Arduino-Pico Documentation

Release 2.6.5

Earle F. Philhower, III

Dec 28, 2022

CONTENTS:

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

This Arduino core uses a custom toolset with GCC 10.3 and Newlib 4.0.0 and doesn't require any system-installed prerequisites.

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

CHAPTER

ONE

GETTING HELP AND CONTRIBUTING

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

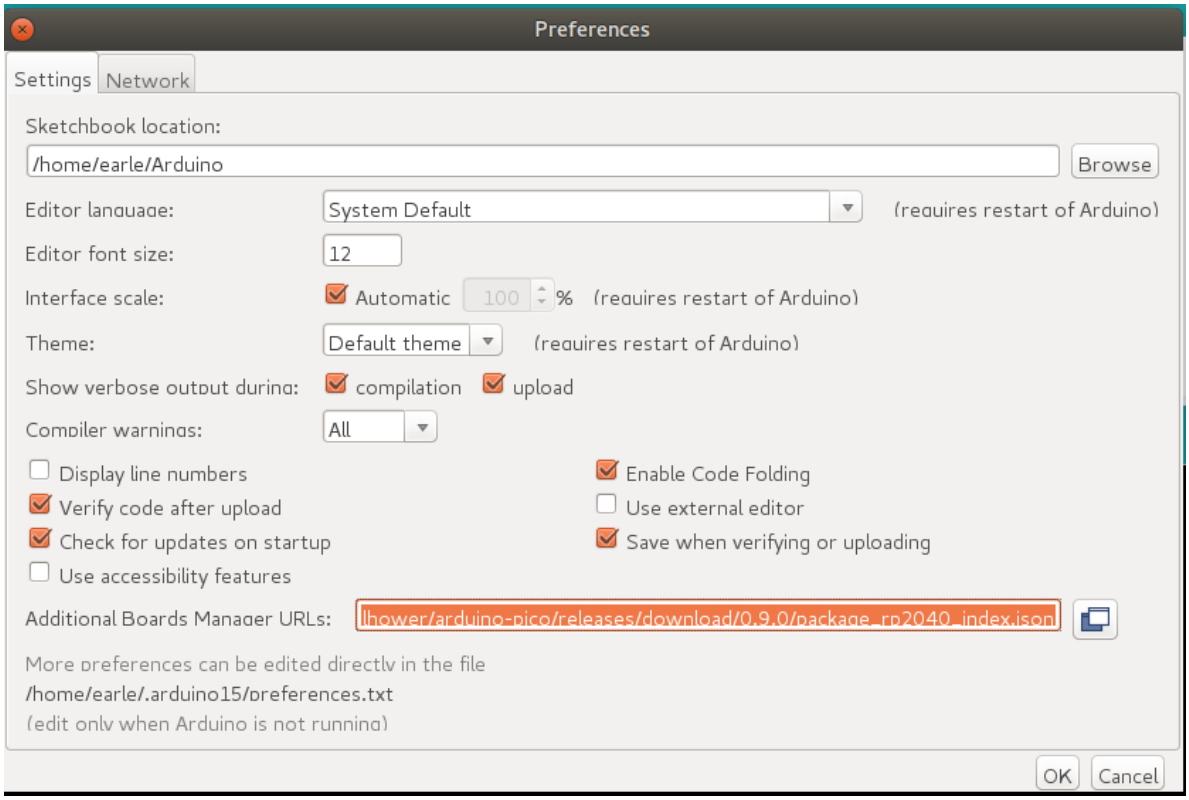
INSTALLATION

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.

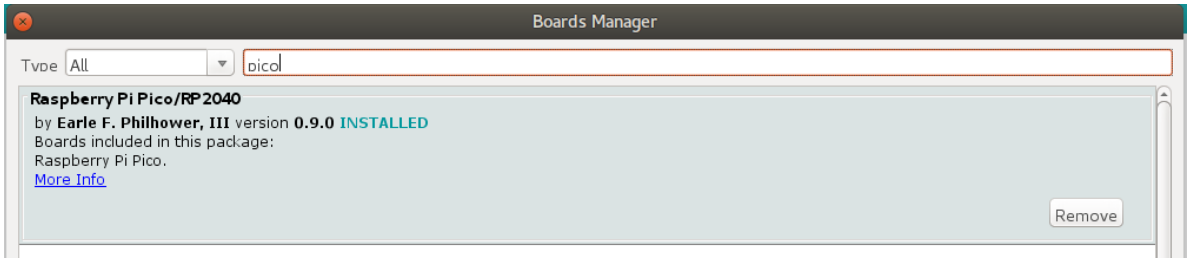
2.1 Installing via Arduino Boards Manager

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. Also, if trying out the 2.0 beta Arduino please install the release 1.8 version beforehand to ensure needed device drivers are present.

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”:



2.2 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
```

2.3 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.

2.4 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.

2.5 Uploading the First Sketch

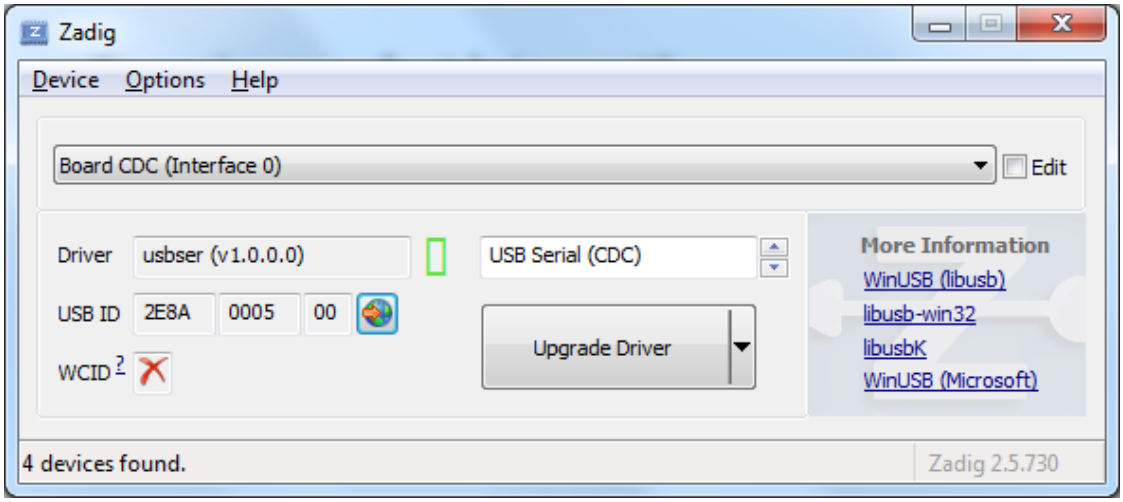
The first time you upload a sketch to a board, you'll need to use the built-in ROM bootloader to handle the upload and not a serial port.

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.

2.6 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.



2.7 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.

2.8 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

- https://github.com/earlephilhower/arduino-pico-littlefs-plugin/releases

To install, follow the directions in

- https://github.com/earlephilhower/arduino-pico-littlefs-plugin/blob/master/README.md

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

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

2.9 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.

2.10 Uploading Sketches with pico-debug

pico-debug differs from Picoprobe in that pico-debug is a virtual debug pod that runs side-by-side on the same RP2040 that you run your code on; so, you only need one RP2040 board instead of two. pico-debug also differs from Picoprobe in that pico-debug is standards-based; it uses the CMSIS-DAP protocol, which means even software not specially written for the Raspberry Pi Pico can support it. pico-debug uses OpenOCD to handle your sketch uploads, and debugging can be accomplished with CMSIS-DAP capable debuggers including GDB.

Under Windows and macOS, any user should be able to access pico-debug 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 board "Raspberry Pi Pico (pico-debug)" in the Tools menu.

When first connecting the USB port to your PC, you must copy pico-debug-gimmecache.uf2 to the Pi Pico to load pico-debug into RAM; after this, upload as normal.

2.11 Debugging with Picoprobe/pico-debug, 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. For pico-debug, replace the raspberrypi-swd and picoprobe example OpenOCD arguments of "-f interface/raspberrypi-swd.cfg -f target/rp2040.cfg" or "-f interface/picoprobe.cfg -f target/rp2040.cfg" respectively in the Pico Getting Started manual with "-f board/pico-debug.cfg".

CHAPTER

THREE

IDE MENUS

3.1 Model

Use the boards menu to select your model of RP2040 board. There will be two options: *Boardname* and *Boardname (Picoprobe)*. If you want to use a Picoprobe to upload your sketches and not the default automatic UF2 upload, use the *(Picoprobe)* option, otherwise use the normal name. No functional or code changes are done because of this.

There is also a *Generic* 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.

3.2 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.

3.3 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.**

3.4 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.

3.5 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.

3.6 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

FOUR

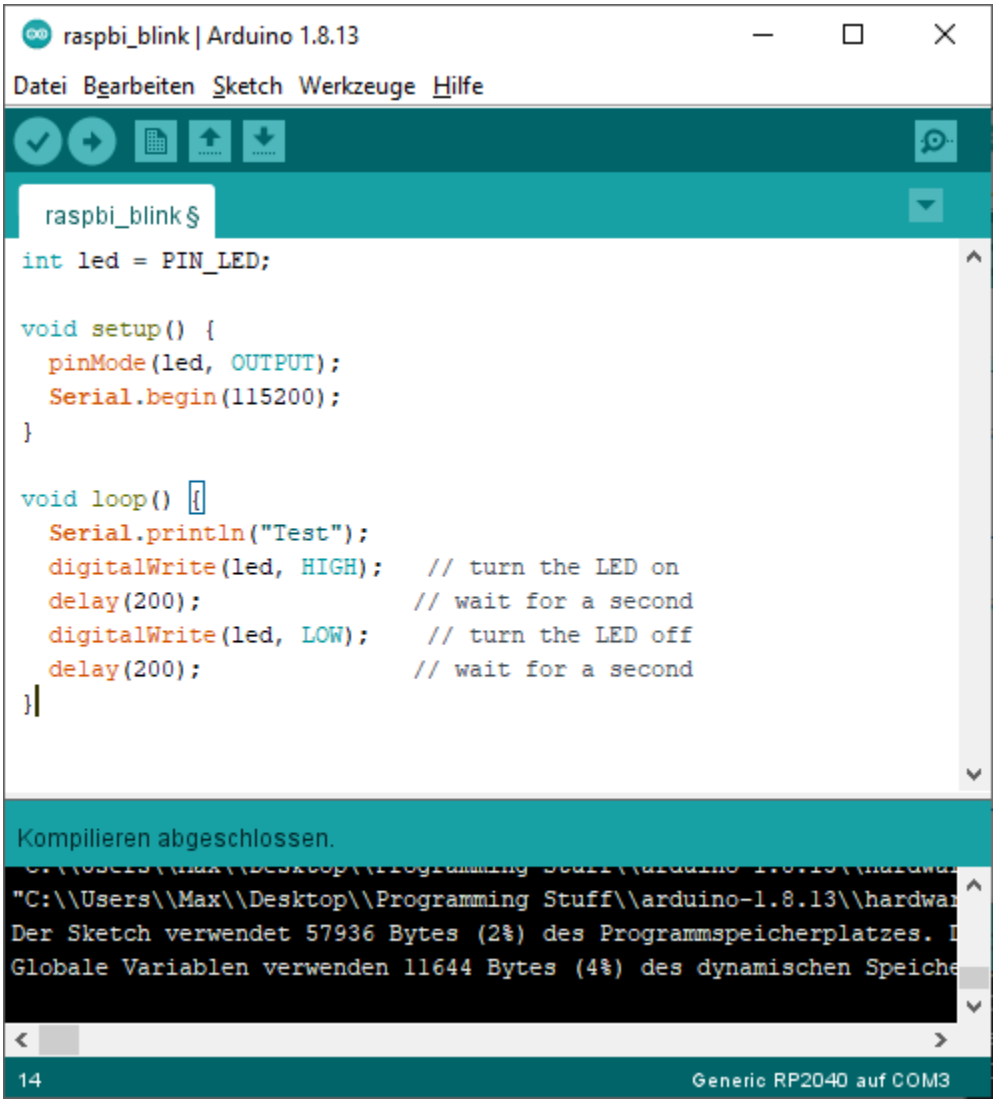
USING THIS CORE WITH PLATFORMIO

4.1 What is PlatformIO?

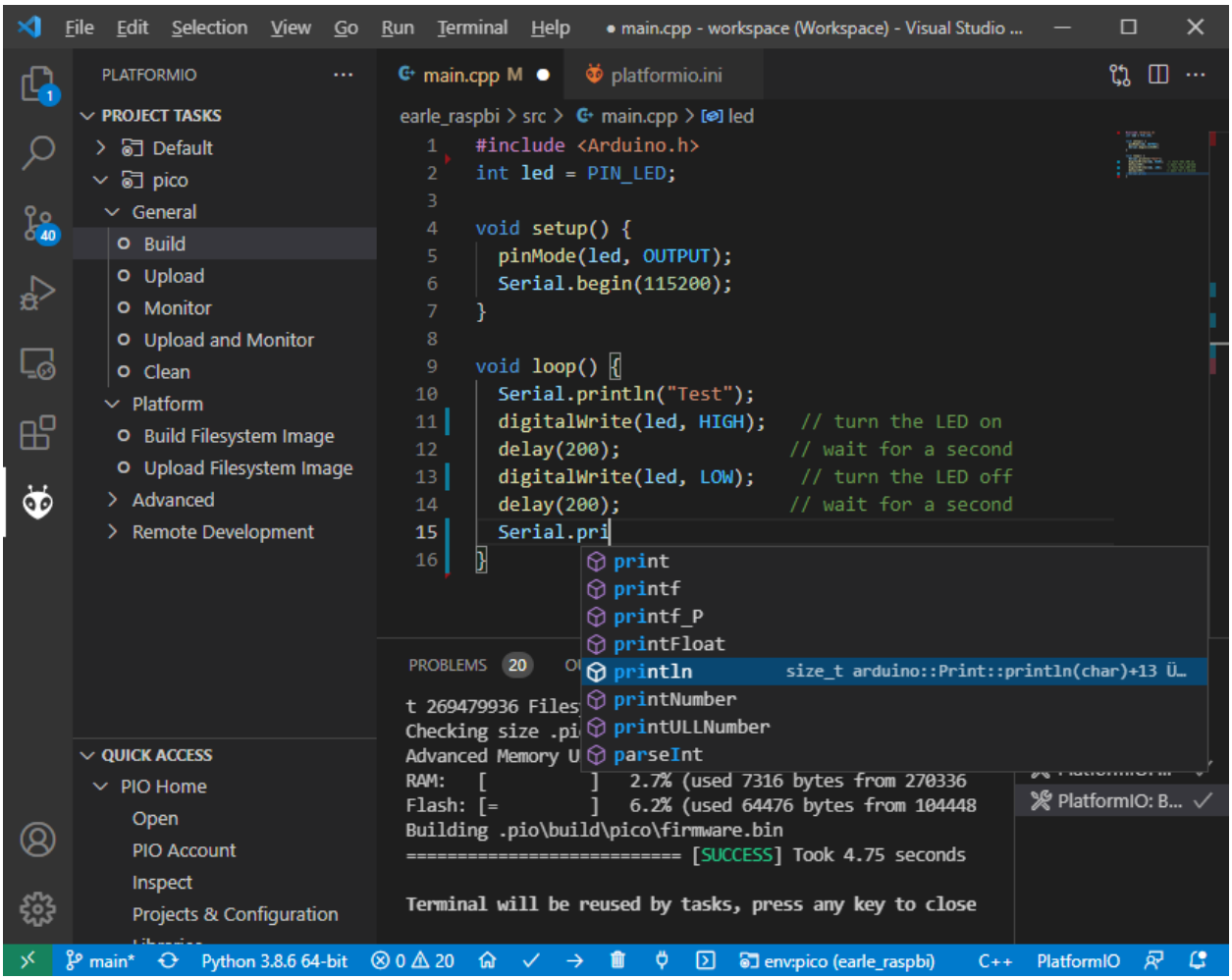
PlatformIO is a free, open-source build-tool written in Python, which also integrates into VSCode code as an extension.

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.

4.2 Current state of development

At the time of writing, PlatformIO integration for this core is a work-in-progress and not yet merged into mainline PlatformIO. This is subject to change once this pull request is merged.

If you want to use the PlatformIO integration right now, make sure you 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.

4.3 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
    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.

4.4 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.

4.5 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
```

4.6 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
```

4.7 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
```

4.8 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 sing 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
```

(continues on next page)

(continued from previous page)

```
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
```

4.9 C++ Exceptions

Exceptions are disabled by default. To enable them, use

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

4.10 Stack Protector

To enable GCC's stack protection feature, use

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

4.11 RTTI

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

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

4.12 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`.

4.13 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`.

4.14 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.

4.15 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.

4.16 Debugging

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

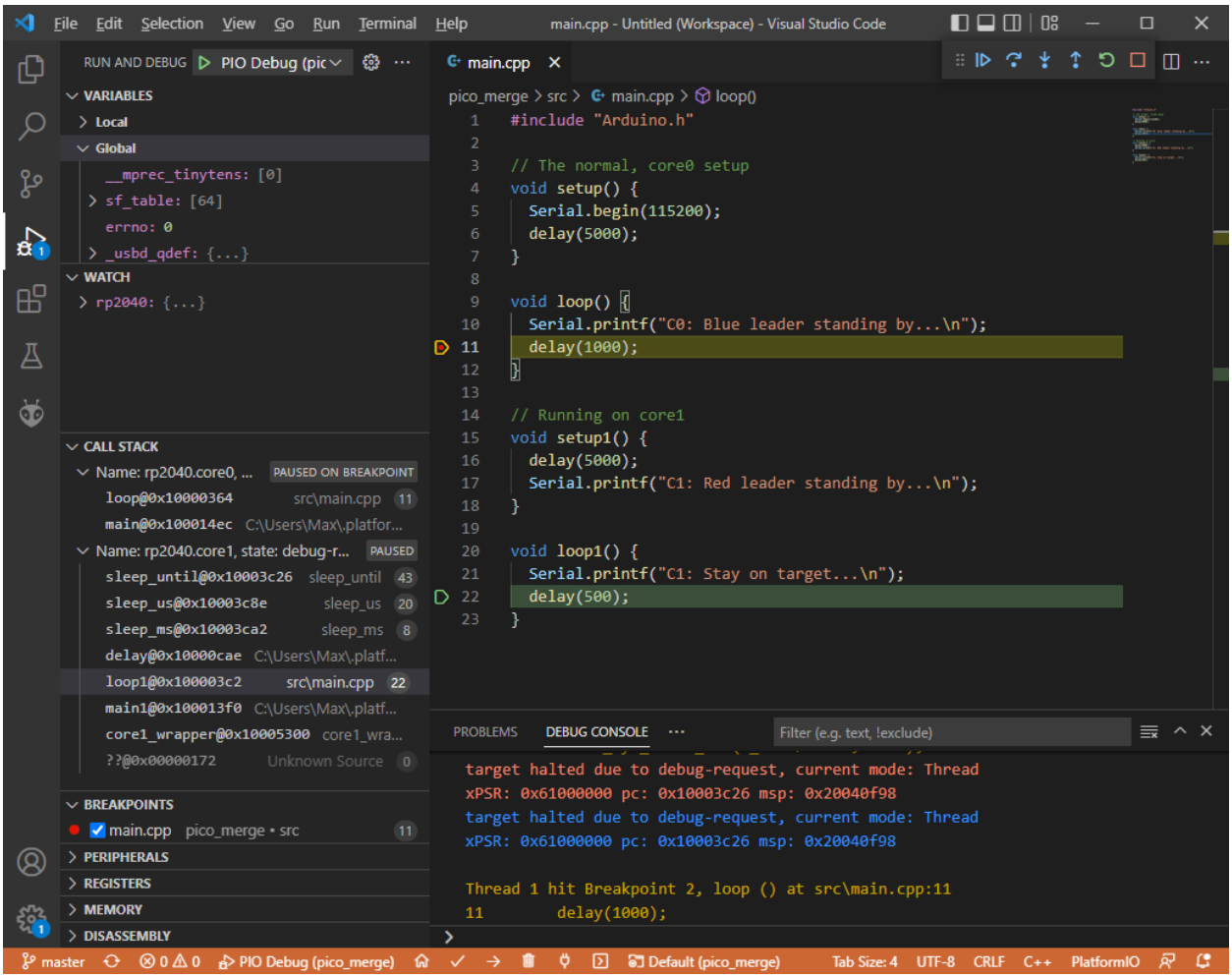
To specify the debugging adapter, use `debug_tool` (documentation). Supported values are:

- `picoprobe`
- `cmsis-dap`
- `jlink`
- `raspberrypi-swd`

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.

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.

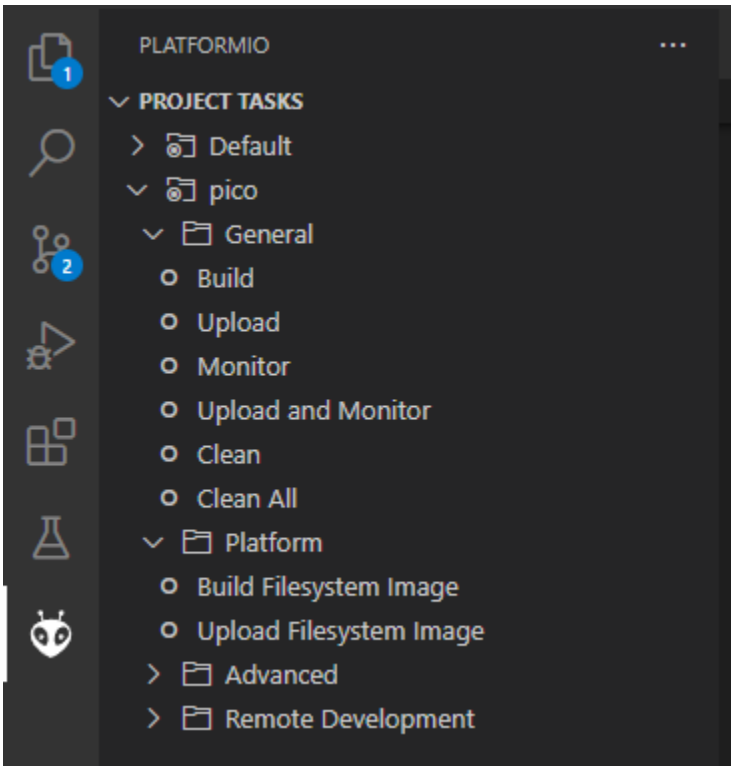


For further information on customizing debug options, like the initial breakpoint or debugging / SWD speed, consult the documentation.

4.17 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`.

CHAPTER

FIVE

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.

5.1 I2S

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

5.2 Serial1 (UART0), Serial2 (UART1)

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

5.3 SPI (SPI0), SPI1 (SPI1)

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

5.4 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);
    SPI.setTX(7);
    SPI.setSCK(6);
    SPI.setCS(5);
    SD.begin(5);
}
```

CHAPTER

SIX

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`.

6.1 Core Internals

6.1.1 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.)

6.1.2 uint32_t rp2040.getCycleCount()

Returns a 32-bit cycle count from then 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.

6.1.3 uint64_t rp2040.getCycleCount64()

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

6.1.4 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.**

6.1.5 void rp2040.reboot()

Forces a hardware reboot of the Pico.

6.2 Hardware Watchdog

6.2.1 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.

6.2.2 void rp2040.wdt_reset()

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

6.3 Memory Information

6.3.1 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.

6.3.2 int rp2040.getUsedHeap()

Returns the number of bytes allocated out of the heap.

6.3.3 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).

6.4 Bootloader

6.4.1 void rp2040.enableDoubleResetBootloader()

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.

CHAPTER

SEVEN

ANALOG I/O

7.1 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 4095).

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

Returns a value from 0…4095 correspionding to the ADC reading of the specific pin.

7.1.2 void analogReadResolution(int bits)

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

7.1.3 float analogReadTemp()

Returns the temperature, in Celsius, of the onboard thermal sensor. 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.

7.2 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 RP20400'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.

7.3 Analog Output Restrictions

The PWM generator source clock restricts the legal combinations of frequency and ranges. 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.

7.3.1 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.

7.3.2 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.

7.3.3 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

EIGHT

DIGITAL I/O

8.1 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.

8.2 Input Modes

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

8.3 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*.

8.4 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

NINE

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.

9.1 EEPROM Class API

9.1.1 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.

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

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

9.1.3 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.

9.1.4 EEPROM.commit()

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

9.1.5 EEPROM.end()

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

9.1.6 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.

9.1.7 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.

9.1.8 EEPROM.length()

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

9.2 EEPROM Examples

Three EEPROM examples are included.

CHAPTER

TEN

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.

10.1 I2S Class API

10.1.1 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.

10.1.2 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.

10.1.3 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()`

10.1.4 bool setDATA(pin_size_t pin)

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

10.1.5 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()`

10.1.6 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()`.

10.1.7 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.

10.1.8 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.

10.1.9 void end()

Stops the I2S device.

10.1.10 void flush()

Waits until all the I2S buffers have been output.

10.1.11 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.

10.1.12 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.

10.1.13 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 bytes were actually written.

10.1.14 int availableForWrite()

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

10.1.15 int read()

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

10.1.16 int peek()

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

10.1.17 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.

10.1.18 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.

10.2 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).

10.2.1 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.

10.2.2 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.

10.2.3 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.

10.2.4 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.

10.2.5 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.

10.2.6 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.

10.2.7 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.

10.2.8 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.

10.3 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

ELEVEN

SERIAL PORTS (USB AND UART)

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);
```

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 .

CHAPTER

TWELVE

"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 `SerialPIO::NOPIN` as the `txpin` or `rxpin`.

For example, to make a transmit-only port on GP16 .. code:: cpp

    SerialPIO transmitter(16, SerialPIO::NOPIN);

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

CHAPTER

THIRTEEN

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:

- Inverted mode is not supported
- All ports are always listening
- `listen` call is a no-op
- `isListening()` always returns `true`

CHAPTER

FOURTEEN

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.

CHAPTER

FIFTEEN

SPI (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);
bool setCS(pin_size_t pin);
bool setSCK(pin_size_t pin);
bool setTX(pin_size_t pin);
```

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

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 (`usingInterrupt`, `notUsingInterrupt`, `attachInterrupt`, and `detachInterrpt`) are not implemented.

CHAPTER

SIXTEEN

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 128 bytes) and only performed on `endTransmission`, as is standard with modern Arduino Wire implementations.

For more detailed information, check the Arduino Wire documentation .

CHAPTER

SEVENTEEN

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.

17.1 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:

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

The file system size is configurable via the IDE menus, rom 64k up to 15MB (assuming you have an RP2040 boad 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>
```

17.2 Compatible Filesystem APIs

LittleFS is an onboard filesystem that sets asidesome 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.

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

17.3 LittleFS File System Limitations

The LittleFS implementation for the RP2040 supports filenames of up to 31 characters + terminating zero (i.e. `char filename[32]`), 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.

17.4 Uploading Files to the LittleFS File System

PicoLittleFS is a tool which integrates into the Arduino IDE. It adds a menu item to **Tools** menu for uploading the contents of sketch data directory into a new LittleFS flash file system.

- 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 theSerial 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.

17.5 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.

17.6 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);
```

17.7 File system object (LittleFS/SD/SDFS)

17.7.1 setConfig

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

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

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, SPIFFS will autoformat the filesystem if it cannot mount it, while SDFS will not.

17.7.2 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. With no options it will format SPIFFS if it is unable to mount it on the first try.

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

17.7.3 end

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

This method unmounts the file system.

17.7.4 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.

17.7.5 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+". Meaning of these modes is the same as for `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) {
    Serial.println("file open failed");
}
```

17.7.6 exists

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

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

17.7.7 mkdir

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

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

17.7.8 rmdir

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

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

17.7.9 openDir

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

Opens a directory given its absolute path. Returns a *Dir* object. Please note the previous discussion on the difference in behavior between LittleFS and SPIFFS for this call.

17.7.10 remove

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

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

17.7.11 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.

17.7.12 info DEPRECATED

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

Fills *FSInfo structure* with information about the file system. Returns `true` if successful, `false` otherwise. Because this cannot report information about filesystemd greater than 4MB, don't use it in new code. Use `info64` instead which uses 64-bit fields for filesystem sizes.

17.8 Filesystem information structure

```
struct FSInfo {
    size_t totalBytes;
    size_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)

17.8.1 info64

```
FSInfo64 fsinfo;
SDFS.info(fsinfo);
or LittleFS(fsinfo);
```

Performs the same operation as `info` but allows for reporting greater than 4GB for filesystem size/used/etc. Should be used with the SD and SDFS filesystems since most SD cards today are greater than 4GB in size.

17.8.2 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.

17.9 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());
    }
}
```

17.9.1 next

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

17.9.2 fileName

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

17.9.3 fileSize

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

17.9.4 fileTime

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

17.9.5 fileCreationTime

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

17.9.6 isFile

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

17.9.7 isDirectory

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

17.9.8 openFile

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

17.9.9 rewind

Resets the internal pointer to the start of the directory.

17.9.10 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.

17.10 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.

17.10.1 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.

17.10.2 position

```
file.position()
```

Returns the current position inside the file, in bytes.

17.10.3 size

```
file.size()
```

Returns file size, in bytes.

17.10.4 name

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

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

17.10.5 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*`.

17.10.6 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.

17.10.7 getCreationTime

Returns the file creation time, if available.

17.10.8 isFile

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

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

17.10.9 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).

17.10.10 close

```
file.close()
```

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

17.10.11 openNextFile (compatibiity 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`.

17.10.12 rewindDirectory (compatibiity 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`.

17.10.13 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

EIGHTEEN

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.

18.1 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.

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.

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

18.2 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.

18.3 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

NINETEEN

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.

19.1 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).

19.1.1 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).

19.1.2 void rp2040.resumeOtherCore()

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

19.1.3 void rp2040.restartCore1()

Hard resets Core1 from Core 0 and restarts its operation from `setup1()`.

19.2 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.

19.2.1 void rp2040.fifo.push(uint32_t)

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

19.2.2 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`.

19.2.3 uint32_t rp2040.fifo.pop()

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

19.2.4 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.

19.2.5 int rp2040.fifo.available()

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

CHAPTER

TWENTY

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.

20.1 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.

20.2 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

TWENTYONE

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.

21.1 Enabling FreeRTOS

To enable FreeRTOS, simply add

```
#include <FreeRTOS.h>
```

to your sketch and it will be included and enabled automatically.

21.2 Configuration and Predefined Tasks

FreeRTOS is configured with 8 priority levels (0 through 7) and a process for `setup()/loop()`, `setup1()/loop1()`, 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.

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.

21.3 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.

21.4 More Information

For full FreeRTOS documentation look at FreeRTOS.org and FreeRTOS SMP support.

CHAPTER

TWENTYTWO

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.

22.1 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.

22.2 Important Information

Please note that WiFi on the Pico W is a work-in-progress and there are some important caveats:

- 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 is a little limited as of now, but fully functional for sending and receiving data
  - Extensible Authentication Protocol (EAP) is not supported
  - Combined STA/AP mode is not supported
- Multicore is supported, but only one core may run `WiFi` code.
  - FreeRTOS is not yet supported due to the requirement for a very different LWIP implementation. PRs always appreciated!

The WiFi library borrows much work from the ESP8266 Arduino Core , especially the `WiFiClient` and `WiFiServer` classes.

22.3 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

TWENTYTHREE

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.

23.1 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()`.

23.2 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);
```

23.3 getNoDelay

Returns whether NoDelay is enabled or not for the current connection.

23.4 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,

23.5 getSync

Returns whether Sync is enabled or not for the current connection.

23.6 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*.

23.7 getDefaultNoDelay and getDefaultSync

Return the values to be used as default for NoDelay and Sync for all future connections.

23.8 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

TWENTYFOUR

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.

24.1 accept

Method `accept()` returns a waiting client connection. accept() is documented for the Arduino Ethernet library.

24.2 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`.

24.3 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.

24.4 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`).

24.5 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

TWENTYFIVE

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

TWENTYSIX

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));
}
```

26.1 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);
```

(continues on next page)

(continued from previous page)

```
  struct tm timeinfo;
  gmtime_r(&now, &timeinfo);
  Serial.print("Current time: ");
  Serial.print(asctime(&timeinfo));
}
```

26.2 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 ".":

```
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

TWENTYSEVEN

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.

27.1 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.

27.2 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.

27.3 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() {
```

(continues on next page)

(continued from previous page)

```
    BearSSL::X509List x509(x509CA);
    client.setTrustAnchor(&x509);
}
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.

27.4 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.

27.5 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.

27.6 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.

27.7 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.

27.8 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.

27.9 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

TWENTYEIGHT

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.

28.1 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:

28.1.1 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.

28.1.2 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.

28.1.3 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.

28.1.4 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).

28.1.5 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.

28.2 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.

28.2.1 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.

28.3 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()* .

28.3.1 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.

28.3.2 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.

28.3.3 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.

28.4 Sessions (Resuming connections fast)

28.4.1 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.

28.5 Errors

BearSSL can fail in many more unique and interesting ways. Use these calls to get more information when something fails.

28.5.1 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();*).

28.6 Limiting Ciphers (New connections faster)

There is very rarely reason to use these calls, but they are available.

28.6.1 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.

28.6.2 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.

28.7 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:

28.7.1 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.

CHAPTER

TWENTYNINE

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

THIRTY

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.

30.1 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()*

30.2 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 re-use 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.**

30.2.1 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()*

30.2.2 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()*.

30.3 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.

30.3.1 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.

30.4 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.

30.4.1 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

THIRTYONE

HTTPCLIENT LIBRARY

A simple HTTP requestor that can handle both HTTP and HTTP 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

THIRTYTWO

OTA UPDATES

32.1 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 (ZModen 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.

32.1.1 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.

32.1.2 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.

32.1.3 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.

32.1.4 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.

32.1.5 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 sigining 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>
```

32.2 Compression

The eboot bootloader incorporates a GZIP decompressor, built for very low code requirements. For applications, this optional decompression is completely transparent. For uploading compressed filesystems, the application must be built with *ATOMIC_FS_UPDATE* defined because, otherwise, eboot will not be involved in writing the filesystem.

No changes to the application are required. The *Updater* class and *eboot* 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 compressino 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
```

32.2.1 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));
```

32.3 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,m the Pico W and the computer must be connected to the same network.

To upload wirelessly from the IDE:

1. Build a sketch starts `WiFi` and includes the appropriare 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.

32.4 Password Protection

Protecting your OTA uploads with 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 is including original simple `123` OTA password. Then attempt to upload sketch again (using OTA). After compilation is complete, once upload is about to begin, you should see prompt for password.

Enter the password and upload should be initiated as usual with the only difference being `Authenticating...OK` message visible in 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 prompt for password only after reopening 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 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.

32.5 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.

32.5.1 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.

32.5.2 Implementation Overview

Updates with a web browser are implemented using `HTTPUpdateServer` class together with `WebServer` and `LEAmDNS` 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.

32.6 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.

There are some cases where you know that you won't be receiving any data but would still like to send progress updates. It's possible to disable the default behaviour (and keep connections open) by calling closeConnectionsOnUpdate(false).

32.6.1 Requirements

- web server

32.6.2 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

Its 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."); // may not be called since we reboot the␣
↪RP2040
        break;
}
```

TLS updater

Please read and try the examples provided with the library.

32.6.3 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

32.7 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();
```

32.7.1 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

THIRTYTHREE

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

THIRTYFOUR

USING THE RASPBERRY PI PICO SDK (PICO-SDK)

34.1 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.

34.2 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.

34.3 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

THIRTYFIVE

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 Corporatio and licensed under the MIT license.
- Some filesystem code taken from the ESP8266 Arduino Core and licensed under the LGPL.