Arduino-Pico Documentation

Release 1.0.0

Earle F. Philhower, III

Jul 08, 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 10.2 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

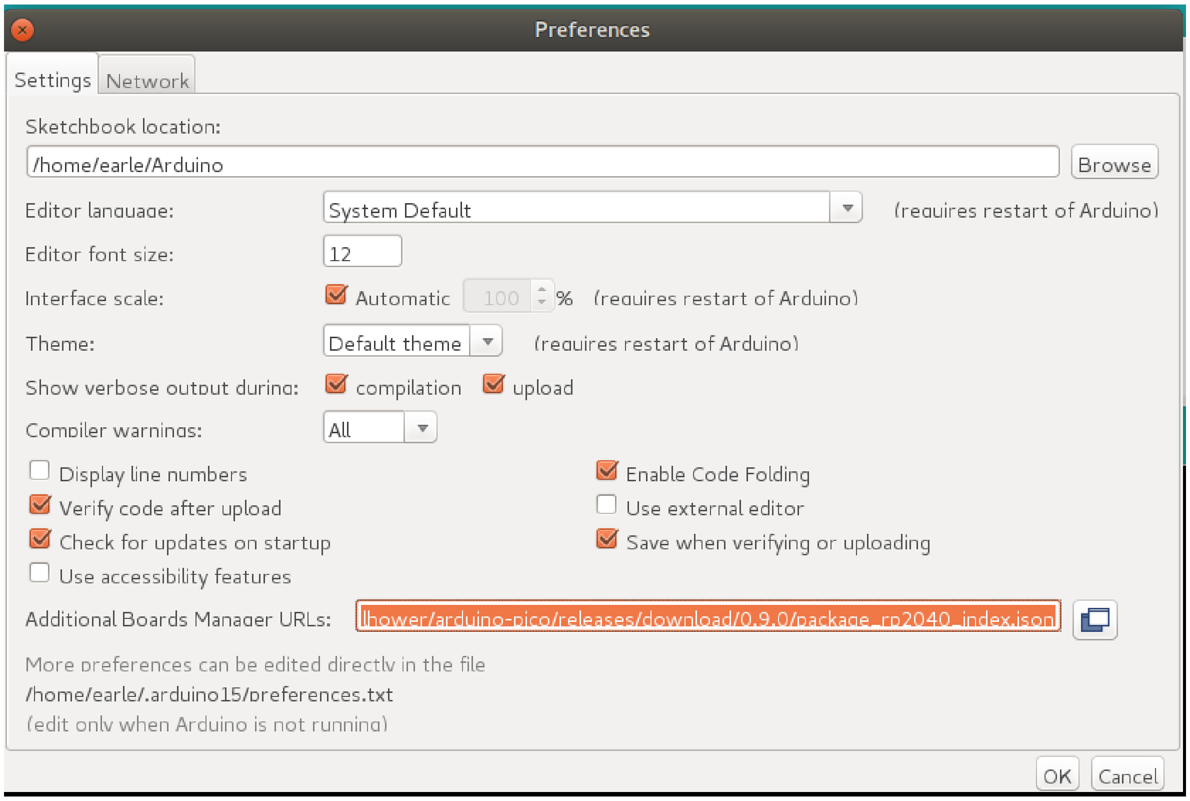
INSTALLATION

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* instllation.

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



Sketchbook location: /home/earle/Arduino

Editor language: System Default (requires restart of Arduino)

Editor font size: 12

Interface scale: ✓ Automatic 100 % (requires restart of Arduino)

Theme: Default theme (requires restart of Arduino)

Show verbose output during: ✓ compilation ✓ upload

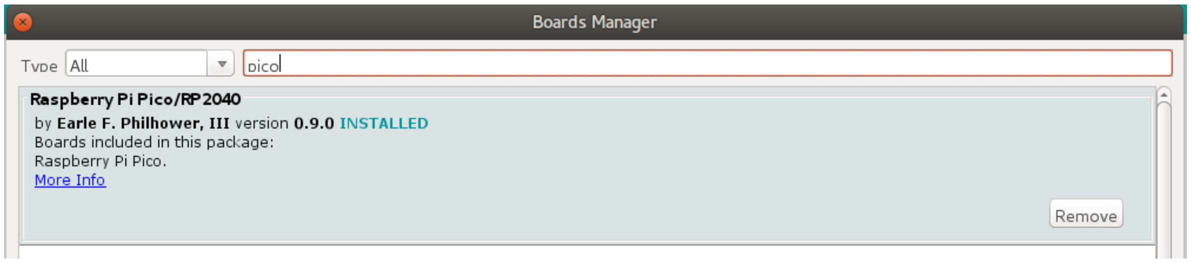
Compiler warnings: All

☐ Display line numbers
✓ Verify code after upload
✓ Check for updates on startup
☐ Use accessibility features
✓ Enable Code Folding
☐ Use external editor
✓ Save when verifying or uploading

Additional Boards Manager URLs: lhower/arduino-pico/releases/download/0.9.0/package_rp2040_index.json

More preferences can be edited directly in the file
/home/earle/.arduino15/preferences.txt
(edit only when Arduino is not running)

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":



Type All pico

Raspberry Pi Pico/RP2040
by **Earle F. Philhower, III** version **0.9.0** INSTALLED
Boards included in this package:
Raspberry Pi Pico.
More Info

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.

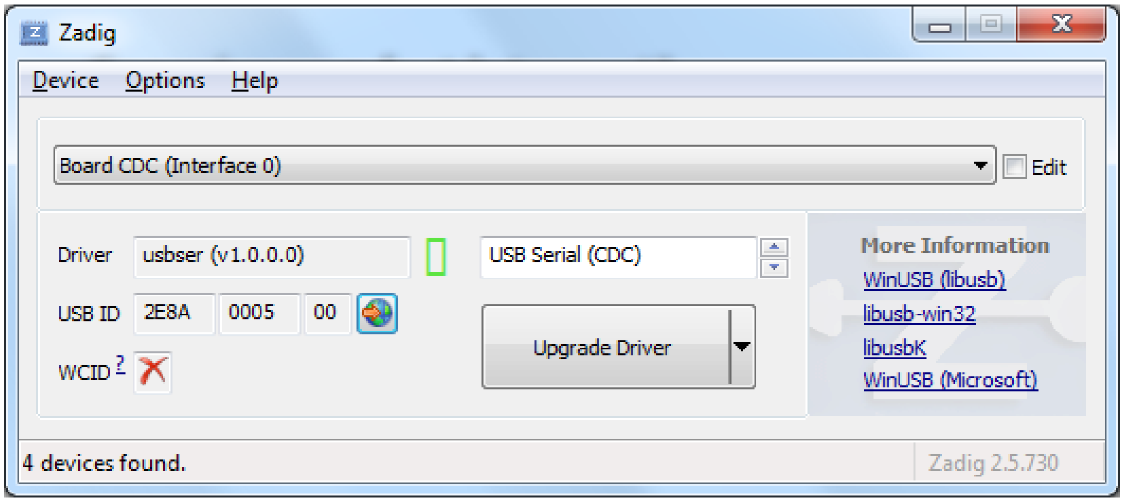
After the first upload, this should not be necessary as the *arduino-pico* core has auto-reset support. Select the appropriate serial port shown in the Arduino Tools->Port->Serial Port menu once (this setting will stick and does not need to be touched for multiple uploads). This selection allows the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



Board CDC (Interface 0)

Driver usbser (v1.0.0.0) USB Serial (CDC)

USB ID 2E8A 0005 00

4 devices found.

Zadig 2.5.730

2.6 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.

2.7 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

2.8 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.

2.9 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.

2.10 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 bu 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

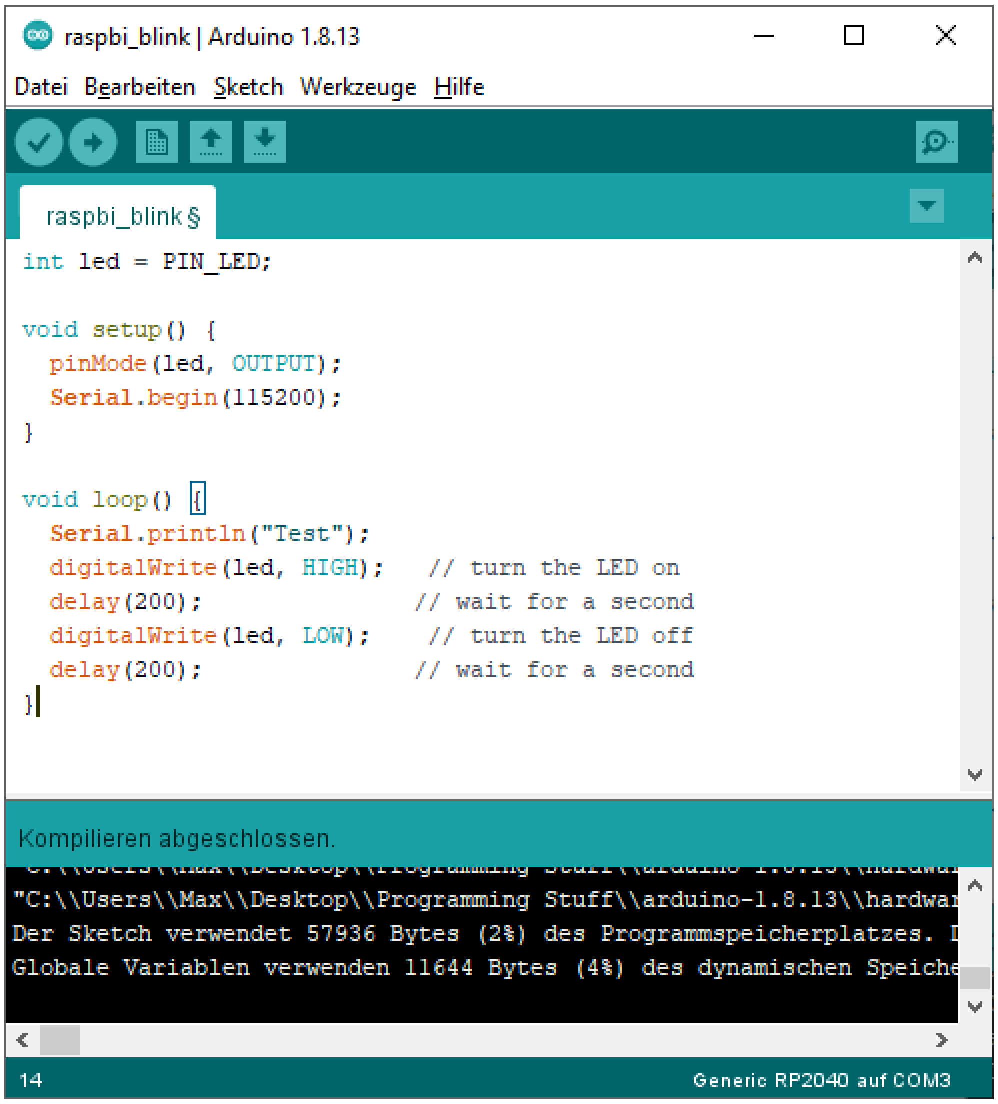
USING THIS CORE WITH PLATFORMIO

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:

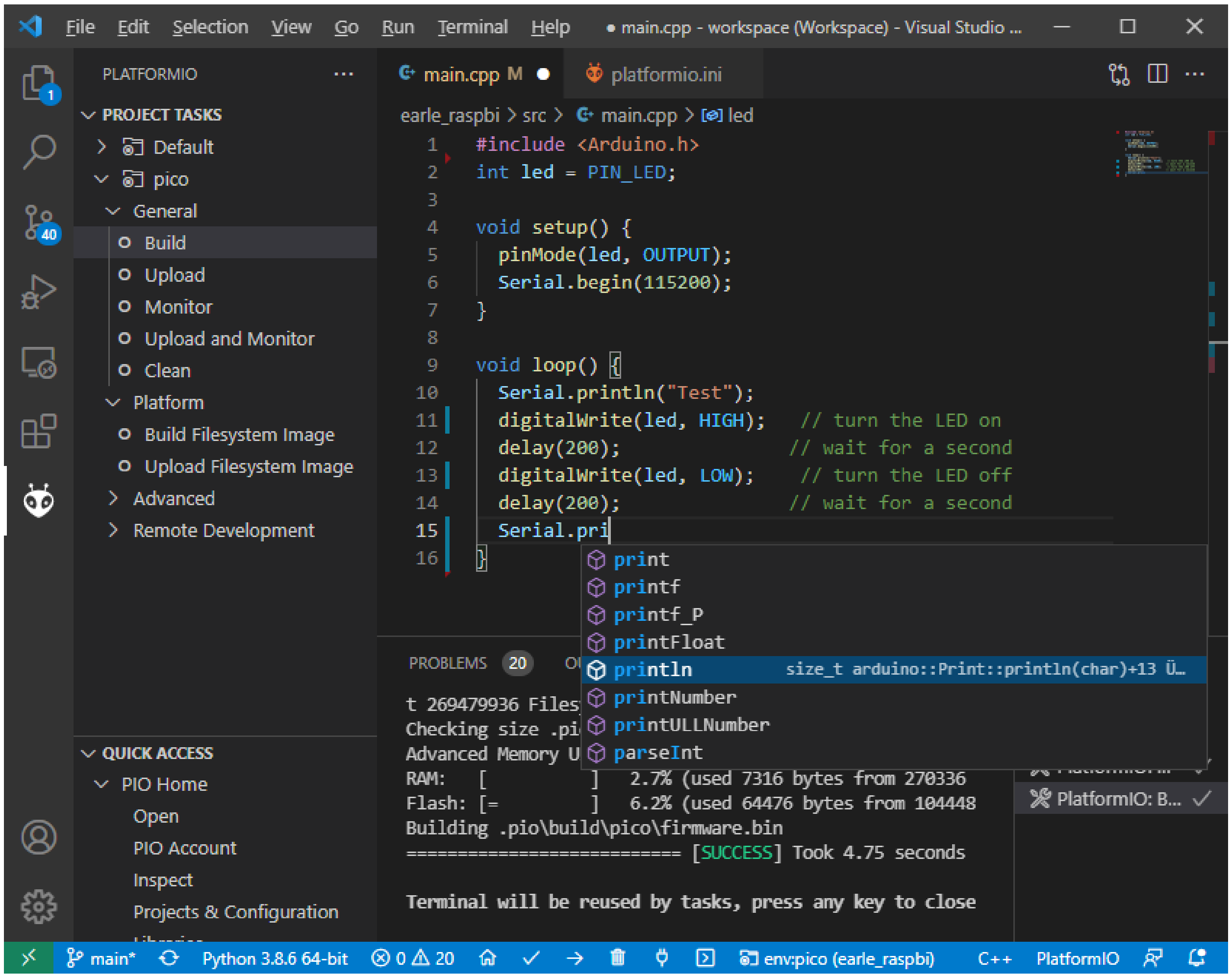

raspbi_blink | Arduino 1.8.13
Datei Bearbeiten Sketch Werkzeuge Hilfe
raspbi_blink §

```
int led = PIN_LED;

void setup() {
  pinMode(led, OUTPUT);
  Serial.begin(115200);
}

void loop() {
  Serial.println("Test");
  digitalWrite(led, HIGH);   // turn the LED on
  delay(200);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off
  delay(200);               // wait for a second
}
```

Kompilieren abgeschlossen.

```
"C:\\Users\\Max\\Desktop\\Programming Stuff\\arduino-1.8.13\\hardwar
Der Sketch verwendet 57936 Bytes (2%) des Programmspeicherplatzes. I
Globale Variablen verwenden 11644 Bytes (4%) des dynamischen Speiche
```

The PlatformIO experience:



```
#include <Arduino.h>
int led = PIN_LED;

void setup() {
  pinMode(led, OUTPUT);
  Serial.begin(115200);
}

void loop() {
  Serial.println("Test");
  digitalWrite(led, HIGH);   // turn the LED on
  delay(200);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off
  delay(200);               // wait for a second
  Serial.pri
}
```

```
print
printf
printf_P
printFloat
println          size_t arduino::Print::println(char)+13 Ü…
printNumber
printULLNumber
parseInt
```

```
t 269479936 Files
Checking size .pi
Advanced Memory U
RAM:   [          ]  2.7% (used 7316 bytes from 270336
Flash: [=         ]  6.2% (used 64476 bytes from 104448
Building .pio\build\pico\firmware.bin
========================== [SUCCESS] Took 4.75 seconds

Terminal will be reused by tasks, press any key to close
```

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 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.

4.14 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.

4.15 Debugging

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

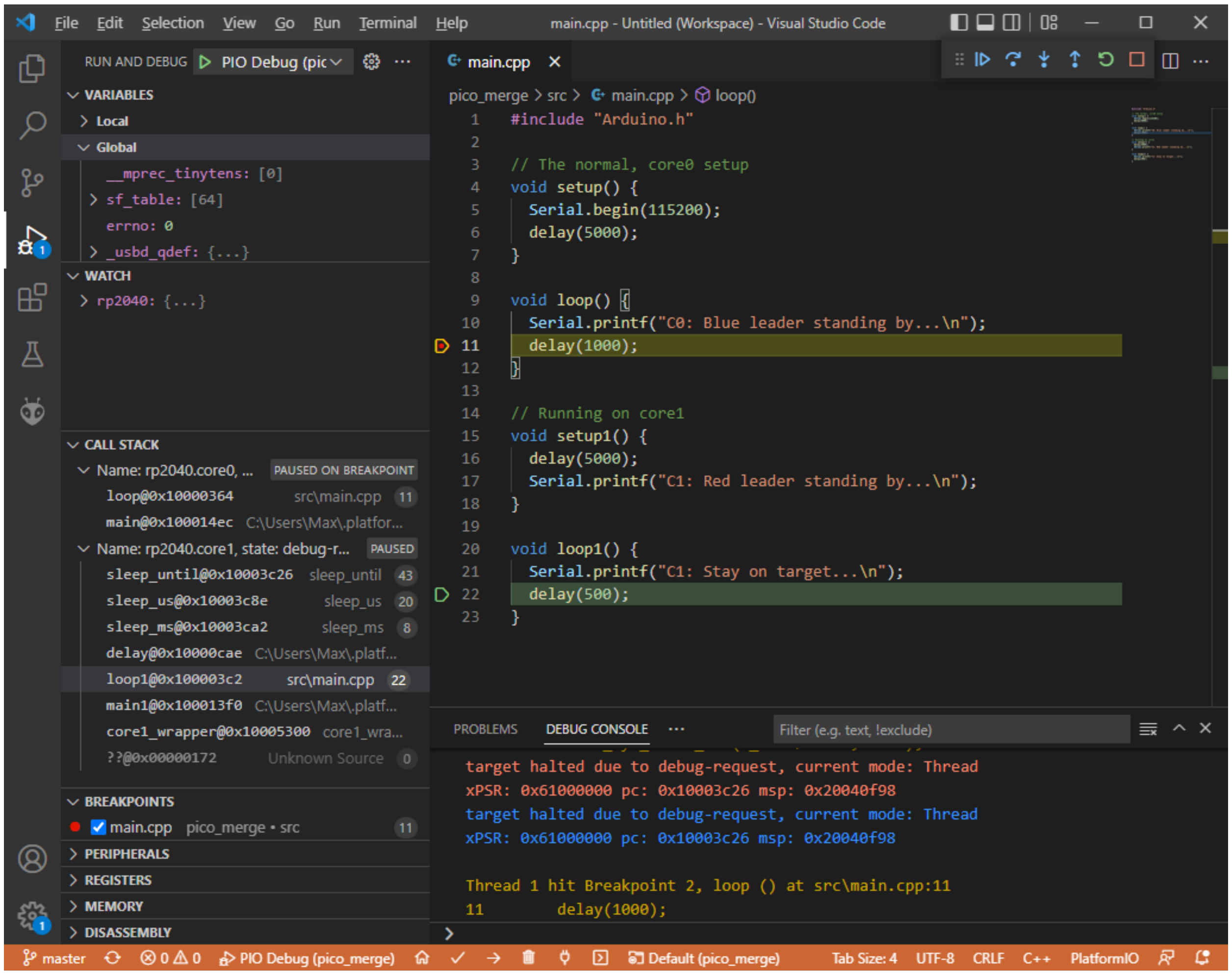
To specify the debugging adapter, use `debug_tool` (documentation). Supported values are:

- `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.



```
__mprec_tinytens: [0]
> sf_table: [64]
errno: 0
> _usbd_qdef: {...}
```

```
> rp2040: {...}
```

```
Name: rp2040.core0, ...          PAUSED ON BREAKPOINT
  loop@0x10000364                src\main.cpp  11
  main@0x100014ec                C:\Users\Max\.platfor...
Name: rp2040.core1, state: debug-r...   PAUSED
  sleep_until@0x10003c26         sleep_until  43
  sleep_us@0x10003c8e            sleep_us  20
  sleep_ms@0x10003ca2            sleep_ms  8
  delay@0x10000cae               C:\Users\Max\.platf...
  loop1@0x100003c2               src\main.cpp  22
  main1@0x100013f0               C:\Users\Max\.platf...
  core1_wrapper@0x10005300       core1_wra...
  ??@0x00000172                  Unknown Source  0
```

```
main.cpp   pico_merge • src   11
```

```cpp
#include "Arduino.h"

// The normal, core0 setup
void setup() {
  Serial.begin(115200);
  delay(5000);
}

void loop() {
  Serial.printf("C0: Blue leader standing by...\n");
  delay(1000);
}

// Running on core1
void setup1() {
  delay(5000);
  Serial.printf("C1: Red leader standing by...\n");
}

void loop1() {
  Serial.printf("C1: Stay on target...\n");
  delay(500);
}
```

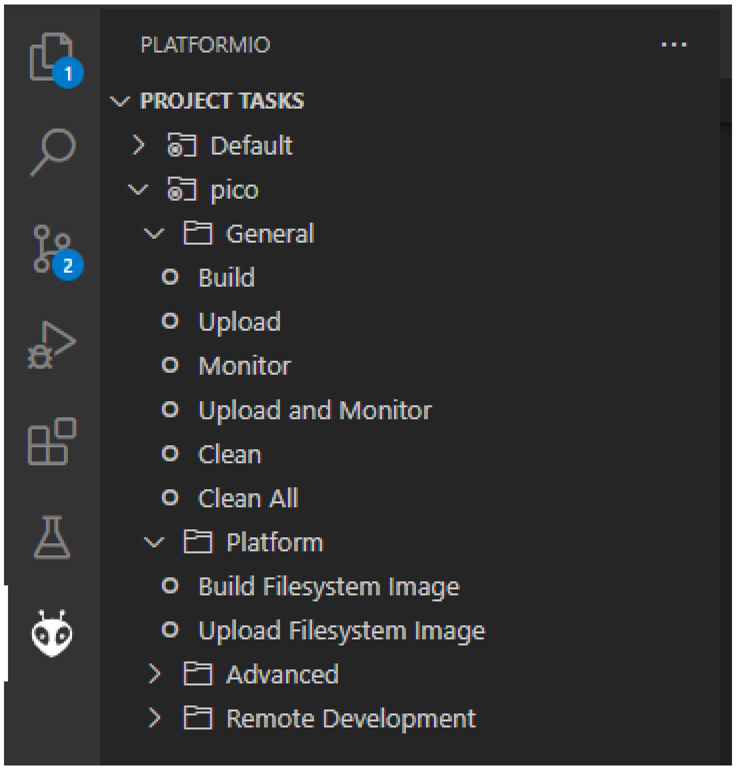
```
target halted due to debug-request, current mode: Thread
xPSR: 0x61000000 pc: 0x10003c26 msp: 0x20040f98
target halted due to debug-request, current mode: Thread
xPSR: 0x61000000 pc: 0x10003c26 msp: 0x20040f98

Thread 1 hit Breakpoint 2, loop () at src\main.cpp:11
11        delay(1000);
```

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

4.16 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

ANALOG I/O

6.1 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).

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

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

6.1.2 void analogReadResolution(int bits)

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

6.1.3 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.

6.2 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.

6.3 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.

6.3.1 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.

6.3.2 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.

6.3.3 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

SEVEN

DIGITAL I/O

7.1 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.

7.2 Input Modes

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

7.3 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*.

7.4 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

EIGHT

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.

8.1 EEPROM Class API

8.1.1 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.

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

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

8.1.3 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.

8.1.4 EEPROM.commit()

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

8.1.5 EEPROM.end()

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

8.1.6 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.

8.1.7 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.

8.1.8 EEPROM.length()

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

8.2 EEPROM Examples

Three EEPROM **`examples<https://github.com/earlephilhower/arduino-pico/tree/master/libraries/EEPROM>`_** are included.

CHAPTER

NINE

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.

9.1 I2S Class API

9.1.1 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.

9.1.2 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.

9.1.3 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()`

9.1.4 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()`

9.1.5 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()`

9.1.6 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()`.

9.1.7 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.

9.1.8 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.

9.1.9 void end()

Stops the I2S device.

9.1.10 void flush()

Waits until all the I2S buffers have been output.

9.1.11 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.

9.1.12 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.

9.1.13 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.

9.1.14 int availableForWrite()

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

9.1.15 int read()

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

9.1.16 int peek()

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

9.1.17 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.

9.1.18 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.

9.2 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).

9.2.1 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.

9.2.2 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.

9.2.3 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.

9.2.4 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.

9.2.5 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.

9.2.6 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.

9.2.7 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.

9.2.8 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.

9.3 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

TEN

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()`

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()`

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

CHAPTER

ELEVEN

"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

TWELVE

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

THIRTEEN

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

FOURTEEN

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 FIFTEEN

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

SIXTEEN

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.

16.1 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>
```

16.2 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.

16.3 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.

16.4 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.

16.5 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.

16.6 File system object (LittleFS/SD/SDFS)

16.6.1 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.

16.6.2 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`

16.6.3 end

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

This method unmounts the file system.

16.6.4 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.

16.6.5 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
```

(continues on next page)

(continued from previous page)

```
       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");
}
```

16.6.6 exists

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

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

16.6.7 mkdir

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

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

16.6.8 rmdir

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

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

16.6.9 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.

16.6.10 remove

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

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

16.6.11 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.

16.6.12 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.

16.7 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)

16.7.1 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.

16.7.2 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.

16.8 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());
    }
}
```

16.8.1 next

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

16.8.2 fileName

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

16.8.3 fileSize

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

16.8.4 fileTime

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

16.8.5 fileCreationTime

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

16.8.6 isFile

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

16.8.7 isDirectory

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

16.8.8 openFile

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

16.8.9 rewind

Resets the internal pointer to the start of the directory.

16.8.10 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.

16.9 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.

16.9.1 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.

16.9.2 position

```
file.position()
```

Returns the current position inside the file, in bytes.

16.9.3 size

```
file.size()
```

Returns file size, in bytes.

16.9.4 name

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

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

16.9.5 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*`.

16.9.6 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.

16.9.7 getCreationTime

Returns the file creation time, if available.

16.9.8 isFile

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

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

16.9.9 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).

16.9.10 close

```
file.close()
```

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

16.9.11 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`.

16.9.12 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`.

16.9.13 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

SEVENTEEN

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.

17.1 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` and `Mouse` libraries. These libraries allow you to emulate a keyboard or mouse 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

17.2 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.

17.3 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

EIGHTEEN

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.

18.1 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).

18.1.1 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).

18.1.2 void rp2040.resumeOtherCore()

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

18.1.3 void rp2040.restartCore1()

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

18.2 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.

18.2.1 void rp2040.fifo.push(uint32_t)

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

18.2.2 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`.

18.2.3 uint32_t rp2040.fifo.pop()

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

18.2.4 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.

18.2.5 int rp2040.fifo.available()

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

CHAPTER

NINETEEN

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.

19.1 Enabling FreeRTOS

To enable FreeRTOS, simply add

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

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

19.2 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.

19.3 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.

19.4 More Information

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

CHAPTER

TWENTY

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

TWENTYONE

USING THE RASPBERRY PI PICO SDK (PICO-SDK)

21.1 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.

21.2 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.

21.3 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

TWENTYTWO

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.