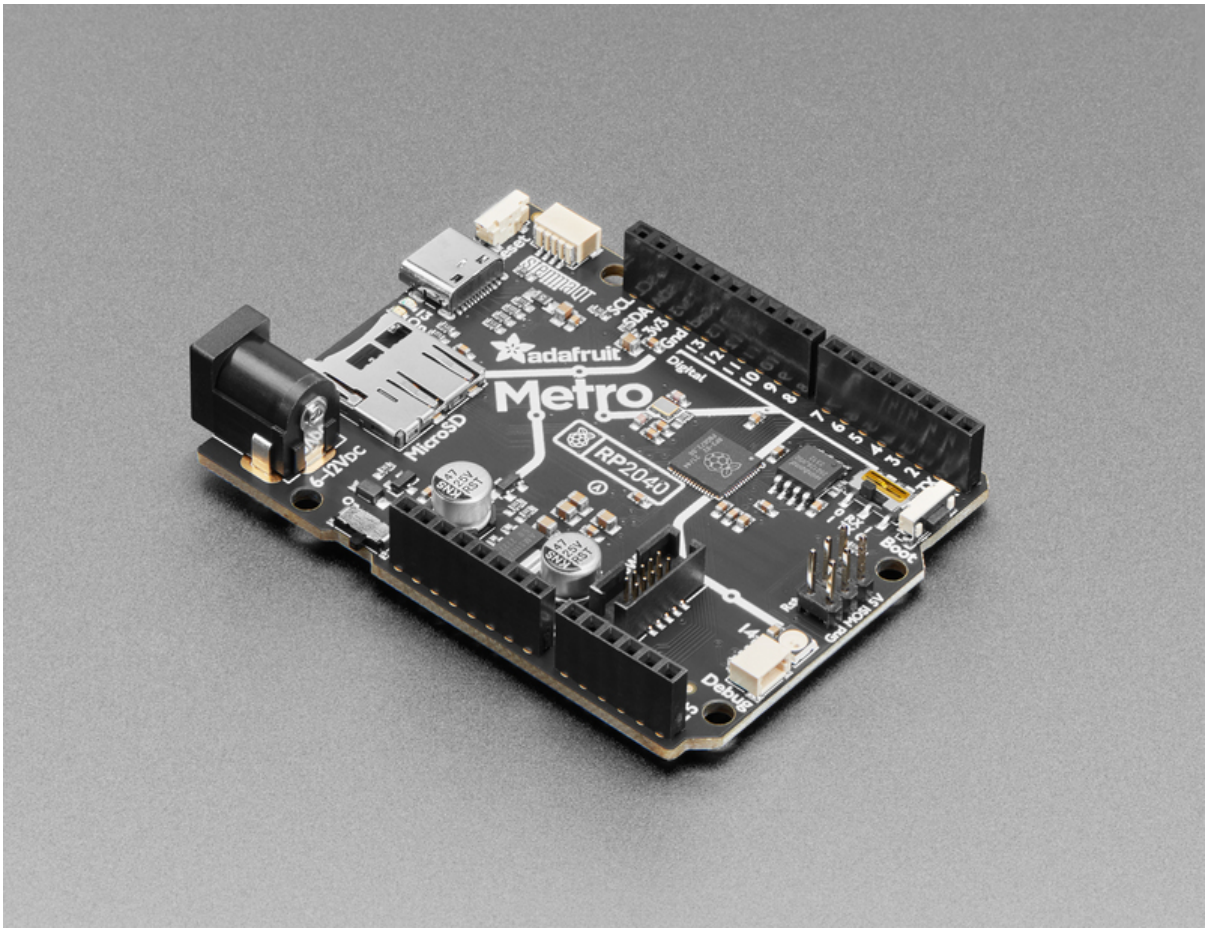
Adafruit Metro RP2040

Created by Kattni Rembor



https://learn.adafruit.com/adafruit-metro-rp2040

Last updated on 2024-02-24 09:21:42 PM EST

Table of Contents

Overview 7

Pinouts 12

- Power
- Power Inputs
- Power Control
- Power Outputs
- Logic Pins
- GPIO Pins by Pin Functionality
- RX/TX Switch
- RP2040 and QSPI Flash
- LEDs and NeoPixel
- STEMMA QT
- MicroSD Card Slot
- Buttons and RST Pin
- Debug Connectors

CircuitPython 23

- CircuitPython Quickstart
- Safe Mode
- Flash Resetting UF2

The CIRCUITPY Drive 27

- Boards Without CIRCUITPY

CircuitPython Pins and Modules 28

- CircuitPython Pins
- import board
- I2C, SPI, and UART
- What Are All the Available Names?
- Microcontroller Pin Names
- CircuitPython Built-In Modules

Installing the Mu Editor 34

- Download and Install Mu
- Starting Up Mu
- Using Mu

Creating and Editing Code 36

- Creating Code
- Editing Code
- Back to Editing Code...
- Naming Your Program File

Exploring Your First CircuitPython Program 41

- Imports & Libraries
- Setting Up The LED
- Loop-de-loops
- What Happens When My Code Finishes Running?
- What if I Don't Have the Loop?

Connecting to the Serial Console 44
• Are you using Mu?
• Serial Console Issues or Delays on Linux
• Setting Permissions on Linux
• Using Something Else?

Interacting with the Serial Console 47

The REPL 50
• Entering the REPL
• Interacting with the REPL
• Returning to the Serial Console

CircuitPython Libraries 54
• The Adafruit Learn Guide Project Bundle
• The Adafruit CircuitPython Library Bundle
• Downloading the Adafruit CircuitPython Library Bundle
• The CircuitPython Community Library Bundle
• Downloading the CircuitPython Community Library Bundle
• Understanding the Bundle
• Example Files
• Copying Libraries to Your Board
• Understanding Which Libraries to Install
• Example: ImportError Due to Missing Library
• Library Install on Non-Express Boards
• Updating CircuitPython Libraries and Examples
• CircUp CLI Tool

CircuitPython Documentation 65
• CircuitPython Core Documentation
• CircuitPython Library Documentation

Recommended Editors 73
• Recommended editors
• Recommended only with particular settings or add-ons
• Editors that are NOT recommended

Advanced Serial Console on Windows 74
• Windows 7 and 8.1
• What's the COM?
• Install Putty

Advanced Serial Console on Mac 78
• What's the Port?
• Connect with screen

Advanced Serial Console on Linux 80
• What's the Port?
• Connect with screen
• Permissions on Linux

Frequently Asked Questions 84
• Using Older Versions
• Python Arithmetic
• Wireless Connectivity

- Asyncio and Interrupts
- Status RGB LED
- Memory Issues
- Unsupported Hardware

Troubleshooting 89

- Always Run the Latest Version of CircuitPython and Libraries
- I have to continue using CircuitPython 7.x or earlier. Where can I find compatible libraries?
- macOS Sonoma 14.x: Disk Errors Writing to CIRCUITPY
- Bootloader (boardnameBOOT) Drive Not Present
- Windows Explorer Locks Up When Accessing boardnameBOOT Drive
- Copying UF2 to boardnameBOOT Drive Hangs at 0% Copied
- CIRCUITPY Drive Does Not Appear or Disappears Quickly
- Device Errors or Problems on Windows
- Serial Console in Mu Not Displaying Anything
- code.py Restarts Constantly
- CircuitPython RGB Status Light
- CircuitPython 7.0.0 and Later
- CircuitPython 6.3.0 and earlier
- Serial console showing ValueError: Incompatible .mpy file
- CIRCUITPY Drive Issues
- Safe Mode
- To erase CIRCUITPY: storage.erase_filesystem()
- Erase CIRCUITPY Without Access to the REPL
- For the specific boards listed below:
- For SAMD21 non-Express boards that have a UF2 bootloader:
- For SAMD21 non-Express boards that do not have a UF2 bootloader:
- Running Out of File Space on SAMD21 Non-Express Boards
- Delete something!
- Use tabs
- On MacOS?
- Prevent & Remove MacOS Hidden Files
- Copy Files on MacOS Without Creating Hidden Files
- Other MacOS Space-Saving Tips
- Device Locked Up or Boot Looping

Welcome to the Community! 109

- Adafruit Discord
- CircuitPython.org
- Adafruit GitHub
- Adafruit Forums
- Read the Docs

CircuitPython Essentials 118

Blink 121

- LED Location
- Blinking an LED

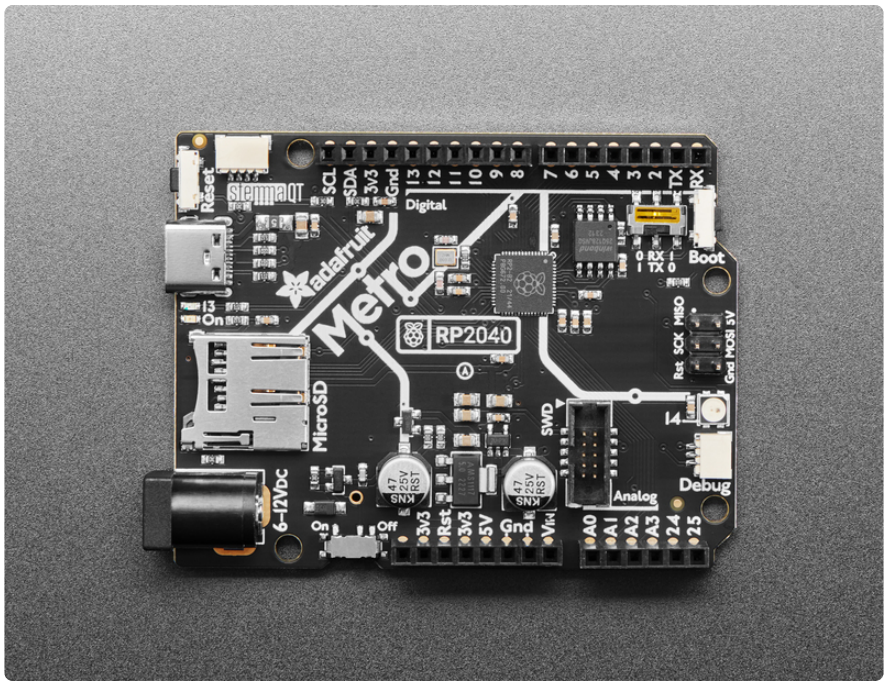
SD Card 123

- MicroSD Card Slot
- CircuitPython Usage
- SD Card Read Test
- SD Card Write Test

Digital Input 127
• LED and Button
• Controlling the LED with a Button
Analog In 128
• Analog to Digital Converter (ADC)
• Potentiometers
• Hardware
• Wire Up the Potentiometer
• Reading Analog Pin Values
• Reading Analog Voltage Values
NeoPixel LED 135
• NeoPixel Location
• NeoPixel Color and Brightness
• RGB LED Colors
• NeoPixel Rainbow
Storage 140
• The boot.py File
• The code.py File
• Logging the Temperature
• Recovering a Read-Only Filesystem
Capacitive Touch 144
• One Capacitive Touch Pin
• Pin Wiring
• Reading Touch on the Pin
• Multiple Capacitive Touch Pins
• Pin Wiring
• Reading Touch on the Pins
• Where are my Touch-Capable pins?
I2C 150
• I2C and CircuitPython
• Necessary Hardware
• Wiring the MCP9808
• Find Your Sensor
• I2C Sensor Data
• Where's my I2C?
asyncio 157
• asyncio Demonstration
• Wiring
• asyncio Example Code
• Code Walkthrough
• My program ended? What happened?
I2S 164
• I2S and CircuitPython
• Necessary Hardware
• Wiring the MAX98357A
• I2S Tone Playback
• I2S WAV File Playback
• CircuitPython I2S-Compatible Pin Combinations

PWM Audio 169
• Necessary Hardware
• Wiring the STEMMA Speaker
• PWM Tone Playback
• PWM WAV File Playback
CPU Temperature 173
• Microcontroller Location
• Reading the Microcontroller Temperature
Arduino IDE Setup 175
• Arduino IDE Download
• Adding the Philhower Board Manager URL
• Add Board Support Package
• Choose Your Board
Arduino Usage 178
• RP2040 Arduino Pins
• Choose Your Board
• Load the Blink Sketch
• Manually Enter the Bootloader
Blink 180
• Pre-Flight Check: Get Arduino IDE & Hardware Set Up
• Start up Arduino IDE and Select Board/Port
• New Blink Sketch
• Verify (Compile) Sketch
• Upload Sketch
• Native USB and manual bootloading
• Enter Manual Bootload Mode
• Finally, a Blink!
I2C Scan Test 189
• Common I2C Connectivity Issues
• Perform an I2C scan!
• Wiring the MCP9808
SD Card Example 194
• Library Installation
• Read/Write Example
Factory Reset 197
• Step 1. Download the factory-reset.uf2 file
• Step 2. Enter RP2040 bootloader mode
• Step 3. Drag UF2 file to RPI-RP2
• Flash Resetting UF2
Downloads 200
• Files
• Schematic and Fab Print

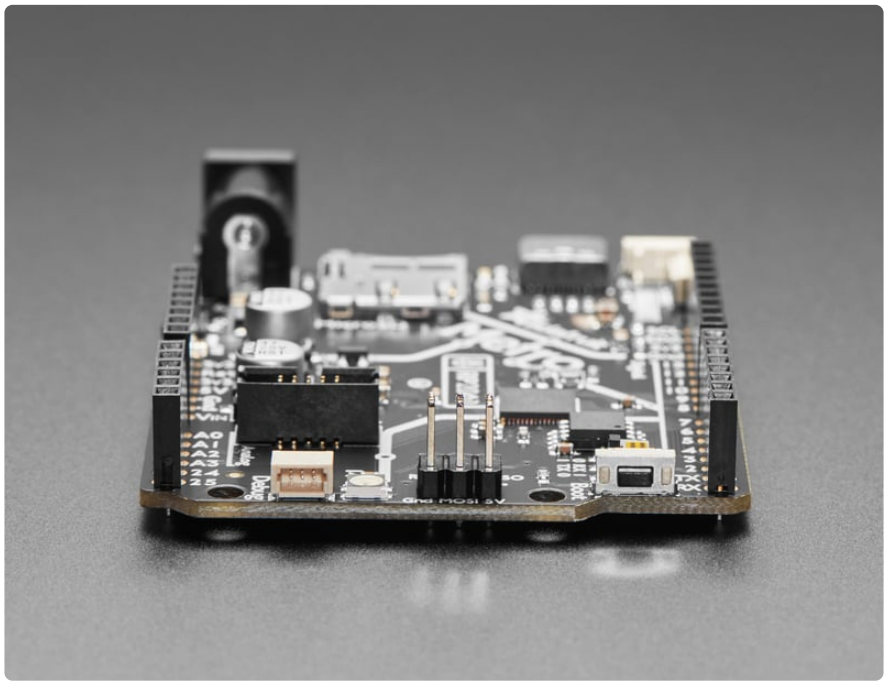
Overview



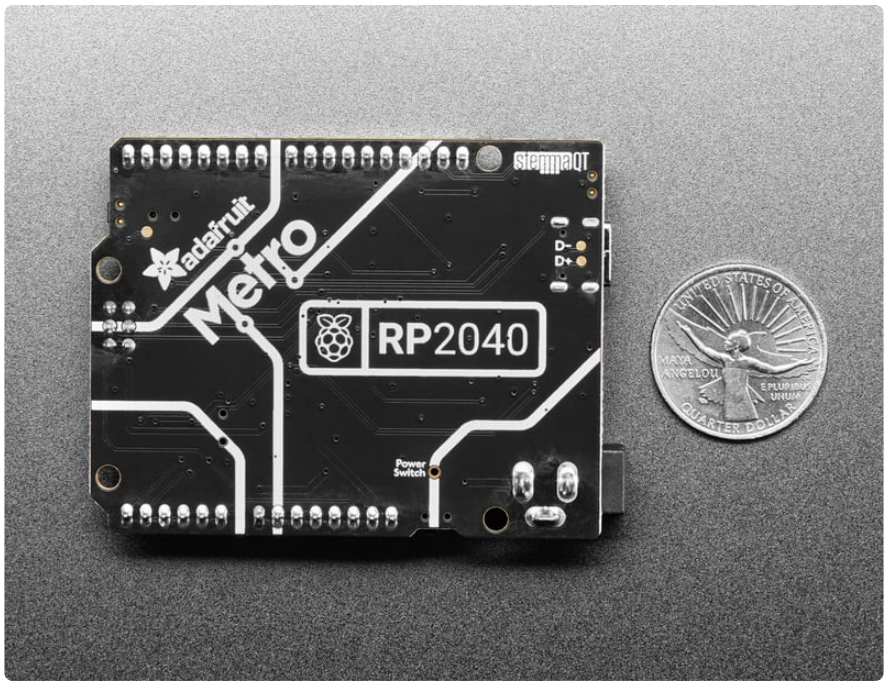
Choo! Choo! This is the RP2040 Metro Line, making all station stops at "Dual Cortex M0+ mountain", "264K RAM round-about" and "16 Megabytes of Flash town". This train is piled high with hardware that complements the Raspberry Pi RP2040 chip to make it an excellent development board for projects that want Arduino-shape-compatibility or just need the extra space and debugging ports.

- RP2040 main chip, 133MHz clock, 3.3V logic
- 16 MB of QSPI flash for program storage
- 24 GPIO, 4 of which are also analog inputs
- Micro SD card wired up for SPI interfacing, and also has extra pins connected for advanced-user SDIO interfacing (note that there's no released usage code for SDIO in Arduino/Python, so this is a super-cutting-edge setup).
- Onboard RGB NeoPixel
- Onboard #13 LED
- STEMMA QT/Qwiic port for I2C peripherals and sensors
- Reset and Boot buttons on PCB edge
- Pico Probe debug port - 3 pin JST SH compatible
- SWD debug port - 2x5 0.05" standard connector
- USB Type C power and data
- 5.5mm / 2.1mm DC jack for 6-12VDC power
- On/off switch for DC jack

- GPIO pin numbers match classic Arduino pins, other than A4/A5 which are D24 and D25 (there's only 4 ADC pins on the RP2040)
- RX / TX switch for swapping D0 and D1 locations



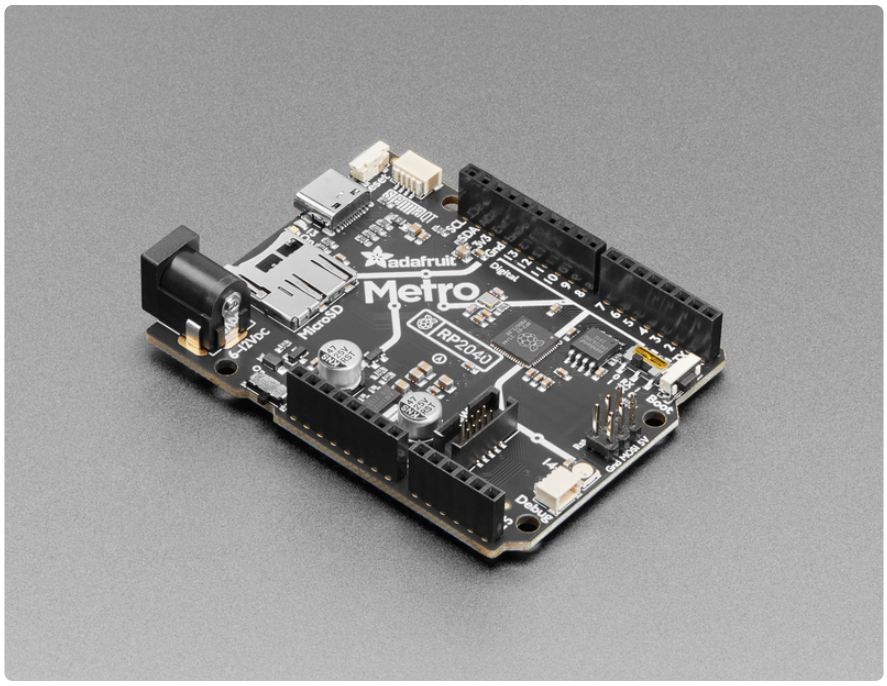
You may be wondering about the RX-TX switch: we added this because traditional Arduino boards start counting the GPIO for the digital pins with 0-7 and then 8-13. However, the D0/D1 pins are also traditionally the hardware UART Serial1, where D0 is Rx and D1 is Tx. On the RP2040, however, the UART pins are the other around: D0 is Tx and D1 is Rx. Thus a DPDT switch: flip one way to have the GPIO go in order of 0-7, flip the other way to have the logical locations of the hardware UART correct but now the pin order is 1, 0, 2, 3..7. Of course, it's also handy if, like us, you often swap the pins - now you don't need to require or cut/solder traces!



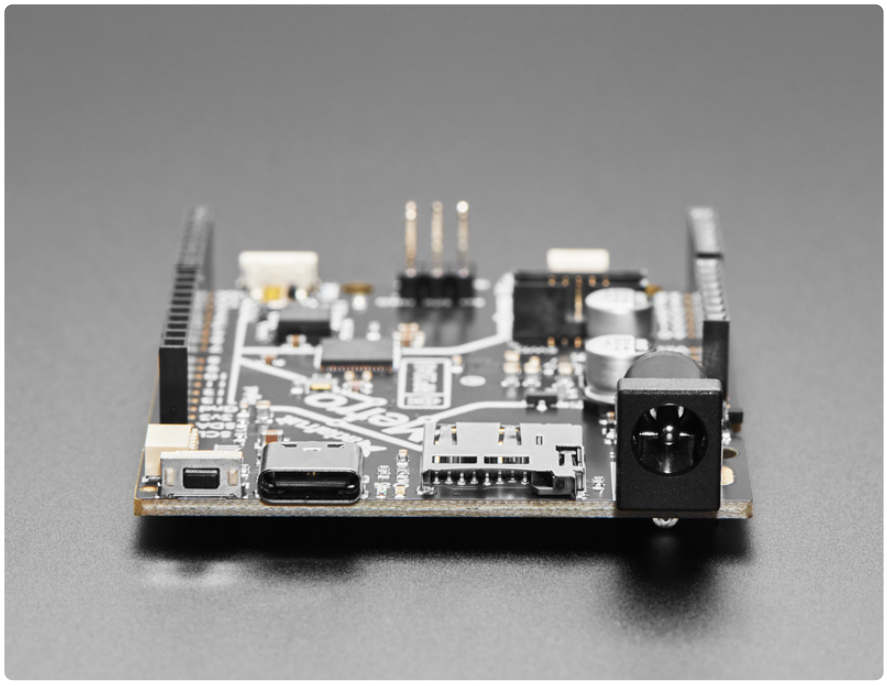
About the RP2040

The RP2040 is a powerful chip, which has the clock speed of our M4 (SAMD51), and two cores that are equivalent to our M0 (SAMD21). Since it is an M0 chip, it does not have a floating point unit or DSP hardware support. If you're doing something with heavy floating-point math, it will be done in software and thus not as fast as an M4. For many other computational tasks, you'll get close-to-M4 speeds!

For peripherals, there are two I2C controllers, two SPI controllers, and two UARTs that are multiplexed across the GPIO - check the pinout for what pins can be set to which. There are 16 PWM channels, each pin has a channel it can be set to (ditto on the pinout).

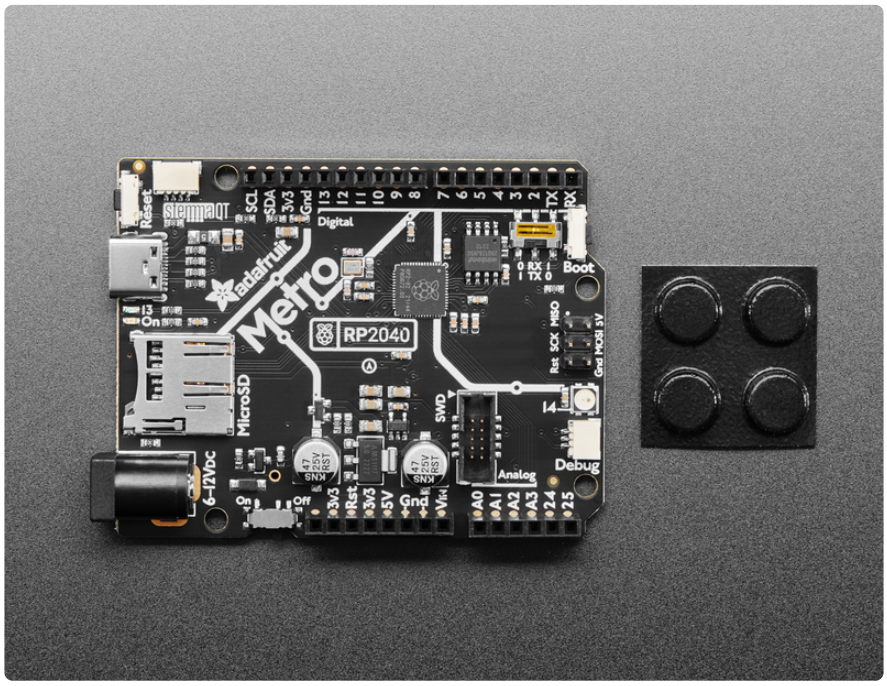


You'll note there's no I2S peripheral, or SDIO, or camera, what's up with that? Well, instead of having specific hardware support for serial-data-like peripherals like these, the RP2040 comes with the PIO state machine system, which is a unique and powerful way to create custom hardware logic and data processing blocks that run on their own without taking up CPU cycles. For example, NeoPixels. Often we bitbang the timing-specific protocol for these LEDs. For the RP2040, we instead use PIO object that reads in the data buffer and clocks out the right bitstream with perfect accuracy. The same with I2S audio in or out, LED matrix displays, 8-bit or SPI based TFTs, even VGA video (https://adafru.it/Qa2)! In MicroPython and CircuitPython, you can create PIO control commands to script the peripheral and load it in at runtime. There are 2 PIO peripherals with 4 state machines each.



There is great C/C++ support **(https://adafru.it/Qa3),** unofficial (but really good) Arduino support, **(https://adafru.it/11BA) an official** MicroPython port **(https://adafru.it/fa3), and a** CircuitPython port **(https://adafru.it/Em8)!** We, of course, recommend CircuitPython because we think it's the easiest way to get started (https://adafru.it/cpy-welcome), and it has support with most of our drivers, displays, sensors, and more, supported out of the box so you can follow along with our CircuitPython projects and tutorials.

While the RP2040 has lots of onboard RAM (264KB), it does not have built-in FLASH memory. Instead, that is provided by the external QSPI flash chip. **On this board, there is 16 MB**, which is shared between the program it's running and any file storage used by MicroPython or CircuitPython. When using C/C++, you get the whole flash memory, if using Python, you will have about 7 MB remaining for code, files, images, fonts, etc.

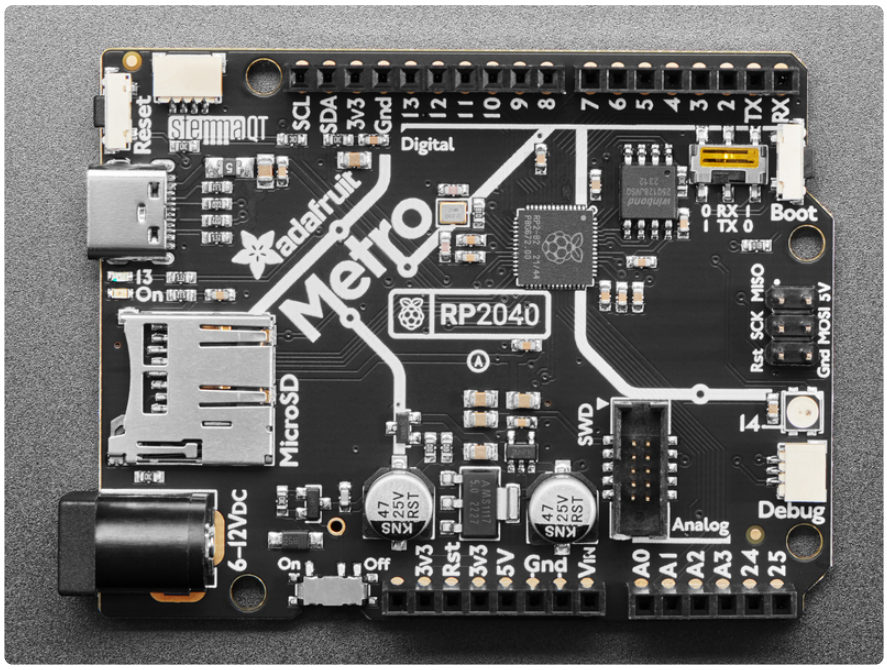


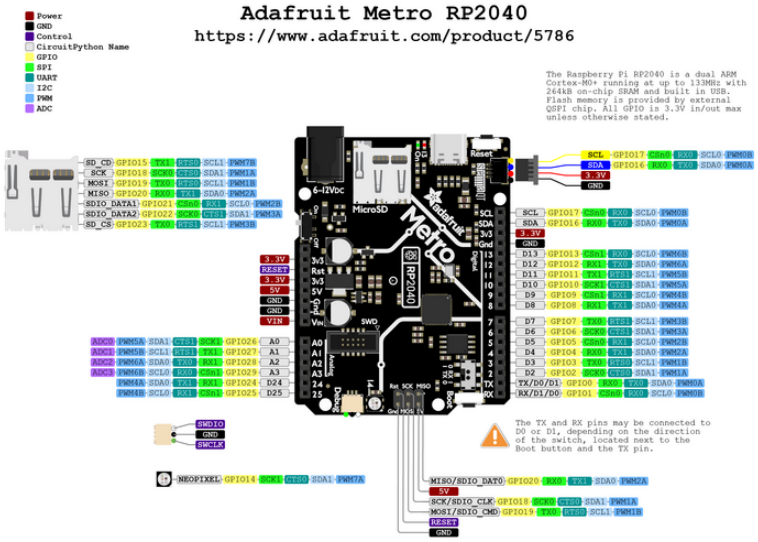
RP2040 Chip features:

- Dual ARM Cortex-M0+ @ 133MHz
- 264kB on-chip SRAM in six independent banks
- Support for up to 16MB of off-chip Flash memory via dedicated QSPI bus
- DMA controller
- Fully-connected AHB crossbar
- Interpolator and integer divider peripherals
- On-chip programmable LDO to generate core voltage
- 2 on-chip PLLs to generate USB and core clocks
- 30 GPIO pins, 4 of which can be used as analog inputs

- Peripherals
  - 2 UARTs
  - 2 SPI controllers
  - 2 I2C controllers
  - 16 PWM channels
  - USB 1.1 controller and PHY, with host and device support
  - 8 PIO state machines

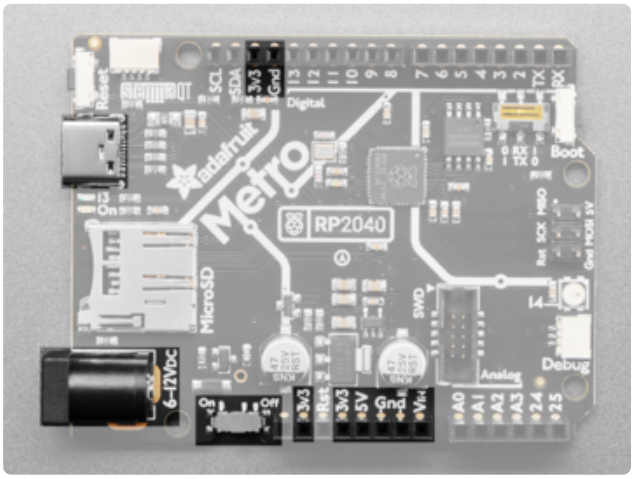
Pinouts





PrettyPins PDF on GitHub (https://adafru.it/18RF).

Power



The Metro RP2040 has two power input options, a power switch, and a few more power pins than your usual Metro/Arduino board.

Power Inputs

You have **two power input options:**

- **USB C port** - This is used for both powering and programming the board. You can power it with any USB C cable and will request 5V from a USB C PD.
- **DC barrel jack** - The DC Jack is a 5.5mm/2.1mm center-positive DC connector, which is the most common available. Provide about 6V-12V here to power the Metro. Great for when you have a wall adapter power supply. Don't use a center-negative adapter, it won't work (the On green LED will not light)

Power Control

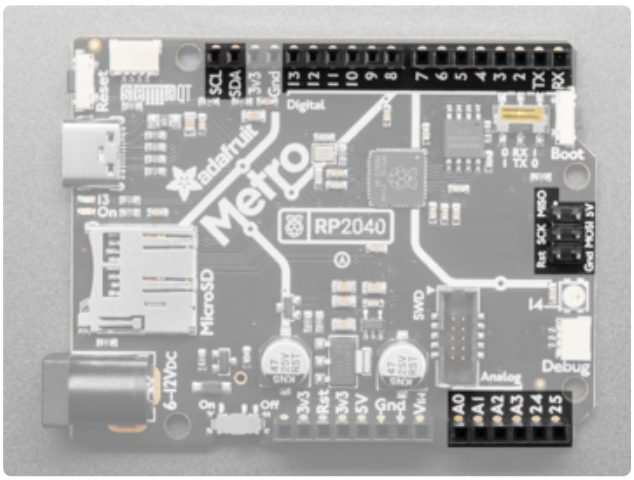
- **On/Off switch** - This switch controls the DC jack power only. If you plug in your board and nothing happens, make sure the switch is flipped to "ON"!

Power Outputs

- **3.3V** - This is the output pin from the 3.3V regulator. It can supply up to around 400mA.
- **5V** - This is the output pin from the 5V regulator (when DC jack is used), or from USB. It can supply up to around 500mA peak from USB and up to 800mA peak from DC.
- **GND** - This is the common ground for all power and logic.

- **VIN** - This is the higher of the DC jack or USB voltage. So if the DC jack is plugged in and 9V, VIN is 9V. If only USB connected, this will be 5V.

Logic Pins



These are the logic pins that can be used to connect shields, sensors, servos, LEDs and more!

I2C and SPI on RP2040

The RP2040 is capable of handling I2C, SPI and UART on many pins. However, there are really only two peripherals each of I2C, SPI and UART: I2C0 and I2C1, SPI0 and SPI1, and UART0 and UART1. So while many pins are capable of I2C, SPI and UART, you can only do two at a time, and only on separate peripherals, 0 and 1. I2C, SPI and UART peripherals are included and numbered below.

PWM on RP2040

The RP2040 supports PWM on all pins. However, it is not capable of PWM on all pins at the same time. There are 8 PWM "slices", each with two outputs, A and B. Each pin on the Feather is assigned a PWM slice and output. For example, A0 is PWM5 A, which means it is first output of the fifth slice. You can have up to 16 PWM objects on the Feather RP2040. The important thing to know is that **you cannot use the same slice and output more than once at the same time**. So, if you have a PWM object on pin A0, you cannot also put a PWM object on D10, because they are both PWM5 A. The PWM slices and outputs are indicated below. Note that PWM7 A and PWM7 B are not available on the Metro RP2040 because not all pins are broken out.

Analog Pins

The RP2040 has four ADCs. These pins are the only pins capable of handling analog, and they can also do digital.

- **A0/GPIO26** - This pin is ADC0. It is also SPI1 SCK, I2C1 SDA and PWM5 A.
- **A1/GPIO27** - This pin is ADC1. It is also SPI1 MOSI, I2C1 SCL and PWM5 B.
- **A2/GPIO28** - This pin is ADC2. It is also SPI1 MISO, UART0 TX, I2C1 SDA and PWM6 A.
- **A3/GPIO29** - This pin is ADC3. It is also SPI1 CS, UART0 RX, I2C0 SCL and PWM6 B.

Digital Pins

These are the digital I/O pins. They all have multiple capabilities.

- **D24/GPIO24** - Digital I/O pin 24. It is also UART1 TX, I2C0 SDA, and PWM4 A.
- **D25/GPIO25** - Digital I/O pin 25. It is also UART1 RX, I2C0 SCL, and PWM4 B.
- **SCL/GPIO17** - This is the main I2C0 SCL. It is also SPI0 CS, UART0 RX, and PWM0 B.
- **SDA/GPIO16** - This is the main I2C0 SDA. It is also SPI0 MISO, UART0 TX, and PWM0 A.
- **D13/GPIO13** - Digital I/O pin 13. It is also SPI1 CS, UART0 RX, I2C0 SCL and PWM6 B.
- **D12/GPIO12** - Digital I/O pin 12. It is also SPI1 MISO, UART0 TX, I2C0 SDA and PWM6 A.
- **D11/GPIO11** - Digital I/O pin 11. It is also SPI1 MOSI, I2C1 SCL and PWM5 B.
- **D10/GPIO10** - Digital I/O pin 10. It is also SPI1 SCK, I2C1 SDA and PWM5 A.
- **D9/GPIO9** - Digital I/O pin 9. It is also SPI1 CS, UART1 RX, I2C0 SCL and PWM4 B.
- **D8/GPIO8** - Digital I/O pin 8. It is also SPI1 MISO, UART1 TX, I2C0 SDA, and PWM4 A.
- **D7/GPIO7** - Digital I/O pin 7. It is also SPI0 MOSI, I2C1 SCL, and PWM3 B.
- **D6/GPIO6** - Digital I/O pin 6. It is also SPI0 SCK, I2C1 SDA, and PWM3 A.
- **D5/GPIO5** - Digital I/O pin 5. It is also SPI0 CS, UART1 RX, I2C0 SCL, and PWM2 B.
- **D4/GPIO4** - Digital I/O pin 4. It is also SPI0 MISO, UART1 TX, I2C0 SDA and PWM2 A.
- **D3/GPIO3** - Digital I/O pin 3. It is also SPI0 MOSI, I2C1 SCL and PWM1 B.
- **D2/GPIO2** - Digital I/O pin 2. It is also SPI0 SCK, I2C1 SDA and PWM1 A.

- **TX/D0/GPIO0** - The main UART0 TX pin. It is also I2C0 SCL, SPI0 MISO and PWM0 A.
- **RX/D1/GPIO1** - The main UART0 RX pin. It is also I2C0 SDA, SPI0 CS and PWM0 B.
- **SCK/SDIO_CLK/GPIO18** - This is the main SPI0 SCK pin. It is also I2C1 SDA, and PWM1 A.
- **MOSI/SDIO_CMD/GPIO19** - This is the main SPI0 MOSI pin. It is also I2C1 SCL, and PWM1 B.
- **MISO/SDIO_DAT/GPIO20** - This is the main SPI0 MISO pin. It is also UART1 TX, I2C0 SDA, and PWM2 A.

CircuitPython I2C, SPI and UART

Note that in CircuitPython, there is a board object each for STEMMA QT, I2C, SPI and UART that use the connector and pins labeled on the Metro. You can use these objects to initialize these peripherals in your code.

- `board.STEMMA_I2C()` uses the STEMMA QT connector (in this case, SCL/SDA pins)
- `board.I2C()` uses SCL/SDA pins (GPIO17 and GPIO16)
- `board.SPI()` uses SCK/MO/MI pins (GPIO18, GPIO19 and GPIO20)
- `board.UART()` uses TX/RX pins (GPIO0 and GPIO1)

Arduino I2C, SPI and UART

I2C, SPI and UART can be accessed with these objects in Arduino:

- `Wire` is used for the default I2C and STEMMA QT connector (GPIO16 and GPIO17).
- `SPI` is used for the default SPI pins (GPIO18, GPIO19 and GPIO20).
- `Serial1` is used for the default UART pins (GPIO0 and GPIO1).

GPIO Pins by Pin Functionality

Primary pins based on the silkscreen pin labels are bold.

I2C Pins

- I2C0 SCL: **SCL**, A3, D25, RX, D17, D13, D9, D5

- I2C0 SDA: **SDA**, A2, D24, TX, D20, D16, D12, D8, D4
- I2C1 SCL: A1, D19, D11, D7, D3
- I2C1 SDA: A0, D18, D10, D6, D2

SPI Pins

- SPI0 SCK: **SCK**, D6, D2
- SPI0 MOSI: **MOSI**, D7, D3
- SPI0 MISO: **MISO**, TX, D16, D4
- SPI0 CS: RX, D5, D17
- SPI1 SCK: A0, D10
- SPI1 MOSI: A1, D11
- SPI1 MISO: A2, D24, D12, D8
- SPI1 CS: A3, D25, D13, D9

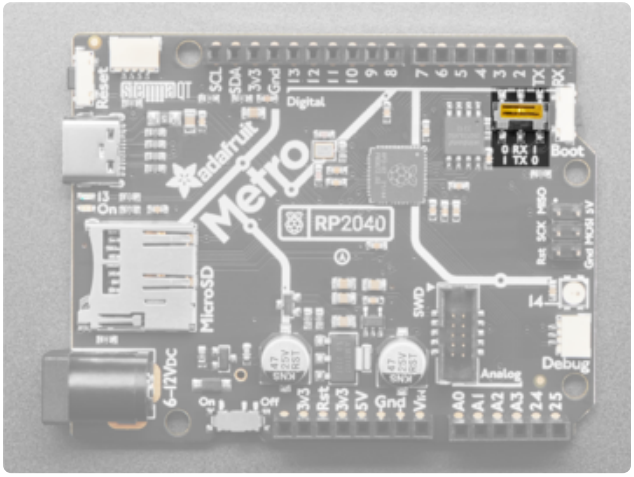
UART Pins

- UART0 TX: **TX**, A2, D12, D16
- UART0 RX: **RX**, A3, D13, D4, D17
- UART1 TX: D24, D4, D8, MISO
- UART1 RX: D25, D9, D5

PWM Pins

- PWM0 A: TX, SDA
- PWM0 B: RX, SCL
- PWM1 A: D2, SCK
- PWM1 B: D3, MOSI
- PWM2 A: D4, MISO
- PWM2 B: D5
- PWM3 A: D6
- PWM3 B: D7
- PWM4 A: D24, D8
- PWM4 B: D25, D9
- PWM5 A: A0, D10
- PWM5 B: A1, D11
- PWM6 A: A2, D12
- PWM6 B: A3, D13

RX/TX Switch



The switch located at the top right corner of the board near the Boot button is the **RX/TX switch**.

When the **switch is to the left**, the pins are as follows:

RX = **0**
TX = **1**
When the **switch is to the right**, the pins are as follows:

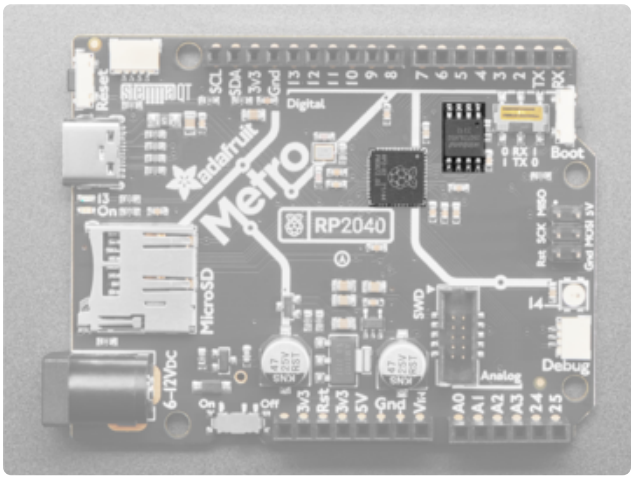
RX = **1**
TX = **0**

The switch on your Metro RP2040 may arrive in either position. If you're running into issues with RX and TX, verify that the switch is in the position you expect it to be.

You may be wondering about the RX-TX switch: we added this because traditional Arduino board start counting the GPIO for the digital pins with 0-7 and then 8-13. However, the D0/D1 pins are also traditionally the hardware UART Serial1, where D0 is Rx and D1 is Tx. On the RP2040, however, the UART pins are the other around: D0 is Tx and D1 is Rx.

Thus a DPDT switch: flip one way to have the GPIO go in order of 0-7, flip the other way to have the logical locations of the hardware UART correct but now the pin order is 1, 0, 2, 3..7. Of course, it's also handy if, like us, you often swap the pins - now you don't need to rewire or cut and solder traces!

RP2040 and QSPI Flash



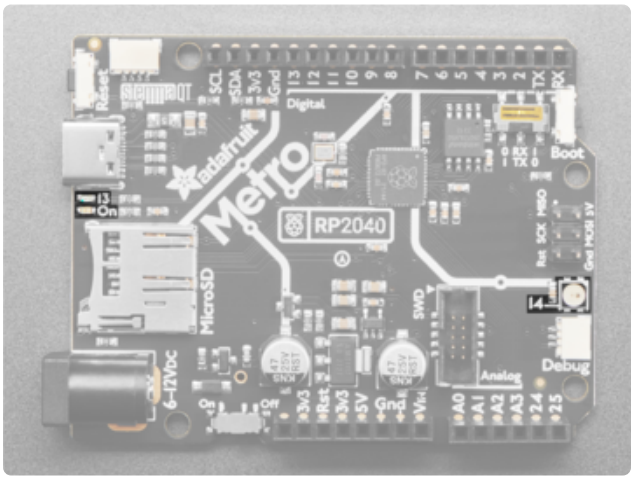
The square above the RP2040 label on the board silk is the **RP2040 microcontroller**, the "brains" of the Metro RP2040 board.

The square near the TX/RX switch is the **QSPI Flash**. It is connected to 6 pins that are not brought out on the GPIO pads. This way you don't have to worry about the SPI flash memory accesses colliding with other devices on the main SPI connection.

QSPI is neat because it allows you to have 4 data in/out lines instead of just SPI's single line in and single line out. This means that QSPI is at least 4 times faster. But in reality is at least 10x faster because you can clock the QSPI peripheral much faster than a plain SPI peripheral.

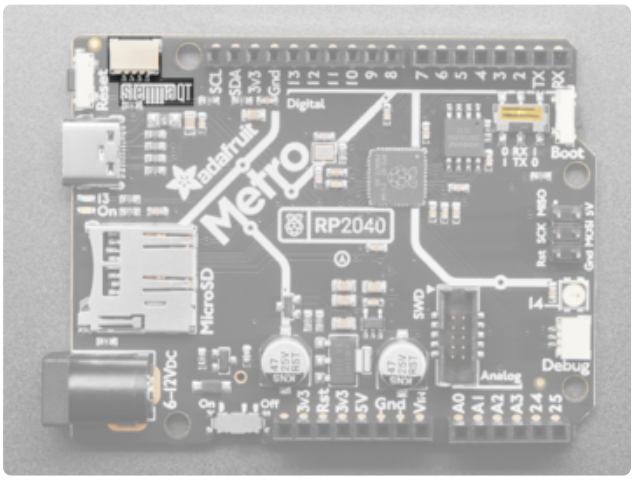
The Metro RP2040 comes with **16 MB of flash memory**, which is shared between the program it's running and any file storage used by MicroPython or CircuitPython. When using C/C++, you get the whole flash memory, if using Python, you will have about 7 MB remaining for code, files, images, fonts, etc.

LEDs and NeoPixel



NeoPixel LED - This addressable RGB NeoPixel LED is both a status LED and user controllable on **14**. It is available in CircuitPython as `board.NEOPIXEL`, and in Arduino as `PIN_NEOPIXEL`.
LED - This red LED on **13** is user controllable for blinky needs. It is available in CircuitPython as `board.LED` and in Arduino as `PIN_LED`.
On LED - This green LED indicates the board is powered on, it is connected to the 3.3V power supply.

STEMMA QT



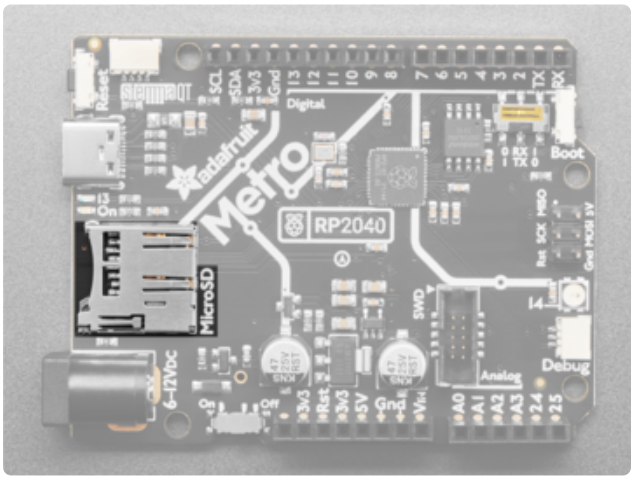
STEMMA QT **(https://adafru.it/Ft4)** - This JST SH 4-pin connector breaks out I2C (SCL, SDA, 3.3V, GND). It allows you to connect to various breakouts and sensors with **STEMMA QT** connectors (https://adafru.it/HMF) or to other things using assorted associated accessories (https://adafru.it/Ft6).

Works great with any STEMMA QT or Qwiic sensor/device. You can also use it with Grove I2C devices thanks to this handy cable (http://adafru.it/4528).

In CircuitPython, you can use the STEMMA connector with `board.SCL` and `board.SDA`, or `board.STEMMA_I2C()`.

In Arduino, you can use the STEMMA connector with `16` (SCL) and `17` (SDA), or `Wire`.

MicroSD Card Slot



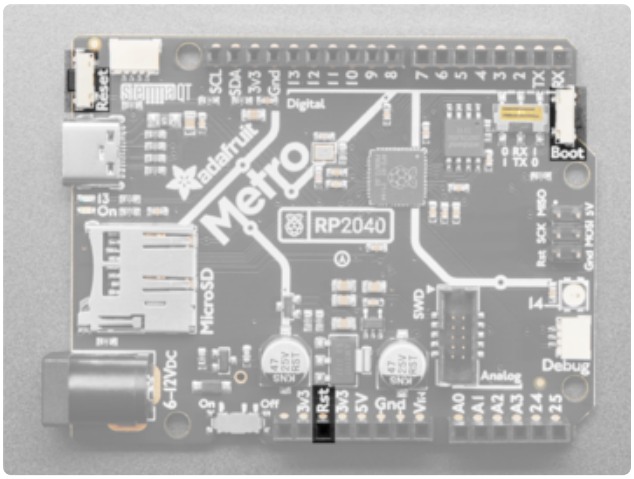
The Metro RP2040 comes with a microSD card slot. It is wired up for SPI interfacing. It also has extra pins connected for advanced-user SDIO interfacing (note that there's no released usage code for SDIO in Arduino/Python, so this is a super-cutting-edge setup).

These are the pins for the MicroSD card slot.

- **SD_CD/GPIO15** - This is the card detect pin. It is also SPI1 MOSI, I2C1 SCL, and PWM7 B.
- **SCK/GPIO18** - This is the main SPI0 SCK pin. It is also I2C1 SDA, and PWM1 A.

- **MOSI/GPIO19** - This is the main SPI0 MOSI pin. It is also I2C1 SCL, and PWM1 B.
- **MISO/GPIO20** - This is the main SPI0 MISO pin. It is also UART1 TX, I2C0 SDA, and PWM2 A.
- **SDIO_DATA1/GPIO21** - This is the SDIO data1 pin. It is also SPI0 CS, UART1 RX, I2C0 SCL and PWM2 B.
- **SDIO_DATA2/GPIO22** - This is the SDIO data2 pin. It is also SPI0 SCK, I2C1 SDA, and PWM3 A.
- **SD_CS/GPIO23** - This is the chip select pin. It is also SPI0 MOSI, I2C1 SCL, and PWM3 B.

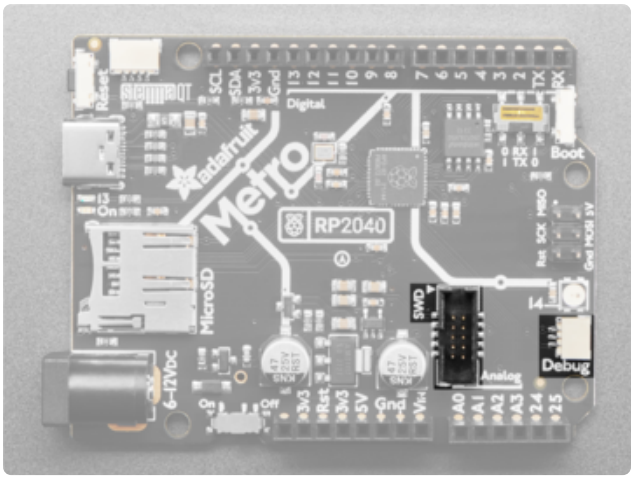
Buttons and RST Pin



The **RESET button** restarts the board and helps enter the bootloader. You can click it to reset the board without unplugging the USB cable or battery.
The **RST pin** is can be used to reset the board. Tie to ground manually to reset the board.
The **Boot button** is used to enter the bootloader.

To enter the bootloader, press and hold Boot and then power up the board (either by plugging it into USB or pressing RESET). The bootloader is used to install/update CircuitPython. It is also sometimes necessary to enter the bootloader to load an Arduino sketch.

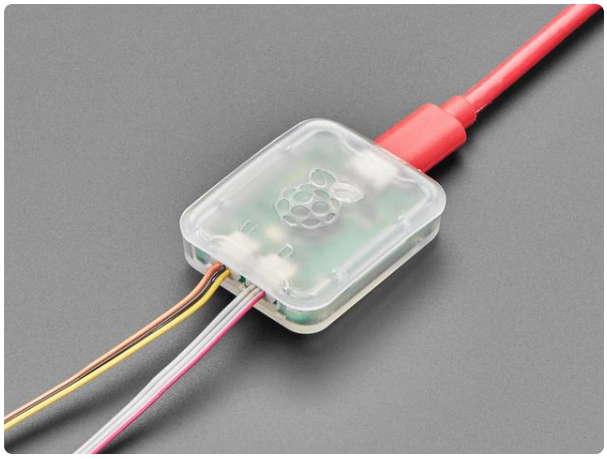
Debug Connectors



There are two different debug interfaces on the Metro RP2040:

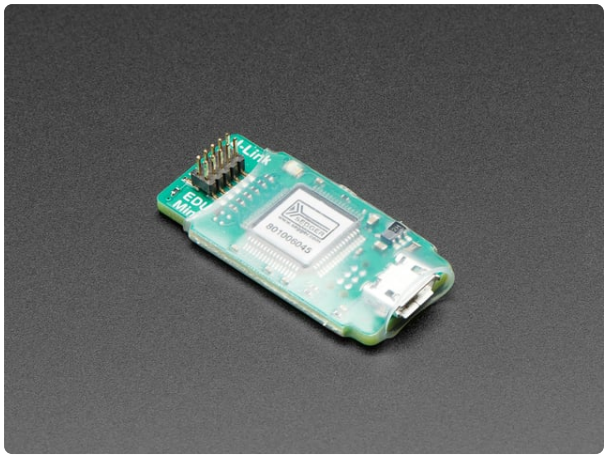
Debug - The **Pico Probe debug port**
SWD - The **JTAG interface**

The connector labeled **Debug** is the **Pico Probe debug port**. The Raspberry Pi Debug Probe is an all-in-one USB-to-debug kit that provides all the necessary hardware and cables for easy, solderless, plug-and-play debugging. For when "printf-style-debugging" just isn't enough, this kit which brings the power of gdb & OpenOCD step and trace debugging to the RP2040 chipset.



Raspberry Pi Debug Probe Kit for Pico and RP2040
The Raspberry Pi Debug Probe is an all-in-one USB-to-debug kit that provides all the necessary hardware and cables for easy, solderless, plug-and-play debugging. For when...
https://www.adafruit.com/product/5699

The connector labeled **SWD** is the **JTAG debug interface**, available if you'd like to do more advanced development, trace-debugging, or not use the bootloader. A J-Link or similar external hardware device is needed to perform debugging.



SEGGER J-Link EDU Mini - JTAG/SWD Debugger
Doing some serious development on any ARM-based platform, and tired of 'printf' plus an LED to debug? A proper JTAG/SWD HW debugger can make debugging more of a pleasure and...
https://www.adafruit.com/product/3571

CircuitPython

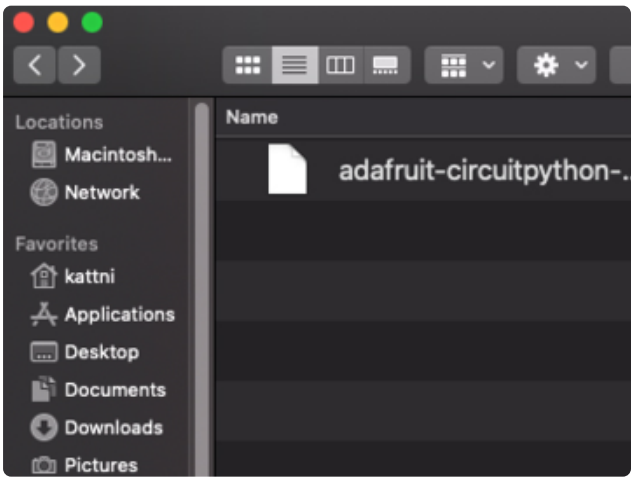
CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the **CIRCUITPY** drive to iterate.

CircuitPython Quickstart

Follow this step-by-step to quickly get CircuitPython running on your board.

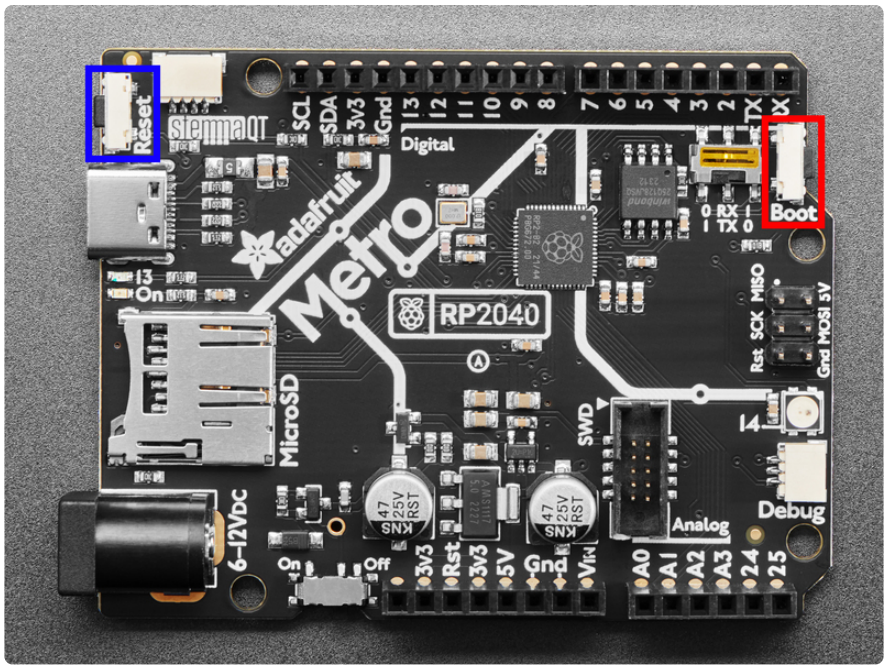
Download the latest version of CircuitPython for this board via circuitpython.org

https://adafru.it/18Sb



Click the link above to download the latest CircuitPython UF2 file.

Save it wherever is convenient for you.

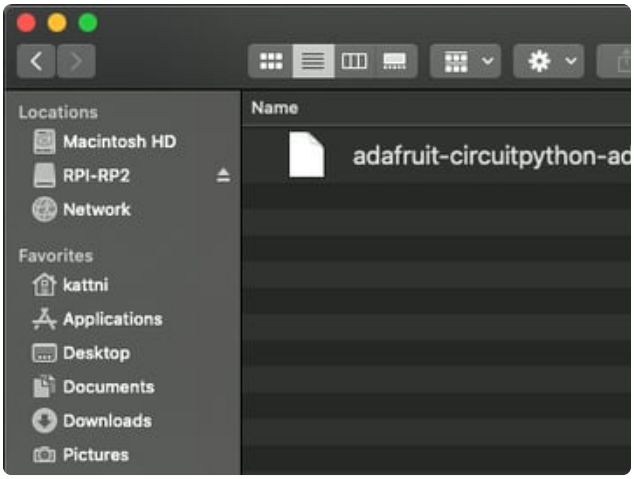


To enter the bootloader, hold down the **BOOT/BOOTSEL button** (highlighted in red above), and while continuing to hold it (don't let go!), press and release the **reset button** (highlighted in blue above). **Continue to hold the BOOT/BOOTSEL button until the RPI-RP2 drive appears!**

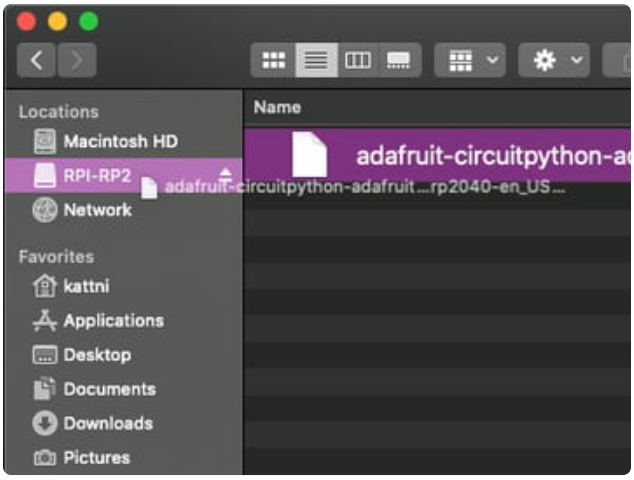
If the drive does not appear, release all the buttons, and then repeat the process above.

You can also start with your board unplugged from USB, press and hold the BOOTSEL button (highlighted in red above), continue to hold it while plugging it into USB, and wait for the drive to appear before releasing the button.

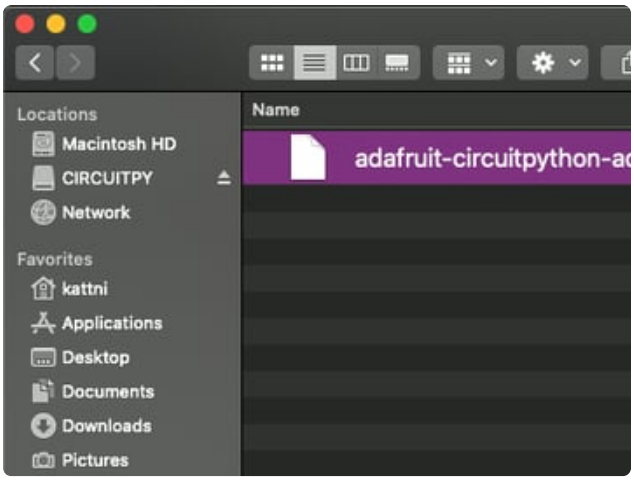
A lot of people end up using charge-only USB cables and it is very frustrating! **Make sure you have a USB cable you know is good for data sync.**



You will see a new disk drive appear called **RPI-RP2**.



Drag the **adafruit_circuitpython_etc.uf2** file to **RPI-RP2.**



The **RPI-RP2** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

That's it, you're done! :)

Safe Mode

You want to edit your **code.py** or modify the files on your **CIRCUITPY** drive, but find that you can't. Perhaps your board has gotten into a state where **CIRCUITPY** is read-only. You may have turned off the **CIRCUITPY** drive altogether. Whatever the reason, safe mode can help.

Safe mode in CircuitPython does not run any user code on startup, and disables auto-reload. This means a few things. First, safe mode bypasses any code in **boot.py** (where you can set **CIRCUITPY** read-only or turn it off completely). Second, it does not run the code in **code.py**. And finally, it does not automatically soft-reload when data is written to the **CIRCUITPY** drive.

Therefore, whatever you may have done to put your board in a non-interactive state, safe mode gives you the opportunity to correct it without losing all of the data on the **CIRCUITPY** drive.

Entering Safe Mode

To enter safe mode when using CircuitPython, plug in your board or hit reset (highlighted in red above). Immediately after the board starts up or resets, it waits 1000ms. On some boards, the onboard status LED (highlighted in green above) will blink yellow during that time. If you press reset during that 1000ms, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a slow double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

In Safe Mode

If you successfully enter safe mode on CircuitPython, the LED will intermittently blink yellow three times.

If you connect to the serial console, you'll find the following message.

```
Auto-reload is off.
Running in safe mode! Not running saved code.

CircuitPython is in safe mode because you pressed the reset button during boot.
Press again to exit safe mode.

Press any key to enter the REPL. Use CTRL-D to reload.
```

You can now edit the contents of the **CIRCUITPY** drive. Remember, your code will not run until you press the reset button, or unplug and plug in your board, to get out of safe mode.

Flash Resetting UF2

If your board ever gets into a really weird state and doesn't even show up as a disk drive when installing CircuitPython, try loading this 'nuke' UF2 which will do a 'deep

clean' on your Flash Memory. **You will lose all the files on the board**, but at least you'll be able to revive it! After loading this UF2, follow the steps above to re-install CircuitPython.

Download flash erasing "nuke" UF2

https://adafru.it/RLE

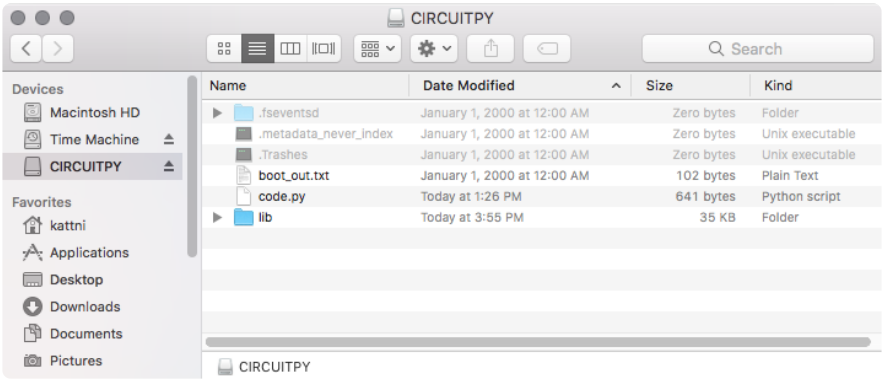
The CIRCUITPY Drive

When CircuitPython finishes installing, or you plug a CircuitPython board into your computer with CircuitPython already installed, the board shows up on your computer as a USB drive called **CIRCUITPY**.

The **CIRCUITPY** drive is where your code and the necessary libraries and files will live. You can edit your code directly on this drive and when you save, it will run automatically. When you create and edit code, you'll save your code in a **code.py** file located on the **CIRCUITPY** drive. If you're following along with a Learn guide, you can paste the contents of the tutorial example into **code.py** on the **CIRCUITPY** drive and save it to run the example.

With a fresh CircuitPython install, on your **CIRCUITPY** drive, you'll find a **code.py** file containing `print("Hello World!")` and an empty **lib** folder. If your **CIRCUITPY** drive does not contain a **code.py** file, you can easily create one and save it to the drive. CircuitPython looks for **code.py** and executes the code within the file automatically when the board starts up or resets. Following a change to the contents of **CIRCUITPY**, such as making a change to the **code.py** file, the board will reset, and the code will be run. You do not need to manually run the code. This is what makes it so easy to get started with your project and update your code!

Note that all changes to the contents of **CIRCUITPY**, such as saving a new file, renaming a current file, or deleting an existing file will trigger a reset of the board.



Boards Without CIRCUITPY

CircuitPython is available for some microcontrollers that do not support native USB. Those boards cannot present a **CIRCUITPY** drive. This includes boards using ESP32 or ESP32-C3 microcontrollers.

On these boards, there are alternative ways to transfer and edit files. You can use the Thonny editor (https://adafru.it/18e7), which uses hidden commands sent to the REPL to read and write files. Or you can use the CircuitPython web workflow, introduced in Circuitpython 8. The web workflow provides browser-based WiFi access to the CircuitPython filesystem. These guides will help you with the web workflow:

- CircuitPython on ESP32 Quick Start (https://adafru.it/10JF)
- CircuitPython Web Workflow Code Editor Quick Start (https://adafru.it/18e8)

CircuitPython Pins and Modules

CircuitPython is designed to run on microcontrollers and allows you to interface with all kinds of sensors, inputs and other hardware peripherals. There are tons of guides showing how to wire up a circuit, and use CircuitPython to, for example, read data from a sensor, or detect a button press. Most CircuitPython code includes hardware setup which requires various modules, such as `board` or `digitalio`. You import these modules and then use them in your code. How does CircuitPython know to look for hardware in the specific place you connected it, and where do these modules come from?

This page explains both. You'll learn how CircuitPython finds the pins on your microcontroller board, including how to find the available pins for your board and what each pin is named. You'll also learn about the modules built into CircuitPython, including how to find all the modules available for your board.

CircuitPython Pins

When using hardware peripherals with a CircuitPython compatible microcontroller, you'll almost certainly be utilising pins. This section will cover how to access your board's pins using CircuitPython, how to discover what pins and board-specific objects are available in CircuitPython for your board, how to use the board-specific objects, and how to determine all available pin names for a given pin on your board.

`import board`

When you're using any kind of hardware peripherals wired up to your microcontroller board, the import list in your code will include `import board`. The `board` module is built into CircuitPython, and is used to provide access to a series of board-specific objects, including pins. Take a look at your microcontroller board. You'll notice that next to the pins are pin labels. You can always access a pin by its pin label. However, there are almost always multiple names for a given pin.

To see all the available board-specific objects and pins for your board, enter the REPL (`>>>`) and run the following commands:

```
import board
dir(board)
```

Here is the output for the QT Py SAMD21. **You may have a different board, and this list will vary, based on the board.**

```
>>> import board
>>> dir(board)
['__class__', 'A0', 'A1', 'A10', 'A2', 'A3', 'A6', 'A7', 'A8', 'A9', 'D0', 'D1',
 'D10', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7', 'D8', 'D9', 'I2C', 'MISO', 'MOSI', '
NEOPIXEL', 'NEOPIXEL_POWER', 'RX', 'SCK', 'SCL', 'SDA', 'SPI', 'TX', 'UART']
```

The following pins have labels on the physical QT Py SAMD21 board: A0, A1, A2, A3, SDA, SCL, TX, RX, SCK, MISO, and MOSI. You see that there are many more entries available in `board` than the labels on the QT Py.

You can use the pin names on the physical board, regardless of whether they seem to be specific to a certain protocol.

For example, you do not have to use the SDA pin for I2C - you can use it for a button or LED.

On the flip side, there may be multiple names for one pin. For example, on the QT Py SAMD21, pin **A0** is labeled on the physical board silkscreen, but it is available in CircuitPython as both `A0` and `D0`. For more information on finding all the names for a given pin, see the What Are All the Available Pin Names? (https://adafru.it/QkA) section below.

The results of `dir(board)` for CircuitPython compatible boards will look similar to the results for the QT Py SAMD21 in terms of the pin names, e.g. A0, D0, etc. However, some boards, for example, the Metro ESP32-S2, have different styled pin names. Here is the output for the Metro ESP32-S2.

```
>>> import board
>>> dir(board)
['__class__', 'A0', 'A1', 'A2', 'A3', 'A4', 'A5', 'DEBUG_RX', 'DEBUG_TX', 'I2C',
 'IO1', 'IO10', 'IO11', 'IO12', 'IO13', 'IO14', 'IO15', 'IO16', 'IO17', 'IO18',
'IO2', 'IO21', 'IO3', 'IO33', 'IO34', 'IO35', 'IO36', 'IO37', 'IO4', 'IO42', 'IO
45', 'IO5', 'IO6', 'IO7', 'IO8', 'IO9', 'LED', 'MISO', 'MOSI', 'NEOPIXEL', 'RX',
 'SCK', 'SCL', 'SDA', 'SPI', 'TX', 'UART']
```

Note that most of the pins are named in an IO# style, such as **IO1** and **IO2**. Those pins on the physical board are labeled only with a number, so an easy way to know how to access them in CircuitPython, is to run those commands in the REPL and find the pin naming scheme.

If your code is failing to run because it can't find a pin name you provided, verify that you have the proper pin name by running these commands in the REPL.

I2C, SPI, and UART

You'll also see there are often (**but not always!**) three special board-specific objects included: `I2C`, `SPI`, and `UART` - each one is for the default pin-set used for each of the three common protocol busses they are named for. These are called singletons.

What's a singleton? When you create an object in CircuitPython, you are instantiating ('creating') it. Instantiating an object means you are creating an instance of the object with the unique values that are provided, or "passed", to it.

For example, When you instantiate an I2C object using the `busio` module, it expects two pins: clock and data, typically SCL and SDA. It often looks like this:

```
i2c = busio.I2C(board.SCL, board.SDA)
```

Then, you pass the I2C object to a driver for the hardware you're using. For example, if you were using the TSL2591 light sensor and its CircuitPython library, the next line of code would be:

```
tsl2591 = adafruit_tsl2591.TSL2591(i2c)
```

However, CircuitPython makes this simpler by including the `I2C` singleton in the `board` module. Instead of the two lines of code above, you simply provide the singleton as the I2C object. So if you were using the TSL2591 and its CircuitPython library, the two above lines of code would be replaced with:

```
tsl2591 = adafruit_tsl2591.TSL2591(board.I2C())
```

The board.I2C(), board.SPI(), and board.UART() singletons do not exist on all boards. They exist if there are board markings for the default pins for those devices.

This eliminates the need for the `busio` module, and simplifies the code. Behind the scenes, the `board.I2C()` object is instantiated when you call it, but not before, and on subsequent calls, it returns the same object. Basically, it does not create an object until you need it, and provides the same object every time you need it. You can call `board.I2C()` as many times as you like, and it will always return the same object.

The UART/SPI/I2C singletons will use the 'default' bus pins for each board - often labeled as RX/TX (UART), MOSI/MISO/SCK (SPI), or SDA/SCL (I2C). Check your board documentation/pinout for the default busses.

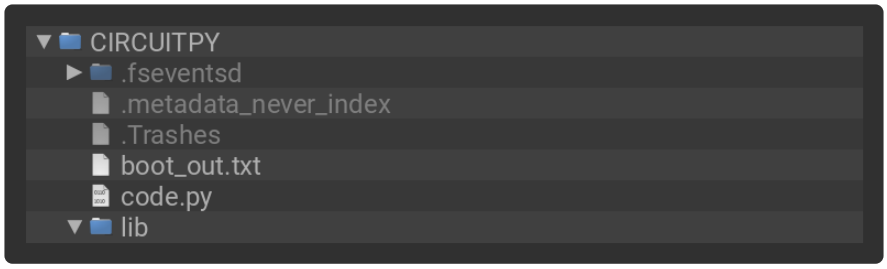
What Are All the Available Names?

Many pins on CircuitPython compatible microcontroller boards have multiple names, however, typically, there's only one name labeled on the physical board. So how do you find out what the other available pin names are? Simple, with the following script! Each line printed out to the serial console contains the set of names for a particular pin.

On a microcontroller board running CircuitPython, first, connect to the serial console.

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Essentials/Pin_Map_Script/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:




```
# SPDX-FileCopyrightText: 2020 anecdata for Adafruit Industries
# SPDX-FileCopyrightText: 2021 Neradoc for Adafruit Industries
# SPDX-FileCopyrightText: 2021-2023 Kattni Rembor for Adafruit Industries
```

```
# SPDX-FileCopyrightText: 2023 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Pin Map Script"""
import microcontroller
import board
try:
    import cyw43  # raspberrypi
except ImportError:
    cyw43 = None

board_pins = []
for pin in dir(microcontroller.pin):
    if (isinstance(getattr(microcontroller.pin, pin), microcontroller.Pin) or
        (cyw43 and isinstance(getattr(microcontroller.pin, pin), cyw43.CywPin))):
        pins = []
        for alias in dir(board):
            if getattr(board, alias) is getattr(microcontroller.pin, pin):
                pins.append(f"board.{alias}")
        # Add the original GPIO name, in parentheses.
        if pins:
            # Only include pins that are in board.
            pins.append(f"({str(pin)})")
            board_pins.append(" ".join(pins))

for pins in sorted(board_pins):
    print(pins)
```

Here is the result when this script is run on QT Py SAMD21:

```
code.py output:
board.A0 board.D0 (PA02)
board.A1 board.D1 (PA03)
board.A10 board.D10 board.MOSI (PA10)
board.A2 board.D2 (PA04)
board.A3 board.D3 (PA05)
board.A6 board.D6 board.TX (PA06)
board.A7 board.D7 board.RX (PA07)
board.A8 board.D8 board.SCK (PA11)
board.A9 board.D9 board.MISO (PA09)
board.D4 board.SDA (PA16)
board.D5 board.SCL (PA17)
board.NEOPIXEL (PA18)
board.NEOPIXEL_POWER (PA15)
```

Each line represents a single pin. Find the line containing the pin name that's labeled on the physical board, and you'll find the other names available for that pin. For example, the first pin on the board is labeled **A0**. The first line in the output is `board.A0 board.D0 (PA02)`. This means that you can access pin **A0** in CircuitPython using both `board.A0` and `board.D0`.

The pins in parentheses are the microcontroller pin names. See the next section for more info on those.

You'll notice there are two "pins" that aren't labeled on the board but appear in the list: `board.NEOPIXEL` and `board.NEOPIXEL_POWER`. Many boards have several of these special pins that give you access to built-in board hardware, such as an LED or an on-board sensor. The QT Py SAMD21 only has one on-board extra piece of hardware, a NeoPixel LED, so there's only the one available in the list. But you can

also control whether or not power is applied to the NeoPixel, so there's a separate pin for that.

That's all there is to figuring out the available names for a pin on a compatible microcontroller board in CircuitPython!

Microcontroller Pin Names

The pin names available to you in the CircuitPython `board` module are not the same as the names of the pins on the microcontroller itself. The board pin names are aliases to the microcontroller pin names. If you look at the datasheet for your microcontroller, you'll likely find a pinout with a series of pin names, such as "PA18" or "GPIO5". If you want to get to the actual microcontroller pin name in CircuitPython, you'll need the `microcontroller.pin` module. As with `board`, you can run `dir(microcontroller.pin)` in the REPL to receive a list of the microcontroller pin names.

```
>>> import microcontroller
>>> dir(microcontroller.pin)
['__class__', 'PA02', 'PA03', 'PA04', 'PA05', 'PA06', 'PA07', 'PA08', 'PA09',
'PA10', 'PA11', 'PA15', 'PA16', 'PA17', 'PA18', 'PA19', 'PA22', 'PA23']
```

CircuitPython Built-In Modules

There is a set of modules used in most CircuitPython programs. One or more of these modules is always used in projects involving hardware. Often hardware requires installing a separate library from the Adafruit CircuitPython Bundle. But, if you try to find `board` or `digitalio` in the same bundle, you'll come up lacking. So, where do these modules come from? They're built into CircuitPython! You can find an comprehensive list of built-in CircuitPython modules and the technical details of their functionality from CircuitPython here (https://adafru.it/QkB) and the Python-like modules included here (https://adafru.it/QkC). However, **not every module is available for every board** due to size constraints or hardware limitations. How do you find out what modules are available for your board?

There are two options for this. You can check the support matrix (https://adafru.it/N2a), and search for your board by name. Or, you can use the REPL.

Plug in your board, connect to the serial console and enter the REPL. Type the following command.

```
help("modules")
```

```
>>> help("modules")
__main__          collections       neopixel_write    supervisor
_pixelbuf         digitalio         os                sys
adafruit_bus_device                 displayio         pulseio           terminalio
analogio          errno             pwmio             time
array             fontio            random            touchio
audiocore         gamepad           re                usb_hid
audioio           gc                rotaryio          usb_midi
board             math              rtc               vectorio
builtins          microcontroller   storage
busio             micropython       struct
Plus any modules on the filesystem
```

That's it! You now know two ways to find all of the modules built into CircuitPython for your compatible microcontroller board.

Installing the Mu Editor

Mu is a simple code editor that works with the Adafruit CircuitPython boards. It's written in Python and works on Windows, MacOS, Linux and Raspberry Pi. The serial console is built right in so you get immediate feedback from your board's serial output!

Mu is our recommended editor - please use it (unless you are an experienced coder with a favorite editor already!).

Download and Install Mu



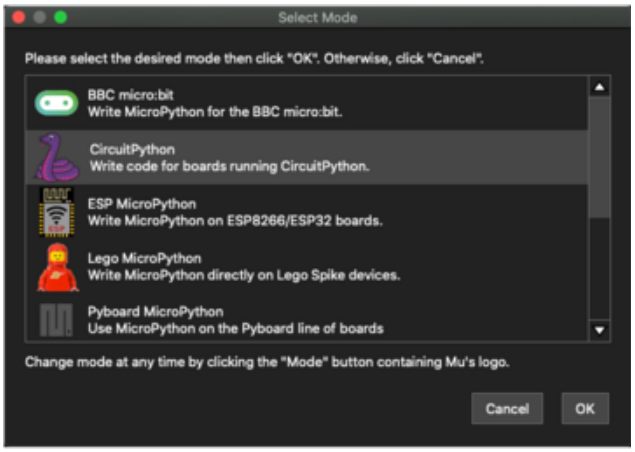
Download Mu from https://codewith.mu (https://adafru.it/Be6).

Click the **Download** link for downloads and installation instructions.

Click **Start Here** to find a wealth of other information, including extensive tutorials and and how-to's.

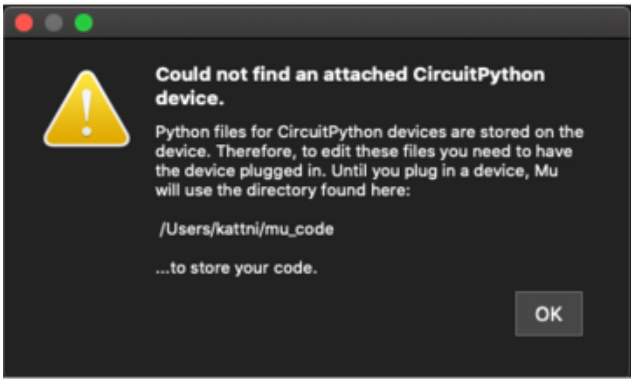
Windows users: due to the nature of MSI installers, please remove old versions of Mu before installing the latest version.

Starting Up Mu



The first time you start Mu, you will be prompted to select your 'mode' - you can always change your mind later. For now please select **CircuitPython**!

The current mode is displayed in the lower right corner of the window, next to the "gear" icon. If the mode says "Microbit" or something else, click the **Mode** button in the upper left, and then choose "CircuitPython" in the dialog box that appears.



Mu attempts to auto-detect your board on startup, so if you do not have a CircuitPython board plugged in with a **CIRCUITPY** drive available, Mu will inform you where it will store any code you save until you plug in a board.

To avoid this warning, plug in a board and ensure that the **CIRCUITPY** drive is mounted before starting Mu.

Using Mu

You can now explore Mu! The three main sections of the window are labeled below; the button bar, the text editor, and the serial console / REPL.



Now you're ready to code! Let's keep going...

Creating and Editing Code

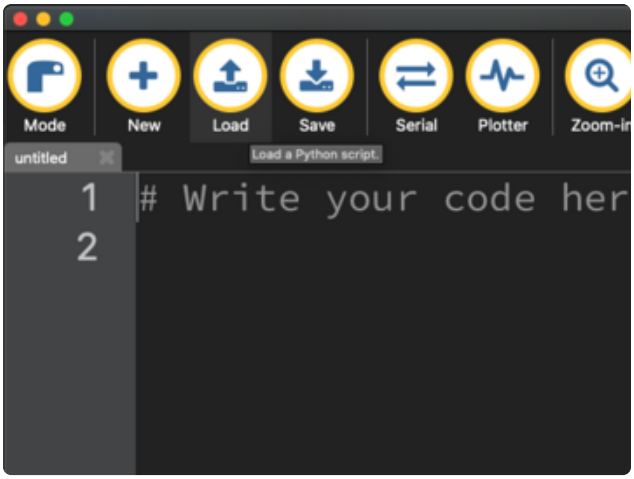
One of the best things about CircuitPython is how simple it is to get code up and running. This section covers how to create and edit your first CircuitPython program.

To create and edit code, all you'll need is an editor. There are many options. **Adafruit strongly recommends using Mu! It's designed for CircuitPython, and it's really simple and easy to use, with a built in serial console!**

If you don't or can't use Mu, there are a number of other editors that work quite well. The Recommended Editors page (https://adafru.it/Vue) has more details. Otherwise, make sure you do "Eject" or "Safe Remove" on Windows or "sync" on Linux after writing a file if you aren't using Mu. (This was formerly not a problem on macOS, but see the warning below.)

macOS Sonoma (14.x) introduced a bug that delays writes to small drives such as CIRCUITPY drives. This causes errors when saving files to CIRCUITPY. For a workaround, see https://learn.adafruit.com/welcome-to-circuitpython/troubleshooting#macos-sonoma-14-dot-x-disk-errors-writing-to-circuitpy-3160304

Creating Code



Installing CircuitPython generates a **code.py** file on your **CIRCUITPY** drive. To begin your own program, open your editor, and load the **code.py** file from the **CIRCUITPY** drive.

If you are using Mu, click the **Load** button in the button bar, navigate to the **CIRCUITPY** drive, and choose **code.py**.

Copy and paste the following code into your editor:

```
import board
import digitalio
import time

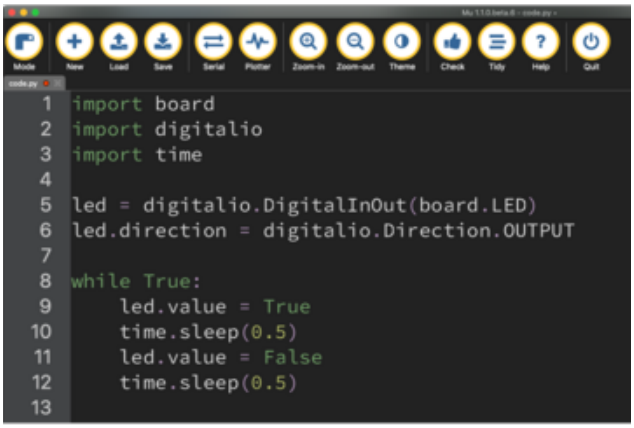
led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

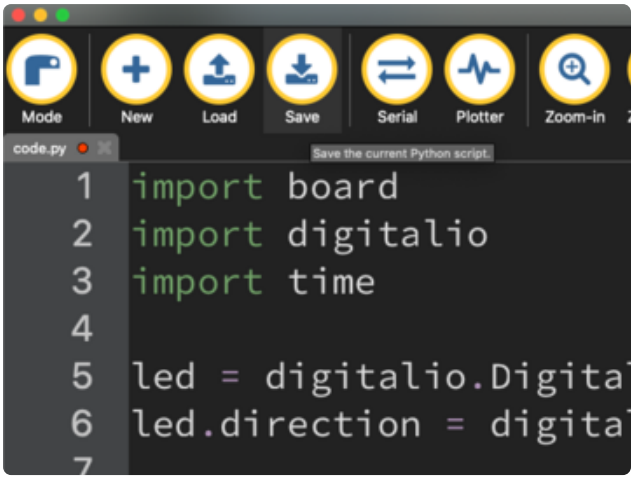
The KB2040, QT Py and the Trinkeys do not have a built-in little red LED! There is an addressable RGB NeoPixel LED. The above example will NOT work on the KB2040, QT Py or the Trinkeys!

If you're using a KB2040, QT Py or a Trinkey, please download the NeoPixel blink example (https://adafru.it/UDU).

The NeoPixel blink example uses the onboard NeoPixel, but the time code is the same. You can use the linked NeoPixel Blink example to follow along with this guide page.



It will look like this. Note that under the `while True:` line, the next four lines begin with four spaces to indent them, and they're indented exactly the same amount. All the lines before that have no spaces before the text.



Save the **code.py** file on your **CIRCUITPY** drive.

The little LED should now be blinking. Once per half-second.

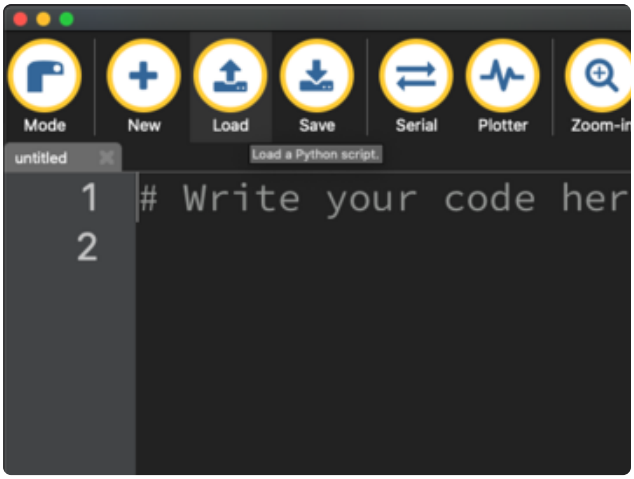
Congratulations, you've just run your first CircuitPython program!

On most boards you'll find a tiny red LED.

On the ItsyBitsy nRF52840, you'll find a tiny blue LED.

On QT Py M0, QT Py RP2040, and the Trinkey series, you will find only an RGB NeoPixel LED.

Editing Code



To edit code, open the **code.py** file on your **CIRCUITPY** drive into your editor.

Make the desired changes to your code. Save the file. That's it!

Your code changes are run as soon as the file is done saving.

There's one warning before you continue...

Don't click reset or unplug your board!

The CircuitPython code on your board detects when the files are changed or written and will automatically re-start your code. This makes coding very fast because you save, and it re-runs. If you unplug or reset the board before your computer finishes writing the file to your board, you can corrupt the drive. If this happens, you may lose the code you've written, so it's important to backup your code to your computer regularly.

There are a couple of ways to avoid filesystem corruption.

1. Use an editor that writes out the file completely when you save it.

Check out the Recommended Editors page (https://adafru.it/Vue) for details on different editing options.

If you are dragging a file from your host computer onto the CIRCUITPY drive, you still need to do step 2. Eject or Sync (below) to make sure the file is completely written.

2. Eject or Sync the Drive After Writing

If you are using one of our not-recommended-editors, not all is lost! You can still make it work.

On Windows, you can Eject or Safe Remove the **CIRCUITPY** drive. It won't actually eject, but it will force the operating system to save your file to disk. On Linux, use the **sync** command in a terminal to force the write to disk.

You also need to do this if you use Windows Explorer or a Linux graphical file manager to drag a file onto **CIRCUITPY**.

Oh No I Did Something Wrong and Now The CIRCUITPY Drive Doesn't Show Up!!!

> Don't worry! Corrupting the drive isn't the end of the world (or your board!). If this happens, follow the steps found on the Troubleshooting (https://adafru.it/Den) page of every board guide to get your board up and running again.

Back to Editing Code...

Now! Let's try editing the program you added to your board. Open your **code.py** file into your editor. You'll make a simple change. Change the first `0.5` to `0.1`. The code should look like this:

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.1)
    led.value = False
    time.sleep(0.5)
```

Leave the rest of the code as-is. Save your file. See what happens to the LED on your board? Something changed! Do you know why?

You don't have to stop there! Let's keep going. Change the second `0.5` to `0.1` so it looks like this:

```
while True:
    led.value = True
```

```
    time.sleep(0.1)
    led.value = False
    time.sleep(0.1)
```

Now it blinks really fast! You decreased the both time that the code leaves the LED on and off!

Now try increasing both of the `0.1` to `1`. Your LED will blink much more slowly because you've increased the amount of time that the LED is turned on and off.

Well done! You're doing great! You're ready to start into new examples and edit them to see what happens! These were simple changes, but major changes are done using the same process. Make your desired change, save it, and get the results. That's really all there is to it!

Naming Your Program File

CircuitPython looks for a code file on the board to run. There are four options: **code.txt**, **code.py**, **main.txt** and **main.py**. CircuitPython looks for those files, in that order, and then runs the first one it finds. While **code.py** is the recommended name for your code file, it is important to know that the other options exist. If your program doesn't seem to be updating as you work, make sure you haven't created another code file that's being read instead of the one you're working on.

Exploring Your First CircuitPython Program

First, you'll take a look at the code you're editing.

Here is the original code again:

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

Imports & Libraries

Each CircuitPython program you run needs to have a lot of information to work. The reason CircuitPython is so simple to use is that most of that information is stored in other files and works in the background. The files built into CircuitPython are called **modules**, and the files you load separately are called **libraries**. Modules are built into CircuitPython. Libraries are stored on your **CIRCUITPY** drive in a folder called **lib**.

```
import board
import digitalio
import time
```

The `import` statements tells the board that you're going to use a particular library or module in your code. In this example, you imported three modules: `board`, `digitalio`, and `time`. All three of these modules are built into CircuitPython, so no separate library files are needed. That's one of the things that makes this an excellent first example. You don't need anything extra to make it work!

These three modules each have a purpose. The first one, `board`, gives you access to the hardware on your board. The second, `digitalio`, lets you access that hardware as inputs/outputs. The third, `time`, let's you control the flow of your code in multiple ways, including passing time by 'sleeping'.

Setting Up The LED

The next two lines setup the code to use the LED.

```
led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT
```

Your board knows the red LED as `LED`. So, you initialise that pin, and you set it to output. You set `led` to equal the rest of that information so you don't have to type it all out again later in our code.

Loop-de-loops

The third section starts with a `while` statement. `while True:` essentially means, "forever do the following:". `while True:` creates a loop. Code will loop "while" the condition is "true" (vs. false), and as `True` is never False, the code will loop forever. All code that is indented under `while True:` is "inside" the loop.

Inside our loop, you have four items:

```
while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

First, you have `led.value = True`. This line tells the LED to turn on. On the next line, you have `time.sleep(0.5)`. This line is telling CircuitPython to pause running code for 0.5 seconds. Since this is between turning the led on and off, the led will be on for 0.5 seconds.

The next two lines are similar. `led.value = False` tells the LED to turn off, and `time.sleep(0.5)` tells CircuitPython to pause for another 0.5 seconds. This occurs between turning the led off and back on so the LED will be off for 0.5 seconds too.

Then the loop will begin again, and continue to do so as long as the code is running!

So, when you changed the first `0.5` to `0.1`, you decreased the amount of time that the code leaves the LED on. So it blinks on really quickly before turning off!

Great job! You've edited code in a CircuitPython program!

What Happens When My Code Finishes Running?

When your code finishes running, CircuitPython resets your microcontroller board to prepare it for the next run of code. That means any set up you did earlier no longer applies, and the pin states are reset.

For example, try reducing the code snippet above by eliminating the loop entirely, and replacing it with `led.value = True`. The LED will flash almost too quickly to see, and turn off. This is because the code finishes running and resets the pin state, and the LED is no longer receiving a signal.

To that end, most CircuitPython programs involve some kind of loop, infinite or otherwise.

What if I Don't Have the Loop?

If you don't have the loop, the code will run to the end and exit. This can lead to some unexpected behavior in simple programs like this since the "exit" also resets the state of the hardware. This is a different behavior than running commands via REPL. So if

you are writing a simple program that doesn't seem to work, you may need to add a loop to the end so the program doesn't exit.

The simplest loop would be:

```
while True:
    pass
```

And remember - you can press CTRL+C to exit the loop.

See also the Behavior section in the docs (https://adafru.it/Bvz).

Connecting to the Serial Console

One of the staples of CircuitPython (and programming in general!) is something called a "print statement". This is a line you include in your code that causes your code to output text. A print statement in CircuitPython (and Python) looks like this:

`print("Hello, world!")`

This line in your code.py would result in:

`Hello, world!`

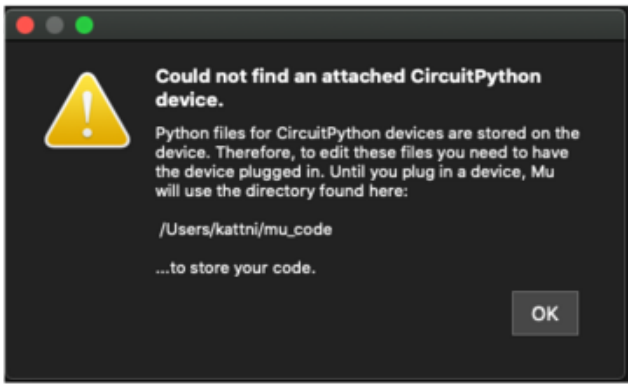
However, these print statements need somewhere to display. That's where the serial console comes in!

The serial console receives output from your CircuitPython board sent over USB and displays it so you can see it. This is necessary when you've included a print statement in your code and you'd like to see what you printed. It is also helpful for troubleshooting errors, because your board will send errors and the serial console will display those too.

The serial console requires an editor that has a built in terminal, or a separate terminal program. A terminal is a program that gives you a text-based interface to perform various tasks.

Are you using Mu?

If so, good news! The serial console **is built into Mu** and will **autodetect your board** making using the serial console really really easy.



First, make sure your CircuitPython board is plugged in.

If you open Mu without a board plugged in, you may encounter the error seen here, letting you know no CircuitPython board was found and indicating where your code will be stored until you plug in a board.

If you are using Windows 7, make sure you installed the drivers (https://adafru.it/VuB).

Once you've opened Mu with your board plugged in, look for the **Serial** button in the button bar and click it.



The Mu window will split in two, horizontally, and display the serial console at the bottom.



If nothing appears in the serial console, it may mean your code is done running or has no print statements in it. Click into the serial console part of Mu, and press CTRL+D to reload.

Serial Console Issues or Delays on Linux

If you're on Linux, and are seeing multi-second delays connecting to the serial console, or are seeing "AT" and other gibberish when you connect, then the `modemmanager` service might be interfering. Just remove it; it doesn't have much use unless you're still using dial-up modems.

To remove `modemmanager`, type the following command at a shell:

```
sudo apt purge modemmanager
```

Setting Permissions on Linux

On Linux, if you see an error box something like the one below when you press the **Serial** button, you need to add yourself to a user group to have permission to connect to the serial console.



On Ubuntu and Debian, add yourself to the **dialout** group by doing:

```
sudo adduser $USER dialout
```

After running the command above, reboot your machine to gain access to the group. On other Linux distributions, the group you need may be different. See the Advanced Serial Console on Linux (https://adafru.it/VAO) for details on how to add yourself to the right group.

Using Something Else?

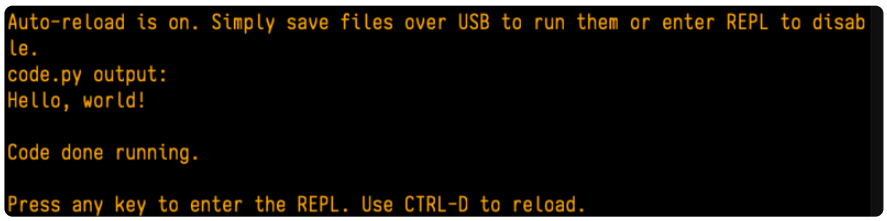
If you're not using Mu to edit, are using or if for some reason you are not a fan of its built in serial console, you can run the serial console from a separate program.

Windows requires you to download a terminal program. Check out the Advanced Serial Console on Windows page for more details. (https://adafru.it/AAH)

MacOS has Terminal built in, though there are other options available for download. Check the Advanced Serial Console on Mac page for more details. (https://adafru.it/AAI)

Linux has a terminal program built in, though other options are available for download. Check the Advanced Serial Console on Linux page for more details. (https://adafru.it/VAO)

Once connected, you'll see something like the following.



Interacting with the Serial Console

Once you've successfully connected to the serial console, it's time to start using it.

The code you wrote earlier has no output to the serial console. So, you're going to edit it to create some output.

Open your code.py file into your editor, and include a `print` statement. You can print anything you like! Just include your phrase between the quotation marks inside the parentheses. For example:

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    print("Hello, CircuitPython!")
    led.value = True
    time.sleep(1)
    led.value = False
    time.sleep(1)
```

Save your file.

Now, let's go take a look at the window with our connection to the serial console.



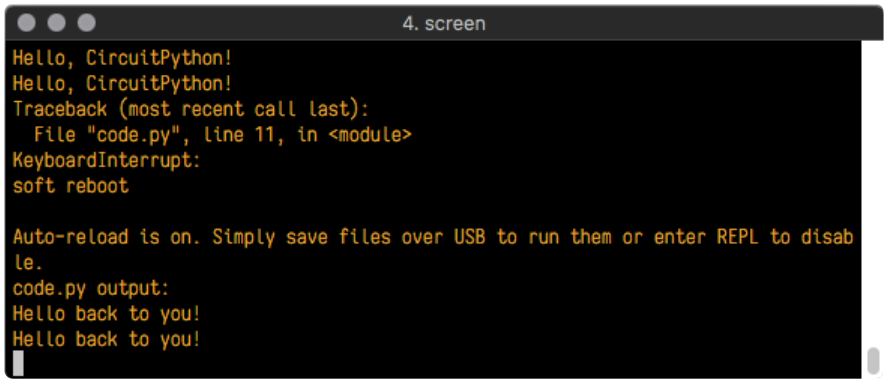
Excellent! Our print statement is showing up in our console! Try changing the printed text to something else.

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    print("Hello back to you!")
    led.value = True
    time.sleep(1)
    led.value = False
    time.sleep(1)
```

Keep your serial console window where you can see it. Save your file. You'll see what the serial console displays when the board reboots. Then you'll see your new change!



The `Traceback (most recent call last):` is telling you the last thing your board was doing before you saved your file. This is normal behavior and will happen every time the board resets. This is really handy for troubleshooting. Let's introduce an error so you can see how it is used.

Delete the `e` at the end of `True` from the line `led.value = True` so that it says `led.value = Tru`

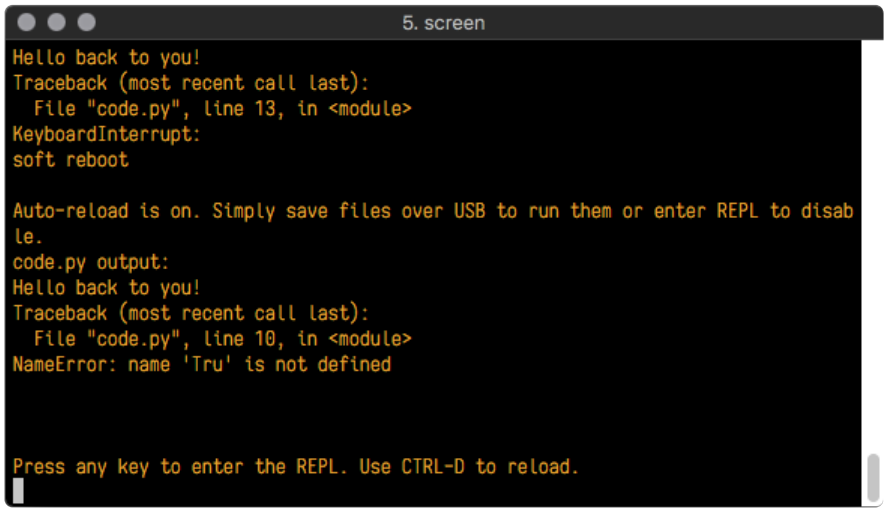
```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    print("Hello back to you!")
    led.value = Tru
    time.sleep(1)
    led.value = False
    time.sleep(1)
```

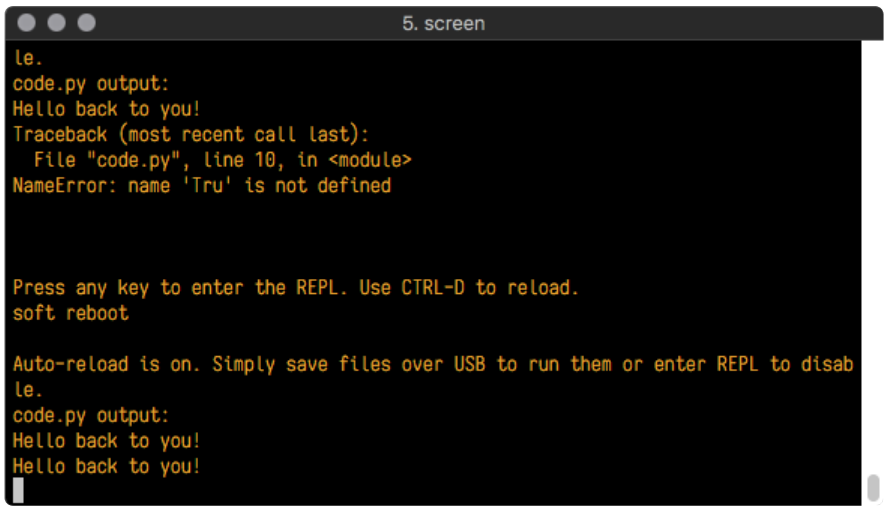
Save your file. You will notice that your red LED will stop blinking, and you may have a colored status LED blinking at you. This is because the code is no longer correct and can no longer run properly. You need to fix it!

Usually when you run into errors, it's not because you introduced them on purpose. You may have 200 lines of code, and have no idea where your error could be hiding. This is where the serial console can help. Let's take a look!



The `Traceback (most recent call last):` is telling you that the last thing it was able to run was `line 10` in your code. The next line is your error: `NameError: name 'Tru' is not defined`. This error might not mean a lot to you, but combined with knowing the issue is on line 10, it gives you a great place to start!

Go back to your code, and take a look at line 10. Obviously, you know what the problem is already. But if you didn't, you'd want to look at line 10 and see if you could figure it out. If you're still unsure, try googling the error to get some help. In this case, you know what to look for. You spelled True wrong. Fix the typo and save your file.



Nice job fixing the error! Your serial console is streaming and your red LED Is blinking again.

The serial console will display any output generated by your code. Some sensors, such as a humidity sensor or a thermistor, receive data and you can use print

statements to display that information. You can also use print statements for troubleshooting, which is called "print debugging". Essentially, if your code isn't working, and you want to know where it's failing, you can put print statements in various places to see where it stops printing.

The serial console has many uses, and is an amazing tool overall for learning and programming!

The REPL

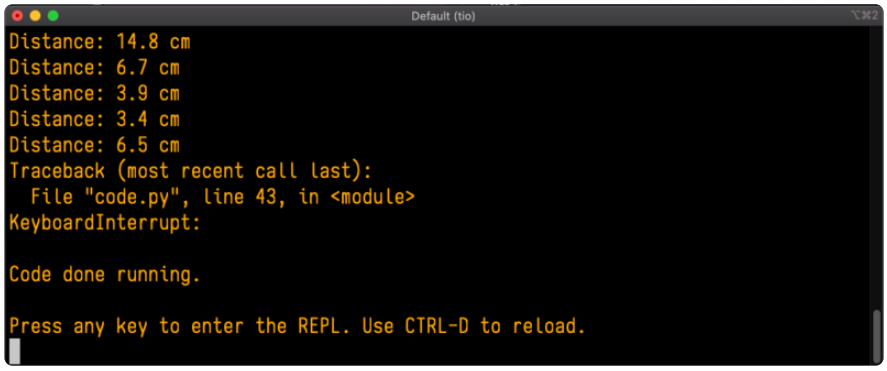
The other feature of the serial connection is the **R**ead-**E**valuate-**P**rint-**L**oop, or REPL. The REPL allows you to enter individual lines of code and have them run immediately. It's really handy if you're running into trouble with a particular program and can't figure out why. It's interactive so it's great for testing new ideas.

Entering the REPL

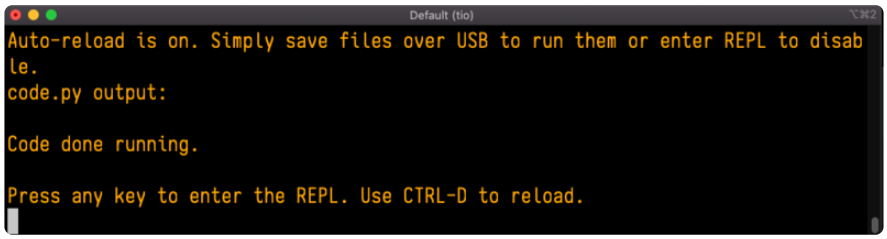
To use the REPL, you first need to be connected to the serial console. Once that connection has been established, you'll want to press **CTRL+C**.

If there is code running, in this case code measuring distance, it will stop and you'll see `Press any key to enter the REPL. Use CTRL-D to reload.` Follow those instructions, and press any key on your keyboard.

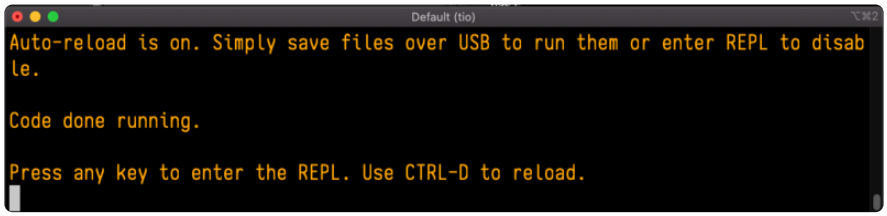
The `Traceback (most recent call last):` is telling you the last thing your board was doing before you pressed Ctrl + C and interrupted it. The `KeyboardInterrupt` is you pressing CTRL+C. This information can be handy when troubleshooting, but for now, don't worry about it. Just note that it is expected behavior.



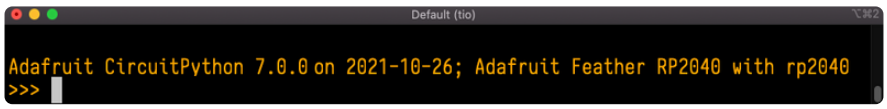
If your **code.py** file is empty or does not contain a loop, it will show an empty output and `Code done running.`. There is no information about what your board was doing before you interrupted it because there is no code running.



If you have no **code.py** on your **CIRCUITPY** drive, you will enter the REPL immediately after pressing CTRL+C. Again, there is no information about what your board was doing before you interrupted it because there is no code running.



Regardless, once you press a key you'll see a `>>>` prompt welcoming you to the REPL!



If you have trouble getting to the `>>>` prompt, try pressing Ctrl + C a few more times.

The first thing you get from the REPL is information about your board.

```
Adafruit CircuitPython 7.0.0 on 2021-10-26; Adafruit Feather RP2040 with rp2040
```

This line tells you the version of CircuitPython you're using and when it was released. Next, it gives you the type of board you're using and the type of microcontroller the board uses. Each part of this may be different for your board depending on the versions you're working with.

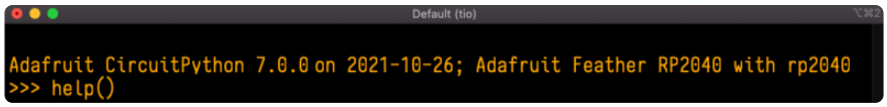
This is followed by the CircuitPython prompt.

```
>>>
```

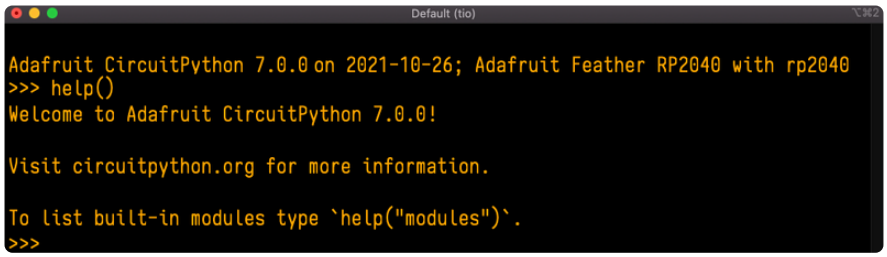
Interacting with the REPL

From this prompt you can run all sorts of commands and code. The first thing you'll do is run `help()`. This will tell you where to start exploring the REPL. To run code in the REPL, type it in next to the REPL prompt.

Type `help()` next to the prompt in the REPL.



Then press enter. You should then see a message.



First part of the message is another reference to the version of CircuitPython you're using. Second, a URL for the CircuitPython related project guides. Then... wait. What's this? `To list built-in modules type `help("modules")`.` Remember the modules you learned about while going through creating code? That's exactly what this is talking about! This is a perfect place to start. Let's take a look!

Type `help("modules")` into the REPL next to the prompt, and press enter.



This is a list of all the core modules built into CircuitPython, including `board`. Remember, `board` contains all of the pins on the board that you can use in your code. From the REPL, you are able to see that list!

Type `import board` into the REPL and press enter. It'll go to a new prompt. It might look like nothing happened, but that's not the case! If you recall, the `import` statement simply tells the code to expect to do something with that module. In this case, it's telling the REPL that you plan to do something with that module.



Next, type `dir(board)` into the REPL and press enter.



This is a list of all of the pins on your board that are available for you to use in your code. Each board's list will differ slightly depending on the number of pins available. Do you see `LED`? That's the pin you used to blink the red LED!

The REPL can also be used to run code. Be aware that **any code you enter into the REPL isn't saved** anywhere. If you're testing something new that you'd like to keep, make sure you have it saved somewhere on your computer as well!

Every programmer in every programming language starts with a piece of code that says, "Hello, World." You're going to say hello to something else. Type into the REPL:

```
print("Hello, CircuitPython!")
```

Then press enter.



That's all there is to running code in the REPL! Nice job!

You can write single lines of code that run stand-alone. You can also write entire programs into the REPL to test them. Remember that nothing typed into the REPL is saved.

There's a lot the REPL can do for you. It's great for testing new ideas if you want to see if a few new lines of code will work. It's fantastic for troubleshooting code by entering it one line at a time and finding out where it fails. It lets you see what modules are available and explore those modules.

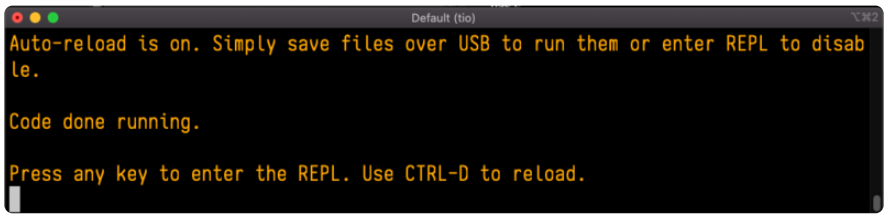
Try typing more into the REPL to see what happens!

Everything typed into the REPL is ephemeral. Once you reload the REPL or return to the serial console, nothing you typed will be retained in any memory space. So be sure to save any desired code you wrote somewhere else, or you'll lose it when you leave the current REPL instance!

Returning to the Serial Console

When you're ready to leave the REPL and return to the serial console, simply press **CTRL+D**. This will reload your board and reenter the serial console. You will restart the program you had running before entering the REPL. In the console window, you'll see any output from the program you had running. And if your program was affecting anything visual on the board, you'll see that start up again as well.

You can return to the REPL at any time!

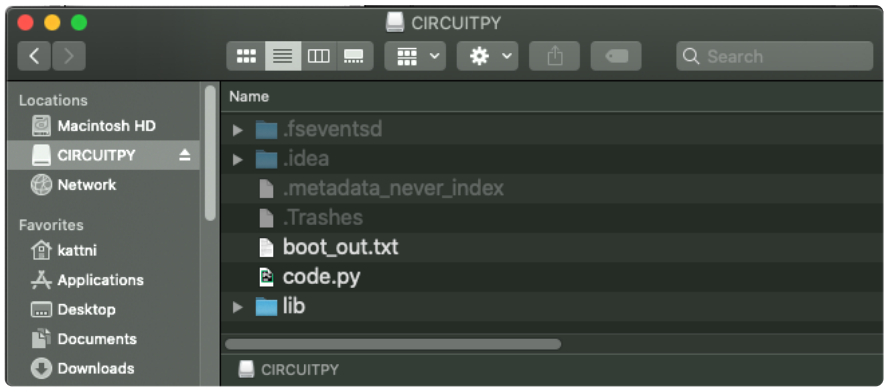


CircuitPython Libraries

As CircuitPython development continues and there are new releases, Adafruit will stop supporting older releases. Visit https://circuitpython.org/downloads to download the latest version of CircuitPython for your board. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update CircuitPython and then visit https://circuitpython.org/libraries to download the latest Library Bundle.

Each CircuitPython program you run needs to have a lot of information to work. The reason CircuitPython is so simple to use is that most of that information is stored in other files and works in the background. These files are called libraries. Some of them are built into CircuitPython. Others are stored on your **CIRCUITPY** drive in a folder called **lib**. Part of what makes CircuitPython so great is its ability to store code separately from the firmware itself. Storing code separately from the firmware makes it easier to update both the code you write and the libraries you depend.

Your board may ship with a **lib** folder already, it's in the base directory of the drive. If not, simply create the folder yourself. When you first install CircuitPython, an empty **lib** directory will be created for you.



CircuitPython libraries work in the same way as regular Python modules so the Python docs (https://adafru.it/rar) are an excellent reference for how it all should work. In Python terms, you can place our library files in the **lib** directory because it's part of the Python path by default.

One downside of this approach of separate libraries is that they are not built in. To use them, one needs to copy them to the **CIRCUITPY** drive before they can be used. Fortunately, there is a library bundle.

The bundle and the library releases on GitHub also feature optimized versions of the libraries with the **.mpy** file extension. These files take less space on the drive and have a smaller memory footprint as they are loaded.

Due to the regular updates and space constraints, Adafruit does not ship boards with the entire bundle. Therefore, you will need to load the libraries you need when you begin working with your board. You can find example code in the guides for your board that depends on external libraries.

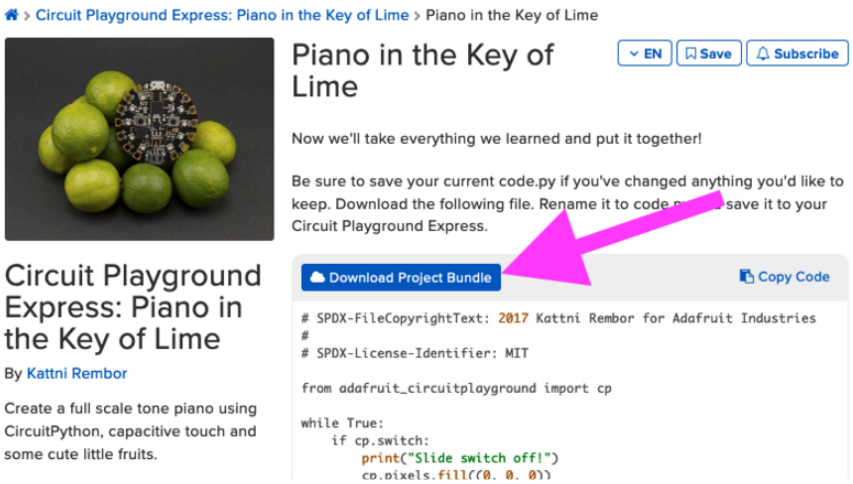
Either way, as you start to explore CircuitPython, you'll want to know how to get libraries on board.

The Adafruit Learn Guide Project Bundle

The quickest and easiest way to get going with a project from the Adafruit Learn System is by utilising the Project Bundle. Most guides now have a **Download Project Bundle** button available at the top of the full code example embed. This button downloads all the necessary files, including images, etc., to get the guide project up and running. Simply click, open the resulting zip, copy over the right files, and you're good to go!

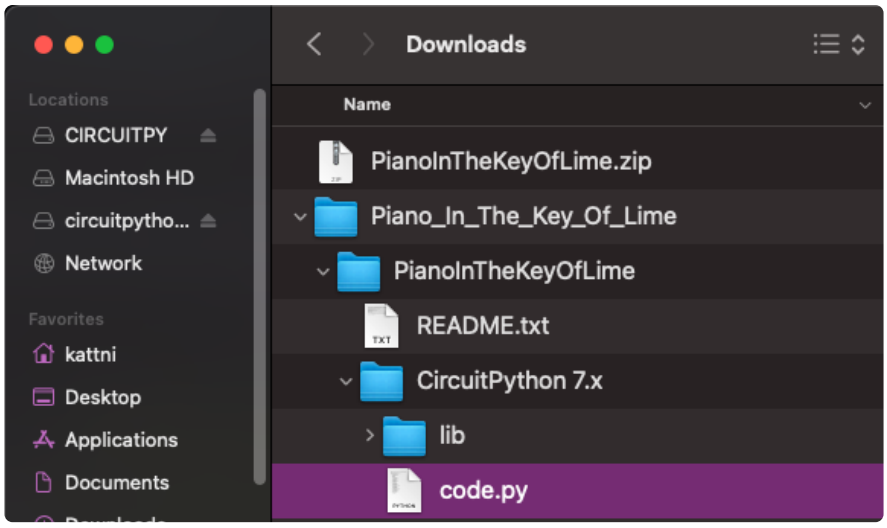
The first step is to find the Download Project Bundle button in the guide you're working on.

The Download Project Bundle button is only available on full demo code embedded from GitHub in a Learn guide. Code snippets will NOT have the button available.



When you copy the contents of the Project Bundle to your CIRCUITPY drive, it will replace all the existing content! If you don't want to lose anything, ensure you copy your current code to your computer before you copy over the new Project Bundle content!

The Download Project Bundle button downloads a zip file. This zip contains a series of directories, nested within which is the **code.py**, any applicable assets like images or audio, and the **lib/** folder containing all the necessary libraries. The following zip was downloaded from the Piano in the Key of Lime guide.



The Piano in the Key of Lime guide was chosen as an example. That guide is specific to Circuit Playground Express, and cannot be used on all boards. Do not

expect to download that exact bundle and have it work on your non-CPX microcontroller.

When you open the zip, you'll find some nested directories. Navigate through them until you find what you need. You'll eventually find a directory for your CircuitPython version (in this case, 7.x). In the version directory, you'll find the file and directory you need: **code.py** and **lib/**. Once you find the content you need, you can copy it all over to your **CIRCUITPY** drive, replacing any files already on the drive with the files from the freshly downloaded zip.

In some cases, there will be other files such as audio or images in the same directory as code.py and lib/. Make sure you include all the files when you copy things over!

Once you copy over all the relevant files, the project should begin running! If you find that the project is not running as expected, make sure you've copied ALL of the project files onto your microcontroller board.

That's all there is to using the Project Bundle!

The Adafruit CircuitPython Library Bundle

Adafruit provides CircuitPython libraries for much of the hardware they provide, including sensors, breakouts and more. To eliminate the need for searching for each library individually, the libraries are available together in the Adafruit CircuitPython Library Bundle. The bundle contains all the files needed to use each library.

Downloading the Adafruit CircuitPython Library Bundle

You can download the latest Adafruit CircuitPython Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

Match up the bundle version with the version of CircuitPython you are running. For example, you would download the 6.x library bundle if you're running any version of CircuitPython 6, or the 7.x library bundle if you're running any version of CircuitPython 7, etc. If you mix libraries with major CircuitPython versions, you will get incompatible mpy errors due to changes in library interfaces possible during major version changes.

Click to visit circuitpython.org for the latest Adafruit CircuitPython Library Bundle

https://adafru.it/ENC

Download the bundle version that matches your CircuitPython firmware version. If you don't know the version, check the version info in **boot_out.txt** file on the **CIRCUITPY** drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.x library bundle.

There's also a **py** bundle which contains the uncompressed python files, you probably don't want that unless you are doing advanced work on libraries.

The CircuitPython Community Library Bundle

The CircuitPython Community Library Bundle is made up of libraries written and provided by members of the CircuitPython community. These libraries are often written when community members encountered hardware not supported in the Adafruit Bundle, or to support a personal project. The authors all chose to submit these libraries to the Community Bundle make them available to the community.

These libraries are maintained by their authors and are not supported by Adafruit. As you would with any library, if you run into problems, feel free to file an issue on the GitHub repo for the library. Bear in mind, though, that most of these libraries are supported by a single person and you should be patient about receiving a response. Remember, these folks are not paid by Adafruit, and are volunteering their personal time when possible to provide support.

Downloading the CircuitPython Community Library Bundle

You can download the latest CircuitPython Community Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

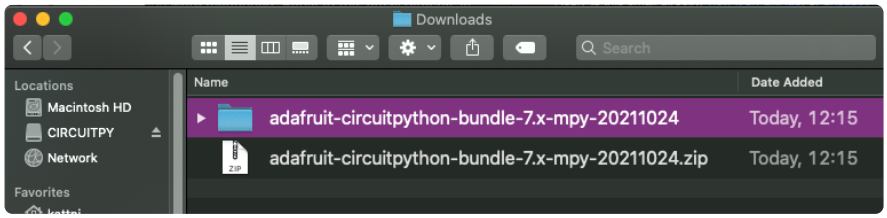
Click for the latest CircuitPython Community Library Bundle release

https://adafru.it/VCn

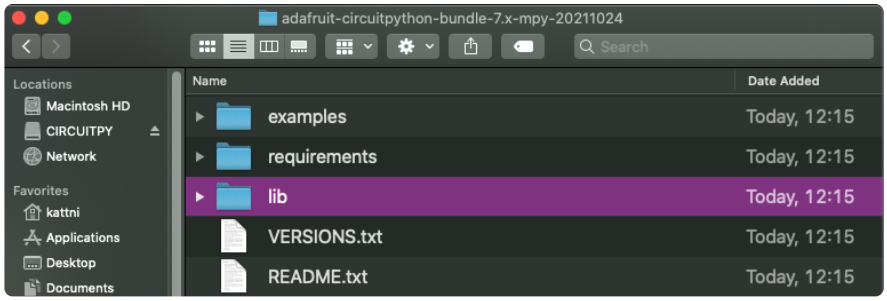
The link takes you to the latest release of the CircuitPython Community Library Bundle on GitHub. There are multiple versions of the bundle available. **Download the bundle version that matches your CircuitPython firmware version.** If you don't know the version, check the version info in **boot_out.txt** file on the **CIRCUITPY** drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.x library bundle.

Understanding the Bundle

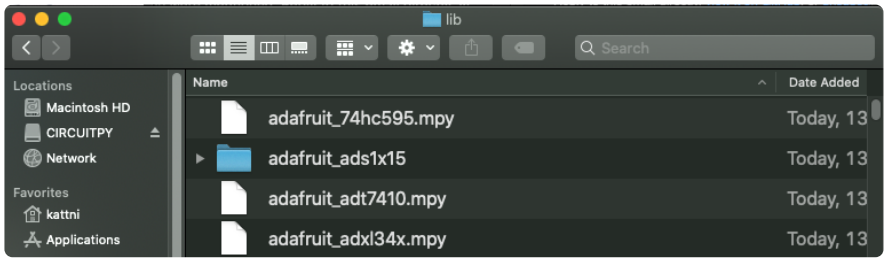
After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.



Open the bundle folder. Inside you'll find two information files, and two folders. One folder is the lib bundle, and the other folder is the examples bundle.



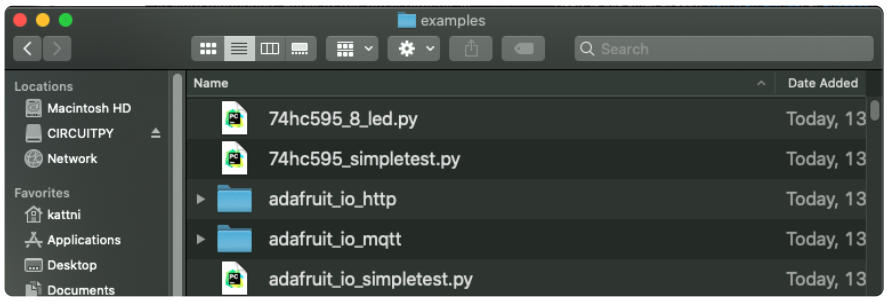
Now open the lib folder. When you open the folder, you'll see a large number of **.mpy** files, and folders.



Example Files

All example files from each library are now included in the bundles in an **examples** directory (as seen above), as well as an examples-only bundle. These are included for two main reasons:

- Allow for quick testing of devices.
- Provide an example base of code, that is easily built upon for individualized purposes.



Copying Libraries to Your Board

First open the **lib** folder on your **CIRCUITPY** drive. Then, open the **lib** folder you extracted from the downloaded zip. Inside you'll find a number of folders and **.mpy** files. Find the library you'd like to use, and copy it to the **lib** folder on **CIRCUITPY**.

If the library is a directory with multiple **.mpy** files in it, be sure to **copy the entire folder to CIRCUITPY/lib**.

This also applies to example files. Open the **examples** folder you extracted from the downloaded zip, and copy the applicable file to your **CIRCUITPY** drive. Then, rename it to **code.py** to run it.

If a library has multiple .mpy files contained in a folder, be sure to copy the entire folder to CIRCUITPY/lib.

Understanding Which Libraries to Install

You now know how to load libraries on to your CircuitPython-compatible microcontroller board. You may now be wondering, how do you know which libraries you need to install? Unfortunately, it's not always straightforward. Fortunately, there is

an obvious place to start, and a relatively simple way to figure out the rest. First up: the best place to start.

When you look at most CircuitPython examples, you'll see they begin with one or more `import` statements. These typically look like the following:

- `import library_or_module`

However, `import` statements can also sometimes look like the following:

- `from library_or_module import name`
- `from library_or_module.subpackage import name`
- `from library_or_module import name as local_name`

They can also have more complicated formats, such as including a `try` / `except` block, etc.

The important thing to know is that **an `import` statement will always include the name of the module or library that you're importing**.

Therefore, the best place to start is by reading through the `import` statements.

Here is an example import list for you to work with in this section. There is no setup or other code shown here, as the purpose of this section involves only the import list.

```
import time
import board
import neopixel
import adafruit_lis3dh
import usb_hid
from adafruit_hid.consumer_control import ConsumerControl
from adafruit_hid.consumer_control_code import ConsumerControlCode
```

Keep in mind, not all imported items are libraries. Some of them are almost always built-in CircuitPython modules. How do you know the difference? Time to visit the REPL.

In the Interacting with the REPL section (https://adafru.it/Awz) on The REPL page (https://adafru.it/Awz) in this guide, the `help("modules")` command is discussed. This command provides a list of all of the built-in modules available in CircuitPython for your board. So, if you connect to the serial console on your board, and enter the REPL, you can run `help("modules")` to see what modules are available for your board. Then, as you read through the `import` statements, you can,

for the purposes of figuring out which libraries to load, ignore the statement that import modules.

The following is the list of modules built into CircuitPython for the Feather RP2040. Your list may look similar or be anything down to a significant subset of this list for smaller boards.



Now that you know what you're looking for, it's time to read through the import statements. The first two, `time` and `board`, are on the modules list above, so they're built-in.

The next one, `neopixel`, is not on the module list. That means it's your first library! So, you would head over to the bundle zip you downloaded, and search for **neopixel**. There is a **neopixel.mpy** file in the bundle zip. Copy it over to the **lib** folder on your **CIRCUITPY** drive. The following one, `adafruit_lis3dh`, is also not on the module list. Follow the same process for **adafruit_lis3dh**, where you'll find **adafruit_lis3dh.mpy**, and copy that over.

The fifth one is `usb_hid`, and it is in the modules list, so it is built in. Often all of the built-in modules come first in the import list, but sometimes they don't! Don't assume that everything after the first library is also a library, and verify each import with the modules list to be sure. Otherwise, you'll search the bundle and come up empty!

The final two imports are not as clear. Remember, when `import` statements are formatted like this, the first thing after the `from` is the library name. In this case, the library name is `adafruit_hid`. A search of the bundle will find an **adafruit_hid folder**. When a library is a folder, you must copy the **entire folder and its contents as it is in the bundle** to the **lib** folder on your **CIRCUITPY** drive. In this case, you would copy the entire **adafruit_hid** folder to your **CIRCUITPY/lib** folder.

Notice that there are two imports that begin with `adafruit_hid`. Sometimes you will need to import more than one thing from the same library. Regardless of how many times you import the same library, you only need to load the library by copying over the **adafruit_hid** folder once.

That is how you can use your example code to figure out what libraries to load on your CircuitPython-compatible board!

There are cases, however, where libraries require other libraries internally. The internally required library is called a dependency. In the event of library dependencies, the easiest way to figure out what other libraries are required is to connect to the serial console and follow along with the `ImportError` printed there. The following is a very simple example of an `ImportError`, but the concept is the same for any missing library.

Example: `ImportError` Due to Missing Library

If you choose to load libraries as you need them, or you're starting fresh with an existing example, you may end up with code that tries to use a library you haven't yet loaded. This section will demonstrate what happens when you try to utilise a library that you don't have loaded on your board, and cover the steps required to resolve the issue.

This demonstration will only return an error if you do not have the required library loaded into the **lib** folder on your **CIRCUITPY** drive.

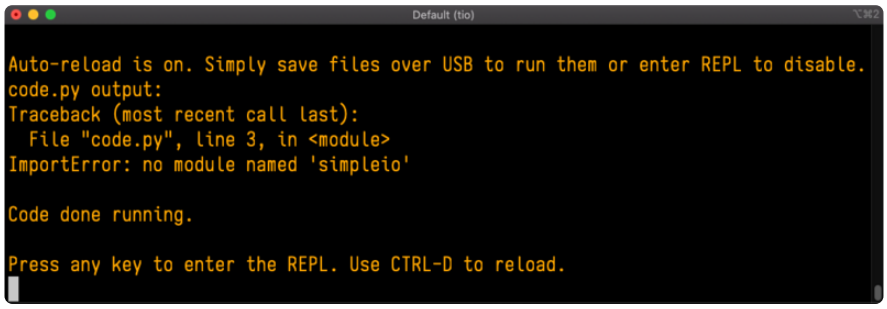
Let's use a modified version of the Blink example.

```
import board
import time
import simpleio

led = simpleio.DigitalOut(board.LED)

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

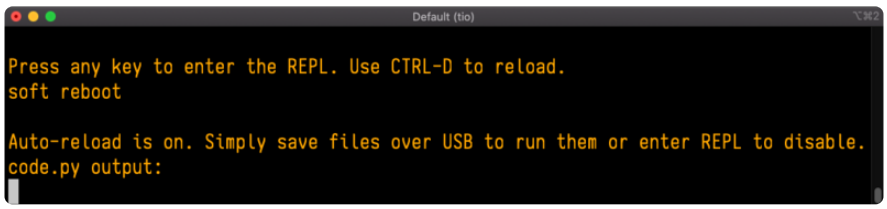
Save this file. Nothing happens to your board. Let's check the serial console to see what's going on.



You have an `ImportError`. It says there is `no module named 'simpleio'`. That's the one you just included in your code!

Click the link above to download the correct bundle. Extract the lib folder from the downloaded bundle file. Scroll down to find **simpleio.mpy**. This is the library file you're looking for! Follow the steps above to load an individual library file.

The LED starts blinking again! Let's check the serial console.



No errors! Excellent. You've successfully resolved an `ImportError`!

If you run into this error in the future, follow along with the steps above and choose the library that matches the one you're missing.

Library Install on Non-Express Boards

If you have an M0 non-Express board such as Trinket M0, Gemma M0, QT Py M0, or one of the M0 Trinkeys, you'll want to follow the same steps in the example above to install libraries as you need them. Remember, you don't need to wait for an `ImportError` if you know what library you added to your code. Open the library bundle you downloaded, find the library you need, and drag it to the **lib** folder on your **CIRCUITPY** drive.

You can still end up running out of space on your M0 non-Express board even if you only load libraries as you need them. There are a number of steps you can use to try to resolve this issue. You'll find suggestions on the Troubleshooting page (https://adafru.it/Den).

Updating CircuitPython Libraries and Examples

Libraries and examples are updated from time to time, and it's important to update the files you have on your **CIRCUITPY** drive.

To update a single library or example, follow the same steps above. When you drag the library file to your lib folder, it will ask if you want to replace it. Say yes. That's it!

A new library bundle is released every time there's an update to a library. Updates include things like bug fixes and new features. It's important to check in every so often to see if the libraries you're using have been updated.

CircUp CLI Tool

There is a command line interface (CLI) utility called CircUp (https://adafru.it/Tfi) that can be used to easily install and update libraries on your device. Follow the directions on the install page within the CircUp learn guide (https://adafru.it/-Ad). Once you've got it installed you run the command `circup update` in a terminal to interactively update all libraries on the connected CircuitPython device. See the usage page in the CircUp guide (https://adafru.it/-Ah) for a full list of functionality

---

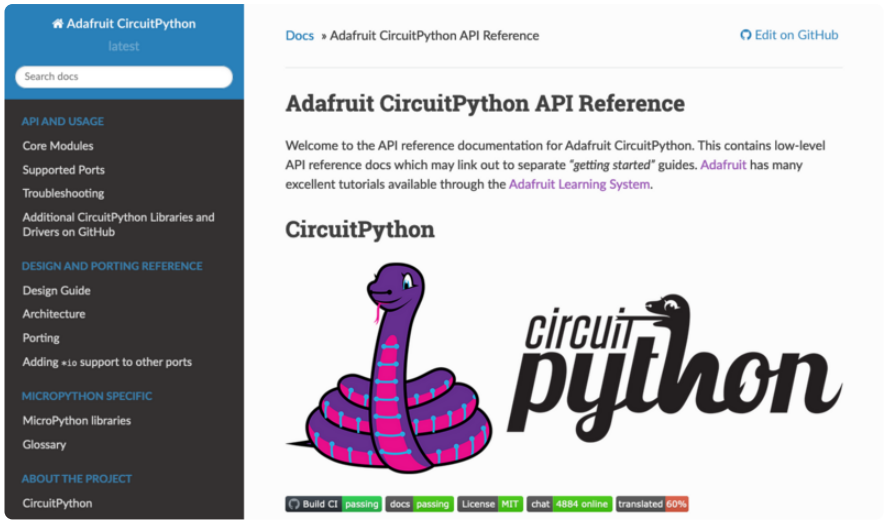
CircuitPython Documentation

You've learned about the CircuitPython built-in modules and external libraries. You know that you can find the modules in CircuitPython, and the libraries in the Library Bundles. There are guides available that explain the basics of many of the modules and libraries. However, there's sometimes more capabilities than are necessarily showcased in the guides, and often more to learn about a module or library. So, where can you find more detailed information? That's when you want to look at the API documentation.

The entire CircuitPython project comes with extensive documentation available on Read the Docs. This includes both the CircuitPython core (https://adafru.it/Beg) and the Adafruit CircuitPython libraries (https://adafru.it/Tra).

CircuitPython Core Documentation

The CircuitPython core documentation (https://adafru.it/Beg) covers many of the details you might want to know about the CircuitPython core and related topics. It includes API and usage info, a design guide and information about porting CircuitPython to new boards, MicroPython info with relation to CircuitPython, and general information about the project.



The main page covers the basics including where to **download CircuitPython**, how to **contribute**, **differences from MicroPython**, information about the **project structure**, and a **full table of contents** for the rest of the documentation.

The list along the left side leads to more information about specific topics.

The first section is **API and Usage**. This is where you can find information about how to use individual built-in **core modules**, such as `time` and `digitalio`, details about the **supported ports**, suggestions for **troubleshooting**, and basic info and links to the **library bundles**. The **Core Modules** section also includes the **Support Matrix**, which is a table of which core modules are available on which boards.

The second section is **Design and Porting Reference**. It includes a **design guide**, **architecture** information, details on **porting**, and **adding module support** to other ports.

The third section is **MicroPython Specific**. It includes information on **MicroPython and related libraries**, and a **glossary** of terms.

The fourth and final section is **About the Project**. It includes further information including details on **building, testing, and debugging CircuitPython**, along with various other useful links including the **Adafruit Community Code of Conduct**.

Whether you're a seasoned pro or new to electronics and programming, you'll find a wealth of information to help you along your CircuitPython journey in the documentation!

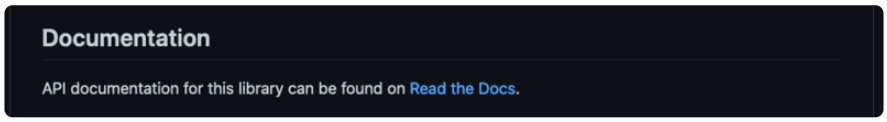
CircuitPython Library Documentation

The Adafruit CircuitPython libraries are documented in a very similar fashion. Each library has its own page on Read the Docs. There is a comprehensive list available here (https://adafru.it/Tra). Otherwise, to view the documentation for a specific library, you can visit the GitHub repository for the library, and find the link in the README.

For the purposes of this page, the LED Animation library (https://adafru.it/O2d) documentation will be featured. There are two links to the documentation in each library GitHub repo. The first one is the **docs badge** near the top of the README.



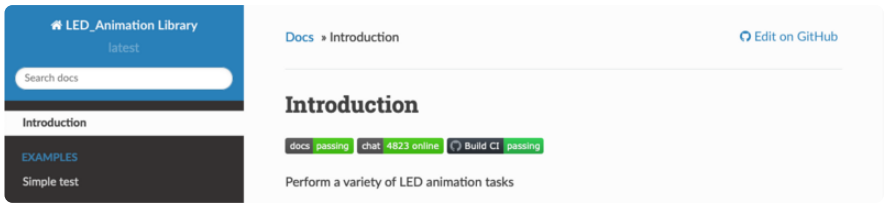
The second place is the **Documentation section** of the README. Scroll down to find it, and click on Read the Docs to get to the documentation.



Now that you know how to find it, it's time to take a look at what to expect.

Not all library documentation will look exactly the same, but this will give you some idea of what to expect from library docs.

The **Introduction** page is generated from the README, so it includes all the same info, such as PyPI installation instructions, a quick demo, and some build details. It also includes a full table of contents for the rest of the documentation (which is not part of the GitHub README). The page should look something like the following.



The left side contains links to the rest of the documentation, divided into three separate sections: **Examples**, **API Reference**, and **Other Links**.

Examples

The Examples section (https://adafru.it/VFD) is a list of library examples. This list contains anywhere from a small selection to the full list of the examples available for the library.

This section will always contain at least one example - the **simple test** example.



The simple test example is usually a basic example designed to show your setup is working. It may require other libraries to run. Keep in mind, it's simple - it won't showcase a comprehensive use of all the library features.

The LED Animation simple test demonstrates the Blink animation.



Simple test

Ensure your device works with this simple test.

examples/led_animation_simpletest.py

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT

"""
This simpletest example displays the Blink animation.

For NeoPixel FeatherWing. Update pixel_pin and pixel_num to match your wiring if using
a different form of NeoPixels.
"""
import board
import neopixel
from adafruit_led_animation.animation.blink import Blink
from adafruit_led_animation.color import RED

# Update to match the pin connected to your NeoPixels
pixel_pin = board.D6
# Update to match the number of NeoPixels you have connected
pixel_num = 32

pixels = neopixel.NeoPixel(pixel_pin, pixel_num, brightness=0.5, auto_write=False)

blink = Blink(pixels, speed=0.5, color=RED)

while True:
    blink.animate()
```

In some cases, you'll find a longer list, that may include examples that explore other features in the library. The LED Animation documentation includes a series of examples, all of which are available in the library. These examples include demonstrations of both basic and more complex features. Simply click on the example that interests you to view the associated code.



When there are multiple links in the Examples section, all of the example content is, in actuality, on the same page. Each link after the first is an anchor link to the specified section of the page. Therefore, you can also view all the available examples by scrolling down the page.

You can view the rest of the examples by clicking through the list or scrolling down the page. These examples are fully working code. Which is to say, while they may rely on other libraries as well as the library for which you are viewing the documentation, they should not require modification to otherwise work.

API Reference

The API Reference section (https://adafru.it/Rqa) includes a list of the library functions and classes. The API (Application Programming Interface) of a library is the set of functions and classes the library provides. Essentially, the API defines how your program interfaces with the functions and classes that you call in your code to use the library.

There is always at least one list item included. Libraries for which the code is included in a single Python (**.py**) file, will only have one item. Libraries for which the code is multiple Python files in a directory (called subpackages) will have multiple items in this list. The LED Animation library has a series of subpackages, and therefore, multiple items in this list.

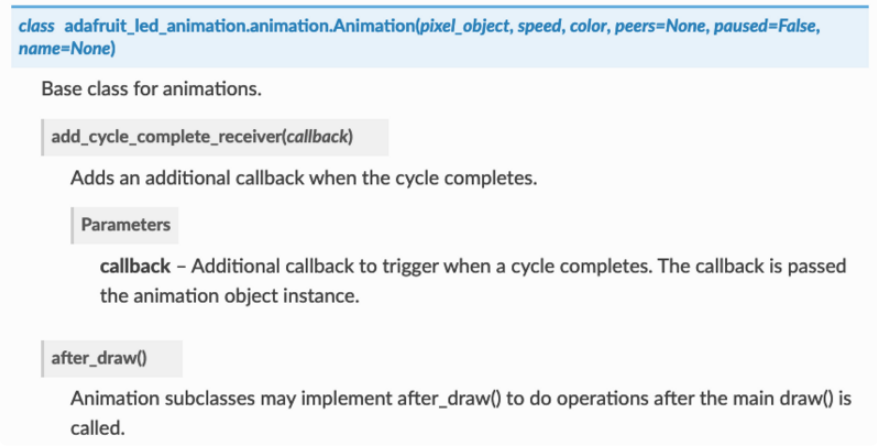
Click on the first item in the list to begin viewing the API Reference section.



As with the Examples section, all of the API Reference content is on a single page, and the links under API Reference are anchor links to the specified section of the page.

When you click on an item in the API Reference section, you'll find details about the classes and functions in the library. In the case of only one item in this section, all the available functionality of the library will be contained within that first and only subsection. However, in the case of a library that has subpackages, each item will contain the features of the particular subpackage indicated by the link. The documentation will cover all of the available functions of the library, including more complex ones that may not interest you.

The first list item is the animation subpackage. If you scroll down, you'll begin to see the available features of animation. They are listed alphabetically. Each of these things can be called in your code. It includes the name and a description of the specific function you would call, and if any parameters are necessary, lists those with a description as well.



class adafruit_led_animation.animation.Animation(*pixel_object*, *speed*, *color*, *peers=None*, *paused=False*, *name=None*)

Base class for animations.

add_cycle_complete_receiver(*callback*)

Adds an additional callback when the cycle completes.

Parameters

callback – Additional callback to trigger when a cycle completes. The callback is passed the animation object instance.

after_draw()

Animation subclasses may implement after_draw() to do operations after the main draw() is called.

You can view the other subpackages by clicking the link on the left or scrolling down the page. You may be interested in something a little more practical. Here is an example. To use the LED Animation library Comet animation, you would run the following example.

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT

"""
This example animates a jade comet that bounces from end to end of the strip.

For QT Py Haxpress and a NeoPixel strip. Update pixel_pin and pixel_num to match
your wiring if
using a different board or form of NeoPixels.

This example will run on SAMD21 (M0) Express boards (such as Circuit Playground
Express or QT Py
Haxpress), but not on SAMD21 non-Express boards (such as QT Py or Trinket).
"""
import board
import neopixel

from adafruit_led_animation.animation.comet import Comet
from adafruit_led_animation.color import JADE

# Update to match the pin connected to your NeoPixels
pixel_pin = board.A3
# Update to match the number of NeoPixels you have connected
pixel_num = 30

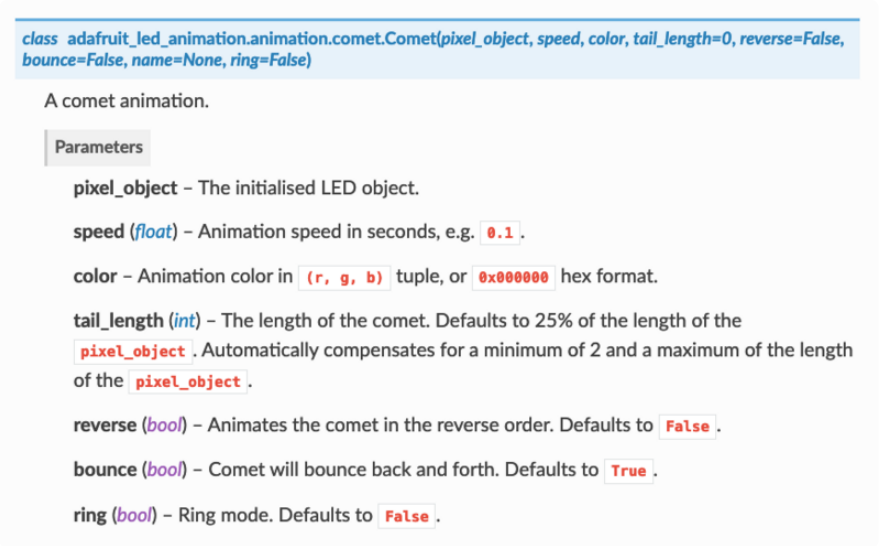
pixels = neopixel.NeoPixel(pixel_pin, pixel_num, brightness=0.5, auto_write=False)

comet = Comet(pixels, speed=0.02, color=JADE, tail_length=10, bounce=True)

while True:
    comet.animate()
```

Note the line where you create the `comet` object. There are a number of items inside the parentheses. In this case, you're provided with a fully working example. But what if you want to change how the comet works? The code alone does not explain what the options mean.

So, in the API Reference documentation list, click the `adafruit_led_animation.animation.comet` link and scroll down a bit until you see the following.



class adafruit_led_animation.animation.comet.Comet(*pixel_object*, *speed*, *color*, *tail_length=0*, *reverse=False*, *bounce=False*, *name=None*, *ring=False*)

A comet animation.

Parameters

pixel_object – The initialised LED object.

speed (*float*) – Animation speed in seconds, e.g. `0.1`.

color – Animation color in `(r, g, b)` tuple, or `0x000000` hex format.

tail_length (*int*) – The length of the comet. Defaults to 25% of the length of the `pixel_object`. Automatically compensates for a minimum of 2 and a maximum of the length of the `pixel_object`.

reverse (*bool*) – Animates the comet in the reverse order. Defaults to `False`.

bounce (*bool*) – Comet will bounce back and forth. Defaults to `True`.

ring (*bool*) – Ring mode. Defaults to `False`.

Look familiar? It is! This is the documentation for setting up the comet object. It explains what each argument provided in the comet setup in the code meant, as well as the other available features. For example, the code includes `speed=0.02`. The documentation clarifies that this is the "Animation speed in seconds". The code doesn't include `ring`. The documentation indicates this is an available setting that enables "Ring mode".

This type of information is available for any function you would set up in your code. If you need clarification on something, wonder whether there's more options available, or are simply interested in the details involved in the code you're writing, check out the documentation for the CircuitPython libraries!

Other Links

This section is the same for every library. It includes a list of links to external sites, which you can visit for more information about the CircuitPython Project and Adafruit.

That covers the CircuitPython library documentation! When you are ready to go beyond the basic library features covered in a guide, or you're interested in understanding those features better, the library documentation on Read the Docs has you covered!

Recommended Editors

The CircuitPython code on your board detects when the files are changed or written and will automatically re-start your code. This makes coding very fast because you save, and it re-runs.

However, you must wait until the file is done being saved before unplugging or resetting your board! On Windows using some editors this can sometimes take up to 90 seconds, on Linux it can take 30 seconds to complete because the text editor does not save the file completely. Mac OS does not seem to have this delay, which is nice!

This is really important to be aware of. If you unplug or reset the board before your computer finishes writing the file to your board, you can corrupt the drive. If this happens, you may lose the code you've written, so it's important to backup your code to your computer regularly.

To avoid the likelihood of filesystem corruption, use an editor that writes out the file completely when you save it. Check out the list of recommended editors below.

Recommended editors

- **mu** (https://adafru.it/ANO) is an editor that safely writes all changes (it's also our recommended editor!)
- **emacs** (https://adafru.it/xNA) is also an editor that will fully write files on save (https://adafru.it/Be7)
- Sublime Text **(https://adafru.it/xNB)** safely writes all changes
- Visual Studio Code **(https://adafru.it/Be9)** appears to safely write all changes
- **gedit** on Linux appears to safely write all changes
- IDLE (https://adafru.it/IWB), in Python 3.8.1 or later, was fixed (https://adafru.it/IWD) to write all changes immediately
- Thonny (https://adafru.it/Qb6) fully writes files on save

Recommended only with particular settings or add-ons

- **vim** (https://adafru.it/ek9) / **vi** safely writes all changes. But set up **vim** to not write swapfiles (https://adafru.it/ELO) (.swp files: temporary records of your edits) to **CIRCUITPY**. Run vim with `vim -n`, set the `no swapfile` option, or set the `directory` option to write swapfiles elsewhere. Otherwise the swapfile writes trigger restarts of your program.

- The PyCharm IDE **(https://adafru.it/xNC)** is safe if "Safe Write" is turned on in Settings->System Settings->Synchronization (true by default).
- If you are using **Atom** (https://adafru.it/fMG), install the fsync-on-save package (https://adafru.it/E9m) or the language-circuitpython package (https://adafru.it/Vuf) so that it will always write out all changes to files on **CIRCUITPY**.
- **SlickEdit** (https://adafru.it/DdP) works only if you add a macro to flush the disk (https://adafru.it/ven).

The editors listed below are specifically NOT recommended!

Editors that are NOT recommended

- **notepad** (the default Windows editor) and **Notepad++** can be slow to write, so the editors above are recommended! If you are using notepad, be sure to eject the drive.
- **IDLE** in Python 3.8.0 or earlier does not force out changes immediately.
- **nano** (on Linux) does not force out changes.
- **geany** (on Linux) does not force out changes.
- **Anything else** - Other editors have not been tested so please use a recommended one!

Advanced Serial Console on Windows

Windows 7 and 8.1

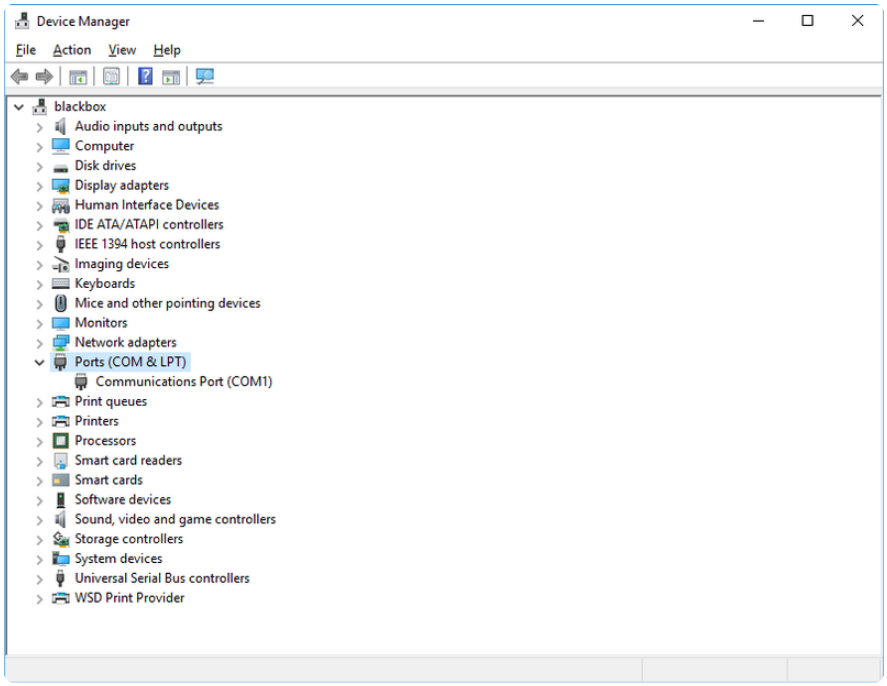
If you're using Windows 7 (or 8 or 8.1), you'll need to install drivers. See the Windows 7 and 8.1 Drivers page (https://adafru.it/VuB) for details. You will not need to install drivers on Mac, Linux or Windows 10.

You are strongly encouraged to upgrade to Windows 10 if you are still using Windows 7 or Windows 8 or 8.1. Windows 7 has reached end-of-life and no longer receives security updates. A free upgrade to Windows 10 is still available (https://adafru.it/RWc).

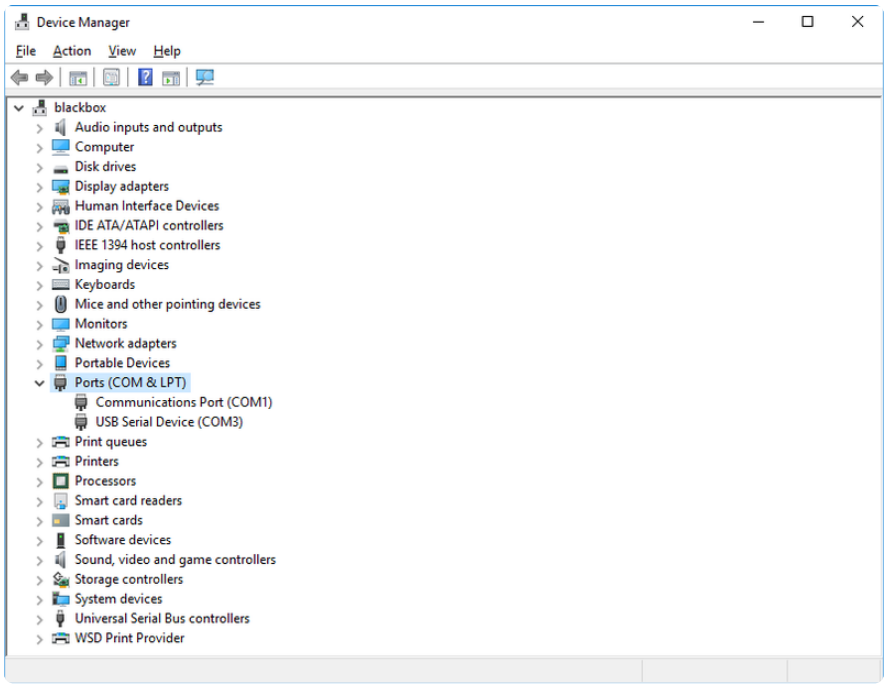
What's the COM?

First, you'll want to find out which serial port your board is using. When you plug your board in to USB on your computer, it connects to a serial port. The port is like a door through which your board can communicate with your computer using USB.

You'll use Windows Device Manager to determine which port the board is using. The easiest way to determine which port the board is using is to first check **without** the board plugged in. Open Device Manager. Click on Ports (COM & LPT). You should find something already in that list with (COM#) after it where # is a number.



Now plug in your board. The Device Manager list will refresh and a new item will appear under Ports (COM & LPT). You'll find a different (COM#) after this item in the list.



Sometimes the item will refer to the name of the board. Other times it may be called something like USB Serial Device, as seen in the image above. Either way, there is a new (COM#) following the name. This is the port your board is using.

Install Putty

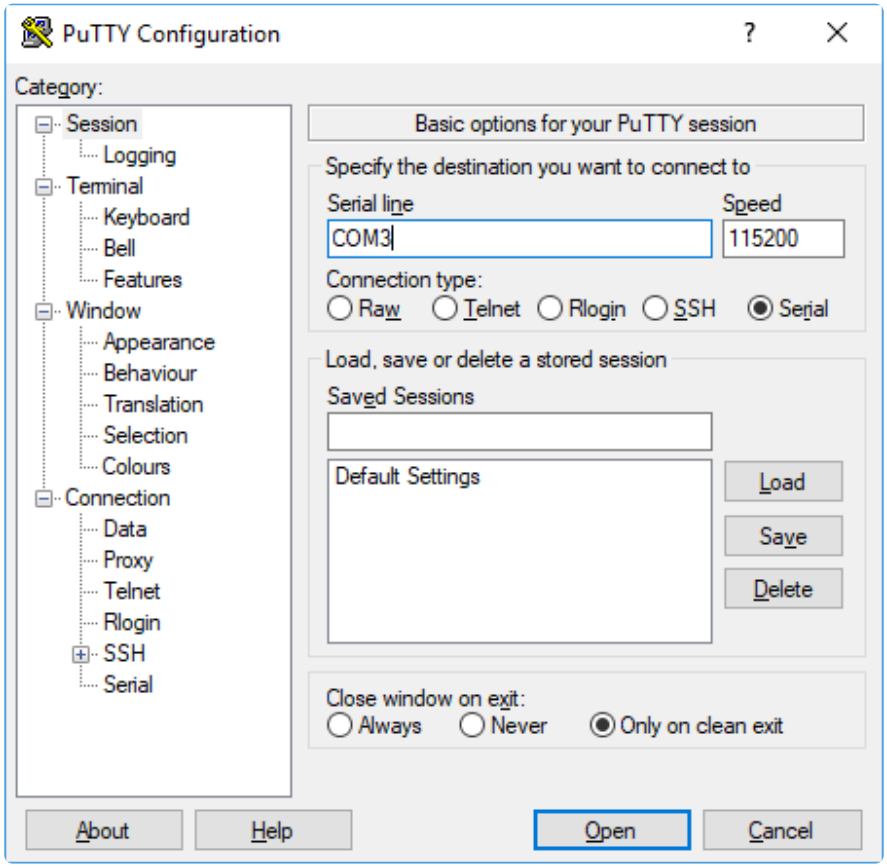
If you're using Windows, you'll need to download a terminal program. You're going to use PuTTY.

The first thing to do is download the latest version of PuTTY (https://adafru.it/Bf1). You'll want to download the Windows installer file. It is most likely that you'll need the 64-bit version. Download the file and install the program on your machine. If you run into issues, you can try downloading the 32-bit version instead. However, the 64-bit version will work on most PCs.

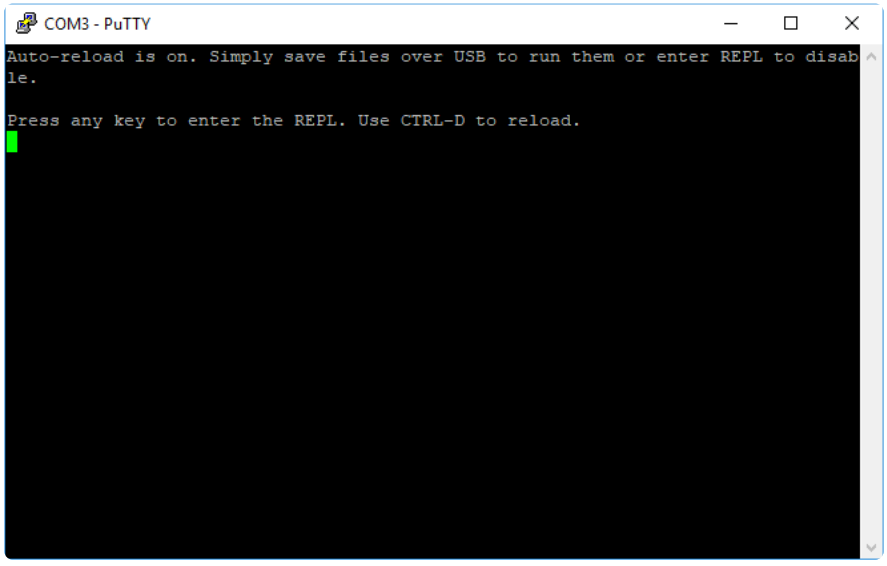
Now you need to open PuTTY.

- Under **Connection type:** choose the button next to **Serial**.
- In the box under **Serial line**, enter the serial port you found that your board is using.
- In the box under **Speed**, enter 115200. This called the baud rate, which is the speed in bits per second that data is sent over the serial connection. For boards with built in USB it doesn't matter so much but for ESP8266 and other board with a separate chip, the speed required by the board is 115200 bits per second. So you might as well just use 115200!

If you want to save those settings for later, use the options under **Load, save or delete a stored session.** Enter a name in the box under **Saved Sessions**, and click the **Save** button on the right.



Once your settings are entered, you're ready to connect to the serial console. Click "Open" at the bottom of the window. A new window will open.



If no code is running, the window will either be blank or will look like the window above. Now you're ready to see the results of your code.

Great job! You've connected to the serial console!

Advanced Serial Console on Mac

Connecting to the serial console on Mac does not require installing any drivers or extra software. You'll use a terminal program to find your board, and `screen` to connect to it. Terminal and `screen` both come installed by default.

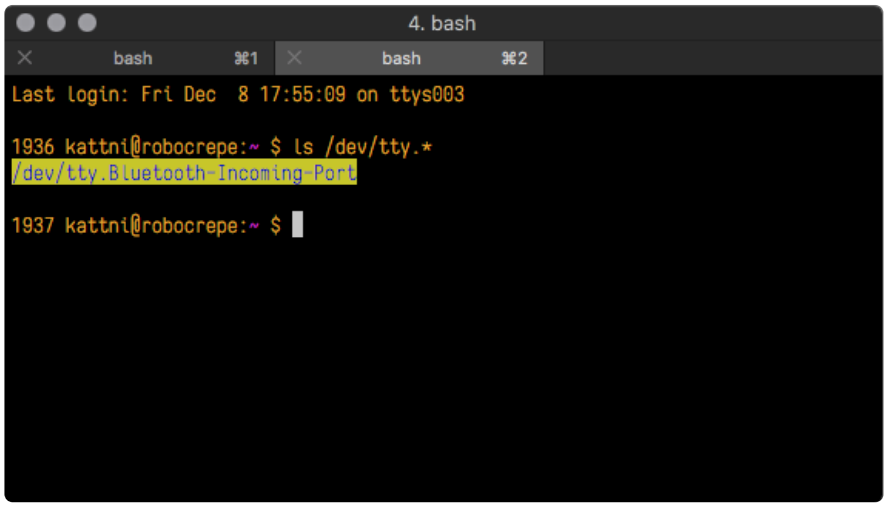
What's the Port?

First you'll want to find out which serial port your board is using. When you plug your board in to USB on your computer, it connects to a serial port. The port is like a door through which your board can communicate with your computer using USB.

The easiest way to determine which port the board is using is to first check **without** the board plugged in. Open Terminal and type the following:

```
ls /dev/tty.*
```

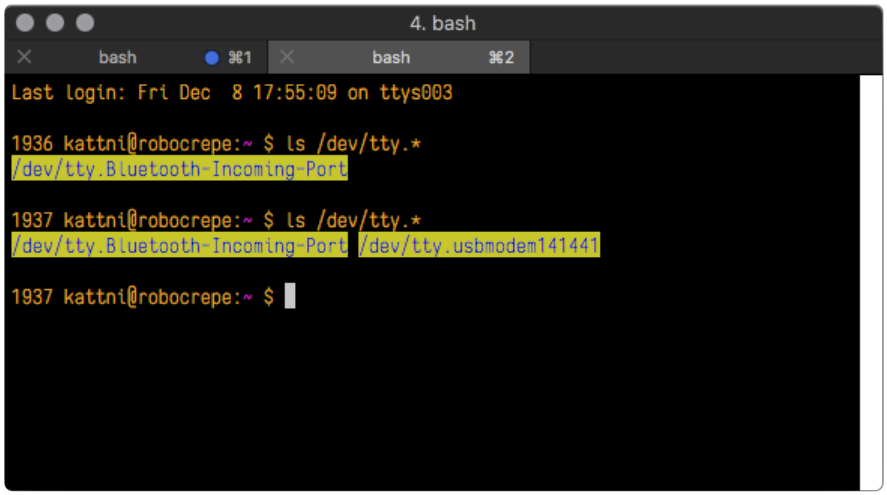
Each serial connection shows up in the `/dev/` directory. It has a name that starts with `tty.`. The command `ls` shows you a list of items in a directory. You can use `*` as a wildcard, to search for files that start with the same letters but end in something different. In this case, you're asking to see all of the listings in `/dev/` that start with `tty.` and end in anything. This will show us the current serial connections.



Now, plug your board. In Terminal, type:

```
ls /dev/tty.*
```

This will show you the current serial connections, which will now include your board.



A new listing has appeared called `/dev/tty.usbmodem141441`.
The `tty.usbmodem141441` part of this listing is the name the example board is using. Yours will be called something similar.

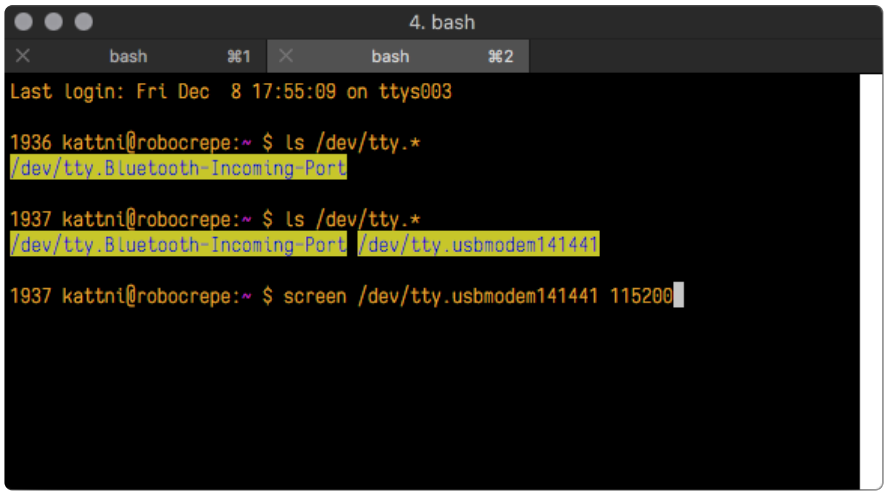
Using Linux, a new listing has appeared called `/dev/ttyACM0`. The `ttyACM0` part of this listing is the name the example board is using. Yours will be called something similar.

Connect with screen

Now that you know the name your board is using, you're ready connect to the serial console. You're going to use a command called `screen`. The `screen` command is included with MacOS. To connect to the serial console, use Terminal. Type the following command, replacing `board_name` with the name you found your board is using:

```
screen /dev/tty.board_name 115200
```

The first part of this establishes using the `screen` command. The second part tells screen the name of the board you're trying to use. The third part tells screen what baud rate to use for the serial connection. The baud rate is the speed in bits per second that data is sent over the serial connection. In this case, the speed required by the board is 115200 bits per second.



Press enter to run the command. It will open in the same window. If no code is running, the window will be blank. Otherwise, you'll see the output of your code.

Great job! You've connected to the serial console!

Advanced Serial Console on Linux

Connecting to the serial console on Linux does not require installing any drivers, but you may need to install `screen` using your package manager. You'll use a terminal program to find your board, and `screen` to connect to it. There are a variety of terminal programs such as gnome-terminal (called Terminal) or Konsole on KDE.

The `tio` program works as well to connect to your board, and has the benefit of automatically reconnecting. You would need to install it using your package manager.

What's the Port?

First you'll want to find out which serial port your board is using. When you plug your board in to USB on your computer, it connects to a serial port. The port is like a door through which your board can communicate with your computer using USB.

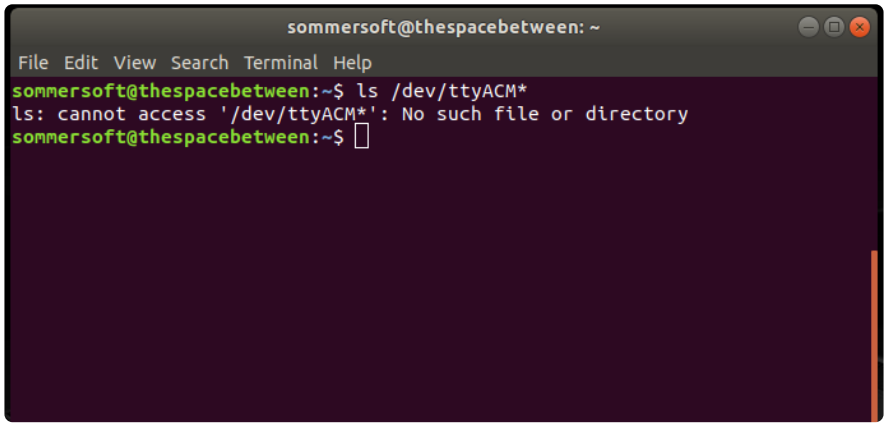
The easiest way to determine which port the board is using is to first check **without** the board plugged in. Open your terminal program and type the following:

```
ls /dev/ttyACM*
```

Each serial connection shows up in the **/dev/** directory. It has a name that starts with **ttyACM**. The command `ls` shows you a list of items in a directory. You can use `*` as a wildcard, to search for files that start with the same letters but end in something

different. In this case, You're asking to see all of the listings in **/dev/** that start with **ttyACM** and end in anything. This will show us the current serial connections.

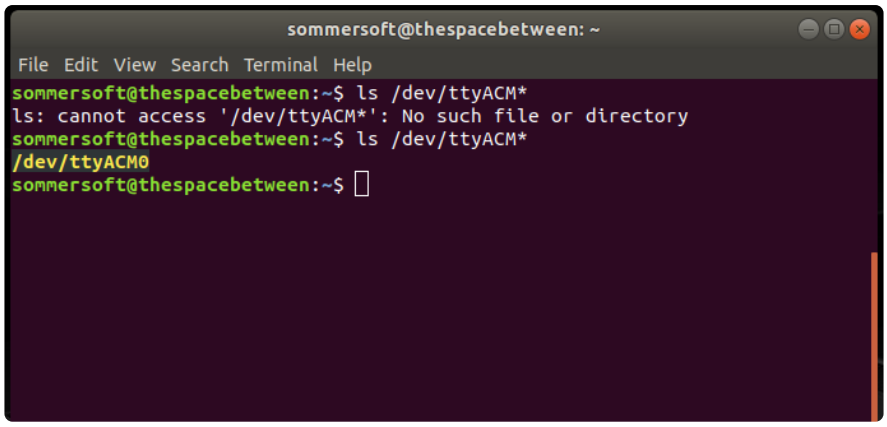
In the example below, the error is indicating that are no current serial connections starting with **ttyACM**.



Now plug in your board. In your terminal program, type:

```
ls /dev/ttyACM*
```

This will show you the current serial connections, which will now include your board.



A new listing has appeared called **/dev/ttyACM0**. The **ttyACM0** part of this listing is the name the example board is using. Yours will be called something similar.

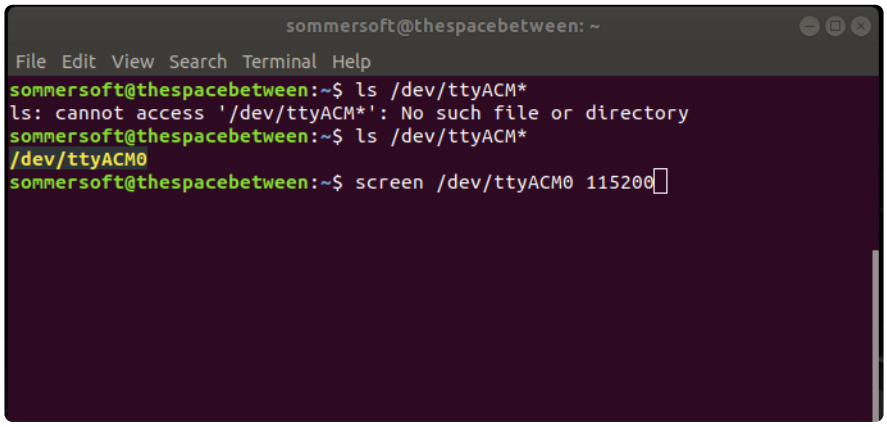
Connect with screen

Now that you know the name your board is using, you're ready connect to the serial console. You'll use a command called `screen`. You may need to install it using the package manager.

To connect to the serial console, use your terminal program. Type the following command, replacing `board_name` with the name you found your board is using:

```
screen /dev/tty.board_name 115200
```

The first part of this establishes using the `screen` command. The second part tells screen the name of the board you're trying to use. The third part tells screen what baud rate to use for the serial connection. The baud rate is the speed in bits per second that data is sent over the serial connection. In this case, the speed required by the board is 115200 bits per second.

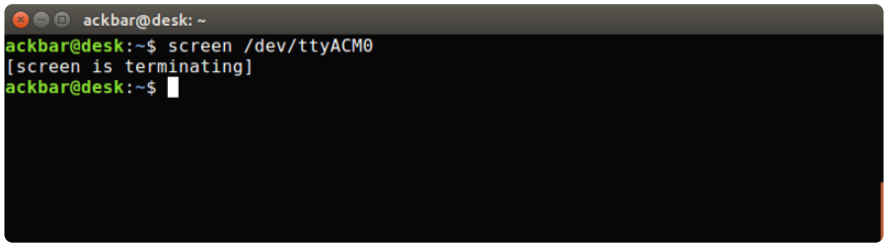


Press enter to run the command. It will open in the same window. If no code is running, the window will be blank. Otherwise, you'll see the output of your code.

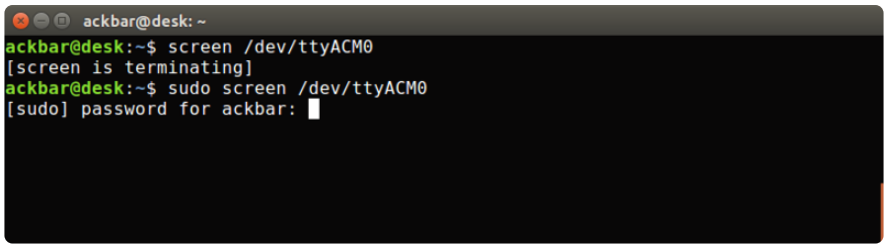
Great job! You've connected to the serial console!

Permissions on Linux

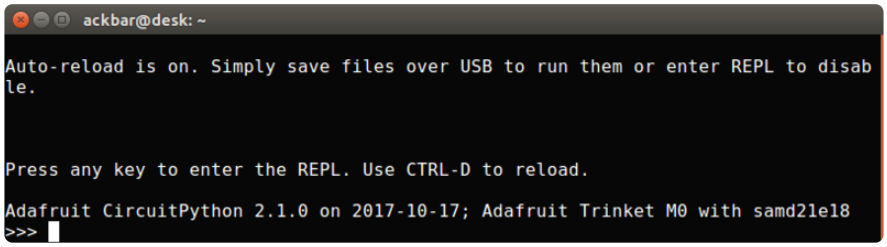
If you try to run `screen` and it doesn't work, then you may be running into an issue with permissions. Linux keeps track of users and groups and what they are allowed to do and not do, like access the hardware associated with the serial connection for running `screen`. So if you see something like this:



then you may need to grant yourself access. There are generally two ways you can do this. The first is to just run `screen` using the `sudo` command, which temporarily gives you elevated privileges.

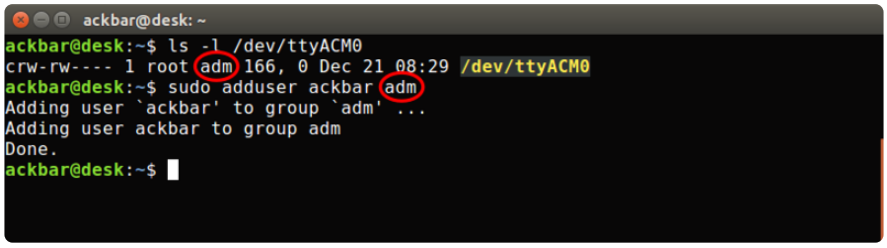


Once you enter your password, you should be in:

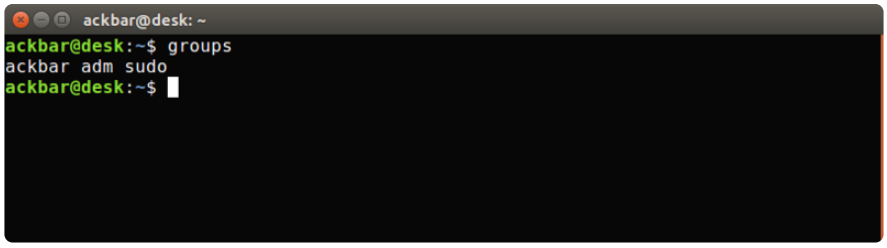


The second way is to add yourself to the group associated with the hardware. To figure out what that group is, use the command `ls -l` as shown below. The group name is circled in red.

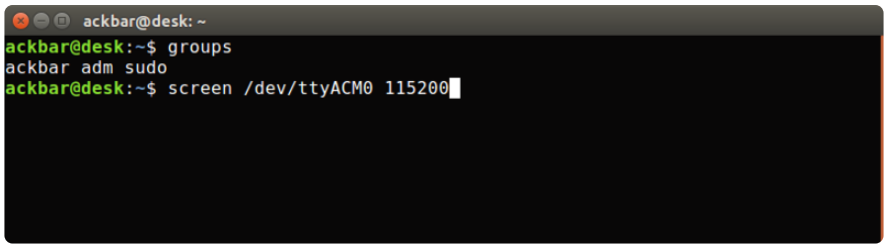
Then use the command `adduser` to add yourself to that group. You need elevated privileges to do this, so you'll need to use `sudo`. In the example below, the group is **adm** and the user is **ackbar**.



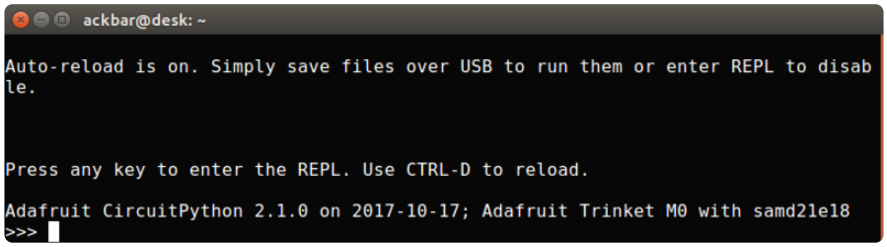
After you add yourself to the group, you'll need to logout and log back in, or in some cases, reboot your machine. After you log in again, verify that you have been added to the group using the command `groups`. If you are still not in the group, reboot and check again.



And now you should be able to run `screen` without using `sudo`.



And you're in:



The examples above use `screen`, but you can also use other programs, such as `putty` or `picocom`, if you prefer.

Frequently Asked Questions

These are some of the common questions regarding CircuitPython and CircuitPython microcontrollers.

What are some common acronyms to know?

CP or CPy = CircuitPython (https://adafru.it/KJD)
CPC = Circuit Playground Classic (http://adafru.it/3000) (does not run CircuitPython)
CPX = Circuit Playground Express (http://adafru.it/3333)
CPB = Circuit Playground Bluefruit (http://adafru.it/4333)

Using Older Versions

As CircuitPython development continues and there are new releases, Adafruit will stop supporting older releases. Visit https://circuitpython.org/downloads to download the latest version of CircuitPython for your board. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update CircuitPython and then visit https://circuitpython.org/libraries to download the latest Library Bundle.

I have to continue using CircuitPython 7.x or earlier. Where can I find compatible libraries?

We are no longer building or supporting the CircuitPython 7.x or earlier library bundles. We highly encourage you to update CircuitPython to the latest version **(https://adafru.it/Em8) and use** the current version of the libraries **(https://adafru.it/ENC).** However, if for some reason you cannot update, here are the last available library bundles for older versions:

- 2.x bundle (https://adafru.it/FJA)
- 3.x bundle (https://adafru.it/FJB)
- 4.x bundle (https://adafru.it/QDL)
- 5.x bundle (https://adafru.it/QDJ)
- 6.x bundle (https://adafru.it/Xmf)
- 7.x bundle (https://adafru.it/18e9)

Python Arithmetic

Does CircuitPython support floating-point numbers?

All CircuitPython boards support floating point arithmetic, even if the microcontroller chip does not support floating point in hardware. Floating point numbers are stored in 30 bits, with an 8-bit exponent and a 22-bit mantissa. Note that this is two bits less than standard 32-bit single-precision floats. You will get about 5-1/2 digits of decimal precision.

(The **broadcom** port may provide 64-bit floats in some cases.)

Does CircuitPython support long integers, like regular Python?

Python long integers (integers of arbitrary size) are available on most builds, except those on boards with the smallest available firmware size. On these boards, integers are stored in 31 bits.

Boards without long integer support are mostly SAMD21 ("M0") boards without an external flash chip, such as the Adafruit Gemma M0, Trinket M0, QT Py M0, and the Trinkey series. There are also a number of third-party boards in this category. There are also a few small STM third-party boards without long integer support.

`time.localtime()`, `time.mktime()`, `time.time()`, and `time.monotonic_ns()` are available only on builds with long integers.

Wireless Connectivity

How do I connect to the Internet with CircuitPython?

If you'd like to include WiFi in your project, your best bet is to use a board that is running natively on ESP32 chipsets - those have WiFi built in!

If your development board has an SPI port and at least 4 additional pins, you can check out this guide (https://adafru.it/F5X) on using AirLift with CircuitPython - extra wiring is required and some boards like the MacroPad or NeoTrellis do not have enough available pins to add the hardware support.

For further project examples, and guides about using AirLift with specific hardware, check out the Adafruit Learn System (https://adafru.it/VBr).

How do I do BLE (Bluetooth Low Energy) with CircuitPython?

The nRF52840 and nRF52833 boards have the most complete BLE implementation. Your program can act as both a BLE central and peripheral. As a central, you can scan for advertisements, and connect to an advertising board. As a peripheral, you can advertise, and you can create services available to a central. Pairing and bonding are supported.

ESP32-C3 and ESP32-S3 boards currently provide an incomplete (https://adafru.it/11Au) BLE implementation. Your program can act as a central, and connect to a peripheral. You can advertise, but you cannot create services. You cannot advertise anonymously. Pairing and bonding are not supported.

The ESP32 could provide a similar implementation, but it is not yet available. Note that the ESP32-S2 does not have Bluetooth capability.

On most other boards with adequate firmware space, BLE is available for use with AirLift (https://adafru.it/11Av) or other NINA-FW-based co-processors. Some boards have this coprocessor on board, such as the PyPortal (https://adafru.it/11Aw). Currently, this implementation only supports acting as a BLE peripheral. Scanning and connecting as a central are not yet implemented. Bonding and pairing are not supported.

Are there other ways to communicate by radio with CircuitPython?

Check out Adafruit's RFM boards (https://adafru.it/11Ay)for simple radio communication supported by CircuitPython, which can be used over distances of 100m to over a km, depending on the version. The RFM SAMD21 M0 boards can be used, but they were not designed for CircuitPython, and have limited RAM and flash space; using the RFM breakouts or FeatherWings with more capable boards will be easier.

Asyncio and Interrupts

Is there asyncio support in CircuitPython?

There is support for asyncio starting with CircuitPython 7.1.0, on all boards except the smallest SAMD21 builds. Read about using it in the Cooperative Multitasking in CircuitPython (https://adafru.it/XnA) Guide.

Does CircuitPython support interrupts?

No. CircuitPython does not currently support interrupts - please use asyncio for multitasking / 'threaded' control of your code

---

Status RGB LED

My RGB NeoPixel/DotStar LED is blinking funny colors - what does it mean?

The status LED can tell you what's going on with your CircuitPython board. Read more here for what the colors mean! (https://adafru.it/Den)

Memory Issues

What is a MemoryError?

Memory allocation errors happen when you're trying to store too much on the board. The CircuitPython microcontroller boards have a limited amount of memory available. You can have about 250 lines of code on the M0 Express boards. If you try to `import` too many libraries, a combination of large libraries, or run a program

with too many lines of code, your code will fail to run and you will receive a `MemoryError` in the serial console.

What do I do when I encounter a MemoryError?

Try resetting your board. Each time you reset the board, it reallocates the memory. While this is unlikely to resolve your issue, it's a simple step and is worth trying.

Make sure you are using **.mpy** versions of libraries. All of the CircuitPython libraries are available in the bundle in a **.mpy** format which takes up less memory than **.py** format. Be sure that you're using the latest library bundle (https://adafru.it/uap) for your version of CircuitPython.

If that does not resolve your issue, try shortening your code. Shorten comments, remove extraneous or unneeded code, or any other clean up you can do to shorten your code. If you're using a lot of functions, you could try moving those into a separate library, creating a **.mpy** of that library, and importing it into your code.

You can turn your entire file into a **.mpy** and `import` that into **code.py**. This means you will be unable to edit your code live on the board, but it can save you space.

Can the order of my `import` statements affect memory?

It can because the memory gets fragmented differently depending on allocation order and the size of objects. Loading **.mpy** files uses less memory so its recommended to do that for files you aren't editing.

How can I create my own **.mpy** files?

You can make your own **.mpy** versions of files with `mpy-cross`.

You can download `mpy-cross` for your operating system from here (https://adafru.it/QDK). Builds are available for Windows, macOS, x64 Linux, and Raspberry Pi Linux. Choose the latest `mpy-cross` whose version matches the version of CircuitPython you are using.

To make a **.mpy** file, run `./mpy-cross path/to/yourfile.py` to create a **yourfile.mpy** in the same directory as the original file.

How do I check how much memory I have free?

Run the following to see the number of bytes available for use:

```
import gc
gc.mem_free()
```

Unsupported Hardware

Is ESP8266 or ESP32 supported in CircuitPython? Why not?

We dropped ESP8266 support as of 4.x - For more information please read about it here (https://adafru.it/CiG)!

As of CircuitPython 8.x we have started to support ESP32 and ESP32-C3 and have added a WiFi workflow for wireless coding! (https://adafru.it/10JF)

We also support ESP32-S2 & ESP32-S3, which have native USB.

Does Feather M0 support WINC1500?

No, WINC1500 will not fit into the M0 flash space.

Can AVRs such as ATmega328 or ATmega2560 run CircuitPython?

No.

Troubleshooting

From time to time, you will run into issues when working with CircuitPython. Here are a few things you may encounter and how to resolve them.

As CircuitPython development continues and there are new releases, Adafruit will stop supporting older releases. Visit https://circuitpython.org/downloads to download the latest version of CircuitPython for your board. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update CircuitPython and then visit https://circuitpython.org/libraries to download the latest Library Bundle.

Always Run the Latest Version of CircuitPython and Libraries

As CircuitPython development continues and there are new releases, Adafruit will stop supporting older releases. **You need to** update to the latest CircuitPython. **(https://adafru.it/Em8).**

You need to download the CircuitPython Library Bundle that matches your version of CircuitPython. **Please update CircuitPython and then** download the latest bundle **(https://adafru.it/ENC)**.

As new versions of CircuitPython are released, Adafruit will stop providing the previous bundles as automatically created downloads on the Adafruit CircuitPython Library Bundle repo. If you must continue to use an earlier version, you can still download the appropriate version of `mpy-cross` from the particular release of CircuitPython on the CircuitPython repo and create your own compatible .mpy library files. **However, it is best to update to the latest for both CircuitPython and the library bundle.**

I have to continue using CircuitPython 7.x or earlier. Where can I find compatible libraries?

Adafruit is no longer building or supporting the CircuitPython 7.x or earlier library bundles. You are highly encourged to update CircuitPython to the latest version **(https://adafru.it/Em8) and use** the current version of the libraries **(https://adafru.it/ENC).** However, if for some reason you cannot update, links to the previous bundles are available in the FAQ (https://adafru.it/FwY).

macOS Sonoma 14.x: Disk Errors Writing to CIRCUITPY

macOS Sonoma before 14.4 beta 2 takes many seconds to complete writes to small FAT drives, 8MB or smaller. This causes errors when writing to CIRCUITPY. The best solution is to remount the CIRCUITPY drive after it is automatically mounted. Or consider downgrading back to Ventura if that works for you. This problem is being tracked in CircuitPython GitHub issue 8449 (https://adafru.it/18ea).

Here is a shell script to do this remount conveniently (courtesy @czei in GitHub (https://adafru.it/18ea)). Copy the code here into a file named, say, **remount-**

CIRCUITPY.sh. Place the file in a directory on your PATH, or in some other convenient place.

macOS Sonoma 14.4 beta and after does not have the problem above, but does take an inordinately long time to write to FAT drives of size 1GB or less (40 times longer than 2GB drives). This problem is being tracked in CircuitPython GitHub issue 8918 (https://adafru.it/19iD).

```
#!/bin/sh
#
# This works around bug where, by default, macOS 14.x writes part of a file
# immediately, and then doesn't update the directory for 20-60 seconds, causing
# the file system to be corrupted.
#

disky=`df | grep CIRCUITPY | cut -d" " -f1`
sudo umount /Volumes/CIRCUITPY
sudo mkdir /Volumes/CIRCUITPY
sleep 2
sudo mount -v -o noasync -t msdos $disky /Volumes/CIRCUITPY
```

Then in a Terminal window, do this to make this script executable:

```
chmod +x remount-CIRCUITPY.sh
```

Place the file in a directory on your `PATH`, or in some other convenient place.

Now, each time you plug in or reset your CIRCUITPY board, run the file **remount-CIRCUITPY.sh**. You can run it in a Terminal window or you may be able to place it on the desktop or in your dock to run it just by double-clicking.

This will be something of a nuisance but it is the safest solution.

This problem is being tracked in this CircuitPython issue (https://adafru.it/18ea).

Bootloader (boardnameBOOT) Drive Not Present

You may have a different board.

Only Adafruit Express boards and the SAMD21 non-Express boards ship with the UF2 bootloader (https://adafru.it/zbX)installed. The Feather M0 Basic, Feather M0 Adalogger, and similar boards use a regular Arduino-compatible bootloader, which does not show a **boardnameBOOT** drive.

MakeCode

If you are running a MakeCode (https://adafru.it/zbY) program on Circuit Playground Express, press the reset button just once to get the **CPLAYBOOT** drive to show up. Pressing it twice will not work.

macOS

DriveDx and its accompanything **SAT SMART Driver** can interfere with seeing the BOOT drive. See this forum post (https://adafru.it/sTc) for how to fix the problem.

Windows 10

Did you install the Adafruit Windows Drivers package by mistake, or did you upgrade to Windows 10 with the driver package installed? You don't need to install this package on Windows 10 for most Adafruit boards. The old version (v1.5) can interfere with recognizing your device. Go to **Settings** -> **Apps** and uninstall all the "Adafruit" driver programs.

Windows 7 or 8.1

To use a CircuitPython-compatible board with Windows 7 or 8.1, you must install a driver. Installation instructions are available here (https://adafru.it/VuB).

Windows 7 and 8.1 have reached end of life. It is recommended (https://adafru.it/Amd) that you upgrade to Windows 10 if possible; an upgrade is probably still free for you. Check here (https://adafru.it/Amd).

The Windows Drivers installer was last updated in November 2020 (v2.5.0.0) . Windows 7 drivers for CircuitPython boards released since then, including RP2040 boards, are not available. There are no plans to release drivers for new boards. The boards work fine on Windows 10.

You should now be done! Test by unplugging and replugging the board. You should see the **CIRCUITPY** drive, and when you double-click the reset button (single click on Circuit Playground Express running MakeCode), you should see the appropriate **boardnameBOOT** drive.

Let us know in the Adafruit support forums (https://adafru.it/jlf) or on the Adafruit Discord () if this does not work for you!

Windows Explorer Locks Up When Accessing **boardnameBOOT** Drive

On Windows, several third-party programs that can cause issues. The symptom is that you try to access the **boardnameBOOT** drive, and Windows or Windows Explorer seems to lock up. These programs are known to cause trouble:

- **AIDA64**: to fix, stop the program. This problem has been reported to AIDA64. They acquired hardware to test, and released a beta version that fixes the problem. This may have been incorporated into the latest release. Please let us know in the forums if you test this.
- **Hard Disk Sentinel**
- **Kaspersky anti-virus**: To fix, you may need to disable Kaspersky completely. Disabling some aspects of Kaspersky does not always solve the problem. This problem has been reported to Kaspersky.
- **ESET NOD32 anti-virus**: There have been problems with at least version 9.0.386.0, solved by uninstallation.

Copying UF2 to **boardnameBOOT** Drive Hangs at 0% Copied

On Windows, a **Western DIgital (WD) utility** that comes with their external USB drives can interfere with copying UF2 files to the **boardnameBOOT** drive. Uninstall that utility to fix the problem.

CIRCUITPY Drive Does Not Appear or Disappears Quickly

Kaspersky anti-virus can block the appearance of the **CIRCUITPY** drive. There has not yet been settings change discovered that prevents this. Complete uninstallation of Kaspersky fixes the problem.

Norton anti-virus can interfere with **CIRCUITPY**. A user has reported this problem on Windows 7. The user turned off both Smart Firewall and Auto Protect, and **CIRCUITPY** then appeared.

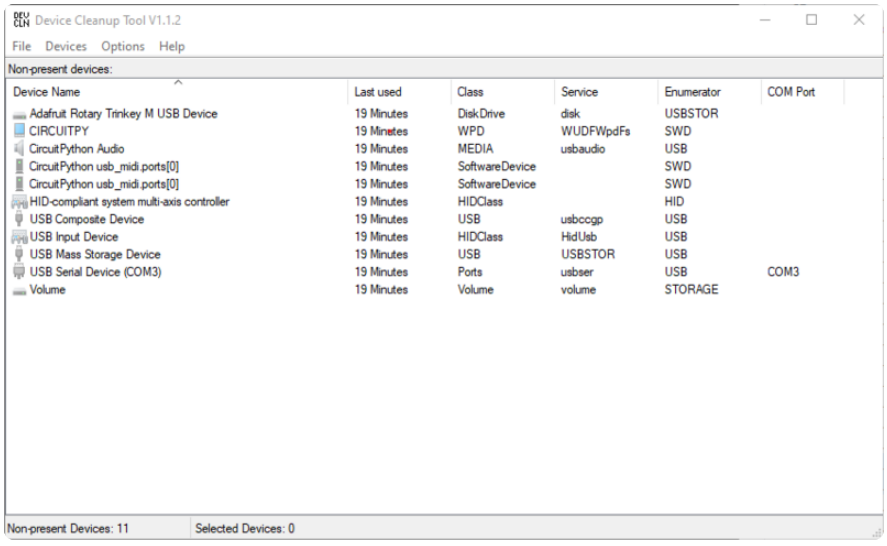
Sophos Endpoint security software can cause CIRCUITPY to disappear (https://adafru.it/ELr) and the BOOT drive to reappear. It is not clear what causes this behavior.

Samsung Magician can cause CIRCUITPY to disappear (reported here (https://adafru.it/18eb) and here (https://adafru.it/18ec)).

Device Errors or Problems on Windows

Windows can become confused about USB device installations. This is particularly true of Windows 7 and 8.1. It is recommended (https://adafru.it/Amd) that you upgrade to Windows 10 if possible; an upgrade is probably still free for you: see this link (https://adafru.it/V2a).

If not, try cleaning up your USB devices. Use Uwe Sieber's Device Cleanup Tool (https://adafru.it/RWd) (on that page, scroll down to "Device Cleanup Tool"). Download and unzip the tool. Unplug all the boards and other USB devices you want to clean up. Run the tool as Administrator. You will see a listing like this, probably with many more devices. It is listing all the USB devices that are not currently attached.



Select all the devices you want to remove, and then press Delete. It is usually safe just to select everything. Any device that is removed will get a fresh install when you plug it in. Using the Device Cleanup Tool also discards all the COM port assignments for the unplugged boards. If you have used many Arduino and CircuitPython boards, you have probably seen higher and higher COM port numbers used, seemingly without end. This will fix that problem.

Serial Console in Mu Not Displaying Anything

There are times when the serial console will accurately not display anything, such as, when no code is currently running, or when code with no serial output is already running before you open the console. However, if you find yourself in a situation where you feel it should be displaying something like an error, consider the following.

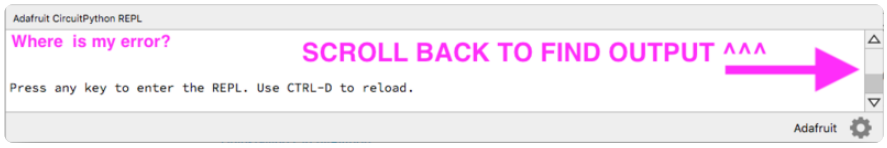
Depending on the size of your screen or Mu window, when you open the serial console, the serial console panel may be very small. This can be a problem. A basic CircuitPython error takes 10 lines to display!

```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Traceback (most recent call last):
  File "code.py", line 7
SyntaxError: invalid syntax


Press any key to enter the REPL. Use CTRL-D to reload.
```

More complex errors take even more lines!

Therefore, if your serial console panel is five lines tall or less, you may only see blank lines or blank lines followed by `Press any key to enter the REPL. Use CTRL-D to reload.`. If this is the case, you need to either mouse over the top of the panel to utilise the option to resize the serial panel, or use the scrollbar on the right side to scroll up and find your message.



This applies to any kind of serial output whether it be error messages or print statements. So before you start trying to debug your problem on the hardware side, be sure to check that you haven't simply missed the serial messages due to serial output panel height.

code.py Restarts Constantly

CircuitPython will restart **code.py** if you or your computer writes to something on the CIRCUITPY drive. This feature is called auto-reload, and lets you test a change to your program immediately.

Some utility programs, such as backup, anti-virus, or disk-checking apps, will write to the CIRCUITPY as part of their operation. Sometimes they do this very frequently, causing constant restarts.

Acronis True Image and related Acronis programs on Windows are known to cause this problem. It is possible to prevent this by disabling the " (https://adafru.it/XDZ)Acronis Managed Machine Service Mini" (https://adafru.it/XDZ).

If you cannot stop whatever is causing the writes, you can disable auto-reload by putting this code in **boot.py** or **code.py**:

```
import supervisor

supervisor.runtime.autoreload = False
```

CircuitPython RGB Status Light

Nearly all CircuitPython-capable boards have a single NeoPixel or DotStar RGB LED on the board that indicates the status of CircuitPython. A few boards designed before CircuitPython existed, such as the Feather M0 Basic, do not.

Circuit Playground Express and Circuit Playground Bluefruit have multiple RGB LEDs, but do NOT have a status LED. The LEDs are all green when in the bootloader. In versions before 7.0.0, they do NOT indicate any status while running CircuitPython.

CircuitPython 7.0.0 and Later

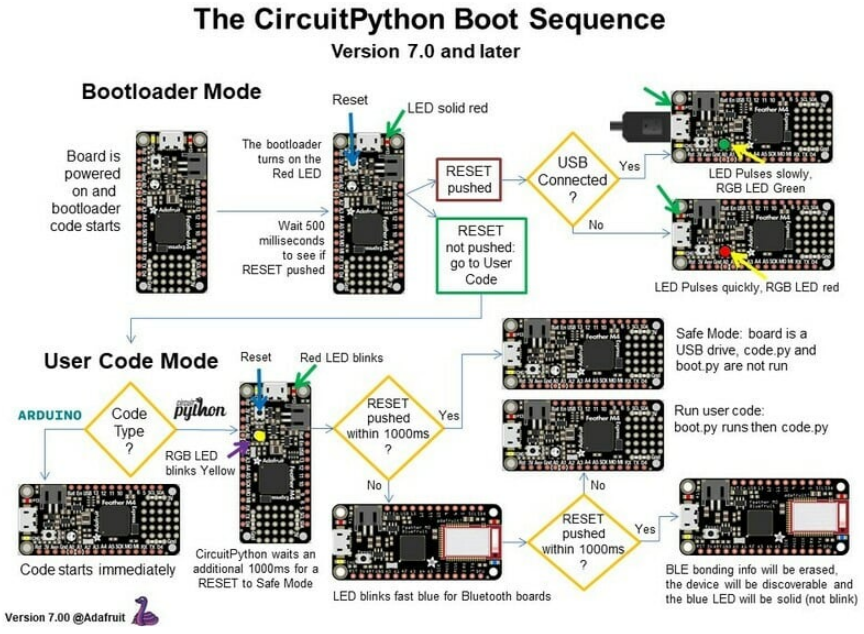
The status LED blinks were changed in CircuitPython 7.0.0 in order to save battery power and simplify the blinks. These blink patterns will occur on single color LEDs when the board does not have any RGB LEDs. Speed and blink count also vary for this reason.

On start up, the LED will blink **YELLOW** multiple times for 1 second. Pressing the RESET button (or on Espressif, the BOOT button) during this time will restart the board and then enter safe mode. On Bluetooth capable boards, after the yellow blinks, there will be a set of faster blue blinks. Pressing reset during the **BLUE** blinks will clear Bluetooth information and start the device in discoverable mode, so it can be used with a BLE code editor.

Once started, CircuitPython will blink a pattern every 5 seconds when no user code is running to indicate why the code stopped:

- 1 **GREEN** blink: Code finished without error.
- 2 **RED** blinks: Code ended due to an exception. Check the serial console for details.
- 3 **YELLOW** blinks: CircuitPython is in safe mode. No user code was run. Check the serial console for safe mode reason.

When in the REPL, CircuitPython will set the status LED to **WHITE**. You can change the LED color from the REPL. The status indicator will not persist on non-NeoPixel or DotStar LEDs.



CircuitPython 6.3.0 and earlier

Here's what the colors and blinking mean:

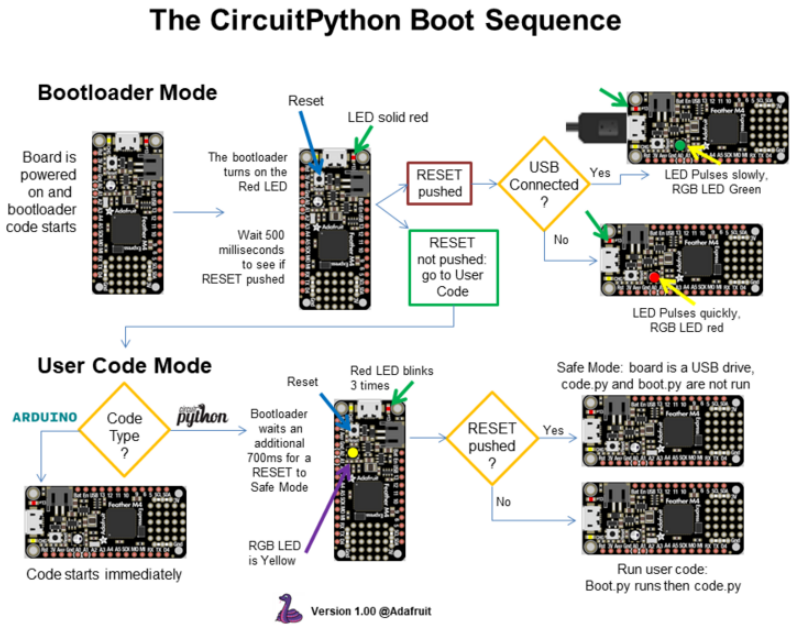
- steady **GREEN**: **code.py** (or **code.txt**, **main.py**, or **main.txt**) is running

- pulsing **GREEN**: **code.py** (etc.) has finished or does not exist
- steady **YELLOW** at start up: (4.0.0-alpha.5 and newer) CircuitPython is waiting for a reset to indicate that it should start in safe mode
- pulsing **YELLOW**: Circuit Python is in safe mode: it crashed and restarted
- steady **WHITE**: REPL is running
- steady **BLUE**: **boot.py** is running

Colors with multiple flashes following indicate a Python exception and then indicate the line number of the error. The color of the first flash indicates the type of error:

- **GREEN**: IndentationError
- **CYAN**: SyntaxError
- **WHITE**: NameError
- **ORANGE**: OSError
- **PURPLE**: ValueError
- **YELLOW**: other error

These are followed by flashes indicating the line number, including place value. **WHITE** flashes are thousands' place, **BLUE** are hundreds' place, **YELLOW** are tens' place, and **CYAN** are one's place. So for example, an error on line 32 would flash **YELLOW** three times and then **CYAN** two times. Zeroes are indicated by an extra-long dark gap.



Serial console showing `ValueError: Incompatible .mpy file`

This error occurs when importing a module that is stored as a **.mpy** binary file that was generated by a different version of CircuitPython than the one its being loaded into. In particular, the mpy binary format changed between CircuitPython versions 6.x and 7.x, 2.x and 3.x, and 1.x and 2.x.

So, for instance, if you upgraded to CircuitPython 7.x from 6.x you'll need to download a newer version of the library that triggered the error on `import`. All libraries are available in the Adafruit bundle (https://adafru.it/y8E).

CIRCUITPY Drive Issues

You may find that you can no longer save files to your **CIRCUITPY** drive. You may find that your **CIRCUITPY** stops showing up in your file explorer, or shows up as **NO_NAME**. These are indicators that your filesystem has issues. When the **CIRCUITPY** disk is not safely ejected before being reset by the button or being disconnected from USB, it may corrupt the flash drive. It can happen on Windows, Mac or Linux, though it is more common on Windows.

Be aware, if you have used Arduino to program your board, CircuitPython is no longer able to provide the USB services. You will need to reload CircuitPython to resolve this situation.

The easiest first step is to reload CircuitPython. Double-tap reset on the board so you get a **boardnameBOOT** drive rather than a **CIRCUITPY** drive, and copy the latest version of CircuitPython (**.uf2**) back to the board. This may restore **CIRCUITPY** functionality.

If reloading CircuitPython does not resolve your issue, the next step is to try putting the board into safe mode.

Safe Mode

Whether you've run into a situation where you can no longer edit your **code.py** on your **CIRCUITPY** drive, your board has gotten into a state where **CIRCUITPY** is read-only, or you have turned off the **CIRCUITPY** drive altogether, safe mode can help.

Safe mode in CircuitPython does not run any user code on startup, and disables auto-reload. This means a few things. First, safe mode bypasses any code in **boot.py** (where you can set **CIRCUITPY** read-only or turn it off completely). Second, it does not run the code in **code.py**. And finally, it does not automatically soft-reload when data is written to the **CIRCUITPY** drive.

Therefore, whatever you may have done to put your board in a non-interactive state, safe mode gives you the opportunity to correct it without losing all of the data on the **CIRCUITPY** drive.

Entering Safe Mode in CircuitPython 7.x and Later

You can enter safe by pressing reset during the right time when the board boots. Immediately after the board starts up or resets, it waits one second. On some boards, the onboard status LED will blink yellow during that time. If you press reset during that one second period, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a "slow" double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

Entering Safe Mode in CircuitPython 6.x

You can enter safe by pressing reset during the right time when the board boots.. Immediately after the board starts up or resets, it waits 700ms. On some boards, the onboard status LED (highlighted in green above) will turn solid yellow during this time. If you press reset during that 700ms, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a slow double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

In Safe Mode

Once you've entered safe mode successfully in CircuitPython 6.x, the LED will pulse yellow.

If you successfully enter safe mode on CircuitPython 7.x, the LED will intermittently blink yellow three times.

If you connect to the serial console, you'll find the following message.

```
Auto-reload is off.
Running in safe mode! Not running saved code.

CircuitPython is in safe mode because you pressed the reset button during boot.
```

```
Press again to exit safe mode.

Press any key to enter the REPL. Use CTRL-D to reload.
```

You can now edit the contents of the **CIRCUITPY** drive. Remember, your code will not run until you press the reset button, or unplug and plug in your board, to get out of safe mode.

At this point, you'll want to remove any user code in **code.py** and, if present, the **boot.py** file from **CIRCUITPY**. Once removed, tap the reset button, or unplug and plug in your board, to restart CircuitPython. This will restart the board and may resolve your drive issues. If resolved, you can begin coding again as usual.

If safe mode does not resolve your issue, the board must be completely erased and CircuitPython must be reloaded onto the board.

You WILL lose everything on the board when you complete the following steps. If possible, make a copy of your code before continuing.

To erase CIRCUITPY: `storage.erase_filesystem()`

CircuitPython includes a built-in function to erase and reformat the filesystem. If you have a version of CircuitPython older than 2.3.0 on your board, you can update to the newest version (https://adafru.it/Amd) to do this.

1. Connect to the CircuitPython REPL (https://adafru.it/Bec) using Mu or a terminal program.
2. Type the following into the REPL:

```
&gt;&gt;&gt; import storage
&gt;&gt;&gt; storage.erase_filesystem()
```

CIRCUITPY will be erased and reformatted, and your board will restart. That's it!

Erase CIRCUITPY Without Access to the REPL

If you can't access the REPL, or you're running a version of CircuitPython previous to 2.3.0 and you don't want to upgrade, there are options available for some specific boards.

The options listed below are considered to be the "old way" of erasing your board. The method shown above using the REPL is highly recommended as the best method for erasing your board.

If at all possible, it is recommended to use the REPL to erase your CIRCUITPY drive. The REPL method is explained above.

For the specific boards listed below:

If the board you are trying to erase is listed below, follow the steps to use the file to erase your board.

1. Download the correct erase file:

Circuit Playground Express
https://adafru.it/AdI

Feather M0 Express
https://adafru.it/AdJ

Feather M4 Express
https://adafru.it/EVK

Metro M0 Express
https://adafru.it/AdK

Metro M4 Express QSPI Eraser
https://adafru.it/EoM

Trellis M4 Express (QSPI)
https://adafru.it/DjD

Grand Central M4 Express (QSPI)
https://adafru.it/DBA

PyPortal M4 Express (QSPI)
https://adafru.it/Eca

Circuit Playground Bluefruit (QSPI)

https://adafru.it/Gnc

Monster M4SK (QSPI)

https://adafru.it/GAN

PyBadge/PyGamer QSPI Eraser.UF2

https://adafru.it/GAO

CLUE_Flash_Erase.UF2

https://adafru.it/Jat

Matrix_Portal_M4_(QSPI).UF2

https://adafru.it/Q5B

RP2040 boards (flash_nuke.uf2)

https://adafru.it/18ed

2. Double-click the reset button on the board to bring up the **boardnameBOOT** drive.
3. Drag the erase **.uf2** file to the **boardnameBOOT** drive.
4. The status LED will turn yellow or blue, indicating the erase has started.
5. After approximately 15 seconds, the status LED will light up green. On the NeoTrellis M4 this is the first NeoPixel on the grid
6. Double-click the reset button on the board to bring up the **boardnameBOOT** drive.
7. Drag the appropriate latest release of CircuitPython (https://adafru.it/Em8) **.uf2** file to the **boardnameBOOT** drive.

It should reboot automatically and you should see **CIRCUITPY** in your file explorer again.

If the LED flashes red during step 5, it means the erase has failed. Repeat the steps starting with 2.

If you haven't already downloaded the latest release of CircuitPython for your board, check out the installation page (https://adafru.it/Amd). You'll also need to load your code and reinstall your libraries!

For SAMD21 non-Express boards that have a UF2 bootloader:

Any SAMD21-based microcontroller that does not have external flash available is considered a SAMD21 non-Express board. Non-Express boards that have a UF2 bootloader include Trinket M0, GEMMA M0, QT Py M0, and the SAMD21-based Trinkey boards.

If you are trying to erase a SAMD21 non-Express board, follow these steps to erase your board.

1. Download the erase file:

SAMD21 non-Express Boards

https://adafru.it/VB-

2. Double-click the reset button on the board to bring up the **boardnameBOOT** drive.
3. Drag the erase **.uf2** file to the **boardnameBOOT** drive.
4. The boot LED will start flashing again, and the **boardnameBOOT** drive will reappear.
5. Drag the appropriate latest release CircuitPython (https://adafru.it/Em8) **.uf2** file to the boardnameBOOT drive.

It should reboot automatically and you should see **CIRCUITPY** in your file explorer again.

If you haven't already downloaded the latest release of CircuitPython for your board, check out the installation page (https://adafru.it/Amd) YYou'll also need to load your code and reinstall your libraries!

For SAMD21 non-Express boards that do not have a UF2 bootloader:

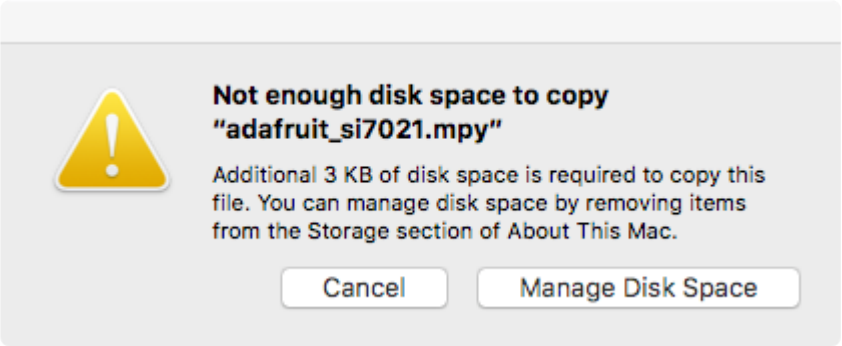
Any SAMD21-based microcontroller that does not have external flash available is considered a SAMD21 non-Express board. Non-Express boards that do **not** have a UF2 bootloader include the Feather M0 Basic Proto, Feather Adalogger, or the Arduino Zero.

If you are trying to erase a non-Express board that does not have a UF2 bootloader, follow these directions to reload CircuitPython using `bossac` (https://adafru.it/Bed), which will erase and re-create **CIRCUITPY**.

Running Out of File Space on SAMD21 Non-Express Boards

Any SAMD21-based microcontroller that does not have external flash available is considered a SAMD21 non-Express board. This includes boards like the Trinket M0, GEMMA M0, QT Py M0, and the SAMD21-based Trinkey boards.

The file system on the board is very tiny. (Smaller than an ancient floppy disk.) So, its likely you'll run out of space but don't panic! There are a number of ways to free up space.



Delete something!

The simplest way of freeing up space is to delete files from the drive. Perhaps there are libraries in the **lib** folder that you aren't using anymore or test code that isn't in use. Don't delete the **lib** folder completely, though, just remove what you don't need.

The board ships with the Windows 7 serial driver too! Feel free to delete that if you don't need it or have already installed it. It's ~12KiB or so.

Use tabs

One unique feature of Python is that the indentation of code matters. Usually the recommendation is to indent code with four spaces for every indent. In general, that is recommended too. **However**, one trick to storing more human-readable code is to use a single tab character for indentation. This approach uses 1/4 of the space for indentation and can be significant when you're counting bytes.

On MacOS?

MacOS loves to generate hidden files. Luckily you can disable some of the extra hidden files that macOS adds by running a few commands to disable search indexing and create zero byte placeholders. Follow the steps below to maximize the amount of space available on macOS.

Prevent & Remove MacOS Hidden Files

First find the volume name for your board. With the board plugged in run this command in a terminal to list all the volumes:

```
ls -l /Volumes
```

Look for a volume with a name like **CIRCUITPY** (the default for CircuitPython). The full path to the volume is the **/Volumes/CIRCUITPY** path.

Now follow the steps from this question (https://adafru.it/u1c) to run these terminal commands that stop hidden files from being created on the board:

```
mdutil -i off /Volumes/CIRCUITPY
cd /Volumes/CIRCUITPY
rm -rf .{,_.}{fseventsd,Spotlight-V*,Trashes}
mkdir .fseventsd
touch .fseventsd/no_log .metadata_never_index .Trashes
cd -
```

Replace **/Volumes/CIRCUITPY** in the commands above with the full path to your board's volume if it's different. At this point all the hidden files should be cleared from the board and some hidden files will be prevented from being created.

Alternatively, with CircuitPython 4.x and above, the special files and folders mentioned above will be created automatically if you erase and reformat the filesystem. **WARNING: Save your files first!** Do this in the REPL:

```
>>> import storage
>>> storage.erase_filesystem()
```

However there are still some cases where hidden files will be created by MacOS. In particular if you copy a file that was downloaded from the internet it will have special metadata that MacOS stores as a hidden file. Luckily you can run a copy command from the terminal to copy files **without** this hidden metadata file. See the steps below.

Copy Files on MacOS Without Creating Hidden Files

Once you've disabled and removed hidden files with the above commands on macOS you need to be careful to copy files to the board with a special command that prevents future hidden files from being created. Unfortunately you **cannot** use drag and drop copy in Finder because it will still create these hidden extended attribute files in some cases (for files downloaded from the internet, like Adafruit's modules).

To copy a file or folder use the **-X** option for the **cp** command in a terminal. For example to copy a **file_name.mpy** file to the board use a command like:

```
cp -X file_name.mpy /Volumes/CIRCUITPY
```

(Replace **file_name.mpy** with the name of the file you want to copy.)

Or to copy a folder and all of the files and folders contained within, use a command like:

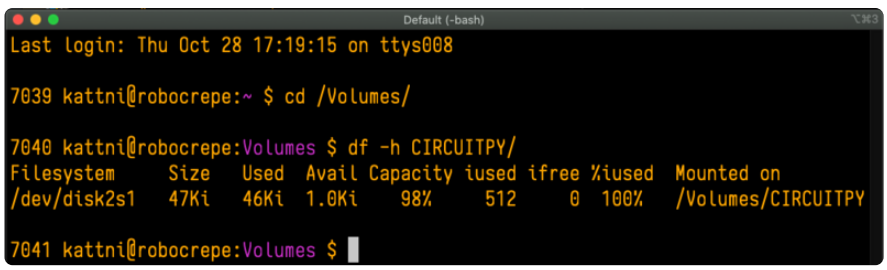
```
cp -rX folder_to_copy /Volumes/CIRCUITPY
```

If you are copying to the **lib** folder, or another folder, make sure it exists before copying.

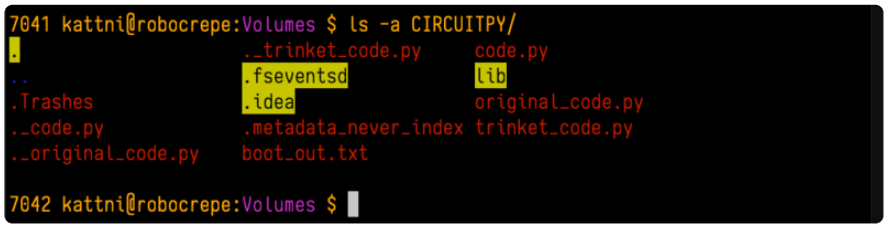
```
# if lib does not exist, you'll create a file named lib !
cp -X file_name.mpy /Volumes/CIRCUITPY/lib
# This is safer, and will complain if a lib folder does not exist.
cp -X file_name.mpy /Volumes/CIRCUITPY/lib/
```

Other MacOS Space-Saving Tips

If you'd like to see the amount of space used on the drive and manually delete hidden files here's how to do so. First, move into the **Volumes**/ directory with `cd /Volumes/`, and then list the amount of space used on the **CIRCUITPY** drive with the `df` command.



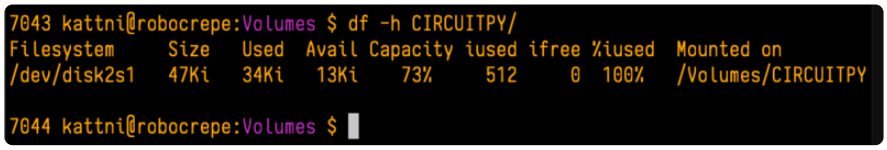
That's not very much space left! The next step is to show a list of the files currently on the **CIRCUITPY** drive, including the hidden files, using the `ls` command. You cannot use Finder to do this, you must do it via command line!



There are a few of the hidden files that MacOS loves to generate, all of which begin with a **._** before the file name. Remove the **._** files using the `rm` command. You can remove them all once by running `rm CIRCUITPY/._*`. The `*` acts as a wildcard to apply the command to everything that begins with **._** at the same time.



Finally, you can run `df` again to see the current space used.



Nice! You have 12Ki more than before! This space can now be used for libraries and code!

Device Locked Up or Boot Looping

In rare cases, it may happen that something in your **code.py** or **boot.py** files causes the device to get locked up, or even go into a boot loop. A boot loop occurs when the board reboots repeatedly and never fully loads. These are not caused by your everyday Python exceptions, typically it's the result of a deeper problem within CircuitPython. In this situation, it can be difficult to recover your device if **CIRCUITPY** is not allowing you to modify the **code.py** or **boot.py** files. Safe mode is one recovery option. When the device boots up in safe mode it will not run the **code.py** or **boot.py** scripts, but will still connect the **CIRCUITPY** drive so that you can remove or modify those files as needed.

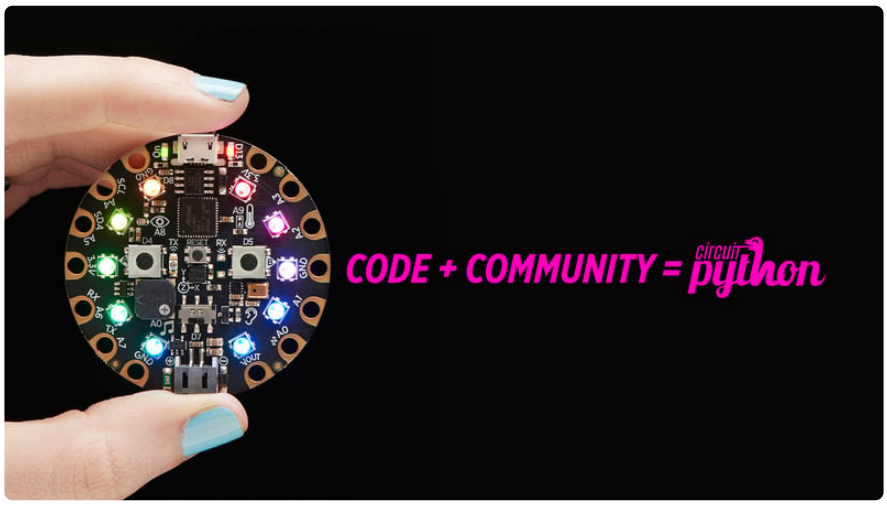
The method used to manually enter safe mode can be different for different devices. It is also very similar to the method used for getting into bootloader mode, which is a different thing. So it can take a few tries to get the timing right. If you end up in bootloader mode, no problem, you can try again without needing to do anything else.

For most devices:
Press the reset button, and then when the RGB status LED blinks yellow, press the reset button again. Since your reaction time may not be that fast, try a "slow" double click, to catch the yellow LED on the second click.

For ESP32-S2 based devices:
Press and release the reset button, then press and release the boot button about 3/4 of a second later.

Refer to the diagrams above for boot sequence details.

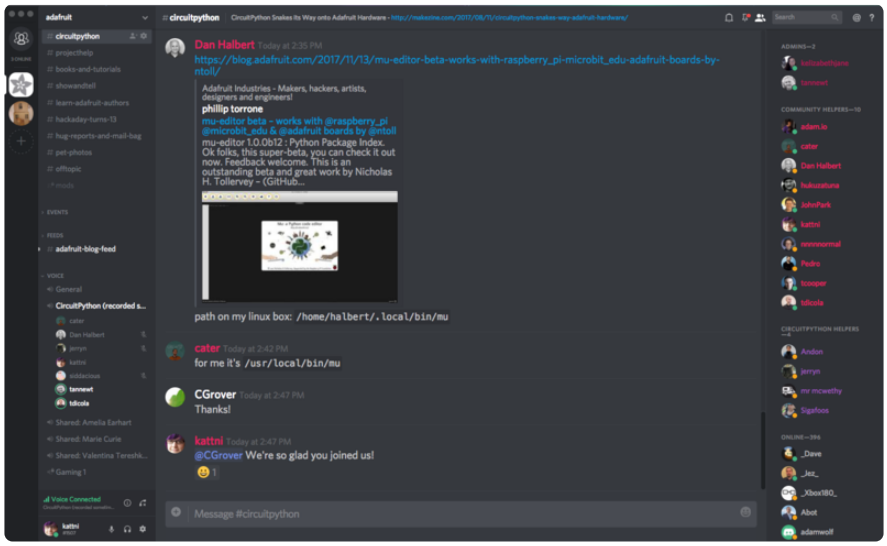
Welcome to the Community!



CircuitPython is a programming language that's super simple to get started with and great for learning. It runs on microcontrollers and works out of the box. You can plug it in and get started with any text editor. The best part? CircuitPython comes with an amazing, supportive community.

Everyone is welcome! CircuitPython is Open Source. This means it's available for anyone to use, edit, copy and improve upon. This also means CircuitPython becomes better because of you being a part of it. Whether this is your first microcontroller board or you're a seasoned software engineer, you have something important to offer the Adafruit CircuitPython community. This page highlights some of the many ways you can be a part of it!

Adafruit Discord



The Adafruit Discord server is the best place to start. Discord is where the community comes together to volunteer and provide live support of all kinds. From general discussion to detailed problem solving, and everything in between, Discord is a digital maker space with makers from around the world.

There are many different channels so you can choose the one best suited to your needs. Each channel is shown on Discord as "#channelname". There's the #help-with-projects channel for assistance with your current project or help coming up with ideas for your next one. There's the #show-and-tell channel for showing off your newest creation. Don't be afraid to ask a question in any channel! If you're unsure, #general is a great place to start. If another channel is more likely to provide you with a better answer, someone will guide you.

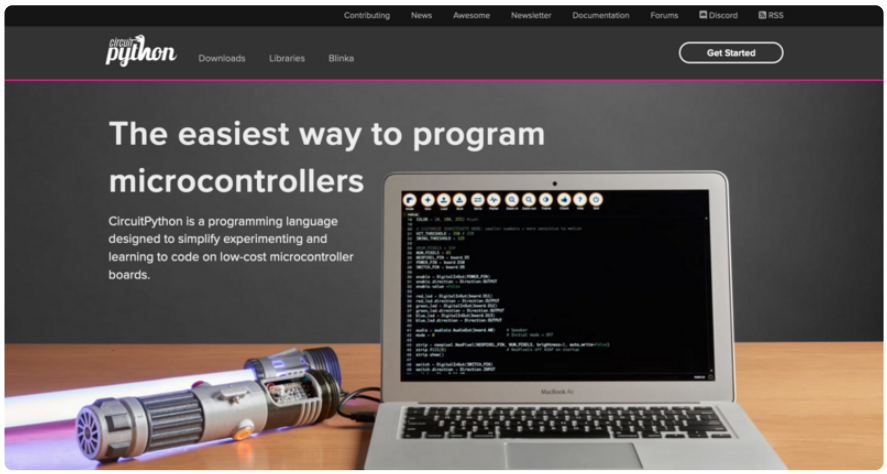
The help with CircuitPython channel is where to go with your CircuitPython questions. #help-with-circuitpython is there for new users and developers alike so feel free to ask a question or post a comment! Everyone of any experience level is welcome to join in on the conversation. Your contributions are important! The #circuitpython-dev channel is available for development discussions as well.

The easiest way to contribute to the community is to assist others on Discord. Supporting others doesn't always mean answering questions. Join in celebrating successes! Celebrate your mistakes! Sometimes just hearing that someone else has gone through a similar struggle can be enough to keep a maker moving forward.

The Adafruit Discord is the 24x7x365 hackerspace that you can bring your granddaughter to.

Visit https://adafru.it/discord ()to sign up for Discord. Everyone is looking forward to meeting you!

CircuitPython.org



Beyond the Adafruit Learn System, which you are viewing right now, the best place to find information about CircuitPython is circuitpython.org (https://adafru.it/KJD). Everything you need to get started with your new microcontroller and beyond is available. You can do things like download CircuitPython for your microcontroller (https://adafru.it/Em8) or download the latest CircuitPython Library bundle (https://adafru.it/ENC), or check out which single board computers support Blinka (https://adafru.it/EA8). You can also get to various other CircuitPython related things like Awesome CircuitPython or the Python for Microcontrollers newsletter. This is all incredibly useful, but it isn't necessarily community related. So why is it included here? The Contributing page (https://adafru.it/VD7).



Contributing

If you'd like to contribute to the CircuitPython project, the CircuitPython libraries are a great way to begin. This page is updated with daily status information from the CircuitPython libraries, including open pull requests, open issues and library infrastructure issues.

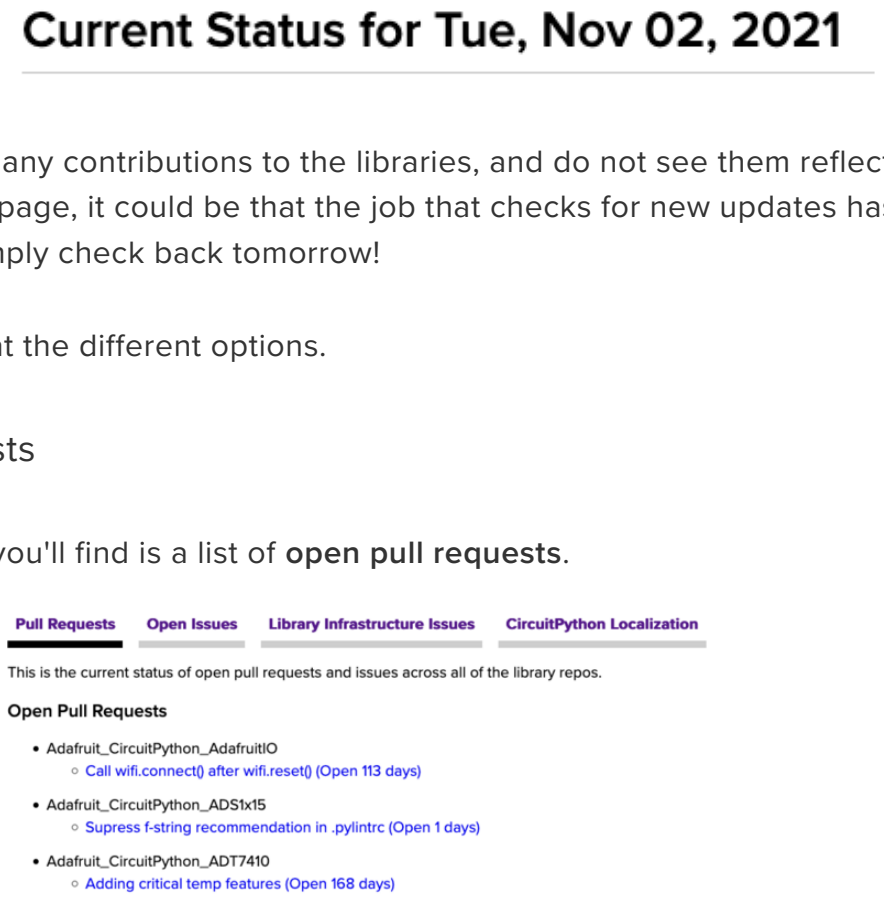
Do you write a language other than English? Another great way to contribute to the project is to contribute new localizations (translations) of CircuitPython, or update current localizations, using Weblate.

If this is your first time contributing, or you'd like to see our recommended contribution workflow, we have a guide on Contributing to CircuitPython with Git and Github. You can also find us in the #circuitpython channel on the Adafruit Discord.

Have an idea for a new driver or library? File an issue on the CircuitPython repo!

CircuitPython itself is written in C. However, all of the Adafruit CircuitPython libraries are written in Python. If you're interested in contributing to CircuitPython on the Python side of things, check out circuitpython.org/contributing (https://adafru.it/VD7). You'll find information pertaining to every Adafruit CircuitPython library GitHub repository, giving you the opportunity to join the community by finding a contributing option that works for you.

Note the date on the page next to **Current Status for**:



If you submit any contributions to the libraries, and do not see them reflected on the Contributing page, it could be that the job that checks for new updates hasn't yet run for today. Simply check back tomorrow!

Now, a look at the different options.

Pull Requests

The first tab you'll find is a list of **open pull requests**.

GitHub pull requests, or PRs, are opened when folks have added something to an Adafruit CircuitPython library GitHub repo, and are asking for Adafruit to add, or merge, their changes into the main library code. For PRs to be merged, they must first be reviewed. Reviewing is a great way to contribute! Take a look at the list of open pull requests, and pick one that interests you. If you have the hardware, you can test code changes. If you don't, you can still check the code updates for syntax. In the case of documentation updates, you can verify the information, or check it for spelling and grammar. Once you've checked out the update, you can leave a comment letting us know that you took a look. Once you've done that for a while, and you're more comfortable with it, you can consider joining the CircuitPythonLibrarians review team. The more reviewers we have, the more authors we can support. Reviewing is a crucial part of an open source ecosystem, CircuitPython included.

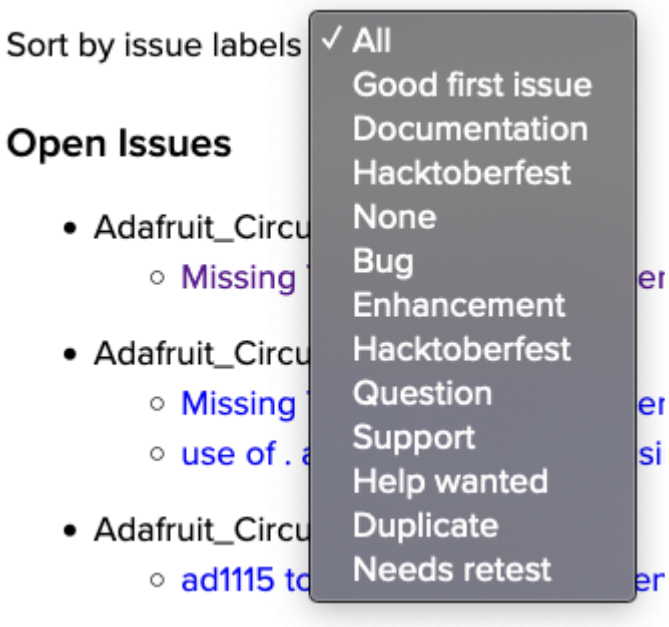
Open Issues

The second tab you'll find is a list of **open issues**.



GitHub issues are filed for a number of reasons, including when there is a bug in the library or example code, or when someone wants to make a feature request. Issues are a great way to find an opportunity to contribute directly to the libraries by updating code or documentation. If you're interested in contributing code or documentation, take a look at the open issues and find one that interests you.

If you're not sure where to start, you can search the issues by label. Labels are applied to issues to make the goal easier to identify at a first glance, or to indicate the difficulty level of the issue. Click on the dropdown next to "Sort by issue labels" to see the list of available labels, and click on one to choose it.



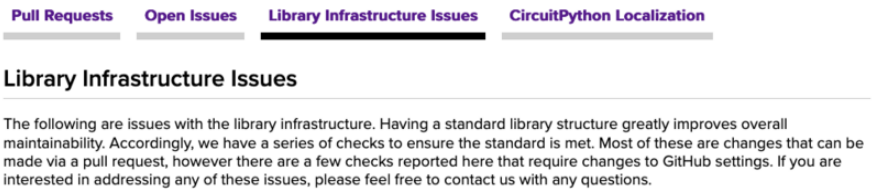
If you're new to everything, new to contributing to open source, or new to contributing to the CircuitPython project, you can choose "Good first issue". Issues with that label are well defined, with a finite scope, and are intended to be easy for someone new to figure out.

If you're looking for something a little more complicated, consider "Bug" or "Enhancement". The Bug label is applied to issues that pertain to problems or failures found in the library. The Enhancement label is applied to feature requests.

Don't let the process intimidate you. If you're new to Git and GitHub, there is a guide (https://adafru.it/Dkh) to walk you through the entire process. As well, there are always folks available on Discord () to answer questions.

Library Infrastructure Issues

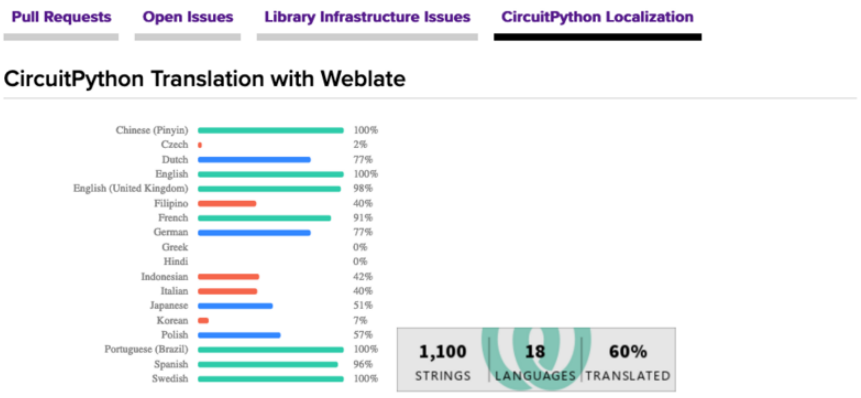
The third tab you'll find is a list of **library infrastructure issues**.



This section is generated by a script that runs checks on the libraries, and then reports back where there may be issues. It is made up of a list of subsections each containing links to the repositories that are experiencing that particular issue. This page is available mostly for internal use, but you may find some opportunities to contribute on this page. If there's an issue listed that sounds like something you could help with, mention it on Discord, or file an issue on GitHub indicating you're working to resolve that issue. Others can reply either way to let you know what the scope of it might be, and help you resolve it if necessary.

CircuitPython Localization

The fourth tab you'll find is the **CircuitPython Localization** tab.

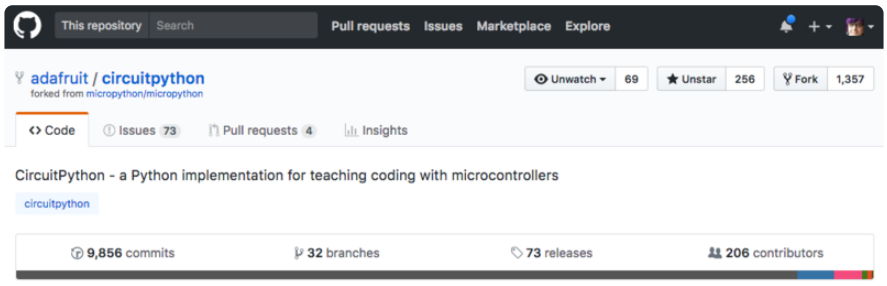


If you speak another language, you can help translate CircuitPython! The translations apply to informational and error messages that are within the CircuitPython core. It means that folks who do not speak English have the opportunity to have these messages shown to them in their own language when using CircuitPython. This is incredibly important to provide the best experience possible for all users.

CircuitPython uses Weblate to translate, which makes it much simpler to contribute translations. You will still need to know some CircuitPython-specific practices and a few basics about coding strings, but as with any CircuitPython contributions, folks are there to help.

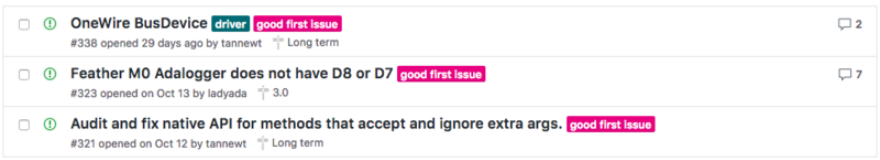
Regardless of your skill level, or how you want to contribute to the CircuitPython project, there is an opportunity available. The Contributing page (https://adafru.it/VD7) is an excellent place to start!

Adafruit GitHub



Whether you're just beginning or are life-long programmer who would like to contribute, there are ways for everyone to be a part of the CircuitPython project. The CircuitPython core is written in C. The libraries are written in Python. GitHub is the best source of ways to contribute to the CircuitPython core (https://adafru.it/tB7), and the CircuitPython libraries (https://adafru.it/VFv). If you need an account, visit https://github.com/ (https://adafru.it/d6C) and sign up.

If you're new to GitHub or programming in general, there are great opportunities for you. For the CircuitPython core, head over to the CircuitPython repository on GitHub, click on "Issues (https://adafru.it/tBb)", and you'll find a list that includes issues labeled "good first issue (https://adafru.it/188e)". For the libraries, head over to the Contributing page Issues list (https://adafru.it/VFv), and use the drop down menu to search for "good first issue (https://adafru.it/VFw)". These issues are things that have been identified as something that someone with any level of experience can help with. These issues include options like updating documentation, providing feedback, and fixing simple bugs. If you need help getting started with GitHub, there is an excellent guide on Contributing to CircuitPython with Git and GitHub (https://adafru.it/Dkh).



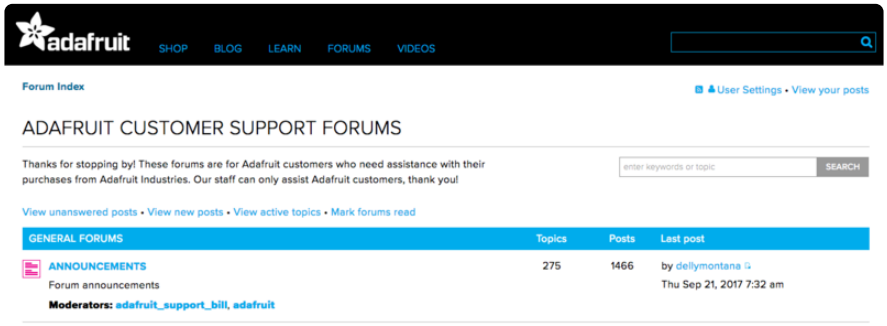
Already experienced and looking for a challenge? Checkout the rest of either issues list and you'll find plenty of ways to contribute. You'll find all sorts of things, from new driver requests, to library bugs, to core module updates. There's plenty of opportunities for everyone at any level!

When working with or using CircuitPython or the CircuitPython libraries, you may find problems. If you find a bug, that's great! The team loves bugs! Posting a detailed issue to GitHub is an invaluable way to contribute to improving CircuitPython. For CircuitPython itself, file an issue here (https://adafru.it/tBb). For the libraries, file an issue on the specific library repository on GitHub. Be sure to include the steps to replicate the issue as well as any other information you think is relevant. The more detail, the better!

Testing new software is easy and incredibly helpful. Simply load the newest version of CircuitPython or a library onto your CircuitPython hardware, and use it. Let us know about any problems you find by posting a new issue to GitHub. Software testing on both stable and unstable releases is a very important part of contributing CircuitPython. The developers can't possibly find all the problems themselves! They need your help to make CircuitPython even better.

On GitHub, you can submit feature requests, provide feedback, report problems and much more. If you have questions, remember that Discord and the Forums are both there for help!

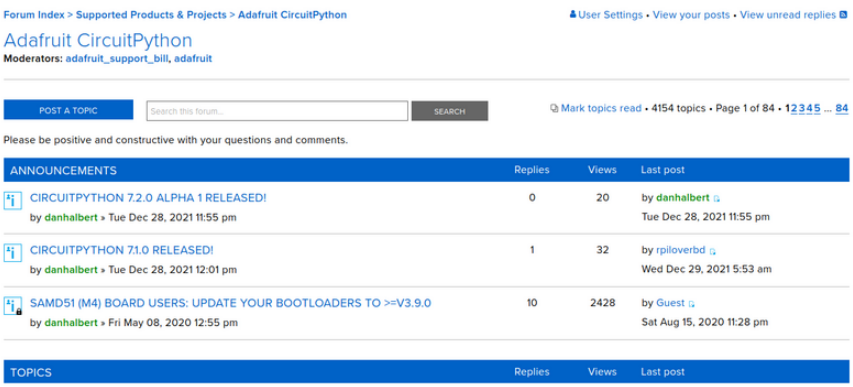
Adafruit Forums



The Adafruit Forums (https://adafru.it/jlf) are the perfect place for support. Adafruit has wonderful paid support folks to answer any questions you may have. Whether your hardware is giving you issues or your code doesn't seem to be working, the forums are always there for you to ask. You need an Adafruit account to post to the forums. You can use the same account you use to order from Adafruit.

While Discord may provide you with quicker responses than the forums, the forums are a more reliable source of information. If you want to be certain you're getting an Adafruit-supported answer, the forums are the best place to be.

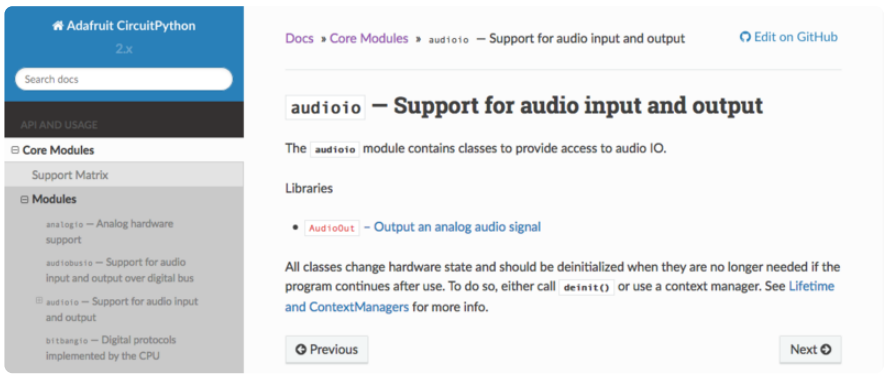
There are forum categories that cover all kinds of topics, including everything Adafruit. The Adafruit CircuitPython (https://adafru.it/xXA) category under "Supported Products & Projects" is the best place to post your CircuitPython questions.



Be sure to include the steps you took to get to where you are. If it involves wiring, post a picture! If your code is giving you trouble, include your code in your post! These are great ways to make sure that there's enough information to help you with your issue.

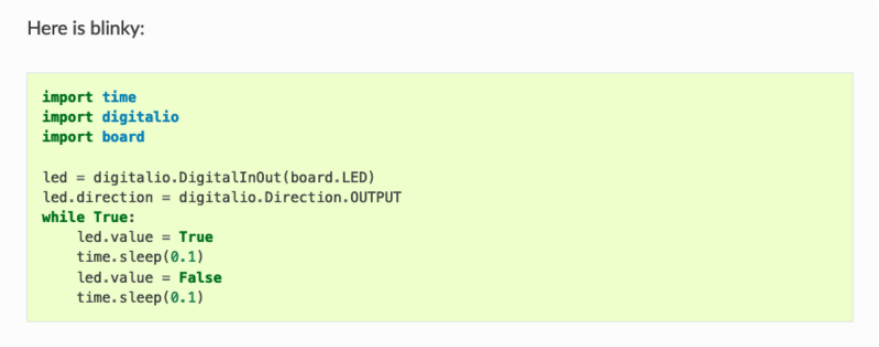
You might think you're just getting started, but you definitely know something that someone else doesn't. The great thing about the forums is that you can help others too! Everyone is welcome and encouraged to provide constructive feedback to any of the posted questions. This is an excellent way to contribute to the community and share your knowledge!

Read the Docs



Read the Docs (https://adafru.it/Beg) is a an excellent resource for a more detailed look at the CircuitPython core and the CircuitPython libraries. This is where you'll find

things like API documentation and example code. For an in depth look at viewing and understanding Read the Docs, check out the CircuitPython Documentation (https://adafru.it/VFx) page!



CircuitPython Essentials



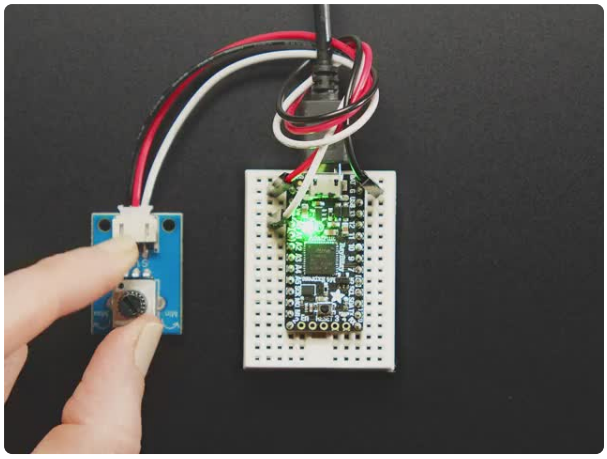
You've been introduced to CircuitPython, and worked through getting everything set up. What's next? CircuitPython Essentials!

There are a number of core modules built into CircuitPython, which can be used along side the many CircuitPython libraries available. The following pages demonstrate some of these modules. Each page presents a different concept including a code example with an explanation. All of the examples are designed to work with your microcontroller board.

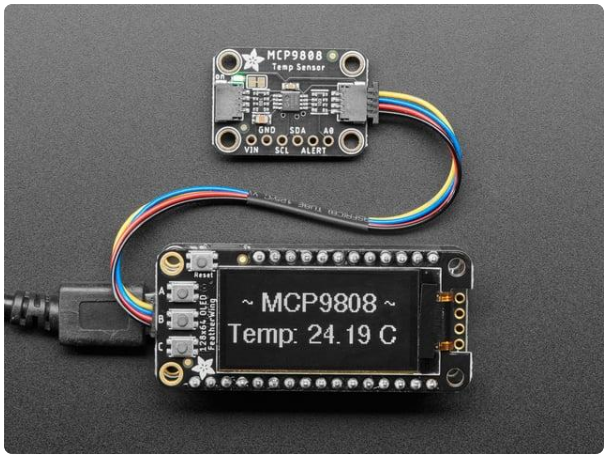
Time to get started learning the CircuitPython essentials!

Some examples require external components, such as switches or sensors. You'll find wiring diagrams where applicable to show you how to wire up the necessary components to work with each example.

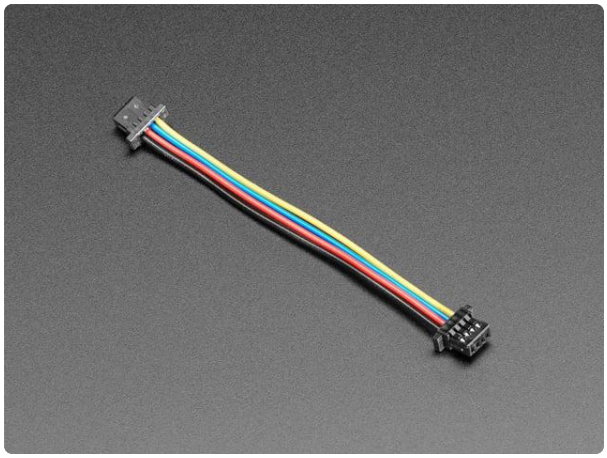
The following components are needed to complete all of the examples:



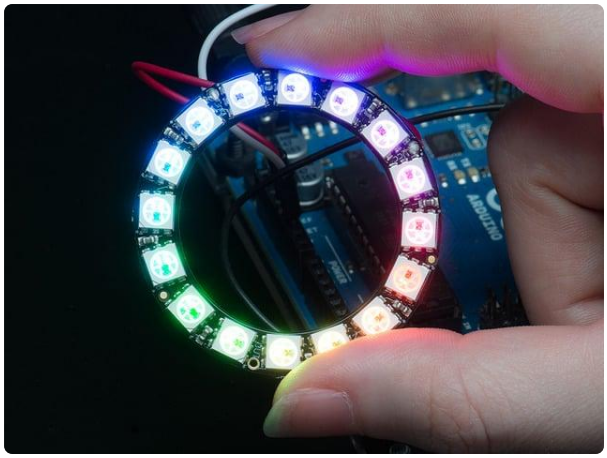
[STEMMA Wired Potentiometer Breakout Board - 10K ohm Linear]
For the easiest way possible to measure twists, turn to this STEMMA potentiometer breakout (ha!). This plug-n-play pot comes with a JST-PH 2mm connector and a matching
https://www.adafruit.com/product/4493



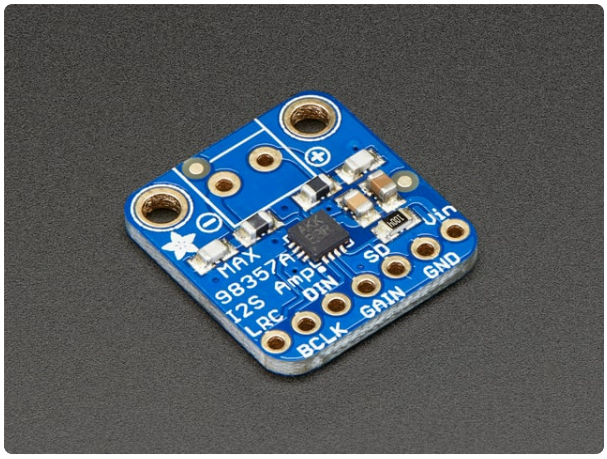
[Adafruit MCP9808 High Accuracy I2C Temperature Sensor Breakout]
The MCP9808 digital temperature sensor is one of the more accurate/precise we've ever seen, with a typical accuracy of ±0.25°C over the sensor's -40°C to...
https://www.adafruit.com/product/5027



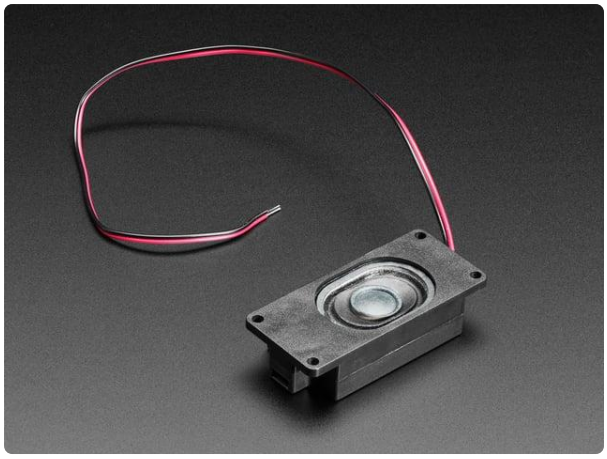
[STEMMA QT / Qwiic JST SH 4-Pin Cable - 50mm Long]
This 4-wire cable is 50mm / 1.9" long and fitted with JST SH female 4-pin connectors on both ends. Compared with the chunkier JST PH these are 1mm pitch instead of 2mm, but...
https://www.adafruit.com/product/4399



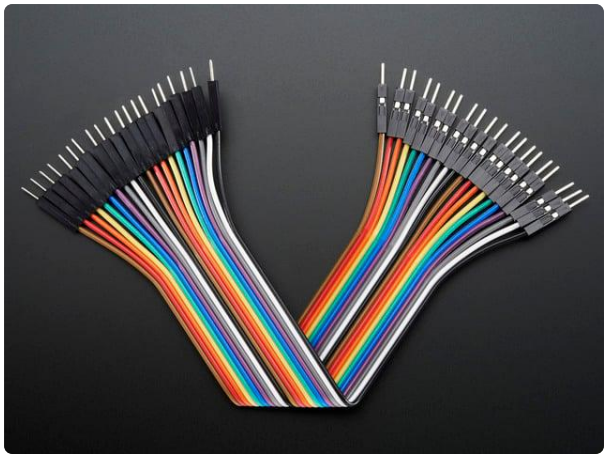
[NeoPixel Ring - 16 x 5050 RGB LED with Integrated Drivers]
Round and round and round they go! 16 ultra bright smart LED NeoPixels are arranged in a circle with 1.75" (44.5mm) outer diameter. The rings are 'chainable' - connect the...
https://www.adafruit.com/product/1463



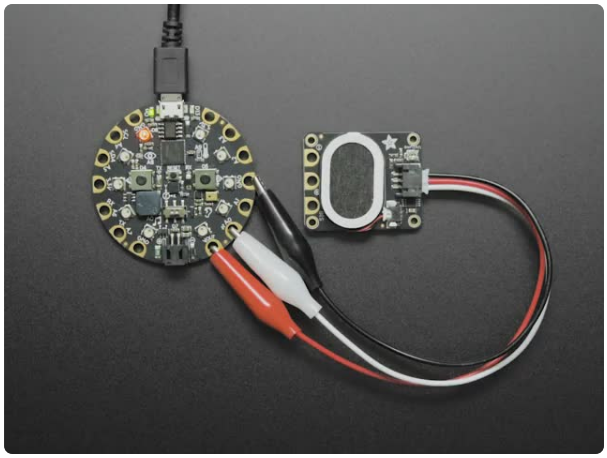
[Adafruit I2S 3W Class D Amplifier Breakout - MAX98357A]
Listen to this good news - we now have an all in one digital audio amp breakout board that works incredibly well with the
https://www.adafruit.com/product/3006



[Mono Enclosed Speaker with Plain Wires - 3W 4 Ohm]
Listen up! This single 2.8" x 1.2" speaker is the perfect addition to any audio project where you need 4 ohm impedance and 3W or less of power. We...
https://www.adafruit.com/product/4445



[Premium Male/Male Jumper Wires - 20 x 6" (150mm)]
These Male/Male Jumper Wires are handy for making wire harnesses or jumpering between headers on PCB's. These premium jumper wires are 6" (150mm) long and come in a...
https://www.adafruit.com/product/1957



[Adafruit STEMMA Speaker - Plug and Play Audio Amplifier]
Hey, have you heard the good news? With Adafruit STEMMA boards you can easily and safely plug sensors and devices together, like this Adafruit STEMMA Speaker - Plug and Play...
https://www.adafruit.com/product/3885



STEMMA JST PH 2mm 3-Pin to Male Header Cable - 200mm
This cable will let you turn a JST PH 3-pin cable port into 3 individual wires with high-quality 0.1" male header plugs on the end. We're carrying these to match up with our...
https://www.adafruit.com/product/3893

Blink

In learning any programming language, you often begin with some sort of `Hello, World!` program. In CircuitPython, Hello, World! is blinking an LED. Blink is one of the simplest programs in CircuitPython. It involves three built-in modules, two lines of set up, and a short loop. Despite its simplicity, it shows you many of the basic concepts needed for most CircuitPython programs, and provides a solid basis for more complex projects. Time to get blinky!

LED Location

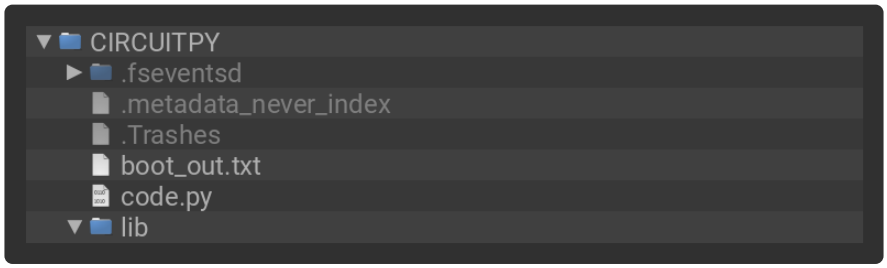


The **red LED** (highlighted in red) is located below the USB connector on the left side of the board.

Blinking an LED

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Templates/blink/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:




```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython Blink Example - the CircuitPython 'Hello, World!'"""
import time
import board
import digitalio

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

The built-in LED begins blinking!

Note that the code is a little less "Pythonic" than it could be. It could also be written as `led.value = not led.value` with a single `time.sleep(0.5)`. That way is more difficult to understand if you're new to programming, so the example is a bit longer than it needed to be to make it easier to read.

It's important to understand what is going on in this program.

First you `import` three modules: `time`, `board` and `digitalio`. This makes these modules available for use in your code. All three are built-in to CircuitPython, so you don't need to download anything to get started.

Next, you set up the LED. To interact with hardware in CircuitPython, your code must let the board know where to look for the hardware and what to do with it. So, you create a `digitalio.DigitalInOut()` object, provide it the LED pin using the `board` module, and save it to the variable `led`. Then, you tell the pin to act as an `OUTPUT`.

Finally, you create a `while True:` loop. This means all the code inside the loop will repeat indefinitely. Inside the loop, you set `led.value = True` which powers on the LED. Then, you use `time.sleep(0.5)` to tell the code to wait half a second before moving on to the next line. The next line sets `led.value = False` which turns the

LED off. Then you use another `time.sleep(0.5)` to wait half a second before starting the loop over again.

With only a small update, you can control the blink speed. The blink speed is controlled by the amount of time you tell the code to wait before moving on using `time.sleep()`. The example uses `0.5`, which is one half of one second. Try increasing or decreasing these values to see how the blinking changes.

That's all there is to blinking an LED using CircuitPython!

SD Card

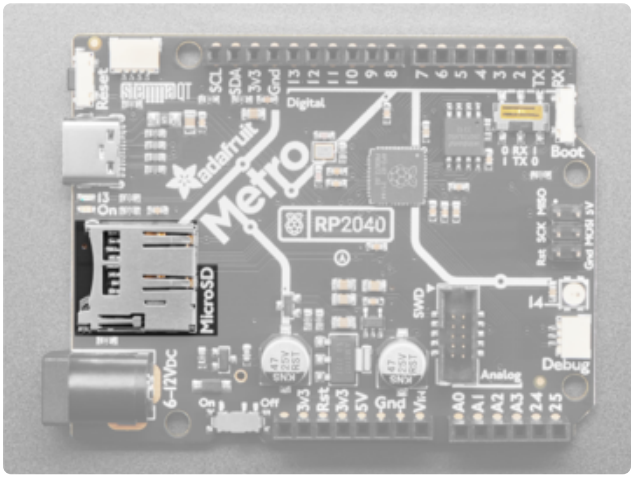
As of CircuitPython 9, a mount point (folder) named /sd is required on the CIRCUITPY drive. Make sure to create that directory after upgrading CircuitPython.

Follow these steps to create the /sd directory

https://adafru.it/19ei

Your board comes with a built in MicroSD card slot! Adding a MicroSD card allows you to include more files, music, images, videos etc. than the flash on the board can hold. You can use a MicroSD card with CircuitPython and the Adafruit_CircuitPython_SD (https://adafru.it/zwC) module. This module allows you to easily write Python code that lets you read and write to an attached MicroSD card.

MicroSD Card Slot



The **microSD card slot**, labeled MicroSD on the board silk, is located above the DC jack.

For these demos to work, you must insert a MicroSD card into the slot.

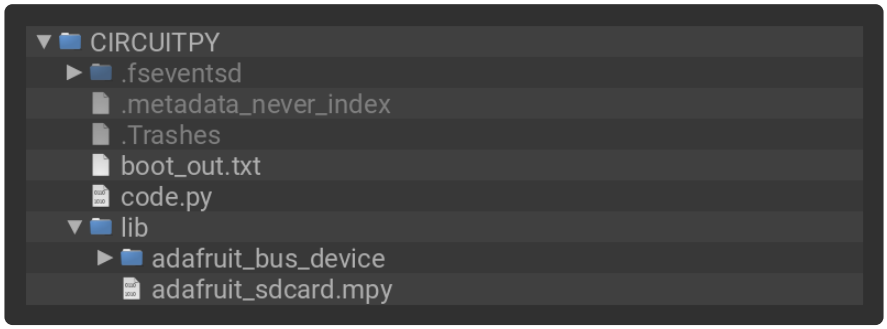
CircuitPython Usage

To use with CircuitPython, you need to first install the **adafruit_sdcard** library, and its dependencies, into the **lib** folder on your **CIRCUITPY** drive. Then you need to update **code.py** with the example script.

Thankfully, we can do this in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, and copy the **entire lib folder** and the **code.py** file to your **CIRCUITPY** drive.

Your **CIRCUITPY/lib** folder should contain the following folders and files:

- **/adafruit_bus_device**
- **adafruit_sdcard.mpy**



SD Card Read Test

Once everything is saved to the **CIRCUITPY** drive, connect to the serial console (https://adafru.it/Bec) to see the data printed out!

```
# SPDX-FileCopyrightText: 2021 ladyada for Adafruit Industries
# SPDX-License-Identifier: MIT

"""
CircuitPython Essentials SD Card Read Demo
"""

import os
import digitalio
import board
import storage
import adafruit_sdcard

# The SD_CS pin is the chip select line.
```

```
SD_CS = board.SD_CS

# Connect to the card and mount the filesystem.
cs = digitalio.DigitalInOut(SD_CS)
sdcard = adafruit_sdcard.SDCard(board.SPI(), cs)
vfs = storage.VfsFat(sdcard)
storage.mount(vfs, "/sd")

# Use the filesystem as normal! Our files are under /sd

# This helper function will print the contents of the SD
def print_directory(path, tabs=0):
    for file in os.listdir(path):
        stats = os.stat(path + "/" + file)
        filesize = stats[6]
        isdir = stats[0] & 0x4000

        if filesize < 1000:
            sizestr = str(filesize) + " bytes"
        elif filesize < 1000000:
            sizestr = "%0.1f KB" % (filesize / 1000)
        else:
            sizestr = "%0.1f MB" % (filesize / 1000000)

        prettyprintname = ""
        for _ in range(tabs):
            prettyprintname += "   "
        prettyprintname += file
        if isdir:
            prettyprintname += "/"
        print("{0:<40} Size: {1:>10}".format(prettyprintname, sizestr))

        # recursively print directory contents
        if isdir:
            print_directory(path + "/" + file, tabs + 1)


print("Files on filesystem:")
print("====================")
print_directory("/sd")
```

```
CircuitPython REPL
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Files on filesystem:
====================
System Volume Information/              Size:    0 bytes
   WPSettings.dat                       Size:   12 bytes
   IndexerVolumeGuid                    Size:   76 bytes
icons/                                  Size:    0 bytes
   01d.bmp                              Size:   460.9 KB
   01n.bmp                              Size:   460.9 KB
   02d.bmp                              Size:   460.9 KB
   02n.bmp                              Size:   460.9 KB
   03d.bmp                              Size:   460.9 KB
   03n.bmp                              Size:   460.9 KB
   04d.bmp                              Size:   460.9 KB
   04n.bmp                              Size:   460.9 KB
   09d.bmp                              Size:   460.9 KB
   09n.bmp                              Size:   460.9 KB
   10d.bmp                              Size:   460.9 KB
   10n.bmp                              Size:   460.9 KB
   11d.bmp                              Size:   460.9 KB
```

In this read test for the SD card, the filesystem on the SD card is mounted and read. Then, the contents of the filesystem are printed to the REPL.

SD Card Write Test

```
# SPDX-FileCopyrightText: 2021 ladyada for Adafruit Industries
# SPDX-License-Identifier: MIT

"""
CircuitPython Essentials SD Card Write Demo
"""

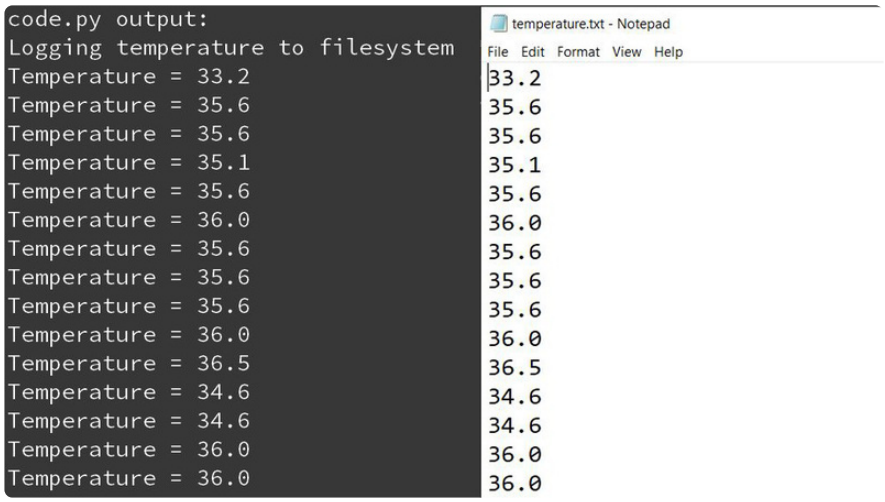
import time
import adafruit_sdcard
import board
import digitalio
import microcontroller
import storage

# The SD_CS pin is the chip select line.
SD_CS = board.SD_CS

# Connect to the card and mount the filesystem.
cs = digitalio.DigitalInOut(SD_CS)
sdcard = adafruit_sdcard.SDCard(board.SPI(), cs)
vfs = storage.VfsFat(sdcard)
storage.mount(vfs, "/sd")

# Use the filesystem as normal! Our files are under /sd

print("Logging temperature to filesystem")
# append to the file!
while True:
    # open file for append
    with open("/sd/temperature.txt", "a") as f:
        t = microcontroller.cpu.temperature
        print("Temperature = %0.1f" % t)
        f.write("%0.1f\n" % t)
    # file is saved
    time.sleep(1)
```



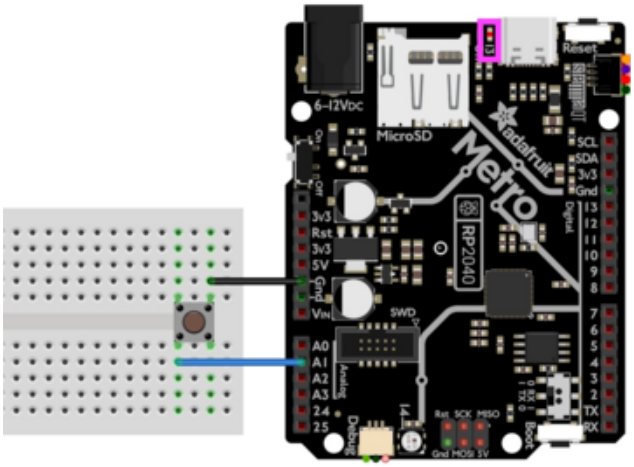
In this example, the code is writing data to the SD card. The microcontroller CPU temperature is printed to the REPL and logged to a text file on the SD card. After running the code, you can read the text file from the SD card. You'll see that the text file matches what was printed to the REPL.

Digital Input

The CircuitPython `digitalio` module has many applications. The basic Blink program sets up the LED as a digital output. You can just as easily set up a **digital input** such as a button to control the LED. This example builds on the basic Blink example, but now includes setup for a button switch. Instead of using the `time` module to blink the LED, it uses the status of the button switch to control whether the LED is turned on or off.

LED and Button

Connect a button to your Metro RP2040 as follows.



The **red led** (highlighted in magenta) is located next to the USB connector and the 13 label on the board silk.

Wiring the button:

One leg of button to **Metro GND**
Opposite leg of button to **Metro A1**

Controlling the LED with a Button

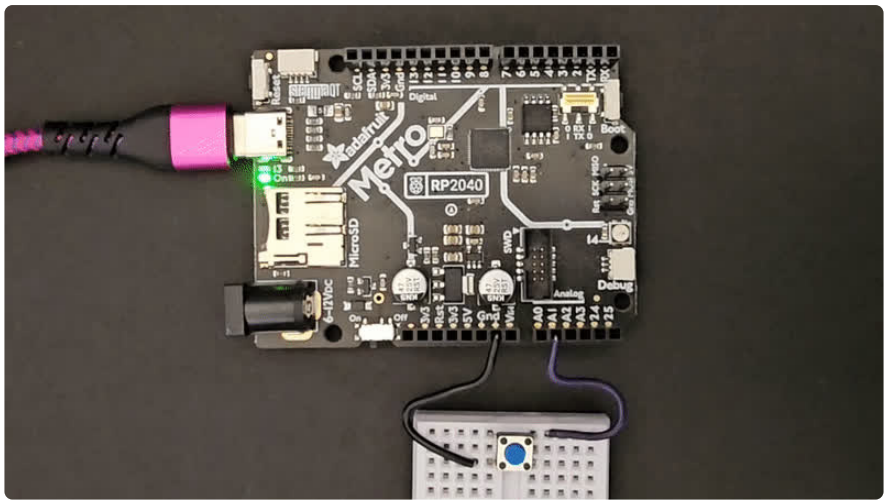
```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
CircuitPython Digital Input Example - Blinking an LED using a button switch.
"""
import board
import digitalio

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

button = digitalio.DigitalInOut(board.A1)
button.switch_to_input(pull=digitalio.Pull.UP)

while True:
    if not button.value:
        led.value = True
    else:
        led.value = False
```

Now, press the button. The LED lights up! Let go of the button and the LED turns off.



Note that the code is a little less "Pythonic" than it could be. It could also be written as `led.value = not button.value`. That way is more difficult to understand if you're new to programming, so the example is a bit longer than it needed to be to make it easier to read.

First you `import` two modules: `board` and `digitalio`. This makes these modules available for use in your code. Both are built-in to CircuitPython, so you don't need to download anything to get started.

Next, you set up the LED. To interact with hardware in CircuitPython, your code must let the board know where to look for the hardware and what to do with it. So, you create a `digitalio.DigitalInOut()` object, provide it the LED pin using the `board` module, and save it to the variable `led`. Then, you tell the pin to act as an `OUTPUT`.

You include setup for the button as well. It is similar to the LED setup, except the button is an `INPUT`, and requires a pull up.

Inside the loop, you check to see if the button is pressed, and if so, turn on the LED. Otherwise the LED is off.

That's all there is to controlling an LED with a button switch!

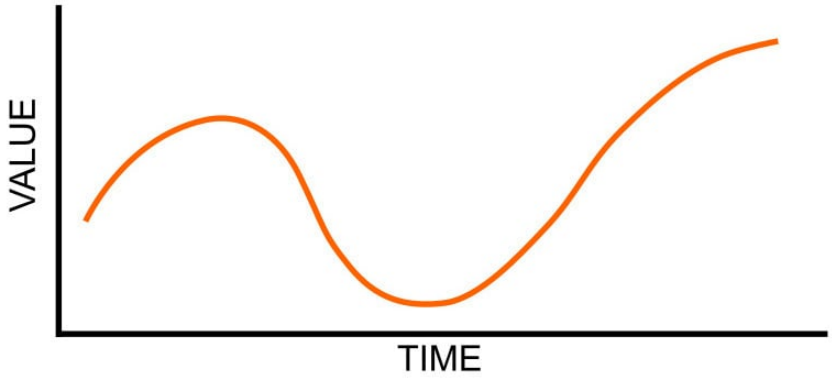
Analog In

Your microcontroller board has both digital and analog signal capabilities. Some pins are analog, some are digital, and some are capable of both. Check the **Pinouts** page in this guide for details about your board.

Analog signals are different from digital signals in that they can be any voltage and can vary continuously and smoothly between voltages. An analog signal is like a dimmer switch on a light, whereas a digital signal is like a simple on/off switch.

Digital signals only can ever have two states, they are either are **on** (high logic level voltage like 3.3V) or **off** (low logic level voltage like 0V / ground).

By contrast, analog signals can be any voltage in-between on and off, such as 1.8V or 0.001V or 2.98V and so on.



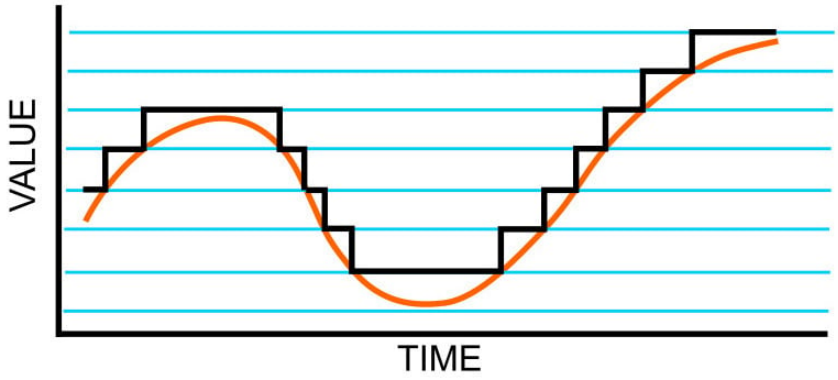
Analog signals are continuous values which means they can be an infinite number of different voltages. Think of analog signals like a floating point or fractional number, they can smoothly transiting to any in-between value like 1.8V, 1.81V, 1.801V, 1.8001V, 1.80001V and so forth to infinity.

Many devices use analog signals, in particular sensors typically output an analog signal or voltage that varies based on something being sensed like light, heat, humidity, etc.

Analog to Digital Converter (ADC)

An analog-to-digital-converter, or ADC, is the key to reading analog signals and voltages with a microcontroller. An ADC is a device that reads the voltage of an analog signal and converts it into a digital, or numeric, value. The microcontroller can't read analog signals directly, so the analog signal is first converted into a numeric value by the ADC.

The black line below shows a digital signal over time, and the red line shows the converted analog signal over the same amount of time.

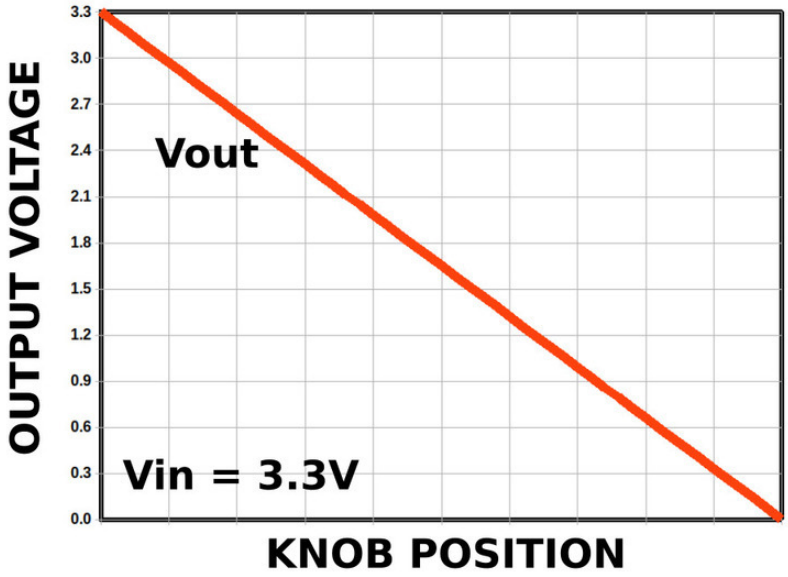


Once that analog signal has been converted by the ADC, the microcontroller can use those digital values any way you like!

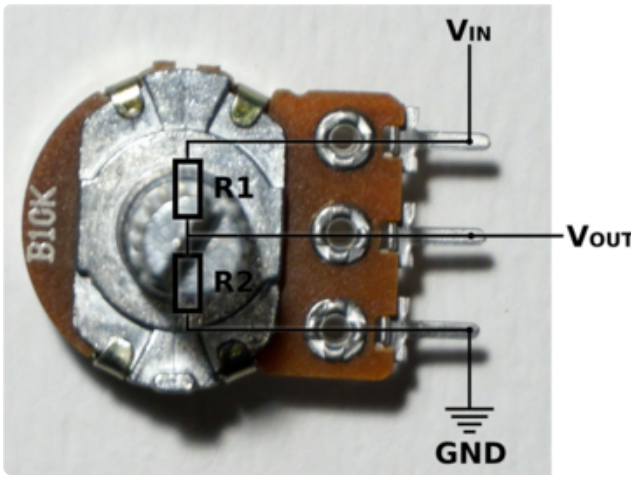
Potentiometers

A potentiometer is a small variable resistor that you can twist a knob or shaft to change its resistance. It has three pins. By twisting the knob on the potentiometer you can change the resistance of the middle pin (called the wiper) to be anywhere within the range of resistance of the potentiometer.

By wiring the potentiometer to your board in a special way (called a voltage divider) you can turn the change in resistance into a change in voltage that your board's analog to digital converter can read.



To wire up a potentiometer as a voltage divider:

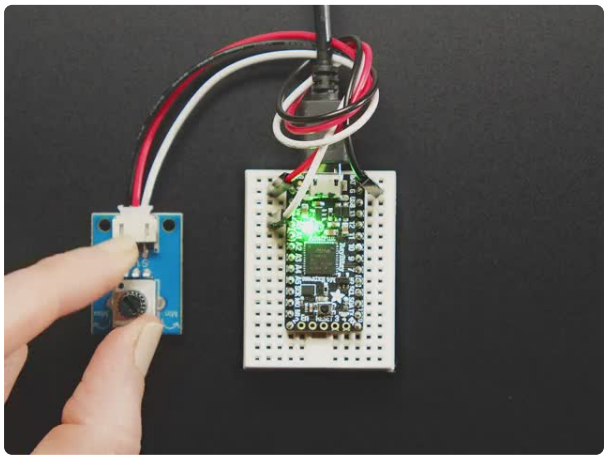


Connect **one outside pin** to **ground**
Connect **the other outside pin** to **voltage in (e.g. 3.3V)**
Connect **the middle pin** to **an analog pin (e.g. A0)**

Hardware

In addition to your microcontroller board, you will need the following hardware to follow along with this example.

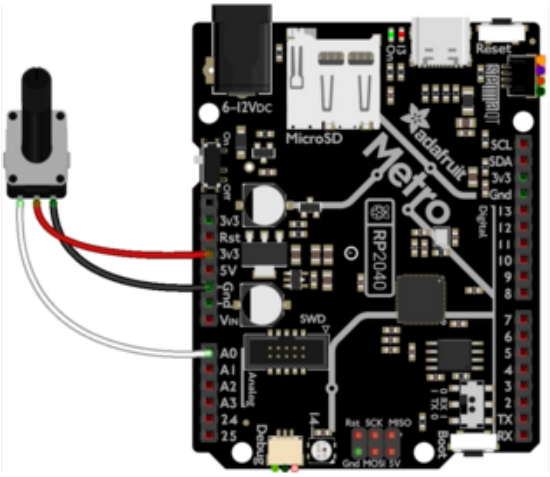
Potentiometer



STEMMA Wired Potentiometer Breakout Board - 10K ohm Linear
For the easiest way possible to measure twists, turn to this STEMMA potentiometer breakout (ha!). This plug-n-play pot comes with a JST-PH 2mm connector and a matching
https://www.adafruit.com/product/4493

Wire Up the Potentiometer

Connect the potentiometer to your board as follows.



Potentiometer left pin (white wire) to **Metro A0**
Potentiometer center pin (red wire) to **Metro 3V3**
Potentiometer right pin (black wire) to **Metro GND**

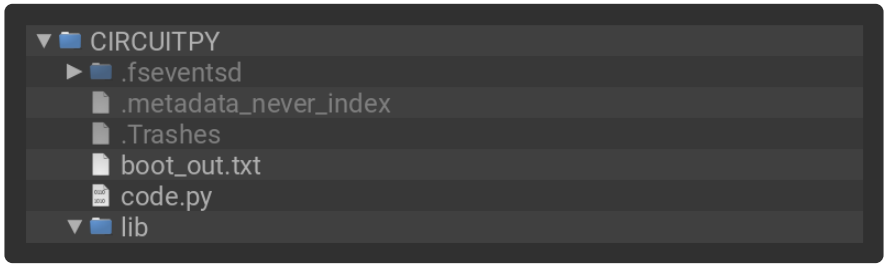
Reading Analog Pin Values

CircuitPython makes it easy to read analog pin values. Simply import two modules, set up the pin, and then print the value inside a loop.

You'll need to connect to the serial console (https://adafru.it/Bec) to see the values printed out.

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Templates/analog_pin_values/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:




```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython analog pin value example"""
import time
import board
import analogio

analog_pin = analogio.AnalogIn(board.A0)

while True:
```

```
    print(analog_pin.value)
    time.sleep(0.1)
```

Now, rotate the potentiometer to see the values change.



What do these values mean? In CircuitPython ADC values are put into the range of 16-bit unsigned values. This means the possible values you'll read from the ADC fall within the range of 0 to 65535 (or 2^16 - 1). When you twist the potentiometer knob to be near ground, or as far to the left as possible, you see a value close to zero. When you twist it as far to the right as possible, near 3.3 volts, you see a value close to 65535. You're seeing almost the full range of 16-bit values!

The code is simple. You begin by importing three modules: `time`, `board` and `analogio`. All three modules are built into CircuitPython, so you don't need to download anything to get started.

Then, you set up the analog pin by creating an `analogio.AnalogIn()` object, providing it the desired pin using the `board` module, and saving it to the variable `analog_pin`.

Finally, in the loop, you print out the analog value with `analog_pin.value`, including a `time.sleep()` to slow down the values to a human-readable rate.

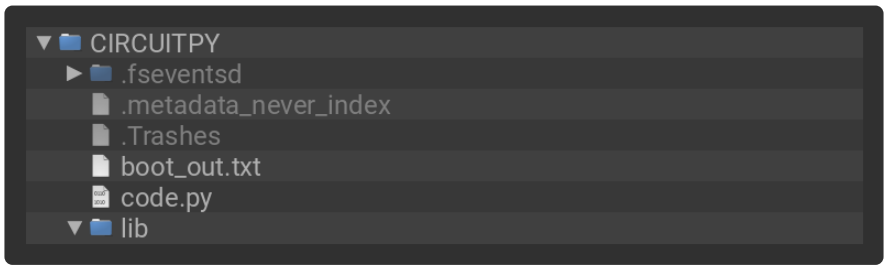
Reading Analog Voltage Values

These values don't necessarily equate to anything obvious. You can get an idea of the rotation of the potentiometer based on where in the range the value falls, but not without doing some math. Remember, you wired up the potentiometer as a voltage divider. By adding a simple function to your code, you can get a more human-readable value from the potentiometer.

You'll need to connect to the serial console (https://adafru.it/Bec) to see the values printed out.

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Templates/analog_voltage_values/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:



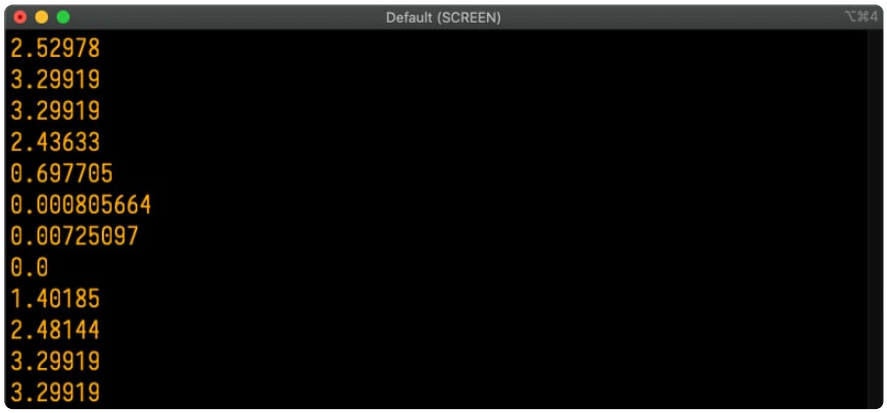

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython analog voltage value example"""
import time
import board
import analogio

analog_pin = analogio.AnalogIn(board.A0)


def get_voltage(pin):
    return (pin.value * 3.3) / 65535


while True:
    print(get_voltage(analog_pin))
    time.sleep(0.1)
```

Now, rotate the potentiometer to see the values change.



Now the values range from around 0 to 3.3! Note that you may not get all the way to 0 or 3.3. This is normal.

The example code begins with the same imports and pin setup.

This time, you include the `get_voltage` helper. This function requires that you provide an analog pin. It then maps the raw analog values, `0` to `65535`, to the voltage values, `0` to `3.3`. It does the math so you don't have to!

Finally, inside the loop, you provide the function with your `analog_pin`, and print the resulting values.

That's all there is to reading analog voltage values using CircuitPython!

NeoPixel LED

Your board has a built-in RGB NeoPixel status LED. You can use CircuitPython code to control the color and brightness of this LED. It is also used to indicate the bootloader status and errors in your CircuitPython code.

A NeoPixel is what Adafruit calls the WS281x family of addressable RGB LEDs. It contains three LEDs - a red one, a green one and a blue one - along side a driver chip in a tiny package controlled by a single pin. They can be used individually (as in the built-in LED on your board), or chained together in strips or other creative form factors. NeoPixels do not light up on their own; they require a microcontroller. So, it's super convenient that the NeoPixel is built in to your microcontroller board!

This page will cover using CircuitPython to control the status RGB NeoPixel built into your microcontroller. You'll learn how to change the color and brightness, and how to make a rainbow. Time to get started!

NeoPixel Location



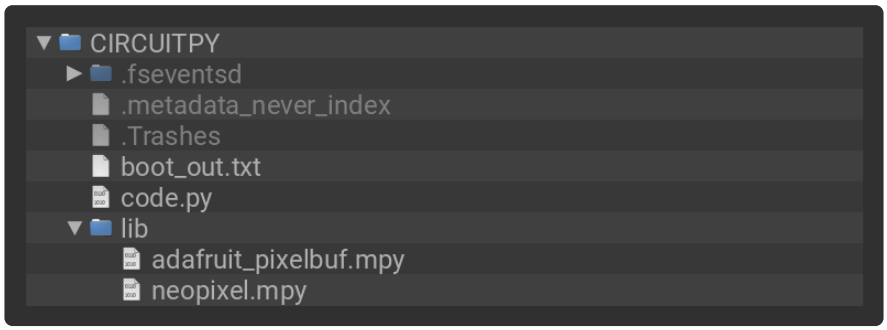
The **NeoPixel LED** (highlighted in red) is on the right side of the board, to the right of the 14 label on the board silk.

NeoPixel Color and Brightness

To use with CircuitPython, you need to first install a few libraries, into the **lib** folder on your **CIRCUITPY** drive. Then you need to update **code.py** with the example script.

Thankfully, we can do this in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Templates/status_led_one_neopixel_rgb/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:



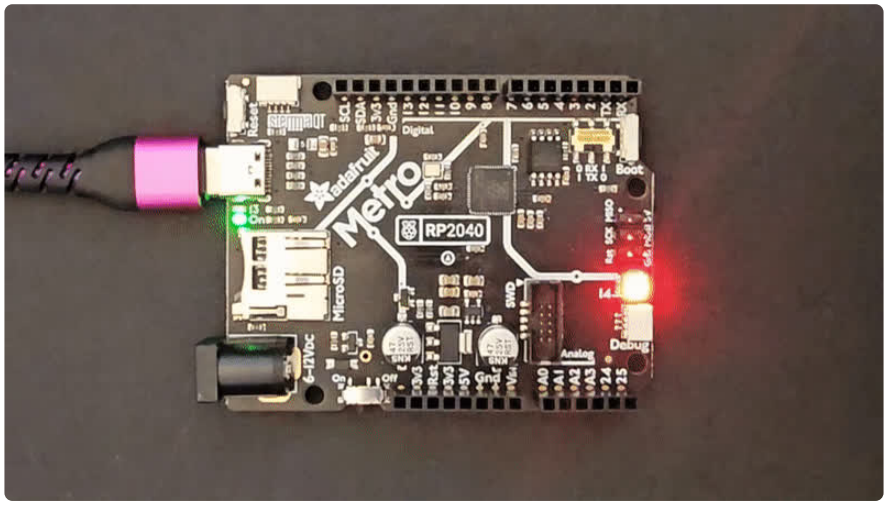

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython status NeoPixel red, green, blue example."""
import time
import board
import neopixel

pixel = neopixel.NeoPixel(board.NEOPIXEL, 1)
```

```
pixel.brightness = 0.3

while True:
    pixel.fill((255, 0, 0))
    time.sleep(0.5)
    pixel.fill((0, 255, 0))
    time.sleep(0.5)
    pixel.fill((0, 0, 255))
    time.sleep(0.5)
```

The built-in NeoPixel begins blinking red, then green, then blue, and repeats!



First you import two modules, `time` and `board`, and one library, `neopixel`. This makes these modules and libraries available for use in your code. The first two are modules built-in to CircuitPython, so you don't need to download anything to use those. The `neopixel` library is separate, which is why you needed to install it before getting started.

Next, you set up the NeoPixel LED. To interact with hardware in CircuitPython, your code must let the board know where to look for the hardware and what to do with it. So, you create a `neopixel.NeoPixel()` object, provide it the NeoPixel LED pin using the `board` module, and tell it the number of LEDs. You save this object to the variable `pixel`.

Then, you set the NeoPixel brightness using the `brightness` attribute. `brightness` expects float between `0` and `1.0`. A float is essentially a number with a decimal in it. The brightness value represents a percentage of maximum brightness; `0` is 0% and `1.0` is 100%. Therefore, setting `pixel.brightness = 0.3` sets the brightness to 30%. The default brightness, which is to say the brightness if you don't explicitly set it, is `1.0`. The default is really bright! That is why there is an option available to easily change the brightness.

Inside the loop, you turn the NeoPixel red for 0.5 seconds, green for 0.5 seconds, and blue for 0.5 seconds.

To turn the NeoPixel red, you "fill" it with an RGB value. Check out the section below for details on RGB colors. The RGB value for red is `(255, 0, 0)`. Note that the RGB value includes the parentheses. The `fill()` attribute expects the full RGB value including those parentheses. That is why there are two pairs of parentheses in the code.

You can change the RGB values to change the colors that the NeoPixel cycles through. Check out the list below for some examples. You can make any color of the rainbow with the right RGB value combination!

That's all there is to changing the color and setting the brightness of the built-in NeoPixel LED!

RGB LED Colors

RGB LED colors are set using a combination of **r**ed, **g**reen, and **b**lue, in the form of an (**R**, **G**, **B**) tuple. Each member of the tuple is set to a number between 0 and 255 that determines the amount of each color present. Red, green and blue in different combinations can create all the colors in the rainbow! So, for example, to set an LED to red, the tuple would be `(255, 0, 0)`, which has the maximum level of red, and no green or blue. Green would be `(0, 255, 0)`, etc. For the colors between, you set a combination, such as cyan which is `(0, 255, 255)`, with equal amounts of green and blue. If you increase all values to the same level, you get white! If you decrease all the values to 0, you turn the LED off.

Common colors include:

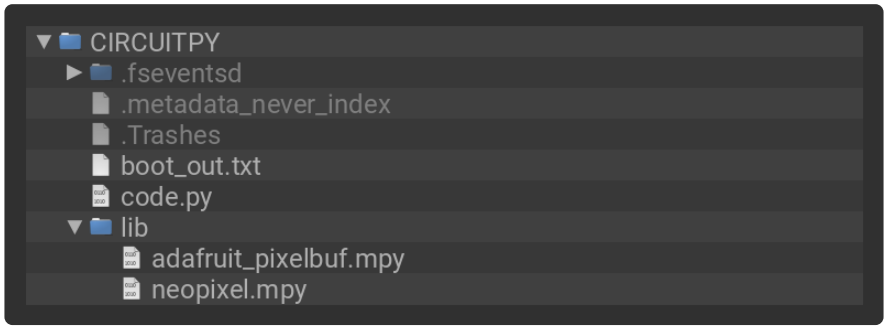
- red: `(255, 0, 0)`
- green: `(0, 255, 0)`
- blue: `(0, 0, 255)`
- cyan: `(0, 255, 255)`
- purple: `(255, 0, 255)`
- yellow: `(255, 255, 0)`
- white: `(255, 255, 255)`
- black (off): `(0, 0, 0)`

NeoPixel Rainbow

You should have already installed the library necessary to use the built-in NeoPixel LED. If not, follow the steps at the beginning of the NeoPixel Color and Brightness section to install it.

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Templates/status_led_one_neopixel_rainbow/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:



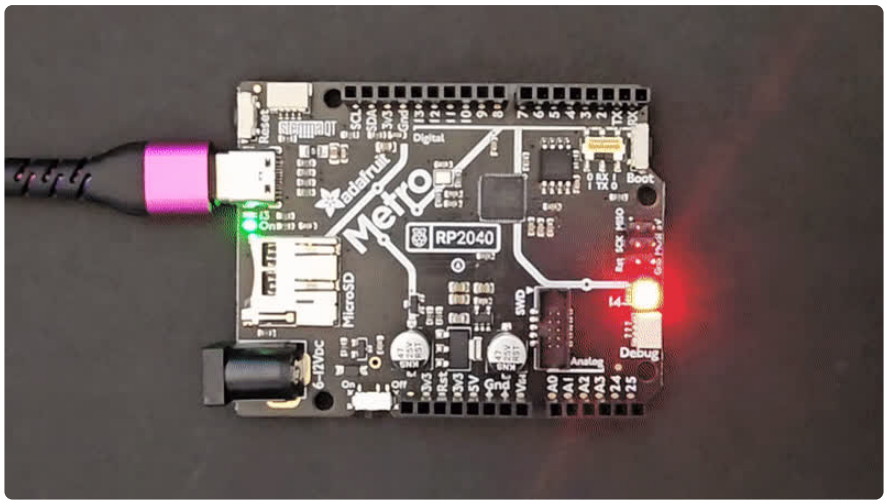

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython status NeoPixel rainbow example."""
import time
import board
from rainbowio import colorwheel
import neopixel

pixel = neopixel.NeoPixel(board.NEOPIXEL, 1)
pixel.brightness = 0.3


def rainbow(delay):
    for color_value in range(255):
        pixel[0] = colorwheel(color_value)
        time.sleep(delay)


while True:
    rainbow(0.02)
```

The NeoPixel displays a rainbow cycle!



This example builds on the previous example.

First, you import the same three modules and libraries. In addition to those, you import `colorwheel`.

The NeoPixel hardware setup and brightness setting are the same.

Next, you have the `rainbow()` helper function. This helper displays the rainbow cycle. It expects a `delay` in seconds. The higher the number of seconds provided for `delay`, the slower the rainbow will cycle. The helper cycles through the values of the color wheel to create a rainbow of colors.

Inside the loop, you call the rainbow helper with a 0.2 second delay, by including `rainbow(0.2)`.

That's all there is to making rainbows using the built-in NeoPixel LED!

Storage

CircuitPython-compatible microcontrollers show up as a **CIRCUITPY** drive when plugged into your computer, allowing you to edit code directly on the board. Perhaps you've wondered whether or not you can write data from CircuitPython directly to the board to act as a data logger. The answer is **yes**!

The `storage` module in CircuitPython enables you to write code that allows CircuitPython to write data to the **CIRCUITPY** drive. This process requires you to include a **boot.py** file on your **CIRCUITPY** drive, along side your **code.py** file.

The **boot.py** file is special - the code within it is executed when CircuitPython starts up, either from a hard reset or powering up the board. It is not run on soft reset, for

example, if you reload the board from the serial console or the REPL. This is in contrast to the code within **code.py**, which is executed after CircuitPython is already running.

The **CIRCUITPY** drive is typically writable by your computer; this is what allows you to edit your code directly on the board. The reason you need a **boot.py** file is that you have to set the filesystem to be read-only by your computer to allow it to be writable by CircuitPython. This is because CircuitPython cannot write to the filesystem at the same time as your computer. Doing so can lead to filesystem corruption and loss of all content on the drive, so CircuitPython is designed to only allow one at at time.

You can only have EITHER your computer edit files on the CIRCUITPY drive, OR have CircuitPython edit files. You cannot have both writing to the CIRCUITPY drive at the same time. CircuitPython doesn't allow it!

The boot.py File

The filesystem will NOT automatically be set to read-only on creation of this file! You'll still be able to edit files on CIRCUITPY after saving this boot.py.

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
CircuitPython Essentials Storage CP Filesystem boot.py file
"""
import board
import digitalio
import storage

pin = digitalio.DigitalInOut(board.A0)
pin.switch_to_input(pull=digitalio.Pull.UP)

# If the pin is connected to ground, the filesystem is writable by CircuitPython
storage.remount("/", readonly=pin.value)
```

The `storage.remount()` command has a `readonly` keyword argument. **This argument refers to the read/write state of CircuitPython.** It does NOT refer to the read/write state of your computer.

When the physical pin is connected to ground, it returns `False`. The `readonly` argument in boot.py is set to the `value` of the pin. When the `value=True`, the CIRCUITPY drive is read-only to CircuitPython (and writable by your computer). **When the `value=False`, the CIRCUITPY drive is writable by CircuitPython** (an read-only by your computer).

The code.py File

Save the following as **code.py** on your **CIRCUITPY** drive.

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
CircuitPython Essentials Storage CP Filesystem code.py file

For use with boards with a built-in red LED.
"""
import time
import board
import digitalio
import microcontroller

led = digitalio.DigitalInOut(board.LED)
led.switch_to_output()

try:
    with open("/temperature.txt", "a") as temp_log:
        while True:
            # The microcontroller temperature in Celsius. Include the
            # math to do the C to F conversion here, if desired.
            temperature = microcontroller.cpu.temperature

            # Write the temperature to the temperature.txt file every 10 seconds.
            temp_log.write('{0:.2f}\n'.format(temperature))
            temp_log.flush()

            # Blink the LED on every write...
            led.value = True
            time.sleep(1)  # ...for one second.
            led.value = False  # Then turn it off...
            time.sleep(9)  # ...for the other 9 seconds.

except OSError as e:  # When the filesystem is NOT writable by CircuitPython...
    delay = 0.5  # ...blink the LED every half second.
    if e.args[0] == 28:  # If the file system is full...
        delay = 0.15  # ...blink the LED every 0.15 seconds!
    while True:
        led.value = not led.value
        time.sleep(delay)
```

First you import the necessary modules to make them available to your code, and you set up the LED.

Next you have a `try`/`except` block, which is used to handle the three potential states of the board: read/write, read-only, or filesystem full. The code in the `try` block will run if the filesystem is writable by CircuitPython. The code in the `except` block will run if the filesystem is read-only to CircuitPython OR if the filesystem is full.

Under the `try`, you open a **temperature.txt** log file. If it is the first time, it will create the file. For all subsequent times, it opens the file and appends data. Inside the loop, you get the microcontroller temperature value and assign it to a `temperature` variable. Then, you write the temperature value to the log file, followed by clearing

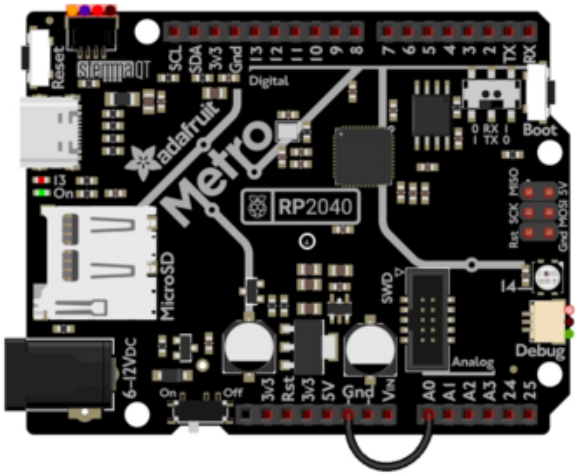
the buffer for the next time through the loop. The temperature data is limited to two decimal points to save space for more data. Finally, you turn the LED on for one second, and then turn it off for the next nine seconds. Essentially, you blink the LED for one second every time the temperature is logged to the file which happens every ten seconds.

Next you `except` an `OSError`. An `OSError` number 30 is raised when trying to create, open or write to a file on a filesystem that is read-only to CircuitPython. If any `OSError` other than 28 is raised (e.g. 30), the `delay` is set to 0.5 seconds. If the filesystem fills up, CircuitPython raises `OSError` number 28. If `OSError` number 28 is raised, the `delay` is set to 0.15 seconds. Inside the loop, the LED is turned on for the duration of the `delay`, and turned off for the duration of the `delay`, effectively blinking the LED at the speed of the `delay`.

Logging the Temperature

At the moment, the LED on your board should be blinking once every half second. This indicates that the board is currently read-only to CircuitPython, and writable to your computer, allowing you to update the files on your **CIRCUITPY** drive as needed.

The way the code in **boot.py** works is, it checks to see if the pin is connected to ground when the board is powered on. To begin logging the temperature, you must connect the pin to ground.



Connect a jumper wire from **A0** to **GND**.

Once the pin is connected to ground, you need to either hard reset the board by pressing the reset button, or by unplugging the USB cable and plugging it back in. This will run the code within **boot.py** and set your board to writable by CircuitPython, and therefore, read-only by the computer.

The blinking will slow down to one second long, every 10 seconds. This indicates that the board is currently logging the temperature, once every 10 seconds.

As long as the pin is connected to ground, you can plug the board in anywhere you have USB power, and log the temperature in that location! The temperature is not the ambient temperature; it is the temperature inside the microcontroller, which will typically be higher than ambient temperature. However, running only this code, once the microcontroller temperature stabilises, it should at least be consistent, and therefore usable for tracking changes in ambient temperature.

If the LED starts blinking really quickly, it means the filesystem is full! You'll need to get your temperature data and delete the temperature log file to begin again.

That's all there is to logging the temperature using CircuitPython!

Recovering a Read-Only Filesystem

In the event that you make your **CIRCUITPY** drive read-only to your computer, and for some reason, it doesn't easily switch back to writable, there are a couple of things you can do to recover the filesystem.

Even when the **CIRCUITPY** drive is read-only to your computer, you can still access the serial console and REPL. If you connect to the serial console and enter the REPL, you can run **either** of the following two sets of commands at the `>>>` prompt. You do not need to run both.

First, you can rename your **boot.py** file to something other than **boot.py**.

```
import os
os.rename("boot.py", "something_else.py")
```

Alternatively, you can remove the **boot.py** file altogether.

```
import os
os.remove("boot.py")
```

Then, restart the board by either hitting the reset button or unplugging USB and plugging it back in. **CIRCUITPY** should show up on your computer as usual, but now it should be writable by your computer.

Capacitive Touch

Your microcontroller board has capacitive touch capabilities on multiple pins. The CircuitPython `touchio` module makes it simple to detect when you touch a pin, enabling you to use it as an input.

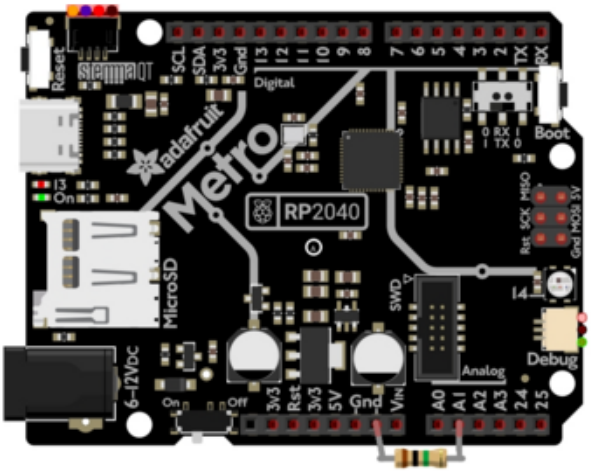
This section first covers using the `touchio` module to read touches on one pin. You'll learn how to setup the pin in your program, and read the touch status. Then, you'll learn how to read touches on multiple pins in a single example. Time to get started!

One Capacitive Touch Pin

The first example covered here will show you how to read touches on one pin.

Pin Wiring

Capacitive touch always benefits from the use of a 1MΩ pulldown resistor. Some microcontrollers have pulldown resistors built in, but using the built-in ones can yield unexpected results. Other microcontrollers to not have built-in pulldowns, and require an external pulldown resistor. Therefore, the best option is to include one regardless.



One leg of 1MΩ resistor to **Metro A1**
Opposite leg of 1MΩ resistor to **Metro GND**

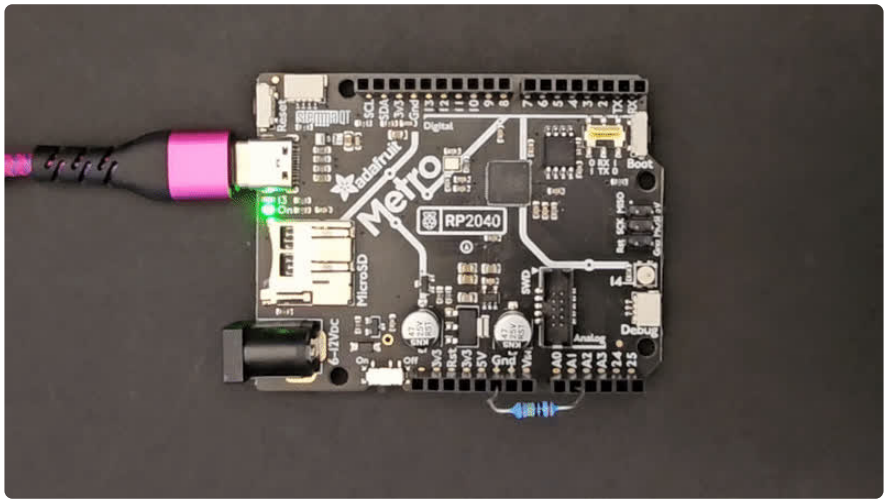
Reading Touch on the Pin

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
CircuitPython Capacitive Touch Pin Example - Print to the serial console when one
pin is touched.
"""
import time
import board
import touchio

touch = touchio.TouchIn(board.A1)

while True:
    if touch.value:
        print("Pin touched!")
    time.sleep(0.1)
```

Now touch the pin indicated in the diagram above. You'll see `Pin touched!` printed to the serial console!



First you `import` three modules: `time`, `board` and `touchio`. This makes these modules available for use in your code. All three are built-in to CircuitPython, so you don't find any library files in the Project Bundle.

Next, you create the `touchio.TouchIn()` object, and provide it the pin name using the `board` module. You save that to the `touch` variable.

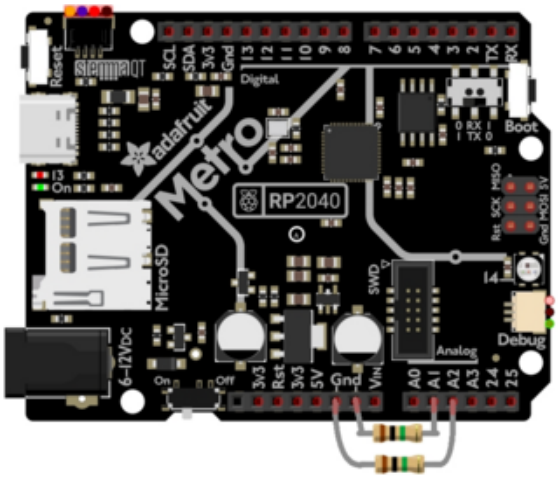
Inside the loop, you check to see if the pin is touched. If so, you print to the serial console. Finally, you include a `time.sleep()` to slow it down a bit so the output is readable.

That's all there is to reading touch on a single pin using `touchio` in CircuitPython!

Multiple Capacitive Touch Pins

The next example shows you how to read touches on multiple pins in a single program.

Pin Wiring



One leg of first 1MΩ resistor to **Metro A1**
Opposite leg of first 1MΩ resistor to **Metro GND**
One leg of second 1MΩ resistor to **Metro A2**
Opposite leg of second 1MΩ resistor to **Metro GND**

Reading Touch on the Pins

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
CircuitPython Capacitive Two Touch Pin Example - Print to the serial console when a
pin is touched.
"""
import time
import board
import touchio

touch_one = touchio.TouchIn(board.A1)
touch_two = touchio.TouchIn(board.A2)

while True:
    if touch_one.value:
        print("Pin one touched!")
    if touch_two.value:
        print("Pin two touched!")
    time.sleep(0.1)
```

Touch the pins to see the messages printed to the serial console!

This example builds on the first. The imports remain the same.

The `touchio.TouchIn()` object is created, but is instead saved to `touch_one`. A second `touchio.TouchIn()` object is also created, the second pin is provided to it using the `board` module, and is saved to `touch_two`.

Inside the loop, we check to see if pin one and pin two are touched, and if so, print to the serial console `Pin one touched!` and `Pin two touched!`, respectively. The same `time.sleep()` is included.

If more touch-capable pins are available on your board, you can easily add them by expanding on this example!

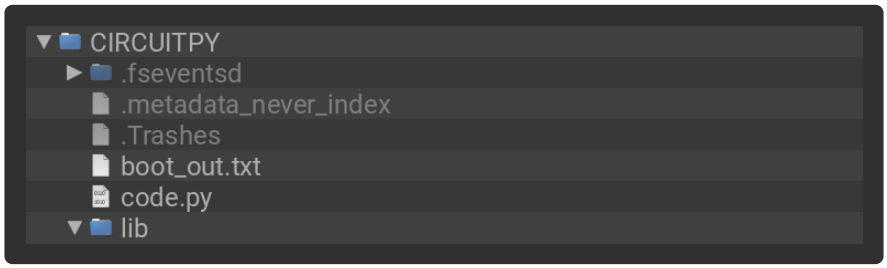
Where are my Touch-Capable pins?

There are specific pins on a microcontroller that support capacitive touch. How do you know which pins will work? Easy! Run the script below to get a list of all the pins that are available.

Save the following to your **CIRCUITPY** drive as **code.py**.

Click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, find your CircuitPython version, and copy the matching **code.py** file to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:




```
# SPDX-FileCopyrightText: 2021-2023 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
CircuitPython Touch-Compatible Pin Identification Script

Depending on the order of the pins in the CircuitPython pin definition, some
inaccessible pins
may be returned in the script results. Consult the board schematic and use your
best judgement.

In some cases, such as LED, the associated pin, such as D13, may be accessible. The
LED pin
name is first in the list in the pin definition, and is therefore printed in the
results. The
pin name "LED" will work in code, but "D13" may be more obvious. Use the schematic
to verify.
"""
import board
import touchio
from microcontroller import Pin


def get_pin_names():
    """
    Gets all unique pin names available in the board module, excluding a defined
list.
    This list is not exhaustive, and depending on the order of the pins in the
CircuitPython
    pin definition, some of the pins in the list may still show up in the script
```

```
results.
    """
    exclude = [
        "NEOPIXEL",
        "APA102_MOSI",
        "APA102_SCK",
        "LED",
        "NEOPIXEL_POWER",
        "BUTTON",
        "BUTTON_UP",
        "BUTTON_DOWN",
        "BUTTON_SELECT",
        "DOTSTAR_CLOCK",
        "DOTSTAR_DATA",
        "IR_PROXIMITY",
        "SPEAKER_ENABLE",
        "BUTTON_A",
        "BUTTON_B",
        "POWER_SWITCH",
        "SLIDE_SWITCH",
        "TEMPERATURE",
        "ACCELEROMETER_INTERRUPT",
        "ACCELEROMETER_SDA",
        "ACCELEROMETER_SCL",
        "MICROPHONE_CLOCK",
        "MICROPHONE_DATA",
        "RFM_RST",
        "RFM_CS",
        "RFM_IO0",
        "RFM_IO1",
        "RFM_IO2",
        "RFM_IO3",
        "RFM_IO4",
        "RFM_IO5",
    ]
    pins = [
        pin
        for pin in [getattr(board, p) for p in dir(board) if p not in exclude]
        if isinstance(pin, Pin)
    ]
    pin_names = []
    for pin_object in pins:
        if pin_object not in pin_names:
            pin_names.append(pin_object)
    return pin_names


for possible_touch_pin in get_pin_names():  # Get the pin name.
    try:
        touch_pin_object = touchio.TouchIn(
            possible_touch_pin
        )  # Attempt to create the touch object on each pin.
        # Print the touch-capable pins that do not need, or already have, an
external pulldown.
        print("Touch on:", str(possible_touch_pin).replace("board.", ""))
    except ValueError as error:  # A ValueError is raised when a pin is invalid or
needs a pulldown.
        # Obtain the message associated with the ValueError.
        error_message = getattr(error, "message", str(error))
        if (
            "pulldown" in error_message  # If the ValueError is regarding needing a
pulldown...
        ):
            print(
                "Touch on:", str(possible_touch_pin).replace("board.", "")
            )
        else:
            print("No touch on:", str(possible_touch_pin).replace("board.", ""))
    except TypeError:  # Error returned when checking a non-pin object in
```

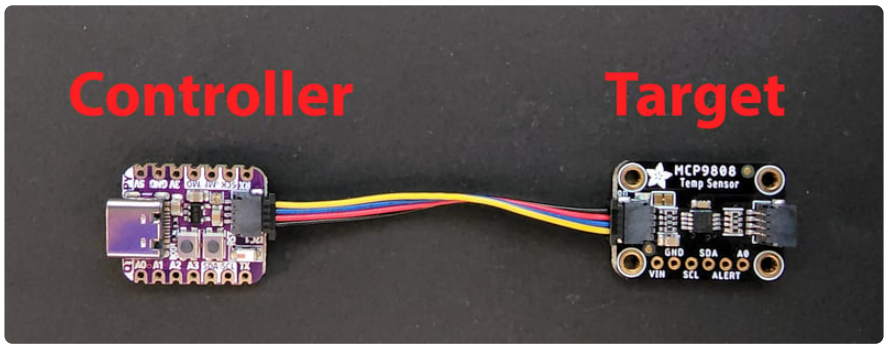
```
dir(board).
        pass  # Passes over non-pin objects in dir(board).
```

Now, connect to the serial console and check out the output! The results print out a nice handy list of pins that support capacitive touch.

The output for the Metro RP2040 is long! The screenshot shows only the beginning. Run the script yourself to see the full output!



I2C



The **I2C**, or inter-integrated circuit (https://adafru.it/u2a), is a 2-wire protocol for communicating with simple sensors and devices, which means it uses two connections, or wires, for transmitting and receiving data. One connection is a clock, called **SCL**. The other is the data line, called **SDA**. Each pair of clock and data pins are referred to as a **bus**.

Typically, there is a device that acts as a **controller** and sends requests to the **target** devices on each bus. In this case, your microcontroller board acts as the controller, and the sensor breakout acts as the target. Historically, the controller is referred to as the master, and the target is referred to as the slave, so you may run into that terminology elsewhere. The official terminology is controller and target (https://adafru.it/TtF).

Multiple I2C devices can be connected to the same clock and data lines. Each I2C device has an address, and as long as the addresses are different, you can connect

them at the same time. This means you can have many different sensors and devices all connected to the same two pins.

Both I2C connections require pull-up resistors, and most Adafruit I2C sensors and breakouts have pull-up resistors built in. If you're using one that does not, you'll need to add your own 2.2-10kΩ pull-up resistors from SCL and SDA to 3.3V.

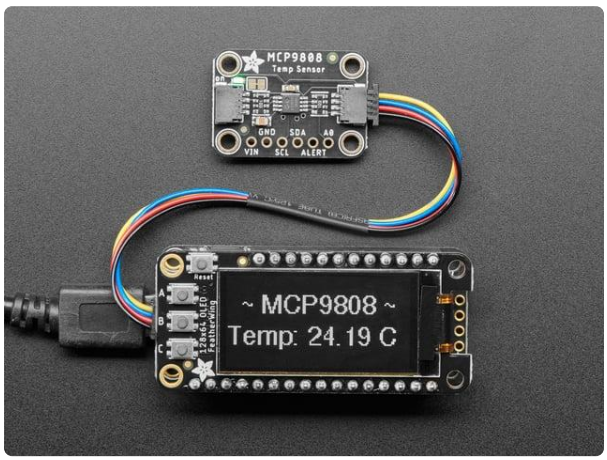
I2C and CircuitPython

CircuitPython supports many I2C devices, and makes it super simple to interact with them. There are libraries available for many I2C devices in the CircuitPython Library Bundle (https://adafru.it/Tra). (If you don't see the sensor you're looking for, keep checking back, more are being written all the time!)

In this section, you'll learn how to scan the I2C bus for all connected devices. Then you'll learn how to interact with an I2C device.

Necessary Hardware

You'll need the following additional hardware to complete the examples on this page.



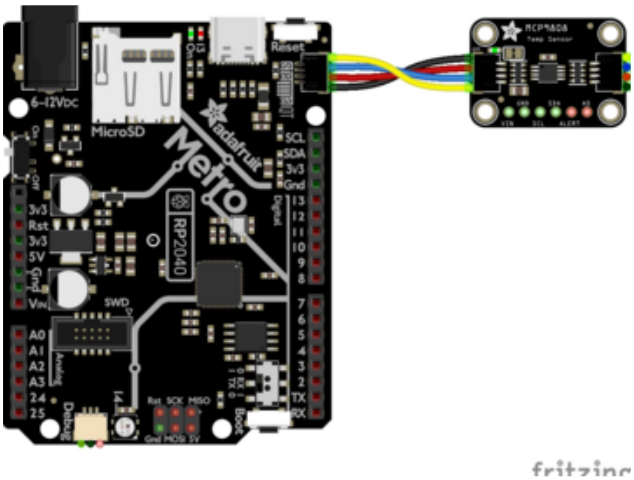
Adafruit MCP9808 High Accuracy I2C Temperature Sensor Breakout
The MCP9808 digital temperature sensor is one of the more accurate/precise we've ever seen, with a typical accuracy of ±0.25°C over the sensor's -40°C to...
https://www.adafruit.com/product/5027

While the examples here will be using the Adafruit MCP9808 (http://adafru.it/5027), a high accuracy temperature sensor, the overall process is the same for just about any I2C sensor or device.

The first thing you'll want to do is get the sensor connected so your board has I2C to talk to.

Wiring the MCP9808

The MCP9808 comes with a STEMMA QT connector, which makes wiring it up quite simple and solder-free.



Simply connect the STEMMA QT cable from **the STEMMA QT port on your board** to **the STEMMA QT port on the MCP9808**.

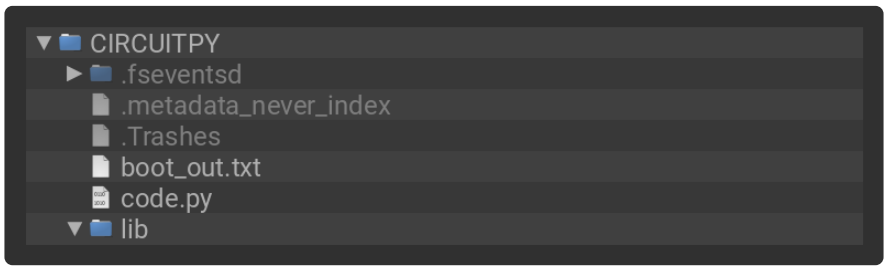
Find Your Sensor

The first thing you'll want to do after getting the sensor wired up, is make sure it's wired correctly. You're going to do an I2C scan to see if the board is detected, and if it is, print out its I2C address.

Save the following to your **CIRCUITPY** drive as **code.py**.

Click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, find your CircuitPython version, and copy the matching **code.py** file to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:




```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython I2C Device Address Scan"""
import time
import board

i2c = board.I2C()  # uses board.SCL and board.SDA
```

```
# i2c = board.STEMMA_I2C()  # For using the built-in STEMMA QT connector on a
microcontroller

# To create I2C bus on specific pins
# import busio
# i2c = busio.I2C(board.GP1, board.GP0)    # Pi Pico RP2040

while not i2c.try_lock():
    pass

try:
    while True:
        print(
            "I2C addresses found:",
            [hex(device_address) for device_address in i2c.scan()],
        )
        time.sleep(2)

finally:  # unlock the i2c bus when ctrl-c'ing out of the loop
    i2c.unlock()
```

```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disab
le.
code.py output:
I2C addresses found: ['0x18']
```

If you run this and it seems to hang, try manually unlocking your I2C bus by running the following two commands from the REPL.

```
import board
board.I2C().unlock()
```

First you create the `i2c` object, using `board.I2C()`. This convenience routine creates and saves a `busio.I2C` object using the default pins `board.SCL` and `board.SDA`. If the object has already been created, then the existing object is returned. No matter how many times you call `board.I2C()`, it will return the same object. This is called a singleton.

To be able to scan it, you need to lock the I2C down so the only thing accessing it is the code. So next you include a loop that waits until I2C is locked and then continues on to the scan function.

Last, you have the loop that runs the actual scan, `i2c_scan()`. Because I2C typically refers to addresses in hex form, the example includes this bit of code that formats the results into hex format: `[hex(device_address) for device_address in i2c.scan()]`.

Open the serial console to see the results! The code prints out an array of addresses. You've connected the MCP9808 which has a 7-bit I2C address of 0x18. The result for this sensor is `I2C addresses found: ['0x18']`. If no addresses are returned, refer back to the wiring diagrams to make sure you've wired up your sensor correctly.

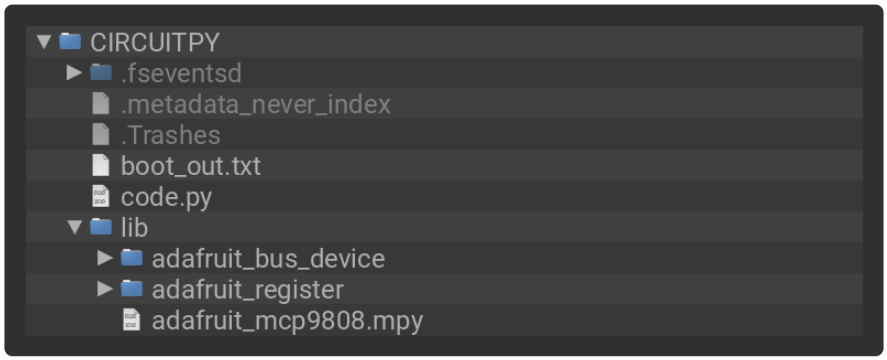
I2C Sensor Data

Now you know for certain that your sensor is connected and ready to go. Time to find out how to get the data from the sensor!

Save the following to your **CIRCUITPY** drive as **code.py**.

Click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, find your CircuitPython version, and copy the matching **entire lib folder** and **code.py** file to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:




```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython I2C MCP9808 Temperature Sensor Example"""
import time
import board
import adafruit_mcp9808

i2c = board.I2C()  # uses board.SCL and board.SDA
# i2c = board.STEMMA_I2C()  # For using the built-in STEMMA QT connector on a
microcontroller
# import busio
# i2c = busio.I2C(board.SCL1, board.SDA1) # For QT Py RP2040, QT Py ESP32-S2
mcp9808 = adafruit_mcp9808.MCP9808(i2c)

while True:
    temperature_celsius = mcp9808.temperature
    temperature_fahrenheit = temperature_celsius * 9 / 5 + 32
    print("Temperature: {:.2f} C {:.2f} F ".format(temperature_celsius,
temperature_fahrenheit))
    time.sleep(2)
```

```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disab
le.
code.py output:
Temperature: 23.38 C 74.07 F
```

This code begins the same way as the scan code, except this time, you create your sensor object using the sensor library. You call it `mcp9808` and provide it the `i2c` object.

Then you have a simple loop that prints out the temperature reading using the sensor object you created. Finally, there's a `time.sleep(2)`, so it only prints once every two seconds. Connect to the serial console to see the results. Try touching the MCP9808 with your finger to see the values change!

Where's my I2C?

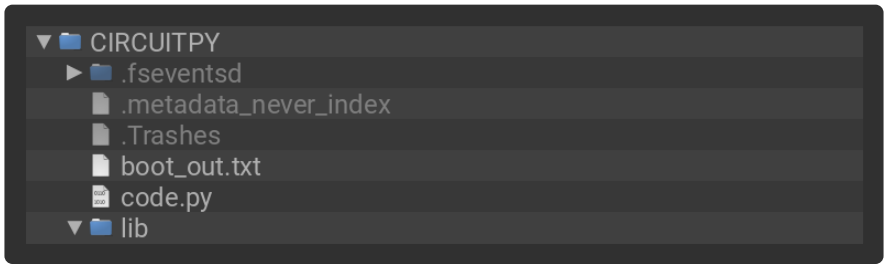
On many microcontrollers, you have the flexibility of using a wide range of pins for I2C. On some types of microcontrollers, any pin can be used for I2C! Other chips require using bitbangio, but can also use any pins for I2C. There are further microcontrollers that may have fixed I2C pins.

Given the many different types of microcontroller boards available, it's impossible to guarantee anything other than the labeled 'SDA' and 'SCL' pins. So, if you want some other setup, or multiple I2C interfaces, how will you find those pins? Easy! Below is a handy script.

Save the following to your **CIRCUITPY** drive as **code.py**.

Click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, find your CircuitPython version, and copy the matching **code.py** file to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:




```
# SPDX-FileCopyrightText: 2021-2023 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython I2C possible pin-pair identifying script"""
import board
import busio
from microcontroller import Pin


def is_hardware_i2c(scl, sda):
    try:
```

```
        p = busio.I2C(scl, sda)
        p.deinit()
        return True
    except ValueError:
        return False
    except RuntimeError:
        return True


def get_unique_pins():
    exclude = [
        getattr(board, p)
        for p in [
            # This is not an exhaustive list of unexposed pins. Your results
            # may include other pins that you cannot easily connect to.
            "NEOPIXEL",
            "DOTSTAR_CLOCK",
            "DOTSTAR_DATA",
            "APA102_SCK",
            "APA102_MOSI",
            "LED",
            "SWITCH",
            "BUTTON",
            "ACCELEROMETER_INTERRUPT",
            "VOLTAGE_MONITOR",
            "MICROPHONE_CLOCK",
            "MICROPHONE_DATA",
            "RFM_RST",
            "RFM_CS",
            "RFM_IO0",
            "RFM_IO1",
            "RFM_IO2",
            "RFM_IO3",
            "RFM_IO4",
            "RFM_IO5",
            "TFT_I2C_POWER",
            "NEOPIXEL_POWER",
        ]
        if p in dir(board)
    ]
    pins = [
        pin
        for pin in [getattr(board, p) for p in dir(board)]
        if isinstance(pin, Pin) and pin not in exclude
    ]
    unique = []
    for p in pins:
        if p not in unique:
            unique.append(p)
    return unique


for scl_pin in get_unique_pins():
    for sda_pin in get_unique_pins():
        if scl_pin is sda_pin:
            continue
        if is_hardware_i2c(scl_pin, sda_pin):
            print("SCL pin:", scl_pin, "\t SDA pin:", sda_pin)
```

Now, connect to the serial console and check out the output! The results print out a nice handy list of SCL and SDA pin pairs that support I2C.

The output for the Metro RP2040 is extremely long! The screenshot shows only the beginning. Run the script yourself to see the full output!

```
code.py output:
SCL pin: board.A1        SDA pin: board.A0
SCL pin: board.A1        SDA pin: board.D10
SCL pin: board.A1        SDA pin: board.D2
SCL pin: board.A1        SDA pin: board.D6
SCL pin: board.A1        SDA pin: board.SCK
SCL pin: board.A1        SDA pin: board.SDIO_DATA2
SCL pin: board.A3        SDA pin: board.A2
SCL pin: board.A3        SDA pin: board.D0
SCL pin: board.A3        SDA pin: board.D12
SCL pin: board.A3        SDA pin: board.D24
SCL pin: board.A3        SDA pin: board.D4
```

This example only runs once, so if you do not see any output when you connect to the serial console, try CTRL+D to reload.

---

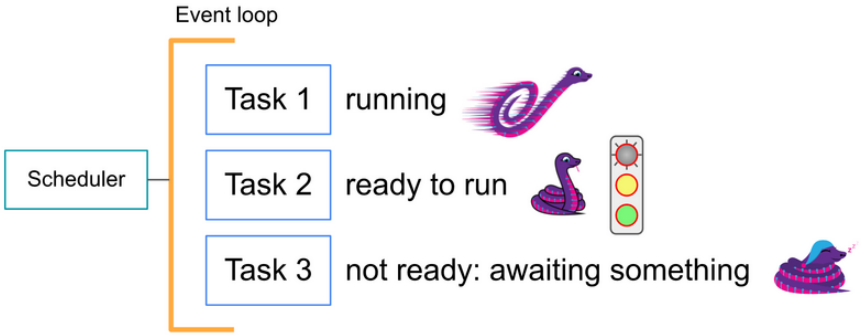
asyncio

CircuitPython uses the **asyncio** library to support cooperative multitasking (https://adafru.it/ZwB) in CircuitPython, which includes the `async` and `await` language keywords. Cooperative multitasking is a style of programming in which multiple tasks take turns running. Each task runs until it needs to wait for something, or until it decides it has run for long enough and should let another task run.

In cooperative multitasking, a scheduler manages the tasks. Only one task runs at a time. When a task gives up control and starts waiting, the scheduler starts another task that is ready to run. The scheduler runs an event loop which repeats this process over and over for all the tasks assigned to the event loop.

A task is a kind of coroutine (https://adafru.it/ZwB). A coroutine can stop in the middle of some code. When the coroutine is called again, it starts where it left off. A coroutine is declared with the keyword `async`, and the keyword `await` indicates that the coroutine is giving up control at that point.

This diagram shows the scheduler, running an event loop, with three tasks: Task 1 is running, Task 2 is ready to run, and is waiting for Task 1 to give up control, and Task 3 is waiting for something else, and isn't ready to run yet.

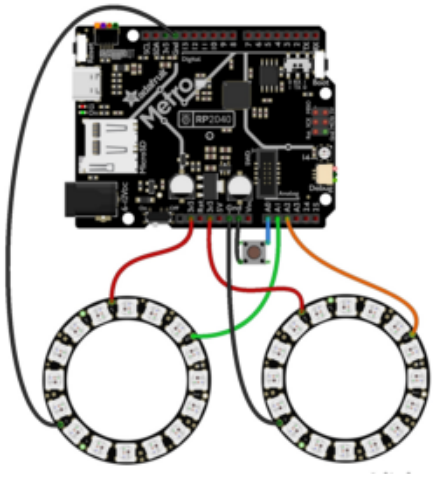


asyncio Demonstration

The example on this page demonstrates a basic use of asyncio. This uses a microcontroller and a button to control two animations displayed on two different NeoPixel rings. One ring displays a rainbow swirl, and the other displays a blink animation at a 0.5 second interval. Pressing the button reverses the direction of the rainbow swirl, and speeds up the blink animation to a 0.1 second interval. Releasing the button returns both to their initial states.

Wiring

The first step is wiring up the button and NeoPixel rings to your microcontroller.



NeoPixel Rings

NeoPixel ring one: data in (DIN) to **microcontroller A1**
NeoPixel ring one: ground to **microcontroller GND**
NeoPixel ring one: V+ to **microcontroller 3V3**
NeoPixel ring two: data in (DIN) to **microcontroller A2**
NeoPixel ring two: ground to **microcontroller GND**
NeoPixel ring two: V+ to **microcontroller 3V3**
Button

One leg of button to **microcontroller A0**
Opposite leg of button to **microcontroller GND**

asyncio Example Code

Once everything is wired up, the next step is to load the example code onto your microcontroller.

To run this example, you'll need to include a few libraries onto your **CIRCUITPY** drive. Then you need to update **code.py** with the example code.

Thankfully, this can be done in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, and copy the **entire lib folder** and the **code.py** file to your **CIRCUITPY** drive.

```
# SPDX-FileCopyrightText: Copyright (c) 2022 Dan Halbert for Adafruit Industries
# SPDX-FileCopyrightText: Copyright (c) 2022 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT
"""
CircuitPython asyncio example for two NeoPixel rings and one button.
"""
import asyncio
import board
import neopixel
import keypad
from rainbowio import colorwheel

button_pin = board.A0  # The pin the button is connected to.
num_pixels = 16  # The number of NeoPixels on a single ring.
brightness = 0.2  # The LED brightness.

# Set up NeoPixel rings.
ring_one = neopixel.NeoPixel(board.A1, num_pixels, brightness=brightness,
auto_write=False)
ring_two = neopixel.NeoPixel(board.A2, num_pixels, brightness=brightness,
auto_write=False)


class AnimationControls:
    """The controls to allow you to vary the rainbow and blink animations."""
    def __init__(self):
        self.reverse = False
        self.wait = 0.0
        self.delay = 0.5


async def rainbow_cycle(controls):
    """Rainbow cycle animation on ring one."""
    while True:
        for j in range(255, -1, -1) if controls.reverse else range(0, 256, 1):
            for i in range(num_pixels):
                rc_index = (i * 256 // num_pixels) + j
                ring_one[i] = colorwheel(rc_index & 255)
            ring_one.show()
            await asyncio.sleep(controls.wait)


async def blink(controls):
    """Blink animation on ring two."""
    while True:
        ring_two.fill((0, 0, 255))
        ring_two.show()
        await asyncio.sleep(controls.delay)
        ring_two.fill((0, 0, 0))
        ring_two.show()
        await asyncio.sleep(controls.delay)
        await asyncio.sleep(controls.wait)


async def monitor_button(button, controls):
    """Monitor button that reverses rainbow direction and changes blink speed.
    Assume button is active low.
    """
    with keypad.Keys((button,), value_when_pressed=False, pull=True) as key:
```

```
        while True:
            key_event = key.events.get()
            if key_event:
                if key_event.pressed:
                    controls.reverse = True
                    controls.delay = 0.1
                elif key_event.released:
                    controls.reverse = False
                    controls.delay = 0.5
            await asyncio.sleep(0)


async def main():
    animation_controls = AnimationControls()
    button_task = asyncio.create_task(monitor_button(button_pin,
animation_controls))
    animation_task = asyncio.create_task(rainbow_cycle(animation_controls))
    blink_task = asyncio.create_task(blink(animation_controls))

    # This will run forever, because no tasks ever finish.
    await asyncio.gather(button_task, animation_task, blink_task)

asyncio.run(main())
```

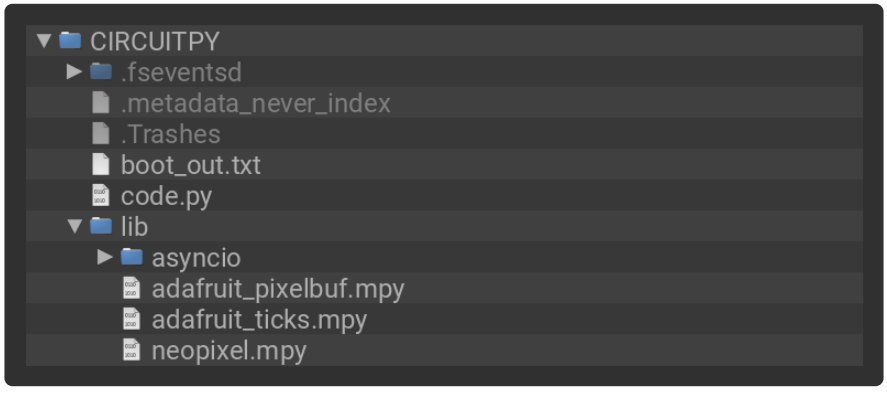
Your **CIRCUITPY** drive contents should resemble the image below.

You should have at least the following file in the top level of the **CIRCUITPY** drive:

- **code.py**

Your **CIRCUITPY/lib** folder should contain at least the following folder and files:

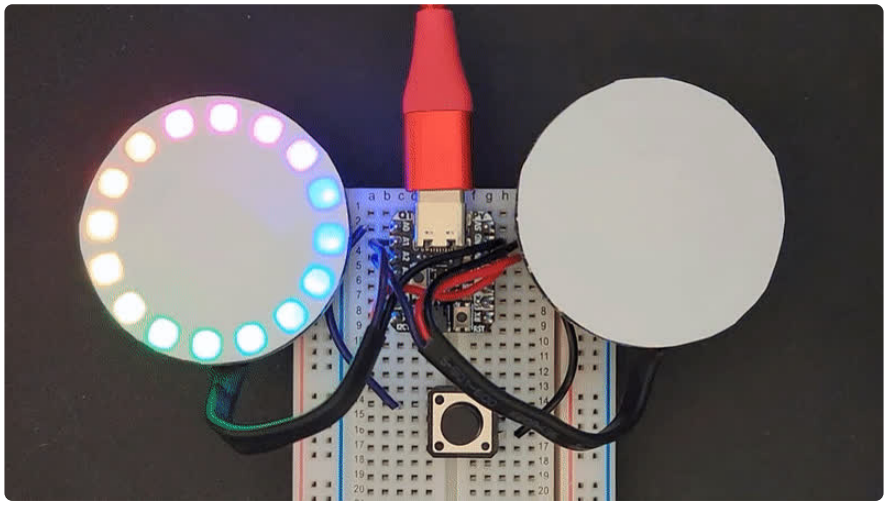
- **asyncio/**
- **adafruit_ticks.mpy**
- **neopixel.mpy**



Ring one will light up in a rainbow swirl. Ring two will begin blinking blue at a 0.5 second interval.

Now, press the button. The rainbow swirl on ring one will reverse direction, and the blinking on ring two will speed up!

Now release the button. The rainbow swirl on ring one returns to its original direction, and the blinking on ring two returns to its original speed!



Code Walkthrough

First you import the necessary modules and libraries.

```
import asyncio
import board
import neopixel
import keypad
from rainbowio import colorwheel
```

Then, you specify the button pin, the number of LEDs in each NeoPixel ring, and the LED brightness.

```
button_pin = board.A0
num_pixels = 16
brightness = 0.2
```

Next you set up the two NeoPixel rings on pins `A1` and `A2`, using the number of pixels and brightness specified above, and setting `auto_write=False`.

```
ring_one = neopixel.NeoPixel(board.A1, num_pixels, brightness=brightness,
auto_write=False)
ring_two = neopixel.NeoPixel(board.A2, num_pixels, brightness=brightness,
auto_write=False)
```

Following set up, you create a class called `AnimationControls`. This class provides ways to control the animations with asyncio.

```
class AnimationControls:
    def __init__(self):
        self.reverse = False
```

```
        self.wait = 0.0
        self.delay = 0.5
```

Then, you have the rainbow and blink animation code. This is where the asyncio-specific code begins.

In terms of the animation parts of the code, the first function is the rainbow cycle animation code. This is pretty standard except for the second line of code. In this example, the line beginning with `for j in` includes non-standard code for the rainbow cycle in reverse: `range(255, -1, -1) if controls.reverse`, followed by the standard forward rainbow cycle code - `range(0, 256, 1)`.

```
async def rainbow_cycle(controls):
    """Rainbow cycle animation on ring one."""
    while True:
        for j in range(255, -1, -1) if controls.reverse else range(0, 256, 1):
            for i in range(num_pixels):
                rc_index = (i * 256 // num_pixels) + j
                ring_one[i] = colorwheel(rc_index &amp;amp; 255)
            ring_one.show()
            await asyncio.sleep(controls.wait)
```

The second function is the blink animation code. This is typical. You fill all the NeoPixel LEDs blue, delay for a specified amount of time, then turn all of the LEDs off, and delay for the same specified amount of time.

```
async def blink(controls):
    """Blink animation on ring two."""
    while True:
        ring_two.fill((0, 0, 255))
        ring_two.show()
        await asyncio.sleep(controls.delay)
        ring_two.fill((0, 0, 0))
        ring_two.show()
        await asyncio.sleep(controls.delay)
        await asyncio.sleep(controls.wait)
```

In both functions, you must call `show()` on the NeoPixel ring object to get the animations to run because you set `auto_write=False` in the NeoPixel ring setup.

Notice that the controls object provides the animation direction (`controls.reverse`), the delay between steps of the animation (`controls.delay`), and the delay between complete animations (`controls.wait`).

In terms of the asyncio-specific parts of this code, you'll notice that both of these functions begin with `async def`. Every function that contains an `await` must be defined as `async def`, to indicate that it's a coroutine.

Both functions contain one or more `await` lines. What does `await` mean? `await` means "I need to wait for something; let other tasks run until I'm ready to resume." Both include `await asyncio.sleep()`. Basically, when when the code goes to "sleep", another task can be run. When the `sleep()` is over, this coroutine will resume.

The `blink()` includes the following line of code twice, which utilizes the `.delay` attribute of the `AnimationsControl` object.

```
await asyncio.sleep(controls.delay)
```

Both functions end with the following line of code which utilizes the `.wait` attribute of the `AnimationsControl` object.

```
await asyncio.sleep(controls.wait)
```

The next function is called `main()`.

In `main()`, first create a task. For the `button_task`, instantiate the `monitor_button()` coroutine by calling it with the arguments desired, and then pass that coroutine to `asyncio.create_task()`. `create_task()` wraps the coroutine in a task, and then schedules the task to run "soon". "Soon" means it will get a turn to run as soon other existing tasks have given up control.

Then the program uses `await asyncio.gather()`, which waits for all the tasks it's passed to finish.

```
async def main():
    animation_controls = AnimationControls()
    button_task = asyncio.create_task(monitor_button(button_pin,
animation_controls))
    animation_task = asyncio.create_task(rainbow_cycle(animation_controls))
    blink_task = asyncio.create_task(blink(animation_controls))

    await asyncio.gather(button_task, animation_task, blink_task)
```

Finally, `run` the `main()` function to execute the code within.

```
asyncio.run(main())
```

My program ended? What happened?

`await.gather(...)` runs until all the listed tasks have finished. If `gather` completes, that means all the tasks listed have finished.

The most common causes of a task ending are:

- an exception occurred causing the task to end
- the task function finished

If you want to ensure the task executes forever, have a loop in your task function (e.g. a `while True` loop).

The following example is greatly oversimplified, but demonstrates what including a loop in your task function might look like.

```
async def never_ending_task():
    while True:
        print("I'm looping!")
        await asyncio.sleep(0)
```

I2S

I2S, or Inter-IC Sound, is a standard for transmitting digital audio data. It requires at least three connections. The first connection is a clock, called **bit clock** (**BCLK**, or sometimes written as serial clock or SCK)**.** The second connection, which determines the channel (left or right) being sent, is called **word select** (**WS**). When stereo data is sent, WS is toggled so that the left and right channels are sent alternately, one data word at a time. The third connection, which transmits the data, is called **serial data** (**SD**).

Typically, there is a **transmitter** device which generates the bit clock, word select signal, and the data, and sends them to a **receiver** device. In this case, your microcontroller acts as the transmitter, and an I2S breakout acts as the receiver. The MAX98357A (http://adafru.it/3006) is an example of an I2S class D amplifier that allows you to connect directly to a speaker such as this one (http://adafru.it/4445).

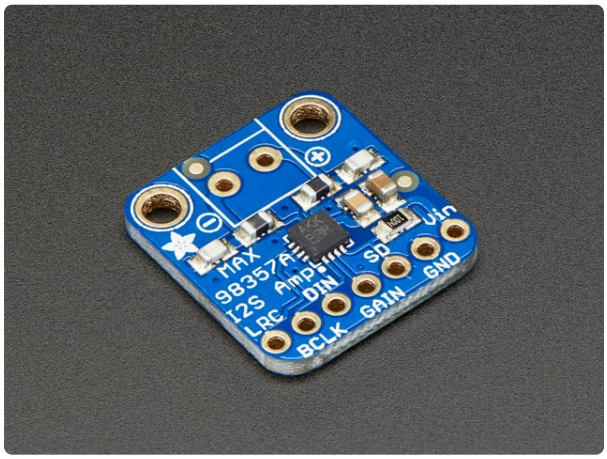
I2S and CircuitPython

CircuitPython supports sending I2S audio signals using the `audiobusio` module, making it simple to use the I2S interface with your microcontroller.

In this section, you'll learn how to use CircuitPython to play different types of audio using I2S, including tones and WAV files.

Necessary Hardware

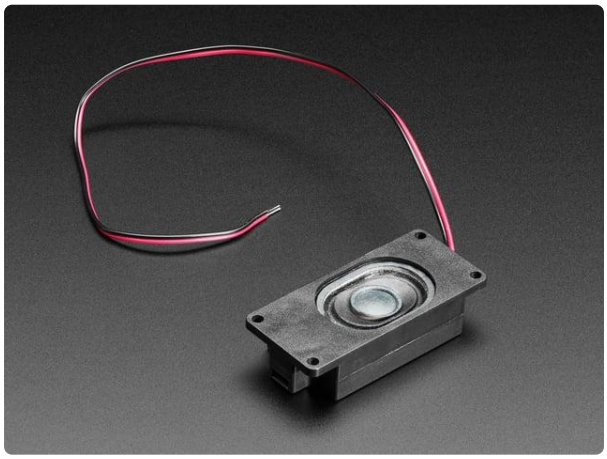
You'll need the following additional hardware to complete the examples on this page.



Adafruit I2S 3W Class D Amplifier Breakout - MAX98357A

Listen to this good news - we now have an all in one digital audio amp breakout board that works incredibly well with the

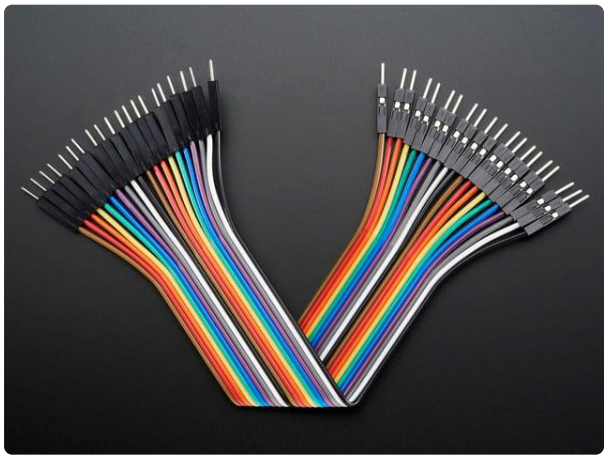
https://www.adafruit.com/product/3006



Mono Enclosed Speaker with Plain Wires - 3W 4 Ohm

Listen up! This single 2.8" x 1.2" speaker is the perfect addition to any audio project where you need 4 ohm impedance and 3W or less of power. We...

https://www.adafruit.com/product/4445



Premium Male/Male Jumper Wires - 20 x 6" (150mm)

These Male/Male Jumper Wires are handy for making wire harnesses or jumpering between headers on PCB's. These premium jumper wires are 6" (150mm) long and come in a...

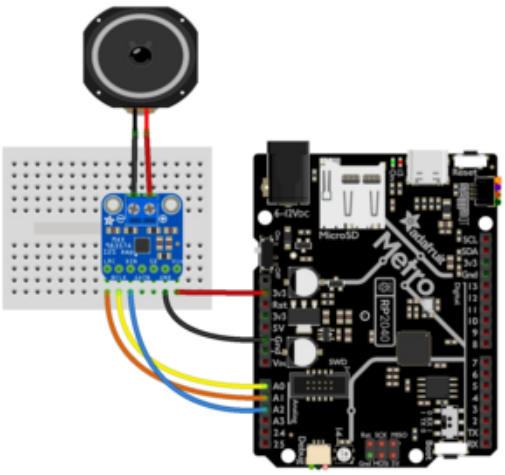
https://www.adafruit.com/product/1957

Wiring the MAX98357A

Connect the MAX98357A breakout to your microcontroller as follows.

Make sure you have a very solid connection between the breakout ground pin and the ground pin on your board! An insufficient connection here can result in raspy-sounding tones with intermittent volume increases. If you experience this result, try reseating your ground connection.

The bit clock and word select pins must be on consecutive pins! They can be on any pins you like, but they must be in consecutive order, for example, A0 for bit clock and A1 for word select.



Metro 3V3 to **breakout VIN**
Metro GND to **breakout GND**
Metro A0 to **breakout BCLK**
Metro A1 to **breakout LRC**
Metro A2 to **breakout DIN**
Speaker + to **screw terminal** +
Speaker - to **screw terminal** -

I2S Tone Playback

The first example generates one period of a sine wave and then loops it to generate a tone. You can change the volume and the frequency (in Hz) of the tone by changing the associated variables. Inside the loop, you play the tone for one second and stop it for one second.

Update your **code.py** to the following.

Click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the folder that matches your CircuitPython version, and copy the **code.py** file to your **CIRCUITPY** drive.

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
CircuitPython I2S Tone playback example.
Plays a tone for one second on, one
second off, in a loop.
"""
import time
import array
import math
```

```
import audiocore
import board
import audiobusio

audio = audiobusio.I2SOut(board.A0, board.A1, board.A2)

tone_volume = 0.1  # Increase this to increase the volume of the tone.
frequency = 440  # Set this to the Hz of the tone you want to generate.
length = 8000 // frequency
sine_wave = array.array("h", [0] * length)
for i in range(length):
    sine_wave[i] = int((math.sin(math.pi * 2 * i / length)) * tone_volume * (2 ** 15
- 1))
sine_wave_sample = audiocore.RawSample(sine_wave)

while True:
    audio.play(sine_wave_sample, loop=True)
    time.sleep(1)
    audio.stop()
    time.sleep(1)
```

Now you'll hear one second of a 440Hz tone, and one second of silence.

You can try changing the `440` Hz of the tone to produce a tone of a different pitch. Try changing the number of seconds in `time.sleep()` to produce longer or shorter tones.

I2S WAV File Playback

The second example plays a WAV file. You open the file in a readable format. Then, you play the file and, once finished, print `Done playing!` to the serial console. You can use any supported wave file (https://adafru.it/BRj).

Update your **code.py** to the following.

Click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the folder that matches your CircuitPython version, and copy the **StreetChicken.wav** file and the **code.py** file to your **CIRCUITPY** drive.

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
CircuitPython I2S WAV file playback.
Plays a WAV file once.
"""
import audiocore
import board
import audiobusio

audio = audiobusio.I2SOut(board.A0, board.A1, board.A2)

with open("StreetChicken.wav", "rb") as wave_file:
    wav = audiocore.WaveFile(wave_file)
```

```
    print("Playing wav file!")
    audio.play(wav)
    while audio.playing:
        pass

print("Done!")
```

Now you'll hear the wave file play, and on completion, print `Done Playing!` to the serial console.

You can play a different WAV file by updating `"StreetChicken.wav"` to be the name of your CircuitPython-compatible WAV file.

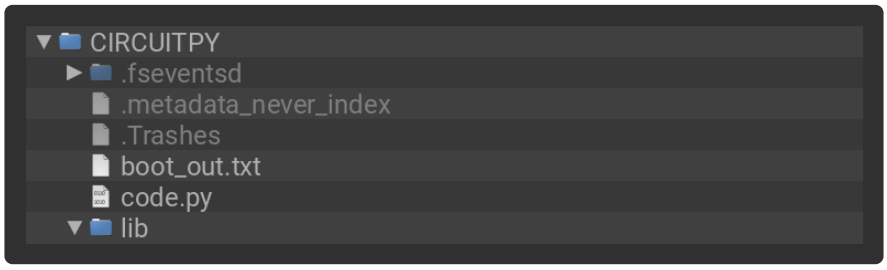
You can do other things while the WAV file plays! There is a `pass` in this example where you can include other code, such as code to blink an LED.

CircuitPython I2S-Compatible Pin Combinations

I2S audio is supported on specific pins. The good news is, there's a simple way to find out which pins support audio playback.

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Templates/i2s_find_pins/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:



Then, connect to the serial console (https://adafru.it/Bec) to see a list of pins printed out. This file runs only once, so if you do not see anything in the output, press CTRL+D to reload and run the code again.

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
CircuitPython I2S Pin Combination Identification Script
"""
import board
import audiobusio
```

```
from microcontroller import Pin


def is_hardware_i2s(bit_clock, word_select, data):
    try:
        p = audiobusio.I2SOut(bit_clock, word_select, data)
        p.deinit()
        return True
    except ValueError:
        return False


def get_unique_pins():
    exclude = [
        getattr(board, p)
        for p in [
            # This is not an exhaustive list of unexposed pins. Your results
            # may include other pins that you cannot easily connect to.
            "NEOPIXEL",
            "DOTSTAR_CLOCK",
            "DOTSTAR_DATA",
            "APA102_SCK",
            "APA102_MOSI",
            "LED",
            "SWITCH",
            "BUTTON",
        ]
        if p in dir(board)
    ]
    pins = [
        pin
        for pin in [getattr(board, p) for p in dir(board)]
        if isinstance(pin, Pin) and pin not in exclude
    ]
    unique = []
    for p in pins:
        if p not in unique:
            unique.append(p)
    return unique


for bit_clock_pin in get_unique_pins():
    for word_select_pin in get_unique_pins():
        for data_pin in get_unique_pins():
            if bit_clock_pin is word_select_pin or bit_clock_pin is data_pin or
word_select_pin \
                    is data_pin:
                continue
            if is_hardware_i2s(bit_clock_pin, word_select_pin, data_pin):
                print("Bit clock pin:", bit_clock_pin, "\t Word select pin:",
word_select_pin,
                      "\t Data pin:", data_pin)
            else:
                pass
```

For details about the I2S API, check out the CircuitPython docs (https://adafru.it/UFh).

PWM Audio

CircuitPython comes with `audiopwmio`, which provides built-in audio output support using **p**ulse **w**idth **m**odulation (PWM).

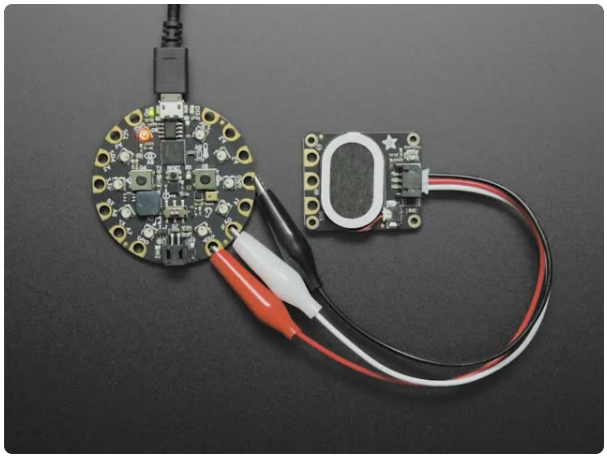
PWM converts the audio signal to a series of rectangular waveforms, or frequencies. By varying the PWM frequency, you can generate tones and play, pause and resume audio files. The faster your microcontroller is, the better sound quality you'll be able to achieve with PWM audio.

audiopwmio Documentation

https://adafru.it/18vd

Necessary Hardware

You'll need the following additional hardware to complete the examples on this page.



Adafruit STEMMA Speaker - Plug and Play Audio Amplifier

Hey, have you heard the good news? With Adafruit STEMMA boards you can easily and safely plug sensors and devices together, like this Adafruit STEMMA Speaker - Plug and Play...

https://www.adafruit.com/product/3885



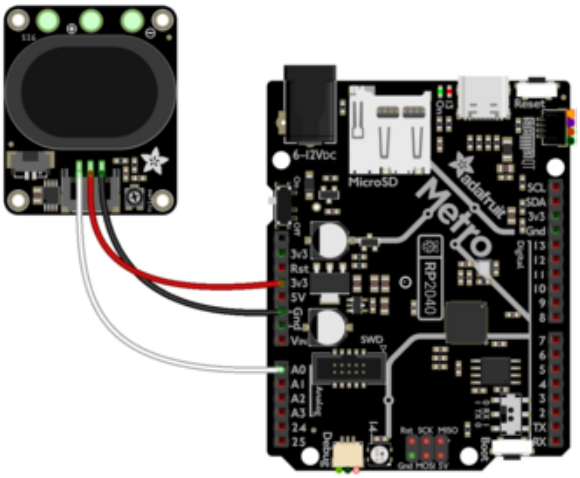
STEMMA JST PH 2mm 3-Pin to Male Header Cable - 200mm

This cable will let you turn a JST PH 3-pin cable port into 3 individual wires with high-quality 0.1" male header plugs on the end. We're carrying these to match up with our...

https://www.adafruit.com/product/3893

Wiring the STEMMA Speaker

Connect the STEMMA Speaker to your microcontroller as follows.



Speaker SIG (white wire) to **Metro A0**
Speaker VDD (red wire) to **Metro 3V3**
Speaker GND (black wire) to **Metro 3V3**

PWM Tone Playback

The first example generates one period of a sine wave and then loops it to generate a tone. You can change the volume and the frequency (in Hz) of the tone by changing the associated variables. Inside the loop, you play the tone for one second and stop it for one second.

Update your **code.py** to the following.

Click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the folder that matches your CircuitPython version, and copy the **code.py** file to your **CIRCUITPY** drive.

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
CircuitPython PWM Audio Out tone example
Plays a tone for one second on, one
second off, in a loop.
"""
import time
import array
import math
import board
from audiocore import RawSample
from audiopwmio import PWMAudioOut as AudioOut

audio = AudioOut(board.A0)

tone_volume = 0.1  # Increase this to increase the volume of the tone.
frequency = 440  # Set this to the Hz of the tone you want to generate.
length = 8000 // frequency
sine_wave = array.array("H", [0] * length)
for i in range(length):
    sine_wave[i] = int((1 + math.sin(math.pi * 2 * i / length)) * tone_volume * (2
** 15 - 1))
```

```
sine_wave_sample = RawSample(sine_wave)

while True:
    audio.play(sine_wave_sample, loop=True)
    time.sleep(1)
    audio.stop()
    time.sleep(1)
```

Now you'll hear one second of a 440Hz tone, and one second of silence.

You can try changing the `440` Hz of the tone to produce a tone of a different pitch. Try changing the number of seconds in `time.sleep()` to produce longer or shorter tones.

PWM WAV File Playback

The second example plays a WAV file. You open the file in a readable format. Then, you play the file and, once finished, print `Done playing!` to the serial console. You can use any supported wave file (https://adafru.it/BRj).

Update your **code.py** to the following.

Click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the folder that matches your CircuitPython version, and copy the **StreetChicken.wav** file and the **code.py** file to your **CIRCUITPY** drive.

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
CircuitPython PWM Audio Out WAV example
Play a WAV file once.
"""
import board
from audiocore import WaveFile
from audiopwmio import PWMAudioOut as AudioOut

audio = AudioOut(board.A0)

with open("StreetChicken.wav", "rb") as wave_file:
    wave = WaveFile(wave_file)
    print("Playing wav file!")
    audio.play(wave)
    while audio.playing:
        pass

print("Done!")
```

Now you'll hear the wave file play, and on completion, print `Done Playing!` to the serial console.

You can play a different WAV file by updating `"StreetChicken.wav"` to be the name of your CircuitPython-compatible WAV file.

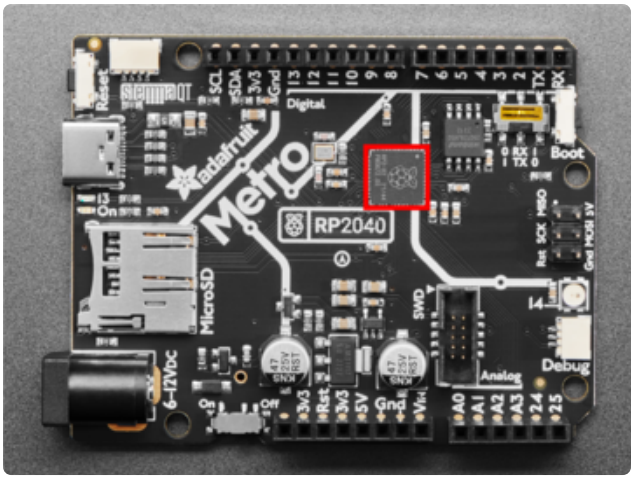
You can do other things while the WAV file plays! There is a `pass` in this example where you can include other code, such as code to blink an LED.

CPU Temperature

There is a temperature sensor built into the CPU on your microcontroller board. It reads the internal CPU temperature, which varies depending on how long the board has been running or how intense your code is.

CircuitPython makes it really simple to read this data from the temperature sensor built into the microcontroller. Using the built-in `microcontroller` module, you can easily read the temperature.

Microcontroller Location



The **RP2040 microcontroller** (highlighted in red) is located to the upper right of the RP2040 label on the board silk.

Reading the Microcontroller Temperature

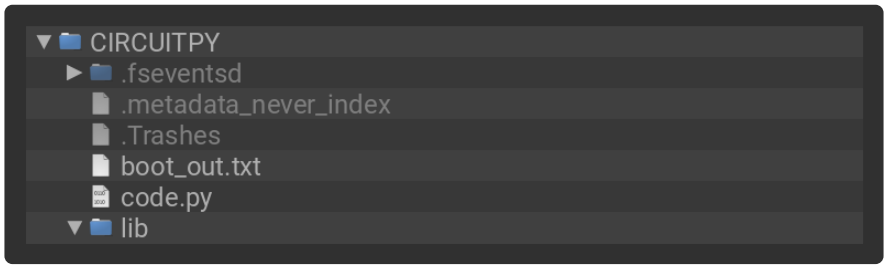
The data is read using two lines of code. All necessary modules are built into CircuitPython, so you don't need to download any extra files to get started.

Connect to the serial console (https://adafru.it/Bec), and then update your **code.py** to the following.

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Templates/cpu_temperature/** and then click on

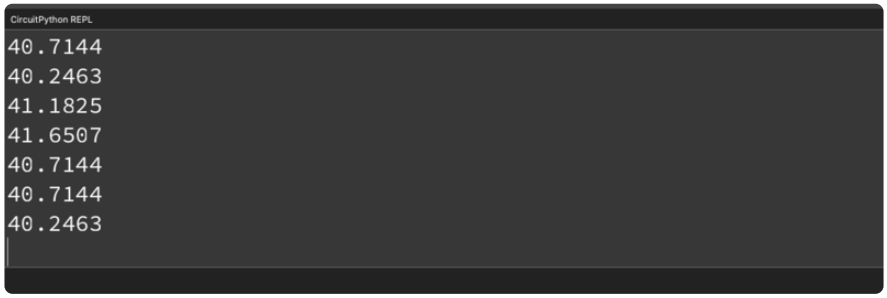
the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:




```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython CPU temperature example in Celsius"""
import time
import microcontroller

while True:
    print(microcontroller.cpu.temperature)
    time.sleep(0.15)
```



The CPU temperature in Celsius is printed out to the serial console!

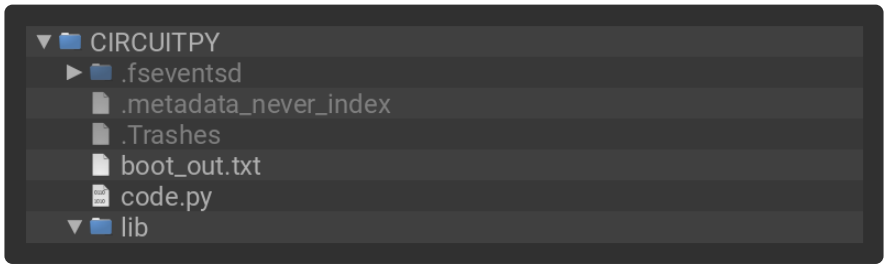
Try putting your finger on the microcontroller to see the temperature change.

The code is simple. First you import two modules: `time` and `microcontroller`. Then, inside the loop, you print the microcontroller CPU temperature, and the `time.sleep()` slows down the print enough to be readable. That's it!

You can easily print out the temperature in Fahrenheit by adding a little math to your code, using this simple formula: Celsius * (9/5) + 32.

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Templates/cpu_temperature_f/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:




```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython CPU temperature example in Fahrenheit"""
import time
import microcontroller

while True:
    print(microcontroller.cpu.temperature * (9 / 5) + 32)
    time.sleep(0.15)
```



The CPU temperature in Fahrenheit is printed out to the serial console!

That's all there is to reading the CPU temperature using CircuitPython!

Arduino IDE Setup

The **Arduino Philhower core** (https://adafru.it/ToC) provides support for RP2040 microcontroller boards. This page covers getting your Arduino IDE set up to include your board.

Arduino IDE Download

The first thing you will need to do is to download the latest release of the Arduino IDE. The Philhower core requires **version 1.8** or higher.

Arduino IDE Download

https://adafru.it/Pd5

Download and install it to your computer.

Once installed, open the Arduino IDE.

Adding the Philhower Board Manager URL

In the Arduino IDE, and navigate to the **Preferences** window. You can access it through **File** > **Preferences** on Windows or Linux, or **Arduino** > **Preferences** on OS X.

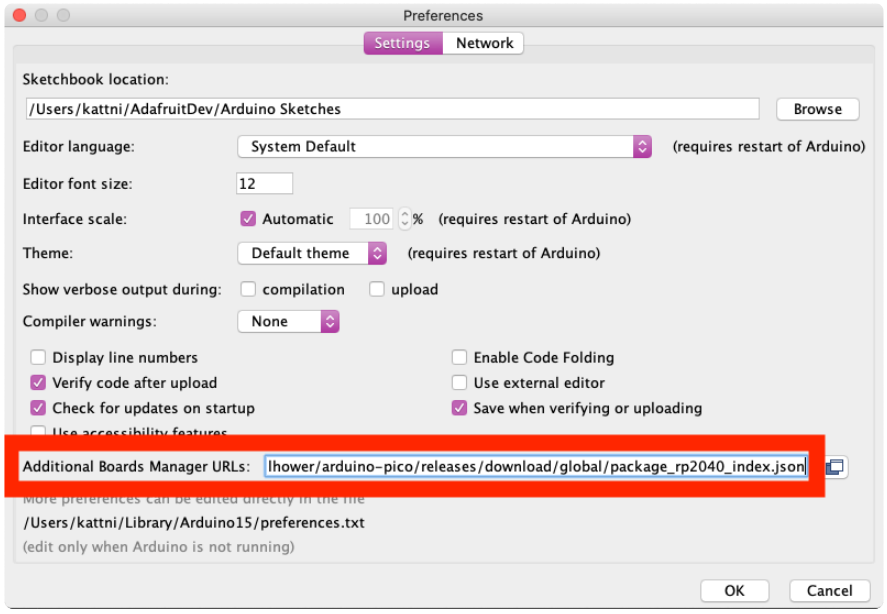
The **Preferences** window will open.

In the **Additional Boards Manager URLs** field, you'll want to add a new URL. The list of URLs is comma separated, and you will only have to add each URL once. The URLs point to index files that the Board Manager uses to build the list of available & installed boards.

Copy the following URL.

```
https://github.com/earlephilhower/arduino-pico/releases/download/global/package_rp2040_index.json
```

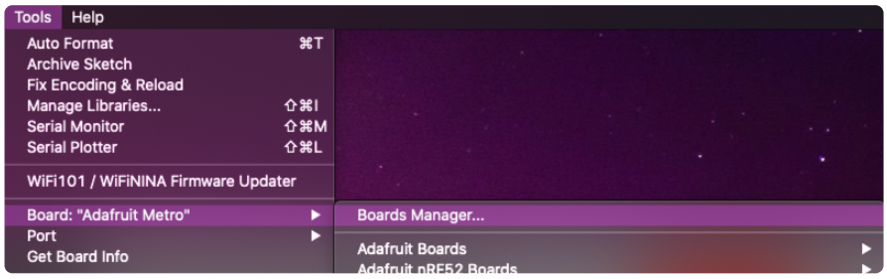
Add the URL to the the **Additional Boards Manager URLs** field (highlighted in red below).



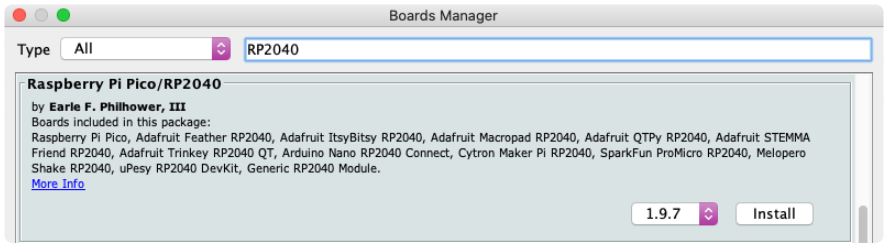
Click **OK** to save and close **Preferences**.

Add Board Support Package

In the Arduino IDE, click on **Tools** > **Board** > **Boards Manager**. If you have previously selected a board, the **Board** menu item may have a board name after it.



In the **Boards Manager**, search for RP2040. Scroll down to the **Raspberry Pi Pico/RP2040 by Earle F Philhower, III** entry. Click **Install** to install it.

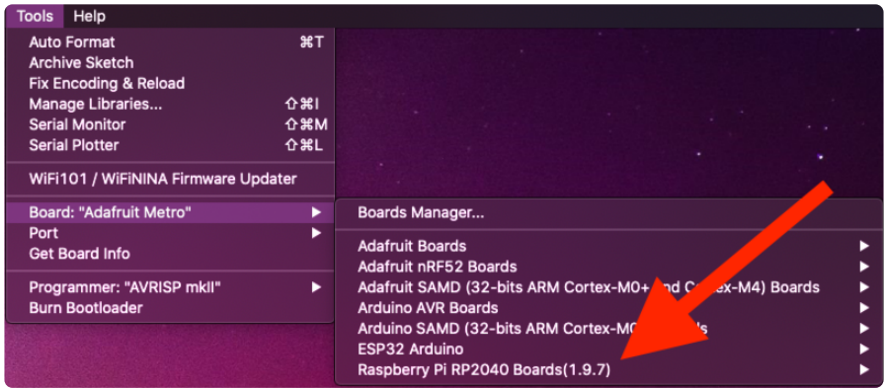


Installing a new board package can take a few minutes. Don't click Cancel!

Once installation is complete, click **Close** to close the Boards Manager.

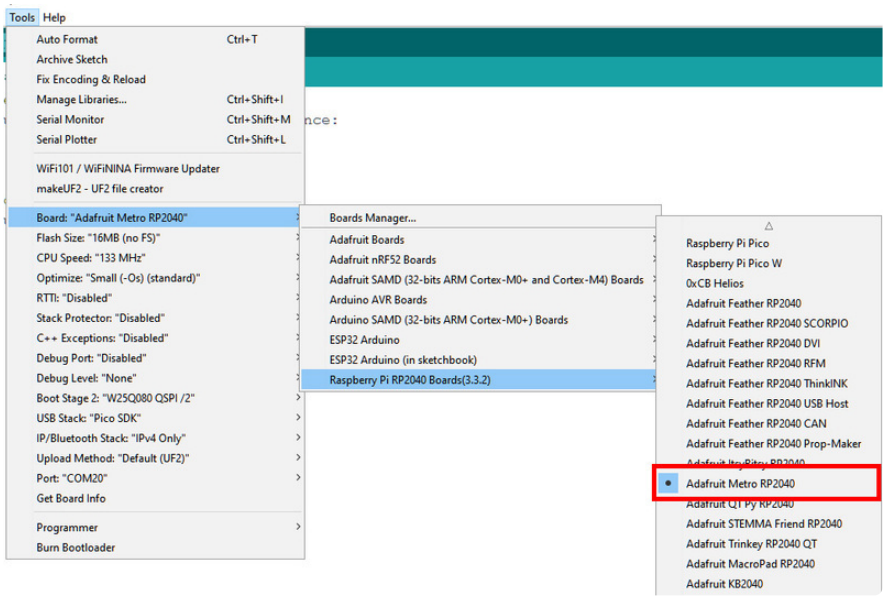
Choose Your Board

In the **Tools** > **Boards** menu, you should now see **Raspberry Pi RP2040 Boards** (possibly followed by a version number).



Navigate to the **Raspberry Pi RP2040 Boards** menu. You will see the available boards listed.

Navigate to the **Raspberry Pi RP2040 Boards** menu and choose **Adafruit Metro RP2040**.



If there is no serial Port available in the dropdown, or an invalid one appears - don't worry about it! The RP2040 does not actually use a serial port to upload, so its OK if it does not appear if in manual bootload mode. You will see a serial port appear after uploading your first sketch.

Now you're ready to begin using Arduino with your RP2040 board!

Arduino Usage

Now that you've set up the Arduino IDE with the Philhower RP2040 Arduino core, you're ready to start using Arduino with your RP2040.

RP2040 Arduino Pins

There is no pin remapping for Arduino on the RP2040. Therefore, the pin names on the top of the board are **not** the pin names used for Arduino. The Arduino pin names are the RP2040 GPIO pin names.

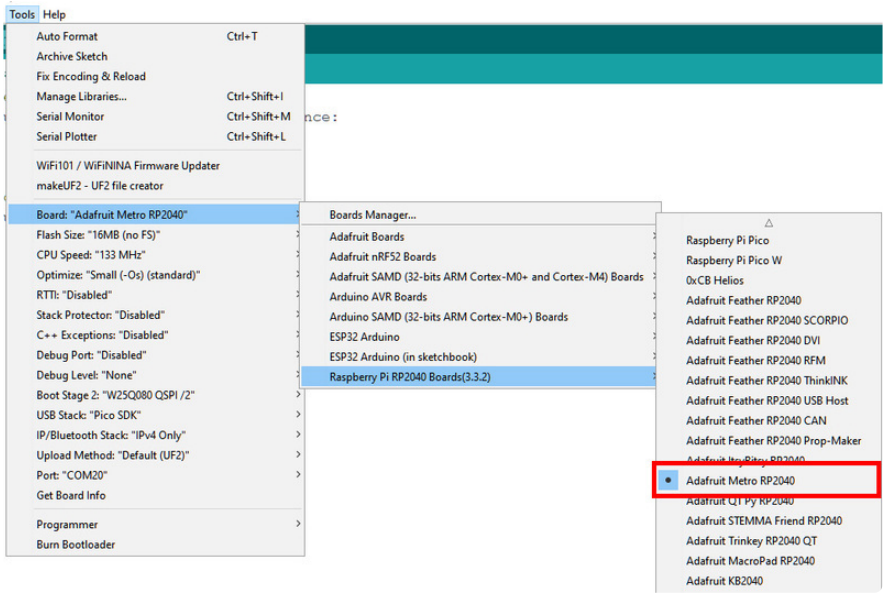
To find the Arduino pin name, check the PrettyPins diagram found on the Pinouts page. Each GPIO pin in the diagram has a **GPIOx** pin name listed, where **x** is the pin number. The Arduino pin name is the number following **GPIO**. For example, **GPIO5** would be Arduino pin `5`.

The Metro RP2040 has the GPIO pin names listed on the front of the board, next to the header sockets, as **X**, where **X** is the pin number. So, for example, pin **5** would be Arduino pin `5`.

Choose Your Board

Navigate to the **Tools** > **Boards** > **Raspberry Pi RP2040 Boards** menu. The Raspberry PI RP2040 Boards menu name may be followed by a version number.

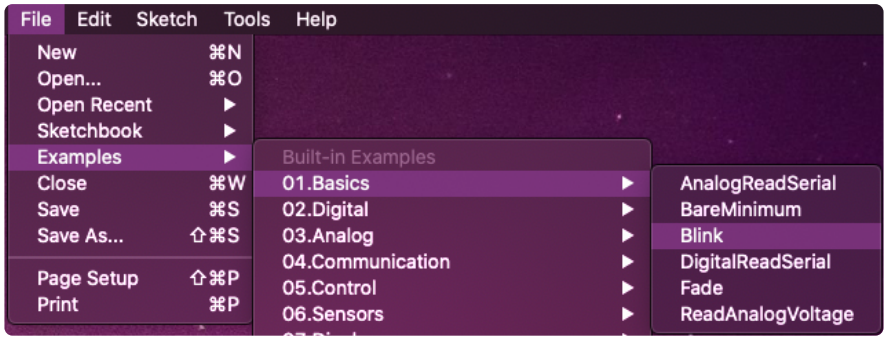
Choose **Adafruit Metro RP2040** from the menu.



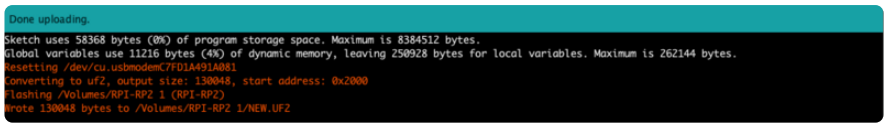
Load the Blink Sketch

Begin by plugging in your board to your computer, and wait a moment for it to be recognised by the OS. It will create a COM/serial port that you can now select from the **Tools** > **Port** menu dropdown.

Open the Blink sketch by clicking through **File** > **Examples** > **01.Basics** > **Blink**.



Click Upload. A successful upload will result in text similar to the following.



Once complete, the little red LED will begin blinking once every second! Try changing up the `delay()` timing to change the rate at which the LED blinks.

Manually Enter the Bootloader

If you get into a state with the bootloader where you can no longer upload a sketch, or you have uploaded code that crashes and doesn't auto-reboot into the bootloader, you may have to manually enter the bootloader.

To enter the bootloader, hold down the **Boot button**, and while continuing to hold it (don't let go!), press and release the **reset button**. **Continue to hold the Boot button until the RPI-RP2 drive appears!**

Once the RPI-RP2 drive shows up, your board is in bootloader mode. There will not be a port available in bootloader mode, this is expected.

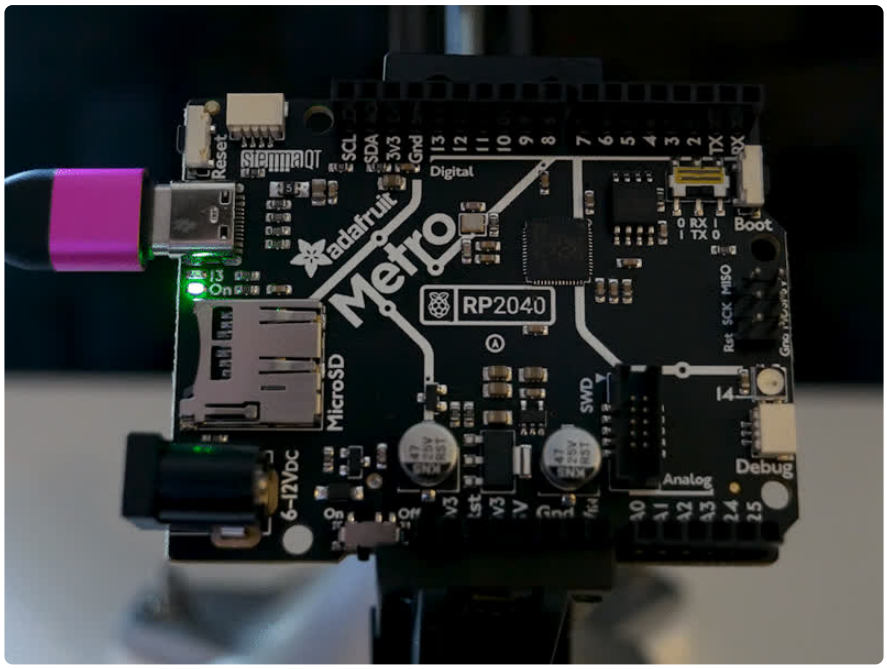
Once you see RPI-RP2 drive, make sure you are no longer holding down any buttons (reset or boot0 button).

Now, click Upload on your sketch to try again.

Blink

The first and most basic program you can upload to your Arduino is the classic Blink sketch. This takes something on the board and makes it, well, blink! On and off. It's a great way to make sure everything is working and you're uploading your sketch to the right board and right configuration.

When all else fails, you can always come back to Blink!



Pre-Flight Check: Get Arduino IDE & Hardware Set Up

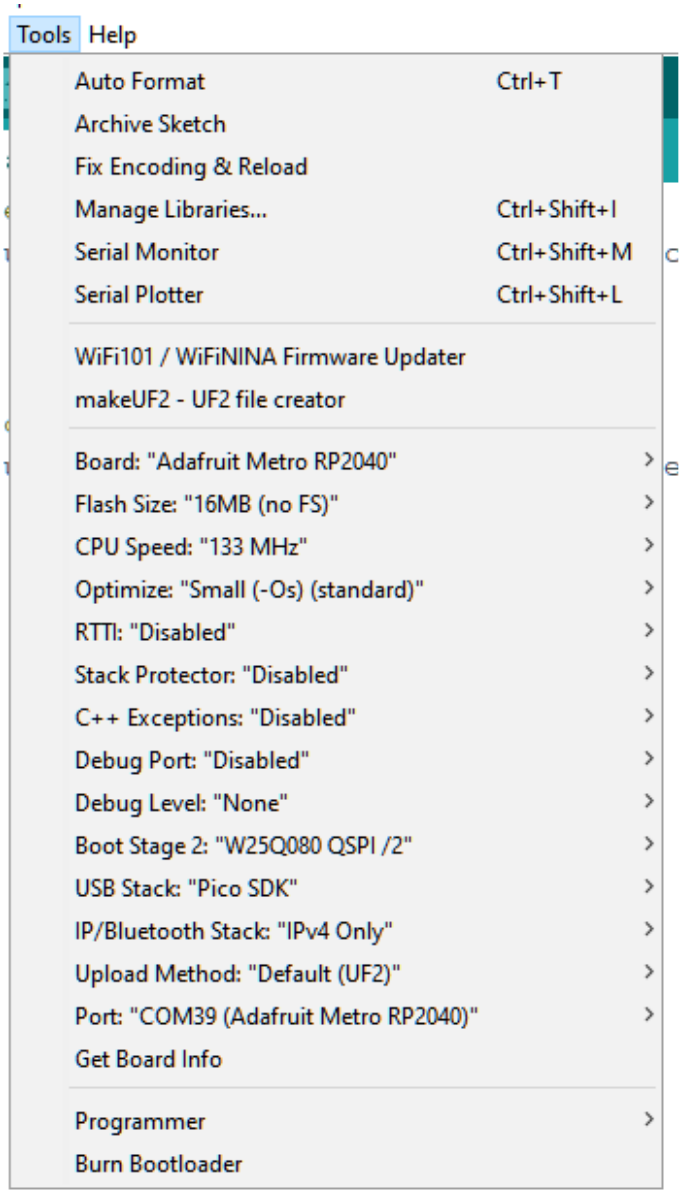
This lesson assumes you have Arduino IDE set up. This is a generalized checklist, some elements may not apply to your hardware. If you haven't yet, check the previous steps in the guide to make sure you:

- **Install the very latest Arduino IDE for Desktop** (not all boards are supported by the Web IDE so we don't recommend it).
- **Install any board support packages (BSP) required for your hardware.** Some boards are built in defaults on the IDE, but lots are not! You may need to install plug-in support which is called the BSP.
- **Get a Data/Sync USB cable for connecting your hardware.** A significant amount of problems folks have stem from not having a USB cable with data pins. Yes, these cursed cables roam the land, making your life hard. If you find a USB cable that doesn't work for data/sync, throw it away immediately! There is no need to keep it around, cables are very inexpensive these days.
- **Install any drivers required** - If you have a board with a FTDI or CP210x chip, you may need to get separate drivers. If your board has native USB, it probably doesn't need anything. After installing, reboot to make sure the driver sinks in.
- **Connect the board to your computer.** If your board has a power LED, make sure its lit. Is there a power switch? Make sure its turned On!

Start up Arduino IDE and Select Board/Port

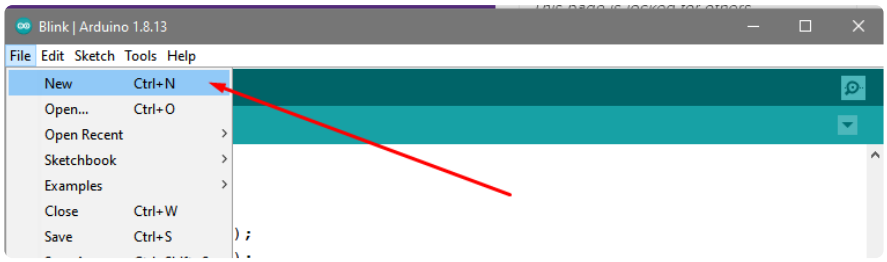
OK now you are prepared! Open the Arduino IDE on your computer. Now you have to tell the IDE what board you are using, and how you want to connect to it.

In the IDE find the **Tools** menu. You will use this to select the board. If you switch boards, you must switch the selection! So always double-check before you upload code in a new session.



New Blink Sketch

OK lets make a new blink sketch! From the **File** menu, select **New**



Then in the new window, copy and paste this text:

```
int led = LED_BUILTIN;

void setup() {
  // Some boards work best if we also make a serial connection
  Serial.begin(115200);

  // set LED to be an output pin
  pinMode(led, OUTPUT);
}

void loop() {
  // Say hi!
  Serial.println("Hello!");

  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(500);                // wait for a half second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(500);                // wait for a half second
}
```

Note that in this example, we are not only blinking the LED but also printing to the Serial monitor, think of it as a little bonus to test the serial connection.

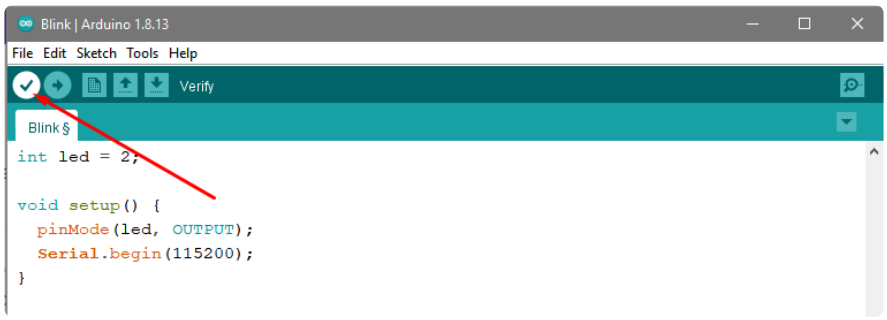
One note you'll see is that we reference the LED with the constant `LED_BUILTIN` rather than a number. That's because, historically, the built in LED was on pin 13 for Arduinos. But in the decades since, boards don't always have a pin 13, or maybe it could not be used for an LED. So the LED could have moved to another pin. It's best to use `LED_BUILTIN` so you don't get the pin number confused!

The LED is directly below the USB port on the front of the board. It is labeled **13** on the board silk.

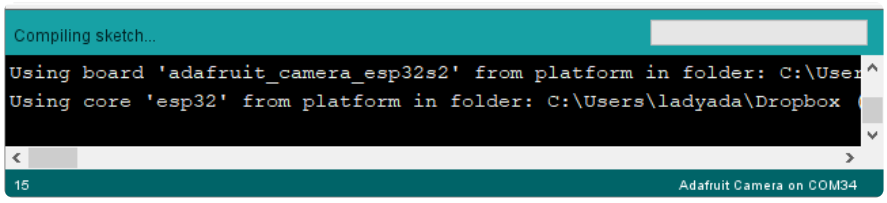
Verify (Compile) Sketch

OK now you can click the Verify button to convert the sketch into binary data to be uploaded to the board.

Note that Verifying a sketch is the same as Compiling a sketch - so we will use the words interchangeably

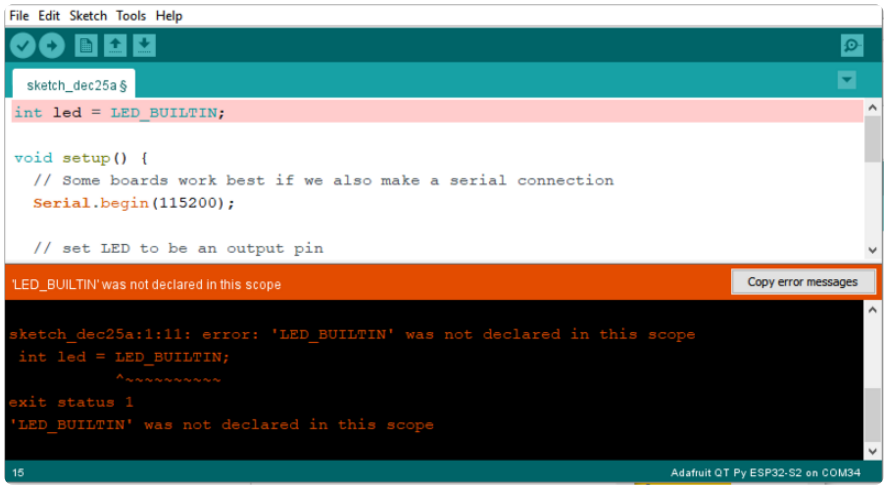


During verification/compilation, the computer will do a bunch of work to collect all the libraries and code and the results will appear in the bottom window of the IDE.

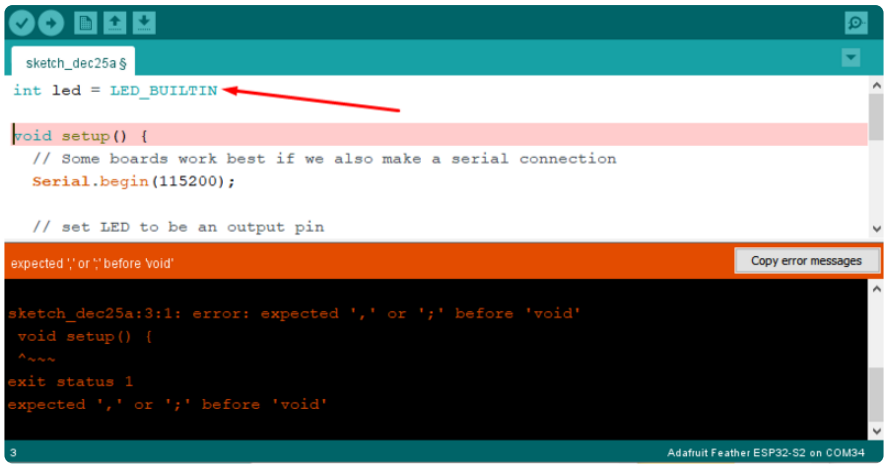


If something went wrong with compilation, you will get red warning/error text in the bottom window letting you know what the error was. It will also highlight the line with an error.

For example, here I had the wrong board selected - and the selected board does not have a built in LED!

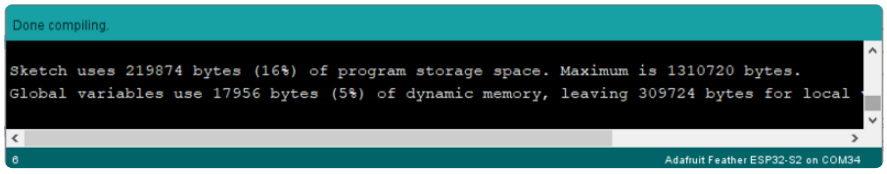


Here's another common error, in my haste I forgot to add a `;` at the end of a line. The compiler warns me that it's looking for one - note that the error is actually a few lines up!



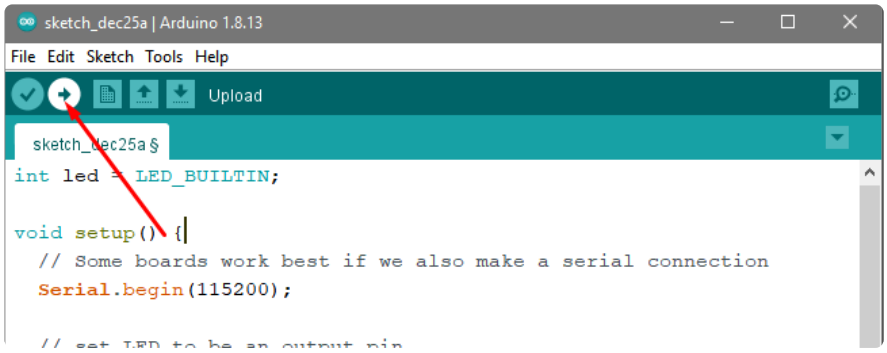
Turning on detailed compilation warnings and output can be very helpful sometimes - Its in Preferences under "Show Verbose Output During:" and check the Compilation button. If you ever need to get help from others, be sure to do this and then provide all the text that is output. It can assist in nailing down what happened!

On success you will see something like this white text output and the message **Done compiling.** in the message area.



Upload Sketch

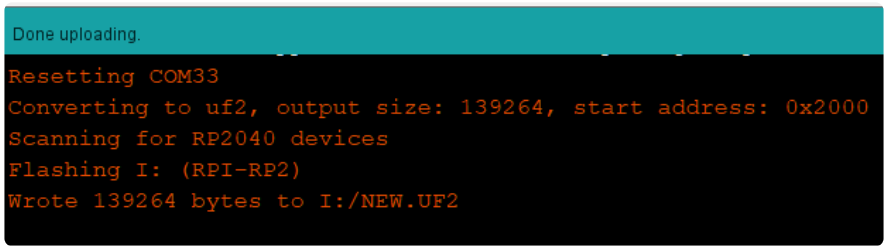
Once the code is verified/compiling cleanly you can upload it to your board. Click the **Upload** button.



The IDE will try to compile the sketch again for good measure, then it will try to connect to the board and upload a the file.

This is actually one of the hardest parts for beginners because it's where a lot of things can go wrong.

However, lets start with what it looks like on success! Here's what your board upload process looks like when it goes right:



Often times you will get a warning like this, which is kind of vague:

`No device found on COM66` (or whatever port is selected)
`An error occurred while uploading the sketch`



This could be a few things.

First up, check again that you have the correct board selected! Many electronics boards have very similar names or look, and often times folks grab a board different from what they thought.

If you're positive the right board is selected, we recommend the next step is to put the board into manual bootloading mode.

Native USB and manual bootloading

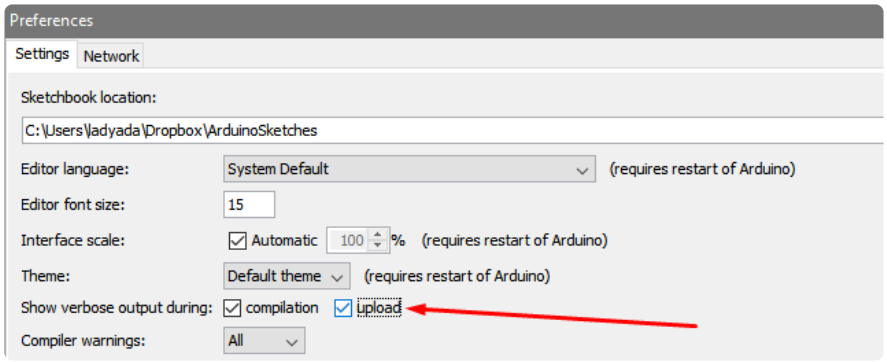
Historically, microcontroller boards contained two chips: the main micro chip (say, ATmega328 or ESP8266 or ESP32) and a separate chip for USB interface that would be used for bootloading (a CH430, FT232, CP210x, etc). With these older designs, the microcontroller is put into a bootloading state for uploading code by the separate chip. It allows for easier uploading but is more expensive as two chips are needed, and also the microcontroller can't act like a keyboard or disk drive.

Modern chips often have 'native' USB - that means that there is no separate chip for USB interface. It's all in one! Great for cost savings, simplicity of design, reduced size

and more control. However, it means the chip must be self-aware enough to be able to put itself into bootload/upload mode on its own. That's fine 99% of the time but is very likely you will at some point get the board into an odd state that makes it too confused to bootload.

A lot of beginners have a little freakout the first time this happens, they think the board is ruined or 'bricked' - it's almost certainly not, it is just crashed and/or confused. You may need to perform a little trick to get the board back into a good state, at which point you won't need to manually bootload again.

Before continuing we really, really suggest turning on **Verbose Upload** messages, it will help in this process because you will be able to see what the IDE is trying to do. It's a checkbox in the **Preferences** menu.

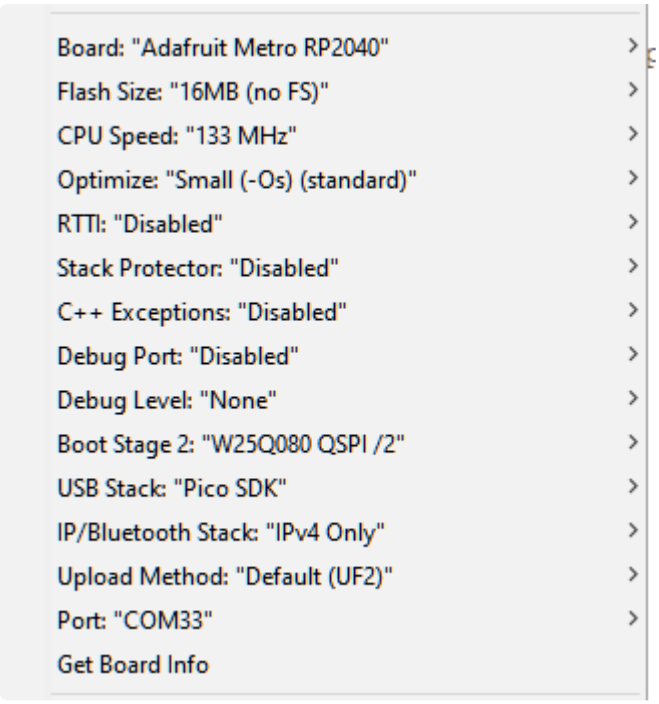


Enter Manual Bootload Mode

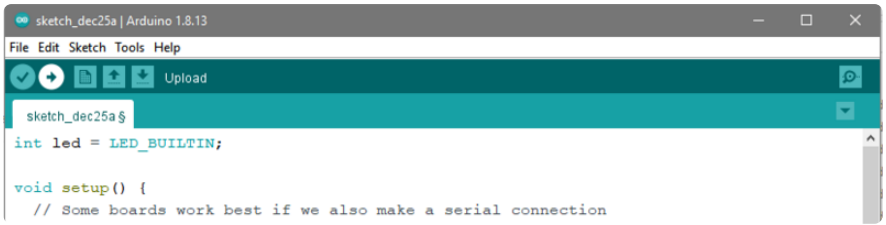
OK now you know it's probably time to try manual bootloading. No problem! Here is how you do that for this board:

To enter the bootloader, hold down the **Boot button**, and while continuing to hold it (don't let go!), press and release the **reset button**. **Continue to hold the Boot button until the RPI-RP2 drive appears!**

Once you are in manual bootload mode, go to the Tools menu, and make sure you have selected the bootloader serial port. **It is almost certain that the serial port has changed now that the bootloader is enabled**



Now you can try uploading again!



Did you remember to select the new Port in the Tools menu since the bootloader port has changed?

This time, you should have success!

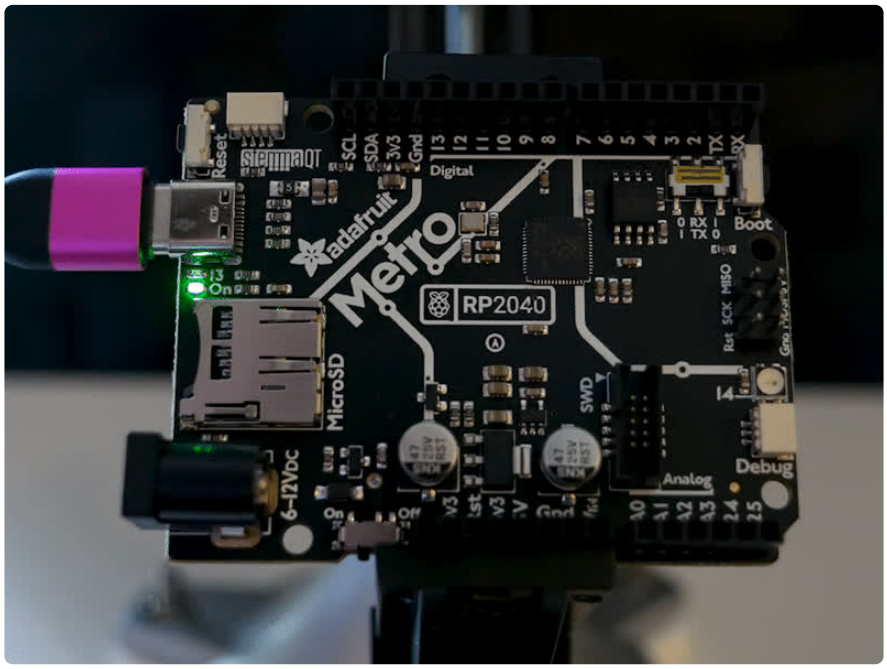
After uploading this way, be sure to **click the reset button** - it sort of makes sure that the board got a good reset and will come back to life nicely.

After uploading with Manual Bootloader - don't forget to re-select the old Port again

It's also a good idea to try to re-upload the sketch again now that you've performed a manual bootload to get the chip into a good state. It should perform an auto-reset the second time, so you don't have to manually bootload again.

Finally, a Blink!

OK it was a journey but now we're here and you can enjoy your blinking LED. Next up, try to change the delay between blinks and re-upload. It's a good way to make sure your upload process is smooth and practiced.



I2C Scan Test

A lot of sensors, displays, and devices can connect over I2C. I2C is a 2-wire 'bus' that allows multiple devices to all connect on one set of pins so it's very convenient for wiring!

When using your board, you'll probably want to connect up I2C devices, and it can be a little tricky the first time. The best way to debug I2C is go through a checklist and then perform an I2C scan

Common I2C Connectivity Issues

- **Have you connected four wires (at a minimum) for each I2C device?** Power the device with whatever is the logic level of your microcontroller board (probably 3.3V), then a ground wire, and a SCL clock wire, and and a SDA data wire.
- **If you're using a STEMMA QT board - check if the power LED is lit.** It's usually a green LED to the left side of the board.
- **Does the STEMMA QT/I2C port have switchable power or pullups?** To reduce power, some boards have the ability to cut power to I2C devices or the pullup

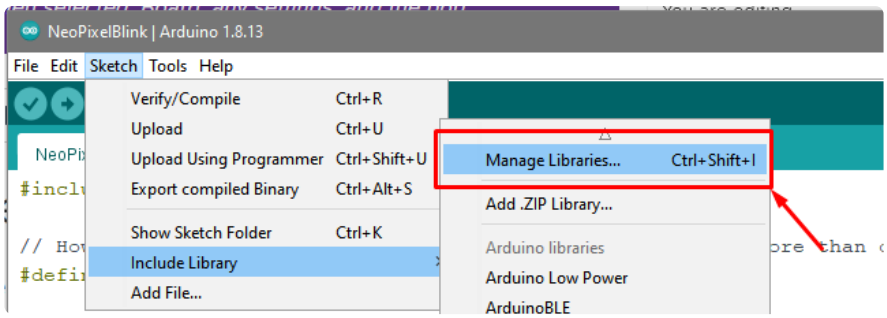
resistors. Check the documentation if you have to do something special to turn on the power or pullups.

- **If you are using a DIY I2C device, do you have pullup resistors?** Many boards do not have pullup resistors built in and they are required! We suggest any common 2.2K to 10K resistors. You'll need two: one each connects from SDA to positive power, and SCL to positive power. Again, positive power (a.k.a VCC, VDD or V+) is often 3.3V
- **Do you have an address collision?** You can only have one board per address. So you cannot, say, connect two AHT20's to one I2C port because they have the same address and will interfere. Check the sensor or documentation for the address. Sometimes there are ways to adjust the address.
- **Does your board have multiple I2C ports?** Historically, boards only came with one. But nowadays you can have two or even three! This can help solve the "hey, but what if I want two devices with the same address" problem: just put one on each bus.
- **Are you hot-plugging devices?** I2C does not support dynamic re-connection, you cannot connect and disconnect sensors as you please. They should all be connected on boot and not change. (Only exception is if you're using a hot-plug assistant but that'll cost you (http://adafru.it/5159)).
- **Are you keeping the total bus length reasonable?** I2C was designed for maybe 6" max length. We like to push that with plug-n-play cables, but really please keep them as short as possible! (Only exception is if you're using an active bus extender (http://adafru.it/4756)).

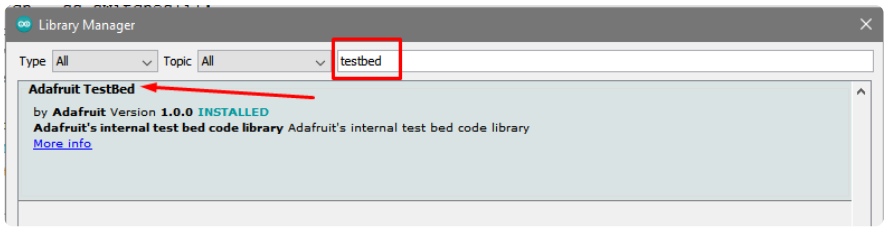
Perform an I2C scan!

Install TestBed Library

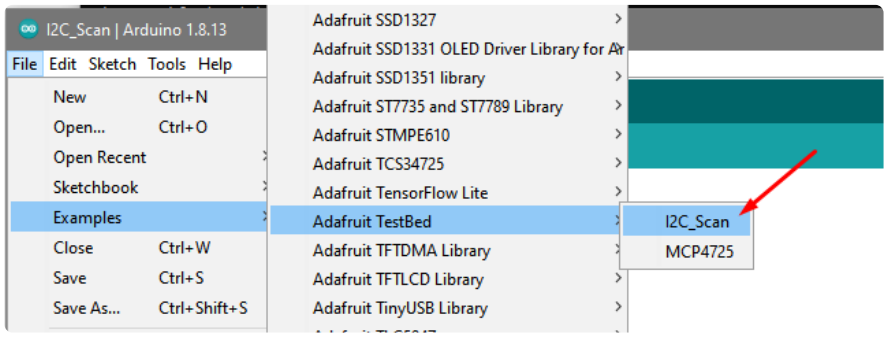
To scan I2C, the Adafruit TestBed library is used. This library and example just makes the scan a little easier to run because it takes care of some of the basics. You will need to add support by installing the library. Good news: it is very easy to do it. Go to the **Arduino Library Manager.**



Search for **TestBed** and install the **Adafruit TestBed** library



Now open up the I2C Scan example




```
#include <Adafruit_TestBed.h>
extern Adafruit_TestBed TB;

#define DEFAULT_I2C_PORT &Wire

// Some boards have TWO I2C ports, how nifty. We should scan both
#if defined(ARDUINO_ARCH_RP2040) \
    || defined(ARDUINO_ADAFRUIT_QTPY_ESP32S2) \
    || defined(ARDUINO_ADAFRUIT_QTPY_ESP32S3_NOPSRAM) \
    || defined(ARDUINO_ADAFRUIT_QTPY_ESP32S3) \
    || defined(ARDUINO_ADAFRUIT_QTPY_ESP32_PICO) \
    || defined(ARDUINO_SAM_DUE) \
    || defined(ARDUINO_ARCH_RENESAS_UNO)
  #define SECONDARY_I2C_PORT &Wire1
#endif

void setup() {
  Serial.begin(115200);

  // Wait for Serial port to open
  while (!Serial) {
    delay(10);
  }
  delay(500);
  Serial.println("Adafruit I2C Scanner");

#if defined(ARDUINO_ADAFRUIT_QTPY_ESP32S2) || \
    defined(ARDUINO_ADAFRUIT_QTPY_ESP32S3_NOPSRAM) || \
    defined(ARDUINO_ADAFRUIT_QTPY_ESP32S3) || \
    defined(ARDUINO_ADAFRUIT_QTPY_ESP32_PICO)
  // ESP32 is kinda odd in that secondary ports must be manually
  // assigned their pins with setPins()!
  Wire1.setPins(SDA1, SCL1);
#endif

#if defined(ARDUINO_ADAFRUIT_FEATHER_ESP32S2)
  // turn on the I2C power by setting pin to opposite of 'rest state'
  pinMode(PIN_I2C_POWER, INPUT);
  delay(1);
  bool polarity = digitalRead(PIN_I2C_POWER);
  pinMode(PIN_I2C_POWER, OUTPUT);
  digitalWrite(PIN_I2C_POWER, !polarity);
#endif
```

```
#if defined(ARDUINO_ADAFRUIT_FEATHER_ESP32S2_TFT)
  pinMode(TFT_I2C_POWER, OUTPUT);
  digitalWrite(TFT_I2C_POWER, HIGH);
#endif

#if defined(ARDUINO_ADAFRUIT_FEATHER_ESP32S2_REVTFT)
  pinMode(TFT_I2C_POWER, OUTPUT);
  digitalWrite(TFT_I2C_POWER, HIGH);
#endif

#if defined(ADAFRUIT_FEATHER_ESP32_V2)
  // Turn on the I2C power by pulling pin HIGH.
  pinMode(NEOPIXEL_I2C_POWER, OUTPUT);
  digitalWrite(NEOPIXEL_I2C_POWER, HIGH);
#endif
}

void loop() {
  Serial.println("");
  Serial.println("");

  Serial.print("Default port (Wire) ");
  TB.theWire = DEFAULT_I2C_PORT;
  TB.printI2CBusScan();

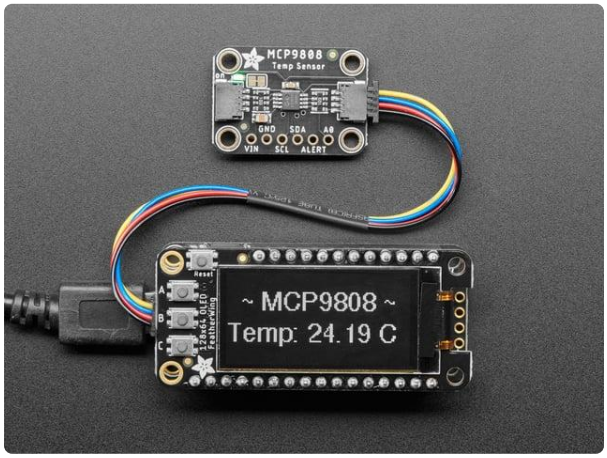
#if defined(SECONDARY_I2C_PORT)
  Serial.print("Secondary port (Wire1) ");
  TB.theWire = SECONDARY_I2C_PORT;
  TB.printI2CBusScan();
#endif

  delay(3000); // wait 3 seconds
}
```

Wire up I2C device

While the examples here will be using the Adafruit MCP9808 (http://adafru.it/5027), a high accuracy temperature sensor, the overall process is the same for just about any I2C sensor or device.

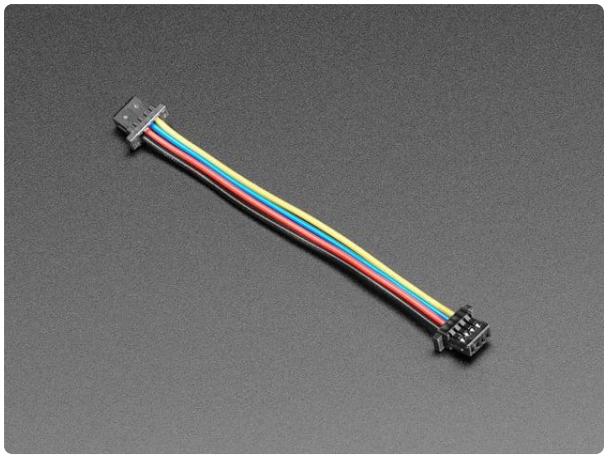
The first thing you'll want to do is get the sensor connected so your board has I2C to talk to.



Adafruit MCP9808 High Accuracy I2C Temperature Sensor Breakout

The MCP9808 digital temperature sensor is one of the more accurate/precise we've ever seen, with a typical accuracy of ±0.25°C over the sensor's -40°C to...

https://www.adafruit.com/product/5027



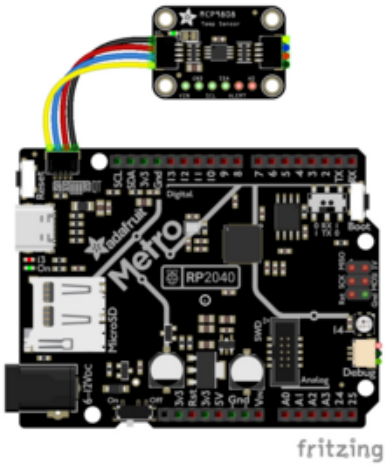
STEMMA QT / Qwiic JST SH 4-Pin Cable - 50mm Long

This 4-wire cable is 50mm / 1.9" long and fitted with JST SH female 4-pin connectors on both ends. Compared with the chunkier JST PH these are 1mm pitch instead of 2mm, but...

https://www.adafruit.com/product/4399

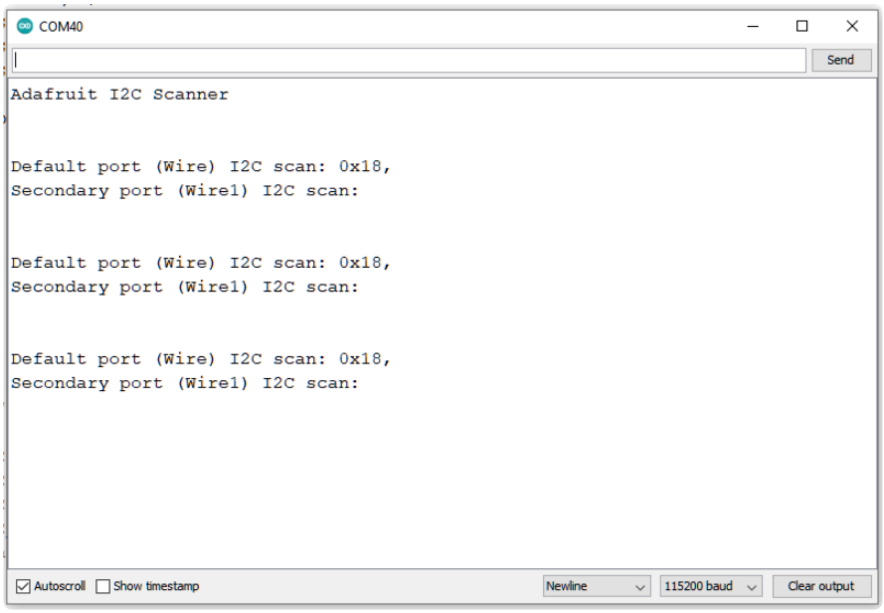
Wiring the MCP9808

The MCP9808 comes with a STEMMA QT connector, which makes wiring it up quite simple and solder-free.



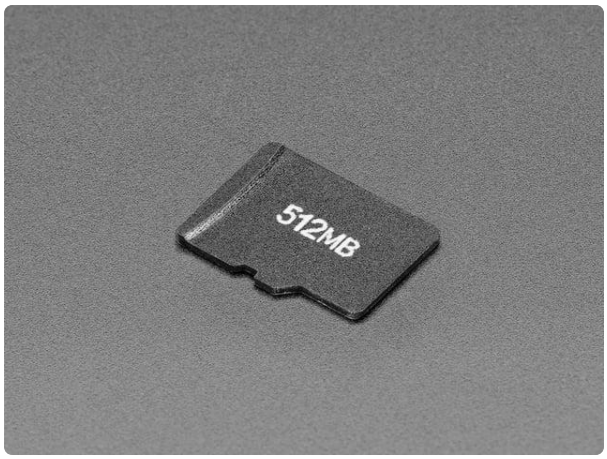
Simply connect the STEMMA QT cable from **the STEMMA QT port on your board** to **the STEMMA QT port on the MCP9808**.

Now upload the scanning sketch to your microcontroller and open the serial port to see the output. You should see something like this:



SD Card Example

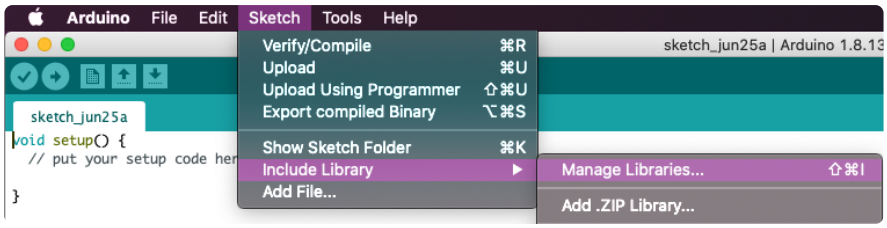
The Metro RP2040 has an onboard SD card slot, making it easy to read and write files from a micro SD card. To do this with Arduino, you'll need to plug in your micro SD card, install the Adafruit fork of the SdFat library (https://adafru.it/18Tf) and run the provided example code.



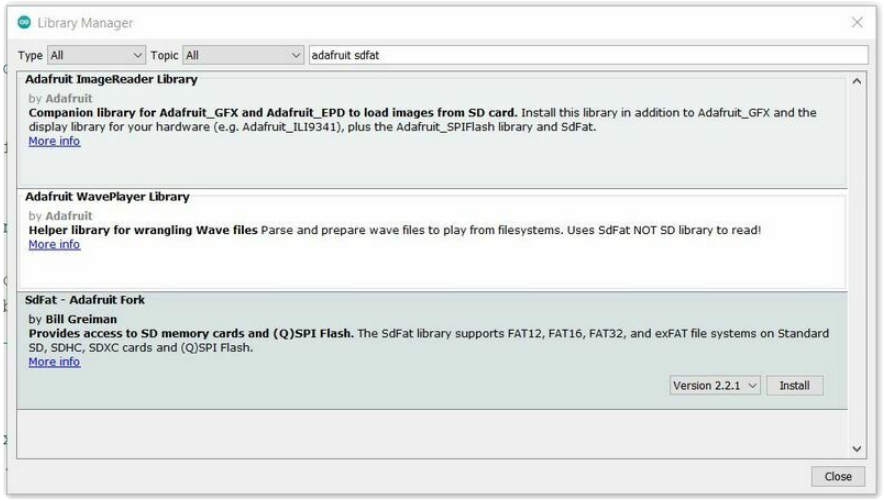
512MB micro SD Memory Card
Add storage in a jiffy using this 512MB microSD card. Preformatted to FAT32, so it works out of the packaging with our projects. Works great with any device in the...
https://www.adafruit.com/product/5252

Library Installation

You can install the **Adafruit Fork of the SDFat** library for Arduino using the Library Manager in the Arduino IDE.



Click the **Manage Libraries ...** menu item, search for **Adafruit SDFat** and select the **SDFat - Adafruit Fork** library:



There are no additional dependencies for the SdFat - Adafruit Fork library.

Read/Write Example

```
// SPDX-FileCopyrightText: 2023 Liz Clark for Adafruit Industries
//
// SPDX-License-Identifier: MIT
/*
  SD card read/write

 This example shows how to read and write data to and from an SD card file
 The circuit:
 * SD card attached to SPI0 bus as follows:
 ** MOSI - pin 19
 ** MISO - pin 20
 ** CLK - pin 18

 created   Nov 2010
 by David A. Mellis
 modified 9 Apr 2012
 by Tom Igoe
 modified 14 Feb 2023
 by Liz Clark

 This example code is in the public domain.

 */

#include "SdFat.h"
SdFat sd;
```

```
#define SD_FAT_TYPE 1

// default CS pin is 23 for Metro RP2040
#define SD_CS_PIN 23

File32 myFile;

void setup() {
  // Open serial communications and wait for port to open:
  Serial.begin(115200);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
  }


  Serial.print("Initializing SD card...");

  if (!sd.begin(SD_CS_PIN)) {
    Serial.println("initialization failed!");
    return;
  }
  Serial.println("initialization done.");

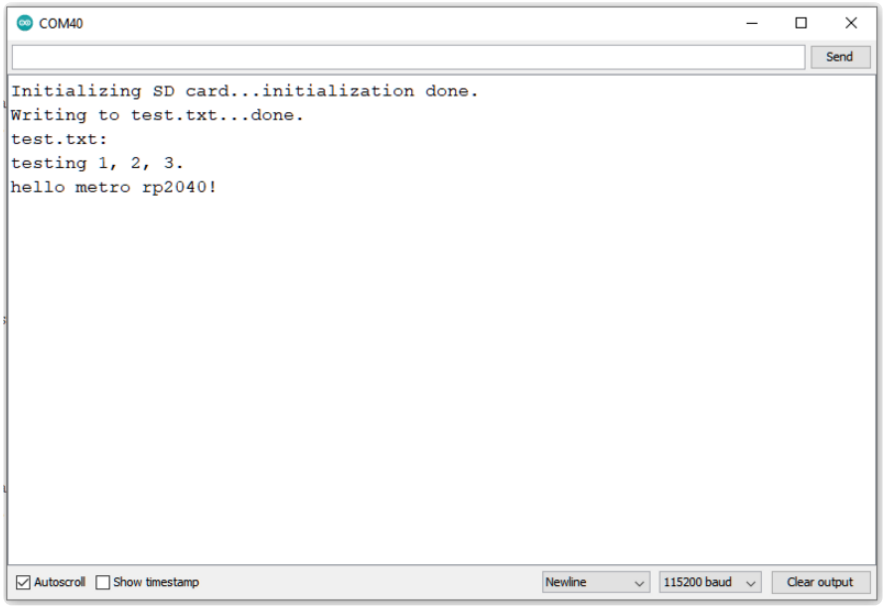
  // open the file. note that only one file can be open at a time,
  // so you have to close this one before opening another.
  myFile.open("test.txt", FILE_WRITE);

  // if the file opened okay, write to it:
  if (myFile) {
    Serial.print("Writing to test.txt...");
    myFile.println("testing 1, 2, 3.");
    myFile.println("hello metro rp2040!");
    // close the file:
    myFile.close();
    Serial.println("done.");
  } else {
    // if the file didn't open, print an error:
    Serial.println("error opening test.txt");
  }

  // re-open the file for reading:
  myFile.open("test.txt");
  if (myFile) {
    Serial.println("test.txt:");

    // read from the file until there's nothing else in it:
    while (myFile.available()) {
      Serial.write(myFile.read());
    }
    // close the file:
    myFile.close();
  } else {
    // if the file didn't open, print an error:
    Serial.println("error opening test.txt");
  }
}

void loop() {
  // nothing happens after setup
}
```



Upload the sketch to your board and open up the Serial Monitor (**Tools -> Serial Monitor**) at 115200 baud. A test text file will be created and written to on the SD card. Then, the text file will be read back with its contents printed to the Serial Monitor.

Factory Reset

The Metro RP2040 microcontroller ships running a NeoPixel swirl example that includes an I2C scan feature that you can view if you open the Arduino IDE Serial Monitor. It's lovely, but you probably had other plans for the board. As you start working with your board, you may want to return to the original code to begin again, or you may find your board gets into a bad state. Either way, this page has you covered.

Completing a factory reset will erase your board's firmware which is also used for storing CircuitPython/Arduino/Files! Be sure to back up your data first.

Step 1. Download the factory-reset.uf2 file

Save the following file wherever is convenient for you. You will need to access it to copy it to your board.

Click to download metro-rp2040-factory-reset.uf2

https://adafru.it/18TA

Step 2. Enter RP2040 bootloader mode

Entering the RP2040 bootloader is easy. Complete the following steps.

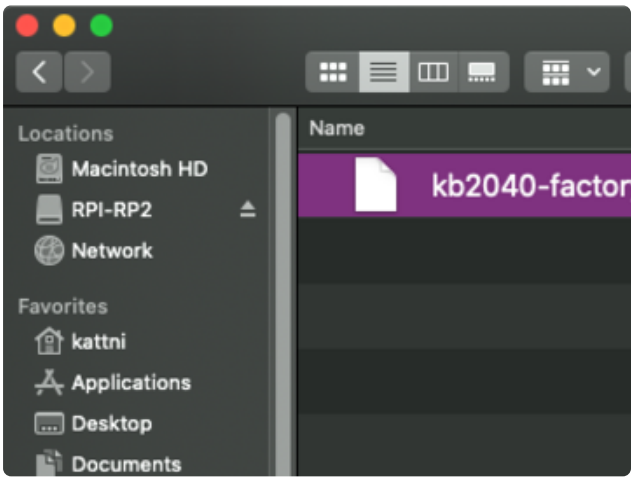
Before you start, make sure your microcontroller is plugged into USB port to your computer using a data/sync cable. Charge-only cables will not work!

Turn on the On/Off switch - check that you see the OK light on so you know the board is powered, a prerequisite!

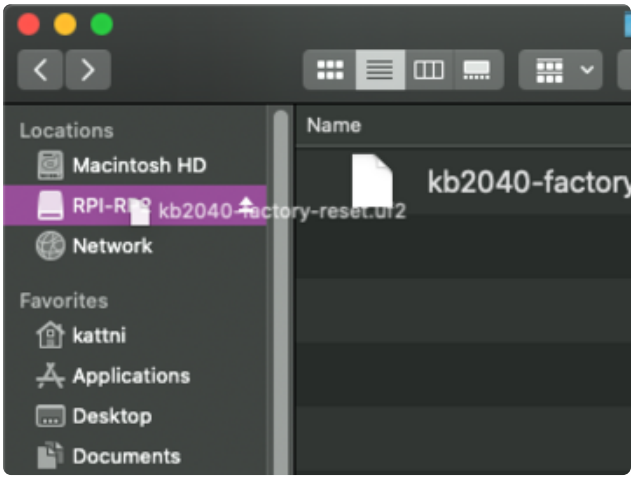
To enter the bootloader:

1. **Press and hold the Boot button down.** Don't let go of it yet!
2. **Press and release the Reset button.** You should still have the Boot button pressed while you do this.
3. **Continue holding the Boot button until you see the RPI-RP2 drive appear.**
4. You can now release but Boot button.

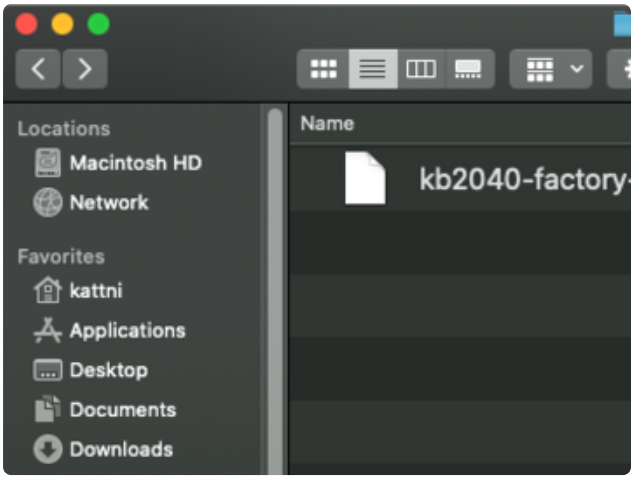
Step 3. Drag UF2 file to RPI-RP2



Navigate to the folder where you downloaded the **factory-reset.uf2** file from Step 1.



Drag the **factory-reset.uf2** file to the **RPI-RP2** drive.



The **RPI-RP2** drive will disappear.

The board will automatically reboot.

The NeoPixel LED on the Metro RP2040 will light up in a rainbow swirl, and you can view the I2C scan if you open the Arduino IDE Serial Monitor.

You've successfully returned your board to a factory reset state!

Flash Resetting UF2

If your board ever gets into a really weird state and doesn't even show up when loading code, try loading this 'nuke' UF2 which will do a 'deep clean' on your Flash Memory. **You will lose all the files on the board**, but at least you'll be able to revive it! Download the file below, and follow the instructions in Step 2 and Step 3 above to load this UF2. Then, start again at Step 1 to return your board to factory reset state.

Download flash erasing "nuke" UF2

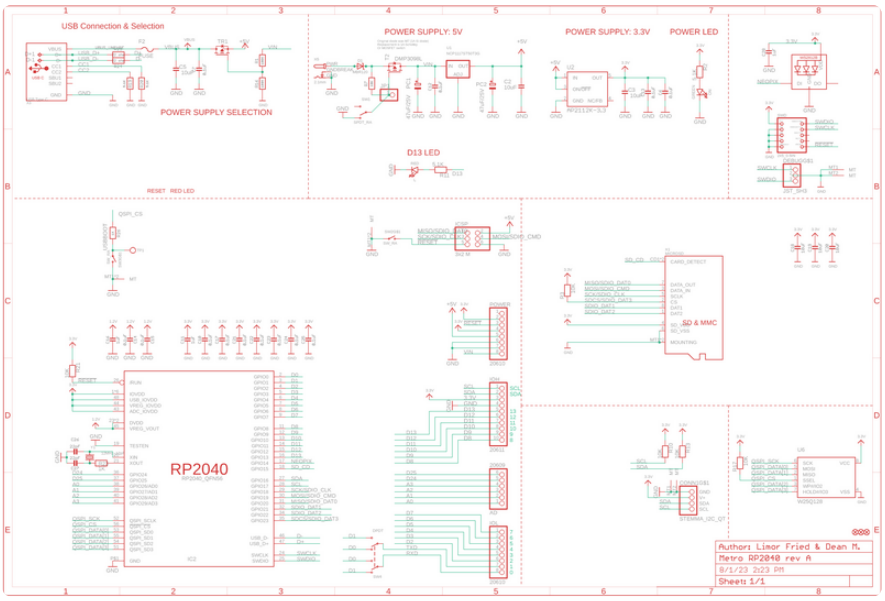
https://adafru.it/RLE

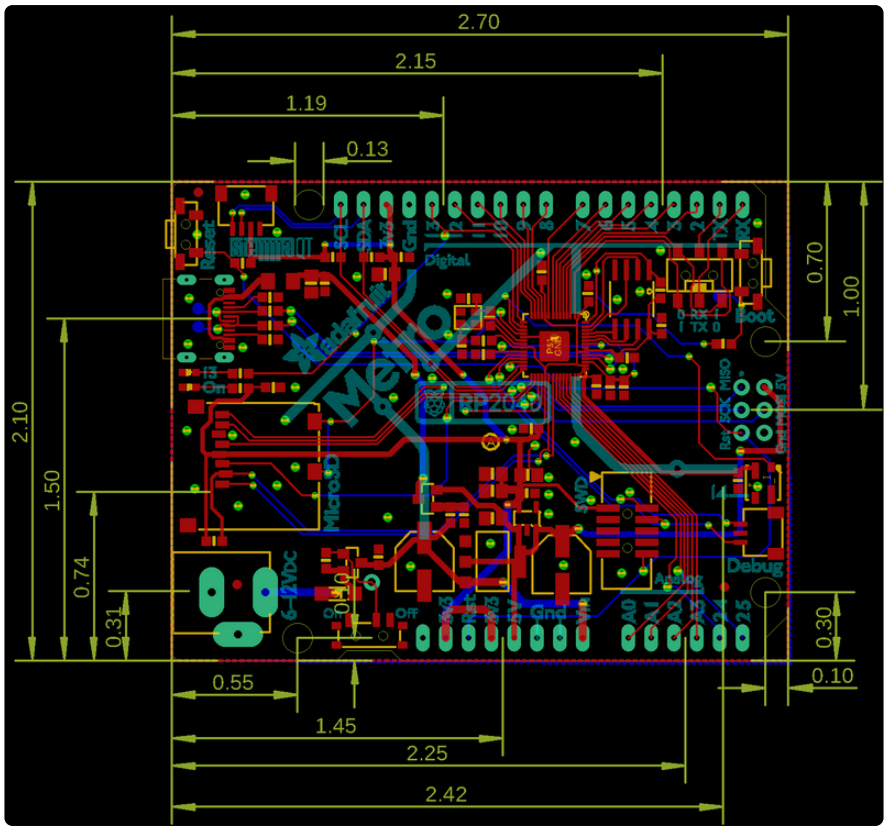
Downloads

Files

- RP2040 Datasheet (https://adafru.it/QTf)
- EagleCAD PCB files on GitHub (https://adafru.it/18TB)
- Fritzing object in the Adafruit Fritzing Library (https://adafru.it/18TC)
- PrettyPins PDF on GitHub (https://adafru.it/18RF)
- PrettyPins SVG (https://adafru.it/18TF)

Schematic and Fab Print




2.70
2.15
1.19
0.13
2.10
1.50
0.74
0.31
0.70
1.00
0.30
0.10
0.55
1.45
2.25
2.42

Mouser Electronics

Authorized Distributor

Click to View Pricing, Inventory, Delivery & Lifecycle Information:

Adafruit:

5786