Arduino-Pico Documentation

Release 6.4.0

Earle F. Philhower, III

Nov 28, 2026

CONTENTS:

1 Getting Help **3**

2 Contributing and Porting to the Core **5**
- 2.1 Contributing to the Core (Pull Requests) . . . 5
- 2.2 Adding a New Board . . . 5
- 2.3 Porting Libraries and Applications to the Core . . . 6

3 Installation **9**
- 3.1 Installing via Arduino Boards Manager . . . 9
- 3.2 Installing via Arduino CLI . . . 10
- 3.3 Using Visual Micro on Windows . . . 10
- 3.4 Installing via GIT . . . 11
- 3.5 Installing both Arduino and CMake . . . 11
- 3.6 Uploading Sketches . . . 11
- 3.7 Uploading the First Sketch . . . 11
- 3.8 Windows 7 Driver Notes . . . 11
- 3.9 Windows 7 Installation Problems . . . 12
- 3.10 Uploading Filesystem Images . . . 12
- 3.11 Uploading Sketches with Picotool . . . 13
- 3.12 Uploading Sketches with Picoprobe . . . 13
- 3.13 Uploading Sketches with OpenOCD . . . 14
- 3.14 Debugging with Picoprobe/Debugprobe, OpenOCD, and GDB . . . 14

4 IDE Menus **15**
- 4.1 Board . . . 15
- 4.2 Flash Size . . . 15
- 4.3 CPU Speed . . . 15
- 4.4 Debug Port and Debug Level . . . 15
- 4.5 Generic RP2040 Support . . . 15
- 4.6 Boot Stage 2 Options for Generic RP2040 . . . 15

5 Using this core with PlatformIO **17**
- 5.1 What is PlatformIO? . . . 17
- 5.2 Important steps for Windows users, before installing . . . 19
- 5.3 Updating platformio.ini to use the proper platform . . . 20
- 5.4 Deprecation warnings . . . 20
- 5.5 Selecting the new core . . . 21
- 5.6 Flash size . . . 21
- 5.7 PSRAM size . . . 21
- 5.8 PSRAM chip select (CS) . . . 22

5.9 Boot2 Source . . . 22
5.10 CPU Speed . . . 22
5.11 Debug Port . . . 22
5.12 Debug Level . . . 23
5.13 C++ Exceptions . . . 23
5.14 Stack Protector . . . 23
5.15 RTTI . . . 23
5.16 USB Stack . . . 24
5.17 USB Customization . . . 24
5.18 IP Stack . . . 24
5.19 Bluetooth Stack . . . 24
5.20 FreeRTOS . . . 24
5.21 Selecting a different core version . . . 25
5.22 Selecting the CPU architecture . . . 25
5.23 Examples . . . 25
5.24 Debugging . . . 25
5.25 Filesystem Uploading . . . 27

6 Pin Assignments **29**
6.1 I2S . . . 29
6.2 Serial1 (UART0), Serial2 (UART1) . . . 29
6.3 SPI (SPI0), SPI1 (SPI1) . . . 29
6.4 Wire (I2C0), Wire1 (I2C1) . . . 29

7 RP2040 Helper Class **31**
7.1 Core Internals . . . 31
7.2 Hardware Watchdog . . . 31
7.3 Memory Information . . . 32
7.4 Hardware Identification . . . 32
7.5 Bootloader . . . 32
7.6 DMA-based MEMCPY . . . 32

8 Analog I/O **35**
8.1 Analog Input . . . 35
8.2 Analog Outputs . . . 35
8.3 Analog Output Restrictions . . . 35

9 Digital I/O **37**
9.1 Board-Specific Pins . . . 37
9.2 Pin Notation . . . 37
9.3 Input Modes . . . 37
9.4 Output Modes (Pad Strength) . . . 37
9.5 Tone/noTone . . . 37

10 BOOTSEL Button **39**

11 EEPROM Library **41**
11.1 EEPROM Class API . . . 41
11.2 EEPROM Examples . . . 42

12 I2S (Digital Audio) Audio Library **43**
12.1 I2S Class API . . . 43
12.2 Sample Writing/Reading API . . . 46
12.3 Note About 24-bit Samples . . . 47

13 PWM Audio Library **49**
13.1 PWM Class API . . . 49

14 ADC Input Library **51**
14.1 ADC Input API . . . 51

15 Camera (PicoCamera) Library **53**
15.1 Configuration (camera_config_t) . . . 53
15.2 Shared SCCB (I2C) Bus . . . 54
15.3 Memory Considerations . . . 54
15.4 Initializing the Camera . . . 55
15.5 The Frame Buffer (camera_fb_t) . . . 55
15.6 Reading and Processing Frames . . . 55
15.7 Runtime Sensor Control . . . 56
15.8 Shutting Down the Camera . . . 57
15.9 Camera API . . . 57
15.10 Error Codes . . . 58
15.11 Example Sketch . . . 58

16 Serial Ports (USB, UART, and BLE "Nordic SPP Service") **61**
16.1 Inversion . . . 61
16.2 Nordic SPP Serial Service . . . 62
16.3 RP2040 Specific SerialUSB methods . . . 62

17 "SoftwareSerial" PIO-based UART **63**
17.1 Inversion . . . 63

18 SoftwareSerial Emulation **65**

19 Servo Library **67**
19.1 Pulse Width Defaults . . . 67

20 SPI Master (Serial Peripheral Interface) **69**

21 Software SPI (Master Only) **71**

22 SPI Slave (SPISlave) **73**

23 Asynchronous Operation **75**
23.1 bool transferAsync(const void *send, void *recv, size_t bytes) . . . 75
23.2 bool finishedAsync() . . . 75
23.3 void abortAsync() . . . 75

24 Examples **77**

25 Wire (I2C Master and Slave) **79**
25.1 Asynchronous Operation . . . 79

26 File Systems **81**
26.1 Flash Layout . . . 81
26.2 Compatible Filesystem APIs . . . 81
26.3 SDFS/SD/FatFS Multicore and Multithreading(FreeRTOS) Caveats . . . 82
26.4 LittleFS Multicore and Multithreading(FreeRTOS) Caveats . . . 82
26.5 FatFS File System Caveats and Warnings . . . 82
26.6 LittleFS File System Limitations . . . 82
26.7 Uploading Files to the LittleFS File System on IDE 1.x (RP2040 only) . . . 83

26.8 Uploading Files to the LittleFS File System on IDE 2.x (RP2040 and RP2350) . . . 83
26.9 Downloading Files from a LittleFS System . . . 83
26.10 SD Library Information . . . 84
26.11 Using Second SPI port for SD . . . 84
26.12 Enabling SDIO operation for SD . . . 84
26.13 Using VFS (Virtual File System) for POSIX support . . . 84
26.14 File system object (LittleFS/SD/SDFS/FatFS) . . . 85
26.15 Filesystem information structure . . . 88
26.16 Directory object (Dir) . . . 88
26.17 File object . . . 89

27 USB (Arduino and Adafruit_TinyUSB) 93
27.1 Pico SDK USB Support . . . 93
27.2 USB class . . . 93
27.3 Dynamic USB Configuration . . . 93
27.4 HID Polling Interval . . . 94
27.5 Ethernet over USB . . . 94
27.6 Native TinyUSB Sketches . . . 94
27.7 Adafruit TinyUSB Arduino Support . . . 95
27.8 Adafruit TinyUSB Configuration and Quirks . . . 95

28 Multicore Processing 97
28.1 Core 1 Operation . . . 97
28.2 Stack Sizes . . . 97
28.3 Pausing Cores . . . 97
28.4 Communicating Between Cores . . . 98

29 Low Power (Sleep and Dormant) 99
29.1 Modes . . . 99
29.2 API . . . 100
29.3 Pins . . . 100
29.4 Waking on a pin or a timeout . . . 101
29.5 Current . . . 101
29.6 Caveats . . . 102
29.7 Keeping peripherals running in light sleep . . . 103
29.8 Examples . . . 104

30 Semihosting Support 105
30.1 Running Semihosting on the Development Host . . . 105
30.2 SerialSemi - Serial over Semihosting . . . 105
30.3 SemiFS - Host filesystem access through Semihosting . . . 105

31 Profiling Applications with GPROF 107
31.1 Enabling Profiling in an Application . . . 107
31.2 Collecting and Analyzing Profile Data . . . 108

32 RP2350 Specific Notes 109
32.1 ARM and RISC-V Modes . . . 109
32.2 P2350-E9 Errata (“Increased leakage current on Bank 0 GPIO when pad input is enabled”) . . . 109

33 RP2350 PSRAM Support 111
33.1 Using PSRAM for regular variables . . . 111
33.2 Using PSRAM for dynamic allocations . . . 111
33.3 Checking on PSRAM space . . . 112

34 Bluetooth on PicoW Support **113**
34.1 Enabling Bluetooth . . . 113
34.2 Included Bluetooth Libraries . . . 113
34.3 Writing Custom Bluetooth Applications . . . 113
34.4 Locking Requirements . . . 113

35 Bluetooth HID Master **115**
35.1 BTDeviceInfo Class . . . 115
35.2 BluetoothHCI Class . . . 115
35.3 BluetoothHIDMaster Operation . . . 115
35.4 Callback Event Handlers . . . 116
35.5 BluetoothHIDMaster Class . . . 117

36 Bluetooth Audio (A2DP Source and Sink) **119**
36.1 A2DPSink . . . 119
36.2 A2DPSource . . . 119

37 BLE (Bluetooth Low Energy) Server/Client **121**
37.1 General Architecture . . . 121
37.2 General Bluetooth Support Classes . . . 122
37.3 BLE (Bluetooth Low Energy) Base . . . 123
37.4 Accessing the Global BLEServer or BLEClient . . . 123
37.5 Configuring BLE Advertising . . . 123
37.6 Finding BLE Servers to Connect To . . . 124
37.7 Implementing a BLE Server (peripheral/device) . . . 124
37.8 Implementing a BLE Client (BLE Central) . . . 127
37.9 BLE Beacons . . . 129
37.10 BLE Battery Service . . . 129
37.11 BLE Serial UART (Nordic SPP Service) . . . 129

38 SingleFileDrive **131**
38.1 Callbacks, Interrupt Safety, and File Operations . . . 131
38.2 Using SingleFileDrive . . . 131

39 FatFSUSB **133**
39.1 Callbacks, Interrupt Safety, and File Operations . . . 133

40 FreeRTOS SMP **135**
40.1 Enabling FreeRTOS . . . 135
40.2 Configuration and Predefined Tasks . . . 135
40.3 Caveats . . . 135
40.4 More Information . . . 136

41 WiFi (Raspberry Pi Pico W) Support **137**
41.1 Non-Raspberry Pi WiFi Support . . . 137
41.2 Supported Features . . . 137
41.3 Powering the WiFi Chip Off (Pico W / Pico 2 W) . . . 137
41.4 Important Information . . . 138
41.5 Special Thanks . . . 138

42 EthernetLWIP (Wired Ethernet) Support **139**
42.1 Supported Wired Ethernet Modules . . . 139
42.2 Enabling Wired Ethernet . . . 139
42.3 Adjusting LWIP Polling . . . 140
42.4 Using Interrupt-Driven Handling . . . 140

42.5 Adjusting SPI Speed . . . 140
42.6 Using the WIZnet W5100S-EVB-Pico . . . 141
42.7 Example Code . . . 141
42.8 Caveats . . . 141
42.9 Special Thanks . . . 141

43 Ethernet over USB **143**
43.1 USB Device configuration on Raspberry Pi Pico . . . 143
43.2 USB Host configuration on Windows . . . 144
43.3 USB Host configuration on Linux . . . 144
43.4 Special Thanks . . . 147

44 WiFiClient **149**
44.1 flush and stop . . . 149
44.2 setNoDelay . . . 149
44.3 getNoDelay . . . 150
44.4 setSync . . . 150
44.5 getSync . . . 150
44.6 setDefaultNoDelay and setDefaultSync . . . 150
44.7 getDefaultNoDelay and getDefaultSync . . . 150
44.8 Other Function Calls . . . 150

45 Server Class **153**
45.1 accept . . . 153
45.2 available . . . 153
45.3 write (write to all clients) not supported . . . 153
45.4 setNoDelay . . . 154
45.5 Other Function Calls . . . 154

46 UDP Class **155**

47 Network Time Protocol (NTP) **157**
47.1 bool NTP.waitSet(uint32_t timeout) . . . 157
47.2 bool NTP.waitSet(void (*cb)(), uint32_t timeout) . . . 158

48 iLabs ESP-NOW **159**
48.1 Supported boards . . . 159
48.2 Selecting the ESP-NOW firmware type . . . 160
48.3 Differences from an ESP32 sketch . . . 160
48.4 A first sketch . . . 160
48.5 Board reset handling . . . 161
48.6 Discovery . . . 161
48.7 Notes and limitations . . . 162

49 iLabs Hearth (Matter) **163**
49.1 Supported boards . . . 163
49.2 Choosing a firmware . . . 163
49.3 Flashing the co-processor . . . 164
49.4 Writing a sketch . . . 164
49.5 How many endpoints fit . . . 165
49.6 Status and limitations . . . 165

50 BearSSL WiFi Classes **167**
50.1 CPU Requirements . . . 167
50.2 Memory Requirements . . . 167

50.3 Object Lifetimes . . . 167
50.4 TLS and HTTPS Basics . . . 168
50.5 Public and Private Keys . . . 168
50.6 TLS Sessions . . . 168
50.7 X.509 Certificate(s) . . . 169
50.8 Certificate Stores . . . 169
50.9 Supported Crypto . . . 169

51 WiFiClientSecure Class **171**
51.1 Validating X509 Certificates (Am I talking to the server I think I'm talking to?) . . . 171
51.2 Client Certificates (Proving I'm who I say I am to the server) . . . 172
51.3 MFLN or Maximum Fragment Length Negotiation (Saving RAM) . . . 172
51.4 Sessions (Resuming connections fast) . . . 173
51.5 Errors . . . 173
51.6 Limiting Ciphers (New connections faster) . . . 173
51.7 Limiting TLS(SSL) Versions . . . 173
51.8 setTLSConnectTimeout(int connectTimeout) . . . 173
51.9 ESP32 Compatibility . . . 174

52 WiFiServerSecure Class **175**
52.1 setBufferSizes(int recv, int xmit) . . . 175
52.2 Setting Server Certificates . . . 175
52.3 Client sessions (Resuming connections fast) . . . 175
52.4 Requiring Client Certificates . . . 176

53 HTTPClient Library **177**

54 OTA Updates **179**
54.1 Introduction . . . 179
54.2 Compression . . . 182
54.3 Uploading from the Arduino IDE . . . 183
54.4 Password Protection . . . 183
54.5 Web Browser . . . 184
54.6 HTTP Server . . . 184
54.7 Stream Interface . . . 185

55 Libraries Ported/Optimized for the RP2040 **187**

56 Using the Raspberry Pi Pico SDK (PICO-SDK) **189**
56.1 Included SDK . . . 189
56.2 Multicore (CORE1) Processing . . . 189
56.3 PIOASM (Compiling for the PIO processors) . . . 189

57 Licensing and Credits **191**

This is the documentation for the Raspberry Pi Pico Arduino core, Arduino-Pico. Arduino-Pico is a community port of Arduino to the RP2040 (Raspberry Pi Pico processor) and RP2350 (Raspberry Pi Pico 2 processor), intended to make it easier and more fun to use and program the Raspberry Pi Pico / RP2040 / RP2350 based boards.

This Arduino core uses a custom toolset with GCC 14.2 and Newlib 4.3 and doesn't require any system-installed prerequisites.

For the latest version, always check https://github.com/earlephilhower/arduino-pico

CHAPTER

ONE

GETTING HELP

This is a community supported project and has multiple ways to get assistance. Posting complete details, in a polite and organized way will get the best response.

For bugs in the Core, or to submit patches, please use the GitHub Issues or GitHub Pull Requests

For general questions/discussions use either GitHub Discussions or live-chat with gitter.im

CHAPTER

TWO

CONTRIBUTING AND PORTING TO THE CORE

First of all, thank you for contributing to the project. It's a lot of work keeping up with all the different uses of the RP2040, so the more people working on the code, the better. Your assistance can help the project succeed.

2.1 Contributing to the Core (Pull Requests)

We use the standard GitHub Pull Request model. If you're unfamiliar with it, this guide gives a simple overview of the process.

All pull requests have to pass a set of Continuous Integration (CI) checks which help make sure the code compiles under different configurations and has no spelling or style errors.

2.1.1 Tips for a Good Pull Request (PR)

All code in the core and libraries, except for INO sketches, uses a 4-space indent with cuddled brackets. When in doubt, copy your formatting from the surrounding code. You should install `astyle` and run `tests/restyle.sh` on your machine before committing and pushing any pull requests to ensure the formatting is correct.

Describe the change you're proposing and why it's important in your `git commit` message. If it fixes an open issue, place `Fixes #xxxx` (where xxxx is the issue number) in the message to link the two.

Try and only change one thing per pull request. That makes it easier to review and prioritize. Opening up a separate PR per change also helps keep track of them when release messages are generated.

2.2 Adding a New Board

Adding a new board requires:

- Updated `tools/makeboards.py` script
- Updated `boards.txt` file, generated by `makeboard.py`
- Updated `package_pico_index.template.json` file, generated by `makeboard.py`
- Updates `README.md` file, generated by `makeboard.py`
- New `tools/json/BOARD_NAME.json` board file for Platform.IO
- New `variants/BOARD_NAME/pins_arduino.h` header defining the I/O pins

To add a new RP2040 board you will need to update the `tools/makeboards.py` script. Do *NOT* manually edit `boards.txt`, that file is machine generated. You will need to add a `MakeBoard` call at the end of the file. Please be sure to add your board so that it sorts alphabetically, starting with the company name and then the board name. Otherwise it is hard to find a specific board in the menu.

Run `python3 tools/makeboards.py` to update the `boards.txt` file and generate a Platform.IO JSON file in the `tools/json` directory.

Create a folder called `variants/BOARD_NAME` and place in a `pins_arduino.h` file in it that contains your default pin name mapping (i.e. SPI0/1 pins, UART pins, LED_DEFAULT, etc.). Copying one of the existing ones as a template can make this task much simpler.

In your `git commit` be sure to add the newly generated `tools/json/XXX.json` file as well as the modified `makeboards` script and `boards.txt`, the new `pins_arduino.h` header you generated, and the Arduino packaging JSON `package/package_pico_index.template.json`.

Submit the updated commit as a PR and, if all goes well, your board will be in on the next core release.

2.3 Porting Libraries and Applications to the Core

We try and follow Arduino standards so, with luck, porting to this core should be relatively straightforward. The `WiFi` library and associates support libraries like `WebServer` are modeled after the ESP32 and ESP8266 versions of those libraries, combined with the "standard" Arduino `WiFi` one.

2.3.1 Compiler Defines for Porting

If you are adding RP2040 support to an existing library and need to isolate code that only runs on this core, use the following define.

```
#if defined(ARDUINO_ARCH_RP2040) && !defined(__MBED__)
~~~ your changes ~~~
#endif
```

2.3.2 Identifying RP2040, RP2350A, or RP2350B

To check if a board is an original RP2040

```
#if defined(PICO_RP2040)
...OG Pico code...
#endif
```

For RP2350(A or B):

```
#if defined(PICO_RP2350)
...Pico 2 code...
#endif
```

For only RP2350A variants (using the compile options, not the onboard ID register):

```
#if defined(PICO_RP2350A) && PICO_RP2350A
...RP2350A only code...
#endif
```

For only RP2350B variants (again, at compile time as identified by the selected board and not the chip ID register):

```
#if defined(PICO_RP2350A) && !PICO_RP2350A
...48-GPIO version code here
#endif
```

2.3.3 Library Architectures

After adding support in the code, libraries need their `library.properties` and `library.json` files updated to indicate support, or the IDE will not know your new code is compatible here.

Add `rp2040` to `architectures` (in `library.properties`) and `"rp2040"` to `platforms[]` (in `library.json`) to let the tools know. Note that even the RP2350 is identified as `rp2040` for this purpose.

2.3.4 Debugging

Diagnostic output can be enabled to help debug the core (or an application using the core) by setting the build-time options:

`Debug Port` (or FQBN `:dbgport=...`) sets the debug output serial port: `Serial`, `Serial1`, `Serial2`, `SerialSemi`, or `Disabled` (default). If the debug port disabled, all debug output is suppressed.

`Debug Level` (or FQBN `:dbglvl=...`) enables extra subsystem debugging: `None` (default), `Core`, `SPI`, `Wire`, `Bluetooth`, `BLE`, `LWIP`, or `All`. Note that even at `None`, some baseline chatter is enabled if a debug port is set.

Debug-logging macros are defined in `cores/rp2040/debug_internal.h` (included by `Arduino.h`):

- `DEBUGV("format", ...)` writes to the debug port if one is defined, otherwise gets compiled out. The macro adds a newline (CR-LF); do not include `\n` in the format string.
- `DEBUGCORE(...)` is `DEBUGV(...)` at debug level `Core` or `All`, otherwise nothing.
- `DEBUGWIRE(...)` is `DEBUGV(...)` at debug level `Wire` or `All`, otherwise nothing.
- `DEBUGSPI(...)` is `DEBUGV(...)` at debug level `SPI` or `All`, otherwise nothing.

Other subsystems (eg. Bluetooth) use other logging systems, see the code.

CHAPTER

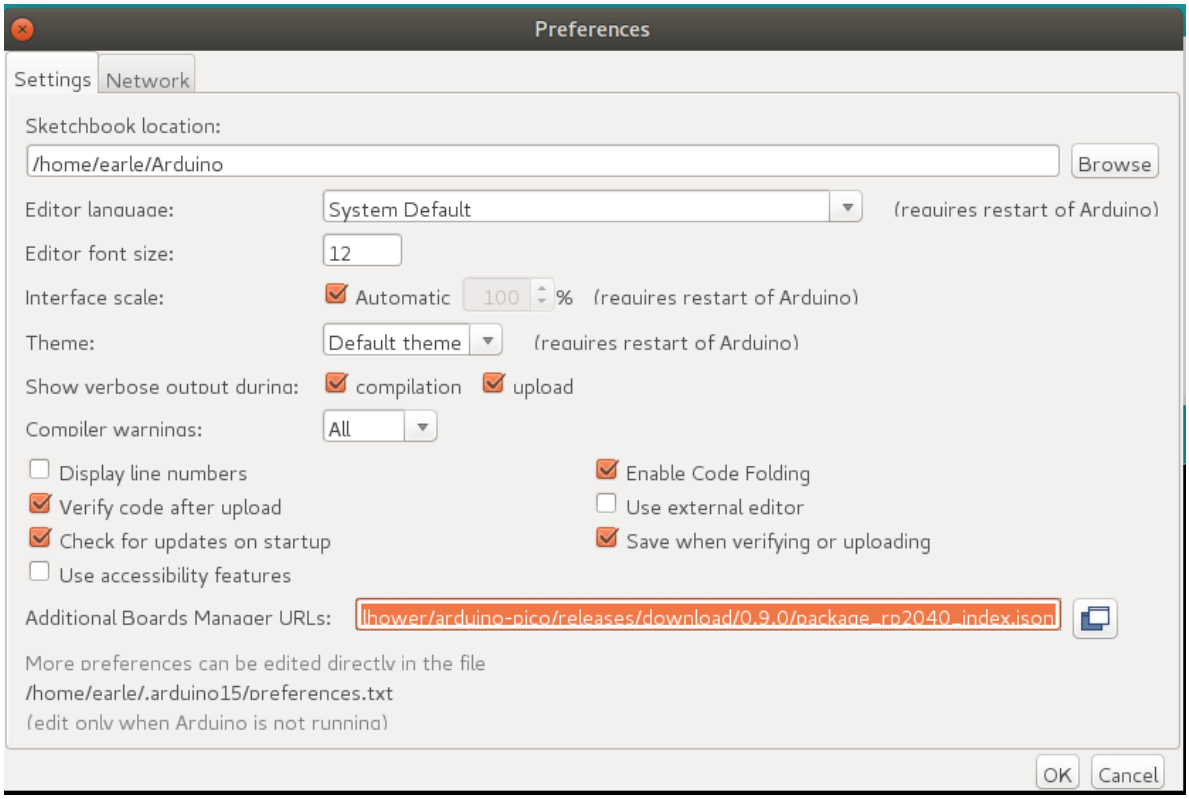
THREE

INSTALLATION

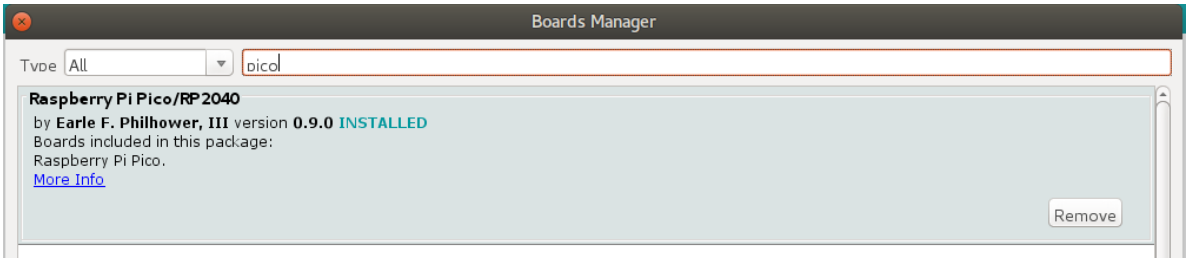
The Arduino-Pico core can be installed using the Arduino IDE Boards Manager or using *git*. If you want to simply write programs for your RP2040 board, the Boards Manager installation will suffice, but if you want to try the latest pre-release versions and submit improvements, you will need the *git* installation.

3.1 Installing via Arduino Boards Manager

1. Open up the Arduino IDE and go to File->Preferences.
2. In the dialog that pops up, enter the following URL in the "Additional Boards Manager URLs" field: https://github.com/earlephilhower/arduino-pico/releases/download/global/package_rp2040_index.json



3. Hit OK to close the dialog.
4. Go to Tools->Boards->Board Manager in the IDE
5. Type "pico" in the search box and select "Add":



3.1.1 Arduino IDE Installation Warning

Note for Windows Users: Please do not use the Windows Store version of the actual Arduino application because it has issues detecting attached Pico boards. Use the "Windows ZIP" or plain "Windows" executable (EXE) download direct from https://arduino.cc. and allow it to install any device drivers it suggests. Otherwise the Pico board may not be detected.

> **Note**
>
> **Windows users with non-ASCII characters in their username** (e.g., accented letters like Í, umlauts like ü, or characters like ñ): The ARM GCC toolchain (pqt-gcc) used by this core does not handle non-ASCII characters in Windows file paths. If your Windows username contains such characters (e.g., `C:\Users\Íñigo` instead of `C:\Users\Inigo`), compilation will fail with linker errors like `cannot open linker script file ... Invalid argument`. The recommended workaround is to create an additional local Windows user with an ASCII-only name and use that profile for RP2040/RP2350 development. See issue #2689 for more details.

Note for Linux Users: If you installed the Arduino IDE using Flatpak, which is common in Pop!_OS, Fedora, and Mint, among others, you may need to configure Flatpak to allow the IDE access to files outside your home folder. The RP2040 device is sometimes mounted as a folder in /opt or /media, which Flatpak will prevent the Arduino IDE from accessing. For Arduino IDE V2, override the filesystem restriction using `flatpak override --user --filesystem=host cc.arduino.IDE2` . For For Arduino IDE < V2, use `flatpak override --user --filesystem=host cc.arduino.arduinoide`.

3.2 Installing via Arduino CLI

To install using the Arduino command line tool (arduino-cli):

```
arduino-cli config add board_manager.additional_urls https://github.com/earlephilhower/
↪arduino-pico/releases/download/global/package_rp2040_index.json
arduino-cli core update-index
arduino-cli core install rp2040:rp2040
```

To list the supported boards:

```
arduino-cli board listall | grep rp2040
```

3.3 Using Visual Micro on Windows

Visual Micro's Pico board-package selection guide describes selecting an Arduino core for a Pico project in Microsoft Visual Studio. Use this Arduino-Pico package and its package index above when following that workflow; Arduino's Mbed OS RP2040 package is a separate core. The Visual Studio integration is a third-party extension with its own licensing and support provided by Visual Micro.

3.4 Installing via GIT

To install via GIT (for latest and greatest versions):

```
mkdir -p ~/Arduino/hardware/pico
git clone https://github.com/earlephilhower/arduino-pico.git ~/Arduino/hardware/pico/
↪rp2040
cd ~/Arduino/hardware/pico/rp2040
git submodule update --init
cd pico-sdk
git submodule update --init
cd ../tools
python3 ./get.py
```

3.5 Installing both Arduino and CMake

Tom's Hardware presented a very nice writeup on installing *arduino-pico* on both Windows and Linux, available at Tom's Hardware .

If you follow their step-by-step you will also have a fully functional *CMake*-based environment to build Pico apps on if you outgrow the Arduino ecosystem.

3.6 Uploading Sketches

To upload your first sketch, you will need to hold the BOOTSEL button down while plugging in the Pico to your computer. Then hit the upload button and the sketch should be transferred and start to run.

After the first upload, this should not be necessary as the *arduino-pico* core has auto-reset support. Select the appropriate serial port shown in the Arduino Tools->Port->Serial Port menu once (this setting will stick and does not need to be touched for multiple uploads). This selection allows the auto-reset tool to identify the proper device to reset. Them hit the upload button and your sketch should upload and run.

In some cases the Pico will encounter a hard hang and its USB port will not respond to the auto-reset request. Should this happen, just follow the initial procedure of holding the BOOTSEL button down while plugging in the Pico to enter the ROM bootloader.

3.7 Uploading the First Sketch

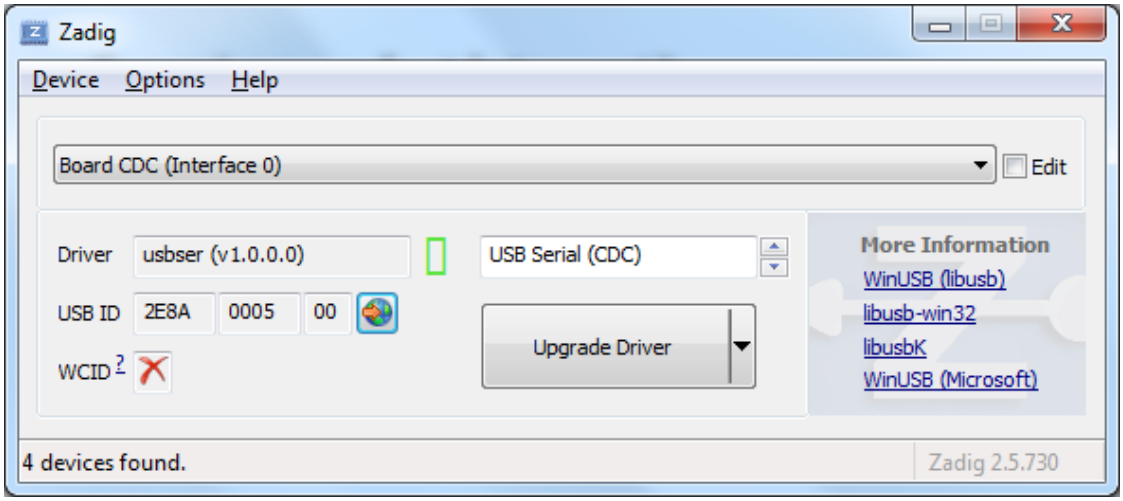
The first time you upload a sketch to a board, you'll need to use the built-in ROM bootloader to handle the upload and not a serial port.

1. Hold the BOOTSEL button while plugging in the board.
2. Select `Tools->Port->UF2 Board` from the menu.
3. Upload as normal.
4. After the board boots up, select the new serial port from the `Tools->Port` menu.

3.8 Windows 7 Driver Notes

Windows 10, Linux, and Mac will all support the Pico CDC/ACM USB serial port automatically. However, Windows 7 may not include the proper driver and therefore no detect the Pico for automatic uploads or the Serial Monitor.

For Windows 7, if this occurs, you can use *Zadig <https://zadig.akeo.ie/>* to install the appropriate driver. Select the USB ID of `2E8A` and use the `USB Serial (CDC)` driver.



3.9 Windows 7 Installation Problems

When running MalwareBytes antivirus (or others) the scanner may lock the compiler or other toolchain executables, causing installation or build failures. (Thanks to @Andy2No)

Symptoms include:

- Access denied during update in the boards manager - affects the .exe files, because MalwareBytes has locked them.
- Access denied during compilation, to one of the .exe files - same reason.
- Can't delete the .exe files - they're locked by MalwareBytes.

A workaround is possible, involving setting the toolchain as an "excluded directory" and reinstalling.

1. In MalwareBytes Settings, click the Exclusions tab. Add an exclusion for the equivalent of this folder path:

```
C:\Users{YOUR_USERNAME_HERE}\AppData\Local\Arduino15\packages\rp2040\tools\pqt-gcc\1.1.
0-a-81a1771
```

2. Reboot to unlock the files.
3. Do the boards manager installation / upgrade again.
4. Set the board type, e.g. to Raspberry Pi Pico and check it can compile.

3.10 Uploading Filesystem Images

The onboard flash filesystem for the Pico, LittleFS, lets you upload a filesystem image from the sketch directory for your sketch to use. Download the needed plugin from

- *IDE 1.x*: https://github.com/earlephilhower/arduino-pico-littlefs-plugin/releases
- *IDE 2.x*: https://github.com/earlephilhower/arduino-littlefs-upload/releases

To install, follow the directions in

- *IDE 1.x*: https://github.com/earlephilhower/arduino-pico-littlefs-plugin/blob/master/README.md
- *IDE 2.x*: https://github.com/earlephilhower/arduino-littlefs-upload/blob/main/README.md

For detailed usage information, please check the repo documentation available at

- https://arduino-pico.readthedocs.io/en/latest/fs.html

3.11 Uploading Sketches with Picotool

Because the Picotool uses a custom device driver in the Pico to handle upload, when using the `Upload Method->Picotool` mode custom code needs to be run on the Pico which is not included by default for compatibility and code savings.

So for the first sketch you will need to rebuild (with the `Upload Method->Picotool` selected in them menus) and then manually hold down BOOTSEL and insert the Pico USB cable to enter the ROM bootloader.

After the initial upload, as long as the running binary was built using the `Picotool` upload method, then the normal upload process should work.

For Ubuntu and other Linux operating systems you may need to add the following lines to a new *udev* config file(`99-picotool.rules`) to allow normal users to access the special USB device the Pico exports:

```
echo 'SUBSYSTEM=="usb", ATTRS{idVendor}=="2e8a", ATTRS{idProduct}=="0003", MODE="660",␣
↪GROUP="plugdev"' | sudo tee -a /etc/udev/rules.d/98-Picotool.rules
echo 'SUBSYSTEM=="usb", ATTRS{idVendor}=="2e8a", ATTRS{idProduct}=="000a", MODE="660",␣
↪GROUP="plugdev"' | sudo tee -a /etc/udev/rules.d/98-Picotool.rules
echo 'SUBSYSTEM=="usb", ATTRS{idVendor}=="2e8a", ATTRS{idProduct}=="f00a", MODE="660",␣
↪GROUP="plugdev"' | sudo tee -a /etc/udev/rules.d/98-Picotool.rules
echo 'SUBSYSTEM=="usb", ATTRS{idVendor}=="2e8a", ATTRS{idProduct}=="000f", MODE="660",␣
↪GROUP="plugdev"' | sudo tee -a /etc/udev/rules.d/98-Picotool.rules
echo 'SUBSYSTEM=="usb", ATTRS{idVendor}=="2e8a", ATTRS{idProduct}=="f00f", MODE="660",␣
↪GROUP="plugdev"' | sudo tee -a /etc/udev/rules.d/98-Picotool.rules
sudo udevadm control --reload
```

Note that in some Linux distributions the `plugdev` above needs to be `dialout` and for boards other than the Raspberry Pi Pico line you may need to manually add their USB `VID` and `PID` to the list above.

3.12 Uploading Sketches with Picoprobe

If you have built a Raspberry Pi Picoprobe, you can use OpenOCD to handle your sketch uploads and for debugging with GDB.

Under Windows a local admin user should be able to access the Picoprobe port automatically, but under Linux *udev* must be told about the device and to allow normal users access.

To set up user-level access to Picoprobes on Ubuntu (and other OSes which use *udev*):

```
echo 'SUBSYSTEMS=="usb", ATTRS{idVendor}=="2e8a", ATTRS{idProduct}=="0004", GROUP="users
↪", MODE="0666"' | sudo tee -a /etc/udev/rules.d/98-PicoProbe.rules
sudo udevadm control --reload
```

The first line creates a file with the USB vendor and ID of the Picoprobe and tells UDEV to give users full access to it. The second causes *udev* to load this new rule. Note that you will need to unplug and re-plug in your device the first time you create this file, to allow udev to make the device node properly.

Once Picoprobe permissions are set up properly, then select the board "Raspberry Pi Pico (Picoprobe)" in the Tools menu and upload as normal.

3.13 Uploading Sketches with OpenOCD

Under Windows and macOS, any user should be able to access OpenOCD automatically, but under Linux *udev* must be told about the device and to allow normal users access.

To set up user-level access to all CMSIS-DAP adapters on Ubuntu (and other OSes which use *udev*):

```
echo 'ATTRS{product}=="*CMSIS-DAP*", MODE="664", GROUP="plugdev"' | sudo tee -a /etc/
↪udev/rules.d/98-CMSIS-DAP.rules
sudo udevadm control --reload
```

The first line creates a file that recognizes all CMSIS-DAP adapters and tells UDEV to give users full access to it. The second causes *udev* to load this new rule. Note that you will need to unplug and re-plug in your device the first time you create this file, to allow udev to make the device node properly.

Once CMSIS-DAP permissions are set up properly, then select the Upload Method "Picoprobe/Debugprobe (CMSIS-DAP)" in the Tools menu.

3.14 Debugging with Picoprobe/Debugprobe, OpenOCD, and GDB

The installed tools include a version of OpenOCD (in the pqt-openocd directory) and GDB (in the pqt-gcc directory). These may be used to run GDB in an interactive window as documented in the Pico Getting Started manuals from the Raspberry Pi Foundation.

CHAPTER

FOUR

IDE MENUS

4.1 Board

Use the boards menu to select your model of RP2040 board.

There is also a *Generic RP2040* board which allows you to individually select things such as flash size or boot2 flash type. Use this if your board isn't yet fully supported and isn't working with the normal *Raspberry Pi Pico* option.

4.2 Flash Size

Arduino-Pico supports onboard filesystems which will set aside some of the flash on your board for the filesystem, shrinking the maximum code size allowed. Use this menu to select the desired ratio of filesystem to sketch.

4.3 CPU Speed

While it is unsupported, the Raspberry Pi Pico RP2040 can often run much faster than the stock 125MHz. Use the *CPU Speed* menu to select a desired over or underclock speed. **If the sketch fails at the higher speed, hold the BOOTSEL while plugging it in to enter update mode and try a lower overclock.**

4.4 Debug Port and Debug Level

Debug messages from *printf* and the Core can be printed to a Serial port to allow for easier debugging. Select the desired port and verbosity. Selecting a port for debug output does not stop a sketch from using it for normal operations.

4.5 Generic RP2040 Support

If your RP2040 board isn't in the menus you can still use it with the IDE by using the *Board->Generic RP2040* menu option. You will need to then set the flash size (see above) and tell the IDE how to communicate with the flash chip using the *Tools->Boot Stage 2* menu.

4.6 Boot Stage 2 Options for Generic RP2040

The Arduino Pico needs to set up its internal flash interface to talk to whatever flash chip is in the system. While all flash chips support a basic (and slow) 1-bit operation using common timings, each different brand (and sometimes model) of flash chip require custom timings to work in QSPI (4-bit) mode. The *Boot Stage 2* menu lets you select from the supported timings.

The options with */2* in them divide the system clock by 2 to drive the bus. Options with */4* divide the clock by 4 and so are slower but more compatible.

If you can't match a chip name in the menu to your flash chip, a simple test can be run to determine which is correct. Simpily load the *Blink* example, select the first option in the *Boot Stage 2* menu, and upload. If that works, note it and continue. Iterate through the options and note which ones work. If an option doesn't work, unplug the chip and hold the BOOTSEL button down while re-inserting it to enter the ROM uploader mode. (The CPU and flash will not be harmed if the test fails.)

If one of the custom bootloaders (not *Generic SPI /2 or /4*) worked, use that option to get best performance. If none worked other than the *Generic SPI /2 or /4* then use that. The */2* options of all models is preferred as it is faster, but some boards do require */4* on the custom chip interfaces.

When in doubt, *Generic SPI /4* should work with any flash chip but is slow.

CHAPTER

FIVE

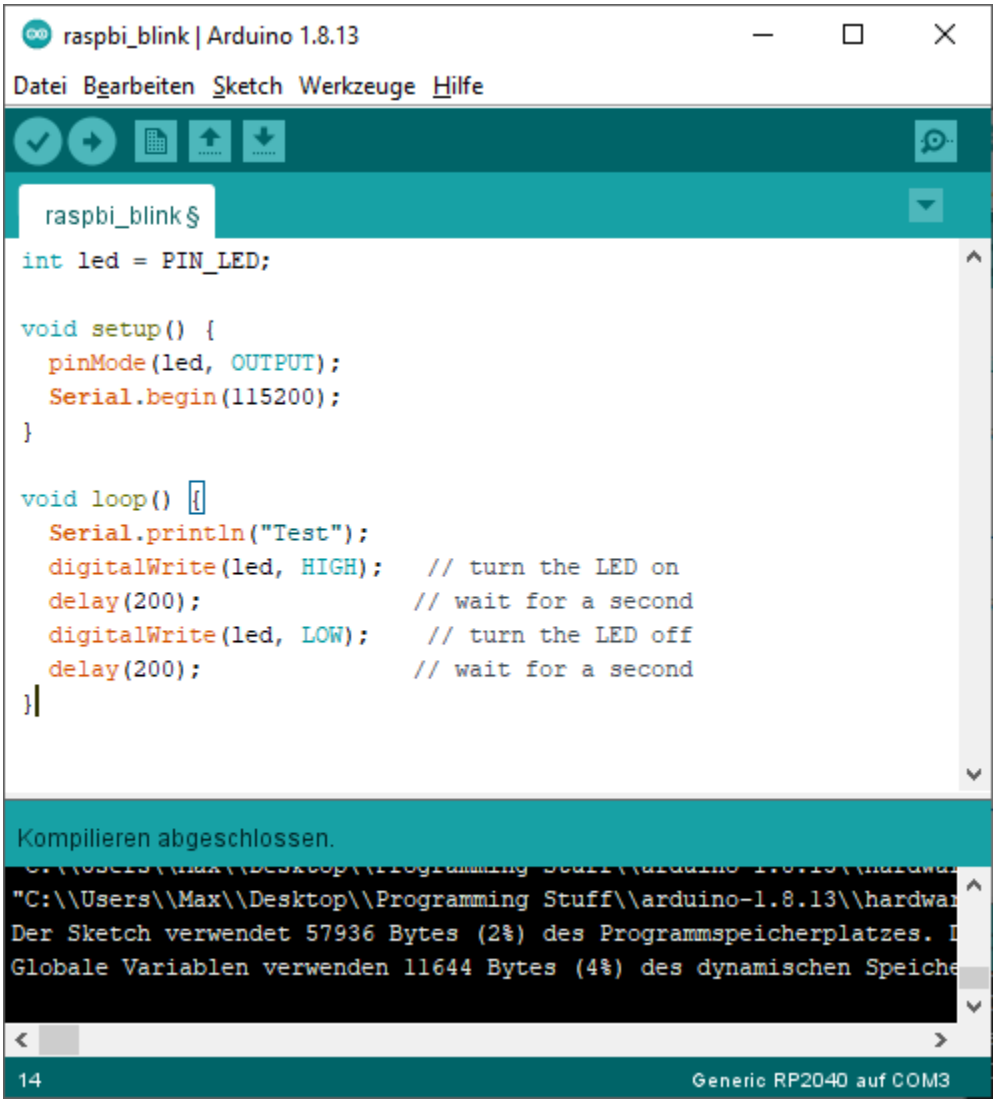
USING THIS CORE WITH PLATFORMIO

5.1 What is PlatformIO?

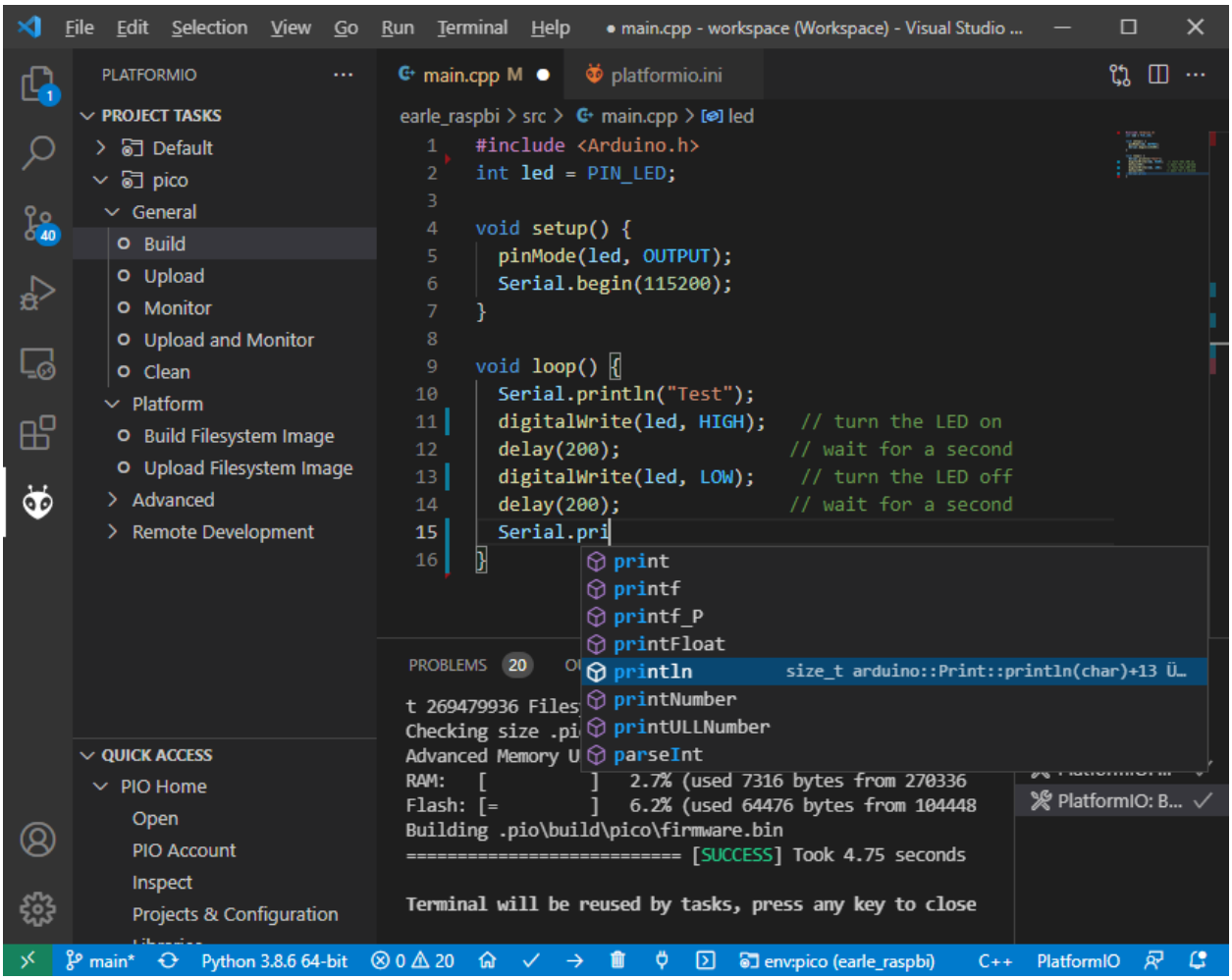
PlatformIO is a free, open-source build-tool written in Python, which also integrates into VSCode code as an extension.

PlatformIO significantly simplifies writing embedded software by offering a unified build system, yet being able to create project files for many different IDEs, including VSCode, Eclipse, CLion, etc. Through this, PlatformIO can offer extensive features such as IntelliSense (autocomplete), debugging, unit testing etc., which not available in the standard Arduino IDE.

The Arduino IDE experience:



The PlatformIO experience:



Refer to the general documentation at https://docs.platformio.org/.

Especially useful is the Getting started with VSCode + PlatformIO, CLI reference and the platformio.ini options page.

Hereafter it is assumed that you have a basic understanding of PlatformIO in regards to project creation, project file structure and building and uploading PlatformIO projects, through reading the above pages.

5.2 Important steps for Windows users, before installing

By default, Windows has a limited path length that is not long enough to fully clone the `Pico-SDK`'s `tinyusb` repository, resulting in error messages like the one below while attempting to fetch the repository.

```
error: unable to create file '.....' : Filename too long
```

To work around this requires performing two steps and rebooting Windows once. These steps will enable longer file paths at the Windows OS and the `git` level.

5.2.1 Step 1: Enabling long paths in git

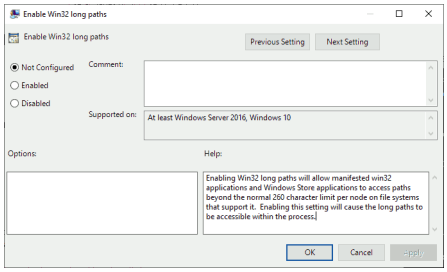
Open up a Windows `cmd` or `terminal` window and execute the following command

```
git config --system core.longpaths true
```

5.2.2 Step 2: Enabling long paths in the Windows OS

(taken from https://www.microfocus.com/documentation/filr/filr-4/filr-desktop/t47bx2ogpfz7.html)

1. Click Window key and type gpedit.msc, then press the Enter key. This launches the Local Group Policy Editor.
2. Navigate to Local Computer Policy > Computer Configuration > Administrative Templates > System > Filesystem.
3. Double click Enable NTFS/Win32 long paths and close the dialog.



5.2.3 Step 3: Reboot the computer

Once the two prior stages are complete, please do a full reboot or power cycle so that the new settings will take effect.

5.3 Updating platformio.ini to use the proper platform

First create a standard Raspberry Pi Pico + Arduino project within PlatformIO. This will give you a project with the `platformio.ini`

```
[env:pico]
platform = raspberrypi
board = pico
framework = arduino
```

Here, you need to change the *platform* to take advantage of the features described hereunder and switch to the new core.

```
[env:pico]
platform = https://github.com/maxgerhardt/platform-raspberrypi.git
board = pico
framework = arduino
board_build.core = earlephilhower
```

When the support for this core has been merged into mainline PlatformIO, this notice will be removed and a standard *platformio.ini* as shown above will work as a base.

5.4 Deprecation warnings

Previous versions of this documentation told users to inject the framework and toolchain package into the project by using

```
; note that download link for toolchain is specific for OS. see https://github.com/
↪earlephilhower/pico-quick-toolchain/releases.
platform_packages =
    maxgerhardt/framework-arduinopico@https://github.com/earlephilhower/arduino-pico.git
```

(continues on next page)

(continued from previous page)

```
    maxgerhardt/toolchain-pico@https://github.com/earlephilhower/pico-quick-toolchain/
↪releases/download/1.3.1-a/x86_64-w64-mingw32.arm-none-eabi-7855b0c.210706.zip
```

This is now **deprecated** and should not be done anymore. Users should delete these `platform_packages` lines and update the platform integration by issuing the command

```
pio pkg update -g -p https://github.com/maxgerhardt/platform-raspberrypi.git
```

in the PlatformIO CLI. The same can be achieved by using the VSCode PIO Home -> Platforms -> Updates GUI.

The toolchain, which was also renamed to `toolchain-rp2040-earlephilhower` is downloaded automatically from the registry. The same goes for the `framework-arduinopico` toolchain package, which points directly to the Arduino-Pico Github repository. However, users can still select a custom fork or branch of the core if desired so, as detailed in a chapter below.

5.5 Selecting the new core

Prerequisite for using this core is to tell PlatformIO to switch to it. There will be board definition files where the Earle-Philhower core will be the default since it's a board that only exists in this core (and not the other https://github.com/arduino/ArduinoCore-mbed). To switch boards for which this is not the default core (which are only `board = pico` and `board = nanorp2040connect`), the directive

```
board_build.core = earlephilhower
```

must be added to the `platformio.ini`. This controls the core switching logic.

When using Arduino-Pico-only boards like `board = rpipico` or `board = adafruit_feather`, this is not needed.

5.6 Flash size

Controlled via specifying the size allocated for the filesystem. Available sketch size is calculated accordingly by using (as in `makeboards.py`) that number and the (constant) EEPROM size (4096 bytes) and the total flash size as known to PlatformIO via the board definition file. The expression on the right can involve "b","k","m" (bytes/kilobytes/megabytes) and floating point numbers. This makes it actually more flexible than in the Arduino IDE where there is a finite list of choices. Calculations happen in the platform.

```
; in reference to a board = pico config (2MB flash)
; Flash Size: 2MB (Sketch: 1MB, FS:1MB)
board_build.filesystem_size = 1m
; Flash Size: 2MB (No FS)
board_build.filesystem_size = 0m
; Flash Size: 2MB (Sketch: 0.5MB, FS:1.5MB)
board_build.filesystem_size = 1.5m
```

5.7 PSRAM size

For RP2350 based boards, this controls how much PSRAM the firmware will think it has available in bytes, mapped at starting address 0x11000000.

To learn more about PSRAM usage, see: *RP2350 PSRAM Support*

```
; PSRAM size: 1MB
board_upload.psram_length = 1048576
; PSRAM size: 2MB
board_upload.psram_length = 2097152
; PSRAM size: 4MB
board_upload.psram_length = 4194304
```

5.8 PSRAM chip select (CS)

For RP2350 based boards, this controls what chip-select (also called: slave-select / SS) pin to use when wanting to talk to the PSRAM chip.

Note that it's not needed to set this with a board that is known to have a PSRAM chip on-board, such as a "Sparkfun Thing Plus 2350". The `pins_arduino.h` of that variant already has the correct definition.

To learn more about PSRAM usage, see: *RP2350 PSRAM Support*

```
; PSRAM CS is at GP47
build_flags =
  -DRP2350_PSRAM_CS=47
```

5.9 Boot2 Source

Boot2 is the second stage bootloader and predominantly used on the RP2040. Its main purpose is to configure the communication with the Flash at the highest, safest speed it can. All known boards have their correct value already configured. However, when choosing `board = generic`, you can freely configure the Boot2 to be for a different flash.

For possible Boot2 filenames, please see here.

```
; expect an ISSI IS25LP080 flash, SPI frequency = CPU frequency divided by 2
board_build.arduino.earlephilhower.boot2_source = boot2_is25lp080_2_padded_checksum.S
```

5.10 CPU Speed

As for all other PlatformIO platforms, the `f_cpu` macro value (which is passed to the core) can be changed as documented

```
; 133MHz
board_build.f_cpu = 133000000L
```

5.11 Debug Port

Via build_flags as done for many other cores (example).

```
; Debug Port: Serial
build_flags = -DDEBUG_RP2040_PORT=Serial
; Debug Port: Serial 1
build_flags = -DDEBUG_RP2040_PORT=Serial1
; Debug Port: Serial 2
build_flags = -DDEBUG_RP2040_PORT=Serial2
```

5.12 Debug Level

Done again by directly adding the needed build flags. When wanting to define multiple build flags, they must be accumulated in either a single line or a newline-separated expression.

```
; Debug level: Core
build_flags = -DDEBUG_RP2040_CORE
; Debug level: SPI
build_flags = -DDEBUG_RP2040_SPI
; Debug level: Wire
build_flags = -DDEBUG_RP2040_WIRE
; Debug level: All
build_flags = -DDEBUG_RP2040_WIRE -DDEBUG_RP2040_SPI -DDEBUG_RP2040_CORE
; Debug level: NDEBUG
build_flags = -DNDEBUG

; example: Debug port on serial 2 and all debug output
build_flags = -DDEBUG_RP2040_WIRE -DDEBUG_RP2040_SPI -DDEBUG_RP2040_CORE -DDEBUG_RP2040_
↪PORT=Serial2
; equivalent to above
build_flags =
   -DDEBUG_RP2040_WIRE
   -DDEBUG_RP2040_SPI
   -DDEBUG_RP2040_CORE
   -DDEBUG_RP2040_PORT=Serial2
```

5.13 C++ Exceptions

Exceptions are disabled by default. To enable them, use

```
; Enable Exceptions
build_flags = -DPIO_FRAMEWORK_ARDUINO_ENABLE_EXCEPTIONS
```

5.14 Stack Protector

To enable GCC's stack protection feature, use

```
; Enable Stack Protector
build_flags = -fstack-protector
```

5.15 RTTI

RTTI (run-time type information) is disabled by default. To enable it, use

```
; Enable RTTI
build_flags = -DPIO_FRAMEWORK_ARDUINO_ENABLE_RTTI
```

5.16 USB Stack

Not specifying any special build flags regarding this gives one the default Pico SDK USB stack. To change it, add

```
; Adafruit TinyUSB
build_flags = -DUSE_TINYUSB
; No USB stack
build_flags = -DPIO_FRAMEWORK_ARDUINO_NO_USB
```

Note that the special "No USB" setting is also supported, through the shortcut-define `PIO_FRAMEWORK_ARDUINO_NO_USB`.

5.17 USB Customization

If you want to change the USB VID, PID, product or manufacturer name that the device will appear under, configure them as follows:

```
board_build.arduino.earlephilhower.usb_manufacturer = Custom Manufacturer
board_build.arduino.earlephilhower.usb_product = Ultra Cool Product
board_build.arduino.earlephilhower.usb_vid = 0xABCD
board_build.arduino.earlephilhower.usb_pid = 0x1337
```

5.18 IP Stack

The lwIP stack can be configured to support only IPv4 (default) or additionally IPv6. To activate IPv6 support, add

```
; IPv6
build_flags = -DPIO_FRAMEWORK_ARDUINO_ENABLE_IPV6
```

to the `platformio.ini`.

5.19 Bluetooth Stack

The Bluetooth Classic (BTC) and Bluetooth Low Energy (BLE) stack can be activated by adding

```
; BTC and BLE
build_flags = -DPIO_FRAMEWORK_ARDUINO_ENABLE_BLUETOOTH
```

to the `platformio.ini`.

5.20 FreeRTOS

FreeRTOS support can be activated by adding

```
; Enable FreeRTOS Support
build_flags = -DPIO_FRAMEWORK_ARDUINO_ENABLE_FREERTOS
```

to the `platformio.ini`.

5.21 Selecting a different core version

If you wish to use a different version of the core, e.g., the latest git `master` version, you can use a platform_packages directive to do so. Simply specify that the framework package (`framework-arduinopico`) comes from a different source.

```
platform_packages =
   framework-arduinopico@https://github.com/earlephilhower/arduino-pico.git#master
```

Whereas the `#master` can also be replaced by a `#branchname` or a `#commithash`. If left out, it will pull the default branch, which is `master`.

The `file://` and `symlink://` pseudo-protocols can also be used instead of `https://` to point to a local copy of the core (with e.g. some modifications) on disk (see documentation).

Note that this can only be done for versions that have the PlatformIO builder script it in, so versions before 1.9.2 are not supported.

5.22 Selecting the CPU architecture

By default Platform.IO will build for the onboard ARM cores on the RP2350. To build RISC-V binaries adjust the `board_build.mcu` option accordingly:

```
; RP2350 based (RISC-V)
[env:rpipico2-riscv]
board = rpipico2
board_build.mcu = rp2350-riscv
```

5.23 Examples

The following example `platformio.ini` can be used for a Raspberry Pi Pico and 0.5MByte filesystem.

```
[env:pico]
platform = https://github.com/maxgerhardt/platform-raspberrypi.git
board = pico
framework = arduino
; board can use both Arduino cores -- we select Arduino-Pico here
board_build.core = earlephilhower
board_build.filesystem_size = 0.5m
```

The initial project structure should be generated just creating a new project for the Pico and the Arduino framework, after which the auto-generated `platformio.ini` can be adapted per above.

5.24 Debugging

With recent updates to the toolchain and OpenOCD, debugging firmwares is also possible.

To specify the debugging adapter, use `debug_tool` (documentation). Supported values are:

- `picoprobe`
- `cmsis-dap`
- `jlink`

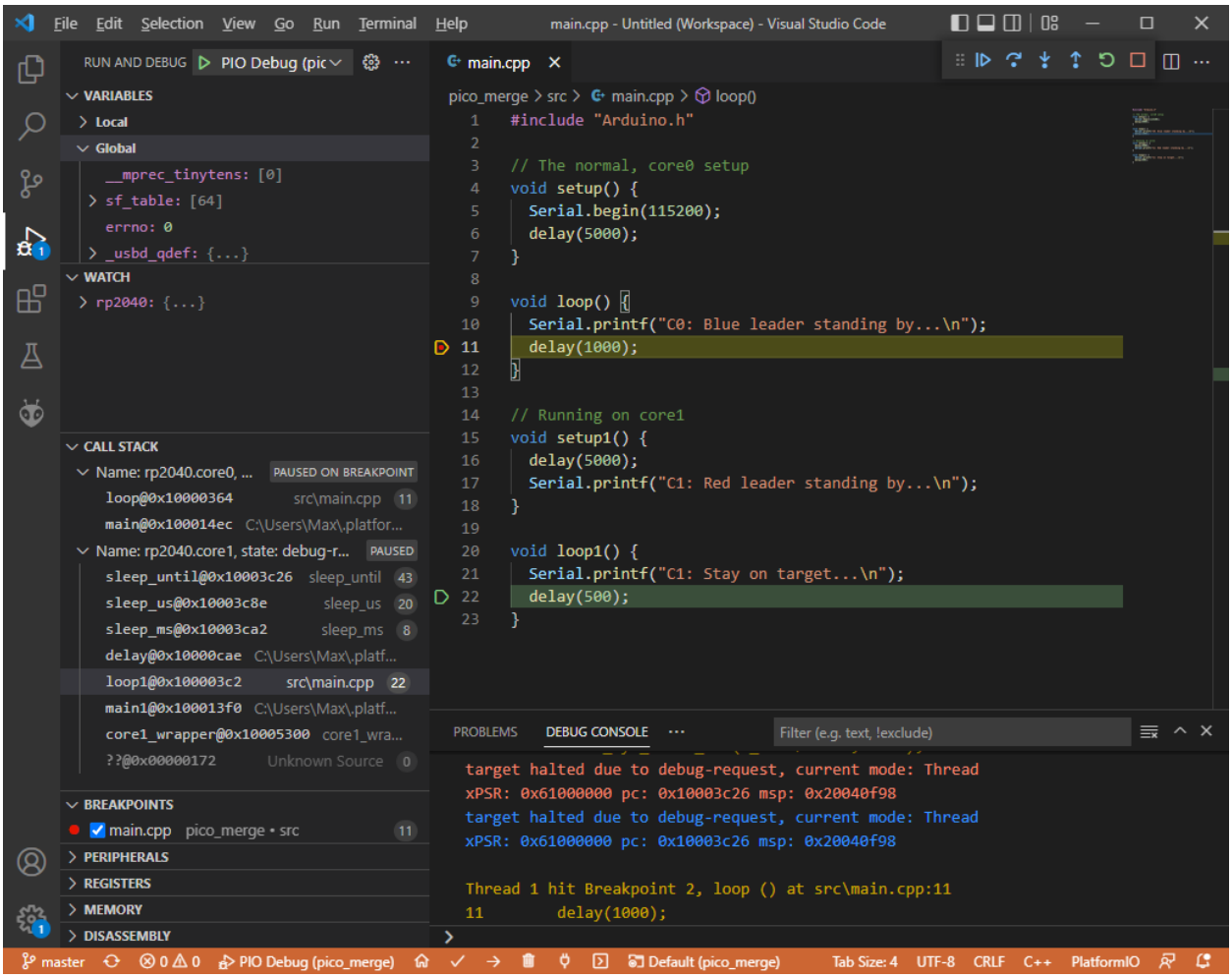
- `raspberrypi-swd`
- `blackmagic`

These values can also be used in `upload_protocol` if you want PlatformIO to upload the regular firmware through this method, which you likely want.

Especially the PicoProbe method is convenient when you have two Raspberry Pi Pico boards. One of them can be flashed with the PicoProbe firmware (documentation) and is then connected to the target Raspberry Pi Pico board (see documentation chapter "Picoprobe Wiring"). Remember that on Windows, you have to use Zadig to also load "WinUSB" drivers for the "Picoprobe (Interface 2)" device so that OpenOCD can speak to it.

> **Note**
>
> Newer PicoProbe firmware versions have dropped the proprietary "PicoProbe" USB communication protocol and emulate a **CMSIS-DAP** instead. Meaning, you have to use `debug_tool = cmsis-dap` for these newer firmwares, such as those obtained from raspberrypi/picoprobe

With that set up, debugging can be started via the left debugging sidebar and works nicely: Setup breakpoints, inspect the value of variables in the code, step through the code line by line. When a breakpoint is hit or execution is halted, you can even see the execution state both Cortex-M0+ cores of the RP2040.



For further information on customizing debug options, like the initial breakpoint or debugging / SWD speed, consult

the documentation.

Note

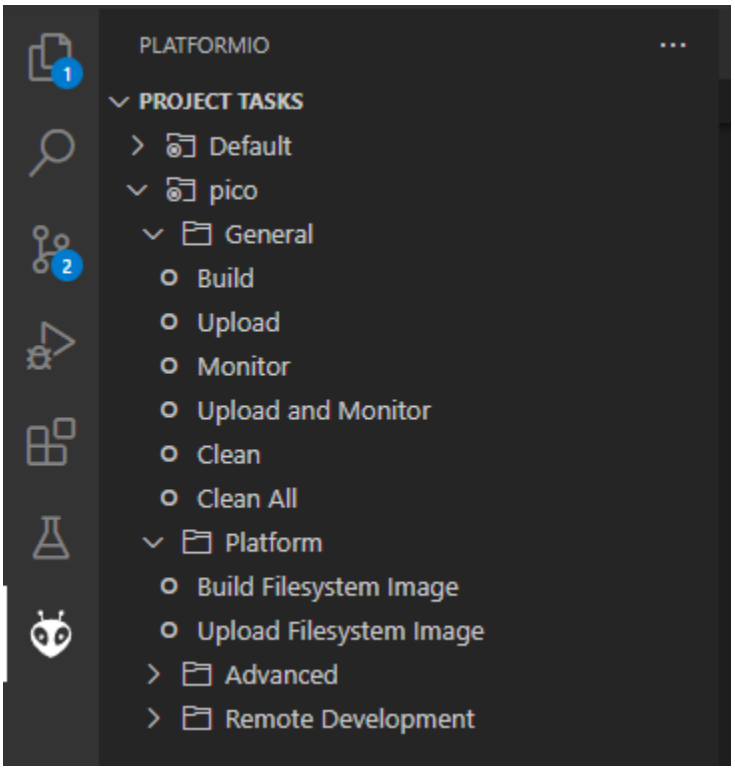
For the BlackMagicProbe debugging probe (as can be e.g., created by simply flashing a STM32F103C8 "Bluepill" board), you currently have to use the branch `fix/rp2040-flash-reliability` (or at least commit `1d001bc`) **and** use the official ARM provided toolchain.

You can obtain precompiled binaries from here. A flashing guide is available here. You then have to configure the target serial port ("GDB port") in your project per documentation.

5.25 Filesystem Uploading

For the Arduino IDE, a plugin is available that enables a data folder to be packed as a LittleFS filesystem binary and uploaded to the Pico.

This functionality is also built-in in the PlatformIO integration. Open the project tasks and expand the "Platform" tasks:



The files you want to upload should be placed in a folder called `data` inside the project. This can be customized if needed.

The task "Build Filesystem Image" will take all files in the data directory and create a `littlefs.bin` file from it using the `mklittlefs` tool.

The task "Upload Filesystem Image" will upload the filesystem image to the Pico via the specified `upload_protocol`.

Note

Set the space available for the filesystem in the `platformio.ini` using e.g., `board_build.filesystem_size = 0.5m`, or filesystem creation will fail!

CHAPTER

SIX

PIN ASSIGNMENTS

The Raspberry Pi Pico has an incredibly flexible I/O configuration and most built-in peripherals (except for the ADC) can be used on multiple sets of pins. Note, however, that not all peripherals can use all I/Os. Refer to the RP2040 datasheet or an online pinout diagram for more details.

Additional methods have been added to allow you to select a peripheral's I/O pins **before calling ::begin**. This is especially helpful when using third party libraries: the library doesn't need to be modified, only your own code in *setup()* is needed to adjust pinouts.

6.1 I2S

```
::setBCLK(pin)
::setDOUT(pin)
```

6.2 Serial1 (UART0), Serial2 (UART1)

```
::setRX(pin)
::setTX(pin)
::setRTS(pin)
::setCTS(pin)
```

6.3 SPI (SPI0), SPI1 (SPI1)

```
::setSCK(pin)
::setCS(pin)
::setRX(pin)
::setTX(pin)
```

6.4 Wire (I2C0), Wire1 (I2C1)

```
::setSDA(pin)
::setSCL(pin)
```

For example, because the *SD* library uses the *SPI* library, we can make it use a non-default pinout with a simple call

```
void setup() {
    SPI.setRX(4);
```

(continues on next page)

(continued from previous page)

```
    SPI.setTX(7);
    SPI.setSCK(6);
    SPI.setCS(5);
    SD.begin(5);
}
```

CHAPTER

SEVEN

RP2040 HELPER CLASS

Some of the core functionality of the RP2040 chip powering the Raspberry Pi Pico is exposed in the RP2040 class variable `rp2040`.

7.1 Core Internals

7.1.1 int rp2040.f_cpu()

Returns the current frequency of the core clock. This is read at runtime, versus the constant `F_CPU` macro that is also available. This is useful in cases where your code changes the core clock (i.e. low power modes, etc.)

7.1.2 int rp2040.cpuid()

Returns the current core ID (0 or 1) of the executing task.

7.1.3 uint32_t rp2040.getCycleCount()

Returns a 32-bit cycle count from when the core started running. Because it is only 32-bits, and the Pico runs at 133MHz, this value can loop around in a matter of seconds.

7.1.4 uint64_t rp2040.getCycleCount64()

Returns a 64-bit cycle count from when the core started running. This value should never loop around in normal mode (at 133MHz it would take over 4,000 years to overflow).

7.1.5 uint32_t rp2040.hwrand32()

Returns a 32-bit value derived from the CPU cycle counter and the ROSC oscillator. Because the ROSC bit is not a true random number generator, the values returned may not meet the most stringent random tests. **If your application needs absolute bulletproof random numbers, consider using dedicated external hardware.**

7.1.6 void rp2040.reboot()

Forces a hardware reboot of the Pico.

7.2 Hardware Watchdog

7.2.1 void rp2040.wdt_begin(uint32_t delay_ms)

Enables the hardware watchdog timer with a delay value of delay_ms milliseconds. Note that on the RP2040, once this function has called, the hardware watchdog can _not_ be disabled.

The maximum `delay_ms` allowed in this call is `8300`, corresponding to 8.3 seconds. Any higher values will be truncated by the hardware.

7.2.2 void rp2040.wdt_reset()

Reloads the watchdog's counter with the amount of time set by wdt_begin.

7.2.3 RP2040::resetReason_t rp2040.getResetReason()

Returns the reason for the last reset, defined in enum `RP2040::resetReason_t`. See example `` `ResetReason` `` for some details.

7.3 Memory Information

7.3.1 int rp2040.getFreeHeap()

Returns the number of bytes free for heap allocation (i.e. malloc, new). Note that because there is some overhead, and there may be heap fragmentation, this number is an *upper bound* and you generally will only be able to allocate less than this returned number.

7.3.2 int rp2040.getUsedHeap()

Returns the number of bytes allocated out of the heap.

7.3.3 int rp2040.getTotalHeap()

Returns the total heap that was available to this program at compile time (i.e. the Pico RAM size minus things like the `.data` and `.bss` sections and other overhead).

7.4 Hardware Identification

7.4.1 bool rp2040.isPicoW()

Returns the core's best guess if this code is running on a Raspberry Pi Pico W. This would let you, possibly, use the same UF2 for Pico and PicoW by simply not doing any WiFi calls.

7.5 Bootloader

7.5.1 void rp2040.enableDoubleResetBootloader() (Pico/RP2040 only)

Add a call anywhere in the sketch to `rp2040.enableDoubleResetBootloader()` and the core will check for a double-tap on reset, and if found will start the USB bootloader.

7.5.2 void rp2040.rebootToBootloader()

Will reboot the RP2040 into USB UF2 upload mode.

7.6 DMA-based MEMCPY

The onboard DMA engines can copy 4-byte aligned quantities faster than the CPU.

7.6.1 void *rp2040.memcpyDMA(void *dest, const void *src, size_t count);

Uses a DMA engine to transfer data from src to dest in 4-byte chunks without CPU intervention. If any arguments are not 4-byte aligned, or if the count is not a multiple of 4, then it will fall back to CPU-managed memcpy.

CHAPTER

EIGHT

ANALOG I/O

8.1 Analog Input

For analog inputs, the RP2040 device has a 12-bit, 4-channel ADC + temperature sensor available on a fixed set of pins (A0…A3). The standard Arduino calls can be used to read their values (with 3.3V nominally reading as 1023 with the default resolution of 10bits).

8.1.1 int analogRead(pin_size_t pin = A0..A3)

Returns a value from 0…1023 (with the default resolution of 10bits) corresponding to the ADC reading of the specific pin.

8.1.2 void analogReadResolution(int bits)

Determines the resolution (in bits) of the value returned by the analogRead() function. Default resolution is 10bit.

8.1.3 float analogReadTemp(float vref = 3.3f)

Returns the temperature, in Celsius, of the onboard thermal sensor. If you have a custom Vref for the ADC on your RP2040 board, you can pass it in as a parameter. Calling with no parameters assumes the normal, 3.3V Vref. This reading is not exceedingly accurate and of relatively low resolution, so it is not a replacement for an external temperature sensor in many cases.

8.2 Analog Outputs

The RP2040 does not have any onboard DACs, so analog outputs are simulated using the standard method of using pulse width modulation (PWM) using the RP2040's hardware PWM units.

While up to 16 PWM channels can be generated, they are not independent and there are significant restrictions as to allowed pins in parallel. See the RP2040 datasheet for full details.

8.3 Analog Output Restrictions

The PWM generator source clock restricts the legal combinations of frequency and ranges. At a CPU frequency of 133MHz, the 16 bit maximum range decreases by 1 bit for every doubling of the default PWM frequency of 1 kHz. For example, at 1MHz only about 6 bits of range are possible. When you define an `analogWriteFreq` and `analogWriteRange` that can't be fulfilled by the hardware, the frequency will be preserved but the accuracy (range) will be reduced automatically. Your code will still send in the range you specify, but the core itself will transparently map it into the allowable PWN range.

8.3.1 void analogWriteFreq(uint32_t freq)

Sets the master PWM frequency used (i.e. how often the PWM output cycles). From 100Hz to 1MHz are supported.

8.3.2 void analogWriteRange(uint32_t range) and analogWriteResolution(int res)

These calls set the maximum PWM value (i.e. writing this value will result in a PWM duty cycle of 100%)/ either explicitly (range) or as a power-of-two (res). A range of 16 to 65535 is supported.

8.3.3 void analogWrite(pin_size_t pin, int val)

Writes a PWM value to a specific pin. The PWM machine is enabled and set to the requested frequency and scale, and the output is generated. This will continue until a `digitalWrite` or other digital output is performed.

CHAPTER

NINE

DIGITAL I/O

9.1 Board-Specific Pins

The Raspberry Pi Pico RP2040 chip supports up to 30 digital I/O pins, however not all boards provide access to all pins.

9.2 Pin Notation

When using Analog or Digital I/Os, if you supply an integer it specifies the RP2040 GPIO pin to use. Using Dx or Ax notation (for example, D4 or A3) may be necessary on boards without a direct PCB pin to GPIO mapping.

9.3 Input Modes

The Raspberry Pi Pico has 3 Input modes settings for use with *pinMode*: *INPUT*, *INPUT_PULLUP* and *INPUT_PULLDOWN*

9.4 Output Modes (Pad Strength)

The Raspberry Pi Pico has the ability to set the current that a pin (actually the pad associated with it) is capable of supplying. The current can be set to values of 2mA, 4mA, 8mA and 12mA. By default, on a reset, the setting is 4mA. A *pinMode(x, OUTPUT)*, where *x* is the pin number, is also the default setting. 4 settings have been added for use with *pinMode*: *OUTPUT_2MA*, *OUTPUT_4MA*, which has the same behavior as *OUTPUT*, *OUTPUT_8MA* and *OUTPUT_12MA*.

9.5 Tone/noTone

Simple square wave tone generation is possible for up to 8 channels using Arduino standard `tone` calls. Because these use the PIO to generate the waveform, they must share resources with other calls such as `I2S` or `Servo` objects.

CHAPTER

TEN

BOOTSEL BUTTON

The BOOTSEL button on the Pico is not connected to a standard GPIO, so it cannot be read using the usual `digitalRead` function. It **can**, however, be read using a special (relatively slow) method.

The `BOOTSEL` object implements a simple way of reading the BOOTSEL button. Simply use the object `BOOTSEL` as a boolean (as a conditional in an `if` or `while`, or assigning to a `bool`):

```
// Print "BEEP" if the BOOTSEL button is pressed
if (BOOTSEL) {
    Serial.println("BEEP!");
    // Wait until BOOTSEL is released
    while (BOOTSEL) {
        delay(1);
    }
}
```

CHAPTER

ELEVEN

EEPROM LIBRARY

While the Raspberry Pi Pico RP2040 does not come with an EEPROM onboard, we simulate one by using a single 4K chunk of flash at the end of flash space.

Note that this is a simulated EEPROM and will only support the number of writes as the onboard flash chip, not the 100,000 or so of a real EEPROM. Therefore, do not frequently update the EEPROM or you may prematurely wear out the flash.

11.1 EEPROM Class API

11.1.1 EEPROM.begin(size=256...4096)

Call before the first use of the EEPROM data for read or write. It makes a copy of the emulated EEPROM sector in RAM to allow random update and access.

11.1.2 EEPROM.read(addr), EEPROM[addr]

Returns the data at a specific offset in the EEPROM. See *EEPROM.get* later for a more

11.1.3 EEPROM.write(addr, data), EEPROM[addr] = data

Writes a byte of data at the offset specified. Not persisted to flash until `EEPROM.commit()` is called.

11.1.4 EEPROM.commit()

Writes the updated data to flash, so next reboot it will be readable.

11.1.5 EEPROM.end()

`EEPROM.commit()` and frees all memory used. Need to call *EEPROM.begin()* before the EEPROM can be used again.

11.1.6 EEPROM.get(addr, val)

Copies the (potentially multi-byte) data in EEPROM at the specific byte offset into the returned value. Useful for reading structures from EEPROM.

11.1.7 EEPROM.put(addr, val)

Copies the (potentially multi-byte) value into EEPROM a the byte offset supplied. Useful for storing `struct` in EEPROM. Note that any pointers inside a written structure will not be valid, and that most C++ objects like `String` cannot be written to EEPROM this way because of it.

11.1.8 EEPROM.length()

Returns the length of the EEPROM (i.e. the value specified in `EEPROM.begin()`).

11.2 EEPROM Examples

Three EEPROM examples are included.

CHAPTER

TWELVE

I2S (DIGITAL AUDIO) AUDIO LIBRARY

While the RP2040 chip on the Raspberry Pi Pico does not include a hardware I2S device, it is possible to use the PIO (Programmable I/O) state machines to implement one dynamically.

Digital audio input and output are supported at 8, 16, 24, and 32 bits per sample.

Theoretically up to 6 I2S ports may be created, but in practice there may not be enough resources (DMA, PIO SM) to actually create and use so many.

Create an I2S port by instantiating a variable of the I2S class specifying the direction. Configure it using API calls below before using it.

12.1 I2S Class API

12.1.1 I2S(OUTPUT)

Creates an I2S output port. Needs to be connected up to the desired pins (see below) and started before any output can happen.

12.1.2 I2S(INPUT)

Creates an I2S input port. Needs to be connected up to the desired pins (see below) and started before any input can happen.

12.1.3 I2S(INPUT_PULLUP)

Creates a bi-directional I2S input and output port. Needs to be connected up to the desired pins (see below) and started before any input or output can happen.

12.1.4 bool setSlave()

Enables slave mode. BCLK and LRCLK are inputs and used to control the timing of the DOUT output. Only normal I2S output and input modes are supported in slave mode.

In I2S input slave mode, the clock pins and swapClocks are ignored. The pins must be consecutive starting with DIN, then BCLK, then LRCLK.

12.1.5 bool setBCLK(pin_size_t pin)

Sets the BCLK pin of the I2S device. The LRCLK/word clock will be `pin + 1` due to limitations of the PIO state machines. Call this before `I2S::begin()`

12.1.6 bool setDATA(pin_size_t pin)

Sets the DOUT or DIN pin of the I2S device. Any pin may be used. In bi-directional operation, must use `I2S::setDOUT()` and `I2S::setDIN` instead. Call before `I2S::begin()`

12.1.7 bool setDOUT(pin_size_t pin)

Sets the DOUT pin of the I2S device. Any pin may be used. Call before `I2S::begin()`

12.1.8 bool setDIN(pin_size_t pin)

Sets the DIN pin of the I2S device. Any pin may be used. Call before `I2S::begin()`

12.1.9 bool setMCLK(pin_size_t pin)

Sets the MCLK pin of the I2S device and enables MCLK output. Any pin may be used. Call before `I2S::begin()`

12.1.10 bool setMCLKmult(int mult)

Sets the sample rate to MCLK multiplier value. Only multiples of 64 are valid. Call before `I2S::begin()`

12.1.11 bool setBitsPerSample(int bits)

Specify how many bits per audio sample to read or write. Note that for 24-bit samples, audio samples must be left-aligned (i.e. bits 31…8). Call before `I2S::begin()`

12.1.12 bool setBuffers(size_t buffers, size_t bufferWords, int32_t silenceSample = 0)

Set the number of DMA buffers and their size in 32-bit words as well as the word to fill when no data is available to send to the I2S hardware. Call before `I2S::begin()`.

12.1.13 bool setFrequency(long sampleRate)

Sets the word clock frequency, but does not start the I2S device if not already running. May be called after `I2S::begin()` to change the sample rate on-the-fly.

12.1.14 bool setSysClk(int samplerate)

Changes the system clock to optimise for the desired samplerate for well known sample rates. If the sample rate is not included in the scan in `I2SBrute.ino`, the sysclock will remain unchanged. Note that using `setSysClk()` may affect the timing of other sysclk-dependent functions. Should be called before any I2S functions and any other sysclk dependent initialisations.

12.1.15 bool setLSBJFormat()

Enables LSB-J format for I2S output. In this mode the MSB comes out at the same time as the LRCLK changes, and not the normal 1-cycle delay. Useful for DAC chips like the PT8211.

12.1.16 bool setTDMFormat()

Enabled TDM formatted multi-channel output. Be sure to set the number of channels to the expected value (8 normally) and the bits per sample to 32.

12.1.17 bool setTDMChannels(int channels)

Sets the number of TDM channels between frame syncs. Generally should be set to 8.

12.1.18 bool swapClocks()

Certain boards are hardwired with the WCLK before the BCLK, instead of the normal way around. This call swaps the WCLK and BCLK pins. Note that you still call `setBCLK(x)` with `x` being the lowest pin ID (i.e. in swapClocks mode the `setBCLK` call actually sets LRCLK).

12.1.19 bool begin()/begin(long sampleRate)

Start the I2S device up with the given sample rate, or with the value set using the prior `setFrequency` call.

12.1.20 void end()

Stops the I2S device.

12.1.21 void flush()

Waits until all the I2S buffers have been output.

12.1.22 void getOverUnderflow()

Returns a flag indicating if the I2S system ran our of data to send on output, or had to throw away data on input.

12.1.23 void getOverflow()

Returns a flag indicating if the I2S system had to throw away data on input.

12.1.24 void getUnderflow()

Returns a flag indicating if the I2S system ran our of data to send on output.

12.1.25 size_t write(uint8_t/int8_t/int16_t/int32_t)

Writes a single sample of `bitsPerSample` to the buffer. It is up to the user to keep track of left/right channels. Note this writes data equivalent to one channel's data, not the size of the passed in variable (i.e. if you have a 16-bit sample size and `write((int8_t)-5); write((int8_t)5);` you will have written **2 samples** to the I2S buffer of whatever the I2S size, not a single 16-bit sample.

This call will block (wait) until space is available to actually write the data.

12.1.26 size_t write(int32_t val, bool sync)

Writes 32 bits of data to the I2S buffer (regardless of the configured I2S bit size). When `sync` is true, it will not return until the data has been writte. When `sync` is false, it will return `0` immediately if there is no space present in the I2S buffer.

12.1.27 size_t write(const uint8_t *buffer, size_t size)

Transfers number of bytes from an application buffer to the I2S output buffer. Be aware that `size` is in *bytes** and not samples. Size must be a multiple of **4 bytes**. Will not block, so check the return value to find out how many 32-bit words were actually written.

12.1.28 int availableForWrite()

Returns the amount of bytes that can be written without potentially blocking.

12.1.29 int read()

Reads a single sample of I2S data, whatever the I2S sample size is configured. Will not return until data is available.

12.1.30 int peek()

Returns the next sample to be read from the I2S buffer (without actually removing it).

12.1.31 size_t read(uint8_t *buffer, size_t size)

Transfers number of bytes from the I2S input buffer to an application buffer. Be aware that `size` is in *bytes** and not samples. Size must be a multiple of **4 bytes**. Will not block, so check the return value to find out how many 32-bit words were actually read.

12.1.32 void onTransmit(void (*fn)(void))

Sets a callback to be called when an I2S DMA buffer is fully transmitted. Will be in an interrupt context so the specified function must operate quickly and not use blocking calls like delay() or write to the I2S.

12.1.33 void onReceive(void (*fn)(void))

Sets a callback to be called when an I2S DMA buffer is fully read in. Will be in an interrupt context so the specified function must operate quickly and not use blocking calls like delay() or read from the I2S.

12.2 Sample Writing/Reading API

Because I2S streams consist of a natural left and right sample, it is often convenient to write or read both with a single call. The following calls allow applications to read or write both samples at the same time, and explicitly indicate the bit widths required (to avoid potential issues with type conversion on calls).

12.2.1 size_t write8(int8_t l, int8_t r)

Writes a left and right 8-bit sample to the I2S buffers. Blocks until space is available.

12.2.2 size_t write16(int16_t l, int16_t r)

Writes a left and right 16-bit sample to the I2S buffers. Blocks until space is available.

12.2.3 size_t write24(int32_t l, int32_t r)

Writes a left and right 24-bit sample to the I2S buffers. See note below about 24-bit mode. Blocks until space is available.

12.2.4 size_t write32(int32_t l, int32_t r)

Writes a left and right 32-bit sample to the I2S buffers. Blocks until space is available.

12.2.5 bool read8(int8_t *l, int8_t *r)

Reads a left and right 8-bit sample and returns `true` on success. Will block until data is available.

12.2.6 bool read16(int16_t *l, int16_t *r)

Reads a left and right 16-bit sample and returns `true` on success. Will block until data is available.

12.2.7 bool read24(int32_t *l, int32_t *r)

Reads a left and right 24-bit sample and returns `true` on success. See note below about 24-bit mode. Will block until data is available.

12.2.8 bool read32(int32_t *l, int32_t *r)

Reads a left and right 32-bit sample and returns `true` on success. Will block until data is available.

12.3 Note About 24-bit Samples

24-bit samples are stored as left-aligned 32-bit values with bits 7..0 ignored. Only the upper 24 bits 31…8 will be transmitted or received. The actual I2S protocol will only transmit or receive 24 bits in this mode, even though the data is 32-bit packed.

CHAPTER

THIRTEEN

PWM AUDIO LIBRARY

Relatively good quality analog audio out can be generated by using the RP2040 onboard PWM hardware. It can drive an amplifier for speaker output, or be decoupled using a capacitor to drive a headphone-level signal. Mono and stereo signals can be generated.

All samples are sent to the `PWMAudio` library as **signed 16 bits per sample**. Due to frequency limitations of the PWM hardware, at higher bit rates these 16-bits will automatically be reduced to the maximum the hardware can handle.

Multiple `PWMAudio` devices are supported, depending on availability of DMA channels.

The interface for the `PWMAudio` device is very similar to the `I2S` device, and most code can be ported simply by instantiating a `PWMAudio` object in lieu of an `I2S` object.

13.1 PWM Class API

13.1.1 PWMAudio(pin)

Creates a mono PWM output port. Any pin can be used, but no `analogWrite` calls are allowed to any other pins using that pin's PWM slice hardware. See the RP2040 datasheet for more details about PWM slices.

13.1.2 PWMAudio(pin, true)

Creates a stereo PWM output port. Only even pins (left signal) can be used, the next odd pin will automatically be assigned to the right channel (i.e. `PWMAudio pwm(0, true);` will make GP0 as the left channel, GP1 as the right channel). The same restriction as in mono mode applies.

13.1.3 bool setBuffers(size_t buffers, size_t bufferWords)

Set the number of DMA buffers and their size in 32-bit words. Call before `PWMAudio::begin()`.

When running at high sample rates, it is recommended to increase the `bufferWords` to 32 or higher (i.e. `pwm.setBuffers(4, 32);`).

13.1.4 bool setPin(pin_size_t pin)

Adjusts the pin to connect to the PWM audio output. Only legal before `PWMAudio::begin()`.

13.1.5 bool setStereo(bool stereo)

Adjusts the mono/stereo setting of the PWM audio output. Only legal before `PWMAudio::begin()`.

13.1.6 bool setFrequency(long sampleRate)

Sets the sample frequency, but does not start the PWM device (however if the device was already running, it will wontinue to run at the new frequency).

13.1.7 bool begin()/begin(long sampleRate)

Start the PWM Audio device up with the given sample rate, or with the value set using the prior `setFrequency` call.

13.1.8 void end()

Stops the PWMAudio device.

13.1.9 void flush()

Waits until all the PWM Audio buffers have been output.

13.1.10 size_t write(int16_t sample, bool sync = true)

Writes a single 16-bit sample to the buffer. It is up to the user to keep track of left/right channels when in stereo mode. In mono mode, one sample is written per timestep while in stereo mode two `write()` calls are required.

This call will block (wait) until space is available to actually write the data if `sync` is not specified or set to `true`.

13.1.11 size_t write(const uint8_t *buffer, size_t size)

Transfers number of bytes from an application buffer to the PWM Audio output buffer. Be aware that `size` is in *bytes** and not samples. Size must be a multiple of **4 bytes**. Will not block, so check the return value to find out how many bytes were actually written.

13.1.12 int availableForWrite()

Returns the number of samples that can be written without potentially blocking.

13.1.13 void onTransmit(void (*fn)(void))

Sets a callback to be called when a PWM Audio DMA buffer is fully transmitted. Will be in an interrupt context so the specified function must operate quickly and not use blocking calls like delay() or write to the PWM Audio.

CHAPTER

FOURTEEN

ADC INPUT LIBRARY

The ADC pins can be sampled and recorded by an application using the same interface as the I2S or PWM Audio libraries. This allows analog devices which need to be periodically sampled to be read by applications, easily, such as:

- Analog electret microphones
- Potentiometers
- Light dependent resistors (LDR), etc.

Up to 4 (or 8 in the case of the RP2350B) analog samples can be recorded by the hardware (`A0` ... `A3`), and all recording is done at 16-bit levels (but be aware that the ADC in the Pico will only ever return values between 0...4095).

The interface for the `ADCInput` device is very similar to the `I2S` input device, and most code can be ported simply by instantiating a `ADCInput` object in lieu of an `I2S` input object and choosing the pins to record.

Since this uses the ADC hardware, no `analogRead` or `analogReadTemp` calls are allowed while in use.

14.1 ADC Input API

14.1.1 ADCInput(pin0 [, pin1, pin2, pin3[, pin4, pin5, pin6, pin7])

Creates an ADC input object which will record the pins specified in the code. Only pins `A0` ... `A3` (`A7` on RP2350B) can be used, and they must be specified in increasing order (i.e. `ADCInput(A0, A1);` is valid, but `ADCInput(A1, A0)` is not.

14.1.2 bool setBuffers(size_t buffers, size_t bufferWords)

Set the number of DMA buffers and their size in 32-bit words. Call before `ADCInput::begin()`.

When running at high sample rates, it is recommended to increase the `bufferWords` to 32 or higher (i.e. `adcinput.setBuffers(4, 32);`).

14.1.3 bool setPins(pin_size_t pin [, pin1, pin2, pin3])

Adjusts the pin to record. Only legal before `ADCInput::begin()`.

14.1.4 bool setFrequency(long sampleRate)

Sets the ADC sampling frequency, but does not start recording (however if the device was already running, it will continue to run at the new frequency). Note that every pin requested will be sampled at this frequency, one after the other. That is, if you have code like this:

```
ADCInput adc(A0, A1);
adc.setFrequency(1000);
```

`A0` will be sampled at 0ms, 1ms, 2ms, etc. and `A1` will be sampled at 0.5ms 1.5ms, 2.5ms, etc. Each input is sampled at the proper frequency but offset in time since there is only one active ADC at a time.

The ADC clock divider limits the slowest rate. The sample rate times the number of pins must be at least 733Hz, otherwise `setFrequency` and `begin(sampleRate)` return `false`.

14.1.5 bool begin()/begin(long sampleRate)

Start the ADC input up with the given sample rate, or with the value set using the prior `setFrequency` call.

14.1.6 void end()

Stops the ADC Input device.

14.1.7 int read()

Reads a single sample of recorded ADC data, as a 16-bit value. When multiple pins are recorded the first read will be pin 0, the second will be pin 1, etc. Applications need to keep track of which pin is being returned (normally by always reading out all pins at once). Will not return until data is available.

14.1.8 int available()

Returns the number of samples that can be read without potentially blocking.

14.1.9 void onReceive(void (*fn)(void))

Sets a callback to be called when a ADC input DMA buffer is fully filled. Will be in an interrupt context so the specified function must operate quickly and not use blocking calls like delay().

CHAPTER

FIFTEEN

CAMERA (PICOCAMERA) LIBRARY

The `PicoCamera` library adds support for camera sensors with a parallel (DVP) interface, such as the Omnivision OV2640, OV3660, and OV7670, and the Galaxycore GC2145 and GC0308. While the RP2040/RP2350 chips do not include a dedicated camera peripheral, the library uses a PIO (Programmable I/O) state machine together with DMA to capture frames in the background with almost no CPU involvement.

The API is intentionally shaped after the well-known `esp32-camera` library: the camera is configured once at init time through a `camera_config_t` structure, frames are captured into pre-allocated buffers which the application borrows and returns, and the sensor can be adjusted at runtime through a set of control function pointers.

RGB565, YUV422 (packed YUYV) and grayscale raw frames are supported, as well as JPEG frames from sensors with an on-chip JPEG encoder, such as the OV2640 and OV3660.

Include the library header and define the configuration before use:

```
#include <PicoCamera.h>
```

15.1 Configuration (camera_config_t)

Everything the driver needs is supplied once, at init time, through `camera_config_t`. There is no board-level hard-coding. A complete configuration looks like this:

```
camera_config_t config = {
    .pin_pwdn     = -1,
    .pin_reset    = -1,
    .pin_xclk     = 10,
    .pin_sccb_sda = 12,
    .pin_sccb_scl = 13,
    .pin_d0 = 2, .pin_d1 = 3, .pin_d2 = 4, .pin_d3 = 5,
    .pin_d4 = 6, .pin_d5 = 7, .pin_d6 = 8, .pin_d7 = 9,
    .pin_vsync = 11,
    .pin_href  = 14,
    .pin_pclk  = 15,
    .xclk_freq_hz = 10000000,
    .sccb_i2c_port = 0,
    .pixel_format  = PIXFORMAT_JPEG,
    .frame_size    = FRAMESIZE_QVGA,
    .jpeg_quality  = 12,
    .fb_count      = 2,
    .fb_location   = PICO_CAMERA_FB_AUTO,
};
```

The most important fields are:

- `pin_pwdn` / `pin_reset` - camera power-down and reset GPIOs. Set to `-1` if not connected.
- `pin_xclk` - master clock (XCLK) output GPIO, driven by a PWM slice.
- `pin_sccb_sda` / `pin_sccb_scl` - SCCB (I2C) data and clock GPIOs, used to detect and configure the sensor. Pins are hard-muxed: SDA must be even, SCL odd, both routing to `sccb_i2c_port`. Set `pin_sccb_sda = -1` to reuse an I2C bus already initialized by the application (see the shared bus note below); `pin_sccb_scl` is then ignored.
- `pin_d0` … `pin_d7` - the 8 parallel data lines. These **must be 8 consecutive GPIOs** (`pin_dN == pin_d0 + N`, a PIO hardware constraint; validated at init).
- `pin_vsync` / `pin_href` / `pin_pclk` - frame sync, line sync and pixel clock inputs. Any free GPIO may be used.
- `xclk_freq_hz` - XCLK frequency in Hz. `0` selects the default of 10 MHz; 10-24 MHz is the typical sensor range.
- `sccb_i2c_port` - the I2C peripheral used for SCCB, `0` or `1`. In shared-bus mode (`pin_sccb_sda = -1`) it selects which already initialized bus to reuse.
- `pixel_format` - `PIXFORMAT_RGB565`, `PIXFORMAT_YUV422`, `PIXFORMAT_GRAYSCALE` or `PIXFORMAT_JPEG`. YUV422 streams packed YUYV (2 bytes per pixel, supported on OV2640/OV3660/OV7670/GC2145/GC0308); grayscale is Y-only (1 byte per pixel, GC0308). JPEG requires a sensor with an on-chip encoder (OV2640/OV3660 yes, the others no).
- `frame_size` - a `FRAMESIZE_*` value. Sizes beyond the sensor maximum are clamped with a warning instead of failing. Available sizes include 96x96, QQVGA (160x120), QCIF (176x144), HQVGA (240x176), 240x240, QVGA (320x240), CIF (400x296), HVGA (480x320), VGA (640x480), SVGA (800x600), XGA (1024x768), HD (1280x720), SXGA (1280x1024), and UXGA (1600x1200).
- `jpeg_quality` - 0-63, **lower means higher quality**. JPEG mode only.
- `fb_count` - number of frame buffers to allocate.
- `fb_location` - where frame buffers live. `PICO_CAMERA_FB_AUTO` (the default) uses PSRAM when available and SRAM otherwise, `PICO_CAMERA_FB_IN_PSRAM` forces PSRAM (init fails if unavailable), and `PICO_CAMERA_FB_IN_SRAM` forces on-chip SRAM.

15.2 Shared SCCB (I2C) Bus

To put the camera sensor on an I2C bus already used for other devices, initialize that bus yourself (`Wire.begin()` or `Wire1.begin()`) and then set `config.pin_sccb_sda = -1` and `config.sccb_i2c_port` to the matching port. The library skips all pin and bus setup in this mode and never deinitializes a shared bus at `pico_camera_deinit()`.

15.3 Memory Considerations

Frame buffers are the largest consumer of memory in this library, so it is worth planning ahead. The RP2040 has 264 KB of on-chip SRAM, while the RP2350 has 520 KB; the frame buffers, the rest of your sketch, and the Arduino core itself all share this space.

For RGB565 and YUV422 frames each buffer needs `width * height * 2` bytes (grayscale needs half of that):

- QQVGA (160x120) - about 38 KB per buffer.
- QVGA (320x240) - about 150 KB per buffer, near the practical ceiling on the RP2040.
- VGA (640x480) - 600 KB per buffer, does not fit in SRAM on either chip.
- SVGA (800x600) - about 940 KB per buffer.

For JPEG frames each buffer is allocated as `width * height / 4 + 8 KB`, which covers typical scenes (the actual JPEG data is usually smaller, so only part of the buffer is used):

- QVGA (320x240) - about 27 KB per buffer.
- VGA (640x480) - about 83 KB per buffer.
- SVGA (800x600) - about 125 KB per buffer.
- UXGA (1600x1200) - about 477 KB per buffer.

Boards with a PSRAM chip (available on many RP2350 boards) have a much more generous allowance: when the core's PSRAM support is enabled, the frame buffers can be placed in PSRAM (see `fb_location` above), making large RGB565 frames and multi-megapixel JPEG captures practical.

15.4 Initializing the Camera

With the configuration filled in, start the camera by calling `pico_camera_init()` once:

```
int err = pico_camera_init(&config);
if (err != PICO_CAMERA_OK) {
    // no sensor found, bad pin layout, out of memory, ...
    // see the Error Codes section below
}
```

This probes the sensor over SCCB, validates the pin layout, allocates the frame buffers, and arms the PIO + DMA capture engine. Afterwards frames can be captured with `pico_camera_fb_get()`. Call `pico_camera_deinit()` before re-initializing with a different configuration.

15.5 The Frame Buffer (camera_fb_t)

Captured frames are delivered in a `camera_fb_t` structure:

```
typedef struct {
    uint8_t *buf;        // pixel data
    size_t   len;        // used bytes in buf
    size_t   width;      // pixels
    size_t   height;     // pixels
    pixformat_t format;  // PIXFORMAT_RGB565 / YUV422 / GRAYSCALE / JPEG
    struct timeval timestamp;  // capture time since boot
} camera_fb_t;
```

For RGB565 and YUV422 frames, `len == width * height * 2` (YUV422 is packed YUYV: one luminance byte per pixel, chroma shared per pixel pair). For grayscale frames, `len == width * height`. For JPEG frames, `buf` holds a complete, standalone JPEG file (starting with `0xFF 0xD8` and ending with `0xFF 0xD9`) and `len` varies per frame; it can be written straight to a file or socket.

With `fb_count > 1` the DMA engine can fill the next buffer while the application still processes the previous one. With `fb_count == 1` each `pico_camera_fb_get()` waits for a new frame to be captured after the previous buffer was returned.

15.6 Reading and Processing Frames

The capture loop is always the same shape: borrow a frame buffer with `pico_camera_fb_get()`, do something with the pixels, then give the buffer back with `pico_camera_fb_return()`. The following `loop()`, adapted from the

push_image_to_python example shipping with the library, grabs an RGB565 frame and streams it raw over the USB serial port; the spot marked in the middle is where your own processing (image filtering, feature detection, driving a display, and so on) goes:

```
void loop() {
    camera_fb_t *fb = pico_camera_fb_get();
    if (fb) {
        // fb->buf holds fb->width * fb->height 16-bit RGB565 pixels.
        // Process the frame here, e.g. walk the pixels:
        //
        //   uint16_t *pixels = (uint16_t *)fb->buf;
        //   for (size_t i = 0; i < fb->width * fb->height; i++) {
        //       uint16_t pixel = pixels[i];
        //       ...
        //   }

        // Stream the raw frame to the PC, framed with start/end markers
        Serial.write("SRGB", 4);
        Serial.write(fb->buf, fb->len);
        Serial.write("ERGB", 4);
        Serial.flush();  // wait until the whole frame left the FIFO

        pico_camera_fb_return(fb);
    }
    delay(10);
}
```

A matching Python receiver (push_image_to_python.py) that displays the stream is included next to the example in libraries/PicoCamera/examples/push_image_to_python.

15.7 Runtime Sensor Control

pico_camera_sensor_get() returns the detected sensor's control structure. All controls are function pointers, and **a sensor only implements a subset of them, so always NULL-check before calling**:

```
sensor_t *s = pico_camera_sensor_get();
if (s && s->set_vflip) {
    s->set_vflip(s, 1);
}
```

Available operations include:

- set_pixformat(fmt) - switch between RGB565, YUV422, grayscale and JPEG at runtime (whichever the sensor supports).
- set_framesize(size) - change resolution at runtime.
- set_brightness(level) / set_contrast(level) / set_saturation(level) / set_sharpness(level) - image tuning, small-integer levels (on the OV2640, brightness/contrast/saturation accept -2…2).
- set_gainceiling(gc) - AGC ceiling, GAINCEILING_2X … GAINCEILING_128X.
- set_quality(q) - JPEG quality 0-63, lower is better.
- set_colorbar(on) - enable a test color-bar pattern.
- set_whitebal(on) / set_wb_mode(mode) - auto white balance switch / white balance preset mode.

- `set_gain_ctrl(on)` / `set_exposure_ctrl(on)` - AGC / AEC switch.
- `set_ae_level(level)` / `set_aec_value(value)` - auto-exposure target level / manual exposure value.
- `set_hmirror(on)` / `set_vflip(on)` - horizontal mirror / vertical flip.
- `set_special_effect(effect)` - effect index (grayscale, negative, etc.).
- `set_reg(reg, mask, value)` / `get_reg(reg, mask)` - raw register write/read, where `mask` selects which bits to touch. This is the escape hatch for anything not wrapped above.
- `set_xclk(timer, xclk)` - change the XCLK frequency at runtime (`timer` is unused).
- `reset()` - sensor soft reset.

Not every sensor implements every operation. The OV2640 implements the full set; the OV7670 has no JPEG encoder and lacks the image-tuning controls; the OV3660 falls in between; the Galaxycore sensors (GC2145, GC0308) have no JPEG encoder and expose the basics (pixformat, framesize, hmirror, vflip, raw register access). Always check the function pointer for `NULL` before calling, as shown above.

15.8 Shutting Down the Camera

When the camera is no longer needed, release it with `pico_camera_deinit()`:

```
pico_camera_deinit();
```

This stops the PIO + DMA capture engine, frees all frame buffers, and releases the SCCB bus. A shared bus (passed in with `pin_sccb_sda = -1`) is left running, since the library did not set it up. Afterwards the camera can be re-initialized with a fresh configuration by calling `pico_camera_init()` again.

15.9 Camera API

15.9.1 int pico_camera_init(const camera_config_t *config)

Probes the sensor over SCCB, validates the pin layout, allocates the frame buffers, and arms the PIO + DMA capture engine. Returns `PICO_CAMERA_OK` on success or one of the error codes listed below. Call it once; call `pico_camera_deinit()` before re-initializing with a different configuration.

15.9.2 camera_fb_t *pico_camera_fb_get()

Blocks until one full frame has been captured by DMA into a buffer and returns that buffer. Returns `NULL` on error or when no free buffer is available.

15.9.3 void pico_camera_fb_return(camera_fb_t *fb)

Returns a frame buffer to the driver for reuse. Always call this when done with a frame; buffers are recycled, and not returning them starves the driver once all `fb_count` buffers are checked out.

15.9.4 int pico_camera_deinit()

Stops the PIO/DMA engine, releases the SCCB bus (unless shared), and frees all frame buffers.

15.9.5 sensor_t *pico_camera_sensor_get()

Returns the detected sensor's runtime control structure, or `NULL` before init. See the runtime sensor control section above.

15.9.6 const sensor_info_t *pico_camera_sensor_info_get()

Returns static information about the detected sensor model: `name`, `sccb_addr`, `pid`, `max_size` (largest `framesize_t` supported), and `support_jpeg`. Useful for adapting the configuration at runtime.

15.9.7 const char *pico_camera_err_str(int err)

Maps an error code to a human-readable string, handy for logging: `Serial.printf("init failed: %s\n", pico_camera_err_str(err));`

15.10 Error Codes

`pico_camera_init()` and `pico_camera_deinit()` return one of the following (pass the code to `pico_camera_err_str()` for a printable description):

- `PICO_CAMERA_OK` (0) - success.
- `PICO_CAMERA_ERR_NOT_DETECTED` - no sensor answered on SCCB.
- `PICO_CAMERA_ERR_NOT_SUPPORTED` - e.g. JPEG requested on an OV7670.
- `PICO_CAMERA_ERR_INVALID_ARG` - bad configuration, such as non-consecutive data pins.
- `PICO_CAMERA_ERR_INVALID_STATE` - init called twice, or deinit without init.
- `PICO_CAMERA_ERR_NO_MEM` - frame buffer allocation failed.
- `PICO_CAMERA_ERR_TIMEOUT` - capture timed out.
- `PICO_CAMERA_ERR_FAILED_TO_SET_FRAME_SIZE` / `PICO_CAMERA_ERR_FAILED_TO_SET_OUT_FORMAT` - the sensor rejected the requested format or size.

15.11 Example Sketch

The following sketch captures JPEG frames from an OV2640 and prints the frame size over the serial port:

```
#include <PicoCamera.h>

camera_config_t config = {
    .pin_pwdn     = -1,
    .pin_reset    = -1,
    .pin_xclk     = 10,
    .pin_sccb_sda = 12,
    .pin_sccb_scl = 13,
    .pin_d0 = 2, .pin_d1 = 3, .pin_d2 = 4, .pin_d3 = 5,
    .pin_d4 = 6, .pin_d5 = 7, .pin_d6 = 8, .pin_d7 = 9,
    .pin_vsync = 11,
    .pin_href  = 14,
    .pin_pclk  = 15,
    .xclk_freq_hz = 10000000,
    .sccb_i2c_port = 0,
    .pixel_format  = PIXFORMAT_JPEG,
```

(continues on next page)

(continued from previous page)

```
    .frame_size    = FRAMESIZE_QVGA,
    .jpeg_quality  = 12,
    .fb_count      = 2,
    .fb_location   = PICO_CAMERA_FB_AUTO,
};

void setup() {
    Serial.begin(115200);
    int err = pico_camera_init(&config);
    if (err != PICO_CAMERA_OK) {
        Serial.printf("Camera init failed: %s (%d)\n",
                      pico_camera_err_str(err), err);
        while (true) { /* halt */ }
    }
}

void loop() {
    camera_fb_t *fb = pico_camera_fb_get();
    if (fb) {
        Serial.printf("Frame: %dx%d, %d bytes\n",
                      (int)fb->width, (int)fb->height, (int)fb->len);
        pico_camera_fb_return(fb);
    }
}
```

Additional examples, including rendering frames to a TFT display and streaming images to a host PC, ship with the library under `libraries/PicoCamera/examples`.

For more information, see the PicoCamera project home page at https://github.com/umeiko/PicoCamera

CHAPTER

SIXTEEN

SERIAL PORTS (USB, UART, AND BLE "NORDIC SPP SERVICE")

The Arduino-Pico core implements a software-based Serial-over-USB port using the USB ACM-CDC model to support a wide variety of operating systems.

`Serial` is the USB serial port, and while `Serial.begin()` does allow specifying a baud rate, this rate is ignored since it is USB-based. (Also be aware that this USB `Serial` port is responsible for resetting the RP2040 during the upload process, following the Arduino standard of 1200bps = reset to bootloader).

The RP2040 provides two hardware-based UARTS with configurable pin selection.

`Serial1` is `UART0`, and `Serial2` is `UART1`.

Configure their pins using the `setXXX` calls prior to calling `begin()`

```
Serial1.setRX(pin);
Serial1.setTX(pin);
Serial1.begin(baud);
```

Pass in `-1` to `setRX` or `setTX` to disable that pin, freeing it for use by your application.

The size of the receive FIFO may also be adjusted from the default 32 bytes by using the `setFIFOSize` call prior to calling `begin()`

```
Serial1.setFIFOSize(128);
Serial1.begin(baud);
```

The FIFO is normally handled via an interrupt, which reduced CPU load and makes it less likely to lose characters.

For applications where an IRQ driven serial port is not appropriate, use `setPollingMode(true)` before calling `begin()`

```
Serial1.setPollingMode(true);
Serial1.begin(300)
```

For detailed information about the Serial ports, see the Arduino Serial Reference .

16.1 Inversion

`Serial1` and `Serial2` can both support inverted input and/or outputs via the methods `Serial1/2::setInvertRX(bool invert)` and `Serial1/2::setInvertTX(bool invert)` and `Serial1/2::serInvertControl(bool invert)`.

16.2 Nordic SPP Serial Service

Please see the `File->Examples->BLE->SerialBLE.ino` example for use over BLE.

16.3 RP2040 Specific SerialUSB methods

16.3.1 void Serial.ignoreFlowControl(bool ignore)

In some cases, the target application will not assert the DTR virtual line, thus preventing writing operations to succeed.

For this reason, the SerialUSB::ignoreFlowControl() method disables the connection's state verification, enabling the program to write on the port, even though the data might be lost.

16.3.2 bool Serial.dtr()

Returns the current state of the DTR virtual line. A USB CDC host (such as the Arduino serial monitor) typically raises the DTR pin when opening the device, and may lower it when closing the device.

16.3.3 bool Serial.rts()

Returns the current state of the RTS virtual line.

CHAPTER

SEVENTEEN

"SOFTWARESERIAL" PIO-BASED UART

Equivalent to the Arduino SoftwareSerial library, an emulated UART using one or two PIO state machines is included in the Arduino-Pico core. This allows for up to 4 bidirectional or up to 8 unidirectional serial ports to be run from the RP2040 without requiring additional CPU resources.

Instantiate a `SerialPIO(txpin, rxpin, fifosize)` object in your sketch and then use it the same as any other serial port. Even, odd, and no parity modes are supported, as well as data sizes from 5- to 8-bits. Fifosize, if not specified, defaults to 32 bytes.

To instantiate only a serial transmit or receive unit, pass in `NOPIN` as the `txpin` or `rxpin`.

For example, to make a transmit-only port on GP16

```
SerialPIO transmitter( 16, NOPIN );
```

For detailed information about the Serial ports, see the Arduino Serial Reference .

17.1 Inversion

`SoftwareSerial` and `SerialPIO` can both support inverted input and/or outputs via the methods `setInvertRX(bool invert)` and `setInvertTX(bool invert)`.

CHAPTER

EIGHTEEN

SOFTWARESERIAL EMULATION

A `SoftwareSerial` wrapper is included to provide plug-and-play compatibility with the Arduino Software Serial library. Use the normal `#include <SoftwareSerial.h>` to include it. The following differences from the Arduino standard are present:

- All ports are always listening
- `listen` call is a no-op
- `isListening()` always returns `true`

CHAPTER

NINETEEN

SERVO LIBRARY

A hardware-based servo controller is provided using the `Servo` library. It utilizes the PIO state machines and generates the appropriate servo control pulses, glitch-free and jitter-free (within crystal limits).

Up to 8 Servos can be controlled in parallel assuming no other tasks require the use of a PIO machine.

See the Arduino standard Servo documentation for detailed usage instructions. There is also an included `sweep` example.

19.1 Pulse Width Defaults

The defaults in the Servo library are conservatively set to avoid damage in the case of over-driving. The pulse widths individual servos, especially the no-name or clones, occasionally need tweaking.

You can set the min and max servo pulse width in the attach command, with default values used in most Arduino cores of 540/2400: `` `myServo.attach(D3, 540, 2400)` ``

CHAPTER

TWENTY

SPI MASTER (SERIAL PERIPHERAL INTERFACE)

The RP2040 has two hardware SPI interfaces, `spi0 (SPI)` and `spi1 (SPI1)`. These interfaces are supported by the `SPI` library in master mode.

SPI pinouts can be set **before SPI.begin()** using the following calls:

```
bool setRX(pin_size_t pin); // or setMISO()
bool setCS(pin_size_t pin);
bool setSCK(pin_size_t pin);
bool setTX(pin_size_t pin); // or setMOSI()
```

Note that the `CS` pin can be hardware or software controlled by the sketch. When software controlled, the `setCS()` call is ignored.

Passing in `NOPIN` to either `setRX` or `setTX` will disable SPI on that interface, making it behave as a unidirectional (output- or input-only SPI interface.

The Arduino SPI documentation gives a detailed overview of the library, except for the following RP2040-specific changes:

- **`SPI.begin(bool hwCS)` can take an options `hwCS` parameter.**
  By passing in `true` for `hwCS` the sketch does not need to worry about asserting and deasserting the `CS` pin between transactions. The default is `false` and requires the sketch to handle the CS pin itself, as is the standard way in Arduino.
- The interrupt calls (`attachInterrupt`, and `detachInterrpt`) are not implemented.

CHAPTER

TWENTYONE

SOFTWARE SPI (MASTER ONLY)

Similar to `SoftwareSerial`, `SoftwareSPI` creates a PIO based SPI interface that can be used in the same manner as the hardware SPI devices. The constructor takes the pins desired, which can be any GPIO pins with the rule that if hardware CS is used then it must be on pin `SCK + 1`. Construct a `SoftwareSPI` object in your code as follows and use it as needed (i.e. pass it into `SD.begin(_CS, softwareSPI);`

```
#include <SoftwareSPI.h>
SoftwareSPI softSPI(_sck, _miso, _mosi); // no HW CS support, any selection of pins can␣
↪be used
```

CHAPTER

TWENTYTWO

SPI SLAVE (SPISLAVE)

Slave mode operation is also supported on either SPI interface. Two callbacks are needed in your app, set through `SPISlave.onDataRecv` and `SPISlave.onDataSent`, in order to consunme the received data and provide data to transmit.

- The callbacks operate at IRQ time and may be called very frequently at high SPI frequencies. So, make then small, fast, and with no memory allocations or locking.

CHAPTER

TWENTYTHREE

ASYNCHRONOUS OPERATION

Applications can use asynchronous SPI calls to allow for processing while long-running SPI transfers are being performed. For example, a game could send a full screen update out over SPI and immediately start processing the next frame without waiting for the first one to be sent. DMA is used to handle the transfer to/from the hardware freeing the CPU from bit-banging or busy waiting.

Note that asynchronous operations can not be intersped with normal, synchronous ones. `transferAsync` should still occur after a `beginTransaction()` and when `finishedAsync()` returns `true` then `endTransaction()` should also be called.

All buffers need to be valid throughout the entire operation. Read data cannot be accessed until the transaction is completed and can't be "peeked" at while the operation is ongoing.

23.1 bool transferAsync(const void *send, void *recv, size_t bytes)

Begins an SPI asynchronous transaction. Either `send` or `recv` can be `nullptr` if data only needs to be transferred in one direction. Check `finishedAsync()` to determine when the operation completes and conclude the transaction. This operation needs to allocate a buffer from heap equal to `bytes` in size if `LSBMODE` is used.

23.2 bool finishedAsync()

Call to check if the asynchronous operations is completed and the buffer passed in can be either read or reused. Frees the allocated memory and completes the asynchronous transaction.

23.3 void abortAsync()

Cancels the outstanding asynchronous transaction and frees any allocated memory.

CHAPTER

TWENTYFOUR

EXAMPLES

See the SPItoMyself and SPItoMyselfAsync examples for a complete Master and Slave application.

CHAPTER

TWENTYFIVE

WIRE (I2C MASTER AND SLAVE)

The RP2040 has two I2C devices, `i2c0 (Wire)` and `i2c1 (Wire1)`.

The default pins for *Wire* and *Wire1* vary depending on which board you're using. (Here are the pinout diagrams for Pico and Adafruit Feather.)

You may change these pins **before calling Wire.begin() or Wire1.begin()** using:

```
bool setSDA(pin_size_t sda);
bool setSCL(pin_size_t scl);
```

Be sure to use pins labeled `I2C0` for Wire and `I2C1` for Wire1 on the pinout diagram for your board, or it won't work.

Other than that, the API is compatible with the Arduino standard. Both master and slave operation are supported.

Master transmissions are buffered (up to 256 bytes) and only performed on `endTransmission`, as is standard with modern Arduino Wire implementations.

For more detailed information, check the Arduino Wire documentation .

25.1 Asynchronous Operation

Applications can use asynchronous I2C calls to allow for processing while long-running I2C operations are being performed. For example, a game could send a full screen update out over I2C and immediately start processing the next frame without waiting for the first one to be sent over I2C. DMA is used to handle the transfer to/from the I2C hardware freeing the CPU from bit-banging or busy waiting.

Note that asynchronous operations can not be intersped with normal, synchronous ones. Fully complete or abort an asynchronous operation before attempting to do a normal `Wire.beginTransaction()` or `Wire.requestFrom`.

25.1.1 bool writeReadAsync(uint8_t address, const void *wbuffer, size_t wbytes, const void *rbuffer, size_t rbytes, bool sendStop)

Executes a master I2C asynchronous write/read transaction to I2C slave `address`. First `wbytes` from `wbuffer` are written to the I2C slave, followed by an I2C restart, then `rbytes` are read from the I2C slave into `rbuffer`. The buffers need to be valid throughout the entire asynchronous operation.

At the end of the transaction an I2C stop is sent if `sendStop` is `true`, and at the beginning of the transaction an I2C start is sent if the previous write/read had `sendStop` set to `true`.

Check `finishedAsync()` to determine when the operation completes, or use `onFinishedAsync()` to set a callback.

Set `rbytes` to 0 to do a write-only operation, set `wbytes` to 0 to do a read-only operation. Or use:

`bool writeAsync(uint8_t address, const void \*buffer, size_t bytes, bool sendStop)`

`bool readAsync(uint8_t address, void \*buffer, size_t bytes, bool sendStop)`

The first call to an asynchronous write/read operation allocates the required DMA channels and internal buffer. If desired, call `end()` to free these resources.

25.1.2 bool finishedAsync()

Call to check if the asynchronous operations is completed.

25.1.3 void onFinishedAsync(void(*function)(void))

Set a (optional) callback for async operation. The `function` will be called when the asynchronous operation finishes.

25.1.4 void abortAsync()

Cancels any outstanding asynchronous transaction.

CHAPTER

TWENTYSIX

FILE SYSTEMS

The Arduino-Pico core supports using some of the onboard flash as a file system, useful for storing configuration data, output strings, logging, and more. It also supports using SD cards as another (FAT32) filesystem, with an API that's compatible with the onboard flash file system.

26.1 Flash Layout

Even though file system is stored on the same flash chip as the program, programming new sketch will not modify file system contents (or EEPROM data).

The following diagram shows the flash layout used in Arduino-Pico:

```
|----|--------------------|-------------|----|
^    ^                    ^             ^
OTA  Sketch               File system   EEPROM
```

The file system size is configurable via the IDE menus, from 64k up to 15MB (assuming you have an RP2040 board with that much flash)

Note: to use any of file system functions in the sketch, add the following include to the sketch:

```
#include "LittleFS.h" // LittleFS is declared
// #include <SDFS.h>
// #include <SD.h>
// #include <FatFS.h>
```

26.2 Compatible Filesystem APIs

LittleFS is an onboard filesystem that sets aside some program flash for use as a filesystem without requiring any external hardware.

SDFS is a filesystem for SD cards, based on [SdFat 2.0](https://github.com/earlephilhower/ESP8266SdFat). It supports FAT16 and FAT32 formatted cards, and requires an external SD card reader.

SD is the Arduino-supported, somewhat old and limited SD card filesystem. It is recommended to use SDFS for new applications instead of SD.

FatFS implements a wear-leveled, FTL-backed FAT filesystem in the onboard flash which can be easily accessed over USB as a standard memory stick via FatFSUSB.

All of these filesystems can open and manipulate `File` and `Dir` objects with the same code because the implement a common end-user filesystem API.

26.3 SDFS/SD/FatFS Multicore and Multithreading(FreeRTOS) Caveats

When using these filesystems and both cores (in No-OS mode) or multiple cores/tasks (FreeRTOS), only a single task or core may access the filesystem or any `File` objects without explicit application-level synchronization.

When in No-OS mode(bare metal) the LWIP callbacks are executed at interrupt time, effectively running in their own "quasi-task". This means that ONLY the main app OR LWIP callbacks (i.e. WebServer handlers) may access these filesystem, even for read-only applications. If your app tries to read a file in a WebServer callback which interrupts a file creation/write from the main `setup` or `loop` you will likely corrupt the filesystem state in memory and/or on flash causing a crash or data loss.

26.4 LittleFS Multicore and Multithreading(FreeRTOS) Caveats

LittleFS has been modified to be fully FreeRTOS `Task` safe and multiple tasks may use the `LittleFS` object and `Files`.

For bare metal operation, only *EITHER* LWIP(WebServer, etc.) callbacks *OR* the main `setup` and `loop` on core 0 may access the filesystem (similar to the SDFS caveats above), but core 1 `setup1` and `loop1` may also access `LittleFS` and `File` objects.

One example on bare metal would be to use `setup1` and `loop1` to write logs that are served through WebServer callbacks on core 0.

See the `SpeedTestMulti.ino` example for a demonstration in both non-OS and FreeRTOS modes.

26.5 FatFS File System Caveats and Warnings

The FAT filesystem is ubiquitous, but it is also around 50 years old and ill suited to SPI flash memory due to having "hot spots" like the FAT copies that are rewritten many times over. SPI flash allows a high, but limited, number of writes before losing the ability to write safely. Applications like data loggers where many writes occur could end up wearing out the SPI flash sector that holds the FAT **years** before coming close to the write limits of the data sectors.

To circumvent this issue, the FatFS implementation here uses a flash translation layer (FTL) developed for SPI flash on embedded systems. This allows for the same LBA to be written over and over by the FAT filesystem, but use different flash locations. For more information see [SPIFTL](https://github.com/earlephilhower/SPIFTL). In this mode the Pico flash appears as a normal, 512-byte sector drive to the FAT.

What this means, practically, is that about 5KB of RAM per megabyte of flash is required for housekeeping. Writes can also become very slow if most of the flash LBA range is used (i.e. if the FAT drive is 99% full) due to the need for garbage collection processes to move data around and preserve the flash lifetime.

Alternatively, if an FTL is not desired or memory is tight, FatFS can use the raw flash directly. In this mode sectors are 4K in size and flash is mapped 1:1 to sectors, so things like the FAT table updates will all use the same physical flash bits. For low-utilization operations this may be fine, but if significant writes are done (from the Pico or the PC host) this may wear out portions of flash very quickly , rendering it unusable.

26.6 LittleFS File System Limitations

The LittleFS implementation for the RP2040 supports filenames of up to 254 characters + terminating zero (i.e. `char filename[255]` or better `char filename[LFS_NAME_MAX]`), and as many subdirectories as space permits.

Filenames are assumed to be in the root directory if no initial "/" is present.

Opening files in subdirectories requires specifying the complete path to the file (i.e. `LittleFS.open("/sub/dir/file.txt", "r");`). Subdirectories are automatically created when you attempt to create a file in a subdirectory, and when the last file in a subdirectory is removed the subdirectory itself is automatically deleted.

26.7 Uploading Files to the LittleFS File System on IDE 1.x (RP2040 only)

For the Pico (RP2040) only, *PicoLittleFS* is a tool which integrates into the obsolete Arduino 1.x IDE. It adds a menu item to **Tools** menu for uploading the contents of sketch data directory into a new LittleFS flash file system. Please note that this JAVA plug in for the obsolete 1.x IDE does **NOT** support the RP2350 (Pico 2).

- Download the tool: https://github.com/earlephilhower/arduino-pico-littlefs-plugin/releases
- In your Arduino sketchbook directory, create `tools` directory if it doesn't exist yet.
- Unpack the tool into `tools` directory (the path will look like `<home_dir>/Arduino/tools/PicoLittleFS/tool/picolittlefs.jar`) If upgrading, overwrite the existing JAR file with the newer version.
- Restart Arduino IDE.
- Open a sketch (or create a new one and save it).
- Go to sketch directory (choose Sketch > Show Sketch Folder).
- Create a directory named `data` and any files you want in the file system there.
- Make sure you have selected a board, port, and closed Serial Monitor.
- Double check the Serial Monitor is closed. Uploads will fail if the Serial Monitor has control of the serial port.
- Select `Tools > Pico LittleFS Data Upload`. This should start uploading the files into the flash file system.

26.8 Uploading Files to the LittleFS File System on IDE 2.x (RP2040 and RP2350)

For the original Pico (RP2040) and Pico 2 (RP2350), use the Typescript 2.x IDE tool which operates in the new Arduino 2.x IDE. This tool works on all Pico and Pico 2s (and ESP32s if you're so inclined).

- Download the new tool: https://github.com/earlephilhower/arduino-littlefs-upload/releases
- Exit the IDE, if running
- Copy the VSIX file manually to (Linux/Mac) `~/.arduinoIDE/plugins/` (you may need to make this directory yourself beforehand) or to (Windows) `C:\Users\<username>\.arduinoIDE\`
- Restart the IDE
- Double check the Serial Monitor is closed. Uploads will fail if the Serial Monitor has control of the serial port.
- Enter (Linux/Windows) `[Ctrl]` + `[Shift]` + `[P]` or (Mac) `[Cmd]` + `[Shift]` + `[P]` to bring up the command palette, then select/type `Upload LittleFS to Pico/ESP8266`

26.9 Downloading Files from a LittleFS System

Using `gdb` it is possible to dump the flash data making up the filesystem and then extract it using the `mklittlefs` tool. A working `OpenOCD` setup, DebugProbe, and `gdb` are required. To download the raw filesystem, from within `GDB` run:

```
^C (break)
(gdb) dump binary memory littlefs.bin &_FS_start &_FS_end
```

It may take a few seconds as GDB reads out the flash to the file. Once the raw file is downloaded it can be extracted using the mklittlefs tool from the BASH/Powershell/command line

```
$ <path-to-mklittlefs>/mklittlefs -u output-dir littlefs.bin
 Directory <output-dir> does not exists. Try to create it.
 gmon.out    > <output-dir>/gmon.out    size: 24518 Bytes
 gmon.bak    > <output-dir>/gmon.bak    size: 1 Bytes
```

The defaults built into mklittlefs should be appropriate for normal LittleFS filesystems built on the device or using the upload tool.

26.10 SD Library Information

The included SD library is the Arduino standard one. Please refer to the [Arduino SD reference](https://www.arduino.cc/en/reference/SD) for more information.

26.11 Using Second SPI port for SD

The SD library begin() has been modified to allow you to use the second SPI port, SPI1. Just use the following call in place of SD.begin(cspin)

```
SD.begin(cspin, SPI1);
```

26.12 Enabling SDIO operation for SD

SDIO support is available thanks to SdFat implementing a PIO-based SDIO controller. This mode can significantly increase IO performance to SD cards but it requires that all 4 DAT0..DAT3 lines to be wired to the Pico (most SD breakout boards only provide 1-but SPI mode of operation).

To enable SDIO mode, simply specify the SD_CLK, SD_CMD, and SD_DAT0 GPIO pins. The clock and command pins can be any GPIO (not limited to legal SPI pins). The DAT0 pin can be any GPIO with remaining DAT1…3 pins consecutively connected.

..code:: cpp

    SD.begin(RP_CLK_GPIO, RP_CMD_GPIO, RP_DAT0_GPIO);

No other changes are required in the application to take advantage of this high performance mode.

26.13 Using VFS (Virtual File System) for POSIX support

The VFS library enables sketches to use standard POSIX file I/O operations using standard FILE * operations. Include the VFS library in your application and add a call to map the VFS.root() to your filesystem. I.e.:

```
#include <VFS.h>
#include <LittleFS.h>

void setup() {
  LittleFS.begin();
  VFS.root(LittleFS);
  FILE *fp = fopen("/thisfilelivesonflash.txt", "w");
  fprintf(fp, "Hello!\n");
```

(continues on next page)

(continued from previous page)

```
  fclose(fp);
}
```

Multiple filesystems can be `VFS.map()` into the VFS namespace under different directory names. For example, the following will make files on `/sd` reside on an externalSD card and files on `/lfs` live in internal flash.

```
#include <VFS.h>
#include <LittleFS.h>
#include <SDFS.h>

void setup() {
  LittleFS.begin();
  SDFS.begin();
  VFS.map("/lfs", LittleFS);
  VFS.map("/sd", SDFS);
  FILE *onSD = fopen("/sd/thislivesonsd.txt", "wb");
  ....
}
```

See the examples in the `VFS` library for more information.

26.14 File system object (LittleFS/SD/SDFS/FatFS)

26.14.1 setConfig

```
LittleFSConfig cfg;
cfg.setAutoFormat(false);
LittleFS.setConfig(cfg);

SDFSConfig c2;
c2.setCSPin(12);
SDFS.setConfig(c2);

FatFSConfig c3;
c3.setUseFTL(false); // Directly access flash memory
c3.setDirEntries(256); // We need 256 root directory entries on a format()
c3.setFATCopies(1); // Only 1 FAT to save 4K of space and extra writes
FatFS.setConfig(c3);
FatFS.format(); // Format using these settings, erasing everything
```

This method allows you to configure the parameters of a filesystem before mounting. All filesystems have their own `*Config` (i.e. `SDFSConfig` or `LittleFSConfig` with their custom set of options. All filesystems allow explicitly enabling/disabling formatting when mounts fail. If you do not call this `setConfig` method before perforing `begin()`, you will get the filesystem's default behavior and configuration. By default, LittleFS and FatFS will autoformat the filesystem if it cannot mount it, while SDFS will not. FatFS will also use the built-in FTL to support 512 byte sectors and higher write lifetime.

26.14.2 begin

```
SDFS.begin()
or LittleFS.begin()
```

This method mounts file system. It must be called before any other FS APIs are used. Returns *true* if file system was mounted successfully, false otherwise.

Note that LittleFS will automatically format the filesystem if one is not detected. This is configurable via `setConfig`.

26.14.3 end

```
SDFS.end()
or LittleFS.end()
```

This method unmounts the file system.

26.14.4 format

```
SDFS.format()
or LittleFS.format()
```

Formats the file system. May be called either before or after calling `begin`. Returns *true* if formatting was successful.

26.14.5 open

```
SDFS.open(path, mode)
or LittleFS.open(path, mode)
```

Opens a file. `path` should be an absolute path starting with a slash (e.g. `/dir/filename.txt`). `mode` is a string specifying access mode. It can be one of "r", "w", "a", "r+", "w+", "a+". The meaning of these modes is the same as for the `fopen` C function.

```
r      Open text file for reading.  The stream is positioned at the
       beginning of the file.

r+     Open for reading and writing.  The stream is positioned at the
       beginning of the file.

w      Truncate file to zero length or create text file for writing.
       The stream is positioned at the beginning of the file.

w+     Open for reading and writing.  The file is created if it does
       not exist, otherwise it is truncated.  The stream is
       positioned at the beginning of the file.

a      Open for appending (writing at end of file).  The file is
       created if it does not exist.  The stream is positioned at the
       end of the file.

a+     Open for reading and appending (writing at end of file).  The
       file is created if it does not exist.  The initial file
       position for reading is at the beginning of the file, but
       output is always appended to the end of the file.
```

Returns *File* object. To check whether the file was opened successfully, use the boolean operator.

```
File f = LittleFS.open("/f.txt", "w");
if (!f) {
```

(continues on next page)

(continued from previous page)

```
    Serial.println("file open failed");
}
```

26.14.6 exists

```
SDFS.exists(path)
or LittleFS.exists(path)
```

Returns *true* if a file with given path exists, *false* otherwise.

26.14.7 mkdir

```
SDFS.mkdir(path)
or LittleFS.mkdir(path)
```

Returns *true* if the directory creation succeeded, *false* otherwise.

26.14.8 rmdir

```
SDFS.rmdir(path)
or LittleFS.rmdir(path)
```

Returns *true* if the directory was successfully removed, *false* otherwise.

26.14.9 openDir

```
SDFS.openDir(path)
or LittleFS.openDir(path)
```

Opens a directory given its absolute path. Returns a *Dir* object.

26.14.10 remove

```
SDFS.remove(path)
or LittleFS.remove(path)
```

Deletes the file given its absolute path. Returns *true* if file was deleted successfully.

26.14.11 rename

```
SDFS.rename(pathFrom, pathTo)
or LittleFS.rename(pathFrom, pathTo)
```

Renames file from `pathFrom` to `pathTo`. Paths must be absolute. Returns *true* if file was renamed successfully.

26.14.12 info

```
FSInfo fs_info;
or LittleFS.info(fs_info);
```

Fills *FSInfo structure* with information about the file system. Returns `true` if successful, `false` otherwise. `info()` has been updated to support filesystems greater than 4GB and `FSInfo64` and `info64()` have been discarded.

26.15 Filesystem information structure

```
struct FSInfo {
    uint64_t totalBytes;
    uint64_t usedBytes;
    size_t blockSize;
    size_t pageSize;
    size_t maxOpenFiles;
    size_t maxPathLength;
};
```

This is the structure which may be filled using FS::info method. - `totalBytes` — total size of useful data on the file system - `usedBytes` — number of bytes used by files - `blockSize` — filesystem block size - `pageSize` — filesystem logical page size - `maxOpenFiles` — max number of files which may be open simultaneously - `maxPathLength` — max file name length (including one byte for zero termination)

26.15.1 setTimeCallback(time_t (*cb)(void))

```
time_t myTimeCallback() {
    return 1455451200; // UNIX timestamp
}
void setup () {
    LittleFS.setTimeCallback(myTimeCallback);
    ...
    // Any files will now be made with Pris' incept date
}
```

The SD, SDFS, and LittleFS filesystems support a file timestamp, updated when the file is opened for writing. By default, the Pico will use the internal time returned from `time(NULL)` (i.e. local time, not UTC, to conform to the existing FAT filesystem), but this can be overridden to GMT or any other standard you'd like by using `setTimeCallback()`. If your app sets the system time using NTP before file operations, then you should not need to use this function. However, if you need to set a specific time for a file, or the system clock isn't correct and you need to read the time from an external RTC or use a fixed time, this call allows you do to so.

In general use, with a functioning `time()` call, user applications should not need to use this function.

26.16 Directory object (Dir)

The purpose of *Dir* object is to iterate over files inside a directory. It provides multiple access methods.

The following example shows how it should be used:

```
Dir dir = LittleFS.openDir("/data");
// or Dir dir = LittleFS.openDir("/data");
while (dir.next()) {
    Serial.print(dir.fileName());
    if(dir.fileSize()) {
        File f = dir.openFile("r");
        Serial.println(f.size());
    }
}
```

26.16.1 next

Returns true while there are files in the directory to iterate over. It must be called before calling `fileName()`, `fileSize()`, and `openFile()` functions.

26.16.2 fileName

Returns the name of the current file pointed to by the internal iterator.

26.16.3 fileSize

Returns the size of the current file pointed to by the internal iterator.

26.16.4 fileTime

Returns the time_t write time of the current file pointed to by the internal iterator.

26.16.5 fileCreationTime

Returns the time_t creation time of the current file pointed to by the internal iterator.

26.16.6 isFile

Returns *true* if the current file pointed to by the internal iterator is a File.

26.16.7 isDirectory

Returns *true* if the current file pointed to by the internal iterator is a Directory.

26.16.8 openFile

This method takes *mode* argument which has the same meaning as for `SDFS/LittleFS.open()` function.

26.16.9 rewind

Resets the internal pointer to the start of the directory.

26.16.10 setTimeCallback(time_t (*cb)(void))

Sets the time callback for any files accessed from this Dir object via openNextFile. Note that the SD and SDFS filesystems only support a filesystem-wide callback and calls to `Dir::setTimeCallback` may produce unexpected behavior.

26.17 File object

`SDFS/LittleFS.open()` and `dir.openFile()` functions return a *File* object. This object supports all the functions of *Stream*, so you can use `readBytes`, `findUntil`, `parseInt`, `println`, and all other *Stream* methods.

There are also some functions which are specific to *File* object.

26.17.1 seek

```
file.seek(offset, mode)
```

This function behaves like `fseek` C function. Depending on the value of `mode`, it moves current position in a file as follows:

- if `mode` is `SeekSet`, position is set to `offset` bytes from the beginning.
- if `mode` is `SeekCur`, current position is moved by `offset` bytes.
- if `mode` is `SeekEnd`, position is set to `offset` bytes from the end of the file.

Returns *true* if position was set successfully.

26.17.2 position

```
file.position()
```

Returns the current position inside the file, in bytes.

26.17.3 size

```
file.size()
```

Returns file size, in bytes.

26.17.4 name

```
String name = file.name();
```

Returns short (no-path) file name, as `const char*`. Convert it to *String* for storage.

26.17.5 fullName

```
// Filesystem:
//   testdir/
//           file1
Dir d = LittleFS.openDir("testdir/");
File f = d.openFile("r");
// f.name() == "file1", f.fullName() == "testdir/file1"
```

Returns the full path file name as a `const char*`.

26.17.6 getLastWrite

Returns the file last write time, and only valid for files opened in read-only mode. If a file is opened for writing, the returned time may be indeterminate.

26.17.7 getCreationTime

Returns the file creation time, if available.

26.17.8 isFile

```
bool amIAFile = file.isFile();
```

Returns *true* if this File points to a real file.

26.17.9 isDirectory

```
bool amIADir = file.isDir();
```

Returns *true* if this File points to a directory (used for emulation of the SD.* interfaces with the `openNextFile` method).

26.17.10 close

```
file.close()
```

Close the file. No other operations should be performed on *File* object after `close` function was called.

26.17.11 openNextFile (compatibility method, not recommended for new code)

```
File root = LittleFS.open("/");
File file1 = root.openNextFile();
File file2 = root.openNextFile();
```

Opens the next file in the directory pointed to by the File. Only valid when `File.isDirectory() == true`.

26.17.12 rewindDirectory (compatibility method, not recommended for new code)

```
File root = LittleFS.open("/");
File file1 = root.openNextFile();
file1.close();
root.rewindDirectory();
file1 = root.openNextFile(); // Opens first file in dir again
```

Resets the `openNextFile` pointer to the top of the directory. Only valid when `File.isDirectory() == true`.

26.17.13 setTimeCallback(time_t (*cb)(void))

Sets the time callback for this specific file. Note that the SD and SDFS filesystems only support a filesystem-wide callback and calls to `Dir::setTimeCallback` may produce unexpected behavior.

CHAPTER

TWENTYSEVEN

USB (ARDUINO AND ADAFRUIT_TINYUSB)

Two USB stacks are present in the core. Users can choose the simpler Pico-SDK version or the more powerful Adafruit TinyUSB library. Use the `Tools->USB Stack` menu to select between the two.

27.1 Pico SDK USB Support

This is the default mode and automatically includes a USB-based serial port, `Serial` as well as supporting automatic reset-to-upload from the IDE. Note that if you `Serial.end()` in your code that the reset-to-upload feature will NOT work.

The Arduino-Pico core includes ported versions of the basic Arduino `Keyboard`, `Mouse` and `Joystick` libraries. These libraries allow you to emulate a keyboard, a gamepad or mouse (or all together) with the Pico in your sketches. These libraries only are available when using the built-in USB, not the Adafruit library.

See the examples and Arduino Reference at https://www.arduino.cc/reference/en/language/functions/usb/keyboard/ and https://www.arduino.cc/reference/en/language/functions/usb/mouse

27.2 USB class

The `USB` class allows for applications and libraries to add, remove, or modify the USB connection. Before calling any of the class functions, be sure to include the appropriate header:

```
#include <USB.h>
```

27.3 Dynamic USB Configuration

Whenever changing the USB device information or adding or removing a device, the USB port must be disconnected in software using `USB.disconnect()` to ensure no partially-updated information gets used by the USB stack. Once configuration is completed calling `USB.connect()` will signal the PC that a new device has been attached to the port and it should re-scan and update the devices seen on your system.

Only the most commonly needed calls will be described in this document. For examples of implementing your own custom HID, please look at the `Keyboard` library shipped with the core.

27.3.1 void USB.disconnect()

Tells the Pico to electrically disconnect from an attached USB host. Any devices exported by the Pico should be removed in the host's OS/ This **must** be called before doing any USB modification calls. It is safe to call this function even if the Pico isn't actually plugged into a PC (i.e. when self-powered).

27.3.2 void USB.connect()

Tells the Pico to signal to the host that a device has been plugged in. The host OS should re-scan the Pico and create the new devices exported by it. Call this after modifying USB configurations

27.3.3 void USB.setVIDPID(uint16_t vid, uint16_t pid)

Sets the VID:PID the Pico will export to the host. By default it will use the settings defined for your specific board. Be aware that Using VIDs that are not registered by you with the USB Implementers Forum may be problematic. Only call after a `usbDisconnect()`.

27.3.4 void USB.setManufacturer(const char *str)

Sets the Manufacturer field text in the USB descriptor. Note that most OSes will use the `VID` (vendor ID) to identify the manufacturer but a `lsusb -v` or examination in the Windows device manager will show this manufacturer string. The string must be long-lived (i.e. not a local stack stream, make it a constant or on the heap only). Only call after a `usbDisconnect()`.

27.3.5 void USB.setProduct(const char *str)

Similar to `usbSetManufacturer`. Again, the OS may use the `PID` (product ID) to identify a device and not this string, but it will be present on deeper inspection. Only call after a `usbDisconnect()`.

27.3.6 void USB.setSerialNumber(const char *str)

Similar to `usbSetManufacturer`. This is a free-form string and need not be only numerical. Only call after a `usbDisconnect()`.

27.4 HID Polling Interval

By default, HID devices will request to be polled every 10ms (i.e. 100x per second). If you have a higher performance need, you can override this value by creating a global variable in your main application, set to the polling period:

```
int usb_hid_poll_interval = 1; // Set HID poll interval to 1ms (1kHz)
void setup() {
    ....
}
```

27.5 Ethernet over USB

See *Ethernet over USB*

27.6 Native TinyUSB Sketches

Sketches or libraries which want to implement their own low-level TinyUSB callbacks need to include the appropriate header to have real TinyUSB driver support for HID, MSC (USB stick), MIDI, and NCM (network).

```
// Include one or more headers from the following list
#include <tusb-hid.h>
#include <tusb-midi.h>
#include <tush-msc.h>
#include <tusb-ncm.h>
```

27.7 Adafruit TinyUSB Arduino Support

Examples are provided in the Adafruit_TinyUSB_Arduino for the more advanced USB stack.

To use Serial with TinyUSB, you must include the TinyUSB header in your sketch to avoid a compile error.

```
#include <Adafruit_TinyUSB.h>
```

If you need to be compatible with the other USB stack, you can use an ifdef:

```
#ifdef USE_TINYUSB
#include <Adafruit_TinyUSB.h>
#endif
```

Also, this stack requires sketches to manually call `Serial.begin(115200)` to enable the USB serial port and automatic sketch upload from the IDE. If a sketch is run without this command in `setup()`, the user will need to use the standard "hold BOOTSEL and plug in USB" method to enter program upload mode.

27.8 Adafruit TinyUSB Configuration and Quirks

The Adafruit TinyUSB's configuration header for RP2040 devices is stored in `libraries/Adafruit_TinyUSB_Arduino/src/arduino/ports/rp2040/tusb_config_rp2040.h` (here).

In some cases it is important to know what TinyUSB is configured with. For example, by having set

```
#define CFG_TUD_CDC 1
#define CFG_TUD_MSC 1
#define CFG_TUD_HID 1
#define CFG_TUD_MIDI 1
#define CFG_TUD_VENDOR 1
```

this configuration file defines the maximum number of USB CDC (serial) devices as 1. Hence, the example sketch cdc_multi.ino that is delivered with the library will not work, it will only create one USB CDC device instead of two. It will however work when the above `CFG_TUD_CDC` macro is defined to 2 instead of 1.

To do such a modification when using the Arduino IDE, the file can be locally modified in the Arduino core's package files. The base path can be found per this article, then navigate further to the `packages/rp2040/hardware/rp2040/<core version>/libraries/Adafruit_TinyUSB_Arduino` folder to find the Adafruit TinyUSB library.

When using PlatformIO, one can also make use of the feature that TinyUSB allows redirecting the configuration file to another one if a certain macro is set.

```
#ifdef CFG_TUSB_CONFIG_FILE
    #include CFG_TUSB_CONFIG_FILE
#else
    #include "tusb_config.h"
#endif
```

And as such, in the `platformio.ini` of the project, one can add

```
build_flags =
  -DUSE_TINYUSB
  -DCFG_TUSB_CONFIG_FILE=\"custom_tusb_config.h\"
  -Iinclude/
```

and further add create the file `include/custom_tusb_config.h` as a copy of the original `tusb_config_rp2040.h` but with the needed modifications.

Note: Some configuration file changes have no effect because upper levels of the library don't properly support them. In particular, even though the maximum number of HID devices can be set to 2, and two `Adafruit_USBD_HID` can be created, it will not cause two HID devices to actually show up, because of code limitations.

CHAPTER

TWENTYEIGHT

MULTICORE PROCESSING

The RP2040 chip has 2 cores that can run independently of each other, sharing peripherals and memory with each other. Arduino code will normally execute only on core 0, with the 2nd core sitting idle in a low power state.

By adding a `setup1()` and `loop1()` function to your sketch you can make use of the second core. Anything called from within the `setup1()` or `loop1()` routines will execute on the second core.

`setup()` and `setup1()` will be called at the same time, and the `loop()` or `loop1()` will be started as soon as the core's `setup()` completes (i.e. not necessarily simultaneously!).

See the `Multicore.ino` example in the `rp2040` example directory for a quick introduction.

28.1 Core 1 Operation

By default, core1 (the second core) has no non-user written code running on it. No interrupts, exceptions, or other background processing is done (but the core is still subject to hardware stalls due to on-die memory resource conflicts). When flash erase or write operations (i.e. `LittleFS` or `EEPROM`) are called from core0, core1 **will** be paused.

If `rp2040.getCycleCount` is needed to operate on the second core, then a periodic (once ever 16M clock cycles) `SYSTICK` exception will happen behind the scenes. For extremely time-critical operations this may not be desirable and can be disabled by defining a new `bool` variable to `true` anywhere in your sketch:

```
bool core1_disable_systick = true;
```

28.2 Stack Sizes

When the Pico is running in single core mode, core 0 has the full 8KB of stack space available to it. When using multicore `setup1`/`loop1` the 8KB is split into two 4K stacks, one per core. It is possible for core 0's stack to overwrite core 1's stack in this case, if you go beyond the 4K limitation.

To allocate a separate 8K stack for core 1, resulting in 8K stacks being available for both cores, simply define the following variable in your sketch and set it to `true`:

```
bool core1_separate_stack = true;
```

28.3 Pausing Cores

Sometimes an application needs to pause the other core on chip (i.e. it is writing to flash or needs to stop processing while some other event occurs). In most cases, however, these calls are **SHOULD NOT BE USED**. To synchronize cross-core operations use normal multiprocessor methods such as circular buffers, global `volatile` flags, mutexes, and the like. Stopping a core has massive implications and can kill networking and USB communications if done too long or too frequently.

28.3.1 void rp2040.idleOtherCore()

Sends a message to stop the other core (i.e. when called from core 0 it pauses core 1, and vice versa). Waits for the other core to acknowledge before returning.

The other core will have its interrupts disabled and be busy-waiting in an RAM-based routine, so flash and other peripherals can be accessed.

NOTE If you idle core 0 too long, then the USB port can become frozen. This is because core 0 manages the USB and needs to service IRQs in a timely manner (which it can't do when idled).

28.3.2 void rp2040.resumeOtherCore()

Resumes processing in the other core, where it left off.

28.3.3 void rp2040.restartCore1()

Hard resets Core1 from Core 0 and restarts its operation from `setup1()`. This can cause unpredictable behavior because globals and the heap are shared between cores and not re-initialized with this call. Use with extreme caution.

28.4 Communicating Between Cores

The RP2040 provides a hardware FIFO for communicating between cores, but it is used exclusively for the idle/resume calls described above. Instead, please use the following functions to access a software-managed, multicore safe FIFO. There are two FIFOs, one written to by core 0 and read by core 1, and the other written to by core 1 and read by core 0.

On the RP2350, the hardware FIFO is available for use using the SDK APIs or the calls below (which just wrap the SDK APIs very lightly). This is because the idle/resume calls above are implemented using a hardware doorbell on the RP2350, not the hardware FIFO.

You can (and probably should) use shared memory (such as `volatile` globals) or other normal multiprocessor communication algorithms to transfer data or work between cores, but for simple tasks these FIFO routines can suffice.

28.4.1 void rp2040.fifo.push(uint32_t)

Pushes a value to the other core. Will block if the FIFO is full.

28.4.2 bool rp2040.fifo.push_nb(uint32_t)

Pushes a value to the other core. If the FIFO is full, returns `false` immediately and doesn't block. If the push is successful, returns `true`.

28.4.3 uint32_t rp2040.fifo.pop()

Reads a value from this core's FIFO. Blocks until one is available.

28.4.4 bool rp2040.fifo.pop_nb(uint32_t *dest)

Reads a value from this core's FIFO and places it in dest. Will return `true` if successful, or `false` if the pop would block.

28.4.5 int rp2040.fifo.available()

Returns the number of values available to read in this core's FIFO.

CHAPTER

TWENTYNINE

LOW POWER (SLEEP AND DORMANT)

The `LowPower` library puts the RP2040 or RP2350 into a low power state and wakes it up again from a timer or a GPIO. It is a thin Arduino wrapper over the Pico SDK `pico_low_power` library, plus the work the Arduino core needs after a wake (restoring the CPU clock, restoring pin state).

```
#include <LowPower.h>

void setup() {
    Serial1.begin(115200);
    pinMode(15, INPUT_PULLUP);
}

void loop() {
    Serial1.println("sleeping 5 seconds");
    Serial1.flush();
    LowPower.sleepFor(5000);                // light sleep, timer wake

    Serial1.println("dormant until GP15 goes low");
    Serial1.flush();
    LowPower.dormantUntilPin(15, FALLING);  // oscillators off, pin wake
}
```

29.1 Modes

Sleep gates most clocks. The SDK's sleep leaves the oscillators and PLLs running, which is where most of the sleep current goes. On RP2040 and on the RP2350's ARM cores the library therefore runs the system from the crystal and stops the system PLL and the ring oscillator around `sleepFor` and both forms of `sleepUntilPin` (about 1.3 ms of extra wake time). What else is stopped depends on USB, and is decided at every sleep:

- `NO_USB` build: the USB PLL, the USB clock and the ADC clock are stopped too, and the USB pins D+ / D- are pulled down for the length of the sleep (the USB PHY's receivers float otherwise and draw about 0.25 mA each).
- USB build with a host using USB (connected and not suspended): the USB PLL and clock keep running, so the `Serial` port stays connected.
- USB build with no host: stopped as with `NO_USB`; all clocks are back before the call returns, so a cable plugged in during a sleep enumerates normally.
- With Adafruit TinyUSB (`USE_TINYUSB`) USB is always treated as in use. The RP2350's RISC-V cores use the SDK's sleep unchanged (no clock gating there).

Light sleep keeps the system timer, and therefore `millis()`, running.

Dormant stops the oscillators and PLLs. It can only be woken by a GPIO (or, with a suitable low-frequency clock, a timer) and restarts more slowly (the crystal has to start again, about 73 ms on an RP2040). USB is shut down while dormant.

Every function returns `0` on success or a negative Pico SDK error code (`PICO_ERROR_*`, see `pico/error.h`).

29.2 API

```
int sleepFor(uint32_t ms);
int sleepUntilPin(pin_size_t pin, PinStatus mode);                    // LOW, HIGH,␣
↪RISING, FALLING
int sleepUntilPin(pin_size_t pin, PinStatus mode, uint32_t timeoutMs);   // pin or␣
↪timeout
int dormantUntilPin(pin_size_t pin, PinStatus mode);
int dormantFor(uint32_t ms);
void keepRunning(uint32_t mask);   // KR_PWM, KR_PIO, KR_UART0, KR_UART1, KR_ADC, KR_
↪DMA: keep these running in sleepFor / sleepUntilPin
void keepPins(uint64_t mask);
void setPinRestore(bool restore);
uint32_t lastSleepMs() const;
WakeReason wakeReason() const;     // LowPower.WAKE_NONE, LowPower.WAKE_PIN, LowPower.
↪WAKE_TIMER
```

`CHANGE` is not supported as a wake condition (`PICO_ERROR_INVALID_ARG`).

Call	RP2040 (Pico, Pico W)	RP2350 (Pico 2, Pico 2 W)
`sleepFor`, `sleepUntilPin` (with or without timeout)	yes	yes
`dormantUntilPin`	yes	yes
`dormantFor`	only with an external clock, see below	yes, using the always-on timer on the low-power oscillator
Minimum `dormantFor`	2000 ms	10 ms

Shorter `dormantFor` requests return `PICO_ERROR_INVALID_ARG`. The RP2040 has no always-on timer that survives dormant, so `dormantFor` needs a 32 kHz-class clock fed into GPIO 20 or 22 and registered with `low_power_set_external_clock_source()` before the call; without it `PICO_ERROR_PRECONDITION_NOT_MET` is returned. `dormantUntilPin` needs no extra hardware. The RP2350's switched-core power-down (the SDK's "P-state" calls) is not wrapped by this library.

29.3 Pins

Before going dormant every GPIO is put into its lowest-leakage state (pulls off, input off, output disconnected), except the wake pin, pins routed to a UART (so a TX line does not float), pins routed to I2C (parking them leaves the controller unusable until `Wire.end()` / `begin()`), and the pins you list with `keepPins(mask)` (bit n = GPIO n). The state of the parked pins is saved and **restored after the wake**; `setPinRestore(false)` leaves them parked. Light sleep never parks pins.

The wake pin must be configured by the sketch before the call (`pinMode(15, INPUT_PULLUP)` for a button to ground); on the RP2350 an untouched pad stays isolated from the input logic and never produces the event.

A pin routed to a UART can be the wake pin (`dormantUntilPin(1, FALLING)` with `Serial1` on GP1; the start bit is the edge). With `dormantUntilPin` the byte that causes the wake is **lost** (the UART clock is stopped and the byte is over

before the clocks are back): send a dummy byte first, wait for the wake and then send the data. With `sleepUntilPin` the wake byte is received normally.

29.4 Waking on a pin or a timeout

`sleepUntilPin(pin, mode, timeoutMs)` ends at whichever comes first; `LowPower.wakeReason()` tells which:

```
int rc = LowPower.sleepUntilPin(15, FALLING, 60000);
if (rc == 0 && LowPower.wakeReason() == LowPower.WAKE_PIN) {
    // the button / radio IRQ
} else {
    // 60 s heartbeat
}
```

- `WAKE_PIN` and `WAKE_TIMER` are also what the other calls report; `WAKE_NONE` before the first call and after a call that failed. The names belong to the library object, so they cannot clash with constants of your own.
- With `LOW` or `HIGH` and the pin already in that state the call returns at once with `WAKE_PIN`. If the pin and the timeout coincide either reason may be reported (never both). `timeoutMs` of 0 is invalid.
- Pins set up with `attachInterrupt()` keep working after every sleep and dormant call (the library does not touch the core's GPIO interrupt handler); do not use the wake pin itself for `attachInterrupt()` as well.

`millis()` and `micros()` use the system timer, which **stops in dormant**: dormant time does not appear in them. Light sleep keeps it running. `lastSleepMs()` describes the last call: the *requested* time after a successful `sleepFor` / `dormantFor`, the *measured* time after `sleepUntilPin`, `LowPower.SLEEP_UNKNOWN` (`0xFFFFFFFF`) after `dormantUntilPin` (no time reference while dormant), and `0` after an error or before the first call.

29.5 Current

Whole-board current at VSYS, mean of INA226 results (0.1 ohm shunt, 25 uA steps, the board's regulator included, so the chip alone uses less). The INA226 zero reading was -0.075 mA: add that to the RP2040 Pico column. The Pico was measured at 133 MHz without USB, the Pico W at 133 MHz with a 3.7 V supply (a switching regulator draws more current from a lower supply for the same power: compare in mW, not mA). `NO_USB`, nothing attached, WiFi never started:

RP2040	Pico	Pico W
running (delay loop)	19.7 mA	26.1 mA
`sleepFor`, SDK sleep (PLLs left on)	7.9 mA	
`sleepFor` / `sleepUntilPin`	0.8 mA	1.4 - 1.5 mA
same, USB build with no host	1.1 mA	1.8 mA
same, USB build with a host using USB	3.1 mA	
`dormantUntilPin`	0.8 mA	1.2 mA

Dormant uses 0.07 to 0.2 mA less than light sleep but costs about 18 uC more per wake (the crystal restart), so it pays off for sleeps longer than roughly 0.1 to 0.3 s, at the price of a wake in about 73 ms instead of 4 to 6 ms.

RP2350 (Pico 2, Pico 2 W), 150 MHz	Board current
running (delay loop)	15.6 mA (Pico 2), 17.0 mA (2 W)
`sleepFor`	2.2 to 4.3 mA
`sleepUntilPin`, with or without timeout	1.6 to 4.7 mA
`dormantUntilPin`, `dormantFor`	3.3 to 3.9 mA
the SDK's switched-core power-down	0.15 mA (not wrapped by the library; the wake is a reboot)

On the RP2350 prefer light sleep: dormant is no lower (the regulator only enters its low-power mode in a power-state change). The light sleep floor is steady within one sleep but differs from sleep to sleep in steps of about 0.6 mA (the cause is not known; a plain Pico 2 shows the same, so it is not the CYW43), and the first sleep after a reset can cost about 1.7 mA more. The RISC-V cores have no clock gating: light sleep draws about 6 to 9.5 mA.

29.6 Caveats

- **RP2350-E9**: a GPIO input with its internal pull-down enabled can latch high. The library's pin parking clears it, and so does a wake pin on an idle-high line. A wake pin that relies on the internal pull-down and a RISING edge (a button to 3.3 V) can stay high after the button is released and then not wake again: use an external pull-down, or wake on FALLING with a pull-up.
- **WiFi on a Pico W / Pico 2 W.** A station that is connected before the sleep is still connected after it (the library keeps the CYW43 pins through dormant and `sleepFor` holds the WiFi driver's lock until the clocks are back); the cost is the chip: about 48 mA asleep while connected. `WiFi.end()` leaves the network but does not power the chip off (about 19 mA); with WiFi never started the board draws what a board without WiFi does. To sleep without the chip's current, call `WiFi.powerOff()` before the sleep and `WiFi.powerOn()` after the wake (Pico 2 W: about 4.4 mA in `sleepFor` with the chip off). `powerOn()` reloads the chip's firmware and re-joins the last network, which takes about 7 s or more, so it pays for sleeps longer than about 9 s. Sockets do not survive and Bluetooth stops too; see *Powering the WiFi Chip Off*.
- **USB.** `sleepFor` does not affect USB. Dormant disconnects it: the board re-attaches about 0.6 s after the wake, the host sees the port disappear, and output printed just before or in the first second after the wake can be lost. Do not call `Serial.end()` / `begin()` after a wake. Log anything important after a dormant wake on a UART. Early RP2040 chips (B0/B1) can hit a rare panic in the SDK's USB enumeration workaround (RP2040-E5) when the host re-enumerates after a dormant wake (about 1 in 100 wakes on one board); builds without USB (`NO_USB`) and RP2040 B2 / RP2350 chips are not affected.
- **Clocks.** The SDK wakes into its default clocks; the library re-applies the core's `F_CPU` setting (and the matching core voltage and PSRAM timing) so peripherals configured before the sleep keep working. The UART clock is gated while sleeping (unless `keepRunning()` keeps it, see below): call `Serial1.flush()` before the call and do not expect to receive data during sleep.
- **Peripherals** configured before the call keep working afterwards without `begin()`: UART, SPI, I2C, `SerialPIO`, PIO programs (`tone()`), ADC, and PWM (`analogWrite` stops while dormant and resumes at the same frequency and duty).
- **Bluetooth (BLE) on a Pico W / Pico 2 W.** Advertising and an established connection survive every sleep and dormant call: tested on a Pico 2 W with a laptop connected, the link stayed up through `sleepFor` of 1, 3 and 10 s, `sleepUntilPin` and `dormantUntilPin`, and reads and notifications worked right after each wake (the link layer runs in the CYW43, so even a sleep longer than the 9.6 s supervision timeout the laptop chose did not drop it). Like WiFi it needs the driver's lock held around the sleep, which the library does for every call.
- **FreeRTOS: not supported.** In a build with `Tools->Operating System->FreeRTOS SMP` every sleep and dormant call returns `PICO_ERROR_NOT_PERMITTED` and does nothing. FreeRTOS owns the tick (SysTick), the second core and the WiFi driver's tasks, which the calls change underneath it. Tested on a Pico 2 W: with plain tasks light sleep worked, but the FreeRTOS tick does not run asleep (tick counts, `vTaskDelay()` and timeouts

do not include the sleep); with `setup1()` / `loop1()` defined or a connected WiFi chip light sleep hung; after a dormant wake the tick never ran again. Use `vTaskDelay()` or the SDK's own sleep with FreeRTOS.

- **Both cores.** With `setup1()` / `loop1()` the library idles the other core around the call and resumes it after the wake; the lowest currents need single-core sketches.
- **Flashing a dormant chip.** A dormant chip cannot be reached over SWD and an upload fails. Reset it through the RUN pin, use BOOTSEL, or wake it first; keep a startup delay at the start of `setup()` so a board can always be re-flashed.

Wake latency (RP2040 at 133 MHz, from the start of a UART wake byte to the first output): `dormantUntilPin` about 73 ms, `sleepUntilPin` about 4 to 6 ms, `sleepFor` and the timeout form return at the requested time (about 1 ms late).

29.7 Keeping peripherals running in light sleep

`sleepFor()` and `sleepUntilPin()` gate the clocks of every peripheral except the timer (and UART0 and USB in the pin forms) and run the system clock from the 12 MHz crystal with the PLLs stopped. A PWM output, a PIO program, an ADC capture or UART1 left running therefore stops, and a byte that arrives on UART0 while the chip wakes can be damaged because the clocks restart while it is being received. `LowPower.keepRunning(mask)` keeps selected peripherals running; `keepRunning(0)`, the default, is the normal gated sleep. Dormant ignores it (the oscillators are off). Any non-zero mask leaves the system clock on PLL_SYS, so the kept peripherals run at their normal rate; the price is the PLL, about 4 mA on a Pico 2 W.

- `KR_PWM` (all PWM slices), `KR_PIO` (PIO0, PIO1 and, on the RP2350, PIO2), `KR_UART1`, `KR_DMA`.
- `KR_ADC` keeps the ADC, the DMA clock and PLL_USB (the ADC's clock): sampling and DMA transfers continue.
- `KR_UART0` keeps UART0 (`Serial1`) running in `sleepFor()` as well, and makes a wake on the UART's RX pin (`sleepUntilPin(1, FALLING)` with `Serial1` on GP1) deliver the first byte intact: 8 of 8 bytes intact at 115200 baud with `KR_UART0`, the first byte lost in every run without it.

Measured on a Pico 2 W (150 MHz, 3.7 V, 12 s `sleepFor` per case, 1 kHz PWM or PIO square wave observed on a pin, a 6000-sample DMA capture of the ADC temperature sensor, a 16-byte UART1 loopback at 9600 baud; the plain sleep varies by about 0.4 mA between runs):

keepRunning mask	works	extra current in the sleep
0 (normal gated sleep)	•	2.6 to 3.0 mA total
any mask, nothing running	•	+3.9 mA (PLL_SYS stays on)
KR_PWM	1000.0 Hz, 50 % (8 runs)	+0.7 mA
KR_PIO	1000 Hz square wave	+1.6 mA (one run +3.3 mA)
KR_UART1	16 of 16 bytes intact	+0.15 mA
KR_ADC	6000 of 6000 samples valid	+5 mA (PLL_USB for clk_adc)
KR_UART0	8 of 8 wake bytes intact	none beyond the PLL

A kept peripheral makes the sleep a 7 to 12 mA one, so a timed wake loop (sleep, do the work, sleep) is often cheaper than a continuous peripheral. On the RP2350's RISC-V cores the library does not stop the PLLs at all, so the mask only keeps the peripheral clocks. Not measured on the RP2040. The mask has no effect on the CYW43 handling: the driver's lock is held around every sleep either way.

29.8 Examples

- `SleepForSerial1` - sleep 5 seconds in a loop, reporting over `Serial1`.
- `SleepPinOrTimeout` - light sleep until a pin or a timeout.
- `DormantUntilPin` - go dormant until GP15 is pulled low.
- `KeepRunningPWM` - keep a PWM output running during a `sleepFor`.

CHAPTER

THIRTY

SEMIHOSTING SUPPORT

Using special debugger breakpoints and commands, the Pico can read and write to the debugging console as well as read and write files on a development PC. The `Semihosting` library allows applications to use the semihosting support as a normal filesystem or serial port.

NOTE Semihosting only works when connected to an OpenOCD + GDB debug session. Running an application compiled for Semihosting without the debugger will cause a panic and hang the chip.

As of now, only ARM has support for Semihosting.

30.1 Running Semihosting on the Development Host

Start OpenOCD normally from inside a directory that you can read and write files within (i.e. do not run from `C:\\Program Files\\..` on Windows where general users aren't allowed to write). The starting directory will be where the Pico will read and write files using the `SemiFS` class. Be sure to keep the terminal window you ran OpenOCD in open, because all `SerialSemi` input and output will go to **that** terminal and not `gdb`'s.

Start GDB normally and connect to the OpenOCD debugger and enable semihosting support

```
(gdb) target extended-remote localhost:3333
(gdb) monitor arm semihosting enable
```

At this point load and run your `ELF` application as normal. Again, all `SerialSemi` output will go to the **OpenOCD** window, not GDB.

See the `hellosemi` example in the `Semihosting` library.

30.2 SerialSemi - Serial over Semihosting

Simply include <`Semihosting.h`> in your application and use `SerialSemi` as you would any other `Serial` port with the following limitations:

- Baud rate, bit width, etc. are all ignored
- Input is limited because `read` may hang indefinitely in the host and `available` is not part of the spec

`SerialSemi` can also be selected as the debug output port in the IDE, in which case `::printf` will write to the debugger directly.

30.3 SemiFS - Host filesystem access through Semihosting

Use `SemiFS` the same way as any other file system. Note that only file creation and renaming are supported, with no provision for iterating over directories or listing files. In most cases simply opening a `File` and writing out a debug dump is all that's needed:

```
SemiFS.begin();
File f = SemiFS.open("debug.dmp", "w");
f.write(buffer, size);
f.close();
SerialSemi.printf("Debug dump now available on host.\n");
```

CHAPTER

THIRTYONE

PROFILING APPLICATIONS WITH GPROF

Applications running on the Pico can be profiled using GNU GPROF to show where the CPU is using its time on the device and how often certain functions are called. It does this by recompiling the application and adding a small preamble to each function built to identify what functions call what others (and how frequently). It also uses the SYSTICK exception timer to sample and record the PC 10,000 times per second. When an application is complete, the recorded date can be dumped to the host PC as a `gmon.,out` file which can be processed by `arm-none-eabi-gprof` into useful date.

s histogram of PCs and tally of function caller/callees can take a significant amount of RAM, from 100KB to 10000KB depending on the size of the application. As such, while the RP2040 **may** be able to profile small applications, this is only really recommended on the RP2350 with external PSRAM. The profiler will automatically use PSRAM when available. Call `rp2040.getProfileMemoryUsage()` to get the memory allocated at runtime.

Profiling also adds processing overhead in terms of the periodic sampling and the function preambles. In most cases there is no reason to enable (and many reasons to disable) profiling when an application is deployed to the field.

To transfer the `GMON.OUT` data from the Pico to the host HP can be done by having the application write it out to an SD card or a LittleFS filesystem which is then manually dumped, but for ease of use semihosting can be used to allow the Pico (under the control of OpenOCD and GDB) to write the `gmon.out` file directly on the host PC, ready for use.

NOTE Semihosting only works when connected to an OpenOCD + GDB debug session. Running an application compiled for Semihosting without the debugger will cause a panic and hang the chip.

As of now, only ARM has support for Semihosting or GPROF.

31.1 Enabling Profiling in an Application

The `Tools->Profiling->Enabled` menu needs to be selected to enable profiling support in GCC. This will add the necessary preamble to every function compiled (**Note** that the `libpico` and `libc` will not be instrumented because they are pre-built so calls from them will not be fully instrumented. However, PC data will still be grabbed and decoded from them at runtime.)

The application will automatically start collecting profiling data even before `setup` starts in this mode. It will continue collecting data until you stop and write out the profiling data using `rp2040.writeProfiling()` to dump to the host, a file, serial port, etc.

For example, an application which does all its processing in `setup()` might look like:

```
#include <SemiFS.h>
void setup() {
    SerialSemi.printf("BEGIN\n");
    do_some_work_that_takes_a_long_time_with_many_function_calls();
    // Do lots of other work...
    // Now all done...
```

(continues on next page)

(continued from previous page)

```
    SerialSemi.printf("Writing GMON.OUT\n");
    SemiFS.begin();
    File gmon = SemiFS.open("gmon.out", "w");
    rp2040.writeProfiling(&gmon);
    gmon.close();
    SerialSemi.printf("END\n");
}
void loop() {}
```

31.2 Collecting and Analyzing Profile Data

Running this application under semihosting GDB and OpenOCD generates a `gmon.out` file in the OpenOCD current working directory. This file, combined with the `ELF` binary build in the IDE and loaded through GDB, can produce profiler output using

```
$ /path/to/arm-none-eabi/bin/arm-none-eabi-gprof /path/to/sketch.ino.elf /path/to/gmon.
↪out
```

See the `rp2040/Profiling.ino` example for more details.

CHAPTER

THIRTYTWO

RP2350 SPECIFIC NOTES

The RP2350 chip (present on the Raspberry Pi Pico 2 board and many others) is supported by the core with some minor caveats:

- PSRAM is supported via a new `pmalloc` call and `PSRAM` variable decorator.
- Both RP2350A and RP2350B (48 GPIOs) are supported.

32.1 ARM and RISC-V Modes

Either set of cores can be used on the RP2350, ARM Cortex-M33 or RISC-V Hazard3. Select the desired core from the IDE menus under `Tools->CPU Architecture`. As of the initial release, all libraries should work under the new RISC-V mode with the exception of FreeRTOS. If not, patches are always welcome.

32.2 P2350-E9 Errata ("Increased leakage current on Bank 0 GPIO when pad input is enabled")

Like all chips, the RP2350-A2 stepping has post-silicon chip errata covering certain bugs found after the chip was manufactured. Probably the most (in)famous and concerning is RP2350-E9 which is noted as resulting in "Increased leakage current on Bank 0 GPIO when pad input is enabled." At a high level, this means that in some cases when an input pin is being driven by a weak (high impedance) input, it may not read properly.

After discussion with the community and exploration of the issue by many users, this core has decided not to implement any forced workarounds for this issue. Given the physical nature of the problem and how no one specific solution is appropriate for all conditions.

For more information please read the errata yourself and check the Raspberry Pi Forums for the latest updates.

CHAPTER

THIRTYTHREE

RP2350 PSRAM SUPPORT

The RP2350 chip in the Raspberry Pi Pico 2, and other RP2350 boards, supports an external interface to PSRAM. When a PSRAM chip is attached to the processor (please note that there is none on the Pico 2 board, but iLabs and SparkFun boards, among others, do have it), up to 16 megabytes of additional memory can be used by the chip.

While this external RAM is slower than the built-in SRAM, it is still able to be used in any place where normal RAM would be used (other than for memory-mapped functions and statically initialized variables).

When present, PSRAM can be used in two ways: for specific instantiated variables, or through a `malloc`-like access method. Both can be used in any single application.

33.1 Using PSRAM for regular variables

Similar to `PROGMEM` in the original Arduino AVR devices, the variable decorator `PSRAM` can be added to map a variable into PSRAM. Simply add `PSRAM` to an array and it will be mapped into PSRAM:

```
...
float weights[4000] PSRAM;  // Place an array of 4000 floats in PSRAM
char samplefile[1'000'000] PSRAM; // Allocate 1M for WAV samples in PSRAM
...
```

These variables can be used just like normal ones, no special handling is required. For example:

```
char buff[4 *1024 * 1024];   // 4MB array

void initBuff() {
    bzero(buff, sizeof(buff));
    for (int i = 0; i < 4 *1024 * 1024; i += 4096) {
        buff[i] = rand();
    }
}
```

The only restriction is that these variables may not be initialized statically. The following example will **NOT** work:

```
char buff[] = "This is illegal and will not function";
```

33.2 Using PSRAM for dynamic allocations

PSRAM can also be used as a heap for dynamic allocations using `pmalloc` and `pcalloc`. These calls function exactly like normal `malloc` and `calloc` except they allocate space from PSRAM.

Simply replace a `malloc` or `calloc` with `pmalloc` or `pcalloc` to use the PSRAM heap. Other calls, such as `free` and `realloc` "just work" and do not need to be modified (they check where the passed-in pointer resides and do the right thing automatically).

For example, to create and modify large buffer in PSRAM:

```
void ex() {
    int *buff;
    // Ignoring OOM error conditions in example for brevity
    buff = (int *)pmalloc(10000 * sizeof(*buff));
    // Something happened and we need more space, so...
    buff = (int *)realloc(buff, 20000 * sizeof(*buff)); // buff now has 20K elements
    for (int i = 0; i < 20000; i++) {
        buff[i] = i;
    }
    // Do some work, now we're done
    free(buff);
}
```

C++ objects can be allocated in PSRAM using "placement new" constructors. Note that this will only place immediate object data in PSRAM: if the object creates any other objects via `new` *those* objects will be placed in normal RAM unless the object also uses placement new constructors.

33.3 Checking on PSRAM space

The `rp2040` helper object has the following calls to return the state of the PSRAM heap with the following calls, similar to the normal RAM heap:

33.3.1 int rp2040.getPSRAMSize()

Return the total size of the attached PSRAM chip. This is the **RAW** space and does not take into account any allocations for static variables or dynamic allocations. (i.e. it will return 1, 2, 4, 8, or 16MV depending on the chip).

33.3.2 int rp2040.getTotalPSRAMHeap()

Returns the total PSRAM heap (free and used) available or used for `pmalloc` allocations.

33.3.3 int rp2040.getUsedPSRAMHeap()

Returns the total used bytes (including any overhead) of the PSRAM heap.

33.3.4 int getFreePSRAMHeap()

Returns the total free bytes in the PSRAM heap. (Note that this may include multiple non-contiguous chunks, so this is not always the maximum block size that can be allocated.)

CHAPTER

THIRTYFOUR

BLUETOOTH ON PICOW SUPPORT

34.1 Enabling Bluetooth

To enable Bluetooth (BT), use the `Tools->IP/Bluetooth Stack` menu. It requires around 80KB of flash and 20KB of RAM when enabled.

Both Bluetooth Classic and BluetoothBLE are enabled in `btstack_config.h`.

34.2 Included Bluetooth Libraries

You may use the `KeyboardBT`, `MouseBT`, or `JoystickBT` to emulate a Bluetooth Classic HID device using the same API as their USB versions.

You may use the `KeyboardBLE`, `MouseBLE`, or `JoystickBLE` to emulate a Bluetooth Low Energy (BLE) HID device using the same API as their USB versions.

The `SerialBT` library implements a very simple SPP (Serial Port Profile) Serial-compatible port.

Connect and use Bluetooth peripherals with the PicoW using the `BluetoothHIDMaster` library.

`BluetoothAudio` (A2DP) is also supported, both sink and source.

34.3 Writing Custom Bluetooth Applications

You may also write full applications using the `BTStack` standard callback method, but please be aware that the Raspberry Pi team has built an interrupt-driven version of the BT execute loop, so there is no need to actually call `btstack_run_loop_execute` because the `async_context` handler will do it for you.

Do not call ``cyw43_arch_init`` in your code, either, as that is part of the PicoW variant booting process.

For many BTStack examples, you simply need call the included `btstack_main()` and make sure that `hci_power_control(HCI_POWER_ON);` is called afterwards to start processing (in the background).

34.4 Locking Requirements

You will also need to acquire the BT `async_context` system lock before calling any BTStack APIs. The proper way to do this is using the RAII locking object `BluetoothLock` which handles FreeRTOS and non-OS cases

```
{
    BluetoothLock lock;  // Creating this object on the stack grabs the lock and ensures␣
↪it's released on exit of this block
    ... Call BTStack APIs ...
} // The lock is automatically released
```

Note that if you need to modify the system `btstack_config.h` file, do so in the `include` directory and rebuild the Pico SDK static library. Otherwise the change will not take effect in the precompiled code, leading to really bad behavior.

CHAPTER

THIRTYFIVE

BLUETOOTH HID MASTER

The PicoW can connect to a Bluetooth Classic or Bluetooth BLE keyboard, mouse, or joystick and receive input events from it. As opposed to the `Keyboard`, `Mouse`, and `Joystick` libraries, which make the PicoW into a peripheral others can use, this lets the PicoW use the same kinds of peripherals in a master role.

35.1 BTDeviceInfo Class

The `BluetoothHCI` class implements a scanning function for classic and BLE devices and can return a `std::vector` of discovered devices to an application. Iterate over the list using any of the STL iteration methods (i.e. `for (auto e : list)`). The elements of this list are `BTDeviceInfo` objects which have the following member functions:

`BTDeviceInfo::deviceClass()` returns the Bluetooth BLE UUID or the Blustooth device class for the device. This specifies the general class of the device (keyboard, mouse, etc.).

`BTDeviceInfo::address()` and `BTDeviceInfo::addressString()` return the Bluetooth address as a binary array or a string that can be used to `print`.

`BTDeviceInfo::addressType()` returns whether the BLE address is random or not, and is not generally needed by a user application.

`BTDeviceInfo::rssi()` returns an approximate dB level for the device. Less negative is stronger signal.

`BTDeviceInfo::name()` returns the advertised name of the device, if present. Some devices or scans do not return names for valid devices.

35.2 BluetoothHCI Class

The `BluetoothHCI` class is responsible for scanning for devices and the lower-level housekeeping for a master-mode Bluetooth application. Most applications will not need to access it directly, but it can be used to discover nearby BT devices. As part of the application Bluetooth setup, call `BluetoothHCI::install()` and then `BluetoothHCI::begin()` to start BT processing. Your application is still responsible for all the non-default HCI initialization and customization. See the `BluetoothScanner.ino` example for more info.

35.3 BluetoothHIDMaster Operation

Most applications will use the `BluetoothHIDMaster` class and, after connecting, receive callbacks from the Bluetooth stack when input events (key presses, mouse moves, button mashes) occur.

Application flow will generally be: 1. Install the appropriate callbacks using the `BluetoothHIDMaster::onXXX` methods 2. Start the Bluetooth processing with `BluetoothHIDMaster::begin()` or `BluetoothHIDMaster::beginBLE()` 3. Connect to the first device found with `BluetoothHIDMaster::connectXXX()` and start receiving callbacks. 4. Callbacks will come at interrupt time and set global state variables, which the main `loop()` will process

35.4 Callback Event Handlers

The main application is informed of new inputs via a standard callback mechanism. These callbacks run at interrupt time and should not do significant work, `delay()`, or allocate or free memory. The most common way of handling this is setting global `volatile` flags and variables that the main `loop()` will poll and process.

35.4.1 Mouse Callbacks

The `BluetoothHIDMaster::onMouseMove` callback gets the delta X, Y, and wheel reported by the device. The `BluetoothHIDMaster::onMouseButton` gets a button number and state (up or down) and will be called each time an individual button is reported changed by the mouse.

```
void mouseMoveCB(void *cbdata, int dx, int dy, int dw) {
    // Process the deltas, adjust global mouse state
}

void mouseButtonCB(void *cbdata, int butt, bool down) {
    // Set the global button array with this new info
}
```

35.4.2 Keyboard Callbacks

The *BluetoothHIDMaster::onKeyDown`* callback receives the raw HID key (**NOT ASCII**) sent by the device on a key press while *BluetoothHIDMaster::onKeyUp`* gets the same when a key is released. Note that up to 6 keys can be pressed at any one time. For media keys ("consumer keys" in the USB HID documentation) the `BluetoothHIDMaster::onConsumerKeyDown` and `BluetoothHIDMaster::onConsumerKeyUp` perform the same function (and receive the consumer key IDs defined by the USB HID spec and not ASCII).

To convert the key press and release (including SHIFT handling), use a `HIDKeyStream` object. Simply write the raw HID key and the up/down state to the stream and read back the ASCII for use in an application.

```
HIDKeyStream keystream;

void keyDownCB(void *cbdata, int key) {
    keystream.write((uint8_t )key);
    keystream.write((uint8_t) true); // Keystream now has 1 ASCII character to read out␣
↪and use
    char ascii = keystream.read();
    // ....
}

void keyUpCB(void *cbdata, int key) {
    // Normally don't do anything on this, the character was read in the keyDownCB
}

void consumerKeyDownCB(void *cbdata, int key) {
    // switch(key) and use cases from the USB Consumer Key page
}

void consumerKeyUpCB(void *cbdata, int key) {
    // switch(key) and use cases from the USB Consumer Key page
}
```

35.4.3 Joystick Callbacks

A single `BluetoothHIDMaster::onJoystick` callback gets activated every time a report from a joystick is processed. It receives (potentially, if supported by the device) 4 analog axes, one 8-way digital hat switch position, and up to 32 button states at a time.

```
void joystickCB(void *cbdata, int x, int y, int z, int rz, uint8_t hat, uint32_t␣
↪buttons) {
    // HAT 0 = UP and continues clockwise.  If no hat direction it is set to 0x0f.
    // Use "buttons & (1 << buttonNumber)" to look at the individual button states
    // ...
}
```

35.4.4 PianoKeyboard Example

See the `PianoKeyboard.ino` and `PianoKeyboardBLE.ino` examples for more information on callback operation.

35.4.5 BluetoothHIDMaster::onXXX Callback Installers

```
void BluetoothHIDMaster::onMouseMove(void (*)(void *, int, int, int), void *cbData =␣
↪nullptr);
void BluetoothHIDMaster::onMouseButton(void (*)(void *, int, bool), void *cbData =␣
↪nullptr);
void BluetoothHIDMaster::onKeyDown(void (*)(void *, int), void *cbData = nullptr);
void BluetoothHIDMaster::onKeyUp(void (*)(void *, int), void *cbData = nullptr);
void BluetoothHIDMaster::onConsumerKeyDown(void (*)(void *, int), void *cbData =␣
↪nullptr);
void BluetoothHIDMaster::onConsumerKeyUp(void (*)(void *, int), void *cbData = nullptr);
void BluetoothHIDMaster::onJoystick(void (*)(void *, int, int, int, int, uint8_t, uint32_
↪t), void *cbData = nullptr);
```

35.5 BluetoothHIDMaster Class

35.5.1 bool BluetoothHIDMaster::begin()

Installs and configures the Bluetooth Classic stack and starts processing events. No connections are made at this point. When running in Classic mode, no BLE devices can be detected or used.

35.5.2 bool BluetoothHIDMaster::begin(const char *BLEName)

Installs and configures the Bluetooth BLE stack and starts processing events. No connections are made at this point. When running in BLE mode, no Classic devices can be detected or used.

35.5.3 bool BluetoothHIDMaster::connected()

Returns if the Bluetooth stack is up and running and a connection to a device is currently active.

35.5.4 void BluetoothHIDMaster::end()

Disables the Bluetooth stack. Note that with the current Bluetooth implementation restarting the stack (i.e. calling `begin()` after `end()`) is not stable and will not work. Consider storing state and rebooting completely if this is necessary.

35.5.5 bool BluetoothHIDMaster::running()

Returns if the Bluetooth stack is running at all. Does not indicate if there is an active connection or not.

35.5.6 bool BluetoothHIDMaster::hciRunning()

Returns if the Bluetooth stack has passed the initial HCI start up phase. Until this returns `true` no Bluetooth operations can be performed.

35.5.7 std::vector<BTDeviceInfo> BluetoothHIDMaster::scan(uint32_t mask, int scanTimeSec, bool async)

Passes through the `BluetoothHCI::scan()` function to manually scan for a list of nearby devices. If you want to connect to the first found device, this is not needed.

35.5.8 bool BluetoothHIDMaster::connect(const uint8_t *addr)

Start the connection process to the Bluetooth Classic device with the given MAC. Note that this returns immediately, but it may take several seconds until `connected()` reports that the connection has been established.

35.5.9 bool BluetoothHIDMaster::connectKeyboard(), connectMouse(), connectJoystick(), connectAny()

Connect to the first found specified Bluetooth Classic device type (or any HID device) in pairing mode. No need to call `scan()` or have an address.

35.5.10 bool BluetoothHIDMaster::connectBLE(const uint8_t *addr, int addrType)

Start the connection process to the Bluetooth BLE device with the given MAC. Note that this returns immediately, but it may take several seconds until `connected()` reports that the connection has been established.

35.5.11 bool BluetoothHIDMaster::connectBLE()

Connect to the first found BLE device that has a HID service UUID (keyboard, mouse, or joystick)

35.5.12 bool BluetoothHIDMaster::disconnect()

Shuts down the connection to the currently connected device.

35.5.13 void BluetoothHIDMaster::clearPairing()

Erases all Bluetooth keys from memory. This effectively "forgets" all pairing between devices and can help avoid issues with the beta Bluetooth stack in the SDK.

35.5.14 void BluetoothHIDMaster::setPasskeyCB(void (*fn)(uint32_t passkey))

Sets the callback function to display a passkey for the user to enter on their keyboard to complete pairing. If none specified, the passkey will be printed on `Serial`.

CHAPTER

THIRTYSIX

BLUETOOTH AUDIO (A2DP SOURCE AND SINK)

The PicoW can be used as a Bluetooth Audio sink or source with the `BluetoothAudio` class. Operation is generally handled "automatically" in the background so while the audio is playing or streaming the main application can perform other operations (like displaying playback info, polling buttons for controls, etc.)

```
#include <BluetoothAudio.h>
...
```

Note about CPU usage: Bluetooth SBC audio is a compressed format. That means that it takes non-trivial amounts of CPU to compress on send, or decompress on receive. Transmitting precompressed audio from, say, MP3 or AAC, requires first decompressing the source file into raw PCM and then re-compressing them in the SBC format. You may want to consider overclocking in this case to avoid underflow.

36.1 A2DPSink

This class implements slave sink-mode operation with player control (play, pause, etc.) and can play the received and decoded SBC audio to `PWMAudio`, `I2S`, or a user-created *BluetoothAudioConsumer* ` class.

The `A2DPSink.ino` example demonstrates turning a PicoW into a Bluetooth headset with `PWMAudio`.

36.2 A2DPSource

This class implements a master source-mode SBC Bluetooth A2DP audio connection which transmits audio using the standard `Stream` interface (like `I2S` or `PWMAudio`. The main application connects to a Bluetooth speaker and then writes samples into a buffer that's automatically transmitted behind the scenes.

The `A2DPSource.ino` example shows how to connect to a Bluetooth speaker, transmit data, and respond to commands from the speaker.

CHAPTER

THIRTYSEVEN

BLE (BLUETOOTH LOW ENERGY) SERVER/CLIENT

Support for using the Pico W as a BLE server (peripheral) or BLE client (central) is available through the BLE library.

Note: These instructions assume knowledge about BT and BLE in general. Look at the official ArduinoBLE documentation for a quick overview of the basics.

This library was developed specifically for the BT implementation used in the Pico SDK, BTStack. Its API is close to the ESP32 BLE and the official ArduinoBLE, but not directly copmatible with either. However, if you have used either of those libraries then coming up to speed on Arduio-Pico BLE should be straightforward.

To use this library, include BLE.h in your sketch and ensure the `Tools->IP/BT Stack` option in the menus has + `Bluetooth` enabled (and, obviously, you need a BT-enabled Pico or compatible board with RM2 coprocessor).

```
#include <BLE.h>
...
void setup() {
    // Optionally set security mode, before .begin, see below
    BLE.begin(); // Initialize the BT stack
    ...
}
```

This library is not compatible with FreeRTOS due to memory allocations required at interrupt time.

37.1 General Architecture

BLE implements a hierarchical object structure, with a top-level `BLE` containing either a `BLEServer` or a `BLEClient`. `BLE` handles scanning for devices to connect to in `BLEClient` mode, but other than that all work is handled by the `BLEServer` or `BLEClient` as appropriate.

Applications can't generate their own `BLEServer` or `BLEClient` objects, they must use the `BLE.server()` or `BLE.client()` calls to get a pointer to the global server or client object.

```
#include <BLE.h>
void setup() {
    BLE.begin();
    auto server = BLE.server(); // Get pointer to single global server object
    // ... create a BLEService with BLECharacteristics somehow...
    server->addService(...); // Add the service we just made to the object
    server->startAdvertising(); // Tell the world about it!
}
```

`BLEServer` holds a collection of application-defined `BLEService` objects which represent a BLE service. Each `BLEService` can contain a list of `BLECharacteristics` which are what the host PC or phone will actually interact with. Each `BLECharacteristic` contains a application-defined value which can be read or written by the host PC.

The sketch can be notified when the host PC or phone writes to one of its characteristics (i.e. a callback could set the dimming of a LED depending on the value written by the PC).

`BLEClient` connects to a remote BLE server (peripheral) and creates a set of `BLERemoteService` objects, each of which can contain zero of more `BLERemoteCharacteristic` objects that the remote device supports. Each of those `BLERemoteCharacteristic` can be read or written (if allowed) and can be optionally notified when the remote changes through the `onNotify` callback (again, if the remote server allows it).

Sketches operate with the BLE at two levels: main application and BLE-event triggered callbacks. The main app can write or read a characteristic at any time. Callbacks happen when the remote end of the connection performs a read or write to a characteristic and execute at interrupt-level (so no filesystem access, etc.).

37.2 General Bluetooth Support Classes

37.2.1 BLEAddress

This class holds a BLE MAC address. As of the initial release, only public addresses are supported. Randomized ones are not yet, but if they're important to you please submit a PR.

BLEAddress(uint8_t *a)

Creates a `BLEAddress` using the 6-byte MAC of a device.

```
uint8_t mySpecialDevice[6] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06};
BLEAddress x(mySpecialDevice);
```

BLEAddress(String a)

Creates a `BLEAddress` by parsing the MAC in human readable form (hex separated with colons)

```
BLEaddress x("01:02:03:04:05:06");
```

37.2.2 BLEUUID

All services and characteristics are identified by UUIDs in BLE. There are many predefined 16-bit UUIDs for common devices such as exercise equipment or home monitoring, but anyone can create their own 128-bit UUID for a customized service and characteristics.

BLEUUID(uint16_t u)

Create a 16-bit UUID. See Assigned Numbers for the values predefined by the Bluetooth SIG.

```
BLEUUID batteryServiceUUID(0x180F);
```

BLEUUID(const uint8_t u[16])

Create a 128-bit UUID (16 bytes) from an array of bytes

```
uint8_t mySuperCool[16] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
BLEUUID mySuperCoolUUID(mySuperCool);
```

BLEUUID(String u)

Creates a 128-bit UUID from a human-readable string

```
BLEUUID myNewUUID("040b4668-cae0-4b57-88c6-a749d7b1e3dd");
```

37.3 BLE (Bluetooth Low Energy) Base

The `BLE` object is how a sketch interacts with the hardware Bluetooth stack. It manages the low-level BTStack and contains either a `BLEClient` or `BLEServer` where the actual BLE operations will happen.

37.3.1 Configuring Security and Starting Up

Before calling `BLE.begin()` you can specify if you want the Pico to request a bonding with "Just Works" security (used for things like mice or headsets where there really isn't a good user input/output capability) or allow any device to connect without building a connection (the default)

BLE.setSecurity(BLESecurityNone or BLESecurityJustWorks)

Configures the BLE security model. By default no pairing will be needed and devices will be able to interact with any services shared on this Pico. `BLESecurityJustWorks` enables PIN-less pairing with a BLE central/client which enables encryption (but not authentication).

This must be called **before** `BLE.begin()` is executed

BLE.begin(String name)

Starts the BLE stack on the Pico. Call it before setting up any services or client operations. The `name` is used when advertising the device.

BLEAddress BLE.address()

Returns this Pico's BLE MAC address. Only valid after `BLE.begin()`.

37.4 Accessing the Global BLEServer or BLEClient

To get a pointer to either the server or client objects to actually do work, call the appropriate `BLE` method

37.4.1 BLEServer *BLE.server()

Returns the global server object which will contain a list of `BLEService` objects that will be used by the outside world to connect to this Pico. See the Server section below for more information.

37.4.2 BLEClient *BLE.client()

Returns the global client object which will be used to connect to a remote BLE server, remote services, and remove characteristics. See the Client section below

37.5 Configuring BLE Advertising

Advertising is the way that a BLE device shares its presence to the world. The BLE hardware will broadcast a small (31 byte) packet containing information such as the device name, what it looks like, and any service UUIDs it provides. BLE clients normally don't need to advertise and can ignore the following information.

37.5.1 BLEAdvertising *BLE.advertising()

This call returns the current advertising object for any defined servers. In general, sketches do not need to work with this call or `BLEAdvertising` at all, but this call lets you do things like add URLs or appearance information.

37.5.2 BLE.startAdvertising(bool advertiseServiceUUID)

After all services are created (see below) the sketch needs to use this call to start broadcasting for BLE clients to find it. Call this after all services are defined, or after `BLE.stopAdvertising()` and changing services or characteristics. Without this call, your Pico won't be discoverable. The Boolean flag `advertiseServiceUUID` controls inclusion of the service UUID in the advertising data.

37.5.3 BLE.stopAdvertising()

Turns off advertising to allow modifying services/etc.

37.6 Finding BLE Servers to Connect To

BLE clients need to listen the BLE advertisements around them to discover servers they can connect to. This library allows for scanning for all servers, or only servers exposing a service they care about (i.e. only discovering available thermostats or heart rate monitors).

Listens to the BLE announcements for `timeoutsec` seconds and returns a `BLEScanReport` which is a `std::list<BLEAddress>` of all seen BLE servers available. Enteires in this list can be used in a `BLE.client()->connect()` call to initiate a BLE client connection (see below).

Like the prior call, except only returns servers who are advertising the service UUID specified. Most sketches will want to use this call to look for only the device types they want (i.e. `auto cyclingInfoList = BLE.scan(BLEUUID(0x1816)); // Only look for cycling devices`)

The information for each BLE device seen is stored in memory, taking around 100 bytes of memory per device. If sketches are very memory constrained, call this method to free all the memory associated with the last `BLE.scan()` after you have selected a single device to connect to. Most sketches don't need to do this.

37.7 Implementing a BLE Server (peripheral/device)

A BLE gadget/peripheral/device offers zero or more `BLEService` to the outside world. Each `BLEService` contains zero or more `BLECharacteristics` that the outside world can read and/or write to. The objects defining these services and characteristics must be created before calling `BLE.startAdvertising()`.

37.7.1 BLEService

A `BLEService` is essentially a container to hold a list of `BLECharacteristics` identified by a `BLEUUID` that identifies the type of service.

For well known, predefined services these UUIDS are standardized by the Bluetooth SIG. If you are implementing a gadget that provides predefined services, using the well known service UUIDs (see section 3.4.1 in Assigned Numbers will allow any BLE client device (PC, phone, home hub) to interact with the Pico.

You can (and should) generate your own UUID (`uuidgen` on Linux or any online generator) when creating custom services.

Create a BLEService with the UUID using `new` in a function

```
void setup() {
    BLE.begin();
    BLEService *cyclingPowerService = new BLEService(BLEUUID(0x1818)); // 0x1818 defined␣
↪by SIG as "Cycling Power Service"
    ...
```

or using a global variable (because you need the `BLEService` object to live for the entire application, you can't take the address of an immediate local variable…it won't be there after the function exits)

```
BLEService cyclingPowerService = BLEService(BLEUUID(0x1818)); // 0x1818 defined by SIG␣
↪as "Cycling Power Service"

void setup() {
    BLE.begin();
    ...
```

37.7.2 Adding Characteristics to a BLEService

Once you have a `BLEService` you need to create `BLECharacteristics` to share data (i.e. current temperature, running speed, solar power, etc.) with a client. They can be though of as a free-form variable that both your sketch and the outside world can read (and update if permitted).

Characteristics have a UUID to identify them (again, there are well-known ones for well-known services and you should use them if appropriate, or generate your own for custom ones you generate).

Note that for custom UUIDs, the UUID used for characteristics should not be the same as the UUID for the custom service. Generate new UUIDs for every characteristic or it could confuse BLE clients.

Create a characteristic with the following constructor (again, be sure it is a "long lived" variable to make it global or use `new` to make it on the heap

BLECharacteristic(BLEUUID u, uint16_t characteristicPermission, const char *desc = nullptr, uint8_t permR = 0, uint8_t permW = 0)

Creates a characteristic with the given UUID.

Permissions is the binary OR (`|`) of `BLERead`, `BLEWrite`, `BLENotify` (i.e. `BLERead | BLENotify` or `BLERead | BLEWrite`) where `BLERead` means the client (and the sketch) can read the value (i.e. it is an output of your sketch), `BLEWrite` means the client can write the value (the sketch can, too, of course), and `BLENotify` means the Pico will let the client know when the Pico changes it with `setValue()` , via the BLE notify message.

The optional `desc` parameter is a human-readable text description of the characteristic. It is not mandatory and only certain BLE tools can read and display it.

```
void setup() {
    BLE.begin();
    BLEService *cyclingPowerService = new BLEService(BLEUUID(0x1818)); // 0x1818 defined␣
↪by SIG as "Cycling Power Service"
    BLECharacteristic *pwrChar = new BLECharacteristic(BLEUUID(0x2A63), BLEREAD, "How␣
↪hard am I working"); // 0x2a64 is "Cycling Power Measurement"
```

Once a characteristic has been created, add it to a service. Services can have many characteristics, and the order in which they are added is not generally important since they are identified by the UUID, not the index.

Keep the characteristic variable pointer handy to set values to it or receive callbacks when remote reads or writes happen.

BLEService::addCharacteristic(BLECharacteristic *c)

Adds the characteristic to the service. As always, the variable needs to be on the heap or a global to ensure the pointer remains valid the lifetime of the sketch.

37.7.3 Setting Characteristic Data

Characteristics can be read and written to by the sketch (and the remote client if permissions allow it). They can be any format desired (character strings, binary floating point, 1-byte booleans), but for well-known services be sure to use the format specified by the Bluetooth SIG.

When values are set by the application, if the BLE client requests (and is permitted) notification, they will automatically get a message with the new data, the sketch itself needs do nothing.

BLECharacteristic::setValue(<simple data type>)

The characteristic object can set by the sketch by calling this function. The single value passed in will be stored in the characteristic in the same format as passed-in (i.e. `setValue((uint16_t)0xabcd)` would set the characteristic to 2 bytes of data, {0xab, 0xcd}).

BLECharacteristic::setValue(const uint8_t *data, size_t size)

Sets the characteristic to any binary data (i.e. a large `struct`) you pass in, at any size. The data is copied over so the passed-in pointer can be invalidated by the sketch at any time.

37.7.4 Reading Characteristic Data

The BLE client (PC, phone, etc.) can write data to characteristics (if permissions allow) and the sketch can read those values and do things with them (i.e. set a thermostat, light an LED, etc.). Use the appropriate `BLECharacteristic::get<type>` call.

```
bool getBool()
char getChar()
uint8_t getInt8()
unsigned char getUChar()
uint8_t getUInt8()
short getShort()
int16_t getInt16()
unsigned short getUShort()
uint16_t getUInt16()
int getInt()
int32_t getInt32()
unsigned int getUInt()
uint32_t getUInt32()
long getLong()
unsigned long getULong()
long long getLongLong()
int64_t getInt64()
unsigned long long getULongLong()
uint64_t getUInt64()
float getFloat()
float getDouble()
String getString()
```

If you need the raw binary data use the following calls

```
const void * valueData()
size_t valueLen()
```

37.7.5 Getting Callbacks for Characteristics

Your sketch can get a callback to a function or class object whenever the remote devices reads or writes a characteristic (i.e. you could configure a callback whenever the home hub writes a new temperature setting to your device instead of polling `characteristic->getFloat()`).

To register a read or write callback (which will be executed at interrupt level so don't do things like filesystem access or flash writes) use

```
void myLEDReadCB(BLECharacteristic *c) {
    Serial.println("Yay, someone read me!");
}

void myLEDWriteCB(BLECharacteristic *c) {
    // Control the LED, immediately
    digitalWrite(LED_BUILTIN, c->getBool());
}

void setup() {
    BLE.begin("LEDMASTER");
    ... set up service
    auto s = new BLEService(BLEUUID(MYCUSTOMSVC));
    auto c = new BLECharacteristic(BLEUUID(MYCUSTOMUUID), BLEWrite | BLEREAD, "Lightbulb␣
↪Control");
    c->onRead(myLEDReadCB);
    c->onWrite(myLEDWriteCB);
    s->addCharacteristic(c);
    BLE.startAdvertising();
}

void loop() {
    // Nothing here, the LED will be controlled thru the callback!
}
```

For a class-based callback demonstration, see the BLEServiceUART.cpp source file.

37.8 Implementing a BLE Client (BLE Central)

A BLE client scans the available device and then connects to one that offers the service it needs. When it connects it then gets a full list of zero or more remote services. For each of those remote services it then gets a full list of remote characteristics it can use to interact with the device.

Once a list of available servers is collected via `BLE.scan()` as explained above, the Pico chooses one to connect to with a call to

37.8.1 BLEClient()

bool BLE.client()->connect(BLEAdvertising a, int timeoutsecs = 10)

Attempts to connect to a remote BLE server and perform the scans described above. If the connection doesn't complete within the timeout, it will fail and return `false` . Note that further BLE operations to that server will use a different

timeout (default 2 seconds) which can be changed via `BLE.client()->setTimeout(int timeoutsecs)`

Accessing Remote Services

After successful connections, it is possible to examine the list of remote services or choose any specific one (most common)

BLERemoteService *BLE.client()->service(BLEUUID)

Returns either *nullptr* if no service exists, or `BLERemoteService` that can be used to find characteristics the sketch cares about. Don't `delete` this object, it is managed completely by the `BLE` object.

37.8.2 Accessing Remote Characteristics

Remote characteristics can be read, written, or registered for notification (remote permissions allowing it, of course). All remote characteristics are wrapped in `BLERemoteCharacteristic` objects.

There is no built-in caching, so every read of a remote characteristic goes over the BLE radio and can take some time. Your sketch should cache any values it needs to use repeatedly.

37.8.3 Remote Characteristic Permissions

The BLE server defines the permissions the Pico is given for each of the remote characteristics it provides.

```
bool BLERemoteCharacteristic::canRead()
bool BLERemoteCharacteristic::canWrite()
bool BLERemoteCharacteristic::canNotify()
```

String BLERemoteCharacteristic::description()

Returns the human readable text description associated with the remote characteristic, if the remote server supplies one. This is the equivalent of the descriptino parameter in the BLECharacteristic constructor.

37.8.4 Reading Remote Characteristics

The same `getXXX` calls as defined for `BLECharacteristic` (local characteristic) work for `BLERemoteCharacteristic` . They may be somewhat slower due to the need to poll the remote server for the data.

37.8.5 Writing Remote Characteristics

The same `setXXX` calls as defined for `BLECharacteristic` work for `BLERemoteCharacteristic` except, of course, that they send the new data to the remote device.

37.8.6 Getting Callbacks for Remote Characteristics

Your sketch can receive notifications when the remote server writes to a characteristic, assuming the remote server gives permission. That avoids having to read a remote characteristic over and over (using up radio time and power) to check for changes.

Both class and functional callbacks are available. Class-based works the same as for `BLECharacteristics` .

BLERemoteCharacteristic::onNotify(void (*notifyCB)(BLERemoteCharacteristic *c, const uint8_t *data, uint32_t len))

Registers a callback that will be called ar interrupt level when a BLE notify message is received for the remote characteristic. Your callback will need to cast the returned `data` pointer to the appropriate format (i.e. a `bool` or a `char[]`).

37.9 BLE Beacons

BLE beacons are devices which just advertise their presence (infrequently, to save power) and provide no services or characteristics. All information about them (their UUID and their major and minor ID numbers) are contained in the advertisement.

The `BLEBeacon` class implements this functionality. Note that no `BLE..service()` calls should be used in this mode, just `BLEBeacon::begin()` (after setting the UUID and major/minor IDs. See the `Beacon.ino` example for usage

37.10 BLE Battery Service

The Bluetooth SIG standard BLE Battery service is implemented in `BLEServiceBattery`. Instantiate an instance of this object and `BLE.server()->addService()` it to gain this functionality.

Call `BLEBatteryService::set(int lvl /* 0-100 */)` from your main sketch as the battery state changes. All other work is handled behind-the-scenes.

This service can be added alongside other services, of course.

37.11 BLE Serial UART (Nordic SPP Service)

While not a Bluetooth SIG standard, the BLE "UART" service created by Nordic for their chipsets is very commonly used for low-power, low-volume data transmission over BLE. The `BLEServiceUART` class implements this "standard" as a `arduino::HardwareSerial` object (like Serial or Serial1/2 or SoftwareSerial). This service can be combined with others, as always. It implements an automatic data flush mechanism, or the user can `flush()` as appropriate for their protocol.

See the `SerialBLE.ino` example for usage.

CHAPTER

THIRTYEIGHT

SINGLEFILEDRIVE

USB drive mode is supported through the `SingleFileDrive` class which allows the Pico to emulate a FAT-formatted USB stick while preserving the onboard LittleFS filesystem. A single file can be exported this way without needing to use FAT as the onboard filesystem (FAT is not appropriate for flash-based devices without complicated wear leveling because of the update frequency of the FAT tables).

This emulation is very simple and only allows for the reading of the single file, and deleting it.

38.1 Callbacks, Interrupt Safety, and File Operations

The `SingleFileDrive` library allows your application to get a callback when a PC attempts to mount or unmount the Pico as a drive. Your app can also get a callback if the user attempts to delete the file (but your sketch does not actually need to delete the file, it's up to you).

Note that when the USB drive is mounted by a PC it is not safe for your main sketch to make changes to the LittleFS filesystem or the file being exported. So, normally, your `onPlug` callback will set a flag letting your application know not to touch the filesystem, with the `onUnplug` callback clearing this flag.

Also, because the USB port can be connected at any time, it is important to disable interrupts using `noInterrupts()` before writing to a file you will be exporting (and restoring them with `interrupts()` afterwards). It is also important to `close()` the file after each update, or the on-flash version the `SingleFileDrive` will attempt to export may not be up to date causing issues later on.

See the included `DataLoggerUSB` sketch for an example of working with these limitations.

38.2 Using SingleFileDrive

Implementing the drive requires including the header file, starting LittleFS, defining your callbacks, and telling the library what file to export. No polling or other calls are required outside of your `setup()`. (Note that the callback routines allow for a parameter to be passed to them, but in most cases this can be safely ignored.)

```
#include <LittleFS.h>
#include <SingleFileDrive.h>

void myPlugCB(uint32_t data) {
    // Tell my app not to write to flash, we're connected
}

void myUnplugCB(uint32_t data) {
    // I can start writing to flash again
}
```

(continues on next page)

(continued from previous page)

```
void myDeleteDB(uint32_t data) {
    // Maybe LittleFS.remove("myfile.txt")?  or do nothing
}

void setup() {
    LittleFS.begin();
    singleFileDrive.onPlug(myPlugCB);
    singleFileDrive.onUnplug(myUnplugCB);
    singleFileDrive.onDelete(myDeleteCB);
    singleFileDrive.begin("littlefsfile.csv", "Data Recorder.csv");
    // ... rest of setup ...
}

void loop() {
    // Take some measurements, delay, etc.
    if (okay-to-write) {
        noInterrupts();
        File f = LittleFS.open("littlefsfile.csv", "a");
        f.printf("%d,%d,%d\n", data1, data2, data3);
        f.close();
        interrupts();
    }
}
```

CHAPTER

THIRTYNINE

FATFSUSB

When the onboard flash memory is used as a `FatFS` filesystem, the `FatFSUSB` can be used to allow exporting it to a PC as a standard memory stick. The PC can then access, add, and remove files as if the Pico was a USB memory stick, and upon ejection the Pico can access any new files just as if it made them itself.

(Note, if you are using LittleFS then you need to use `SingleFileDrive` to export a single file, not this class, because the PC does not understand the LittleFS disk format.)

39.1 Callbacks, Interrupt Safety, and File Operations

The `FatFSUSB` library allows your application to get a callback when a PC attempts to mount or unmount the Pico as a FAT drive.

When the drive is being used by the Pico (i.e. any `File` is open for read or write, the `FatFS` is not `end()` -ed and still mounted, etc.) the host may not access it. Conversely, while the host PC is connected to the drive no `FatFS` access by the Pico is allowed.

Your `driveReady` callback will be called when the PC attempts to mount the drive. If you have any files open, then this callback can report back that the drive is not yet ready. When you complete file processing, the PC can re-attempt to mount the drive and your callback can return `true` .

The `onPlug` callback will generally `FatFS.end()` and set a global flag letting your application know not to touch the filesystem until the flag is cleared by the `onUnplug` callback (which will also do a `FatFS.begin()` call).

Failing to close all files **and** `FatFS.end()` before granting the PC access to flash memory will result in corruption. FAT does not allow multiple writers to access the same drive. Even mounting and only reading files from the PC may cause hidden writes (things like access time, etc.) which would also cause corruption.

See the included `Listfiles-USB` sketch for an example of working with these limitations.

CHAPTER

FORTY

FREERTOS SMP

The SMP (multicore) port of FreeRTOS is included with the core. This allows complex task operations and real preemptive multithreading in your sketches. While the `setup1` and `loop1` way of multitasking is simplest for most folks, FreeRTOS is much more powerful.

40.1 Enabling FreeRTOS

To enable FreeRTOS, simply add

```
#include <FreeRTOS.h>
```

to your sketch and select `Tools->Operating System->FreeRTOS SMP` to enable it.

When using Platform.IO you need to add the following define to your .ini file:

```
; Enable FreeRTOS Support
build_flags = -DPIO_FRAMEWORK_ARDUINO_ENABLE_FREERTOS
```

40.2 Configuration and Predefined Tasks

FreeRTOS is configured with 32 priority levels (0 through 31) and a process for `setup()/loop()`, `setup1()/loop1()`, LWIP, and the USB port will be created. The task quantum is 1 millisecond (i.e. 1,000 switches per second).

`setup()` and `loop()` are assigned to only run on core 0, while `setup1()` and `loop1()` only run in core 1 in this mode, the same as the default multithreading mode.

There is an LWIP worker thread running on core 0 at priority `(configMAX_PRIORITIES - 2)` This should allow for LWIP to always make progress even with applications that don't `yield` or `delay`, while leaving the highest priority available for hard real time processes. This setting can be changed by defining `LWIP_TASK_PRIORITY` in your build process.

You can launch and manage additional processes using the standard FreeRTOS routines.

`delay()` and `yield()` free the CPU for other tasks, while `delayMicroseconds()` does not.

40.3 Caveats

While the core now supports FreeRTOS, most (probably all) Arduino libraries were not written to support preemptive multithreading. This means that all calls to a particular library should be made from a single task.

In particular, the `LittleFS` and `SDFS` libraries can not be called from different threads. Do all `File` operations from a single thread or else undefined behavior (aka strange crashes or data corruption) can occur.

40.4 More Information

For full FreeRTOS documentation look at FreeRTOS.org and FreeRTOS SMP support.

CHAPTER

FORTYONE

WIFI (RASPBERRY PI PICO W) SUPPORT

WiFi is supported on the Raspberry Pi Pico W by selecting the "Raspberry Pi Pico W" board in the Boards Manager. It is generally compatible with the Arduino WiFi library and the ESP8266 Arduino WiFi library.

Enable WiFi support by selecting the *Raspberry Pi Pico W* board in the IDE and adding `#include <WiFi.h>` in your sketch.

41.1 Non-Raspberry Pi WiFi Support

In addition to the official WiFi chip from Raspberry Pi, this core supports additional WiFi interface devices:

- Microchip WINC1500
- ESP32-based devices using ESPHost

See the examples in the respective library folders for instructions on instantiating and using these devices. Most of the features documented here will also work on these WiFi chips.

41.2 Supported Features

- WiFi connection (Open, WPA/WPA2)
  - Static IP or dynamic DHCP supported
  - Station Mode (STA, connects to an existing network)
  - Access Point Mode (AP, creates its own wireless network) with 4 clients
- WiFi Scanning and Reporting
  - See the `ScanNetworks.ino` example to better understand the process.

41.3 Powering the WiFi Chip Off (Pico W / Pico 2 W)

`WiFi.end()` and `WiFi.disconnect()` leave the network but keep the CYW43 chip powered, which costs about 20 mA on the board whatever the CPU does (about 50 mA with the station connected and idle). To save power while the WiFi is not needed (a sleep of more than about 10 seconds, see below) power the chip off:

```
bool WiFi.powerOff();
bool WiFi.powerOn(bool reconnect = true, uint32_t timeoutMs = 15000);
bool WiFi.isPoweredOn();
```

- `powerOff()` leaves the network (and stops the access point, if any), then pulls `WL_ON` low. The SSID and password of the last `begin()` are remembered; a static address set with `config()` is kept. **Bluetooth stops too**: the CYW43 chip carries both WiFi and Bluetooth, and `powerOff()` powers all of it off, ending a Bluetooth link

and advertising. `powerOn()` restores WiFi only; a sketch that uses Bluetooth (BTstack) has to start the Bluetooth stack again after `powerOn()` (tested on a Pico 2 W with BTstack and a laptop connected, 10 of 10 cycles: the link was gone after `powerOff()` and nothing advertised again after `powerOn()`).

- `powerOn()` powers the chip up again (the firmware is loaded again) and, with `reconnect`, joins the same network again like `begin()` does, blocking until connected or `timeoutMs` (twice that when `begin()` itself blocks) have passed, also in a sketch that called `WiFi.mode()`. With `reconnect = false` the chip is only powered up (and left in station mode); call `begin()` later.
- The three calls exist on every board so that a sketch builds everywhere; on a board without the CYW43 chip `powerOff()` and `powerOn()` do nothing and return `false`, and `isPoweredOn()` is `false`.
- Sockets and server objects do not survive: re-create them after `powerOn()`.
- Timing, Pico 2 W: the chip is up and joined after about 6.8 s (firmware load about 0.4 s, the rest is scan, association and DHCP). A remembered channel and BSSID, or a static address, did not shorten it measurably, so none is kept. Some reconnects take 10 s or more (up to 18 s seen), when the first join attempt fails and the driver retries. How long it takes depends on the access point: 6 to 15 s were measured on one, so the default `timeoutMs` of 15 s can be too short for a slow one (`powerOn()` then returns false while the join may still complete a little later); pass a larger `timeoutMs` for such a network. The reconnect costs about 0.43 C (62 mA for 7 s), so powering off pays when the sleep is longer than about 9 s; for shorter ones keep the station connected (`LowPower` light sleep keeps a connected station alive).
- Pico 2 W board current at 3.7 V: connected and in `LowPower.sleepFor()` about 51 mA, chip powered off about 4.4 mA (the RP2350's own sleep floor; 3.6 mA in `dormantFor`). See the `WiFiPowerOffSleep` example.

41.4 Important Information

- Adding WiFi increases flash usage by over 220KB
  - There is a 220KB binary firmware blob for the WiFi chip (CYW43-series) which the Pico W uses, even to control the onboard LED.
- Adding WiFi increases RAM usage by ~40KB.
  - LWIP, the TCP/IP driver, requires preallocated buffers to allow it to run in non-polling mode (i.e. packets can be sent and received in the background without the application needing to explicitly do anything).
- The WiFi driver has some limitations stemming from the upstream SDK:
  - Extensible Authentication Protocol (EAP) is not supported
  - Combined STA/AP mode is not supported
- Multicore is supported, but only core 0 may run `WiFi` related code when in bare metal. When using FreeRTOS, any Task on any core can perform networking operations.

The WiFi library borrows much work from the ESP8266 Arduino Core , especially the `WiFiClient` and `WiFiServer` classes.

41.5 Special Thanks

Special thanks to:

- @todbot for donating one of his Pico W boards to the effort
- @d-a-v for much patient explanation about LWIP internals
- The whole ESP8266 Arduino team for their network classes
- Adafruit Industries for their kind donation

CHAPTER

FORTYTWO

ETHERNETLWIP (WIRED ETHERNET) SUPPORT

Wired Ethernet interfaces are supported for all the internal networking libraries (`WiFiClient`, `WiFiClientSecure`, `WiFiServer`, `WiFiServerSecure`, `WiFiUDP`, `WebServer`, `Updater`, `HTTPClient`, etc.).

Using these wired interfaces is very similar to using the Pico-W WiFi so most examples in the core only require minor modifications to use a wired interface.

42.1 Supported Wired Ethernet Modules

- Wiznet W5100(s)
- Wiznet W5500
- Wiznet W55RP20
- Wiznet W6100
- Wiznet W6300
- ENC28J60
- USB (see *Ethernet over USB*)

42.2 Enabling Wired Ethernet

Simply replace the WiFi include at the top with:

```
#include <W5500lwIP.h> // Or W5100lwIP.h or ENC28J60.h
```

And add a global Ethernet object of the same type:

```
Wiznet5500lwIP eth(1); // Parameter is the Chip Select pin
```

In your `setup()` you may adjust the SPI pins you're using to match your hardware (be sure they are legal for the RP2040!), or skip this if you're using the default ones:

```
void setup() {
    SPI.setRX(0);
    SPI.setCS(1);
    SPI.setSCK(2);
    SPI.setTX(3);
    ....
}
```

And finally replace the `WiFi.begin()` and `WiFi.connected()` calls with `eth.begin()` and `eth.connected()`:

```
void setup() {
    ....
    // WiFi.begin(SSID, PASS)
    eth.begin();

    //while (!WiFi.connected()) {
    while (!eth.connected()) {
        Serial.print(".");
    }

    Serial.print("IP address: ");
    //Serial.println(WiFi.localIP());
    Serial.println(eth.localIP());

    ....
}
```

42.3 Adjusting LWIP Polling

LWIP operates in a polling mode for the wired Ethernet devices. By default it will run every 20ms, meaning that on average it will take half that time (10ms) before a packet received in the Ethernet module is received and operated upon by the Pico. This gives very low CPU utilization but in some cases this latency can affect performance.

Adding a call to `lwipPollingPeriod(XXX)` (where `XXXX` is the polling period in milliseconds) can adjust this setting on the fly. Note that if you set it too low, the Pico may not have enough time to service the Ethernet port before the timer fires again, leading to a lock up and hang.

42.4 Using Interrupt-Driven Handling

The WizNet and ENC28J60 devices support generating an interrupt when a packet is received, removing the need for polling and decreasing latency. Simply specify the SPI object to use and the interrupt pin when instantiating the Ethernet object:

```
#include <W5100lwIP.h>
Wiznet5100lwIP eth(SS /* Chip Select*/, SPI /* SPI interface */, 17 /* Interrupt GPIO */␣
↪);
```

42.5 Adjusting SPI Speed

By default a 4MHz clock will be used to clock data into and out of the Ethernet module. Depending on the module and your wiring, a higher SPI clock may increase performance (but too high of a clock will cause communications problems or hangs).

This value may be adjusted using the `eth.setSPISpeed(hz)` call **before** starting the device. (You may also use custom `SPISettings` instead via `eth.setSPISettings(spis)`)

For example, to set the W5500 to use a 30MHZ clock:

```
#include <W5500lwIP.h>
Wiznet5500lwIP eth(1);

void setup() {
```

(continues on next page)

(continued from previous page)

```
    eth.setSPISpeed(30000000);
    lwipPollingPeriod(3);
    ...
    eth.begin();
    ...
}
```

42.6 Using the WIZnet W5100S-EVB-Pico

You can use the onboard Ethernet chip with these drivers, in interrupt mode, by utilizing the following options:

```
#include <W5100lwIP.h>
Wiznet5100lwIP eth(17, SPI, 21);  // Note chip select is **17**

void setup() {
    // Set SPI to the onboard Wiznet chip
    SPI.setRX(16);
    SPI.setCS(17);
    SPI.setSCK(18);
    SPI.setTX(19);
    ...
    eth.begin();
    ...
}
```

42.7 Example Code

The following examples allow switching between WiFi and Ethernet:

- `WebServer/AdvancedWebServer`
- `HTTPClient/BasicHTTPSClient`

42.8 Caveats

The same restrictions for `WiFi` apply to these Ethernet classes, namely:

- Only core 0 may run any networking related code.
- In FreeRTOS, only the `setup` and `loop` task can call networking libraries, not any tasks.

42.9 Special Thanks

- LWIPEthernet classes come from the ESP8266 Arduino team
- Some individual Ethernet drivers were written by Nicholas Humfrey, others submitted by users.

CHAPTER

FORTYTHREE

ETHERNET OVER USB

A USB connection with a PC or Smartphone can be used as a wired network connection. It requires no additional hardware. Other USB functions like `Serial`, `Keyboard`, `Mouse` or `Joystick` can be used at the same time. Both IPv4 and IPv6 are supported.

The protocol is called Network Control Model (NCM) and is the newest of three possible protocols defined by the USB standard. NCM is natively supported by Windows, Linux, macOS, Android, ChromeOS, iOS and more.

To use it, both the Raspberry Pi Pico and the USB Host need to be configured.

43.1 USB Device configuration on Raspberry Pi Pico

Add this to your sketch:

```
#include <NCMEthernetlwIP.h>
```

And add a global Ethernet object of the same type:

```
NCMEthernetlwIP eth;
```

In your `setup()`, add this:

```
void setup() {
    eth.begin();
    ....
}
```

You can use `eth.connected()` to check the state of the connection.

```
void setup() {
    ....
    eth.begin();

    while (!eth.connected()) {
        Serial.print(".");
    }

    Serial.print("IP address: ");
    Serial.println(eth.localIP());

    ....
}
```

The Raspberry Pi Pico will try to get an IP Address via DHCP and stateless DHCPv6. Alternatively, static addresses may be set:

```
IPAddress my_static_ip_addr(192, 168, 137, 100);
IPAddress my_static_gateway(192, 168, 137, 1);
IPAddress my_static_dns(192, 168, 137, 1);

void setup() {
    ....
    eth.config(my_static_ip_addr, my_static_gateway, IPAddress(255, 255, 255, 0), my_
↪static_dns);
    eth.begin();
    ....
}
```

See also the examples:

- `lwIP_USB_NCM/WiFiClient-NCMEthernet`
- `lwIP_USB_NCM/WiFiClient-NCMEthernet-platformio`

43.2 USB Host configuration on Windows

The Raspberry Pi Pico will appear as a USB to Ethernet adapter, creating an additional network interface on the host. It must be configured to allow a successful network connection.

On Windows we must use Internet Connection Sharing. It provides DHCP on the 192.168.137.0/24 subnet and routing to the rest of the network using network address translation.

1. Win + R, then type `ncpa.cpl`
2. Connect Raspberry Pi Pico with a NCM-enabled sketch
3. A new network connection should appear
4. Right click your **normal** network connection, select *Properties*.
5. Select the *Sharing* Tab
6. Check the box for *Allow other network users to connect through this computer's Internet connection*
7. As *Home networking connection*, choose the **new** network connection that appeared in Step 3.

The Pico may get any address in 192.168.137.0/24. Either print it on a serial terminal or use a static IP address. It will respond to pings if everything is setup correctly.

43.3 USB Host configuration on Linux

The Raspberry Pi Pico will appear as a USB to Ethernet adapter, creating an additional network interface on the host. It must be configured to allow a successful network connection. Network configuration on Linux depends on your network management software. If you don't know which one you are running, try these commands:

- `sudo systemctl status NetworkManager`
- `sudo systemctl status systemd-networkd`
- `sudo systemctl status dhcpcd`

One of them will be `active (running)`.

43.3.1 NetworkManager

1. Prepare your Raspberry Pi Pico with a NCM-enabled sketch and configure it with DHCP or a static ip address.
2. `ip link` to check your existing network interfaces
3. Connect the Raspberry Pi Pico
4. `ip link` again, a new network interface should be listed. This example assumes `usb0`
5. `nmcli connection add type ethernet ifname usb0 con-name pico ipv4.method shared ipv4.address 192.168.142.1/24`
6. `sudo iptables -A FORWARD -i usb0 -j ACCEPT`
7. `sudo iptables -A FORWARD -o usb0 -j ACCEPT`
8. `pyserial-miniterm /dev/ttyACM0`. Check the IP address it reports. Check for successful requests. Alternatives to `pyserial-miniterm` are `picocom` or `minicom`.
9. `ping <ip address from previous step>`
10. make iptables rules persistent by adding them to the files in `/etc/iptables/*.rules`

43.3.2 dhcpcd

dhcpcd cannot act as a DHCP server, so we must use static ip addresses.

1. Prepare your Raspberry Pi Pico with a NCM-enabled sketch and configure it with static ip address `192.168.137.100` and gateway `192.168.137.1`.
2. `ip link` to check your existing network interfaces
3. Connect the Raspberry Pi Pico
4. `ip link` again, a new network interface should be listed. This example assumes `usb0`
5. add the following to `/etc/dhcpcd.conf`:

```
interface usb0
static ip_address=192.168.137.1/24
```

6. `systemctl restart dhcpcd`
7. `ping 192.168.137.100`
8. `sudo sysctl -w net.ipv4.conf.all.forwarding=1`
9. `sudo iptables -A FORWARD -i usb0 -j ACCEPT`
10. `sudo iptables -A FORWARD -o usb0 -j ACCEPT`
11. `pyserial-miniterm /dev/ttyACM0`. Check for successful requests. Alternatives to `pyserial-miniterm` are `picocom` or `minicom`.
12. make iptables rules persistent by adding them to the files in `/etc/iptables/*.rules`

43.3.3 systemd-networkd

1. Prepare your Raspberry Pi Pico with a NCM-enabled sketch and configure it with DHCP or static ip address `192.168.137.100` and gateway `192.168.137.1`.
2. `ip link` to check your existing network interfaces
3. Connect the Raspberry Pi Pico

4. `ip link` again, a new network interface should be listed. This example assumes `usb0`
5. create `/etc/systemd/network/pico.network` with this content:

```
[Match]
Name=usb0

[Network]
Address=192.168.137.1/24

IPv4Forwarding=yes
IPMasquerade=yes
DHCPServer=yes

IPv6Forwarding=yes
IPv6SendRA=yes
DHCPPrefixDelegation=yes

[DHCPServer]
EmitDNS=yes
UplinkInterface=:auto

[IPv6SendRA]
EmitDNS=yes
UplinkInterface=:auto
```

6. `sudo iptables -A FORWARD -i usb0 -j ACCEPT`
7. `sudo iptables -A FORWARD -o usb0 -j ACCEPT`
8. `sudo systemctl restart systemd-networkd`
9. `pyserial-miniterm /dev/ttyACM0`. Check the IP address it reports. Check for successful requests. Alternatives to `pyserial-miniterm` are `picocom` or `minicom`.
10. `ping <ip address from previous step>`
11. make iptables rules persistent by adding them to the files in `/etc/iptables/*.rules`

43.3.4 Manual config

This config will not survive a Linux reboot, but may be ok for testing.

1. Prepare your Raspberry Pi Pico with a NCM-enabled sketch and configure it with static ip address `192.168.137.100` and gateway `192.168.137.1`.
2. `ip link` to check your existing network interfaces
3. Connect the Raspberry Pi Pico
4. `ip link` again, a new network interface should be listed. This example assumes `usb0`
5. You can also check `lsusb -t` and `dmesg`
6. `ip link set usb0 up`
7. `ip addr add 192.168.137.1/24 dev usb0`
8. `ping 192.168.137.100`

43.4 Special Thanks

- tinyUSB contributors for the NCM implementation
- NCMEthernetlwIP driver written by functionpointer

CHAPTER

FORTYFOUR

WIFICLIENT

Methods documented for Client in Arduino

1. WiFiClient()
2. connected()
3. connect()
4. write()
5. print()
6. println()
7. available()
8. read()
9. flush()
10. stop()

Methods and properties described further down are specific to ESP8266. They are not covered in Arduino WiFi library documentation. Before they are fully documented please refer to information below.

44.1 flush and stop

`flush(timeoutMs)` and `stop(timeoutMs)` both have now an optional argument: `timeout` in millisecond, and both return a boolean.

Default input value 0 means that effective value is left at the discretion of the implementer.

`flush()` returning `true` indicates that output data have effectively been sent, and `false` that a timeout has occurred.

`stop()` returns `false` in case of an issue when closing the client (for instance a timed-out `flush`). Depending on implementation, its parameter can be passed to `flush()`.

44.2 setNoDelay

```
setNoDelay(nodelay)
```

With `nodelay` set to `true`, this function will to disable Nagle algorithm.

This algorithm is intended to reduce TCP/IP traffic of small packets sent over the network by combining a number of small outgoing messages, and sending them all at once. The downside of such approach is effectively delaying individual messages until a big enough packet is assembled.

Example:

```
client.setNoDelay(true);
```

44.3 getNoDelay

Returns whether NoDelay is enabled or not for the current connection.

44.4 setSync

This is an experimental API that will set the client in synchronized mode. In this mode, every `write()` is flushed. It means that after a call to `write()`, data are ensured to be received where they went sent to (that is `flush` semantic).

When set to `true` in `WiFiClient` implementation,

- It slows down transfers, and implicitly disable the Nagle algorithm.
- It also allows to avoid a temporary copy of data that otherwise consumes at most `TCP_SND_BUF` = (2 * `MSS`) bytes per connection,

44.5 getSync

Returns whether Sync is enabled or not for the current connection.

44.6 setDefaultNoDelay and setDefaultSync

These set the default value for both `setSync` and `setNoDelay` for every future instance of `WiFiClient` (including those coming from `WiFiServer.available()` by default).

Default values are false for both `NoDelay` and `Sync`.

This means that Nagle is enabled by default *for all new connections*.

44.7 getDefaultNoDelay and getDefaultSync

Return the values to be used as default for NoDelay and Sync for all future connections.

44.8 Other Function Calls

```
uint8_t  status ()
virtual size_t  write (const uint8_t *buf, size_t size)
size_t  write_P (PGM_P buf, size_t size)
size_t  write (Stream &stream)
size_t  write (Stream &stream, size_t unitSize) __attribute__((deprecated))
virtual int  read (uint8_t *buf, size_t size)
virtual int  peek ()
virtual size_t  peekBytes (uint8_t *buffer, size_t length)
size_t  peekBytes (char *buffer, size_t length)
virtual  operator bool ()
IPAddress  remoteIP ()
uint16_t  remotePort ()
IPAddress  localIP ()
uint16_t  localPort ()
```

Documentation for the above functions is not yet available.

CHAPTER

FORTYFIVE

SERVER CLASS

Methods documented for the Server Class in Arduino

1. WiFiServer()
2. begin()
3. available()
4. write()
5. print()
6. println()

In ESP8266WiFi library the `ArduinoWiFiServer` class implements `available` and the write-to-all-clients functionality as described in the Arduino WiFi library reference. The PageServer example shows how `available` and the write-to-all-clients works.

For most use cases the basic WiFiServer class of the ESP8266WiFi library is suitable.

Methods and properties described further down are specific to ESP8266. They are not covered in Arduino WiFi library documentation. Before they are fully documented please refer to information below.

45.1 accept

Method `accept()` returns a waiting client connection. accept() is documented for the Arduino Ethernet library.

45.2 available

> see `accept`

`available` in the ESP8266WiFi library's WiFiServer class doesn't work as documented for the Arduino WiFi library. It works the same way as `accept`.

45.3 write (write to all clients) not supported

Please note that the `write` method on the `WiFiServer` object is not implemented and returns failure always. Use the returned `WiFiClient` object from the `WiFiServer::accept()` method to communicate with individual clients. If you need to send the exact same packets to a series of clients, your application must maintain a list of connected clients and iterate over them manually.

45.4 setNoDelay

```
setNoDelay(nodelay)
```

With `nodelay` set to `true`, this function will to disable Nagle algorithm.

This algorithm is intended to reduce TCP/IP traffic of small packets sent over the network by combining a number of small outgoing messages, and sending them all at once. The downside of such approach is effectively delaying individual messages until a big enough packet is assembled.

Example:

```
server.begin();
server.setNoDelay(true);
```

By default, `nodelay` value will depends on global `WiFiClient::getDefaultNoDelay()` (currently false by default).

However, a call to `wiFiServer.setNoDelay()` will override `NoDelay` for all new `WiFiClient` provided by the calling instance (`wiFiServer`).

45.5 Other Function Calls

```
bool  hasClient ()
size_t  hasClientData ()
bool  hasMaxPendingClients ()
bool  getNoDelay ()
virtual size_t  write (const uint8_t *buf, size_t size)
uint8_t  status ()
void  close ()
void  stop ()
```

Documentation for the above functions is not yet prepared.

CHAPTER

FORTYSIX

UDP CLASS

Methods documented for WiFiUDP Class in Arduino

1. begin()
2. available()
3. beginPacket()
4. endPacket()
5. write()
6. parsePacket()
7. peek()
8. read()
9. flush()
10. stop()
11. remoteIP()
12. remotePort()

CHAPTER

FORTYSEVEN

NETWORK TIME PROTOCOL (NTP)

NTP allows the Pico to set its internal clock using the internet, and is required for secure connections because the certificates used have valid date stamps.

After `WiFi.begin()` use `NTP.begin(s1)` or `NTP.begin(s1, s2)` to use one or two NTP servers (common ones are `pool.ntp.org` and `time.nist.gov`) .

```
WiFi.begin("ssid", "pass");
NTP.begin("pool.ntp.org", "time.nist.gov");
```

Either names or `IPAddress` may be used to identify the NTP server to use.

It may take seconds to minutes for the system time to be updated by NTP, depending on the server. It is often useful to check that `time(NULL)` returns a sane value before continuing a sketch:

```
void setClock() {
  NTP.begin("pool.ntp.org", "time.nist.gov");

  Serial.print("Waiting for NTP time sync: ");
  time_t now = time(nullptr);
  while (now < 8 * 3600 * 2) {
    delay(500);
    Serial.print(".");
    now = time(nullptr);
  }
  Serial.println("");
  struct tm timeinfo;
  gmtime_r(&now, &timeinfo);
  Serial.print("Current time: ");
  Serial.print(asctime(&timeinfo));
}
```

47.1 bool NTP.waitSet(uint32_t timeout)

This call will wait up to timeout milliseconds for the time to be set, and returns success or failure. It will also begin NTP with a default "pool.ntp.org" server if it is not already running. Using this method, the above code becomes:

```
void setClock() {
  NTP.begin("pool.ntp.org", "time.nist.gov");
  NTP.waitSet();
  time_t now = time(nullptr);
  struct tm timeinfo;
```

(continues on next page)

(continued from previous page)

```
  gmtime_r(&now, &timeinfo);
  Serial.print("Current time: ");
  Serial.print(asctime(&timeinfo));
}
```

47.2 bool NTP.waitSet(void (*cb)(), uint32_t timeout)

Allows for a callback that will be called every 1/10th of a second while waiting for NTP sync. For example, using lambdas you can simply print "."s:"

```
void setClock() {
  NTP.begin("pool.ntp.org", "time.nist.gov");
  NTP.waitSet([]() { Serial.print("."); });
  time_t now = time(nullptr);
  struct tm timeinfo;
  gmtime_r(&now, &timeinfo);
  Serial.print("Current time: ");
  Serial.print(asctime(&timeinfo));
}
```

CHAPTER

FORTYEIGHT

ILABS ESP-NOW

ESP-NOW is a connectionless 2.4GHz protocol from Espressif. It lets small packets travel directly between devices without a WiFi access point and without any TCP/IP stack. This is useful for sensor networks, remote controls and other cases where you only need to move a few bytes now and then with low latency.

The RP2040 and RP2350 do not have a radio of their own. On the iLabs Challenger WiFi boards the radio is a separate ESP32-C6 or ESP32-C3 co-processor sitting next to the host processor. This library gives you the Arduino ESP-NOW API on the host and talks to the co-processor over a UART link, using the iLabs `AT+EN` command set. The co-processor must be flashed with the iLabs ESP-NOW interpreter firmware for this to work.

The programming surface is the same `ESP_NOW` object and `ESP_NOW_Peer` class as in the arduino-esp32 core. A sketch written for an ESP32 therefore builds against this library with only the radio bring-up changed, as described below.

```
 RP2040 / RP2350 host              ESP32-C6 co-processor
(this Arduino library)            (AT+EN interpreter)
      |                                    |
      |   UART  (ESP_SERIAL_PORT)          |
      | ESP_NOW.begin()  --> AT+ENINIT -->|             ESP-NOW
      | peer.send()      --> AT+ENSEND -->| ))) 2.4GHz ((( peers
      | ESP_NOW.poll()   <-- +ENRECV -----|
```

48.1 Supported boards

ESP-NOW is available on the iLabs boards that carry an ESP32 co-processor:

- Challenger 2040 WiFi
- Challenger 2040 WiFi/BLE
- Challenger 2040 WiFi6/BLE
- Challenger NB 2040 WiFi
- Connectivity 2040 LTE/WiFi/BLE
- RPICO32
- Challenger 2350 WiFi6/BLE5

On any other board the library will not build. The reason is that it needs a board variant that wires up the ESP32 UART, and that is only true for the boards above.

48.2 Selecting the ESP-NOW firmware type

Pick the board in the IDE, open the Tools menu and set **ESP Wifi Type** to **ESP-NOW**. This defines `ILABS_ESPNOW` for the build. The default value is **ESP AT**, so a board keeps its normal AT firmware behaviour until you change the menu. On the two boards that also offer **ESP Hosted**, ESP-NOW is simply added as a third choice.

When ESP-NOW is selected the board variant no longer resets the ESP32 during start-up. The library takes over the reset sequence instead, so that it can wait for the co-processor to report that it is ready. See *Board reset handling* below.

If you build with PlatformIO the Tools menu is not available. Add the define yourself in `platformio.ini`:

```
build_flags = -DILABS_ESPNOW
```

48.3 Differences from an ESP32 sketch

Because the radio is a co-processor over a UART, and not a radio inside the main chip, two things change compared to a normal ESP32 ESP-NOW sketch.

First, the radio bring-up. On an ESP32 you would start WiFi and set the channel. Here you call one function instead. The library already knows which UART the board variant uses, so the sketch never names a serial port:

```
ESP_NOW.setLink(channel);   // opens the ESP32 UART and resets the co-processor
```

Second, the servicing of incoming packets. Call `ESP_NOW.poll()` once every time through `loop()`, in the same way you would call `ArduinoOTA.handle()`. Received frames are handed to the peer callbacks from inside `poll()`. Keep `loop()` free of long blocking calls. On the RP2040 a plain `delay()` does not yield, so a long `delay()` will hold up the receive path. For work that must happen at a fixed interval, use a timer to raise a flag and do the sending from `loop()` when the flag is set.

Everything else stays the same as in the arduino-esp32 API. Peer subclassing, `begin()`, `add()`, `send()`, `onReceive()`, `onSent()`, `onNewPeer()`, broadcast peers and `ESP_NOW.write()` all behave as before.

48.4 A first sketch

This is a small unicast example. It sends a short message once a second and prints whatever it receives. Set the peer MAC address to the address of the other board.

```
#include <Arduino.h>
#include "ESP32_NOW.h"

class Peer : public ESP_NOW_Peer {
public:
    Peer(const uint8_t *mac) : ESP_NOW_Peer(mac, 6, WIFI_IF_STA, nullptr) {}
    void onReceive(const uint8_t *data, size_t len, bool broadcast) {
        Serial.write(data, len);
        Serial.println();
    }
    using ESP_NOW_Peer::add;
    using ESP_NOW_Peer::send;
};

uint8_t peerMac[6] = { 0xF0, 0xF5, 0xBD, 0x31, 0x9B, 0xB0 };
Peer peer(peerMac);
```

(continues on next page)

(continued from previous page)

```
void setup() {
    Serial.begin(115200);
    ESP_NOW.setLink(6);   // channel 6, uses the board variant's ESP32 UART
    ESP_NOW.begin();
    peer.add();
}

void loop() {
    peer.send((const uint8_t *)"hello", 5);
    for (uint32_t t = millis(); millis() - t < 1000;) {
        ESP_NOW.poll();
        delay(5);
    }
}
```

More examples are installed with the library under **File, Examples**. They include the two canonical arduino-esp32 examples, **Broadcast_Master** and **Broadcast_Slave**, a **Unicast_PingPong**, a **Discovery_PingPong** that finds the other board by itself, and a **RegressionSuite** that runs a two-board self test of the whole API.

48.5 Board reset handling

`setLink()` opens the ESP32 UART. When the board variant also describes the ESP32 reset pins, which all the iLabs Challenger WiFi and WiFi6 boards do, the library performs a hardware reset of the co-processor into run mode and waits for its `+ENREADY` message before it returns. Every time the host starts, by power-on or by reset, the co-processor is therefore given a clean cold start with no peers, keys or baud rate left over from an earlier session.

If the co-processor restarts on its own while running, for example after a brownout, it sends `+ENREADY` again and the library records it. You can react to this by registering a handler, or by polling `wasReset()` from `loop()`:

```
ESP_NOW.onReset([](void *) {
    // The co-processor lost its peers and keys. The simplest recovery
    // is a full restart of the host.
    rp2040.reboot();
}, nullptr);
```

48.6 Discovery

`ESP_NOW.discover()` is an iLabs addition and has no arduino-esp32 counterpart. It sends a broadcast probe, and every board that runs the interpreter answers in firmware without any help from its host. This is a simple way to find the other boards on the current channel without hard-coding MAC addresses:

```
ESP_NOW_Found found[8];
int n = ESP_NOW.discover(found, 8, 1000);   // collect answers for 1 second
for (int i = 0; i < n; i++) {
    // found[i].mac   the responder STA MAC address
    // found[i].rssi  the signal strength this board measured
}
```

The call blocks for the collection window and returns the number of boards found, or -1 on error. It only covers the channel that is currently in use.

48.7 Notes and limitations

- The callbacks run in the context of whoever calls `poll()`, not in a background task. Keep them short, as you would on the ESP32.
- You may call library methods such as `peer.send()` or `add()` from inside a callback. The library defers such calls and runs them in order once the callback returns. For that reason a command issued from a callback returns before it has actually run, so do not issue a query from a callback and expect its answer straight away.
- `getMaxDataLen()` reports 250 bytes, which is the ESP-NOW version 1 limit. A unicast send above 248 bytes is fragmented by the firmware for you.
- Per-peer PHY rate, RSSI and statistics accessors, and ESP-NOW version 2 framing are not wrapped yet.

CHAPTER

FORTYNINE

ILABS HEARTH (MATTER)

Matter is a smart-home standard from the Connectivity Standards Alliance (CSA) that lets lights, sensors and appliances from different vendors join one fabric and be driven by one controller, over WiFi, Thread or Ethernet.

The RP2040 and RP2350 have no radio of their own. On the iLabs Challenger WiFi6 boards, Matter is spoken by a separate ESP32-C6 co-processor running the **iLabs Hearth** firmware. This library gives you the Arduino Matter API on the host and talks to the co-processor over a UART link, using the iLabs `AT+MT` command set. The co-processor must be flashed with the Hearth firmware for any of this to work; see *Flashing the co-processor* below.

```
 RP2040 / RP2350 host              ESP32-C6 co-processor
(this Arduino library)               (Hearth firmware)
      |                                    |
      |   UART  (ESP_SERIAL_PORT)          |
      | Matter.begin()   --> AT+MTEP  --->|          BLE commissioning
      | light.setOnOff() --> AT+MTATTR -->| ))) then WiFi or Thread (((
      | Hearth.poll()    <-- +MTEVT ------|
```

The programming surface is source-compatible with the arduino-esp32 `Matter*` endpoint class API. A sketch written against the arduino-esp32 core therefore builds against this library largely unmodified, with the firmware bring-up and Tools menu described below as the only difference. One ordering rule applies to every sketch: call `Matter.begin()` last, after every endpoint object has been constructed and initialised. Endpoints declared after `Matter.begin()` are not seen by the reconcile that runs inside it.

49.1 Supported boards

Matter over Hearth is available on the two iLabs Challenger boards that carry an ESP32-C6 co-processor and offer the menu described below:

- Challenger RP2350 WiFi6/BLE5 (`challenger_2350_wifi6_ble5`)
- Challenger RP2040 WiFi6/BLE (`challenger_2040_wifi6_ble`)

49.2 Choosing a firmware

Pick the board in the IDE, open the Tools menu and set **ESP Wifi Type** to one of the three Hearth options:

- **Hearth (Matter, WiFi)**: the usual choice. Commissioning is over BLE; the C6 joins WiFi on the credentials it is handed during commissioning.
- **Hearth (Matter, Thread)**: for a Thread network with a border router. Commissioning is still over BLE.
- **Hearth (Matter, WiFi+Thread)**: one image that can run either transport, selected at runtime. It carries both stacks, so it has less free RAM than a single-transport image; see *How many endpoints fit*.

Every option defines `-DILABS_HEARTH` for the sketch build. In every case, commissioning itself happens over BLE: the operational transport (WiFi or Thread) only comes into play afterwards, once the controller has handed the C6 its network credentials.

49.3 Flashing the co-processor

Building a sketch with a Hearth option selected only builds the sketch. The co-processor needs its own firmware, and it is written from a terminal with the flasher the library ships, the same way the iLabs ESP-NOW firmware is. The IDE has no menu item for it: the Arduino platform offers a core no action hook that fits, so the flasher is a command line tool on purpose.

The library's `fw/` directory holds the three prebuilt images, the USB-to-serial bridge for the RP2040/RP2350 host, and `flash.py`. From the core's `libraries/iLabs_Hearth/fw` directory:

```
python3 flash.py                  # interactive: pick WiFi, Thread or WiFi+Thread
python3 flash.py --variant wifi   # non-interactive
python3 flash.py --list           # verify esptool, list the variants, exit
python3 flash.py --dry-run        # show what would happen, write nothing
```

Pick the variant that matches the **ESP Wifi Type** option you build the sketch with. The flasher resets the host into its USB boot mode on its own (no BOOTSEL press needed when exactly one board is attached, or with `--port`), copies the bridge, flashes the C6 through it with a progress bar, and verifies every file it wrote against `fw/manifest.json`.

Requirements, the same as for the ESP-NOW flasher:

- **The iLabs fork of esptool.** It adds the reset profile that holds the C6 in its download bootloader while the RP2040/RP2350 bridges; stock esptool cannot flash these boards and `flash.py` refuses to run without the fork. Clone https://github.com/PontusO/esptool and point the flasher at it with `ILABS_ESPTOOL_PATH` or `--esptool-path`.
- **pyserial** (`pip install pyserial`).
- **rich** is optional (`pip install rich`) for nicer output.

Flashing the C6 overwrites the sketch on the host. The bridge is a sketch written onto the RP2040/RP2350 in place of whatever was there, so upload your sketch again afterwards.

The flasher works on Linux and macOS only. It finds the host's mass-storage mount only on those platforms, and the esptool fork skips the bridge reset profile on Windows. Flash from a Linux or macOS machine.

49.4 Writing a sketch

A sketch declares its endpoints the same way an arduino-esp32 Matter sketch would, then calls `Matter.begin()` once, last:

```
#include <Matter.h>

MatterOnOffLight light;

void setup() {
    Serial.begin(115200);
    light.begin();
    Matter.begin();   // reconciles the declared endpoints against the C6,
                      // called last, after every endpoint's own begin()
}
```

(continues on next page)

(continued from previous page)

```
void loop() {
    Hearth.poll();     // services URCs from the co-processor; call every loop()
}
```

The library covers 52 Matter device types in total: the twenty arduino-esp32 `Matter*` endpoint classes, matching upstream one for one, are demonstrated under `examples/Matter*`, and the remaining beyond-parity types the Hearth firmware also supports are demonstrated under `examples/FullAPI`.

49.5 How many endpoints fit

24 is the ceiling from the Arduino IDE, and there is no way around it from here. The firmware itself accepts up to 28 endpoints (`MT_COMP_MAX_ENDPOINTS`), but `HEARTH_MAX_ENDPOINTS` (`src/MatterEndPoint.h`) sizes the host-side declaration registry at 24, and a sketch cannot declare a 25th endpoint: the registry refuses it before the C6 is ever asked. Raising the define needs a `-D` on the compiler command line, and this core has no `build_opt.h` hook for a sketch folder to add one (the `esp32` core has one; this one does not), so 24 is the ceiling for any sketch built through the IDE.

On the WiFi+Thread image with WiFi as the active transport, the practical ceiling is lower still, because that image keeps both stacks linked and has less free RAM: roughly 20 endpoints for a typical mix of simple device types, and roughly 12 for a mix weighted toward the energy device types.

49.6 Status and limitations

This firmware is uncertified and uses Matter's public development credentials (vendor ID `0xFFF1`). Every board running it presents the same pairing code. Consumer hubs such as Apple Home, Google Home and Alexa are expected to refuse it, either outright or behind an uncertified-accessory warning; that is expected of any Matter project before its vendor pays for certification, not a defect in this firmware or this library. A verified path to commissioning it is the NXP Matter Chip-tool Android app, or the CLI `chip-tool`.

CHAPTER

FIFTY

BEARSSL WIFI CLASSES

Methods and properties described in this section are specific to the Raspberry Pi Pico W and the ESP8266. They are not covered in Arduino WiFi library documentation. Before they are fully documented please refer to information below.

The BearSSL library (with modifications for ESP8266 compatibility and to use ROM tables whenever possible) is used to perform all cryptography and TLS operations. The main ported repo is available on GitHub.

50.1 CPU Requirements

SSL operations take significant CPU cycles to run, so it will connect significantly slower than unprotected connections on the Pico, but the actual data transfer rates once connected are similar.

See the section on *sessions* and *limiting cryptographic negotiation* for ways of ensuring faster modes are used.

50.2 Memory Requirements

BearSSL doesn't perform memory allocations at runtime, but it does require allocation of memory at the beginning of a connection. There are two memory chunks required: . A per-application secondary stack . A per-connection TLS receive/transmit buffer plus overhead

The per-application secondary stack is approximately 7KB in size and is used for temporary variables during BearSSL processing. Only one stack is required, and it will be allocated whenever any *BearSSL::WiFiClientSecure* or *BearSSL::WiFiServerSecure* are instantiated. So, in the case of a global client or server, the memory will be allocated before *setup()* is called.

The per-connection buffers are approximately 22KB in size, but in certain circumstances it can be reduced dramatically by using MFLN or limiting message sizes. See the *MLFN section* below for more information.

50.3 Object Lifetimes

There are many configuration options that require passing in a pointer to an object (i.e. a pointer to a private key, or a certificate list). In order to preserve memory, BearSSL does NOT copy the objects passed in via these pointers and as such any pointer passed in to BearSSL needs to be preserved for the life of the client object. For example, the following code is **in error**:

```
BearSSL::WiFiClientSecure client;
const char x509CA PROGMEM = ".......";
void setup() {
    BearSSL::X509List x509(x509CA);
    client.setTrustAnchor(&x509);
}
```

(continues on next page)

(continued from previous page)

```
void loop() {
    client.connect("192.168.1.1", 443);
}
```

Because the pointer to the local object *x509* no longer is valid after setup(), expect to crash in the main *loop()* where it is accessed by the *client* object.

As a rule, either keep your objects global, use *new* to create them, or ensure that all objects needed live inside the same scope as the client.

50.4 TLS and HTTPS Basics

The following discussion is only intended to give a rough idea of TLS/HTTPS(which is just HTTP over a TLS connection) and the components an application needs to manage to make a TLS connection. For more detailed information, please check the relevant RFC 5246 and others.

TLS can be broken into two stages: verifying the identities of server (and potentially client), and then encrypting blocks of data bidirectionally. Verifying the identity of the other partner is handled via keys encoded in X509 certificates, optionally signed by a series of other entities.

50.5 Public and Private Keys

Cryptographic keys are required for many of the BearSSL functions. Both public and private keys are supported, with either Elliptic Curve or RSA key support.

To generate a public or private key from an existing PEM (ASCII format) or DER (binary format), the simplest method is to use the constructor:

```
BearSSL::PublicKey(const char *pemString)
... or ...
BearSSL::PublicKey(const uint8_t *derArray, size_t derLen)
```

Note that *PROGMEM* strings and arrays are natively supported by these constructors and no special *_P modes are required. There are additional functions to identify the key type and access the underlying BearSSL proprietary types, but they are not needed by user applications.

50.6 TLS Sessions

TLS supports the notion of a session (completely independent and different from HTTP sessions) which allow clients to reconnect to a server without having to renegotiate encryption settings or validate X509 certificates. This can save significant time (3-4 seconds in the case of EC keys) and can help save power by allowing the ESP8266 to sleep for a long time, reconnect and transmit some samples using the SSL session, and then jump back to sleep quicker.

BearSSL::Session is an opaque class. Use the *BearSSL::WiFiClientSecure.setSession(&BearSSLSession)* method to apply it before the first *BearSSL::WiFiClientSecure.connect()* and it will be updated with session parameters during the operation of the connection. After the connection has had *.close()* called on it, serialize the *BearSSL::Session* object to stable storage (EEPROM, RTC RAM, etc.) and restore it before trying to reconnect. See the *BearSSL_Sessions* example for a detailed example.

Sessions contains additional information on the sessions API.

50.7 X.509 Certificate(s)

X509 certificates are used to identify peers in TLS connections. Normally only the server identifies itself, but the client can also supply an X509 certificate if desired (this is often done in MQTT applications). The certificate contains many fields, but the most interesting in our applications are the name, the public key, and potentially a chain of signing that leads back to a trusted authority (like a global internet CA or a company-wide private certificate authority).

Any call that takes an X509 certificate can also take a list of X509 certificates, so there is no special *X509* class, simply *BearSSL::X509List* (which may only contain a single certificate).

Generating a certificate to be used to validate using the constructor

```
BearSSL::X509List(const char *pemX509);
...or...
BearSSL::X509List(const uint8_t *derCert, size_t derLen);
```

If you need to add additional certificates (unlikely in normal operation), the *::append()* operation can be used.

50.8 Certificate Stores

The web browser you're using to read this document keeps a list of 100s of certification authorities (CAs) worldwide that it trusts to attest to the identity of websites.

In many cases your application will know the specific CA it needs to validate web or MQTT servers against (often just a single, self-signing CA private to your institution). Simply load your private CA in a *BearSSL::X509List* and use that as your trust anchor.

However, there are cases where you will not know beforehand which CA you will need (i.e. a user enters a website through a keypad), and you need to keep the list of CAs just like your web browser. In those cases, you need to generate a certificate bundle on the PC while compiling your application, upload the *certs.ar* bundle to LittleFS or SD when uploading your application binary, and pass it to a *BearSSL::CertStore()* in order to validate TLS peers.

See the *BearSSL_CertStore* example for full details.

50.9 Supported Crypto

Please see the BearSSL website for detailed cryptographic information. In general, TLS 1.2, TLS 1.1, and TLS 1.0 are supported with RSA and Elliptic Curve keys and a very rich set of hashing and symmetric encryption codes. Please note that Elliptic Curve (EC) key operations take a significant amount of time.

CHAPTER

FIFTYONE

WIFICLIENTSECURE CLASS

BearSSL::WiFiClientSecure is the object which actually handles TLS encrypted WiFi connections to a remote server or client. It extends *WiFiClient* and so can be used with minimal changes to code that does unsecured communications.

51.1 Validating X509 Certificates (Am I talking to the server I think I'm talking to?)

Prior to connecting to a server, the *BearSSL::WiFiClientSecure* needs to be told how to verify the identity of the other machine. **By default BearSSL will not validate any connections and will refuse to connect to any server.**

There are multiple modes to tell BearSSL how to verify the identity of the remote server. See the *BearSSL_Validation* example for real uses of the following methods:

51.1.1 setInsecure()

Don't verify any X509 certificates. There is no guarantee that the server connected to is the one you think it is in this case.

51.1.2 setKnownKey(const BearSSL::PublicKey *pk)

Assume the server is using the specific public key. This does not verify the identity of the server or the X509 certificate it sends, it simply assumes that its public key is the one given. If the server updates its public key at a later point then connections will fail.

51.1.3 setFingerprint(const uint8_t fp[20]) / setFingerprint(const char *fpStr)

Verify the SHA1 fingerprint of the certificate returned matches this one. If the server certificate changes, it will fail. If an array of 20 bytes are sent in, it is assumed they are the binary SHA1 values. If a *char** string is passed in, it is parsed as a series of human-readable hex values separated by spaces or colons (e.g. *setFingerprint("00:01:02:03:...:1f");*)

This fingerprint is calculated on the raw X509 certificate served by the server. In very rare cases, these certificates have certain encodings which should be normalized before taking a fingerprint (but in order to preserve memory BearSSL does not do this normalization since it would need RAM for an entire copy of the cert), and the fingerprint BearSSL calculates will not match the fingerprint OpenSSL calculates. In this case, you can enable SSL debugging and get a dump of BearSSL's calculated fingerprint and use that one in your code, or use full certificate validation. See the original issue and debug here.

51.1.4 setTrustAnchors(BearSSL::X509List *ta)

Use the passed-in certificate(s) as a trust anchor, accepting remote certificates signed by any of these. If you have many trust anchors it may make sense to use a *BearSSL::CertStore* because it will only require RAM for a single trust anchor (while the *setTrustAnchors* call requires memory for all certificates in the list).

51.1.5 setX509Time(time_t now)

For *setTrustAnchors* and *CertStore* , the current time (set via SNTP) is used to verify the certificate against the list, so SNTP must be enabled and functioning before the connection is attempted. If you cannot use SNTP for some reason, you can manually set the "present time" that BearSSL will use to validate a certificate with this call where *now* is standard UNIX time.

51.2 Client Certificates (Proving I'm who I say I am to the server)

TLS servers can request that a client identify themselves with an X509 certificate signed by a trust anchor it honors (i.e. a global TA or a private CA). This is commonly done for applications like MQTT. By default the client doesn't send a certificate, and in cases where a certificate is required the server will disconnect and no connection will be possible.

51.2.1 setClientRSACert / setClientECCert

Sets a client certificate to send to a TLS server that requests one. It should be called before *connect()* to add a certificate to the client in case the server requests it. Note that certificates include both a certificate and a private key. Both should be provided to you by your certificate generator. Elliptic Curve (EC) keys require additional information, as shown in the prototype.

51.3 MFLN or Maximum Fragment Length Negotiation (Saving RAM)

Because TLS was developed on systems with many megabytes of memory, they require by default a 16KB buffer for receive and transmit. That's enormous for the ESP8266, which has only around 40KB total heap available.

We can (and do) minimize the transmission buffer down to slightly more than 512 bytes to save memory, since BearSSL can internally ensure transmissions larger than that are broken up into smaller chunks that do fit. But that still leaves the 16KB receive buffer requirement since we cannot in general guarantee the TLS peer will send in smaller chunks.

TLS 1.2 added MFLN, which lets a client negotiate smaller buffers with a server and reduce the memory requirements on the ESP8266. Unfortunately, BearSSL needs to know the buffer sizes before it begins connection, so applications that want to use smaller buffers need to check the remote server's support before *connect()* .

51.3.1 probeMaxFragmentLength(host, port, len)

Use one of these calls **before** connection to determine if a specific fragment length is supported (len must be a power of two from 512 to 4096, per the specification). This does **not** initiate a SSL connection, it simply opens a TCP port and performs a trial handshake to check support.

51.3.2 setBufferSizes(int recv, int xmit)

Once you have verified (or know beforehand) that MFLN is supported you can use this call to set the size of memory buffers allocated by the connection object. This must be called **before** *connect()* or it will be ignored.

In certain applications where the TLS server does not support MFLN (not many do as of this writing as it is relatively new to OpenSSL), but you control both the ESP8266 and the server to which it is communicating, you may still be able to *setBufferSizes()* smaller if you guarantee no chunk of data will overflow those buffers.

51.3.3 bool getMFLNStatus()

After a successful connection, this method returns whether or not MFLN negotiation succeeded or not. If it did not succeed, and you reduced the receive buffer with *setBufferSizes* then you may experience reception errors if the server attempts to send messages larger than your receive buffer.

51.4 Sessions (Resuming connections fast)

51.4.1 setSession(BearSSL::Session &sess)

If you are connecting to a server repeatedly in a fixed time period (usually 30 or 60 minutes, but normally configurable at the server), a TLS session can be used to cache crypto settings and speed up connections significantly.

51.5 Errors

BearSSL can fail in many more unique and interesting ways. Use these calls to get more information when something fails.

51.5.1 getLastSSLError(char *dest = NULL, size_t len = 0)

Returns the last BearSSL error code encountered and optionally set a user-allocated buffer to a human-readable form of the error. To only get the last error integer code, just call without any parameters (*int errCode = getLastSSLError();*).

51.6 Limiting Ciphers (New connections faster)

There is very rarely reason to use these calls, but they are available.

51.6.1 setCiphers()

Takes an array (in PROGMEM is valid) or a std::vector of 16-bit BearSSL cipher identifiers and restricts BearSSL to only use them. If the server requires a different cipher, then connection will fail. Generally this is not useful except in cases where you want to connect to servers using a specific cipher. See the BearSSL headers for more information on the supported ciphers.

51.6.2 setCiphersLessSecure()

Helper function which essentially limits BearSSL to less secure ciphers than it would natively choose, but they may be helpful and faster if your server depended on specific crypto options.

51.7 Limiting TLS(SSL) Versions

By default, BearSSL will connect with TLS 1.0, TLS 1.1, or TLS 1.2 protocols (depending on the request of the remote side). If you want to limit to a subset, use the following call:

51.8 setTLSConnectTimeout(int connectTimeout)

Because the Pico doesn't have any cryptographic acceleration, TLS connections can take up to 10 or 15 seconds to complete whereas the default *Stream* and *WiFiClient* timeout is only 5 seconds. To allow for this large difference, the timeout used for a *WiFiClientSecure::connect()* call is different from that used for a normal read or write (or normal *WiFiClient::connect()* call).

By default, the connection phase of a TLS *WiFiClientSecure::connect()* is set to 15 seconds (all other read/writes use the value set with *setTimeout*). If your application needs to reduce this connection timeout, simply call:

```
WiFiClientSecure::setTLSConnectTimeout(5000); // Value is in milliseconds
```

This setting is global for all TLS connections.

51.8.1 setSSLVersion(uint32_t min, uint32_t max)

Valid values for min and max are *BR_TLS10*, *BR_TLS11*, *BR_TLS12*. Min and max may be set to the same value if only a single TLS version is desired.

51.9 ESP32 Compatibility

Simple ESP32 `WiFiClientSecure` compatibility is built-in, allow for some sketches to run without any modification. The following methods are implemented:

```
void setCACert(const char *rootCA);
void setCertificate(const char *client_ca);
void setPrivateKey(const char *private_key);
bool loadCACert(Stream& stream, size_t size);
bool loadCertificate(Stream& stream, size_t size);
bool loadPrivateKey(Stream& stream, size_t size);
int connect(IPAddress ip, uint16_t port, int32_t timeout);
int connect(const char *host, uint16_t port, int32_t timeout);
int connect(IPAddress ip, uint16_t port, const char *rootCABuff, const char *cli_cert,␣
↪const char *cli_key);
int connect(const char *host, uint16_t port, const char *rootCABuff, const char *cli_
↪cert, const char *cli_key);
```

Note that the SSL backend is very different between Arduino-Pico and ESP32-Arduino (BearSSL vs. mbedTLS). This means that, for instance, the SSL connection will check valid dates of certificates (and hence require system time to be set on the Pico, which is automatically done in this case).

TLS-Pre Shared Keys (PSK) is not supported by BearSSL, and hence not implemented here. Neither is ALPN.

For more advanced control, it is recommended to port to the native Pico calls which allows much more flexibility and control.

CHAPTER

FIFTYTWO

WIFISERVERSECURE CLASS

Implements a TLS encrypted server with optional client certificate validation. See Server Class for general information and BearSSL Secure Client Class for basic server and BearSSL concepts.

52.1 setBufferSizes(int recv, int xmit)

Similar to the *BearSSL::WiFiClientSecure* method, sets the receive and transmit buffer sizes. Note that servers cannot request a buffer size from the client, so if these are shrunk and the client tries to send a chunk larger than the receive buffer, it will always fail. Needs to be called before *begin()*

52.2 Setting Server Certificates

TLS servers require a certificate identifying itself and containing its public key, and a private key they will use to encrypt information with. The application author is responsible for generating this certificate and key, either using a self-signed generator or using a commercial certification authority. **Do not reuse the certificates included in the examples provided.**

This example command will generate a RSA 2048-bit key and certificate:

```
openssl req -x509 -nodes -newkey rsa:2048 -keyout key.pem -out cert.pem -days 4096
```

Again, it is up to the application author to generate this certificate and key and keep the private key safe and **private.**

52.2.1 setRSACert(const BearSSL::X509List *chain, const BearSSL::PrivateKey *sk)

Sets a RSA certificate and key to be used by the server when connections are received. Needs to be called before *begin()*

52.2.2 setECCert(const BearSSL::X509List *chain, unsigned cert_issuer_key_type, const BearSSL::PrivateKey *sk)

Sets an elliptic curve certificate and key for the server. Needs to be called before *begin()*.

52.3 Client sessions (Resuming connections fast)

The TLS handshake process takes a long time because of all the back and forth between the client and the server. You can shorten it by caching the clients' sessions which will skip a few steps in the TLS handshake. In order for this to work, your client also needs to cache the session. BearSSL::WiFiClientSecure can do that as well as modern web browsers.

Here are the kind of performance improvements that you'll be able to see for TLS handshakes with an ESP8266 with it's clock set at 160MHz on a network with fairly low latency:

- With an EC key of 256 bits, a request taking ~360ms without caching takes ~60ms with caching.
- With an RSA key of 2048 bits, a request taking ~1850ms without caching takes ~70ms with caching.

52.3.1 setCache(BearSSL::ServerSessions *cache)

Sets the cache for the server's sessions. When choosing the size of the cache, remember that each client session takes 100 bytes. If you setup a cache for 10 sessions, it will take 1000 bytes. Needs to be called before *begin()*

When creating the cache, you can use any of the 2 available constructors:

- *BearSSL::ServerSessions(ServerSession *sessions, uint32_t size)*: Creates a cache with the given buffer and number of sessions.
- *BearSSL::ServerSessions(uint32_t size)*: Dynamically allocates a cache for the given number of sessions.

52.4 Requiring Client Certificates

TLS servers can request the client to identify itself by transmitting a certificate during handshake. If the client cannot transmit the certificate, the connection will be dropped by the server.

52.4.1 setClientTrustAnchor(const BearSSL::X509List *client_CA_ta)

Sets the trust anchor (normally a self-signing CA) that all received certificates will be verified against. Needs to be called before *begin()*.

CHAPTER

FIFTYTHREE

HTTPCLIENT LIBRARY

A simple HTTP requester that can handle both HTTP and HTTPS requests is included as the `HTTPClient` library.

Check the examples for use under HTTP and HTTPS configurations. In general, for HTTP connections (unsecured and very uncommon on the internet today) simply passing in a URL and performiung a GET is sufficient to transfer data.

```
// Error checking is left as an exercise for the reader...
HTTPClient http;
if (http.begin("http://my.server/url")) {
    if (http.GET() > 0) {
        String data = http.getString();
    }
    http.end();
}
```

For HTTPS connections, simply add the appropriate WiFiClientSecure calls as needed (i.e. `setInsecure()`, `setTrustAnchor`, etc.). See the WiFiClientSecure documentation for more details.

```
// Error checking is left as an exercise for the reader...
HTTPClient https;
https.setInsecure();  // Use certs, but do not check their authenticity
if (https.begin("https://my.secure.server/url")) {
    if (https.GET() > 0) {
        String data = https.getString();
    }
    https.end();
}
```

Unlike the ESP8266 and ESP32 `HTTPClient` implementations it is not necessary to create a `WiFiClient` or `WiFiClientSecure` to pass in to the `HTTPClient` object.

CHAPTER

FIFTYFOUR

OTA UPDATES

54.1 Introduction

OTA (Over the Air) update is the process of uploading firmware to a Pico using a Wi-Fi, Ethernet, or other connection rather than a serial port. This is especially useful for WiFi enabled Picos, like the Pico W, because it lets systems be updated remotely, without needing physical access.

OTA may be done using:

- *Arduino IDE*
- *Web Browser*
- *HTTP Server*
- Any other method (ZModem receive over a UART port, etc.) by using the `Updater` object in your sketch

The Arduino IDE option is intended primarily for the software development phase. The other two options would be more useful after deployment, to provide the module with application updates either manually with a web browser, or automatically using an HTTP server.

In any case, the first firmware upload has to be done over a serial port. If the OTA routines are correctly implemented in the sketch, then all subsequent uploads may be done over the air.

By default, there is no imposed security for the OTA process. It is up to the developer to ensure that updates are allowed only from legitimate / trusted sources. Once the update is complete, the module restarts, and the new code is executed. The developer should ensure that the application running on the module is shut down and restarted in a safe manner. Chapters below provide additional information regarding security and safety of OTA updates.

54.1.1 OTA Requirements

OTA requires a LittleFS partition to store firmware upgrade files. Make sure that you configure the sketch with a filesystem large enough to handle whatever size firmware binary you expect. Updates may be compressed, minimizing the total space needed.

54.1.2 Power Fail Safety

The update commands are all stored in flash, so a power cycle during update (except if the OTA bootloader is being changed) should not brick the device because when power is restored the OTA bootloader will begin the process from scratch once again.

54.1.3 Security Disclaimer

No guarantees as to the level of security provided for your application by the following methods is implied. Please refer to the GNU LGPL license associated for this project for full disclaimers. If you do find security weaknesses, please don't hesitate to contact the maintainers or supply pull requests with fixes. The MD5 verification and password protection schemes are already known to supply a very weak level of security.

54.1.4 Basic Security

The module has to be exposed wirelessly to get it updated with a new sketch. That poses a risk of the module being violently hacked and programmed with some other code. To reduce the likelihood of being hacked, consider protecting your uploads with a password, selecting certain OTA port, etc.

Check functionality provided with the ArduinoOTA library that may improve security:

```
void setPort(uint16_t port);
void setHostname(const char* hostname);
void setPassword(const char* password);
```

Certain basic protection is already built in and does not require any additional coding by the developer. ArduinoOTA and espota.py use Digest-MD5 to authenticate uploads. Integrity of transferred data is verified on the Pico side using MD5 checksum.

Make your own risk analysis and, depending on the application, decide what library functions to implement. If required, consider implementation of other means of protection from being hacked, like exposing modules for uploads only according to a specific schedule, triggering OTA only when the user presses a dedicated "Update" button wired to the Pico, etc.

54.1.5 Advanced Security - Signed Updates

While the above password-based security will dissuade casual hacking attempts, it is not highly secure. For applications where a higher level of security is needed, cryptographically signed OTA updates can be required. This uses SHA256 hashing in place of MD5 (which is known to be cryptographically broken) and RSA-2048 bit level public-key encryption to guarantee that only the holder of a cryptographic private key can produce signed updates accepted by the OTA update mechanisms.

Signed updates are updates whose compiled binaries are signed with a private key (held by the developer) and verified with a public key (stored in the application and available for all to see). The signing process computes a hash of the binary code, encrypts the hash with the developer's private key, and appends this encrypted hash (also called a signature) to the binary that is uploaded (via OTA, web, or HTTP server). If the code is modified or replaced in any way by anyone except the holder of the developer's private key, the signature will not match and the Pico will reject the upload.

Cryptographic signing only protects against tampering with binaries delivered via OTA. If someone has physical access, they will always be able to flash the device over the serial port. Signing also does not encrypt anything but the hash (so that it can't be modified), so this does not protect code inside the device: if a user has physical access they can read out your program.

Securing your private key is paramount. The same private/public key pair that was used with the original upload must also be used to sign later binaries. Loss of the private key associated with a binary means that you will not be able to OTA-update any of your devices in the field. Alternatively, if someone else copies the private key, then they will be able to use it to sign binaries which will be accepted by the Pico.

Signed Binary Format

The format of a signed binary is compatible with the standard binary format, and can be uploaded to a non-signed Pico via serial or OTA without any conditions. Note, however, that once an unsigned OTA app is overwritten by this signed version, further updates will require signing.

As shown below, the signed hash is appended to the unsigned binary, followed by the total length of the signed hash (i.e., if the signed hash was 64 bytes, then this uint32 data segment will contain 64). This format allows for extensibility (such as adding a CA-based validation scheme allowing multiple signing keys all based on a trust anchor). Pull requests are always welcome. (currently it uses SHA256 with RSASSA-PKCS1-V1_5-SIGN signature scheme from RSA PKCS #1 v1.5)

```
NORMAL-BINARY <SIGNATURE> <uint32 LENGTH-OF-SIGNATURE>
```

Signed Binary Prerequisites

OpenSSL is required to run the standard signing steps, and should be available on any UNIX-like or Windows system. As usual, the latest stable version of OpenSSL is recommended.

Signing requires the generation of an RSA-2048 key (other bit lengths are supported as well, but 2048 is a good selection today) using any appropriate tool. The following shell commands will generate a new public/private key pair. Run them in the sketch directory:

```
openssl genrsa -out private.key 2048
openssl rsa -in private.key -outform PEM -pubout -out public.key
```

Automatic Signing

The simplest way of implementing signing is to use the automatic mode, which presently is only possible on Linux and Mac due to some of the tools not being available for Windows. This mode uses the IDE to configure the source code to enable signing verification with a given public key, and signs binaries as part of the standard build process using a given public key.

To enable this mode, just include *private.key* and *public.key* in the sketch *.ino* directory. The IDE will call a helper script (*tools/signing.py*) before the build begins to create a header to enable key validation using the given public key, and to actually do the signing after the build process, generating a *sketch.bin.signed* file. When OTA is enabled (ArduinoOTA, Web, or HTTP), the binary will automatically only accept signed updates.

When the signing process starts, the message:

```
Enabling binary signing
```

will appear in the IDE window before a compile is launched. At the completion of the build, the signed binary file well be displayed in the IDE build window as:

```
Signed binary: /full/path/to/sketch.bin.signed
```

If you receive either of the following messages in the IDE window, the signing was not completed and you will need to verify the *public.key* and *private.key*:

```
Not enabling binary signing
... or ...
Not signing the generated binary
```

Manual Signing of Binaries

Users may also manually sign executables and require the OTA process to verify their signature. In the main code, before enabling any update methods, add the following declarations and function call:

```
<in globals>
BearSSL::PublicKey signPubKey( ... key contents ... );
BearSSL::HashSHA256 hash;
BearSSL::SigningVerifier sign( &signPubKey );
...
<in setup()>
Update.installSignature( &hash, &sign );
```

The above snippet creates a BearSSL public key and a SHA256 hash verifier, and tells the Update object to use them to validate any updates it receives from any method.

Compile the sketch normally and, once a *.bin* file is available, sign it using the signer script:

```
<PicoArduinoPath>/tools/signing.py --mode sign --privatekey <path-to-private.key> --bin
↪<path-to-unsigned-bin> --out <path-to-signed-binary>
```

54.2 Compression

The bootloader incorporates a GZIP decompressor, built for very low code requirements. For applications, this optional decompression is completely transparent.

No changes to the application are required. The *Updater* class and bootloader (which performs actual application overwriting on update) automatically search for the *gzip* header in the uploaded binary, and if found, handle it.

Compress an application *.bin* file or filesystem package using any *gzip* available, at any desired compression level (*gzip -9* is recommended because it provides the maximum compression and uncompresses as fast as any other compression level). For example:

```
gzip -9 sketch.bin  # Maximum compression, output sketch.bin.gz
<Upload the resultant sketch.bin.gz>
```

If signing is desired, sign the gzip compressed file *after* compression.

```
gzip -9 sketch.bin
<PicoPath>/tools/signing.py --mode sign --privatekey <path-to-private.key> --bin sketch.
↪bin.gz --out sketch.bin.gz.signed
```

54.2.1 Safety

The OTA process consumes some of the Pico's resources and bandwidth during upload. Then, the module is restarted and a new sketch executed. Analyse and test how this affects the functionality of the existing and new sketches.

If the Pico is in a remote location and controlling some equipment, you should devote additional attention to what happens if operation of this equipment is suddenly interrupted by the update process. Therefore, decide how to put this equipment into a safe state before starting the update. For instance, your module may be controlling a garden watering system in a sequence. If this sequence is not properly shut down and a water valve is left open, the garden may be flooded.

The following functions are provided with the ArduinoOTA library and intended to handle functionality of your application during specific stages of OTA, or on an OTA error:

```
void onStart(OTA_CALLBACK(fn));
void onEnd(OTA_CALLBACK(fn));
void onProgress(OTA_CALLBACK_PROGRESS(fn));
void onError(OTA_CALLBACK_ERROR (fn));
```

54.3 Uploading from the Arduino IDE

Uploading modules wirelessly from Arduino IDE is intended for the following typical scenarios:

- During firmware development as a quicker alternative to loading over a serial port,
- For updating a small number of modules,
- Only if modules are accessible on the same network as the computer with the Arduino IDE.
- For all IDE uploads, the Pico W and the computer must be connected to the same network.

To upload wirelessly from the IDE:

1. Build a sketch that starts `WiFi` and includes the appropriate calls to `ArduinoOTA` (see the examples for reference). These include the `ArduinoOTA.begin()` call in `setup()` and periodically calling `ArduinoOTA.handle();` from the `loop()`
2. Upload using standard USB connection the first time.
3. The `Tools->Port` should now list `pico-######` under the `Network Ports`. Select it (you won't be able to use the serial monitor, of course).
4. Try another upload. It should display the OTA process in place of the serial port upload.

54.4 Password Protection

Protecting your OTA uploads with a password is really straightforward. All you need to do, is to include the following statement in your code:

```
ArduinoOTA.setPassword((const char *)"123");
```

Where `123` is a sample password that you should replace with your own.

Before implementing it in your sketch, it is a good idea to check how it works using *BasicOTA.ino* sketch available under *File* > *Examples* > *ArduinoOTA*. Go ahead, open *BasicOTA.ino*, uncomment the above statement that is already there, and upload the sketch. To make troubleshooting easier, do not modify example sketch besides what is absolutely required. This includes the original simple `123` OTA password. Then attempt to upload a sketch again (using OTA). After compilation is complete, once upload is about to begin, you should see a prompt for password.

Enter the password and upload should be initiated as usual with the only difference being `Authenticating...OK` message visible in the upload log.

You will not be prompted for a reentering the same password next time. Arduino IDE will remember it for you. You will see a prompt for password only after reopening the IDE, or if you change it in your sketch, upload the sketch and then try to upload it again.

Please note, it is possible to reveal password entered previously in Arduino IDE, if the IDE has not been closed since last upload. This can be done by enabling *Show verbose output during: upload* in *File* > *Preferences* and attempting to upload the module.

54.5 Web Browser

Updates described in this chapter are done with a web browser that can be useful in the following typical scenarios:

- after application deployment if loading directly from Arduino IDE is inconvenient or not possible,
- after deployment if user is unable to expose module for OTA from external update server,
- to provide updates after deployment to small quantity of modules when setting an update server is not practicable.

54.5.1 Requirements

- The Pico and the computer must be connected to the same network, or the IP of the Pico should be known if on a different network.

54.5.2 Implementation Overview

Updates with a web browser are implemented using `HTTPUpdateServer` class together with `WebServer` and `LEAmDNS` or `SimpleMDNS` classes. The following code is required to get it work:

setup()

```
MDNS.begin(host);

httpUpdater.setup(&httpServer);
httpServer.begin();

MDNS.addService("http", "tcp", 80);
```

loop()

```
httpServer.handleClient();
```

In case OTA update fails dead after entering modifications in your sketch, you can always recover module by loading it over a serial port. Then diagnose the issue with sketch using Serial Monitor. Once the issue is fixed try OTA again.

54.6 HTTP Server

`HTTPUpdate` class can check for updates and download a binary file from HTTP web server. It is possible to download updates from every IP or domain address on the network or Internet.

Note that by default this class closes all other connections except the one used by the update, this is because the update method blocks. This means that if there's another application receiving data then TCP packets will build up in the buffer leading to out of memory errors causing the OTA update to fail. There's also a limited number of receive buffers available and all may be used up by other applications.

54.6.1 Requirements

- web server

54.6.2 Arduino code

Simple updater

Simple updater downloads the file every time the function is called.

```
WiFiClient client;
HTTPUpdate.update(client, "192.168.0.2", 80, "/arduino.bin");
```

Advanced updater

It's possible to point the update function to a script on the server. If a version string argument is given, it will be sent to the server. The server side script can use this string to check whether an update should be performed.

The server-side script can respond as follows: - response code 200, and send the firmware image, - or response code 304 to notify Pico that no update is required.

```
WiFiClient client;
t_httpUpdate_return ret = HTTPUpdate.update(client, "192.168.0.2", 80, "/pico/update/
↪arduino.php", "optional current version string here");
switch(ret) {
    case HTTP_UPDATE_FAILED:
        Serial.println("[update] Update failed.");
        break;
    case HTTP_UPDATE_NO_UPDATES:
        Serial.println("[update] Update no Update.");
        break;
    case HTTP_UPDATE_OK:
        Serial.println("[update] Update ok.");
        break;
}
```

TLS updater

Please read and try the examples provided with the library.

54.6.3 Server request handling

Simple updater

For the simple updater the server only needs to deliver the binary file for update.

Advanced updater

For advanced update management a script (such as a PHP script) can run on the server side. It will receive the following headers which it may use to choose a specific firmware file to serve:

::

[User-Agent] => Pico-HTTP-Update [x-Pico-STA-MAC] => 18:FE:AA:AA:AA:AA [x-Pico-AP-MAC] => 1A:FE:AA:AA:AA:AA [x-Pico-Version] => DOOR-7-g14f53a19 [x-Pico-Mode] => sketch

54.7 Stream Interface

The Stream Interface is the base for all other update modes like OTA, HTTP Server / client. Given a Stream-class variable *streamVar* providing *byteCount* bytes of firmware, it can store the firmware as follows:

```
Update.begin(firmwareLengthInBytes);
Update.writeStream(streamVar);
Update.end();
```

54.7.1 OTA Bootloader and Memory Map

A firmware file is uploaded via any method (Ethernet, WiFi, serial ZModem, etc.) and stored on the LittleFS filesystem as a normal file. The Updater class (or the underlying PicoOTA) will make a special "OTA command" file on the filesystem, which will be read by the OTA bootloader. On a reboot, this OTA bootloader will check for an upgrade file, verify its contents, and then perform the requested update and reboot. If no upgrade file is present, the OTA bootloader simply jumps to the main sketch.

The ROM layout consists of:

```
[boot2.S] [OTA Bootloader] [0-pad] [OTA partition table] [Main sketch] [LittleFS␣
↪filesystem] [EEPROM]
```

CHAPTER

FIFTYFIVE

LIBRARIES PORTED/OPTIMIZED FOR THE RP2040

Most Arduino libraries that work on modern 32-bit CPU based Arduino boards will run fine using Arduino-Pico.

The following libraries have undergone additional porting and optimizations specifically for the RP2040 and you should consider using them instead of the generic versions available in the Library Manager

- Adafruit GFX Library by @Bodmer, 2-20x faster than the standard version on the Pico
- Adafruit ILI9341 Library again by @Bodmer
- ESP8266Audio ported to use the included `I2S` library

CHAPTER

FIFTYSIX

USING THE RASPBERRY PI PICO SDK (PICO-SDK)

56.1 Included SDK

A complete copy of the Raspberry Pi Pico SDK is included with the Arduino core, and all functions in the core are available inside the standard link libraries.

For simple programs wishing to call these functions, simply include the appropriate header as shown below

```
#include "pico/stdlib.h"

void setup() {
    const uint LED_PIN = 25;
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);
    while (true) {
        gpio_put(LED_PIN, 1);
        sleep_ms(250);
        gpio_put(LED_PIN, 0);
        sleep_ms(250);
    }
}
void loop() {}
```

Note: When you call SDK functions in your own app, the core and libraries are not aware of any changes to the Pico you perform. So, you may break the functionality of certain libraries in doing so.

56.2 Multicore (CORE1) Processing

Warning: While you may spawn multicore applications on CORE1 using the SDK, the Arduino core may have issues running properly with them. In particular, anything involving flash writes (i.e. EEPROM, filesystems) will probably crash due to CORE1 attempting to read from flash while CORE0 is writing to it.

56.3 PIOASM (Compiling for the PIO processors)

A precompiled version of the PIOASM tool is included in the download package and can be run from the CLI.

There is also a fully online version of PIOASM that runs in a web browser without any CLI required, thanks to @jake653: https://wokwi.com/tools/pioasm (GitHub source: https://github.com/wokwi/pioasm-wasm)

There is also Docker code available for the tool at: https://github.com/kahara/pioasm-docker

CHAPTER

FIFTYSEVEN

LICENSING AND CREDITS

Arduino-Pico is licensed under the LGPL license as detailed in the included README.

In addition, it contains code from additional open source projects:

- The Arduino IDE and ArduinoCore-API are developed and maintained by the Arduino team. The IDE is licensed under GPL.
- The RP2040 GCC-based toolchain is licensed under under the GPL.
- The Pico-SDK and Pico-Extras are by Raspberry Pi (Trading) Ltd. and licensed under the BSD 3-Clause license.
- Arduino-Pico core files are licenses under the LGPL.
- LittleFS library written by ARM Limited and released under the BSD 3-clause license .
- UF2CONV.PY is by Microsoft Corporation and licensed under the MIT license.
- Some filesystem code taken from the ESP8266 Arduino Core and licensed under the LGPL.