_REMAP _u(0xA0)
#define SSD1306_SET_ENTIRE_ON _u(0xA4)
#define SSD1306_SET_INV_DISP _u(0xA7)
#define SSD1306_SET_MUX_RATIO _u(0x70)
#define SSD1306_SET_DISP _u(0xAE)
#define SSD1306_SET_COM_OUT_DIR _u(0xC0)
#define SSD1306_SET_COM_OUT_DIR_FLIP _u(0xC8)
```c
#define SSD1306_SET_DISP_OFFSET _u(0xD3)
#define SSD1306_SET_DISP_CLK_DIV    _u(0xD5)
#define SSD1306_SET_PRECHARGE       _u(0xD9)
#define SSD1306_SET_COM_PIN_CFG     _u(0xDA)
#define SSD1306_SET_VCOM_DESEL      _u(0xDB)
#define SSD1306_PAGE_HEIGHT         _u(8)
#define SSD1306_NUM_PAGES           (SSD1306_HEIGHT / SSD1306_PAGE_HEIGHT)
#define SSD1306_BUF_LEN             (SSD1306_NUM_PAGES * SSD1306_WIDTH)
#define SSD1306_WRITE_MODE         _u(0xFE)
#define SSD1306_READ_MODE          _u(0xFF)

struct render_area {
    uint8_t start_col;
    uint8_t end_col;
    uint8_t start_page;
    uint8_t end_page;
    int buflen;
};

void calc_render_area_buflen(struct render_area *area)
{
    // calculate how long the flattened buffer will be for a render area
    area->buflen = (area->end_col - area->start_col + 1) * (area->end_page - area->start_page + 1);
}

#ifdef i2c_default

void SSD1306_send_cmd(uint8_t cmd) {
    // I2C write process expects a control byte followed by data
    // this "data" can be a command or data to follow up a command
    // Co = 1, D/C = 0 => the driver expects a command
    uint8_t buf[2] = {0x80, cmd};
    i2c_write_blocking(i2c_default, SSD1306_I2C_ADDR, buf, 2, false);
}

void SSD1306_send_cmd_list(uint8_t *buf, int num) {
    for (int i=0; i<num; i++)
        SSD1306_send_cmd(buf[i]);
}

void SSD1306_send_buf(uint8_t buf[], int buflen) {
    // in horizontal addressing mode, the column address pointer auto-increments
    // and then wraps around to the next page, so we can send the entire frame
    // buffer in one gooooooo!
    // copy our frame buffer into a new buffer because we need to add the control byte
    // to the beginning
    uint8_t *temp_buf = malloc(buflen + 1);
    temp_buf[0] = 0x40;
    memcpy(temp_buf+1, buf, buflen);
    i2c_write_blocking(i2c_default, SSD1306_I2C_ADDR, temp_buf, buflen + 1, false);
    free(temp_buf);
}

void SSD1306_init() {
    // Some of these commands are not strictly necessary as the reset
```
// process defaults to some of these but they are shown here to demonstrate what the initialization sequence looks like

// Some configuration values are recommended by the board manufacturer

uint8_t cmds[] = {
    SSD1306_SET_DISP, // set display off
    /* memory mapping */
    SSD1306_SET_MEM_MODE, // set memory address mode 0 = horizontal, 1 = vertical, 2 = page
    0x00, // horizontal addressing mode
    /* resolution and layout */
    SSD1306_SET_DISP_START_LINE, // set display start line to 0
    SSD1306_SETSEG_REMAP | 0x01, // set segment re-map, column address 127 is mapped to SEG0
    SSD1306_SET_MUX_RATIO, // set multiplex ratio
    SSD1306_HEIGHT - 1, // Display height - 1
    SSD1306_SET_COM_PIN_CFG, // set COM (common) pins hardware configuration.
    Board specific magic number.
    // 0x02 Works for 128x32, 0x12 Possibly works for 128x64. Other options 0x22, 0x32
    #if ((SSD1306_WIDTH == 128) && (SSD1306_HEIGHT == 32))
        0x02,
    #elif ((SSD1306_WIDTH == 128) && (SSD1306_HEIGHT == 64))
        0x12,
    #else
        0x02,
    #endif
    /* timing and driving scheme */
    SSD1306_SET_DISP_CLK_DIV, // set display clock divide ratio
    0x80, // div ratio of 1, standard freq
    SSD1306_SET_PRECHARGE, // set pre-charge period
    0xF1, // Vcc internally generated on our board
    SSD1306_SET_VCOM_DESEL, // set VCOMH deselect level
    0x30, // 0.83xVcc
    /* display */
    SSD1306_SETCONTRAST, // set contrast control
    0xFF,
    /* timing and driving scheme */
    SSD1306_SET_DISP | 0x01, // turn display on
    SSD1306_SEND_CMD_LIST(cmds, count_of(cmds));
};

void SSD1306_scroll(bool on) {
    uint8_t cmds[] = {
        SSD1306_SET_HORIZ_SCROLL | 0x00,
        0x00, // dummy byte
        0x00, // start page 0
        0x00, // time interval
        0x00, // dummy byte
        0x FF, // dummy byte
        SSD1306_SET_SCROLL | (on ? 0x01 : 0) // Start/stop scrolling
    };
    SSD1306_SEND_CMD_LIST(cmds, count_of(cmds));
}
200 void render(uint8_t *buf, struct render_area *area) {
201 // update a portion of the display with a render area
202 uint8_t cmds[] = {
203 SSD1306_SET_COL_ADDR,
204 area->start_col,
205 area->end_col,
206 SSD1306_SET_PAGE_ADDR,
207 area->start_page,
208 area->end_page
209 };
210 SSD1306_send_cmd_list(cmds, count_of(cmds));
211 SSD1306_send_buf(buf, area->buflen);
212 }
213
214 static void SetPixel(uint8_t *buf, int x, int y, bool on) {
215 assert(x >= 0 && x < SSD1306_WIDTH && y >= 0 && y < SSD1306_HEIGHT);
216 // The calculation to determine the correct bit to set depends on which address
217 // mode we are in. This code assumes horizontal
218 // The video ram on the SSD1306 is split up in to 8 rows, one bit per pixel.
219 // Each row is 128 long by 8 pixels high, each byte vertically arranged, so byte 0 is x=0,
220 // y=0->7,
221 // byte 1 is x = 1, y=8->7 etc
222 // This code could be optimised, but is like this for clarity. The compiler
223 // should do a half decent job optimising it anyway.
224 const int BytesPerRow = SSD1306_WIDTH / 8; // x pixels, 1bpp, but each row is 8 pixel high,
225 // so (x / 8) * 8
226 int byte_idx = (y / 8) * BytesPerRow + x;
227 uint8_t byte = buf[byte_idx];
228 if (on)
229 byte |= 1 << (y % 8);
230 else
231 byte &= ~(1 << (y % 8));
232 buf[byte_idx] = byte;
233 }
234 // Basic Bresenham.
235 static void DrawLine(uint8_t *buf, int x0, int y0, int x1, int y1, bool on) {
236 int dx = abs(x1-x0);
237 int sx = x0<x1 ? 1 : -1;
238 int dy = -abs(y1-y0);
239 int sy = y0<y1 ? 1 : -1;
240 int err = dx+dy;
241 int e2;
242 while (true) {
243 SetPixel(buf, x0, y0, on);
244 if (x0 == x1 && y0 == y1)
245 break;
246 e2 = 2*err;
247 if (e2 >= dy) {
248 x0 += sx;
249 y0 += sy;
250 err += dy;
251 } else {
252 x0 += sx;
253 y0 += sy;
254 err -= dx;
255 }}}
257 err += dy;
258 x0 += sx;
259 }
260 if (e2 <= dx) {
261 err += dx;
262 y0 += sy;
263 }
264 }
265 }
266
267 static inline int GetFontIndex(uint8_t ch) {
268 if (ch >= 'A' && ch <= 'Z') {
269 return ch - 'A' + 1;
270 }
271 else if (ch >= '0' && ch <= '9') {
272 return ch - '0' + 27;
273 }
274 else return 0; // Not got that char so space.
275 }
276
277 static uint8_t reversed[sizeof(font)] = {0};
278
279 static uint8_t reverse(uint8_t b) {
280 b = (b & 0xF0) >> 4 | (b & 0x0F) << 4;
281 b = (b & 0xCC) >> 2 | (b & 0x33) << 2;
282 b = (b & 0xAA) >> 1 | (b & 0x55) << 1;
283 return b;
284 }
285 static void FillReversedCache() {
286 // calculate and cache a reversed version of the font, because I defined it upside
287 // down...doh!
288 for (int i=0; i<sizeof(font); i++)
289 reversed[i] = reverse(font[i]);
289 }
290
291 static void WriteChar(uint8_t *buf, int16_t x, int16_t y, uint8_t ch) {
292 if (reversed[0] == 0)
293 FillReversedCache();
294 if (x > SSD1306_WIDTH - 8 || y > SSD1306_HEIGHT - 8)
295 return;
296
297 // For the moment, only write on Y row boundaries (every 8 vertical pixels)
298 y = y/8;
299
300 ch = toupper(ch);
301 int idx = GetFontIndex(ch);
302 int fb_idx = y * 128 + x;
303
304 for (int i=0; i<8; i++)
305 buf[fb_idx++] = reversed[idx * 8 + i];
306 }
307 }
308 }
309
310 static void WriteString(uint8_t *buf, int16_t x, int16_t y, char *str) {
311 // Cull out any string off the screen
312 if (x > SSD1306_WIDTH - 8 || y > SSD1306_HEIGHT - 8)
313 return;
314 while (*str) {
315 WriteChar(buf, x, y, *str++);
316 x += 8;
317 }
319 }
#ifdef i2c_default
int main()
{
    stdio_init_all();

    #if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) ||
        !defined(PICO_DEFAULT_I2C_SCL_PIN)
    #warning i2c / SSD1306_i2d example requires a board with I2C pins
    puts("Default I2C pins were not defined");
    #else
        // useful information for picotool
        bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN,
                                   PICO_DEFAULT_I2C_SCL_PIN,
                                   GPIO_FUNC_I2C));
        bi_decl(bi_program_description("SSD1306 OLED driver I2C example for the Raspberry Pi Pico"));
    #endif
    printf("Hello, SSD1306 OLED display! Look at my raspberries..\n\n"),

    // I2C is "open drain", pull ups to keep signal high when no data is being sent
    i2c_init(i2c_default, SSD1306_I2C_CLK * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);

    // run through the complete initialization process
    SSD1306_init();

    // Initialize render area for entire frame (SSD1306_WIDTH pixels by SSD1306_NUM_PAGES pages)
    struct render_area frame_area = {
        start_col: 0,
        end_col: SSD1306_WIDTH - 1,
        start_page: 0,
        end_page: SSD1306_NUM_PAGES - 1
    };
    calc_render_area_buflen(&frame_area);

    // zero the entire display
    uint8_t buf[SSD1306_BUF_LEN];
    memset(buf, 0, SSD1306_BUF_LEN);
    render(buf, &frame_area);

    // intro sequence: flash the screen 3 times
    for (int i = 0; i < 3; i++) {
        SSD1306_send_cmd(SSD1306_SET_ALL_ON); // Set all pixels on
        sleep_ms(500);
        SSD1306_send_cmd(SSD1306_SET_ENTIRE_ON); // go back to following RAM for pixel state
        sleep_ms(500);
    }

    // render 3 cute little raspberries
    struct render_area area = {
        start_page: 0,
        end_page: (IMG_HEIGHT / SSD1306_PAGE_HEIGHT) - 1
    };
    restart:
}
area.start_col = 0;
area.end_col = IMG_WIDTH - 1;
calc_render_area_buflen(&area);

uint8_t offset = 5 + IMG_WIDTH; // 5px padding
for (int i = 0; i < 3; i++) {
    render(raspberry26x32, &area);
    area.start_col += offset;
    area.end_col += offset;
}
SSD1306_scroll(true);
sleep_ms(5000);
SSD1306_scroll(false);

char *text[] = {
    "A long time ago",
    " on an OLED ",
    " display",
    " far far away",
    "Lived a small",
    "red raspberry",
    "by the name of",
    " PICO"
};

int y = 0;
for (int i = 0 ;i < count_of(text); i++) {
    WriteString(buf, 5, y, text[i]);
    y+=8;
}
render(buf, &frame_area);

// Test the display invert function
sleep_ms(3000);
SSD1306_send_cmd(SSD1306_SET_INV_DISP);
sleep_ms(3000);
SSD1306_send_cmd(SSD1306_SET_NORM_DISP);

bool pix = true;
for (int i = 0; i < 2;i++) {
    for (int x = 0;x < SSD1306_WIDTH;x++) {
        DrawLine(buf, x, 0, SSD1306_WIDTH - 1 - x, SSD1306_HEIGHT - 1, pix);
        render(buf, &frame_area);
    }
    pix = false;
}
goto restart;

#endif
return 0;
ssd1306_font.h

A simple font used in the example.

Pico Examples: https://github.com/raspberry-pico-examples/blob/master/l2c/ssd1306_l2c/ssd1306_font.h

```c
1 /*
2 * Copyright (c) 2022 Raspberry Pi (Trading) Ltd.
3 */
4 * SPDX-License-Identifier: BSD-3-Clause
5 */
6
7 // Vertical bitmaps, A-Z, 0-9. Each is 8 pixels high and wide
8 // These are defined vertically to make them quick to copy to FB
9
10 static uint8_t font[] = {
11 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // Nothing
12 0x1e, 0x28, 0x48, 0x48, 0x48, 0x28, 0x1e, 0x00, // A
13 0xfe, 0x92, 0x92, 0x92, 0x92, 0xfe, 0x00, 0x00, // B
14 0x7e, 0x82, 0x82, 0x82, 0x82, 0x82, 0x82, 0x82, // C
15 0xfe, 0x82, 0x82, 0x82, 0x82, 0x82, 0x7e, 0x00, // D
16 0xfe, 0x92, 0x92, 0x92, 0x92, 0x92, 0xe0, 0x00, // E
17 0xfe, 0x90, 0x90, 0x90, 0x90, 0x88, 0x88, 0x80, // F
18 0xfe, 0x82, 0x82, 0x82, 0x8a, 0x8a, 0xce, 0x00, // G
19 0xfe, 0x10, 0x10, 0x10, 0x10, 0x10, 0xfe, 0x00, // H
20 0x00, 0x00, 0x00, 0xfe, 0x00, 0x00, 0x00, 0x00, // I
21 0x84, 0x82, 0x82, 0xfc, 0x80, 0x80, 0x80, 0x80, // J
22 0x00, 0xfe, 0x10, 0x10, 0x28, 0x44, 0x82, 0x00, // K
23 0xfe, 0x82, 0x02, 0x02, 0x02, 0x02, 0x02, 0x00, // L
24 0xfe, 0x40, 0x20, 0x10, 0x20, 0x40, 0xfe, 0x00, // M
25 0xfe, 0x40, 0x20, 0x10, 0x08, 0x04, 0xfe, 0x00, // N
26 0x7c, 0x82, 0x82, 0x82, 0x82, 0x7c, 0x00, 0x00, // O
27 0xfe, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08, 0x70, // P
28 0x7c, 0x82, 0x82, 0x92, 0x8a, 0x86, 0x7e, 0x00, // Q
29 0xfe, 0x08, 0x08, 0x08, 0x8c, 0x8a, 0x70, 0x00, // R
30 0x02, 0x92, 0x92, 0x92, 0x92, 0x8c, 0x00, 0x00, // S
31 0x08, 0x00, 0x80, 0xfe, 0x88, 0x88, 0x88, 0x80, // T
32 0xc0, 0x82, 0x82, 0x82, 0x82, 0xc0, 0x00, 0x00, // U
33 0xf0, 0x08, 0x04, 0x02, 0x04, 0x05, 0xf0, 0x00, // V
34 0xfe, 0x04, 0x08, 0x10, 0x08, 0x04, 0xfe, 0x00, // W
35 0x00, 0x82, 0x44, 0x28, 0x28, 0x44, 0x82, 0x00, // X
36 0x00, 0x40, 0x20, 0x1e, 0x20, 0x40, 0x80, 0x00, // Y
37 0x82, 0x86, 0x9a, 0xa2, 0xc2, 0x82, 0x80, 0x00, // Z
38 0x7c, 0x82, 0x82, 0x92, 0x82, 0x7c, 0x00, 0x00, // a
39 0x00, 0x00, 0x42, 0xfe, 0x82, 0x00, 0x80, 0x00, // b
40 0x8c, 0x92, 0x92, 0x92, 0x92, 0x62, 0x00, 0x00, // c
41 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x00, // d
42 0xc0, 0x04, 0x04, 0x1e, 0x04, 0x04, 0x00, 0x00, // e
43 0xf2, 0x92, 0x92, 0x92, 0x92, 0x8c, 0x00, 0x00, // f
44 0xc0, 0x12, 0x12, 0x12, 0x12, 0x8c, 0x00, // g
45 0x00, 0x80, 0x80, 0x86, 0x8c, 0xb0, 0xc8, 0x00, // h
46 0xc0, 0x92, 0x92, 0x92, 0x92, 0x92, 0x92, 0x00, // i
47 0x00, 0x90, 0x90, 0x90, 0x90, 0x90, 0xfe, 0x00, // j
48 `);
```

**img_to_array.py**

A helper to convert an image file to an array that can be used in the example.
#!/usr/bin/env python3

# Converts a grayscale image into a format able to be
# displayed by the SSD1306 driver in horizontal addressing mode

# usage: python3 img_to_array.py <logo.bmp>

# depends on the Pillow library
# `python3 -m pip install --upgrade Pillow`

from PIL import Image
import sys
from pathlib import Path

OLED_HEIGHT = 32
OLED_WIDTH = 128
OLED_PAGE_HEIGHT = 8

if len(sys.argv) < 2:
    print("No image path provided.")
    sys.exit()

img_path = sys.argv[1]

try:
im = Image.open(img_path)
except OSError:
    raise Exception("Oops! The image could not be opened.")

im_width = im.size[0]
im_height = im.size[1]

if im_width > OLED_WIDTH or im_height > OLED_HEIGHT:
    print(f"Your image is f{im_width} pixels wide and {im_height} pixels high, but...")
    raise Exception(f"OLED display only {OLED_WIDTH} pixels wide and {OLED_HEIGHT} pixels high!")

if not (im.mode == "1" or im.mode == "L"):
    raise Exception("Image must be grayscale only")

# black or white
out = im.convert("1")

img_name = Path(im.filename).stem

# 'pixels' is a flattened array with the top left pixel at index 0
# and bottom right pixel at the width*height-1
pixels = list(out.getdata())

# swap white for black and swap (255, 0) for (1, 0)
pixels = [0 if x == 255 else 1 for x in pixels]

# our goal is to divide the image into 8-pixel high pages
# and turn a pixel column into one byte, eg for one page:
# 0 1 0 ....
# 0 1 0
# 0 1 1
# 0 0 1
# 0 1 0
# 0 1 0
# 0 1 1

# Attach an OLED display via I2C

Raspberry Pi Pico C/C++ SDK
Table 29. A list of materials required for the example

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>SSD1306-based OLED display</td>
<td>1</td>
<td>Adafruit part</td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>4</td>
<td>generic part</td>
</tr>
</tbody>
</table>
Attaching a PA1010D Mini GPS module via I2C

This example code shows how to interface the Raspberry Pi Pico to the PA1010D Mini GPS module. This allows you to read basic location and time data from the Recommended Minimum Specific GNSS Sentence (GNRMC protocol) and displays it in a user-friendly format. The datasheet for the module can be found on https://cdn-learn.adafruit.com/assets/assets/000/084/295/original/CD_PA1010D_Datasheet_v.03.pdf?1573833002. The output sentence is read and parsed to split the data fields into a 2D character array, which are then individually printed out. The commands to use different protocols and change settings are found on https://www.sparkfun.com/datasheets/GPS/Modules/PMTK_Protocol.pdf. Additional protocols can be used by editing the init_command array.

NOTE

Each command requires a checksum after the asterisk. The checksum can be calculated for your command using the following website: https://nmeachecksum.eqth.net/.

The GPS needs to be used outdoors in open skies and requires about 15 seconds to acquire a satellite signal in order to display valid data. When the signal is detected, the device will blink a green LED at 1 Hz.

Wiring information

Wiring up the device requires 4 jumpers, to connect VDD, GND, SDA and SCL. The example here uses I2C port 0, which is assigned to GPIO 4 (SDA) and 5 (SCL) in software. Power is supplied from the 3V pin.

![Wiring Diagram for PA1010D](image)

List of Files

CMakeLists.txt

CMake file to incorporate the example into the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/pa1010d_i2c/CMakeLists.txt

```python
1  add_executable(pa1010d_i2c)
2      pa1010d_i2c.c
3  )
4
5  # pull in common dependencies and additional i2c hardware support
6  target_link_libraries(pa1010d_i2c pico_stdlib hardware_i2c)
```
pa1010d_i2c.c

The example code.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/pa1010d_i2c/pa1010d_i2c.c

```c
#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
#include "pico/binary_info.h"
#include "hardware/i2c.h"
#include "string.h"

const int addr = 0x10;
#define MAX_READ 250

#define i2c_default

void pa1010d_write_command(const char command[], int com_length) {
    // Convert character array to bytes for writing
    uint8_t int_command[com_length];
    for (int i = 0; i < com_length; ++i) {
        int_command[i] = command[i];
        i2c_write_blocking(i2c_default, addr, &int_command[i], 1, true);
    }
}

void pa1010d_parse_string(char output[], char protocol[]) {
    // Finds location of protocol message in output
    char *com_index = strstr(output, protocol);
    int p = com_index - output;
    // Splits components of output sentence into array
    #define NO_OF_FIELDS 14
    #define MAX_LEN 15
}
```

Attaching a PA1010D Mini GPS module via I2C
int n = 0;
int m = 0;
char gps_data[NO_OF_FIELDS][MAX_LEN];
memset(gps_data, 0, sizeof(gps_data));
bool complete = false;
while (output[p] != '$' && n < MAX_LEN && complete == false) {
    if (output[p] == 'I' || output[p] == '*') {
        n += 1;
        m = 0;
    } else {
        gps_data[n][m] = output[p];
        // Checks if sentence is complete
        if (m < NO_OF_FIELDS) {
            m++;
        } else {
            complete = true;
        }
    }
    p++;
}
// Displays GNRMC data
// Similarly, additional if statements can be used to add more protocols
if (strcmp(protocol, "GNRMC") == 0) {
    printf("Protocol: %s\n", gps_data[0]);
    printf("UTC Time: %s\n", gps_data[1]);
    printf("Status: %s\n", gps_data[2][0] == 'v' ? "Data valid." : "Data invalid. GPS fix not found.");
    printf("Latitude: %s\n", gps_data[3]);
    printf("N/S indicator: %s\n", gps_data[4]);
    printf("Longitude: %s\n", gps_data[5]);
    printf("E/W indicator: %s\n", gps_data[6]);
    printf("Speed over ground: %s\n", gps_data[7]);
    printf("Course over ground: %s\n", gps_data[8]);
    printf("Date: %s\n", gps_data[9][0], gps_data[9][1], gps_data[9][2],
        gps_data[9][3], gps_data[9][4],
        gps_data[9][5]);
    printf("Magnetic Variation: %s\n", gps_data[10]);
    printf("E/W degree indicator: %s\n", gps_data[11]);
    printf("Mode: %s\n", gps_data[12]);
    printf("Checksum: %c%c\n", gps_data[13][8], gps_data[13][11]);
}
void pa1010d_read_raw(char numcommand[]) {
    uint8_t buffer[MAX_READ];
    int i = 0;
    bool complete = false;
    i2c_read_blocking(i2c_default, addr, buffer, MAX_READ, false);
    // Convert bytes to characters
    while (i < MAX_READ && complete == false) {
        numcommand[i] = buffer[i];
        // Stop converting at end of message
        if (buffer[i] == 10 && buffer[i + 1] == 10) {
            complete = true;
        }
        i++;
    }
}
```c
int main() {
    // Decide which protocols you would like to retrieve data from
    char init_command[] = "$PMTK314,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0*29\r\n"
        ;

    // This example will use I2C0 on the default SDA and SCL pins (4, 5 on a Pico)
    i2c_init(i2c_default, 400 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);

    // Make the I2C pins available to picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));

    printf("Hello, PA1010D! Reading raw data from module...
\n");
    pa1010d_write_command(init_command, sizeof(init_command));

    while (1) {
        // Clear array
        memset(numcommand, 0, MAX_READ);
        // Read and re-format
        pa1010d_read_raw(numcommand);
        pa1010d_parse_string(numcommand, "GNRMC");

        // Wait for data to refresh
        sleep_ms(1000);

        // Clear terminal
        printf("\033[1;3H\033[2J");
    }
} #endif
```

### Bill of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>4</td>
<td>generic part</td>
</tr>
</tbody>
</table>
Attaching a PCF8523 Real Time Clock via I2C

This example code shows how to interface the Raspberry Pi Pico to the PCF8523 Real Time Clock

This example allows you to initialise the current time and date and then displays it every half-second. Additionally it lets you set an alarm for a particular time and date and raises an alert accordingly. More information about the module is available at https://learn.adafruit.com/adafruit-pcf8523-real-time-clock.

Wiring information

Wiring up the device requires 4 jumpers, to connect VDD, GND, SDA and SCL. The example here uses I2C port 0, which is assigned to GPIO 4 (SDA) and 5 (SCL) in software. Power is supplied from the 5V pin.

List of Files

CMakeLists.txt

CMake file to incorporate the example into the examples build tree.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/i2c/pcf8523_i2c/CMakeLists.txt

```
add_executable(pcf8523_i2c
  pcf8523_i2c.c
)

# pull in common dependencies and additional i2c hardware support
target_link_libraries(pcf8523_i2c pico_stdlib hardware_i2c)

# create map/bin/hex file etc.
pico_add_extra_outputs(pcf8523_i2c)

# add url via pico_set_program_url
example_auto_set_url(pcf8523_i2c)
```

pcf8523_i2c.c

The example code.
1 /**
2 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
3 *
4 * SPDX-License-Identifier: BSD-3-Clause
5 */
6 #include <stdio.h>
7 #include <string.h>
8 #include "pico/stdlib.h"
9 #include "pico/binary_info.h"
10 #include "hardware/i2c.h"
11
12 /* Example code to talk to a PCF8520 Real Time Clock module
13
14 Connections on Raspberry Pi Pico board, other boards may vary.
15
16 GPIO PICO_DEFAULT_I2C_SDA_PIN (On Pico this is 4 (physical pin 6)) -> SDA on PCF8520 board
17 GPIO PICO_DEFAULT_I2C_SCK_PIN (On Pico this is 5 (physical pin 7)) -> SCL on PCF8520 board
18 5V (physical pin 40) -> VCC on PCF8520 board
19 GND (physical pin 38)  -> GND on PCF8520 board
20 */
21
22 #ifdef i2c_default
23
24 // By default these devices  are on bus address 0x68
25 static int addr = 0x68;
26
27 static void pcf8520_reset() {
28     // Two byte reset. First byte register, second byte data
29     // There are a load more options to set up the device in different ways that could be
30     // added here
31     uint8_t buf[] = {0x00, 0x58};
32     i2c_write_blocking(i2c_default, addr, buf, 2, false);
33 }
34
35 static void pcf820_write_current_time() {
36     // buf[0] is the register to write to
37     // buf[1] is the value that will be written to the register
38     uint8_t buf[2];
39
40     //Write values for the current time in the array
41     //index 0 -> second: bits 4-6 are responsible for the ten’s digit and bits 0-3 for the
42     //unit’s digit
43     //index 1 -> minute: bits 4-6 are responsible for the ten’s digit and bits 0-3 for the
44     //unit’s digit
45     //index 2 -> hour: bits 4-5 are responsible for the ten’s digit and bits 0-3 for the
46     //unit’s digit
47     //index 3 -> day of the month: bits 4-5 are responsible for the ten’s digit and bits 0-3
48     //for the unit’s digit
49     //index 4 -> day of the week: where Sunday = 0x00, Monday = 0x01, Tuesday... ....Saturday =
50     //index 5 -> month: bit 4 is responsible for the ten’s digit and bits 0-3 for the unit’s
gigit
51     //index 6 -> year: bits 4-7 are responsible for the ten’s digit and bits 0-3 for the unit’s
gigit
52
53     //NOTE: If the value in the year register is a multiple for 4, it will be considered a
54     //leap year and hence will include the 29th of February
55     uint8_t current_val[7] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
56 }
for (int i = 3; i < 10; ++i) {
    buf[0] = 1;
    buf[1] = current_val[i - 3];
    i2c_write_blocking(i2c_default, addr, buf, 2, false);
}

static void pcf8520_read_raw(uint8_t *buffer) {

    // For this particular device, we send the device the register we want to read
    // first, then subsequently read from the device. The register is auto incrementing
    // so we don't need to keep sending the register we want, just the first.

    // Start reading acceleration registers from register 0x3B for 6 bytes
    uint8_t val = 0x03;
    i2c_write_blocking(i2c_default, addr, &val, 1, true); // true to keep master control of bus
    i2c_read_blocking(i2c_default, addr, buffer, 7, false);
}

void pcf8520_set_alarm() {

    // buf[0] is the register to write to
    // buf[1] is the value that will be written to the register

    uint8_t buf[2];

    // Default value of alarm register is 0x80
    // Set bit 8 of values to 0 to activate that particular alarm

    // Index 0 -> minute: bits 4-5 are responsible for the ten's digit and bits 0-3 for the unit's digit
    // Index 1 -> hour: bits 4-6 are responsible for the ten's digit and bits 0-3 for the unit's digit
    // Index 2 -> day of the month: bits 4-5 are responsible for the ten's digit and bits 0-3 for the unit's digit
    // Index 3 -> day of the week: where Sunday = 0x00, Monday = 0x01, Tuesday... ...Saturday = 0x06

    uint8_t alarm_val[4] = {0x01, 0x80, 0x80, 0x80};

    // Write alarm values to registers

    for (int i = 10; i < 14; ++i) {
        buf[0] = (uint8_t) i;
        buf[1] = alarm_val[i - 10];
        i2c_write_blocking(i2c_default, addr, buf, 2, false);
    }
}

void pcf8520_check_alarm() {

    // Check bit 3 of control register 2 for alarm flags

    uint8_t status[1];

    uint8_t val = 0x01;
    i2c_write_blocking(i2c_default, addr, &val, 1, true); // true to keep master control of bus
    i2c_read_blocking(i2c_default, addr, status, 1, false);

    if ((status[0] & 0x08) == 0x08) {
        printf("ALARM RINGING");
    } else {
        printf("Alarm not triggered yet");
    }
}

void pcf8520_convert_time(int conv_time[7], const uint8_t raw_time[7]) {

    // Convert raw data into time

    conv_time[0] = (10 + (int) ((raw_time[0] & 0x7F) >> 4)) + (int) (raw_time[0] & 0x0F));

# Attaching a PCF8523 Real Time Clock via I2C

```c
111  conv_time[1] = (10 * (int) ((raw_time[1] & 0x70) >> 4)) + ((int) (raw_time[1] & 0x0F));
114  conv_time[4] = (int) (raw_time[4] & 0x07);
117 }
118 #endif
119
120 int main() {
121     stdio_init_all();
122 #if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN)
123        #warning i2c/pcf8520_i2c example requires a board with I2C pins
124     puts("Default I2C pins were not defined");
125 #else
126     printf("Hello, PCF8520! Reading raw data from registers...
"n");
127     // This example will use I2C0 on the default SDA and SCL pins (4, 5 on a Pico)
128     i2c_init(i2c_default, 400 * 1000);
129     gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
130     gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
131     gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
132     gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
133     // Make the I2C pins available to picotool
134     bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN,
135             GPIO_FUNC_I2C));
136     pcf8520_reset();
137     pcf8520_write_current_time();
138     pcf8520_set_alarm();
139     pcf8520_check_alarm();
140     uint8_t raw_time[7];
141     int real_time[7];
143     while (1) {
144         pcf8520_read_raw(raw_time);
145         pcf8520_convert_time(real_time, raw_time);
146         printf("Time: %02d : %02d : %02d\n", real_time[2], real_time[1], real_time[0]);
147         printf("Date: %s %02d / %02d / %02d\n", days_of_week[real_time[4]], real_time[3],
148             real_time[5], real_time[6]);
149         pcf8520_check_alarm();
150         sleep_ms(500);
151 #else
152     // Clear terminal
153     printf("\033[1;1H\033[2J");
154 
155 #endif
156 }
157

Bill of Materials

Attaching a PCF8523 Real Time Clock via I2C
Table 31. A list of materials required for the example

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>PCF8523 board</td>
<td>1</td>
<td><a href="https://www.adafruit.com/product/3295">https://www.adafruit.com/product/3295</a></td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>4</td>
<td>generic part</td>
</tr>
</tbody>
</table>

Interfacing 1-Wire devices to the Pico

This example demonstrates how to use 1-Wire devices with the Raspberry Pi Pico (RP2040). 1-Wire is an interface that enables a master to control several slave devices over a simple shared serial bus.

The example provides a 1-Wire library that is used to take readings from a set of connected DS18B20 1-Wire temperature sensors. The results are sent to the default serial terminal connected via USB or UART as configured in the SDK.

The library uses a driver based on the RP2040 PIO state machine to generate accurate bus timings and control the 1-Wire bus via a GPIO pin.

1-Wire® is a registered trademark of Maxim Integrated Products, Inc.

Wiring information

Connect one or more DS18B20 sensors to the Pico as shown in the diagram and table below.

Connect GPIO 15 to 3V3(OUT) with a pull-up resistor of about 4k ohms.

Table 32. Connections table

<table>
<thead>
<tr>
<th>Pico</th>
<th>pin</th>
<th>DS18B20</th>
<th>pin / sensor wire</th>
</tr>
</thead>
<tbody>
<tr>
<td>GND</td>
<td>38 or equivalent</td>
<td>GND</td>
<td>1 / Black</td>
</tr>
<tr>
<td>GPIO 15</td>
<td>20</td>
<td>DQ</td>
<td>2 / Yellow</td>
</tr>
<tr>
<td>3V3(OUT)</td>
<td>36</td>
<td>VDD</td>
<td>3 / Red</td>
</tr>
</tbody>
</table>
Bill of materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>DS18B20</td>
<td>3</td>
<td>chip or wired sensor</td>
</tr>
<tr>
<td>3900 ohm resistor</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>M/M jumper wire</td>
<td>13</td>
<td>generic part</td>
</tr>
</tbody>
</table>

List of files

CMakeLists.txt

CMake file to incorporate the example in the build tree.


```cpp
1  add_executable(pio_onewire)
2  3  target_sources(pio_onewire PRIVATE onewire.c)
4  5  add_subdirectory(onewire_library)
6  7  target_link_libraries(pio_onewire PRIVATE
8     pico_stdlib
9     hardware_pio
10    onewire_library)
11  12  pico_add_extra_outputs(pio_onewire)
13  14  # add url via pico_set_program_url
15  example_auto_set_url(pio_onewire)
```

onewire.c

Source code for the example program.

PicocExamples: [https://github.com/raspberrypi/pico-examples/blob/master/pio/onewire/onewire.c](https://github.com/raspberrypi/pico-examples/blob/master/pio/onewire/onewire.c)

```cpp
1  /**
2  * Copyright (c) 2023 mjcross
3  *
4  * SPDX-License-Identifier: BSD-3-Clause
5  **/
6  
7  #include <stdio.h>
8  #include "pico/stdlib.h"
9  #include "pico/binary_info.h"
10 
11  #include "onewire_library.h" // onewire library functions
12  #include "ow_rom.h" // onewire ROM command codes
13  #include "ds18b20.h" // ds18b20 function codes
14 
15  // Demonstrates the PIO onewire driver by taking readings from a set of
16  // ds18b20 1-wire temperature sensors.
```
```c
int main() {
    stdio_init_all();
    PIO pio = pio0;
    uint gpio = 15;
    uint offset;
    // add the program to the PIO shared address space
    if (pio_can_add_program(pio, 0x10000)) {
        offset = pio_add_program(pio, 0x10000);
    } else {
        puts("could not add the program");
    }
    // claim a state machine and initialise a driver instance
    if (ow_init(&ow, pio, offset, gpio)) {
        // find and display 64-bit device addresses
        int maxdevs = 10;
        uint64_t romcode[maxdevs];
        int num_devs = ow_romsearch(&ow, romcode, maxdevs, OW_SEARCH_ROM);
        printf("Found %d devices\n", num_devs);
        for (int i = 0; i < num_devs; i += 1) {
            printf("\t%016llx\n", i, romcode[i]);
        }
        putchar (\n");
        while (num_devs > 0) {
            // start temperature conversion in parallel on all devices
            // (see ds18b20 datasheet)
            ow_reset(&ow);
            ow_send(&ow, OW_SKIP_ROM);
            ow_send(&ow, DS18B20_CONVERT_T);
            // wait for the conversions to finish
            while (ow_read(&ow) == 0);  
            // read the result from each device
            for (int i = 0; i < num_devs; i += 1) {
                ow_reset (0x1);
                ow_send (0x1, OW_MATCH_ROM);
                for (int b = 0; b < 64; b += 8) {
                    ow_send (0x1, romcode[i] >> b);
                }
                ow_send (0x1, DS18B20_READ_SCRATCHPAD);
                int16_t temp = 0;
                temp = ow_read (0x1) | (ow_read (0x1) << 8);
                printf ("\t%f", i, temp / 16.0);
            }
            putchar (\n");
        } else {
            puts("could not initialise the driver");
        }
    } else {
        puts("could not initialise the driver");
    }
    while (true);
}  
```

ow_rom.h

Header file with generic ROM command codes for 1-Wire devices.

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

```c
// Generic ROM commands for 1-Wire devices

#define OW_READ_ROM         0x33
#define OW_MATCH_ROM        0x55
#define OW_SKIP_ROM         0xCC
#define OW_ALARM_SEARCH     0xEC
#define OW_SEARCH_ROM       0xF0
```

ds18b20.h

Header file with function command codes for the DS18B20 device.

Pico Examples: https://github.com/raspberrypi/pico-examples/blob/master/pio/onewire/ds18b20.h

```c
// Function commands for d18b20 1-Wire temperature sensor

#define DS18B20_CONVERT_T           0x44
#define DS18B20_WRITE_SCRATCHPAD    0x4e
#define DS18B20_READ_SCRATCHPAD     0xbe
#define DS18B20_COPY_SCRATCHPAD     0x48
#define DS18B20_RECALL_EE           0xb8
#define DS18B20_READ_POWER_SUPPLY   0xb4
```

onewire_library/

Subdirectory containing the 1-Wire library and driver.

onewire_library/CMakeLists.txt

CMake file to build the 1-Wire library and driver.


```cmake
add_library(onewire_library INTERFACE)
target_sources(onewire_library INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}/onewire_library.c)

# invoke pio_asm to assemble the state machine programs
#
pico_generate_pio_header(onewire_library ${CMAKE_CURRENT_LIST_DIR}/onewire_library.pio)

target_link_libraries(onewire_library INTERFACE
  pico_stdlib
  hardware_pio
)

# add the `binary` directory so that the generated headers are included in the project
#
target_include_directories(onewire_library INTERFACE
  ${CMAKE_CURRENT_SOURCE_DIR}
  ${CMAKE_CURRENT_BINARY_DIR}
)`
onewire_library/onewire_library.c

Source code for the 1-Wire user functions.

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

```c
/* Copyright (c) 2023 mjcross
 * SPDX-License-Identifier: BSD-3-Clause */

#include "pico/stdlib.h"
#include "hardware/gpio.h"
#include "hardware/pio.h"
#include "onewire_library.h"

// Create a driver instance and populate the provided OW structure.
// Returns: True on success.
// ow: A pointer to a blank OW structure to hold the driver parameters.
// pio: The PIO hardware instance to use.
// offset: The location of the onewire program in the PIO shared address space.
// gpio: The pin to use for the bus (NB: see the README).
bool ow_init (OW *ow, PIO pio, uint offset, uint gpio) {
    int sm = pio_claim_unused_sm (pio, false);
    if (sm == -1) {
        return false;
    }
    gpio_init (gpio);
    pio_gpio_init (pio, gpio);
    ow->gpio = gpio;
    ow->pio = pio;
    ow->offset = offset;
    ow->sm = (uint)sm;
    ow->jmp_reset = onewire_reset_instr (ow->offset); // assemble the bus reset instruction
    onewire_sm_init (ow->pio, ow->sm, ow->offset, ow->gpio, 8); // set 8 bits per word
    return true;
}

// Send a binary word on the bus (LSB first).
// ow: A pointer to an OW driver struct.
// data: The word to be sent.
void ow_send (OW *ow, uint data) {
    pio_sm_put_blocking (ow->pio, ow->sm, (uint32_t)data);
    pio_sm_get_blocking (ow->pio, ow->sm); // discard the response
}

// Read a binary word from the bus.
// Returns: the word read (LSB first).
// ow: pointer to an OW driver struct
// data: The word to be sent.
uint8_t ow_read (OW *ow) {
    pio_sm_put_blocking (ow->pio, ow->sm, 0xff); // generate read slots
    return (uint8_t)(pio_sm_get_blocking (ow->pio, ow->sm) >> 24); // shift response into bits 0..7
}

// Reset the bus and detect any connected slaves.
// Returns: true if any slaves responded.
// ow: pointer to an OW driver struct
```

Interfacing 1-Wire devices to the Pico
bool ow_reset (OW *ow) {
    pio_sm_exec_wait_blocking (ow->pio, ow->sm, ow->jmp_reset);
    if ((pio_sm_get_blocking (ow->pio, ow->sm) & 1) == 0) { // apply pin mask (see pio program)
        return true; // a slave pulled the bus low
    }
    return false;
}

int ow_romsearch (OW *ow, uint64_t *romcodes, int maxdevs, uint command) {
    int index;
    uint64_t romcode = 0ull;
    int branch_point;
    int next_branch_point = -1;
    int num_found = 0;
    bool finished = false;

    onewire_sm_init (ow->pio, ow->sm, ow->offset, ow->gpio, 1); // set driver to 1-bit mode

    while (finished == false && (maxdevs == 0 || num_found < maxdevs)) {
        finished = true;
        branch_point = next_branch_point;
        if (ow_reset (ow) == false) {
            num_found = 0; // no slaves present
            finished = true;
            break;
        }
        for (int i = 0; i < 8; i += 1) { // send search command as single bits
            ow_send (ow, branch_point >> i);
        }
        for (index = 0; index < 64; index += 1) { // determine romcode bits 0..63 (see ref)
            uint a = ow_read (ow);
            uint b = ow_read (ow);
            if (a == 0 && b == 0) { // (a, b) = (0, 0)
                if (index == branch_point) {
                    ow_send (ow, 1);
                    romcode |= (1ull << index);
                } else {
                    if (index > branch_point || (romcode & (1ull << index)) == 0) {
                        ow_send (ow, 0);
                        finished = false;
                        romcode &= ~(1ull << index);
                        next_branch_point = index;
                    } else { // index < branch_point or romcode[index] = 1
                        ow_send (ow, 1);
                    }
                }
            } else if (a != 0 && b != 0) { // (a, b) = (1, 1) error (e.g. device disconnected)
                num_found = -2; // function will return -1
                finished = true;
                break; // terminate for loop
            } else {
                if (a == 0) { // (a, b) = (0, 1) or (1, 0)
                    ow_send (ow, 0);
                    romcode &= ~(1ull << index);
                }
            }
        }
    }
}

// command: 1-Wire search command (e.g. OW_SEARCHROM or OW_ALARM_SEARCH)
// romcodes: location at which store the addresses (NULL means don't store)
// maxdevs: maximum number of devices to find (0 means no limit)
// ow: pointer to an OW driver struct
// Returns: the number of devices found (up to maxdevs) or -1 if an error occurred.
// Find ROM codes (64-bit hardware addresses) of all connected devices.
// a slave pulled the bus low
// no slaves present
// a, b) = (0, 0)
```c
else {
    ow_send (ow, 1);
    romcode |= (ull << index);
}
}
// end of for loop

if (romcodes != NULL) {
    romcodes[num_found] = romcode; // store the romcode
}
num_found += 1; // end of while loop

onewire_sm_init (ow->pio, ow->sm, ow->offset, ow->gpio, 8); // restore 8-bit mode
return num_found;
```

**onewire_library/onewire_library.h**

Header file for the 1-Wire user functions and types.

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

```c
#include "hardware/pio.h"
#include "hardware/clocks.h"  // for clock_get_hz() in generated header
#include "onewire_library.pio.h" // generated by pioasm

typedef struct {
    PIO pio;
    uint sm;
    uint jmp_reset;
    int offset;
    int gpio;
} OW;

bool ow_init (OW *ow, PIO pio, uint offset, uint gpio);
void ow_send (OW *ow, uint data);
uint8_t ow_read (OW *ow);
bool ow_reset (OW *ow);
int ow_romsearch (OW *ow, uint64_t *romcodes, int maxdevs, uint command);
```

**onewire_library/onewire_library.pio**

PIO assembly code for the 1-Wire driver.


```c
; Copyright (c) 2023 mjcross
; SPDX-License-Identifier: BSD-3-Clause
; Implements a Maxim 1-Wire bus with a GPIO pin.
; Place data words to be transmitted in the TX FIFO and read the results from the RX FIFO. To reset the bus execute a jump to 'reset_bus' using the opcode from the provided function.
; At 1us per cycle as initialised below the timings are those recommended by:
; https://www.analog.com/en/technical-articles/1wire-communication-through-software.html
```
Notes:
(1) The code will stall with the bus in a safe state if the FIFOs are empty/full.
(2) The bus must be pulled up with an external pull-up resistor of about 4k.
(3) Do not connect the GPIO pin directly to a bus powered at more than 3.3V.

PROGRAM onewire

PUBLIC reset_bus:
  set x, 28 side 1 [15] ; pull bus low
  loop_a: jmp x-- loop_a side 1 [15] ; 29 x 16
  set x, 8 side 0 [6] ; release bus
  loop_b: jmp x-- loop_b side 0 [6] ; 9 x 7
  mov isr, pins side 0 ; read all pins to ISR (avoids autopush) 1
  push side 0 ; push result manually 1
  set x, 24 side 0 [7] ;
  loop_c: jmp x-- loop_c side 0 [15] ; 25 x 16

PUBLIC fetch_bit:
  out x, 1 side 0 ; shift next bit from OSR (autopull) 1
  jmp !x send_0 side 1 [5] ; pull bus low, branch if sending '0' 6

send_1: ; send a '1' bit
  set x, 2 side 0 [8] ; release bus, wait for slave response 9
  in pins, 1 side 0 [4] ; read bus, shift bit to ISR (autopull) 5
  loop_e: jmp x-- loop_e side 0 [15] ; 3 x 16
  jmp fetch_bit side 0 ;

send_0: ; send a '0' bit
  set x, 2 side 1 [5] ; continue pulling bus low 6
  loop_d: jmp x-- loop_d side 1 [15] ; 3 x 16
  in null, 1 side 0 [8] ; release bus, shift 0 to ISR (autopush) 9

PUBLIC onewire_sm_init (PIO pio, uint sm, uint offset, uint pin_num, uint bits_per_word) {
  // create a new state machine configuration
  pio_sm_config c = onewire_program_get_default_config (offset);

  // Input Shift Register configuration settings
  sm_config_set_in_shift (c,
    true, // shift direction: right
    true, // autopush: enabled
    bits_per_word // autopush threshold
  );

  // Output Shift Register configuration settings
  sm_config_set_out_shift (c,
    true, // shift direction: right
    true, // autopull: enabled
    bits_per_word // autopull threshold
  );

  // configure the input and sideset pin groups to start at `pin_num`
  sm_config_set_in_pins (c, pin_num);
Communicating as master and slave via SPI

This example code shows how to interface two RP2040 microcontrollers to each other using SPI.

Wiring information

<table>
<thead>
<tr>
<th>Function</th>
<th>Master (RP2040)</th>
<th>Slave (RP2040)</th>
<th>Master (Pico)</th>
<th>Slave (Pico)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOSI</td>
<td>D00</td>
<td>DI0</td>
<td>25</td>
<td>21</td>
</tr>
<tr>
<td>SCLK</td>
<td>SCK0</td>
<td>SCK0</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>GND</td>
<td>GND</td>
<td>GND</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>CS</td>
<td>CS0</td>
<td>CS0</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>MISO</td>
<td>D10</td>
<td>D00</td>
<td>21</td>
<td>25</td>
</tr>
</tbody>
</table>

At least one of the boards should be powered, and will share power to the other.

If the master is not connected properly to a slave, the master will report reading all zeroes.

If the slave is not connected properly to a master, it will initialize but never transmit nor receive, because it’s waiting for clock signal from the master.
Outputs

Both master and slave boards will give output to stdio.

With master and slave properly connected, the master should output something like this:

```
SPI master example
SPI master says: The following buffer will be written to MOSI endlessly:
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
```

The slave should output something like this:

```
SPI slave example
SPI slave says: When reading from MOSI, the following buffer will be written to MISO:
ff fe fd fc fb fa f9 f8 f7 f6 f5 f4 f3 f2 f1 f0
ef ee ed ec eb ea e9 e8 e7 e6 e5 e4 e3 e2 e1 e0
df de dd dc db da d9 d8 d7 d6 d5 d4 d3 d2 d1 d0
cf ce cd cc cb ca c9 c8 c7 c6 c5 c4 c3 c2 c1 c0
bf be bd bc bb ba b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
af ae ad ac ab aa a9 a8 a7 a6 a5 a4 a3 a2 a1 a0
5f 5e 5d 5c 5b 5a 59 58 57 56 55 54 53 52 51 50
4f 4e 4d 4c 4b 4a 49 48 47 46 45 44 43 42 41 40
3f 3e 3d 3c 3b 3a 39 38 37 36 35 34 33 32 31 30
2f 2e 2d 2c 2b 2a 29 28 27 26 25 24 23 22 21 20
1f 1e 1d 1c 1b 1a 19 18 17 16 15 14 13 12 11 10
0f 0e 0d 0c 0b 0a 09 08 07 06 05 04 03 02 01 00
```
SPI slave says: read page 0 from the MOSI line:
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff

If you look at the communication with a logic analyzer, you should see this:

![Figure 29. Data capture as seen in Saleae Logic.](image)

List of Files

CMakeLists.txt

CMake file to incorporate the example in to the examples build tree.


1 add_subdirectory(spi_master)
2 add_subdirectory(spi_slave)

spi_master/spi_master.c

The example code for SPI master.


1 // Copyright (c) 2021 Michael Stoops. All rights reserved.
2 // Portions copyright (c) 2021 Raspberry Pi (Trading) Ltd.
3 //
4 // Redistribution and use in source and binary forms, with or without modification, are
5 // permitted provided that the
6 // following conditions are met:
7 //
8 // 1. Redistributions of source code must retain the above copyright notice, this list of

Communicating as master and slave via SPI
conditions and the following
8 //    disclaimer.
9 // 2. Redistributions in binary form must reproduce the above copyright notice, this list of
10 //    conditions and the
11 // 3. Neither the name of the copyright holder nor the names of its contributors may be used to
12 //    endorse or promote
13 //  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS
14 // INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
15 // DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
16 // SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
17 // SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
18 // WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
19 // USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
20 // SPDX-License-Identifier: BSD-3-Clause
21 // Example of an SPI bus master using the PL022 SPI interface
22 #include <stdio.h>
23 #include "pico/stdlib.h"
24 #include "pico/binary_info.h"
25 #include "hardware/spi.h"
26
27 #define BUF_LEN         0x100
28
29 void printbuf(uint8_t buf[], size_t len) {
30     int i;
31     for (i = 0; i < len; ++i) {
32         if (i % 16 == 15)
33             printf("%02x\n", buf[i]);
34         else
35             printf("%02x ", buf[i]);
36     }
37     // append trailing newline if there isn't one
38     if (i % 16)
39         putchar('\n');
40 }
41
42 int main() {
43     // Enable UART so we can print
44     stdio_init_all();
45     #if !defined(spi_default) || !defined(PICO_DEFAULT_SPI_SCK_PIN) ||
46         !defined(PICO_DEFAULT_SPI_TX_PIN) || !defined(PICO_DEFAULT_SPI_RX_PIN) ||
47         !defined(PICO_DEFAULT_SPI_CSN_PIN)
48         #warning spi/spi_master example requires a board with SPI pins
49         puts("Default SPI pins were not defined");
50     #else
51         printf("SPI master example\n");
52         // Enable SPI 0 at 1 MHz and connect to GPIOs
53         spi_init(spi_default, 1000 * 1000);
60  gpio_set_function(PICO_DEFAULT_SPI_RX_PIN, GPIO_FUNC_SPI);
61  gpio_set_function(PICO_DEFAULT_SPI_SCK_PIN, GPIO_FUNC_SPI);
62  gpio_set_function(PICO_DEFAULT_SPI_TX_PIN, GPIO_FUNC_SPI);
63  gpio_set_function(PICO_DEFAULT_SPI_CSN_PIN, GPIO_FUNC_SPI);
64  // Make the SPI pins available to picotool
65  bi_decl(bi_4pins_with_func(PICO_DEFAULT_SPI_RX_PIN, PICO_DEFAULT_SPI_TX_PIN,
66  PICO_DEFAULT_SPI_SCK_PIN, PICO_DEFAULT_SPI_CSN_PIN, GPIO_FUNC_SPI));
67  uint8_t out_buf[BUF_LEN], in_buf[BUF_LEN];
68  // Initialize output buffer
69  for (size_t i = 0; i < BUF_LEN; ++i) {
70      out_buf[i] = i;
71  }
72  printf("SPI master says: The following buffer will be written to MOSI endlessly:\n");
73  printbuf(out_buf, BUF_LEN);
74  for (size_t i = 0; i < BUF_LEN; ++i) {
75      // Write the output buffer to MOSI, and at the same time read from MISO.
76      spi_write_read_blocking(spi_default, out_buf, in_buf, BUF_LEN);
77      // Write to stdio whatever came in on the MISO line.
78      printf("SPI master says: read page %d from the MISO line:\n", i);
79      printbuf(in_buf, BUF_LEN);
80  }
81  // Sleep for ten seconds so you get a chance to read the output.
82  sleep_ms(10 * 1000);
83  #endif
84
85 // Copyright (c) 2021 Michael Stoops. All rights reserved.
86 // Portions copyright (c) 2021 Raspberry Pi (Trading) Ltd.
87 // Redistribution and use in source and binary forms, with or without modification, are
88 // permitted provided that the
89 // following conditions are met:
90 // 1. Redistributions of source code must retain the above copyright notice, this list of
91 // conditions and the following
92 // disclaimer.
93 // 2. Redistributions in binary form must reproduce the above copyright notice, this list of
94 // conditions and the
95 // following disclaimer in the documentation and/or other materials provided with the
96 // distribution.
97 // 3. Neither the name of the copyright holder nor the names of its contributors may be used to
98 // endorse or promote
99 // products derived from this software without specific prior written permission.
100 // THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS
101 // OR IMPLIED WARRANTIES,
102 // INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
103 // PARTICULAR PURPOSE ARE
104 // DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
105 // INDIRECT, INCIDENTAL,
106 // SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

// SPDX-License-Identifier: BSD-3-Clause

// Example of an SPI bus slave using the PL022 SPI interface

#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
#include "pico/binary_info.h"
#include "hardware/spi.h"
#define BUF_LEN 0x100

void printbuf(uint8_t buf[], size_t len) {
    int i;
    for (i = 0; i < len; ++i) {
        if (i % 16 == 15)
            printf("%02x\n", buf[i]);
        else
            printf("%02x ", buf[i]);
    }
    // append trailing newline if there isn’t one
    if (i % 16)
        putchar(‘\n’);
}

int main() {
    // Enable UART so we can print
    stdio_init_all();
    #if !defined(spi_default) || !defined(PICO_DEFAULT_SPI_SCK_PIN) ||
       !defined(PICO_DEFAULT_SPI_TX_PIN) || !defined(PICO_DEFAULT_SPI_RX_PIN) ||
       !defined(PICO_DEFAULT_SPI_CSN_PIN)
        #warning spi/spi_slave example requires a board with SPI pins
        puts("Default SPI pins were not defined");
    #else
        printf("SPI slave example\n");
        // Enable SPI 0 at 1 MHz and connect to GPIOs
        spi_init(spi_default, 1000 * 1000);
        spi_set_slave(spi_default, true);
        gpio_set_function(PICO_DEFAULT_SPI_RX_PIN, GPIO_FUNC_SPI);
        gpio_set_function(PICO_DEFAULT_SPI_SCK_PIN, GPIO_FUNC_SPI);
        gpio_set_function(PICO_DEFAULT_SPI_TX_PIN, GPIO_FUNC_SPI);
        gpio_set_function(PICO_DEFAULT_SPI_CSN_PIN, GPIO_FUNC_SPI);
        // Make the SPI pins available to picotool
        bi_decl(bi_4pins_with_func(PICO_DEFAULT_SPI_RX_PIN, PICO_DEFAULT_SPI_SCK_PIN, PICO_DEFAULT_SPI_CSN_PIN, PICO_DEFAULT_SPI_TX_PIN, GPIO_FUNC_SPI));
        uint8_t out_buf[BUF_LEN], in_buf[BUF_LEN];
        // Initialize output buffer
        for (size_t i = 0; i < BUF_LEN; ++i) {
            // bit-inverted from i. The values should be: {0xff, 0xfe, 0xfd...}
            out_buf[i] = ~i;
        }
        ...
printf("SPI slave says: When reading from MOSI, the following buffer will be written to MISO:\n");
printf(out_buf, BUF_LEN);
for (size_t i = 0; ++i) {
    // Write the output buffer to MISO, and at the same time read from MOSI.
    spi_write_read_blocking(spi_default, out_buf, in_buf, BUF_LEN);
    // Write to stdio whatever came in on the MOSI line.
    printf("SPI slave says: read page \d from the MOSI line:\n", i);
    printf(in_buf, BUF_LEN);
}

#endif

---

### Bill of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>M/M Jumper wires</td>
<td>8</td>
<td>generic part</td>
</tr>
</tbody>
</table>
Appendix B: SDK configuration

SDK configuration is the process of customising the SDK code for your particular build/application. As the parts of the SDK that you use are recompiled as part of your build, configuration options can be chosen at compile time resulting in smaller and more efficient customized versions of the code.

This chapter will show what configuration parameters are available, and how they can be changed. SDK configuration parameters are passed as C preprocessor definitions to the build. The most common way to override them is to specify them in your `CMakeLists.txt` when you define your executable or library:

```
e.g.
add_executable(my_program main.c)
...
target_compile_definitions(my_program PRIVATE
    PICO_STACK_SIZE=4096
)
```

or if you are creating a library, and you want to add compile definitions whenever your library is included:

```
add_library(my_library INTERFACE)
...
target_compile_definitions(my_library INTERFACE
    PICO_STDIO_DEFAULT_CRLF=0
    PICO_DEFAULT_UART=1
)
```

The definitions can also be overridden in header files, as is commonly done for board configuration (see Appendix D).

For example, the Pimoroni Tiny2040 board header configures the following to specify appropriate board settings for the default I2C channel exposed on that board.

```
// --- I2C ---
#ifndef PICO_DEFAULT_I2C
#define PICO_DEFAULT_I2C 1
#endif
#ifndef PICO_DEFAULT_I2C_SDA_PIN
#define PICO_DEFAULT_I2C_SDA_PIN 2
#endif
#ifndef PICO_DEFAULT_I2C_SCL_PIN
#define PICO_DEFAULT_I2C_SCL_PIN 3
#endif
```

**NOTE**

The `#ifdef` allows these values to still be overridden by the build (i.e. in `CMakeLists.txt`).

If you would rather set values in your own header file rather than via `CMake`, then you must make sure the header is included by all compilation (including the SDK sources). Using a custom `PICO_BOARD` header is one way of doing this, but a more advanced way is to have the SDK include your header via `pico/config.h` which itself is included by every SDK source file.
This can be done by adding the following before the `pico_sdk_init()` in your `CMakeLists.txt`:

```
list(APPEND PICO_CONFIG_HEADER_FILES path/to/your/header.h)
```

### Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Defined in</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYW43_TASK_PRIORITY</td>
<td>arch_freertos.h</td>
<td>tskIDLE_PRIORITY + 4</td>
<td>Priority for the CYW43 FreeRTOS task</td>
</tr>
<tr>
<td>CYW43_TASK_STACK_SIZE</td>
<td>arch_freertos.h</td>
<td>1024</td>
<td>Stack size for the CYW43 FreeRTOS task in 4-byte words</td>
</tr>
<tr>
<td>GPIO_IRQ_CALLBACK_ORDER_PRIORITY</td>
<td>gpio.h</td>
<td>PICO_SHARED_IRQ_HANDLER_LOWEST_ORDER_PRIORITY</td>
<td>the irq priority order of the default IRQ callback</td>
</tr>
<tr>
<td>GPIO_RAW_IRQ_HANDLER_DEFAULT_ORDER_PRIORITY</td>
<td>gpio.h</td>
<td>PICO_SHARED_IRQ_HANDLER_DEFAULT_ORDER_PRIORITY</td>
<td>the irq priority order of raw IRQ handlers if the priority is not specified</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_DISABLE_ALL</td>
<td>assert.h</td>
<td>0</td>
<td>Global assert disable</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_ADC</td>
<td>adc.h</td>
<td>0</td>
<td>Enable/disable assertions in the ADC module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_ADDRESS_ALIAS</td>
<td>address_mapped.h</td>
<td>0</td>
<td>Enable/disable assertions in memory address aliasing macros</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_CLOCKS</td>
<td>clocks.h</td>
<td>0</td>
<td>Enable/disable assertions in the clocks module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_CYW43_ARCH</td>
<td>cyw43_arch.h</td>
<td>0</td>
<td>Enable/disable assertions in the pico_cyw43_arch module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_DMA</td>
<td>dma.h</td>
<td>0</td>
<td>Enable/disable DMA assertions</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_EXCEPTION</td>
<td>exception.h</td>
<td>0</td>
<td>Enable/disable assertions in the exception module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_FLASH</td>
<td>flash.h</td>
<td>0</td>
<td>Enable/disable assertions in the flash module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_GPIO</td>
<td>gpio.h</td>
<td>0</td>
<td>Enable/disable assertions in the GPIO module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_I2C</td>
<td>i2c.h</td>
<td>0</td>
<td>Enable/disable assertions in the I2C module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_INTERP</td>
<td>interp.h</td>
<td>0</td>
<td>Enable/disable assertions in the interpolation module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_IRQ</td>
<td>irq.h</td>
<td>0</td>
<td>Enable/disable assertions in the IRQ module</td>
</tr>
<tr>
<td>Parameter name</td>
<td>Defined in</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>-------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_LOC_K_CORE</td>
<td>lock_core.h</td>
<td>0</td>
<td>Enable/disable assertions in the lock core</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_PHEAP</td>
<td>pheap.h</td>
<td>0</td>
<td>Enable/disable assertions in the pheap module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLEDPIO</td>
<td>pio.h</td>
<td>0</td>
<td>Enable/disable assertions in the PIO module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLEDPIO_INSTRUCTIONS</td>
<td>pio_instructions.h</td>
<td>0</td>
<td>Enable/disable assertions in the PIO instructions</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLEDPWM</td>
<td>pwm.h</td>
<td>0</td>
<td>Enable/disable assertions in the PWM module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_SPI</td>
<td>spi.h</td>
<td>0</td>
<td>Enable/disable assertions in the SPI module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_SYNC</td>
<td>sync.h</td>
<td>0</td>
<td>Enable/disable assertions in the HW sync module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_TIME</td>
<td>time.h</td>
<td>0</td>
<td>Enable/disable assertions in the time module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_TIMER</td>
<td>timer.h</td>
<td>0</td>
<td>Enable/disable assertions in the timer module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLED_UART</td>
<td>uart.h</td>
<td>0</td>
<td>Enable/disable assertions in the UART module</td>
</tr>
<tr>
<td>PARAM_ASSERTIONS_ENABLE_ALL</td>
<td>assert.h</td>
<td>0</td>
<td>Global assert enable</td>
</tr>
<tr>
<td>PICO_BOOTSEL_VIA_DOUBLE_RESET_ACTIVITY_LED</td>
<td>pico_bootsel_via_double_reset.c</td>
<td></td>
<td>Optionally define a pin to use as bootloader activity LED when BOOTSEL mode is entered via reset double tap</td>
</tr>
<tr>
<td>PICO_BOOTSEL_VIA_DOUBLE_RESET_INTERFACE_DISABLE_MASK</td>
<td>pico_bootsel_via_double_reset.c</td>
<td>0</td>
<td>Optionally disable either the mass storage interface (bit 0) or the PICOBUILD interface (bit 1) when entering BOOTSEL mode via double reset</td>
</tr>
<tr>
<td>PICO_BOOTSEL_VIA_DOUBLE_RESET_TIMEOUT_MS</td>
<td>pico_bootsel_via_double_reset.c</td>
<td>200</td>
<td>Window of opportunity for a second press of a reset button to enter BOOTSEL mode (milliseconds)</td>
</tr>
<tr>
<td>PICO_BOOT_STAGE2_CHOOSE_AT25SF128A</td>
<td>config.h</td>
<td>0</td>
<td>Select boot2_at25sf128a as the boot stage 2 when no boot stage 2 selection is made by the CMake build</td>
</tr>
<tr>
<td>PICO_BOOT_STAGE2_CHOOSE_GENERIC_03H</td>
<td>config.h</td>
<td>1</td>
<td>Select boot2_generic_03h as the boot stage 2 when no boot stage 2 selection is made by the CMake build</td>
</tr>
<tr>
<td>PICO_BOOT_STAGE2_CHOOSE_IS25LP080</td>
<td>config.h</td>
<td>0</td>
<td>Select boot2_is25p080 as the boot stage 2 when no boot stage 2 selection is made by the CMake build</td>
</tr>
<tr>
<td>PICO_BOOT_STAGE2_CHOOSE_W25Q080</td>
<td>config.h</td>
<td>0</td>
<td>Select boot2_w25q080 as the boot stage 2 when no boot stage 2 selection is made by the CMake build</td>
</tr>
<tr>
<td>Parameter name</td>
<td>Defined in</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PICO_BOOT_STAGE2_CHOOSE_W25X10CL</td>
<td>config.h</td>
<td>0</td>
<td>Select boot2_w25x10cl as the boot stage 2 when no boot stage 2 selection is made by the CMake build</td>
</tr>
<tr>
<td>PICO_BUILD_BOOT_STAGE2_NAME</td>
<td>config.h</td>
<td></td>
<td>The name of the boot stage 2 if selected by the build</td>
</tr>
<tr>
<td>PICO_CLOCK_ADJUST_PERI_CLOCK_WITH_SYS_CLOCK</td>
<td>stdlib.c</td>
<td>0</td>
<td>When the SYS clock PLL is changed keep the peripheral clock attached to it</td>
</tr>
<tr>
<td>PICO_CMSIS_RENAME_EXCEPTIONS</td>
<td>rename_exceptions.h</td>
<td>1</td>
<td>Whether to rename SDK exceptions such as isr_nmi to their CMSIS equivalent i.e. NMI_Handler</td>
</tr>
<tr>
<td>PICO_CONFIG_HEADER</td>
<td>pico.h</td>
<td></td>
<td>unquoted path to header include in place of the default pico/config.h which may be desirable for build systems which can’t easily generate the config_autogen header</td>
</tr>
<tr>
<td>PICO_CONFIG_RRTOS_ADAPTER_HEADER</td>
<td>config.h</td>
<td></td>
<td>unquoted path to header include in the default pico/config.h for RTOS integration defines that must be included in all sources</td>
</tr>
<tr>
<td>PICO_CORE1_STACK_SIZE</td>
<td>multicore.h</td>
<td>PICO_STACK_SIZE (0x800)</td>
<td>Stack size for core 1</td>
</tr>
<tr>
<td>PICO_CYW43_ARCH_DEBUG_ENABLE</td>
<td>cyw43.arch.h</td>
<td>1 in debug builds</td>
<td>Enable/disable some debugging output in the pico_cyw43_arch module</td>
</tr>
<tr>
<td>PICO_CYW43_ARCH_DEFAULT_COUNTRY_CODE</td>
<td>cyw43.arch.h</td>
<td>CYW43_COUNTRY_WORLDWIDE</td>
<td>Default country code for the cyw43 wireless driver</td>
</tr>
<tr>
<td>PICO_DEBUG_MALLOC</td>
<td>malloc.h</td>
<td>0</td>
<td>Enable/disable debug printf from malloc</td>
</tr>
<tr>
<td>PICO_DEBUG_MALLOC_LOW_WATER</td>
<td>malloc.h</td>
<td>0</td>
<td>Define the lower bound for allocation addresses to be printed by PICO_DEBUG_MALLOC</td>
</tr>
<tr>
<td>PICO_DEBUG_PIN_BASE</td>
<td>gpio.h</td>
<td>19</td>
<td>First pin to use for debug output (if enabled)</td>
</tr>
<tr>
<td>PICO_DEBUG_PIN_COUNT</td>
<td>gpio.h</td>
<td>3</td>
<td>Number of pins to use for debug output (if enabled)</td>
</tr>
<tr>
<td>PICO_DEFAULT_I2C</td>
<td>i2c.h</td>
<td></td>
<td>Define the default I2C for a board</td>
</tr>
<tr>
<td>PICO_DEFAULT_I2C_SCL_PIN</td>
<td>i2c.h</td>
<td></td>
<td>Define the default I2C SCL pin</td>
</tr>
<tr>
<td>PICO_DEFAULT_I2C_SDA_PIN</td>
<td>i2c.h</td>
<td></td>
<td>Define the default I2C SDA pin</td>
</tr>
<tr>
<td>PICO_DEFAULT_IRQ_PRIORITY</td>
<td>irq.h</td>
<td>0x80</td>
<td>Define the default IRQ priority</td>
</tr>
<tr>
<td>PICO_DEFAULT_LED_PIN</td>
<td>stdlib.h</td>
<td></td>
<td>Optionally define a pin that drives a regular LED on the board</td>
</tr>
<tr>
<td>PICO_DEFAULT_LED_PIN_INVERTED</td>
<td>stdlib.h</td>
<td>0</td>
<td>1 if LED is inverted or 0 if not</td>
</tr>
<tr>
<td>PICO_DEFAULT_SPI</td>
<td>spi.h</td>
<td></td>
<td>Define the default SPI for a board</td>
</tr>
<tr>
<td>PICO_DEFAULT_SPI_CSNO_PIN</td>
<td>spi.h</td>
<td></td>
<td>Define the default SPI CSN pin</td>
</tr>
<tr>
<td>Parameter name</td>
<td>Defined in</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PICO_DEFAULT_SPI_RX_PIN</td>
<td>spi.h</td>
<td></td>
<td>Define the default SPI RX pin</td>
</tr>
<tr>
<td>PICO_DEFAULT_SPI_SCK_PIN</td>
<td>spi.h</td>
<td></td>
<td>Define the default SPI SCK pin</td>
</tr>
<tr>
<td>PICO_DEFAULT_SPI_TX_PIN</td>
<td>spi.h</td>
<td></td>
<td>Define the default SPI TX pin</td>
</tr>
<tr>
<td>PICO_DEFAULT_UART</td>
<td>uart.h</td>
<td></td>
<td>Define the default UART used for printf etc</td>
</tr>
<tr>
<td>PICO_DEFAULT_UART_BAUD_RATE</td>
<td>uart.h</td>
<td>115200</td>
<td>Define the default UART baudrate</td>
</tr>
<tr>
<td>PICO_DEFAULT_UART_RX_PIN</td>
<td>uart.h</td>
<td></td>
<td>Define the default UART RX pin</td>
</tr>
<tr>
<td>PICO_DEFAULT_UART_TX_PIN</td>
<td>uart.h</td>
<td></td>
<td>Define the default UART TX pin</td>
</tr>
<tr>
<td>PICO_DEFAULT_WS2812_PIN</td>
<td>stdlib.h</td>
<td></td>
<td>Optionally define a pin that controls data to a WS2812 compatible LED on the board</td>
</tr>
<tr>
<td>PICO_DEFAULT_WS2812_POWER_PIN</td>
<td>stdlib.h</td>
<td></td>
<td>Optionally define a pin that controls power to a WS2812 compatible LED on the board</td>
</tr>
<tr>
<td>PICO_DISABLE_SHARED_IRQ_HANDLER</td>
<td>irq.h</td>
<td>0</td>
<td>Disable shared IRQ handlers</td>
</tr>
<tr>
<td>PICO_DOUBLE_SUPPORT_ROM_V1</td>
<td>platform.h</td>
<td>1</td>
<td>Include double support code for RP2040 B0 when that chip revision is supported</td>
</tr>
<tr>
<td>PICO_FLASH_ASSERT_ON_UNSAFE</td>
<td>flash.h</td>
<td>1</td>
<td>Assert in debug mode rather than returning an error if flash_safe_execute cannot guarantee safety to catch bugs early</td>
</tr>
<tr>
<td>PICO_FLASH_ASSUME_CORE0_SAFE</td>
<td>flash.h</td>
<td>0</td>
<td>Assume that core 0 will never be accessing flash and so doesn’t need to be considered during flash_safe_execute</td>
</tr>
<tr>
<td>PICO_FLASH_ASSUME_CORE1_SAFE</td>
<td>flash.h</td>
<td>0</td>
<td>Assume that core 1 will never be accessing flash and so doesn’t need to be considered during flash_safe_execute</td>
</tr>
<tr>
<td>PICO_FLASH_BANK_STORAGE_OFFSET</td>
<td>btstack_flash_bank.h</td>
<td>PICO_FLASH_SIZE_BYTES * PICO_FLASH_BANK_TOTAL_SIZE</td>
<td>Offset in flash of the Bluetooth flash storage</td>
</tr>
<tr>
<td>PICO_FLASH_BANK_TOTAL_SIZE</td>
<td>btstack_flash_bank.h</td>
<td>FLASH_SECTOR_SIZE * 2</td>
<td>Total size of the Bluetooth flash storage. Must be an even multiple of FLASH_SECTOR_SIZE</td>
</tr>
<tr>
<td>PICO_FLASH_SAFE_EXECUTE_PICO_SUPPORT_MULTICORE_LOCKOUT</td>
<td>flash.h</td>
<td>1 when using pico_multicore</td>
<td>Support using multicore_lockout functions to make the other core safe during flash_safe_execute</td>
</tr>
<tr>
<td>PICO_FLASH_SAFE_EXECUTE_SUPPORT_FREERTOS_SMP</td>
<td>flash.h</td>
<td>1 when using FreeRTOS SMP</td>
<td>Support using FreeRTOS SMP to make the other core safe during flash_safe_execute</td>
</tr>
<tr>
<td>PICO_FLASH_SIZE_BYTES</td>
<td>flash.h</td>
<td></td>
<td>size of primary flash in bytes</td>
</tr>
<tr>
<td>Parameter name</td>
<td>Defined in</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PICO_FLOAT_SUPPORT_ROM_V1</td>
<td>platform.h</td>
<td>1</td>
<td>Include float support code for RP2040 B0 when that chip revision is supported</td>
</tr>
<tr>
<td>PICO_HEAP_SIZE</td>
<td>platform.h</td>
<td>0x800</td>
<td>Heap size to reserve</td>
</tr>
<tr>
<td>PICO_MALLOC_PANIC</td>
<td>malloc.h</td>
<td>1</td>
<td>Enable/disable panic when an allocation failure occurs</td>
</tr>
<tr>
<td>PICO_MAX_SHARED_IRQ_HANDLERS</td>
<td>irq.h</td>
<td>4</td>
<td>Maximum number of shared IRQ handlers</td>
</tr>
<tr>
<td>PICO_NO_FPGA_CHECK</td>
<td>platform.h</td>
<td>0</td>
<td>Remove the FPGA platform check for small code size reduction</td>
</tr>
<tr>
<td>PICO_NO_RAM_VECTOR_TABLE</td>
<td>platform.h</td>
<td>0</td>
<td>Enable/disable the RAM vector table</td>
</tr>
<tr>
<td>PICO_PANIC_FUNCTION</td>
<td>runtime.c</td>
<td></td>
<td>Name of a function to use in place of the stock panic function or empty string to simply breakpoint on panic</td>
</tr>
<tr>
<td>PICO_PHEAP_MAX_ENTRIES</td>
<td>pheap.h</td>
<td>255</td>
<td>Maximum number of entries in the pheap</td>
</tr>
<tr>
<td>PICO_PRINTF_ALWAYS_INCLUDED</td>
<td>printf.h</td>
<td>1 in debug build 0 otherwise</td>
<td>Whether to always include printf code even if only called weakly (by panic)</td>
</tr>
<tr>
<td>PICO_PRINTF_DEFAULT_FLOAT_PRECISION</td>
<td>printf.c</td>
<td>6</td>
<td>Define default floating point precision</td>
</tr>
<tr>
<td>PICO_PRINTF_FTOA_BUFFER_SIZE</td>
<td>printf.c</td>
<td>32</td>
<td>Define printf ftoa buffer size</td>
</tr>
<tr>
<td>PICO_PRINTF_MAX_FLOAT</td>
<td>printf.c</td>
<td>1e9</td>
<td>Define the largest float suitable to print with %f</td>
</tr>
<tr>
<td>PICO_PRINTF_NTOA_BUFFER_SIZE</td>
<td>printf.c</td>
<td>32</td>
<td>Define printf ntoa buffer size</td>
</tr>
<tr>
<td>PICO_PRINTF_SUPPORT_EXPONENTIAL</td>
<td>printf.c</td>
<td>1</td>
<td>Enable exponential floating point printing</td>
</tr>
<tr>
<td>PICO_PRINTF_SUPPORT_FLOAT</td>
<td>printf.c</td>
<td>1</td>
<td>Enable floating point printing</td>
</tr>
<tr>
<td>PICO_PRINTF_SUPPORT_LONG_LONG</td>
<td>printf.c</td>
<td>1</td>
<td>Enable support for long long types (%llu or %p)</td>
</tr>
<tr>
<td>PICO_PRINTF_SUPPORT_PTRDIFF_T</td>
<td>printf.c</td>
<td>1</td>
<td>Enable support for the ptrdiff_t type (%t)</td>
</tr>
<tr>
<td>PICO_QUEUE_MAX_LEVEL</td>
<td>queue.h</td>
<td>0</td>
<td>Maintain a field for the highest level that has been reached by a queue</td>
</tr>
<tr>
<td>PICO_RAND_BUS_PERF_COUNTER_EVENT</td>
<td>rand.h</td>
<td></td>
<td>Bus performance counter event to use for sourcing entropy</td>
</tr>
<tr>
<td>PICO_RAND_BUS_PERF_COUNTER_INDEX</td>
<td>rand.h</td>
<td></td>
<td>Bus performance counter index to use for sourcing entropy</td>
</tr>
<tr>
<td>PICO_RAND_ENTROPY_SRC_BUS_PERF_COUNTER</td>
<td>rand.h</td>
<td>1</td>
<td>Enable/disable use of a bus performance counter as an entropy source</td>
</tr>
<tr>
<td>PICO_RAND_ENTROPY_SRC_ROSC</td>
<td>rand.h</td>
<td>1</td>
<td>Enable/disable use of ROSC as an entropy source</td>
</tr>
<tr>
<td>Parameter name</td>
<td>Defined in</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>--------------</td>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PICO_RAND_ENTROPY_SRC_TIME</td>
<td>rand.h</td>
<td>1</td>
<td>Enable/disable use of hardware timestamp as an entropy source</td>
</tr>
<tr>
<td>PICO_RAND_MIN_ROSC_BIT_SAMPLE_TIME_US</td>
<td>rand.h</td>
<td>10</td>
<td>Define a default minimum time between sampling the ROSC random bit</td>
</tr>
<tr>
<td>PICO_RAND_RAM_HASH_END</td>
<td>rand.h</td>
<td>SRAM_END</td>
<td>end of address in RAM (non-inclusive) to hash during pico_rand seed initialization</td>
</tr>
<tr>
<td>PICO_RAND_RAM_HASH_START</td>
<td>rand.h</td>
<td>PICO_RAND_RAM_HASH_END - 1024</td>
<td>start of address in RAM (inclusive) to hash during pico_rand seed initialization</td>
</tr>
<tr>
<td>PICO_RAND_ROSC_BIT_SAMPLE_COUNT</td>
<td>rand.h</td>
<td>1</td>
<td>Number of samples to take of the ROSC random bit per random number generation</td>
</tr>
<tr>
<td>PICO_RAND_SEED_ENTROPY_SRC_BOARD_ID</td>
<td>rand.h</td>
<td>1</td>
<td>Enable/disable use of board id as part of the random seed</td>
</tr>
<tr>
<td>PICO_RAND_SEED_ENTROPY_SRC_RAM_HASH</td>
<td>rand.h</td>
<td>1</td>
<td>Enable/disable use of a RAM hash as an entropy source for the random seed</td>
</tr>
<tr>
<td>PICO_RAND_SEED_ENTROPY_SRC_ROSC</td>
<td>rand.h</td>
<td>1</td>
<td>Enable/disable use of ROSC as an entropy source for the random seed</td>
</tr>
<tr>
<td>PICO_RAND_SEED_ENTROPY_SRC_TIME</td>
<td>rand.h</td>
<td>1</td>
<td>Enable/disable use of hardware timestamp as an entropy source for the random seed</td>
</tr>
<tr>
<td>PICO_RP2040_B0_SUPPORTED</td>
<td>platform.h</td>
<td>1</td>
<td>Whether to include any specific software support for RP2040 B0 revision</td>
</tr>
<tr>
<td>PICO_RP2040_B1_SUPPORTED</td>
<td>platform.h</td>
<td>1</td>
<td>Whether to include any specific software support for RP2040 B1 revision</td>
</tr>
<tr>
<td>PICO_RP2040_B2_SUPPORTED</td>
<td>platform.h</td>
<td>1</td>
<td>Whether to include any specific software support for RP2040 B2 revision</td>
</tr>
<tr>
<td>PICO_SHARED_IRQ_HANDLER_DEFAULT_ORDER_PRIORITY</td>
<td>irq.h</td>
<td>0x80</td>
<td>Set default shared IRQ order priority</td>
</tr>
<tr>
<td>PICO_SPINLOCK_ID_CLAIM_FREE_FIRST</td>
<td>sync.h</td>
<td>24</td>
<td>Lowest Spinlock ID in the 'claim free' range</td>
</tr>
<tr>
<td>PICO_SPINLOCK_ID_CLAIM_FREE_LAST</td>
<td>sync.h</td>
<td>31</td>
<td>Highest Spinlock ID in the 'claim free' range</td>
</tr>
<tr>
<td>PICO_SPINLOCK_ID_HARDWARE_CLAIM</td>
<td>sync.h</td>
<td>11</td>
<td>Spinlock ID for Hardware claim protection</td>
</tr>
<tr>
<td>PICO_SPINLOCK_ID_IRQ</td>
<td>sync.h</td>
<td>9</td>
<td>Spinlock ID for IRQ protection</td>
</tr>
<tr>
<td>PICO_SPINLOCK_ID_OS1</td>
<td>sync.h</td>
<td>14</td>
<td>First Spinlock ID reserved for use by low level OS style software</td>
</tr>
<tr>
<td>Parameter name</td>
<td>Defined in</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>---------------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PICO_SPINLOCK_ID_OS2</td>
<td>sync.h</td>
<td>15</td>
<td>Second Spinlock ID reserved for use by low level OS style software</td>
</tr>
<tr>
<td>PICO_SPINLOCK_ID_RAND</td>
<td>sync.h</td>
<td>12</td>
<td>Spinlock ID for Random Number Generator</td>
</tr>
<tr>
<td>PICO_SPINLOCK_ID_STRIPED_FIRST</td>
<td>sync.h</td>
<td>16</td>
<td>Lowest Spinlock ID in the 'striped' range</td>
</tr>
<tr>
<td>PICO_SPINLOCK_ID_STRIPED_LAST</td>
<td>sync.h</td>
<td>23</td>
<td>Highest Spinlock ID in the 'striped' range</td>
</tr>
<tr>
<td>PICO_SPINLOCK_ID_TIMER</td>
<td>sync.h</td>
<td>10</td>
<td>Spinlock ID for Timer protection</td>
</tr>
<tr>
<td>PICO_STACK_SIZE</td>
<td>platform.h</td>
<td>0x800</td>
<td>Stack Size</td>
</tr>
<tr>
<td>PICO_stdio_DEADLOCK_TIMEOUT_MS</td>
<td>stdio.h</td>
<td>1000</td>
<td>Time after which to assume stdio_usb is deadlocked by use in IRQ and give up</td>
</tr>
<tr>
<td>PICO_stdio_DEFAULT_CRLF</td>
<td>stdio.h</td>
<td>1</td>
<td>Default for CR/LF conversion enabled on all stdio outputs</td>
</tr>
<tr>
<td>PICO_stdio_ENABLE_CRLF_SUPPORT</td>
<td>stdio.h</td>
<td>1</td>
<td>Enable/disable CR/LF output conversion support</td>
</tr>
<tr>
<td>PICO_stdio_SEMIHOSTING_DEFAULT_CRLF</td>
<td>stdio_semihosting.h</td>
<td>PICO_stdio_DEFAULT_CRLF</td>
<td>Default state of CR/LF translation for semihosting output</td>
</tr>
<tr>
<td>PICO_stdio_STACK_BUFFER_SIZE</td>
<td>stdio.h</td>
<td>128</td>
<td>Define printf buffer size (on stack)... this is just a working buffer not a max output size</td>
</tr>
<tr>
<td>PICO_stdio_UART_DEFAULT_CRLF</td>
<td>stdio_uart.h</td>
<td>PICO_stdio_DEFAULT_CRLF</td>
<td>Default state of CR/LF translation for UART output</td>
</tr>
<tr>
<td>PICO_stdio_UART_SUPPORT_CHARS_AVAILABLE_CALLBACK</td>
<td>stdio_uart.h</td>
<td>1</td>
<td>Enable UART STDIO support for stdio_set_chars_available_callback. Can be disabled to make use of the uart elsewhere</td>
</tr>
<tr>
<td>PICO_stdio_USB_CONNECTION_WIOUT_DTR</td>
<td>stdio_usb.h</td>
<td>0</td>
<td>Disable use of DTR for connection checking meaning connection is assumed to be valid</td>
</tr>
<tr>
<td>PICO_stdio_USB_CONNECT_WAIT_TIMEOUT_MS</td>
<td>stdio_usb.h</td>
<td>0</td>
<td>Maximum number of milliseconds to wait during initialization for a CDC connection from the host (negative means indefinite) during initialization</td>
</tr>
<tr>
<td>PICO_stdio_USB_DEFAULT_CRLF</td>
<td>stdio_usb.h</td>
<td>PICO_stdio_DEFAULT_CRLF</td>
<td>Default state of CR/LF translation for USB output</td>
</tr>
<tr>
<td>PICO_stdio_USB_DEVICE_SELFPOWERED</td>
<td>stdio_usb.h</td>
<td>0</td>
<td>Set USB device as self powered device</td>
</tr>
<tr>
<td>PICO_stdio_USB_ENABLE_RESET_VIA_BAUD_RATE</td>
<td>stdio_usb.h</td>
<td>1</td>
<td>Enable/disable resetting into BOOTSEL mode if the host sets the baud rate to a magic value (PICO_stdio_USB_RESET_MAGIC_BAUD_RATE)</td>
</tr>
<tr>
<td>Parameter name</td>
<td>Defined in</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------------</td>
<td>------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PICO_STDIO_USB_ENABLE_RESET_VIA_VENDOR_INTERFACE</td>
<td>stdio_usb.h</td>
<td>1</td>
<td>Enable/disable resetting into BOOTSEL mode via an additional VENDOR USB interface - enables picotool based reset</td>
</tr>
<tr>
<td>PICO_STDIO_USB_LOW_PRIORITY_IRQ</td>
<td>stdio_usb.h</td>
<td></td>
<td>Explicit User IRQ number to claim for tud_task() background execution instead of letting the implementation pick a free one dynamically (deprecated)</td>
</tr>
<tr>
<td>PICO_STDIO_USB_POST_CONNECT_WAIT_DELAY_MS</td>
<td>stdio_usb.h</td>
<td>50</td>
<td>Number of extra milliseconds to wait when using PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS after a host CDC connection is detected (some host terminals seem to sometimes lose transmissions sent right after connection)</td>
</tr>
<tr>
<td>PICO_STDIO_USB_RESET_BOOTSEL_ACTIVITY_LED</td>
<td>stdio_usb.h</td>
<td></td>
<td>Optionally define a pin to use as bootloader activity LED when BOOTSEL mode is entered via USB (either VIA_BAUD_RATE or VIA_VENDOR_INTERFACE)</td>
</tr>
<tr>
<td>PICO_STDIO_USB_RESET_BOOTSEL_FIXED_ACTIVITY_LED</td>
<td>stdio_usb.h</td>
<td>0</td>
<td>Whether the pin specified by PICO_STDIO_USB_RESET_BOOTSEL_ACTIVITY_LED is fixed or can be modified by picotool over the VENDOR USB interface</td>
</tr>
<tr>
<td>PICO_STDIO_USB_RESET_BOOTSEL_INTERFACE_DISABLE_MASK</td>
<td>stdio_usb.h</td>
<td>0</td>
<td>Optionally disable either the mass storage interface (bit 0) or the PICOBOOT interface (bit 1) when entering BOOTSEL mode via USB (either VIA_BAUD_RATE or VIA_VENDOR_INTERFACE)</td>
</tr>
<tr>
<td>PICO_STDIO_USB_RESET_INTERFACE_SUPPORT_RESET_TO_BOOTSEL</td>
<td>stdio_usb.h</td>
<td>1</td>
<td>If vendor reset interface is included allow rebooting to BOOTSEL mode</td>
</tr>
<tr>
<td>PICO_STDIO_USB_RESET_INTERFACE_SUPPORT_RESET_TO_FLASH_BOOT</td>
<td>stdio_usb.h</td>
<td>1</td>
<td>If vendor reset interface is included allow rebooting with regular flash boot</td>
</tr>
<tr>
<td>PICO_STDIO_USB_RESET_MAGIC_BAUD_RATE</td>
<td>stdio_usb.h</td>
<td>1200</td>
<td>baud rate that if selected causes a reset into BOOTSEL mode (if PICO_STDIO_USB_ENABLE_RESET_VIA_BAUD_RATE is set)</td>
</tr>
<tr>
<td>PICO_STDIO_USB_RESET_RESET_TO_FLASH_DELAY_MS</td>
<td>stdio_usb.h</td>
<td>100</td>
<td>delays in ms before rebooting via regular flash boot</td>
</tr>
<tr>
<td>PICO_STDIO_USB_STDOUT_TIMEOUT_US</td>
<td>stdio_usb.h</td>
<td>500000</td>
<td>Number of microseconds to be blocked trying to write USB output before assuming the host has disappeared and discarding data</td>
</tr>
<tr>
<td>Parameter name</td>
<td>Defined in</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>------------------</td>
<td>---------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PICO_STDIO_USB_SUPPORT_CHARS_AVAILABLE_CALLBACK</td>
<td>stdio_usb.h</td>
<td>1</td>
<td>Enable USB STDIO support for stdio_set_chars_available_callback. Can be disabled to make use of USB CDC RX callback elsewhere</td>
</tr>
<tr>
<td>PICO_STDIO_USB_TASK_INTERVAL_US</td>
<td>stdio_usb.h</td>
<td>1000</td>
<td>Period of microseconds between calling tud_task in the background</td>
</tr>
<tr>
<td>PICO_STDOUT_MUTEX</td>
<td>stdio.h</td>
<td>1</td>
<td>Enable/disable mutex around stdout</td>
</tr>
<tr>
<td>PICO_TIME_DEFAULT_ALARM_POOL_DISABLED</td>
<td>time.h</td>
<td>0</td>
<td>Disable the default alarm pool</td>
</tr>
<tr>
<td>PICO_TIME_DEFAULT_ALARM_POOL_HARDWARE_ALARM_NUM</td>
<td>time.h</td>
<td>3</td>
<td>Select which HW alarm is used for the default alarm pool</td>
</tr>
<tr>
<td>PICO_TIME_DEFAULT_ALARM_POOL_MAX_TIMERS</td>
<td>time.h</td>
<td>16</td>
<td>Selects the maximum number of concurrent timers in the default alarm pool</td>
</tr>
<tr>
<td>PICO_TIME_SLEEP_OVERHEAD_ADJUST_US</td>
<td>time.h</td>
<td>6</td>
<td>How many microseconds to wake up early (and then busy_wait) to account for timer overhead when sleeping in low power mode</td>
</tr>
<tr>
<td>PICO_UART_DEFAULT_CRLF</td>
<td>uart.h</td>
<td>0</td>
<td>Enable/disable CR/LF translation on UART</td>
</tr>
<tr>
<td>PICO_UART_ENABLE_CRLF_SUPPORT</td>
<td>uart.h</td>
<td>1</td>
<td>Enable/disable CR/LF translation support</td>
</tr>
<tr>
<td>PICO_USE_MALLOC_MUTEX</td>
<td>malloc.h</td>
<td>1 with pico_multicore, 0 otherwise</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Whether to protect malloc etc with a mutex</td>
</tr>
<tr>
<td>PICO_VTABLE_PER_CORE</td>
<td>irq.h</td>
<td>0</td>
<td>user is using separate vector tables per core</td>
</tr>
<tr>
<td>PICO_XOSC_STARTUP_DELAY_MULTIPLIER</td>
<td>xosc.h</td>
<td>1</td>
<td>Multiplier to lengthen xosc startup delay to accommodate slow-starting oscillators</td>
</tr>
<tr>
<td>PLL_COMMON_REFDIV</td>
<td>clocks.h</td>
<td>1</td>
<td>PLL reference divider setting - used for both PLLs</td>
</tr>
<tr>
<td>PLL_SYS_POSTDIV1</td>
<td>clocks.h</td>
<td>6</td>
<td>System clock PLL post divider 1 setting</td>
</tr>
<tr>
<td>PLL_SYS_POSTDIV2</td>
<td>clocks.h</td>
<td>2</td>
<td>System clock PLL post divider 2 setting</td>
</tr>
<tr>
<td>PLL_SYS_VCO_FREQ_KHZ</td>
<td>clocks.h</td>
<td>1500 * KHZ</td>
<td>System clock PLL frequency</td>
</tr>
<tr>
<td>PLL_USB_POSTDIV1</td>
<td>clocks.h</td>
<td>5</td>
<td>USB clock PLL post divider 1 setting</td>
</tr>
<tr>
<td>PLL_USB_POSTDIV2</td>
<td>clocks.h</td>
<td>5</td>
<td>USB clock PLL post divider 2 setting</td>
</tr>
<tr>
<td>PLL_USB_VCO_FREQ_KHZ</td>
<td>clocks.h</td>
<td>1200 * KHZ</td>
<td>USB clock PLL frequency</td>
</tr>
<tr>
<td>SYS_CLK_KHZ</td>
<td>platform_defs.h</td>
<td>125000</td>
<td>The system operating frequency in kHz</td>
</tr>
<tr>
<td>Parameter name</td>
<td>Defined in</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>USB_CLK_KHZ</td>
<td>platform_defs.h</td>
<td>48000</td>
<td>USB clock frequency. Must be 48MHz for the USB interface to operate correctly</td>
</tr>
<tr>
<td>USB_DPRAM_MAX</td>
<td>usb.h</td>
<td>4096</td>
<td>Set amount of USB RAM used by USB system</td>
</tr>
<tr>
<td>XOSC_KHZ</td>
<td>platform_defs.h</td>
<td>12000</td>
<td>The crystal oscillator frequency in kHz</td>
</tr>
</tbody>
</table>
Appendix C: CMake build configuration

Use CMake cache variables to customize SDK builds.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Defined in</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICO_BARE_METAL</td>
<td>CMakeLists.txt</td>
<td>0</td>
<td>Flag to exclude anything except base headers from the build</td>
</tr>
<tr>
<td>PICO_BOARD</td>
<td>board_setup.cmake</td>
<td>pico</td>
<td>The board name being built for. This is overridable from the user environment</td>
</tr>
<tr>
<td>PICO_BOARD_CMAKE_DIRS</td>
<td>board_setup.cmake</td>
<td></td>
<td>Directories to look for <code>&lt;PICO_BOARD&gt;.cmake</code>. This is overridable from the user environment</td>
</tr>
<tr>
<td>PICO_BOARD_HEADER_DIRS</td>
<td>generic_board.cmake</td>
<td></td>
<td>Directories to look for <code>&lt;PICO_BOARD&gt;.h</code>. This is overridable from the user environment</td>
</tr>
<tr>
<td>PICO_CMAKE_RELOAD_PLATFORM_FILE</td>
<td>pico_pre_load_platform.cmake</td>
<td>none</td>
<td>Custom CMake file to use to set up the platform environment</td>
</tr>
<tr>
<td>PICO_COMPILER</td>
<td>pico_pre_load_toolchain.cmake</td>
<td>none</td>
<td>Optionally specifies a different compiler (other than pico_arm_gcc.cmake) - this is not yet fully supported</td>
</tr>
<tr>
<td>PICO_CONFIG_HEADER_FILES</td>
<td>CMakeLists.txt</td>
<td></td>
<td>List of extra header files to include from pico/config.h for all platforms</td>
</tr>
<tr>
<td>PICO_CONFIG_HOST_HEADER_FILES</td>
<td>CMakeLists.txt</td>
<td></td>
<td>List of extra header files to include from pico/config.h for host platform</td>
</tr>
<tr>
<td>PICO_CONFIG_RP2040_HEADER_FILES</td>
<td>CMakeLists.txt</td>
<td></td>
<td>List of extra header files to include from pico/config.h for rp2040 platform</td>
</tr>
<tr>
<td>PICO_CXX_ENABLE_CXA_ATEXIT</td>
<td>CMakeLists.txt</td>
<td>0</td>
<td>Enabled cxa-atexit</td>
</tr>
<tr>
<td>PICO_CXX_ENABLE_EXCEPTIONS</td>
<td>CMakeLists.txt</td>
<td>0</td>
<td>Enabled CXX exception handling</td>
</tr>
<tr>
<td>PICO_CXX_ENABLE_RTTI</td>
<td>CMakeLists.txt</td>
<td>0</td>
<td>Enabled CXX rtti</td>
</tr>
<tr>
<td>PICO_DEFAULT_BOOT_STAGE2_FILE</td>
<td>CMakeLists.txt</td>
<td></td>
<td>Default stage2 file to use unless overridden by pico_set_boot_stage2 on the TARGET</td>
</tr>
<tr>
<td>PICO_NO_GC_SECTIONS</td>
<td>CMakeLists.txt</td>
<td>0</td>
<td>Disable -ffunction-sections -fdata-sections and --gc-sections</td>
</tr>
<tr>
<td>PICO_NO_HARDWARE</td>
<td>rp2_common.cmake</td>
<td>1 for PICO_PLATFORM host 0 otherwise</td>
<td>OPTION: Whether the build is not targeting an RP2040 device</td>
</tr>
</tbody>
</table>
### Control of binary type produced (advanced)

These variables control the memory layout of produced executables. By default, RP2040 stores code and data entirely in flash storage. At startup, RP2040 copies writeable data and explicitly marked methods into memory.

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Values</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>PICO_DEFAULT_BINARY_TYPE</td>
<td>default</td>
<td>Stores binaries in flash storage. Runs binaries from flash storage.</td>
</tr>
<tr>
<td></td>
<td>no_flash</td>
<td>Stores binaries in memory. Runs binaries from memory. Does not require any flash storage. Note: You must reload this type of binary after every reboot via UF2 file or debugger.</td>
</tr>
<tr>
<td></td>
<td>copy_to_ram</td>
<td>Stores binaries in flash, but copies them to memory (RAM) before executing.</td>
</tr>
<tr>
<td></td>
<td>blocked_ram</td>
<td></td>
</tr>
<tr>
<td>PICO_NO_FLASH</td>
<td>0 / 1</td>
<td>Equivalent to PICO_DEFAULT_BINARY_TYPE=no_flash if =1.</td>
</tr>
<tr>
<td>PICO_COPY_TO_RAM</td>
<td>0 / 1</td>
<td>Equivalent to PICO_DEFAULT_BINARY_TYPE=copy_to_ram if =1.</td>
</tr>
<tr>
<td>PICO_USE_BLOCKED_RAM</td>
<td>0 / 1</td>
<td>Equivalent to PICO_DEFAULT_BINARY_TYPE=blocked_ram if =1.</td>
</tr>
</tbody>
</table>

---

**TIP**

Specify cache variables as arguments when invoking CMake, e.g. `cmake -DPICO_STDIO_UART=0`. 
TIP

You can set the binary type for each executable target (as created by `add_executable`) by calling `pico_set_binary_type(target_type)` using the same type as `PICO_DEFAULT_BINARY_TYPE`. 
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 with some predefined configurations for boards produced by Raspberry Pi and other manufacturers, the main (and default) example being the Raspberry Pi Pico.

Configurations specify a number of parameters that could vary between hardware designs. For example, default UART ports, on-board LED locations and flash capacities etc.

This chapter will go through where these configurations files are, how to make changes and set parameters, and how to build your SDK using CMake with your customisations.

The Configuration files

Board specific configuration files are stored in the SDK source tree, at …/src/boards/include/boards/<boardname>.h. The default configuration file is that for the Raspberry Pi Pico, and at the time of writing is:

<sdk_path>/src/boards/include/boards/pico.h

This relatively short file contains overrides from default of a small number of parameters used by the SDK when building code.

SDK: https://github.com/raspberrypi/pico-sdk/blob/master/src/boards/include/boards/pico.h

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

// -----------------------------------------------------
// NOTE: THIS HEADER IS ALSO INCLUDED BY ASSEMBLER SO
// SHOULD ONLY CONSIST OF PREPROCESSOR DIRECTIVES
// -----------------------------------------------------

// This header may be included by other board headers as "boards/pico.h"

#ifndef _BOARDS_PICO_H
#define _BOARDS_PICO_H

// For board detection
#define RASPBERRYPI_PICO

// --- UART ---
#ifndef PICO_DEFAULT_UART
#define PICO_DEFAULT_UART 0
#endif
#ifndef PICO_DEFAULT_UART_TX_PIN
#define PICO_DEFAULT_UART_TX_PIN 0
#endif
#ifndef PICO_DEFAULT_UART_RX_PIN
#define PICO_DEFAULT_UART_RX_PIN 1
#endif
#endif
```
// --- LED ---
#ifndef PICO_DEFAULT_LED_PIN
#define PICO_DEFAULT_LED_PIN 25
#endif

// --- I2C ---
#ifndef PICO_DEFAULT_I2C
#define PICO_DEFAULT_I2C 0
#endif
#ifndef PICO_DEFAULT_I2C_SDA_PIN
#define PICO_DEFAULT_I2C_SDA_PIN 4
#endif
#ifndef PICO_DEFAULT_I2C_SCL_PIN
#define PICO_DEFAULT_I2C_SCL_PIN 5
#endif

// --- SPI ---
#ifndef PICO_DEFAULT_SPI
#define PICO_DEFAULT_SPI 0
#endif
#ifndef PICO_DEFAULT_SPI_SCK_PIN
#define PICO_DEFAULT_SPI_SCK_PIN 18
#endif
#ifndef PICO_DEFAULT_SPI_TX_PIN
#define PICO_DEFAULT_SPI_TX_PIN 19
#endif
#ifndef PICO_DEFAULT_SPI_RX_PIN
#define PICO_DEFAULT_SPI_RX_PIN 16
#endif
#ifndef PICO_DEFAULT_SPI_CSN_PIN
#define PICO_DEFAULT_SPI_CSN_PIN 17
#endif

// --- FLASH ---
#define PICO_BOOT_STAGE2_CHOOSE_W25Q080 1
#ifndef PICO_FLASH_SPI_CLKDIV
#define PICO_FLASH_SPI_CLKDIV 2
#endif
#ifndef PICO_FLASH_SIZE_BYTES
#define PICO_FLASH_SIZE_BYTES (2 * 1024 * 1024)
#endif

// Drive high to force power supply into PWM mode (lower ripple on 3V3 at light loads)
#define PICO_SMPS_MODE_PIN 23

#ifndef PICO_RP2040_B0_SUPPORTED
#define PICO_RP2040_B0_SUPPORTED 1
#endif

// The GPIO Pin used to read VBUS to determine if the device is battery powered.
#ifndef PICO_VBUS_PIN
#define PICO_VBUS_PIN 24
#endif

// The GPIO Pin used to monitor VSYS. Typically you would use this with ADC.
#ifndef PICO_VSYS_PIN
#define PICO_VSYS_PIN 29
#endif
As can be seen, it sets up the default UART to `uart0`, the GPIO pins to be used for that UART, the GPIO pin used for the on-board LED, and the flash size.

To create your own configuration file, create a file in the board `../source/folder` with the name of your board, for example, `myboard.h`. Enter your board specific parameters in this file.

### 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.

```bash
$ cd pico-examples
$ mkdir myboard_build
$ cd myboard_build
```

then run cmake as follows:

```bash
$ 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 35 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 be viewed online, but is also part of the SDK itself and can be built directly from the command line. If you haven't already checked out the SDK repository you should do so,

```bash
$ cd ~/
$ mkdir pico
$ cd pico
$ git clone https://github.com/raspberrypi/pico-sdk.git --branch master
$ cd pico-sdk
$ git submodule update --init
$ cd ..
$ git clone https://github.com/raspberrypi/pico-examples.git --branch master
```

Install doxygen if you don't already have it,

```bash
$ sudo apt install doxygen
```

Then afterwards you can go ahead and build the documentation,

```bash
$ cd pico-sdk
$ mkdir build
$ cd build
$ cmake -DPICO_EXAMPLES_PATH=../../pico-examples ..
$ make docs
```

The API documentation will be built and can be found in the `pico-sdk/build/docs/doxygen/html` directory, see Figure 30.
Appendix F: SDK release history

Release 1.0.0 (20/Jan/2021)
Initial release

Release 1.0.1 (01/Feb/2021)

- add `pico_get_unique_id` method to return a unique identifier for a Pico board using the identifier of the external flash
- exposed all 4 pacing timers on the DMA peripheral (previously only 2 were exposed)
- fixed ninja build (i.e. `cmake -G ninja .. ; ninja`)
- minor other improvements and bug fixes

Boot Stage 2

Additionally, a low level change was made to the way flash binaries start executing after `boot_stage2`. This was at the request of folks implementing other language runtimes. It is not generally of concern to end users, however it did require a change to the linker scripts so if you have cloned those to make modifications then you need to port across the relevant changes. If you are porting a different language runtime using the SDK bootstage2 implementations then you should be now have a vector table (rather than executable code) - at `0x10000100`.

Release 1.1.0 (05/Mar/2021)

- Added board headers for Adafruit, Pimoroni & SparkFun boards
  - new values for `PICO_BOARD` are `adafruit_feather_rp2040`, `adafruit_itsybitsy_rp2040`, `adafruit_qtpy_rp2040`, `pimoroni_keybow28040`, `pimoroni_picosystem`, `pimoroni_tiny28040`, `sparkfun_micromod`, `sparkfun_promicro`, `sparkfun_thingplus`, in addition to the existing `pico` and `vgaboard`
  - Added additional definitions for a default SPI, I2C pins as well as the existing ones for UART
  - Allow `default` pins to be undefined (not all boards have UART for example), and SDK will compile but warn as needed in the absence of default.
  - Added additional definition for a default WS2812 compatible pin (currently unused).
- New reset options
  - Added `pico_bootsel_via_double_reset` library to allow reset to `BOOTSEL` mode via double press of a `RESET` button
  - When using `pico_stdio_usb` i.e. `stdio` connected via USB CDC to host, setting baud rate to 1200 (by default) can optionally be used to reset into `BOOTSEL` mode.
  - When using `pico_stdio_usb` i.e. `stdio` connected via USB CDC to host, an additional interface may be added to give `picotool` control over resetting the device.
- Build improvement for non-SDK or existing library builds
  - Removed additional compiler warnings (register headers now use `_u(x)` macro for unsigned values though).
  - Made build more clang friendly.
This release also contains many bug fixes, documentation updates and minor improvements.

**Backwards incompatibility**

There are some nominally backwards incompatible changes not worthy of a major version bump:

- `PICO_DEFAULT_UART_` defines now default to undefined if there is no default rather than \(-1\) previously
- The broken `multicore_sleep_core1()` API has been removed; `multicore_reset_core1` is already available to put core 1 into a deep sleep.

**Release 1.1.1 (01/Apr/2021)**

This fixes a number of bugs, and additionally adds support for a board configuration header to choose the `boot_stage2`

**Release 1.1.2 (07/Apr/2021)**

Fixes issues with `boot_stage2` selection

**Release 1.2.0 (03/Jun/2021)**

This release contains numerous bug fixes and documentation improvements. Additionally it contains the following improvements/notable changes:

⚠️ **CAUTION**

The `lib/tinyusb` submodule has been updated from 0.8.0 and now tracks upstream [https://github.com/hathach/tinyusb.git](https://github.com/hathach/tinyusb.git). It is worth making sure you do a

```
$ git submodule sync
$ git submodule update
```

to make sure you are correctly tracking upstream TinyUSB if you are not checking out a clean `pico-sdk` repository.

Moving from TinyUSB 0.8.0 to TinyUSB 0.10.1 may require some minor changes to your USB code.

**New/improved Board headers**

- New board headers support for `PICO_BOARDs` `arduinonano_rp240_connect`, `pimoroni_picolipo_4mb` and `pimoroni_picolipo_16mb`
- Missing/new `#defines` for default SPI and I2C pins have been added

**Updated TinyUSB to 0.10.1**

The `lib/tinyusb` submodule has been updated from 0.8.0 and now tracks upstream [https://github.com/hathach/tinyusb.git](https://github.com/hathach/tinyusb.git)
Added CMSIS core headers

CMSIS core headers (e.g. `core_cm0plus.h` and `RP2040.h`) are made available via `cmsis_core` INTERFACE library. Additionally, CMSIS standard exception naming is available via `PICO_CMSIS_RENAME_EXCEPTIONS=1`.

API improvements

**pico_sync**
- Added support for recursive mutexes via `recursive_mutex_init()` and `auto_init_recursive_mutex()`
- Added `mutex_enter_timeout_us()`
- Added `critical_section_deinit()`
- Added `sem_acquire_timeout_ms()` and `sem_acquire_block_until()`

**hardware_adc**
- Added `adc_get_selected_input()`

**hardware_clocks**
- `clock_get_hz()` now returns actual achieved frequency rather than desired frequency

**hardware_dma**
- Added `dma_channel_is_claimed()`
- Added new methods for configuring/acknowledging DMA IRQs. `dma_irqn_set_channel_enabled()`, `dma_irqn_set_channel_mask_enabled()`, `dma_irqn_get_channel_status()`, `dma_irqn_ acknowledge_channel()` etc.

**hardware_exception**

New library for setting ARM exception handlers:
- Added `exception_set_exclusive_handler()`, `exception_restore_handler()`, `exception_get_vtable_handler()`

**hardware_flash**
- Exposed previously private function `flash_do_cmd()` for low level flash command execution

**hardware_gpio**
- Added `gpio_set_input_hysteresis_enabled()`, `gpio_is_input_hysteresis_enabled()`, `gpio_set_slew_rate()`, `gpio_get_slew_rate()`, `gpio_set_drive_strength()`, `gpio_get_drive_strength()`, `gpio_get_out_level()`, `gpio_set_irqover()`

**hardware_i2c**
- Corrected a number of incorrect hardware register definitions
- A number of edge cases in the i2c code fixed
hardware_interp

- Added `interp_lane_is_claimed()`, `interp_unclaim_lane_mask()`

hardware_irq

- Notably fixed the `PICO_LOWEST/HIGHEST_IRQ_PRIORITY` values which were backwards!

hardware_pio

- Added new methods for configuring/acknowledging PIO interrupts (`pio_set_irqn_source_enabled()`, `pio_set_irqn_source_mask_enabled()`, `pio_interrupt_get()`, `pio_interrupt_clear()` etc.)
- Added `pio_sm_is_claimed()`

hardware_spi

- Added `spi_get_baudrate()`
- Changed `spi_init()` to return the set/achieved baud rate rather than void
- Changed `spi_is_writable()` to return bool not size_t (it was always 1/0)

hardware_sync

- Notable documentation improvements for spin lock functions
- Added `spin_lock_is_claimed()`

hardware_timer

- Added `busy_wait_ms()` to match `busy_wait_us()`
- Added `hardware_alarm_is_claimed()`

pico_float/pico_double

- Correctly save/restore divider state if floating point is used from interrupts

pico_int64_ops

- Added `PICO_INT64_OPS_IN_RAM` flag to move code into RAM to avoid veneers when calling code is in RAM

pico_runtime

- Added ability to override panic function by setting `PICO_PANIC_FUNCTION=foo` to then use `foo` as the implementation, or setting `PICO_PANIC_FUNCTION=` to simply breakpoint, saving some code space

pico_unique_id

- Added `pico_get_unique_board_id_string()`.
General code improvements

- Removed additional classes of compiler warnings
- Added some missing `const` to method parameters

SVD

- USB DPRAM for device mode is now included

pioasm

- Added `#pragma once` to C/C++ output

RTOS interoperability

Improvements designed to make porting RTOSes either based on the SDK or supporting SDK code easier.

- Added `PICO_DIVIDER_DISABLE_INTERRUPTS` flag to optionally configure all uses of the hardware divider to be guarded by disabling interrupts, rather than requiring on the RTOS to save/restore the divider state on context switch
- Added new abstractions to `pico/lock_core.h` to allow an RTOS to inject replacement code for SDK based low level wait, notify and sleep/timeouts used by synchronization primitives in `pico_sync` and for `sleep_` methods. If an RTOS implements these few simple methods, then all SDK semaphore, mutex, queue, sleep methods can be safely used both within/to/from RTOS tasks, but also to communicate with non-RTOS task aware code, whether it be existing libraries and IRQ handlers or code running perhaps (though not necessarily) on the other core

CMake build changes

Substantive changes have been made to the CMake build, so if you are using a hand crafted non-CMake build, you will need to update your compile/link flags. Additionally changed some possibly confusing status messages from CMake build generation to be debug only

Boot Stage 2

- New boot stage 2 for AT25SF128A

Release 1.3.0 (02/Nov/2021)

This release contains numerous bug fixes and documentation improvements. Additionally, it contains the following notable changes/improvements:

Updated TinyUSB to 0.12.0

- The `lib/tinyusb` submodule has been updated from 0.10.1 to 0.12.0. See https://github.com/hathach/tinyusb/releases/tag/0.11.0 and https://github.com/hathach/tinyusb/releases/tag/0.12.0 for release notes.
- Improvements have been made for projects that include TinyUSB and also compile with enhanced warning levels and `-Werror`. Warnings have been fixed in rp2040 specific TinyUSB code, and in TinyUSB headers, and a new cmake function `suppress_tinyusb_warnings()` has been added, that you may call from your `CMakeLists.txt` to suppress warnings in other TinyUSB C files.
New Board Support

The following boards have been added and may be specified via `PICO_BOARD`:

- `adafruit_trinkey_qt2040`
- `melopero_shake_rp2040`
- `pimoroni_interstate75`
- `pimoroni_plasma2040`
- `pybstick26_rp2040`
- `waveshare_rp2040_lcd_0.96`
- `waveshare_rp2040_plus_4mb`
- `waveshare_rp2040_plus_16mb`
- `waveshare_rp2040_zero`

Updated SVD, `hardware_regs`, `hardware_structs`

The RP2040 SVD has been updated, fixing some register access types and adding new documentation.

The `hardware_regs` headers have been updated accordingly.

The `hardware_structs` headers which were previously hand coded, are now generated from the SVD, and retain select documentation from the SVD, including register descriptions and register bit-field tables.

e.g. what was once

```c
typedef struct {
  io_rw_32 ctrl;
  io_ro_32 fstat;
  ...
}
```

becomes:

```c
//
// The _REG_ macro is intended to help make the register navigable in your IDE (for example, using the "Go to Definition" feature)
// _REG_(x) will link to the corresponding register in hardware/regs/pio.h.
//
// Bit-field descriptions are of the form:
// BITMASK [BITRANGE]: FIELDNAME (RESETVALUE): DESCRIPTION

typedef struct {
  _REG_(PIO_CTRL_OFFSET) // PIO_CTRL
  // PIO control register
  // 0x00000f00 [11:8]  : CLKDIV_RESTART (0): Restart a state machine's clock divider from an initial phase of 0
  // 0x000000f0 [7:4]   : SM_RESTART (0): Write 1 to instantly clear internal SM state which may be otherwise difficult...
  // 0x0000000f [3:0]   : SM_ENABLE (0): Enable/disable each of the four state machines by writing 1/0 to each of these four bits
  io_rw_32 ctrl;
  ...
  _REG_(PIO_FSTAT_OFFSET) // PIO_FSTAT
}
```
Behavioural Changes

There were some behavioural changes in this release:

**pico_sync**

SDK 1.2.0 previously added recursive mutex support using the existing (previously non-recursive) `mutex_` functions. This caused a performance regression, and the only clean way to fix the problem was to return the `mutex_` functions to their pre-SDK 1.2.0 behaviour, and split the recursive mutex functionality out into separate `recursive_mutex_` functions with a separate `recursive_mutex_` type.

Code using the SDK 1.2.0 recursive mutex functionality will need to be changed to use the new type and functions, however as a convenience, the pre-processor define `PICO_MUTEX_ENABLE_SDK120_COMPATIBILITY` may be set to 1 to retain the SDK 1.2.0 behaviour at the cost of an additional performance penalty. The ability to use this pre-processor define will be removed in a subsequent SDK version.

**pico_platform**

- `pico.h` and its dependencies have been slightly refactored so it can be included by assembler code as well as C/C code. This ensures that assembler code and C/C code follow the same board configuration/override order and see the same configuration defines. This should not break any existing code, but is notable enough to mention.
- `pico/platform.h` is now fully documented.

**pico_standard_link**

`-Wl,max-page-size=4096` is now passed to the linker, which is beneficial to certain users and should have no discernible impact on the rest.

Other Notable Improvements

**hardware_base**

- Added `xip_noalloc_alias(addr), xip_nocache_alias(addr), xip_nocache_noalloc_alias(addr)` macros for converting a flash address between XIP aliases (similar to the `hw_xxx_alias(addr)` macros).

**hardware_dma**

- Added `dma_timer_claim(), dma_timer_unclaim(), dma_claim_unused_timer()` and `dma_timer_is_claimed()` to manage ownership of DMA timers.
- Added `dma_timer_set_fraction()` and `dma_get_timer_dreq()` to facilitate pacing DMA transfers using DMA timers.
hardware_i2c

- Added `i2c_get_dreq()` function to facilitate configuring DMA transfers to/from an I2C instance.

hardware_irq

- Added `irq_get_priority()`.
- Fixed implementation when `PICO_DISABLE_SHARED_IRQ_HANDLERS=1` is specified, and allowed `irq_add_shared_handler` to be used in this case (as long as there is only one handler - i.e. it behaves exactly like `irq_set_exclusive_handler`).
- Sped up IRQ priority initialization which was slowing down per core initialization.

hardware_pio

- `pio_encode_` functions in `hardware/pico_instructions.h` are now documented.

hardware_pwm

- Added `pwm_get_dreq()` function to facilitate configuring DMA transfers to a PWM slice.

hardware_spi

- Added `spi_get_dreq()` function to facilitate configuring DMA transfers to/from an SPI instance.

hardware_uart

- Added `uart_get_dreq()` function to facilitate configuring DMA transfers to/from a UART instance.

hardware_watchdog

- Added `watchdog_enable_caused_reboot()` to distinguish a watchdog reboot caused by a watchdog timeout after calling `watchdog_enable()` from other watchdog reboots (e.g. that are performed when a UF2 is dragged onto a device in BOOTSEL mode).

pico_bootrom

- Added new constants and function signature typedefs to `pico/bootrom.h` to facilitate calling bootrom functions directly.

pico_multicore

- Improved documentation in `pico/multicore.h`; particularly, `multicore_lockout_` functions are newly documented.

pico_platform

- `PICO_RP2040` is now defined to 1 in `PICO_PLATFORM=rp2040` (i.e. normal) builds.

pico_stdio

- Added `puts_raw()` and `putchar_raw()` to skip CR/LF translation if enabled.
• Added stdio_usb_connected() to detect CDC connection when using stdio_usb.

• Added PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS define that can be set to wait for a CDC connection to be established during initialization of stdio_usb. Note: value -1 means indefinite. This can be used to prevent initial program output being lost, at the cost of requiring an active CDC connection.

• Fixed semihosting_putchar which was completely broken.

pico_usb_reset_interface

• This new library contains pico/usb_reset_interface.h split out from stdio_usb to facilitate inclusion in external projects.

CMake build

• OUTPUT_NAME target property is now respected when generating supplemental files (.BIN, .HEX, .MAP, .UF2)

pioasm

• Operator precedence of *, /, -, + have been fixed

• Incorrect MicroPython output has been fixed.

elf2uf2

• A bug causing an error with binaries produced by certain other languages has been fixed.

Release 1.3.1 (18/May/2022)

This release contains numerous bug fixes and documentation improvements which are not all listed here; you can see the full list of individual commits here.

New Board Support

The following boards have been added and may be specified via PICO_BOARD:

• adafruit_kb2040
• adafruit_macropad_rp2040
• eetree_gamekit_rp2040
• garatronic_pybstick26_rp2040 (renamed from pybstick26_rp2040)
• pimoroni_badger2040
• pimoroni_motor2040
• pimoroni_servo2040
• pimoroni_tiny2040_2mb
• seeed_xiao_rp2040
• solderparty_rp2040_stamp_carrier
• solderparty_rp2040_stamp
Notable Library Changes/Improvements

**hardware_dma**
- New documentation has been added to the `dma_channel_abort()` function describing errata `RP2040-E13`, and how to work around it.

**hardware_irq**
- Fixed a bug related to removing and then re-adding shared IRQ handlers. It is now possible to add/remove handlers as documented.
- Added new documentation clarifying the fact the shared IRQ handler ordering “priorities” have values that increase with higher priority vs. Cortex M0+ IRQ priorities which have values that decrease with priority!

**hardware_pwm**
- Added a `pwm_config_set_clkdiv_int_frac()` method to complement `pwm_config_set_clkdiv_int()` and `pwm_config_set_clkdiv()`.

**hardware_pio**
- Fixed the `pio_set_irqn_source_mask_enabled()` method which previously affected the wrong IRQ.

**hardware_rtc**
- Added clarification to `rtc_set_datetime()` documentation that the new value may not be visible to a `rtc_get_datetime()` very soon after, due to crossing of clock domains.

**pico_platform**
- Added a `busy_wait_at_least_cycles()` method as a convenience method for a short tight-loop counter-based delay.

**pico_stdio**
- Fixed a bug related to removing stdio “drivers”. `stdio_set_driver_enabled()` can now be used freely to dynamically enable and disable drivers during runtime.

**pico_time**
- Added an `is_at_the_end_of_time()` method to check if a given time matches the SDK’s maximum time value.

**Runtime**
A bug in `__ctzdi2()` aka `__builtin_ctz(uint64_t)` was fixed.
Build

- Compilation with GCC 11 is now supported.
- `PIOASM_EXTRA_SOURCE_FILES` is now actually respected.

pioasm

- Input files with Windows (CRLF) line endings are now accepted.
- A bug in the python output was fixed.

elf2uf2

- Extra padding was added to the UF2 output of misaligned or non-contiguous binaries to work around errata RP2040-E14.

ℹ️ NOTE

The 1.3.0 release of the SDK incorrectly squashed the history of the changes. A new merge commit has been added to restore the full history, and the 1.3.0 tag has been updated.

Release 1.4.0 (30/Jun/2022)

This release adds wireless support for the Raspberry Pi Pico W, adds support for other new boards, and contains various bug fixes, documentation improvements, and minor improvements/added functionality. You can see the full list of individual commits here.

New Board Support

The following boards have been added and may be specified via `PICO_BOARD`:

- `pico_w`
- `datanoisetv_rp2040_dsp`
- `solderparty_rp2040_stamp_round_carrier`

Wireless Support

- Support for the Raspberry Pi Pico W is now included with the SDK (`PICO_BOARD=pico_w`). The Pico W uses a driver for the wireless chip called `cyw43_driver` which is included as a submodule of the SDK. You need to initialize this submodule for Pico W wireless support to be available. Note that the LED on the Pico W board is only accessible via the wireless chip, and can be accessed via `cyw43_arch_gpio_put()` and `cyw43_arch_gpio_get()` (part of the `pico_cyw43_arch` library described below). As a result of the LED being on the wireless chip, there is no `PICO_DEFAULT_LED_PIN` setting and the default LED based examples in `pico-examples` do not work with the Pico W.
- IP support is provided by `lwIP` which is also included as a submodule which you should initialize if you want to use it.

The following libraries exposing lwIP functionality are provided by the SDK:

- `pico_lwip_core` (included in `pico_lwip`)
- `pico_lwip_core4` (included in `pico_lwip`)
- pico_lwip_core6 (included in pico_lwip)
- pico_lwip_netif (included in pico_lwip)
- pico_lwip_sixlowpan (included in pico_lwip)
- pico_lwip_ppp (included in pico_lwip)
- pico_lwip_api (this is a blocking API that may be used with FreeRTOS and is not included in pico_lwip)

As referenced above, the SDK provides a pico_lwip which aggregates all of the commonly needed lwIP functionality. You are of course free to use the substituent libraries explicitly instead.

The following libraries are provided that contain the equivalent lwIP application support:
- pico_lwip_snmp
- pico_lwip_http
- pico_lwip_makesfsdata
- pico_lwip_iperf
- pico_lwip_sctp
- pico_lwip_sctp
- pico_lwip_mDNS
- pico_lwip_netbios
- pico_lwip_tftp
- pico_lwip_mbedtls

- Integration of the IP stack and the cyw43_driver network driver into the user’s code is handled by pico_cyw43_arch. Both the IP stack and the driver need to do work in response to network traffic, and pico_cyw43_arch provides a variety of strategies for servicing that work. Four architecture variants are currently provided as libraries:
  - pico_cyw43_arch_lwip_poll - For using the RAW lwIP API (NO_SYS=1 mode) with polling. With this architecture the user code must periodically poll via cyw43_arch_poll() to perform background work. This architecture matches the common use of lwIP on microcontrollers, and provides no multicore safety
  - pico_cyw43_arch_lwip_threadsafe_background - For using the RAW lwIP API (NO_SYS=1 mode) with multicore safety, and automatic servicing of the cyw43_driver and lwIP in the background. User polling is not required with this architecture, but care should be taken as lwIP callbacks happen in an IRQ context.
  - pico_cyw43_arch_lwip_sys_freertos - For using the full lwIP API including blocking sockets in OS mode (NO_SYS=0), along with multicore/task safety, and automatic servicing of the cyw43_driver and the lwIP stack in a separate task. This powerful architecture works with both SMP and non-SMP variants of the RP2040 port of FreeRTOS-Kernel. Note you must set FREERTOS_KERNEL_PATH in your build to use this variant.
  - pico_cyw43_arch_none - If you do not need the TCP/IP stack but wish to use the on-board LED or other wireless chip connected GPIOs.

See the library documentation or the pico/cyw43_arch.h header for more details.

**Notable Library Changes/Improvements**

**hardware_dma**

- Added dma_unclaim_mask() function for un-claiming multiple DMA channels at once.
- Added channel_config_set_high_priority() function to set the channel priority via a channel config object.
hardware_gpio

- Improved the documentation for the pre-existing gpio IRQ functions which use the "one callback per core" callback mechanism, and added a `gpio_set_irq_callback()` function to explicitly set the callback independently of enabling per pin GPIO IRQs.
- Reduced the latency of calling the existing "one callback per core" GPIO IRQ callback.
- Added new support for the user to add their own shared GPIO IRQ handler independent of the pre-existing "one callback per core" callback mechanism, allowing for independent usage of GPIO IRQs without having to share one handler.

See the documentation in `hardware/irq.h` for full details of the functions added:
  - `gpio_add_raw_irq_handler()`
  - `gpio_add_raw_irq_handler_masked()`
  - `gpio_add_raw_irq_handler_with_order_priority()`
  - `gpio_add_raw_irq_handler_with_order_priority_masked()`
  - `gpio_remove_raw_irq_handler()`
  - `gpio_remove_raw_irq_handler_masked()`
- Added a `gpio_get_irq_event_mask()` utility function for use by the new 'raw' IRQ handlers.

hardware_irq

- Added `user_irq_claim()`, `user_irq_unclaim()`, `user_irq_claim_unused()` and `user_irq_is_claimed()` functions for claiming ownership of the user IRQs (the ones numbered 26-31 and not connected to any hardware). Uses of the user IRQs have been updated to use these functions. For `stdio_usb`, the `PICO_STDIO_USB_LOW_PRIORITY_IRQ` define is still respected if specified, but otherwise an unclaimed one is chosen.
- Added an `irq_is_shared_handler()` function to determine if a particular IRQ uses a shared handler.

pico_sync

- Added a `sem_try_acquire()` function, for non-blocking acquisition of a semaphore.

pico_stdio

- `stderr` is now supported and goes to the same destination as `stdout`.
- Zero timeouts for `getchar_timeout_us()` are now correctly honored (previously they were a 1μs minimum).

stdio_usb

- The use of a 1ms timer to handle background TinyUSB work has been replaced with use of a more interrupt driven approach using a user IRQ for better performance. Note this new feature is disabled if shared IRQ handlers are disabled via `PICO_DISABLE_SHARED_IRQ_HANDLERS=1`.

Miscellaneous

- `get_core_num()` has been moved to `pico/platform.h` from `hardware/sync.h`.
- The C library function `realloc()` is now multicore safe too.
- The minimum PLL frequency has been increased from 400Mhz to 750Mhz to improve stability across operating
conditions. This should not affect the majority of users in any way, but may impact those trying to set particularly low clock frequencies. If you do wish to return to the previous minimum, you can set PICO_PLL_VCO_MIN_FREQ_MHZ back to 400. There is also a new PICO_PLL_VCO_MAX_FREQ_MHZ which defaults to 1600.

Build

• Compilation with GCC 12 is now supported.

Release 1.5.0 (11/Feb/2023)

This release contains new libraries and functionality, along with numerous bug fixes and documentation improvements. Highlights are listed below, or you can see the full list of individual commits here, and the full list of resolved issues here.

New Board Support

The following boards have been added and may be specified via PICO_BOARD:

• nullbits_bit_c_pro
• waveshare_rp2040_lcd_1.28
• waveshare_rp2040_one

Library Changes/Improvements

hardware_clocks

• clock_gpio_init() now takes a float for the clock divider value, rather than an int.

• Added clock_gpio_init_int_frac() function to allow initialization of integer and fractional part of the clock divider value, without using float.

• Added --ref-min option to vcocalc.py to override the minimum reference frequency allowed.

• vcocalc.py now additionally considers reference frequency dividers greater than 1.

hardware_divider

• Improved the performance of hw_divider functions.

hardware_dma

• Added dma_sniffer_set_output_invert_enabled() and dma_sniffer_set_output_reverse_enabled() functions to configure the DMA sniffer.

• Added dma_sniffer_set_data_accumulator() and dma_sniffer_get_data_accumulator() functions to access the DMA sniffer accumulator.
hardware_i2c

- Added `i2c_get_instance()` function for consistency with other hardware libraries.
- Added `i2c_read_byte_raw()`, `i2c_write_byte_raw()` functions to directly read and write the I2C data register for an I2C instance.

hardware_timer

- Added `hardware_alarm_claim_unused()` function to claim an unused hardware timer.

pico_cyw43_arch

- Added `cyw43_arch_wifi_connect_bssid_variants` of `cyw43_arch_wifi_connect_` functions to allow connection to a specific access point.
- Blocking `cyw43_arch_wifi_connect_` functions now continue trying to connect rather than failing immediately if the network is not found.
- `cyw43_arch_wifi_connect_` functions now return consistent return codes (PICO_OK, or PICO_ERROR_XXX).
- The `pico_cyw43_arch` library has been completely rewritten on top of the new `pico_async_context` library that generically abstracts the different types of asynchronous operation (poll, threadsafe_background and freertos) previously handled in a bespoke fashion by `pico_cyw43_arch`. Many edge case bugs have been fixed as a result of this. Note that this change should be entirely backwards compatible from the user point of view.
- `cyw43_arch_init()` and `cyw43_arch_deinit()` functions are now very thin layers which handle async_context life-cycles, along with adding support for the `cyw43_driver`, lwIP, BTstack etc. to that async_context. Currently, these mechanisms remain the preferred documented way to initialize Pico W networking, however you are free to do similar initialization/de-initialization yourself.
- Added `cyw43_arch_set_async_context()` function to specify a custom async_context prior to calling `cyw43_arch_init*()`.
- Added `cyw43_arch_async_context()` function to get the async_context used by the CYW43 architecture support.
- Added `cyw43_arch_init_default_async_context()` function to return the async_context that `cyw43_arch_init*()` would initialize if one has not been set by the user.
- Added `cyw43_arch_wait_for_work_until()` function to block until there is networking work to be done. This is most useful for poll style applications that have no other work to do and wish to sleep until `cyw43_arch_poll()` needs to be called again.

pico_cyw43_driver

- The functionality has been clarified into 3 separate libraries:
  - `cyw43_driver` - the raw `cyw43_driver` code.
  - `cyw43_driver_picow` - additional support for communication with the Wi-Fi chip over SPI on Pico W.
  - `pico_cyw43_driver` - integration of the `cyw43_driver` with the `pico-sdk` via async_context
- Added `CYW43_WIFI_NVRAM_INCLUDE_FILE` define to allow user to override the NVRAM file.

pico_divider

- Improved the performance of 64-bit divider functions.
pico_platform

- Add panic_compact() function that discards the message to save space in non-debug (NEBUG defined) builds.

pico_runtime

- Added proper implementation of certain missing newlib system APIs: _gettimeofday(), _times(), _isatty(), _getpid().
- The above changes enable certain additional C/C++ library functionality such as gettimeofday(), times() and std::chrono.
- Added gettimeofday() implementation such that gettimeofday() can be meaningfully used.
- Added default (return -1) implementations of the remaining newlib system APIs: _open(), _close(), _lseek(), _fstat(), _isatty(), _kill(), to prevent warnings on GCC 12.
- Made all newlib system API implementations weak so the user can override them.

pico_stdio

- pico_stdio allows for outputting from within an IRQ handler that creates the potential for deadlocks (especially with pico_stdio_usb), and the intention is to not deadlock but instead discard output in any cases where a deadlock would otherwise occur. The code has been revamped to avoid more deadlock cases, and a new define PICO_STDIO_DEADLOCK_TIMEOUT_MS has been added to catch remaining cases that might be caused by user level locking.
- Addedstdio_set_chars_available_callback() function to set a callback to be called when input is available. See also the new PICO_STDIO_USB_SUPPORT_CHARS_AVAILABLE_CALLBACK and PICO_STDIO_UART_SUPPORT_CHARS_AVAILABLE_CALLBACK defines which both default to 1 and control the availability of this new feature for USB and UART stdio respectively (at the cost of a little more code).
- Improved performance of stdio_semihosting.
- Give the user more control over the USB descriptors of stdio_usb via USBD_VID, USBD_PID, USBD_PRODUCT, PICO_STDIO_USB_CONNECTION_WITHOUT_DTR and PICO_STDIO_USB_DEVICE_SELFPOWERED

pico_sync

- Added critical_section_is_initialized() function to test if a critical section has been initialized.
- Added mutex_try_enter_block_until() function to wait only up to a certain time to acquire a mutex.

pico_time

- Added from_us_since_boot() function to convert a uint64_t timestamp to an absolute_time_t.
- Added absolute_time_min() function to return the earlier of two absolute_time_t values.
- Added alarm_pool_create_with_unused_hardware_alarm() function to create an alarm pool using a hardware alarm number claimed using hardware_alarm_claim().
- Added alarm_pool_core_num() function to determine what core an alarm pool runs on.
- Added alarm_pool_add_alarm_at_force_in_context() function to add an alarm, and have it always run in the IRQ context even if the target time is in the past, or during the call. This may be simpler in some cases than dealing with the fire_if_past parameters to existing functions, and avoids some callbacks happening from non-IRQ context.
pico_lwip
- Added pico_lwip_mqtt library to expose the MQTT app functionality in lwIP.
- Added pico_lwip_mdns library to expose the MDNS app functionality in lwIP.
- Added pico_lwip_freertos library for NO_SYS=0 with FreeRTOS as a complement to pico_lwip_nosys for NO_SYS=1.

TinyUSB
- TinyUSB has upgraded from 0.12.0 to 0.15.0. See TinyUSB release notes here for details.
- Particularly host support should be massively improved.
- Defaulted new TinyUSB dcd_rp2040 driver’s TUD_OPT_RP2040_USB_DEVICE_UFRAME_FIX variable to 1 as a workaround for errata RP2040-E15. This fix is required for correctness, but comes at the cost of some performance, so applications that won't ever be plugged into a Pi 4 or Pi 400 can optionally disable this by setting the value of TUD_OPT_RP2040_USB_DEVICE_UFRAME_FIX to 0 either via target_compile_definitions in their CMakeLists.txt or in their tusb_config.h.

New Libraries

pico_async_context
- Provides support for asynchronous events (timers/IRQ notifications) to be handled in a safe context without concurrent execution (as required by many asynchronous 3rd party libraries).
- Provides implementations matching those previously implemented in pico_cyw43_arch:
  - poll - Not thread-safe; the user must call async_context_poll() periodically from their main loop, but can call async_context_wait_for_work_until() to block until work is required.
  - threadsafe_background - No polling is required; instead asynchronous work is performed in a low priority IRQ. Locking is provided such that IRQ/non-IRQ or multiple cores can interact safely.
  - freertos - Asynchronous work is performed in a separate FreeRTOS task.
- async_context guarantees all callbacks happen on a single core.
- async_context supports multiple instances for providing independent context which can execute concurrently with respect to each other.

pico_i2c_slave
- A (slightly modified) pico_i2c_slave library from https://github.com/vmilea/pico_i2c_slave
- Adds a callback style event API for handling I2C slave requests.

pico_mbedtls
- Added pico_mbedtls library to provide MBed TLS support. You can depend on both pico_lwip_mbedtls and pico_mbedtls to use MBed TLS and lwIP together. See the tls_client example in pico-examples for more details.

pico_rand
- Implements a new Random Number Generator API.
• **pico_rand** generates random numbers at runtime utilizing a number of possible entropy sources, and uses those sources to modify the state of a 128-bit 'Pseudo Random Number Generator' implemented in software.

• Adds **get_rand_32()**, **get_rand_64()** and **get_rand_128()** functions to return largely unpredictable random numbers (which should be different on each board/run for example).

**Miscellaneous**

• Added a new header `hardware/structs/nvic.h` with a struct for the Arm Cortex M0+ NVIC available via the `nvic_hu` pointer.

• Added new `PICO_CXX_DISABLE_ALLOCATION_OVERRIDES` which can be set to 1 if you do not want `pico_standard_link` to include non-exceptional overrides of `std::new`, `std::new[]`, `std::delete` and `std::delete[]` when exceptions are disabled.

• `elf2uf2` now correctly uses LMA instead of VMA of the entry point to determine binary type (flash/RAM). This is required to support some exotic binaries correctly.

**Build**

• The build will now check for a functional compiler via the standard **CMake** mechanism.

• The build will pick up pre-installed `elf2uf2` and `pioasm` if found via an installed `pico-sdk-tools CMake` package. If it can do so, then no native compiler is required for the build!

• It is now possible to switch the board type `PICO_BOARD` in an existing **CMake** build directory.

• `ARCHIVE_OUTPUT_DIRECTORY` is now respected in build for **UF2** output files.

• Spaces are now supported in the path to the `pico-sdk`

• All libraries `xxx` in the `pico-sdk` now support a `xxx_headers` variant that just pulls in the libraries' headers. These `xxx_headers` libraries correctly mirror the dependencies of the `xxx` libraries, so you can use `xxx_headers` instead of `xxx` as your dependency if you do not want to pull in any implementation files (perhaps if you are making a STATIC library). Actually the "all" is not quite true, non-code libraries such as `pico_standard_link` and `pico_cxx_options` are an exception.

**Bluetooth Support for Pico W (BETA)**

The support is currently available as a beta. More details will be forthcoming with the actual release. In the meantime, there are examples in `pico-examples`.

**Key changes:**

• The Bluetooth API is provided by **BTstack**.

• The following new libraries are provided that expose core BTstack functionality:
  
  o **pico_btstack_BLE** - Adds Bluetooth Low Energy (LE) support.

  o **pico_btstack_classic** - Adds Bluetooth Classic support.

  o **pico_btstack_sbc_encoder** - Adds Bluetooth Sub Band Coding (SBC) encoder support.

  o **pico_btstack_sbc_decoder** - Adds Bluetooth Sub Band Coding (SBC) decoder support.

  o **pico_btstack_bnep_lwip** - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP.

  o **pico_btstack_bnep_lwip_sys_freertos** - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP with FreeRTOS for NO_SYS=0.

• The following integration libraries are also provided:
o `pico_btstack_run_loop_async_context` - provides a common `async_context` backed implementation of a BTstack "run loop" that can be used for all BTstack use with the `pico-sdk`.

- Provides a sample implementation for storing required Bluetooth state in flash.

- Integrates BTstack with the CYW43 driver.

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

- Updated `pico_cyw43_driver` and `cyw43_driver` to support HCI communication for Bluetooth.

- Updated `cyw43_driver_picow` to support Pico W specific HCI communication for Bluetooth over SPI.

- Updated `cyw43_arch_init()` and `cyw43_arch_deinit()` to additionally handle Bluetooth support if `CYW43_ENABLE_BLUETOOTH` is 1 (as it will be automatically if you depend on `pico_btstack_cyw43`).

---

**Release 1.5.1 (14/Jun/2023)**

This release is largely a bug fix release, however it also makes Bluetooth support official and adds some new libraries and functionality.

Highlights are listed below, or you can see the full list of individual commits [here](#), and the full list of resolved issues [here](#).

**Board Support**

The following board has been added and may be specified via `PICO_BOARD`:

- `pololu_3pi_2040_robot`

The following board configurations have been modified:

- `adafruit_itsybitsy_rp2040` - corrected the mismatched `PICO_DEFAULT_I2C` bus number (favors the breadboard pins not the stemma connector).

- `sparkfun_thingplus` - added WS2812 pin config.

**Library Changes/Improvements**

**hardware_dma**

- Added `dma_channel_cleanup()` function that can be used to clean up a dynamically claimed DMA channel after use, such that it won’t be in a surprising state for the next user, making sure that any in-flight transfer is aborted, and no interrupts are left pending.

**hardware_spi**

- The `spi_set_format`, `spi_set_slave`, `spi_set_baudrate` functions that modify the configuration of an SPI instance, now disable the SPI while changing the configuration as specified in the data sheet.

**pico_async_context**

- Added `user_data` member to `async_when_pending_worker_t` to match `async_at_time_worker_t`.
**pico_cyw43_arch**

- Added cyw43_arch_disable_sta_mode() function to complement cyw43_arch_enable_sta_mode().
- Added cyw43_arch_disable_ap_mode() function to complement cyw43_arch_enable_ap_mode().

**pico_stdio_usb**

- The 20-character limit for descriptor strings USBD_PRODUCT and USBD_MANUFACTURER can now be extended by defining USBD_DESC_STR_MAX.
- PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS is now supported in the build as well as compiler definitions; if it is set in the build, it is added to the compile definitions.

**pico_rand**

- Fixed poor randomness when PICO_RAND_ENTROPY_SRC_PERF_COUNTER=1.

**PLL and Clocks**

- The set_sys_clock_pll and set_sys_clock_khz methods now reference a pre-processor define PICO_CLOCK_AJDJUST_PERI_CLOCK_WITH_SYS_CLOCK. If set to 1, the peripheral clock is updated to match the new system clock, otherwise the preexisting behavior (of setting the peripheral clock to a safe 48Mhz) is preserved.
- Support for non-standard crystal frequencies, and compile-time custom clock configurations:
  - The new define XOSC_KHZ is used in preference to the preexisting XOSC_MHZ to define the crystal oscillator frequency. This value is now also correctly plumbed through the various clock setup functions, such that they behave correctly with a crystal frequency other than 12Mhz. XOSC_MHZ will be automatically defined for backwards compatibility if XOSC_KHZ is an exact multiple of 1000 Khz. Note that either XOSC_MHZ or XOSC_KHZ may be specified by the user, but not both.
  - The new define PLL_COMMON_REFDIV can be specified to override the default reference divider of 1.
  - The new defines PLL_SYS_VCO_FREQ_KHZ, PLL_SYS_POSTDIV1 and PLL_SYS_POSTDIV2 are used to configure the system clock PLL during runtime initialization. These are defaulted for you if SYS_CLK_KHZ=125000, XOSC_KHZ=12000 and PLL_COMMON_REFDIV=1. You can modify these values in your CMakeLists.txt if you want to configure a different system clock during runtime initialization, or are using a non-standard crystal.
  - The new defines PLL_USB_VCO_FREQ_KHZ, PLL_USB_POSTDIV1 and PLL_USB_POSTDIV2 are used to configure the USB clock PLL during runtime initialization. These are defaulted for you if USB_CLK_KHZ=48000, XOSC_KHZ=12000 and PLL_COMMON_REFDIV=1. You can modify these values in your CMakeLists.txt if you want to configure a different USB clock if you are using a non-standard crystal.
  - The new define PICO_PLL_VCO_MIN_FREQ_KHZ is used in preference to the pre-existing PICO_PLL_VCO_MIN_FREQ_MHZ, though specifying either is supported.
  - The new define PICO_PLL_VCO_MAX_FREQ_KHZ is used in preference to the pre-existing PICO_PLL_VCO_MAX_FREQ_MHZ, though specifying either is supported.

**New Libraries**

**pico_flash**

- This is a new higher level library than hardware_flash. It provides helper functions to facilitate getting into a state where it is safe to write to flash (the default implementation disables interrupts on the current core, and if necessary, makes sure the other core is running from RAM, and has interrupts disabled).
• Adds a `flash_safe_execute()` function to execute a callback function while in the ‘safe’ state.

• Adds a `flash_safe_execute_core_init()` function which must be called from the ‘other core’ when using `pico_multicore` to enable the cooperative support for entering a ‘safe’ state.

• Supports user override of the mechanism by overriding the `get_flash_safety_helper()` function.

**Miscellaneous**

• All assembly (including inline) in the SDK now uses the unified syntax.
  
  ◦ New C macros `pico_default_asm( ... )` and `pico_default_asm_volatile( ... )` are provided that are equivalent to `asm` and `asm volatile` blocks, but with a `.syntax unified` at the beginning.

• A new assembler macro `pico_default_asm_setup` is provided to configure the correct CPU and dialect.

• Some code cleanup to make the SDK code at least compile cleanly on Clang and IAR.

**Build**

• `PICO_BOARD` and `PICO_BOARD_HEADER_DIRS` now correctly use the latest environment variable value if present.

• A CMake performance regression due to repeated calls to `find_package` has been fixed.

• Experimental support is provided for compiling with Clang. As an example, you can build with the LLVM Embedded Toolchain for Arm, noting however that currently only version 14.0.0 works, as later versions use `picolib` rather than `newlib`.
  
  ◦ Note that if you are using TinyUSB you need to use the latest master to compile with Clang.

```
$ mkdir clang_build
$ cd clang_build
$ cmake -DPICO_COMPILER=pico_arm_clang -DPICO_TOOLCHAIN_PATH=/path/to/arm-embedded-llvm
        -14.0.0 ..
$ make
```

**Bluetooth Support for Pico W**

The support is now official. Please find examples in `pico-examples`.

• The Bluetooth API is provided by `BTstack`.

• The following libraries are provided that expose core BTstack functionality:
  
  ◦ `pico_btstack_ble` - Adds Bluetooth Low Energy (LE) support.
  ◦ `pico_btstack_classic` - Adds Bluetooth Classic support.
  ◦ `pico_btstack_sbc_encoder` - Adds Bluetooth Sub Band Coding (SBC) encoder support.
  ◦ `pico_btstack_sbc_decoder` - Adds Bluetooth Sub Band Coding (SBC) decoder support.
  ◦ `pico_btstack_bnep_lwip` - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP.
  ◦ `pico_btstack_bnep_lwip_sys_freertos` - Adds Bluetooth Network Encapsulation Protocol (BNEP) support using LwIP with FreeRTOS for `NO_SYS=0`.

• The following integration libraries are also provided:
  
  ◦ `pico_btstack_run_loop_async_context` - provides a common `async_context` backed implementation of a BTstack "run loop" that can be used for all BTstack use with the `pico-sdk`. 
- **pico_btstack_flash_bank** - provides a sample implementation for storing required Bluetooth state in flash.
- **pico_btstack_cyw43** - integrates BTstack with the CYW43 driver.

- The CMake function `pico_btstack_make_gatt_header` can be used to run the BTstack compile_gatt tool to make a GATT header file from a BTstack GATT file.
- `pico_cyw43_driver` and `cyw43_driver` now support HCI communication for Bluetooth.
- `cyw43_driver_picow` now supports Pico W specific HCI communication for Bluetooth over SPI.
- `cyw43_arch_init()` and `cyw43_arch_deinit()` automatically handle Bluetooth support if `CYW43_ENABLE_BLUETOOTH` is 1 (as it will be automatically if you depend on `pico_btstack_cyw43`).

**Key changes since 1.5.0:**

- Added Raspberry Pi specific BTstack license.
- The storage offset in flash for `pico_btstack_flash_bank` can be specified at runtime by defining `pico_flash_bank_get_storage_offset_func` to your own function to return the offset within flash.
- `pico_btstack_flash_bank` is now safe for multicore / FreeRTOS SMP use, as it uses the new `pico_flash` library to make sure the other core is not accessing flash during flash updates. If you are using `pico_multicore` you must have called `flash_safe_execute_core_init` from the "other" core (to the one Bluetooth is running on).
- Automatically set Bluetooth MAC address to the correct MAC address (Wi-Fi MAC address + 1), as some devices do not have it set in OTP and were using the same default MAC from the Bluetooth chip causing collisions.
- Various bug fixes and stability improvements (especially with concurrent Wi-Fi), including updating `cyw43_driver` and `btstack` to the newest versions.
## Appendix G: Documentation release history

<table>
<thead>
<tr>
<th>Release</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>21 Jan 2021</td>
<td>• Initial release</td>
</tr>
<tr>
<td>1.1</td>
<td>26 Jan 2021</td>
<td>• Minor corrections&lt;br&gt;• Extra information about using DMA with ADC&lt;br&gt;• Clarified M0+ and SIO CPUID registers&lt;br&gt;• Added more discussion of Timers&lt;br&gt;• Update Windows and macOS build instructions&lt;br&gt;• Renamed books and optimised size of output PDFs</td>
</tr>
<tr>
<td>1.2</td>
<td>01 Feb 2021</td>
<td>• Minor corrections&lt;br&gt;• Small improvements to PIO documentation&lt;br&gt;• Added missing TIMER2 and TIMER3 registers to DMA&lt;br&gt;• Explained how to get MicroPython REPL on UART&lt;br&gt;• To accompany the V1.0.1 release of the C SDK</td>
</tr>
<tr>
<td>1.3</td>
<td>23 Feb 2021</td>
<td>• Minor corrections&lt;br&gt;• Changed font&lt;br&gt;• Additional documentation on sink/source limits for RP2040&lt;br&gt;• Major improvements to SWD documentation&lt;br&gt;• Updated MicroPython build instructions&lt;br&gt;• MicroPython UART example code&lt;br&gt;• Updated Thonny instructions&lt;br&gt;• Updated Project Generator instructions&lt;br&gt;• Added a FAQ document&lt;br&gt;• Added errata E7, E8 and E9</td>
</tr>
<tr>
<td>1.3.1</td>
<td>05 Mar 2021</td>
<td>• Minor corrections&lt;br&gt;• To accompany the V1.1.0 release of the C SDK&lt;br&gt;• Improved MicroPython UART example&lt;br&gt;• Improved Pinout diagram</td>
</tr>
<tr>
<td>1.4</td>
<td>07 Apr 2021</td>
<td>• Minor corrections&lt;br&gt;• Added errata E10&lt;br&gt;• Note about how to update the C SDK from Github&lt;br&gt;• To accompany the V1.1.2 release of the C SDK</td>
</tr>
<tr>
<td>Release</td>
<td>Date</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 1.4.1   | 13 Apr 2021| • Minor corrections  
• Clarified that all source code in the documentation is under the 3-Clause BSD license. |
| 1.5     | 07 Jun 2021| • Minor updates and corrections  
• Updated FAQ  
• Added SDK release history  
• To accompany the V1.2.0 release of the C SDK |
| 1.6     | 23 Jun 2021| • Minor updates and corrections  
• ADC information updated  
• Added errata E11 |
| 1.6.1   | 30 Sep 2021| • Minor updates and corrections  
• Information about B2 release  
• Updated errata for B2 release |
| 1.7     | 03 Nov 2021| • Minor updates and corrections  
• Fixed some register access types and descriptions  
• Added core 1 launch sequence info  
• Described SDK “panic” handling  
• Updated picotool documentation  
• Additional examples added to Appendix A: App Notes appendix in the Raspberry Pi Pico C/C++ SDK book  
• To accompany the V1.3.0 release of the C SDK |
| 1.7.1   | 04 Nov 2021| • Minor updates and corrections  
• Better documentation of USB double buffering  
• Picoprobe branch changes  
• Updated links to documentation |
| 1.8     | 17 Jun 2022| • Minor updates and corrections  
• Updated setup instructions for Windows in Getting started with Raspberry Pi Pico  
• Additional explanation of SDK configuration  
• RP2040 now qualified to -40°C, minimum operating temperature changed from -20°C to -40°C  
• Increased PLL min VCO from 400MHz to 750MHz for improved stability across operating conditions  
• Added reflow-soldering temperature profile  
• Added errata E12, E13 and E14  
• To accompany the V1.3.1 release of the C SDK |
<table>
<thead>
<tr>
<th>Release</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1.9     | 30 Jun 2022| - Minor updates and corrections  
|         |            | - Update to VGA board hardware description for launch of Raspberry Pi Pico W  
|         |            | - To accompany the V1.4.0 release of the C SDK  |
|         |            | Pico and Pico W databooks combined into a unified release history |
| 2.0     | 01 Dec 2022| - Minor updates and corrections  
|         |            | - Added RP2040 availability information  
|         |            | - Added RP2040 storage conditions and thermal characteristics  
|         |            | - Replace SDK library documentation with links to the online version  
|         |            | - Updated Picoprobe build and usage instructions  |
| 2.1     | 03 Mar 2023| - A large number of minor updates and corrections  
|         |            | - SMT footprint of Pico W corrected  
|         |            | - Updated for the 1.5.0 release of the Raspberry Pi Pico C SDK  
|         |            | - Added errata E15  
|         |            | - Added documentation around the new Pico Windows Installer  
|         |            | - Added documentation around the Pico-W-Go extension for Python development  
|         |            | - Added a wireless networking example to the Python documentation  
|         |            | - Added package marking specifications  
|         |            | - Added RP2040 baseline power consumption figures  
|         |            | - Added antenna keep out diagram to Pico W datasheet  |
| 2.2     | 14 Jun 2023| - Minor updates and corrections  
|         |            | - Updated for the 1.5.1 release of the Raspberry Pi Pico C SDK  
|         |            | - Documentation around Bluetooth support for Pico W  |
| 2.3     | 02 Feb 2024| - Numerous minor updates and corrections  
|         |            | - Update ROSC register information  
|         |            | - Updated getting started documentation for MS Windows and Apple macOS  
|         |            | - Updates arising from the release of Raspberry Pi 5  
|         |            | - Reinroduced updated SDK library documentation (was withdrawn in 2.0 due to XML conflicts)  
|         |            | - Updated to include the new recommended part number for crystals used with RP2040  
|         |            | - Added new paste stencil information for Pico and Pico W  
<p>|         |            | - Other updates to supporting documentation  |</p>
<table>
<thead>
<tr>
<th>Release</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
</table>
| 2.4     | 02 May 2024| • Numerous minor updates and corrections  
• Formatting fixes for Pico C SDK API level documentation  
• Renamed `picoprobe` firmware to `debugprobe`  
• Clarified that CMake build configuration uses cache variables, not configuration variables  
• Fixed incorrect parameter names used in the `@asm_pio` decorator and `StateMachine` initialiser examples  
• Expanded MicroPython `rshell` examples to include a complete guide on loading and running programs on your device  
• Added an example that demonstrates how to reset a Pico from the command line using OpenOCD  
• Enhanced VS Code MicroPico plugin documentation to reflect the plugin’s new name, removal of the built-in FTP server, and some additional usage instruction  
• Added documentation on the official Raspberry Pi Pico VS Code extension. |
