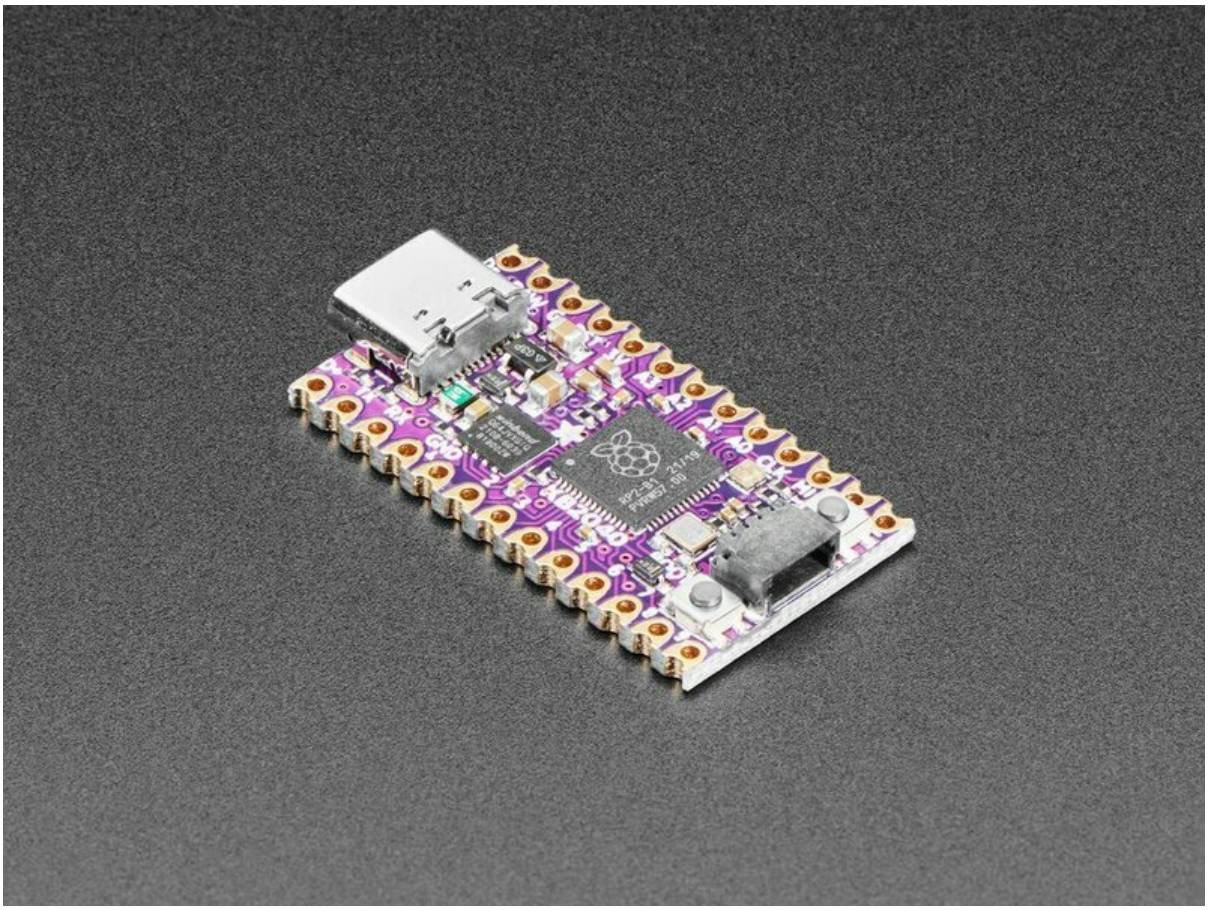
Adafruit KB2040

Created by Kattni Rembor



https://learn.adafruit.com/adafruit-kb2040

Last updated on 2026-11-19 07:13:59 PM UTC

Table of Contents

Overview 7

Pinouts 11

- Power
- RAW Jumper
- Logic Pins
- Bottom Edge
- Top Edge
- STEMMA QT
- GPIO Pins by Pin Functionality
- Microcontroller and Flash
- Buttons
- NeoPixel LED
- D+/D- USB Pins

CircuitPython 21

- CircuitPython Quickstart
- Safe Mode
- Flash Resetting UF2

Installing the Mu Editor 24

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

The CIRCUITPY Drive 27

- Boards Without CIRCUITPY

Creating and Editing Code 28

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

Exploring Your First CircuitPython Program 34

- 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 37

- Are you using Mu?
- Serial Console Issues or Delays on Linux
- Setting Permissions on Linux
- Using Something Else?

Interacting with the Serial Console 40

The REPL 43

- Entering the REPL
- Interacting with the REPL

• Returning to the Serial Console

CircuitPython Pins and Modules 48
• CircuitPython Pins
• import board
• I2C, SPI, and UART
• What Are All the Available Names?
• Microcontroller Pin Names
• CircuitPython Built-In Modules

CircuitPython Libraries 55
• 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 66
• CircuitPython Core Documentation
• CircuitPython Library Documentation

Welcome to the Community! 74
• Adafruit Discord
• CircuitPython.org
• Adafruit GitHub
• Adafruit Forums
• Read the Docs

Recommended Editors 83
• Recommended editors
• Recommended only with particular settings or add-ons
• Editors that are NOT recommended
• Moving editor swap files off CIRCUITPY

Advanced Serial Console on Windows 86
• What's the COM?
• Windows Serial Port Terminal Programs
• Install Putty
• Windows 7 and 8.1

Advanced Serial Console on Mac 89
• What's the Port?
• macOS Serial Port Terminal Programs
• Connect with screen

Advanced Serial Console on Linux 92
• What's the Port?
• Linux Serial Port Terminal Programs

• Connect with tio
• Permissions on Linux

Frequently Asked Questions 96

• Using Older Versions
• Python Arithmetic
• Wireless Connectivity
• Asyncio and Interrupts
• Status RGB LED
• Memory Issues
• Unsupported Hardware

Troubleshooting 106

• 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 before 14.4: Errors Writing to CIRCUITPYmacOS 14.4 - 15.1: Slow Writes to CIRCUITPY
• Bootloader (boardnameBOOT) Drive Not Present
• Windows Explorer Locks Up When Accessing boardnameBOOT Drive
• boardnameBOOT on SAMD boards Does Not Appear or is Inaccessible on Linux or Chromebooks
• Copying UF2 to boardnameBOOT Drive Hangs at 0% Copied
• CIRCUITPY Drive Does Not Appear or Disappears Quickly
• "M105" Seen on Display, Crashes, Missing CIRCUITPY
• 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

CircuitPython Essentials 126

Blink 128

• NeoPixel Location
• Blinking a NeoPixel LED
• RGB LED Colors

Digital Input 131

• NeoPixel and Button
• Controlling the NeoPixel with a Button

Analog In 134
• Analog to Digital Converter (ADC)
• Potentiometers
• Hardware
• Wire Up the Potentiometer
• Reading Analog Pin Values
• Reading Analog Voltage Values
CPU Temperature 141
• Microcontroller Location
• Reading the Microcontroller Temperature
NeoPixel LED 144
• NeoPixel Location
• NeoPixel Color and Brightness
• RGB LED Colors
• NeoPixel Rainbow
Storage 149
• The boot.py File
• The code.py File
• Logging the Temperature
• Recovering a Read-Only Filesystem
Capacitive Touch 155
• 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 162
• I2C and CircuitPython
• Necessary Hardware
• Wiring the MCP9808
• Find Your Sensor
• I2C Sensor Data
• Where's my I2C?
Arduino IDE Setup 170
• Arduino IDE Download
• Adding the Philhower Board Manager URL
• Add Board Support Package
• Choose Your Board
Arduino Usage 175
• RP2040 Arduino Pins
• Choose Your Board
• Load the Blink Sketch
• Manually Enter the Bootloader
Arduino NeoPixel Blink 177
• Pre-Flight Check: Get Arduino IDE & Hardware Set Up
• Start up Arduino IDE and Select Board/Port

- Install NeoPixel Library
- New NeoPixel Blink Sketch
- Verify (Compile) Sketch
- Upload Sketch
- Native USB and manual bootloading
- Enter Manual Bootload Mode
- Finally, a Blink!

I2C Scan Test 188

- Common I2C Connectivity Issues
- Perform an I2C scan!
- Wiring the MCP9808

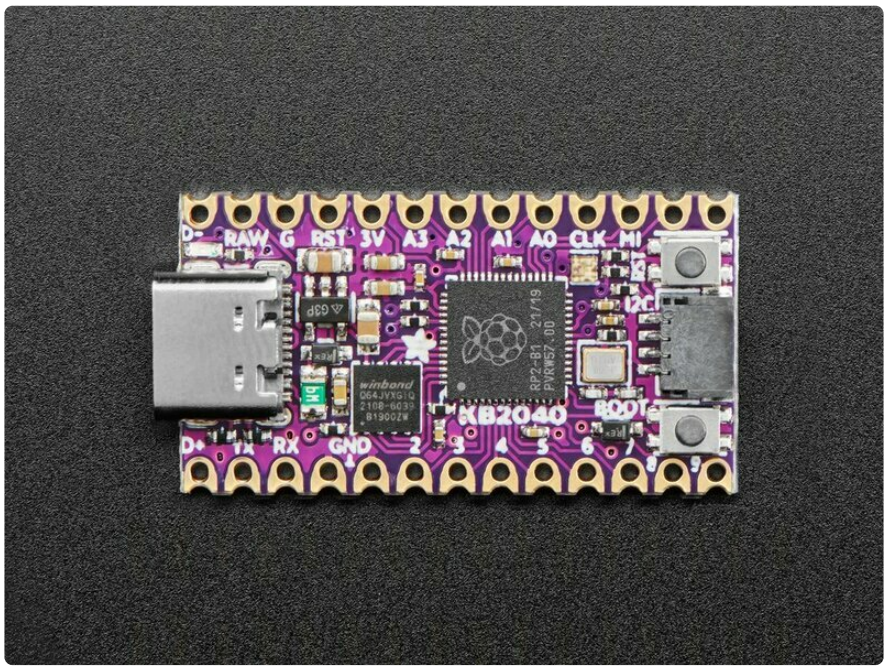
Factory Reset 193

- 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 196

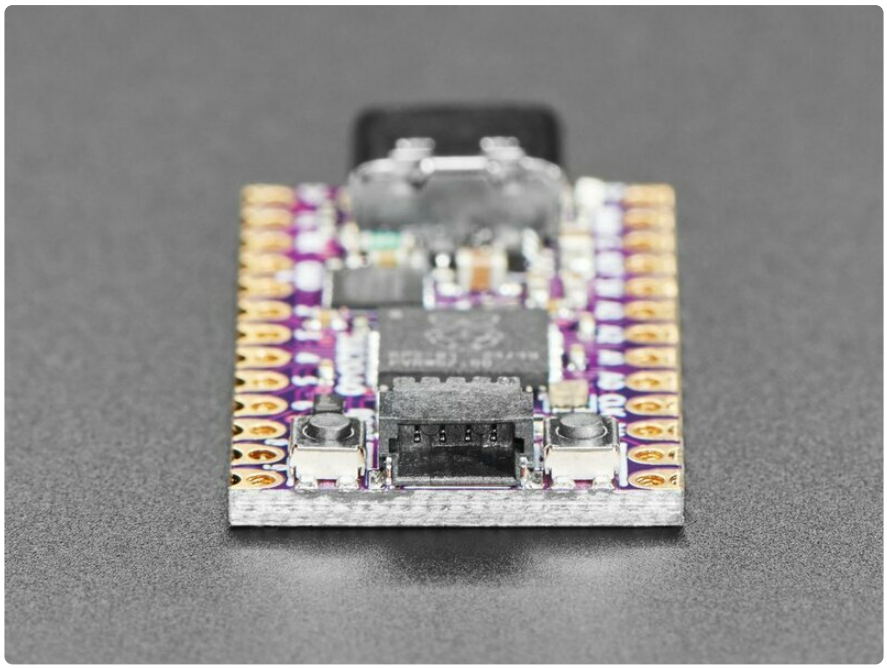
- Files:
- Schematic and Fab Print

Overview

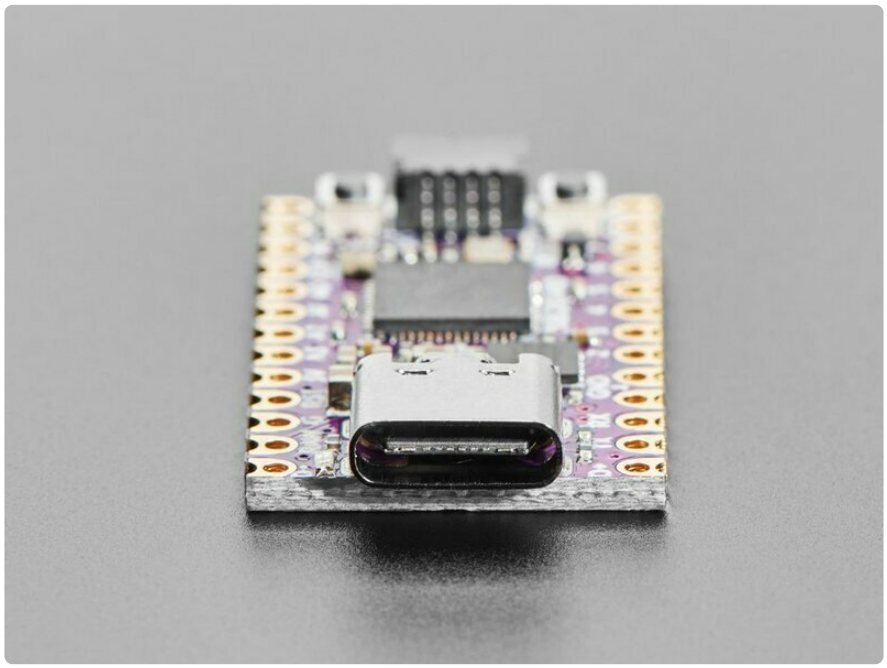


A wild Kee Boar appears! It's a shiny **KB2040**! An Arduino Pro Micro-shaped board for Keebs with RP2040. (#keeblife 4 evah) A lot of folks like using Adafruit parts for their Keeb builds – but with the ItsyBitsy not being pin-compatible with the Pro Micro pinout, it really wasn't very easy without some sort of adapter plate.

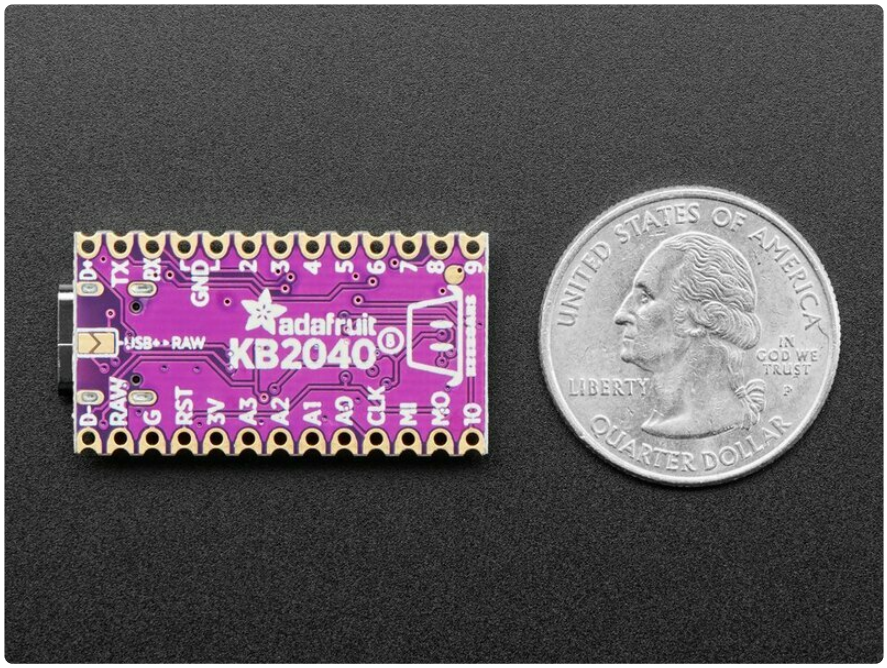
Now we're seeing lots of people use CircuitPython for keebs, which is awesome! So why not try our hands at spinning up a pro-micro-compatible RP2040 board? The RP2040 is plenty powerful, low-cost, and makes for an excellent keeb driver chip.



We mixed together what we liked most about the SparkFun Pro Micro RP2040 (**Qwiic / STEMMA QT I2C port on the end**, so good!) and Elite-C (**castellated pads & pins for D+ and D-**) and our existing RP2040 boards (**boot button can be used for user, 8MB QSPI flash, onboard NeoPixel, jumper for skipping the diode/fuse for high power RGB LEDs or USB hosting**). We even got it to all fit on a 2-layer PCB with 7/7 routing – just needed to make the smallest caps and resistors 0402.



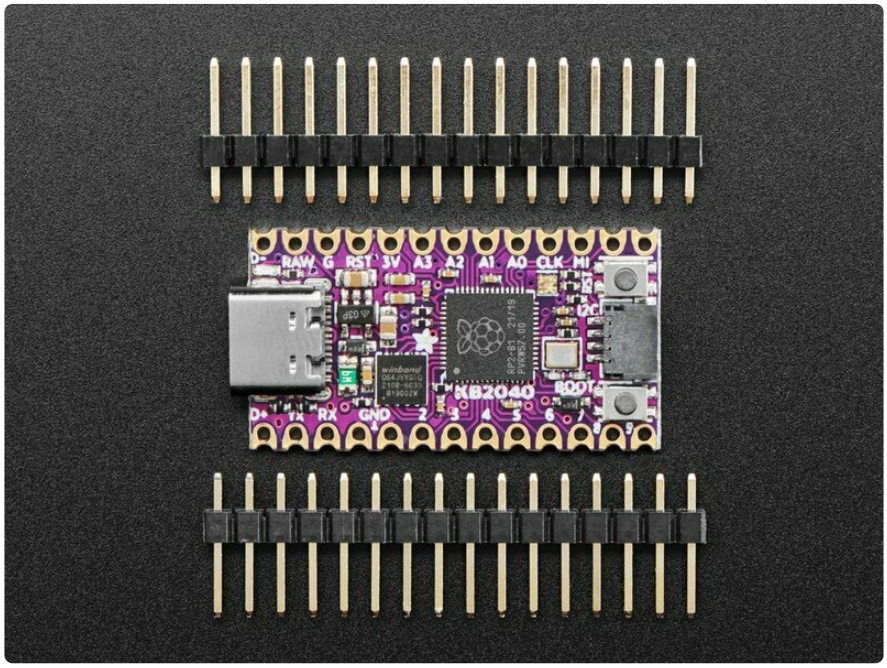
With 20 GPIO available (18 on castellated pins, 2 on STEMMA QT port) you can easily make up to 100-keys matrices, or common 65% 5x15 layouts. Use a plug-and-play QT cable to connect to the last two pins (http://adafru.it/4209) without having to do any desoldering/rework.



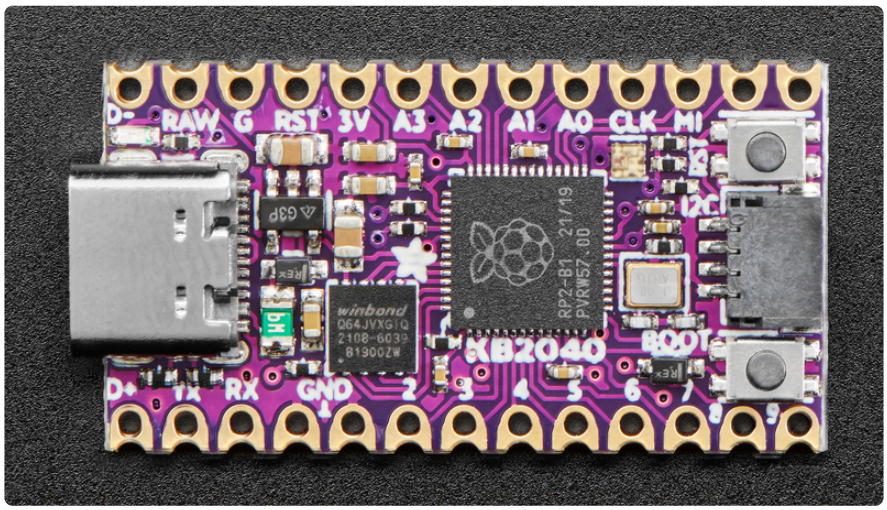
Board features:

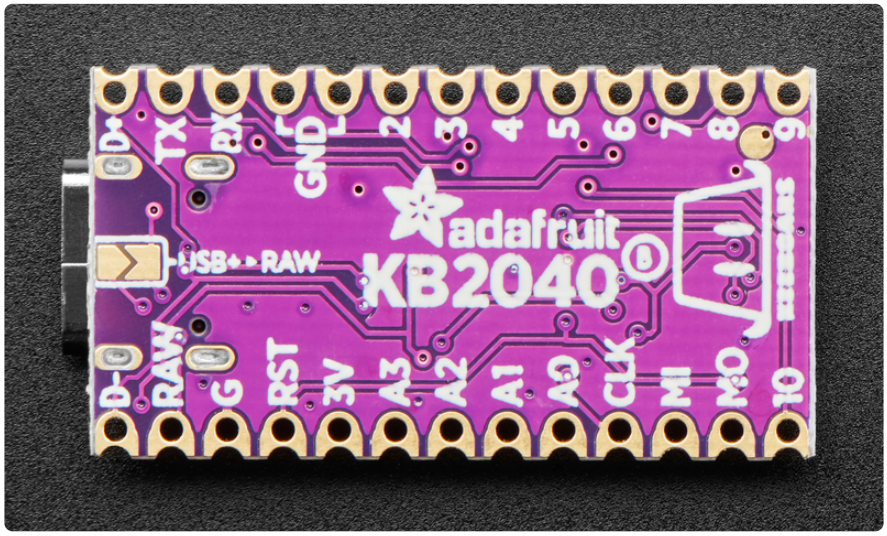
- Same size and form-factor as a Pro Micro breakout () and nearly-identical pinout (this board has fewer analog pins, for example)
- Measures 1.3" x 0.7" without headers soldered in
- RP2040 32-bit Cortex M0+ dual core running at ~125 MHz @ 3.3V logic and power. 264 KB RAM, No EEPROM. 12 MHz crystal for perfect timing.
- **8 MB SPI FLASH** chip for storing files and CircuitPython/MicroPython code storage.
- **20 x GPIO pins with following capabilities:**
  - 18 GPIO on castellated/pin breakout pads. 2 GPIO on QT port that can be easily accessed for 5x15 keyboard layouts.
  - Four 12 bit ADCs
  - Two I2C, Two SPI and two UART peripherals, we label one of for the 'main' interface in standard Pro Micro locations
  - 16 x PWM outputs - for servos, LEDs, etc
  - The 10 digital non-ADC GPIO are consecutive for maximum PIO compatibility
- **RGB NeoPixel** for colorful status indiction

- Classic **green power LED**
- Both Reset button and Bootloader select button for quick restarts. Bootloader button is also available as a generic GPIO input button.
- STEMMA QT connector () on the end is compatible with the SparkFun Qwiic () I2C connector, and can be used to plug and play I2C devices, or just as 2 extra GPIO pins.
- 3.3V regulator with 500mA peak current output
- **RAW** output, for powering NeoPixels or other 5V devices. Jumper on bottom lets you skip over the 500mA fuse, for up to 2A from USB ports.
- **USB Type C connector** lets you access built-in ROM USB bootloader and serial port debugging
- **Extra D- and D+ breakouts** for alternative USB connection options.

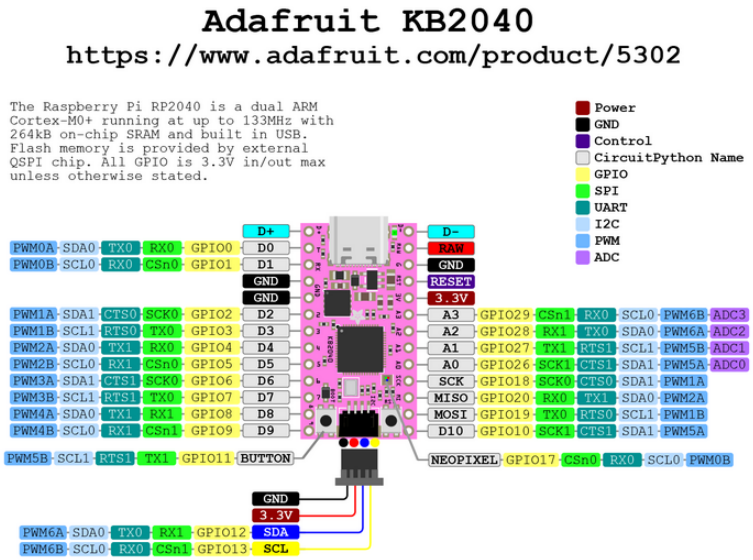


Pinouts



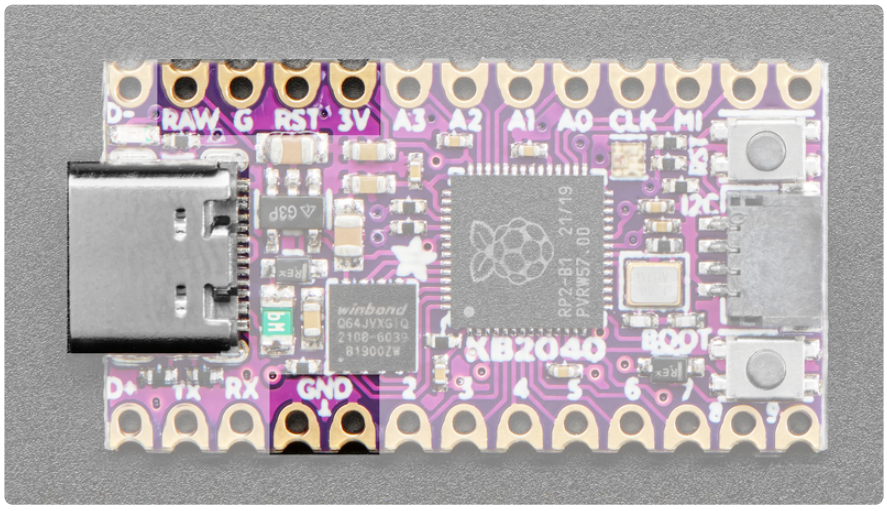


The KB2040 has a many pins, and most pins have multiple functions. This page is a detailed tour of the board.



PrettyPins PDF () on GitHub.

Power



- The KB2040 has a USB C connector on the left end of the board. This connector is used to power and program the board. Use a USB Type C cable to plug the KB2040 into your computer.

Along with the USB C connector, the KB2040 has 6 power related pins.

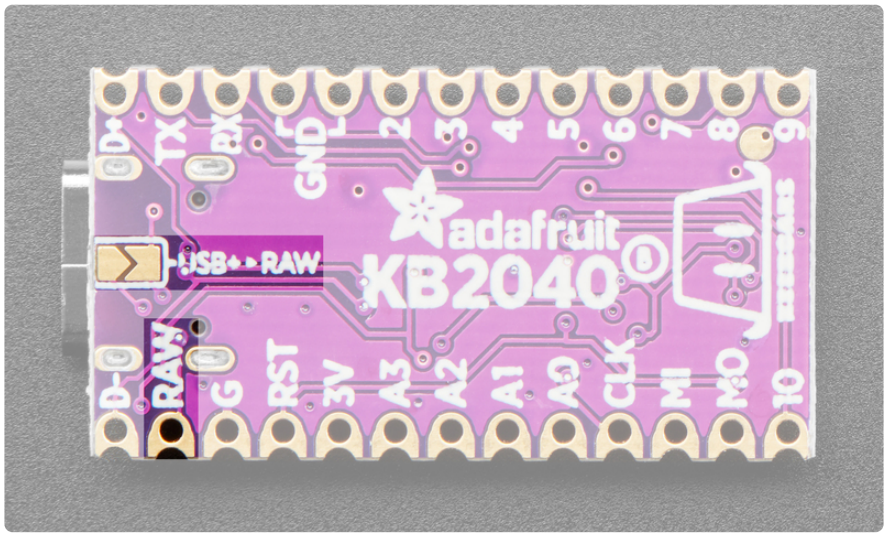
Along the top edge:

- **RAW** - This pin is for powering NeoPixels or other 5V devices. By default this pin is after the 500mA fuse and polarity-protection diode. That means you can get max 500ma to 1A from this pin before the fuse will pop. If you want to connect directly to the USB power pin (either for USB host support or because you want 2A) You can solder the jumper on the bottom closed: It lets you bypass the 500mA fuse, for up to 2A from USB ports, and is also for USB hosting.
- **G** - This is common ground for power and logic.
- **RST** - This is the reset pin. Tie to ground to manually reset the board.
- **3V** - This pin is the regulated output from the onboard regulator. You can draw 500mA whether powered by USB or battery.

Along the bottom edge:

- **GND** - Same as the G pin on the top edge - common ground for power and logic.

RAW Jumper



On the back of the board, aligned with the USB C connector, is the **USB+ > RAW jumper**. Solder this closed to bypass the 500mA fuse, for up to 2A from USB ports, as well as USB hosting. Once soldered closed, you can use the RAW pin for powering NeoPixels or other 5V devices.

Logic Pins

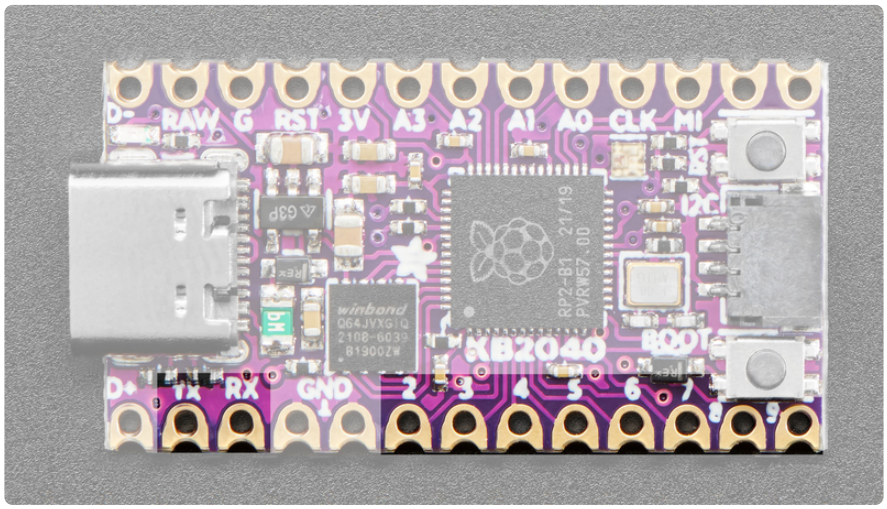
I2C and SPI on RP2040

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 KB2040 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 KB2040. 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

Bottom Edge



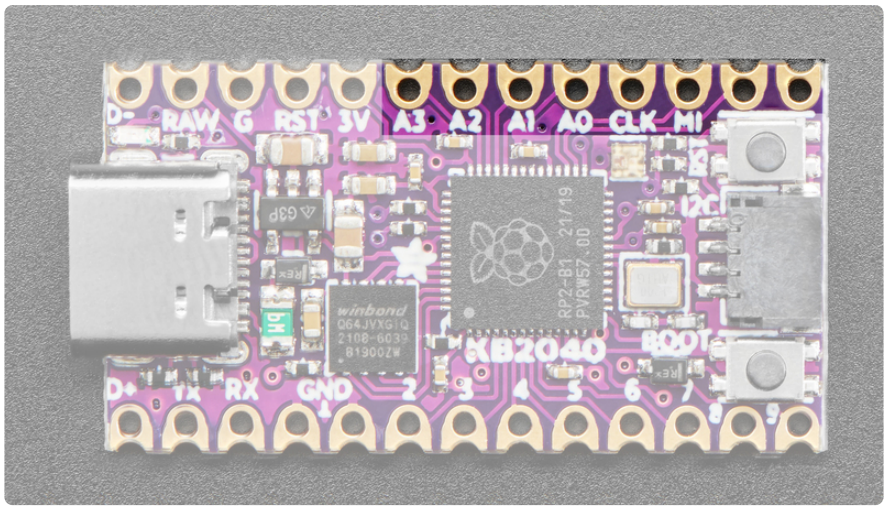
Digital Pins

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

- **TX/D0** - The main UART0 TX pin. It is also I2C0 SDA, SPI0 MISO and PWM0 A.
- **RX/D1** - The main UART0 RX pin. It is also I2C0 SCL, SPI0 CS and PWM0 B.
- **D2** - Digital I/O pin 2. It is also I2C1 SDA, SPI0 SCK and PWM1 A.

- **D3** - Digital I/O pin 3. It is also I2C1 SCL, SPI0 MOSI and PWM1 B.
- **D4** - Digital I/O pin 4. It is also SPI0 MISO, UART1 TX, I2C0 SDA, and PWM2 A.
- **D5** - Digital I/O pin 5. It is also SPI0 CS, UART1 RX, I2C0 SCL, and PWM2 B.
- **D6** - Digital I/O pin 6. It is also SPI0 SCK, I2C1 SDA, and PWM3 A.
- **D7** - Digital I/O pin 7. It is also SPI0 MOSI, I2C1 SCL, and PWM3 B.
- **D8** - Digital I/O pin 8. It is also SPI1 MISO, UART1 TX, I2C0 SDA, and PWM4 A.
- **D9** - Digital I/O pin 9. It is also SPI1 CS, UART1 RX, I2C0 SCL, and PWM4 B.

Top Edge



Analog Pins

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

- **A3** - This pin is ADC3. It is also SPI1 CS, I2C0 SCL and PWM6 B.
- **A2** - This pin is ADC2. It is also SPI1 MISO, I2C0 SDA and PWM6 A.
- **A1** - This pin is ADC1. It is also SPI1 MOSI, I2C1 SCL and PWM5 B.
- **A0** - This pin is ADC0. It is also SPI1 SCK, I2C1 SDA and PWM5 A.

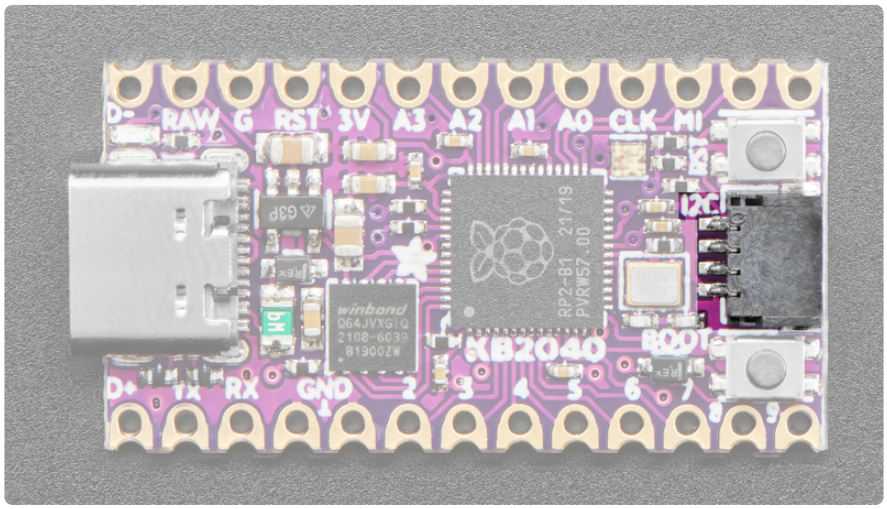
Digital Pins

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

- **CLK** - The main SPI0 SCK pin. It is also I2C1 SDA and PWM1 A.
- **MISO** - The main SPI0 MISO pin. It is also UART1 TX, I2C0 SDA and PWM2 A.
- **MOSI** - The main SPI0 MOSI pin. It is also I2C1 SCL and PWM1 B.

- **D10** - Digital I/O pin 10. It is also SPI1 SCK, I2C1 SDA and PWM5 A.

STEMMA QT



The KB2040 comes with a built in **STEMMA QT connector**! This means you can connect up all sorts of I2C sensors and breakouts (), no soldering required! This connector uses the SCL and SDA pins for I2C. In CircuitPython, you can use the STEMMA connector with `board.SCL` and `board.SDA`, or `board.STEMMA_I2C()`.

- **SCL/D13** - This is the main I2C0 SCL pin. It is also SPI1 CS, UART0 RX and PWM6 B.
- **SDA/D12** - This is the main I2C0 SDA pin. It is also SPI1 MISO, UART0 TX and PWM6 A.

CircuitPython I2C, SPI and UART

Note that in CircuitPython, there is a board object each for I2C, SPI and UART that use the pins and connector labeled on the KB2040. You can use these objects to initialise these peripherals in your code.

- `board.STEMMA_I2C()` uses I2C STEMMA QT connector (SCL/SDA)
- `board.SPI()` uses CLK/MOSI/MISO
- `board.UART()` uses RX/TX

GPIO Pins by Pin Functionality

Primary pins based on the KB2040 silk are bold.

I2C Pins

- I2C0 SCL: **SCL**, RX, D5, D9, A3, MOSI
- I2C0 SDA: **SDA**, TX, D4, D8, A2, D10
- I2C1 SCL: D3, D7, A1, MOSI
- I2C1 SDA: D2, D6, A0, CLK

SPI Pins

- SPI0 SCK: **SCK**, D2, D6
- SPI0 MOSI: **MO**, D3, D7
- SPI0 MISO: **MI**, TX, D4
- SPI0 CS: RX, D5
- SPI1 SCK: A0, D10
- SPI1 MOSI: A1,
- SPI1 MISO: D8, A2
- SPI1 CS: D9, A3

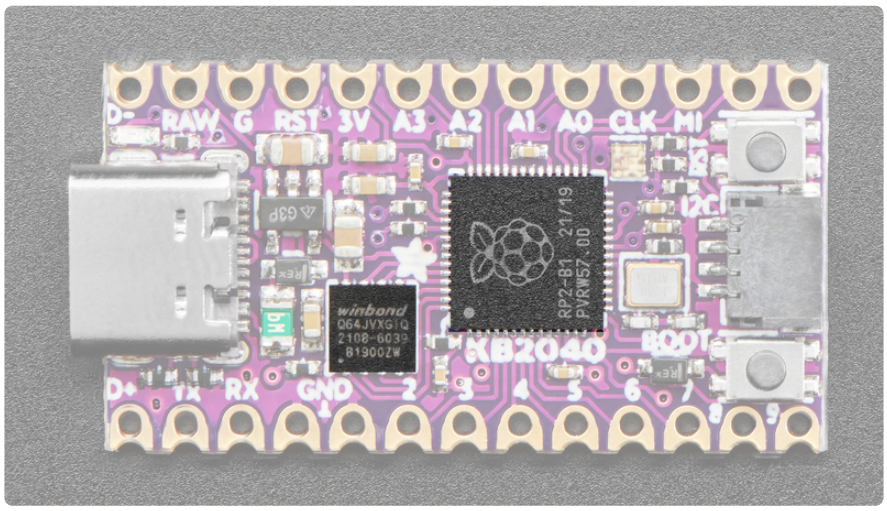
UART Pins

- UART0 TX: **TX**,
- UART0 RX: **RX**
- UART1 TX: D4, D8, MISO
- UART1 RX: D5, D9

PWM Pins

- PWM0 A: TX,
- PWM0 B: RX
- PWM1 A: D2, CLK
- PWM1 B: D3, MOSI
- PWM2 A: D4, MISO
- PWM2 B: D5
- PWM3 A:D6
- PWM3 B: D7
- PWM4 A: D8
- PWM4 B: D9
- PWM5 A: A0, D10
- PWM5 B: A1
- PWM6 A: A2
- PWM6 B: A3

Microcontroller and Flash

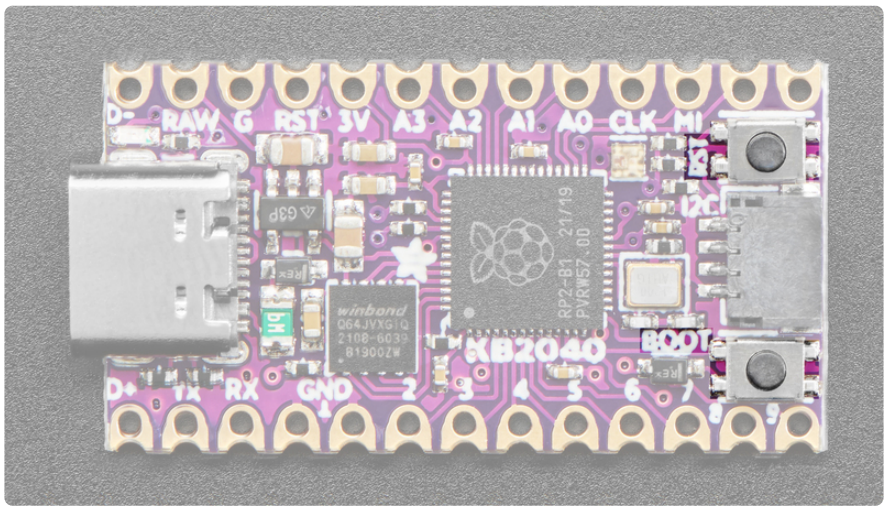


The square towards the middle is the **RP2040 microcontroller**, the "brains" of the KB2040 board.

The square between the RP2040 and the USB connector 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 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.

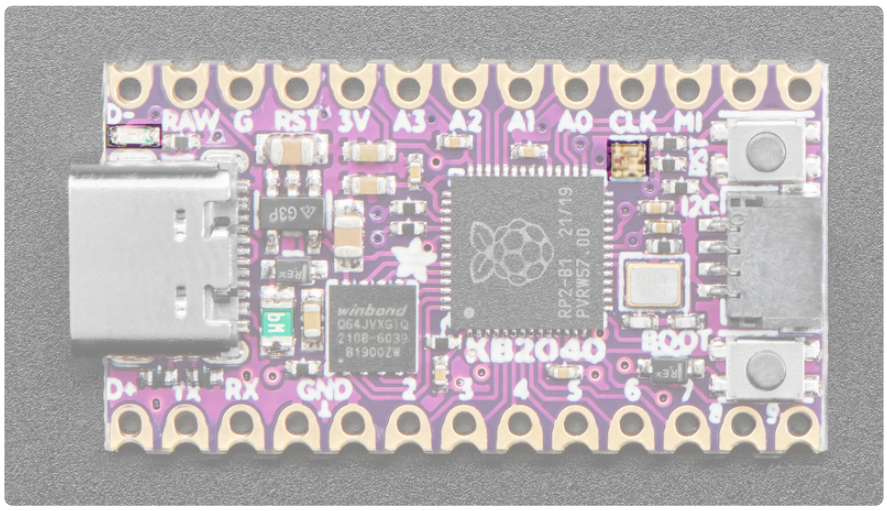
Buttons



The KB2040 has two buttons.

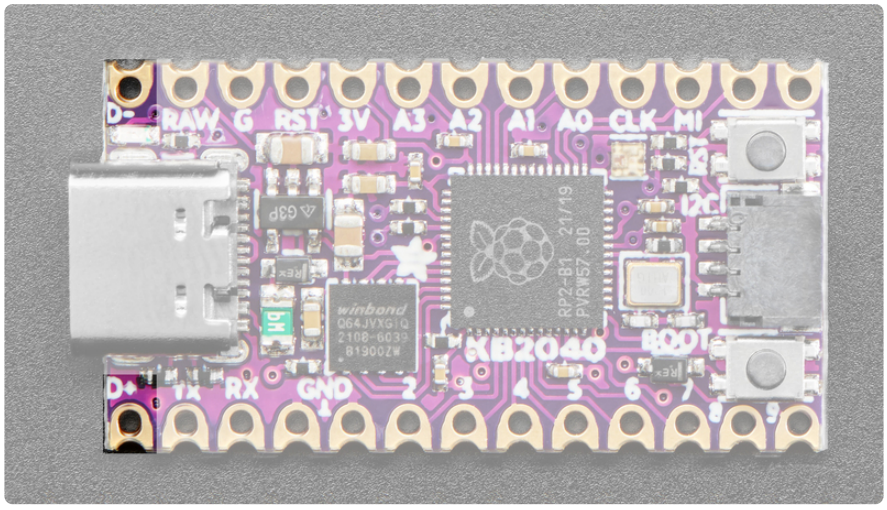
- The **RST button** (reset) restarts the board and helps enter the bootloader. You can click it to reset the board without unplugging the USB cable or battery.
- The **BOOT (boot select)** button is used to enter the bootloader. To enter the bootloader, press and hold BOOT and then power up the board (either by plugging it into USB or pressing RST). The bootloader is used to install/update CircuitPython. This button is also usable as an input in CircuitPython code, on pin `board.BUTTON`.

NeoPixel LED



- The RGB **NeoPixel LED** is located near CLK on the silk. It is controllable in CircuitPython code using the pin `board.NEOPIXEL`.
- Above the USB C connector is the **green power LED**.

D+/D- USB Pins



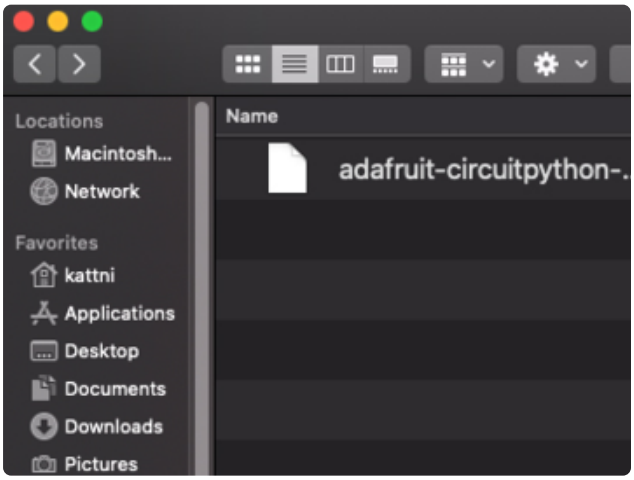
On either side of the USB C connector, on the USB-end of the board, are the **D+ and D- pins** for alternative USB connections.

CircuitPython

CircuitPython () is a derivative of MicroPython () 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.

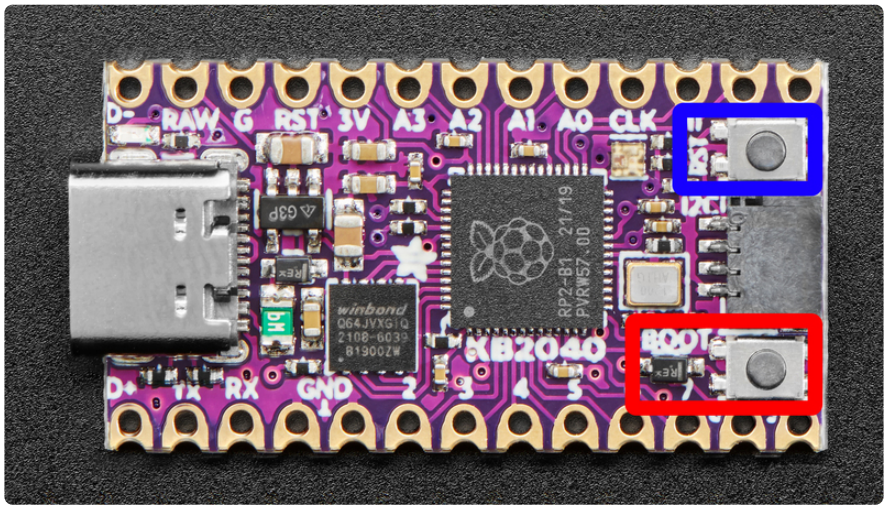
CircuitPython Quickstart

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



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

Save it wherever is convenient for you.

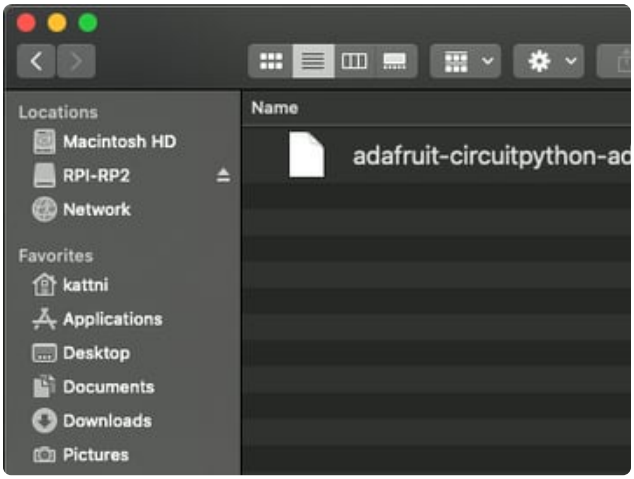


To enter the bootloader, hold down the **BOOT/BOOTSEL button** (highlighted in red above), and while continuing to hold it (don't let go!), press and release the **reset button** (highlighted in red or blue above). **Continue to hold the BOOT/BOOTSEL button until the RPI-RP2 drive appears!**

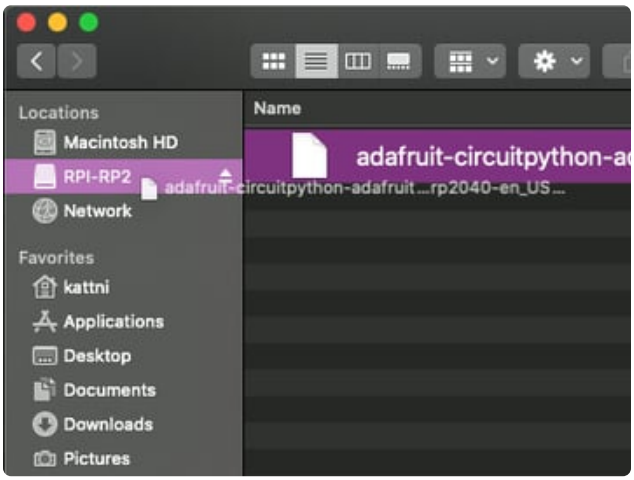
If the drive does not appear, release all the buttons, and then repeat the process above.

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.

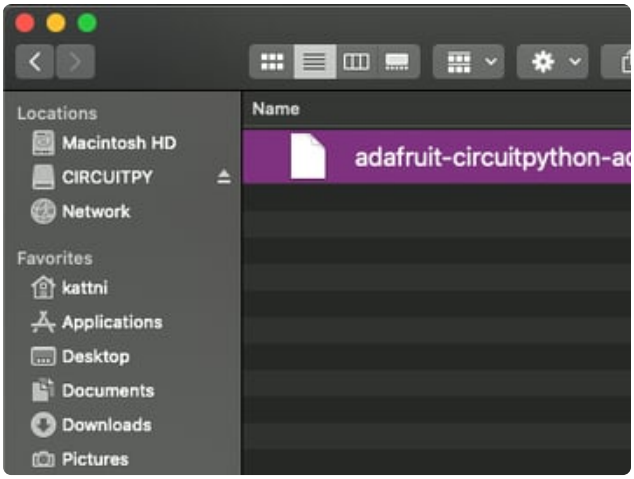
A lot of people end up using charge-only USB cables and it is very frustrating! **Make sure you have a USB cable you know is good for data sync.**



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

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 CIRCUITPY doesn't show up as a disk drive after installing CircuitPython, try loading this 'nuke' UF2 to RPI-RP2. 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.

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!

Text emphasized with a blue exclamation:

Mu is our recommended editor - please use it (unless you are an experienced coder with a favorite editor already!). While it has been announced end of life in 2026, it still works fine. You are free to use whatever text editor you wish along with a terminal program to connect to the CircuitPython REPL. Thonny is one such editor.

Download and Install Mu



Download Mu from https://codewith.mu ().

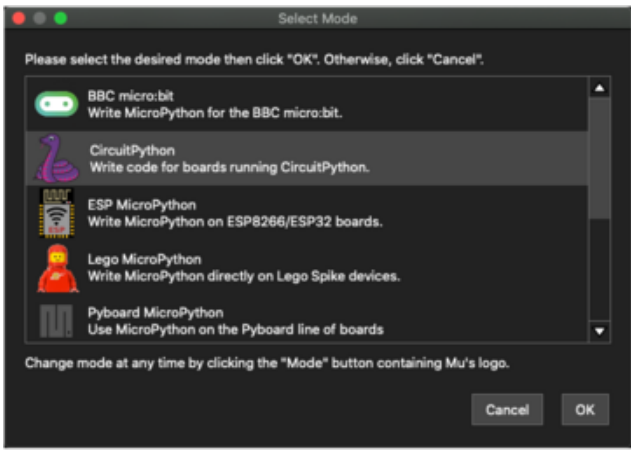
Click the **Download** link for downloads and installation instructions.

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

Text emphasized with a yellow exclamation:

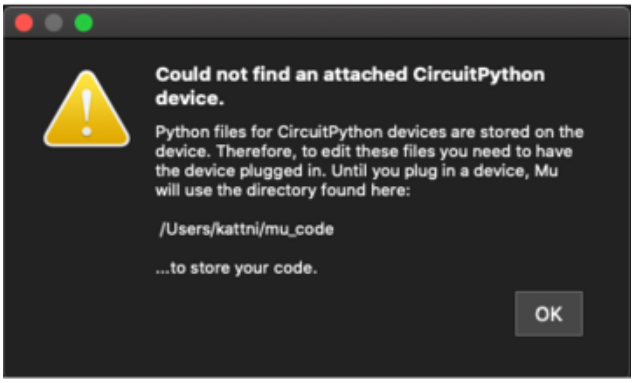
Windows users: due to the nature of MSI installers, please remove old versions of Mu before installing the latest version.

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...

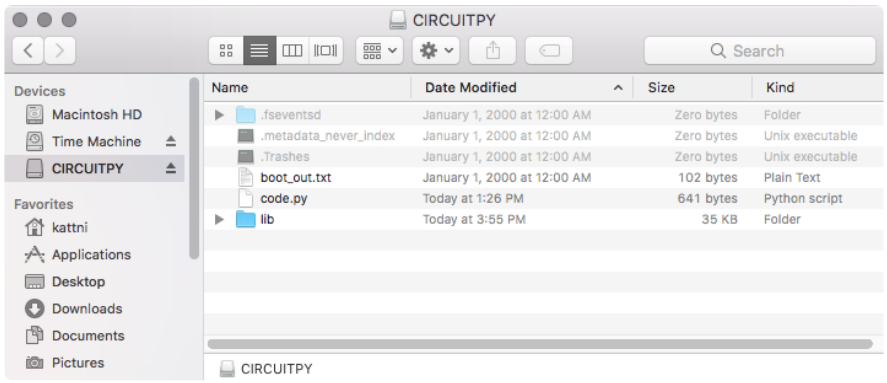
The CIRCUITPY Drive

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 (), 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 ()
- CircuitPython Web Workflow Code Editor Quick Start ()

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.

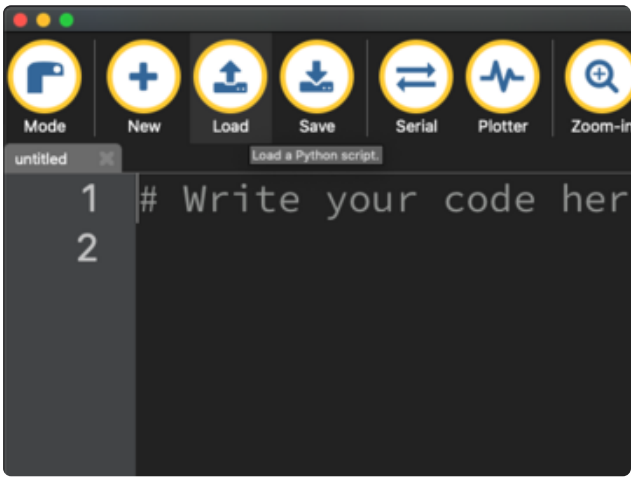
To create and edit code, all you'll need is an editor. There are many options. **Adafruit strongly recommends using Mu! It's designed for CircuitPython, and it's really simple and easy to use, with a built in serial console!**

If you don't or can't use Mu, there are a number of other editors that work quite well. The Recommended Editors page () 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.)

Text emphasized with a yellow exclamation:

macOS Sonoma 14.1 introduced a bug that delays writes to small drives such as CIRCUITPY drives. This caused errors when saving files to CIRCUITPY. There is a workaround. The bug was fixed in Sonoma 14.4, but at the cost of greatly slowed writes to drives 1GB or smaller.

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

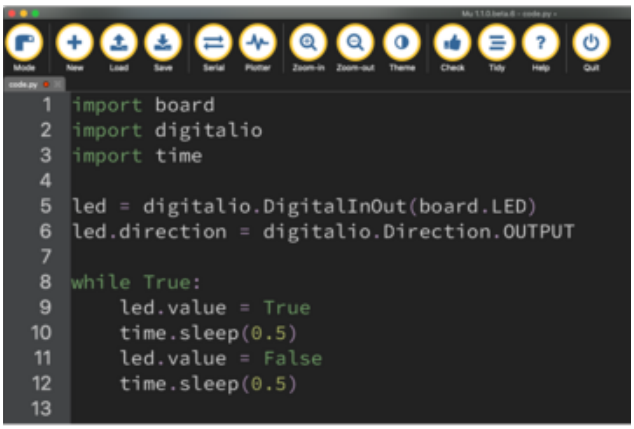
while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

Text emphasized with a red exclamation:

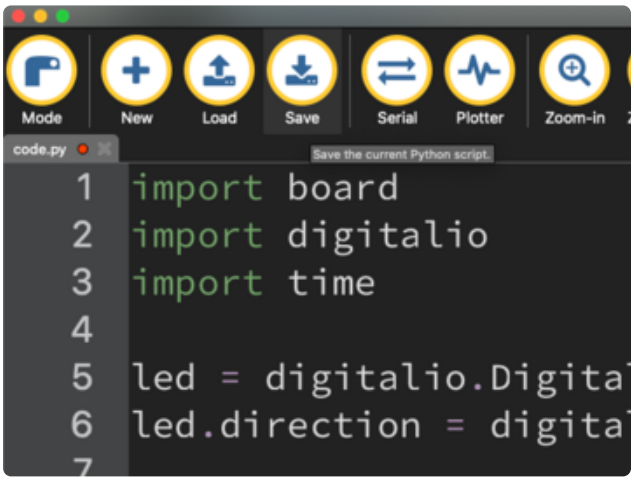
The KB2040, QT Py , Qualia, and the Trinkeys do not have a built-in little red LED! There is an addressable RGB NeoPixel LED. The above example will NOT work on the KB2040, QT Py, Qualia, or the Trinkeys!

If you're using a KB2040, QT Py, Quaila, or a Trinkey, or any other board without a single-color LED that can blink, please download the NeoPixel blink example ().

Text emphasized with a blue exclamation:	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.

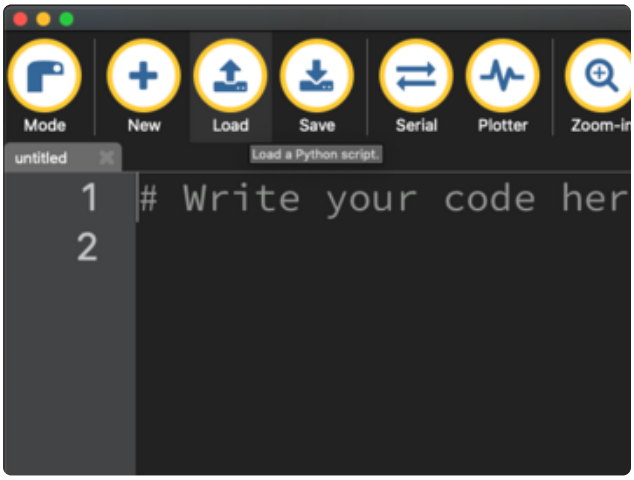
The little LED should now be blinking. Once per half-second.

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

Text emphasized with a blue exclamation:

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, Qualia, 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...

Text emphasized with a red exclamation:

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 () for details on different editing options.

Text emphasized with a blue exclamation:	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 () page of every board guide to get your board up and running again.

Text emphasized with a blue exclamation:

If you are having trouble saving code on Windows 10, try including this code snippet at the top of code.py:

```
import supervisor
supervisor.runtime.autoreload = False
```

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 for the LED blink example (if your board doesn't have a single-color LED to blink, look instead at the NeoPixel blink example):

```
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:

```
1 while True:
2     pass
```

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

See also the Behavior section in the docs ().

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!
```

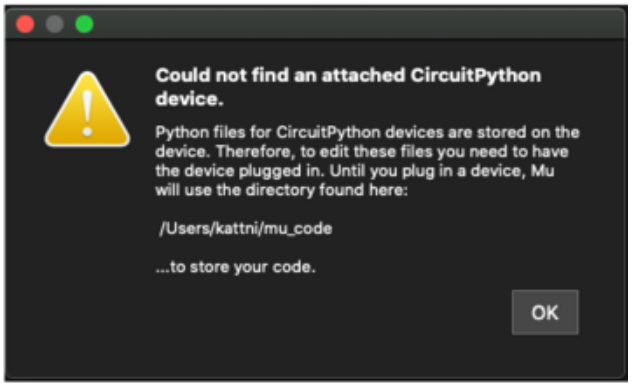
However, these print statements need somewhere to display. That's where the serial console comes in!

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 ().

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.



Text emphasized with a blue exclamation:	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 () for details on how to add yourself to the right group.

Using Something Else?

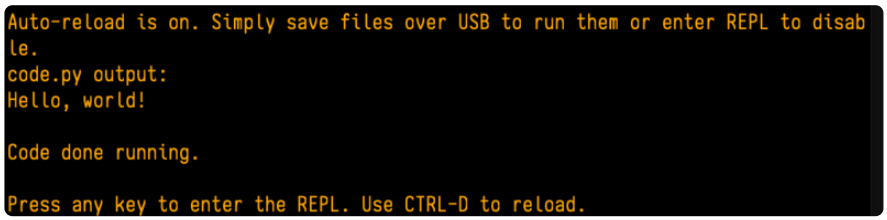
If you're not using Mu to edit, or if for some reason you are not a fan of its built in serial console, you can run the serial console from a separate program.

Windows requires you to download a terminal program. Check out the Advanced Serial Console on Windows page for more details. ()

MacOS has serial connection programs you can run in Terminal. Check the Advanced Serial Console on Mac page for more details. ()

Linux has multiple terminal programs included options are available for download. Check the Advanced Serial Console on Linux page for more details. ()

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.



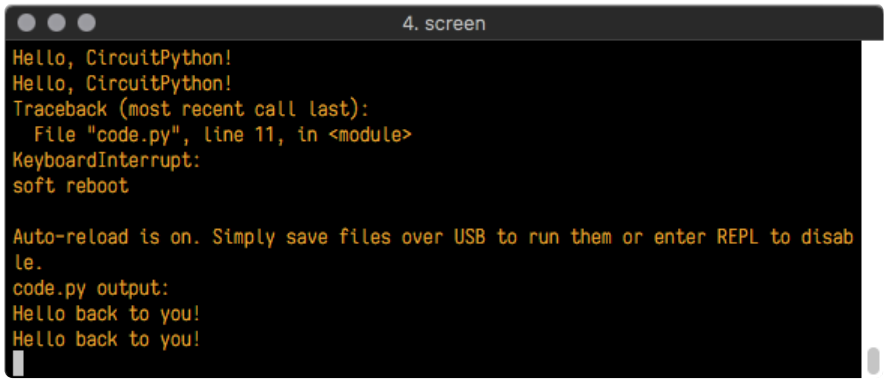
Excellent! Our print statement is showing up in our console! Try changing the printed text to something else.

```
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`

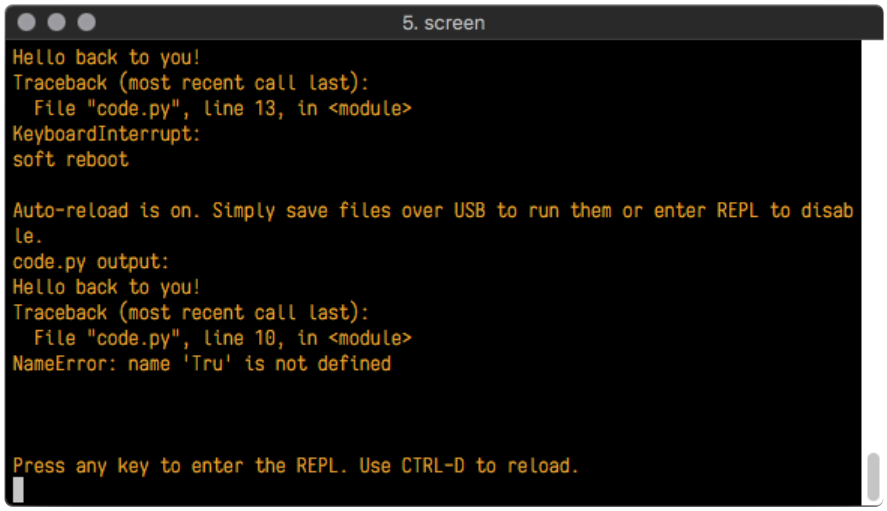
```
import board
import digitalio
import time

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!

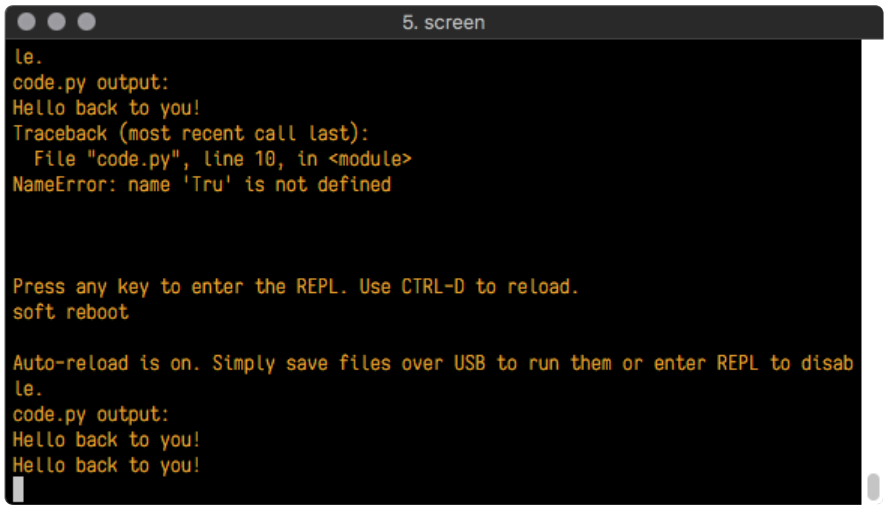
Usually when you run into errors, it's not because you introduced them on purpose. You may have 200 lines of code, and have no idea where your error could be hiding. This is where the serial console can help. Let's take a look!



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

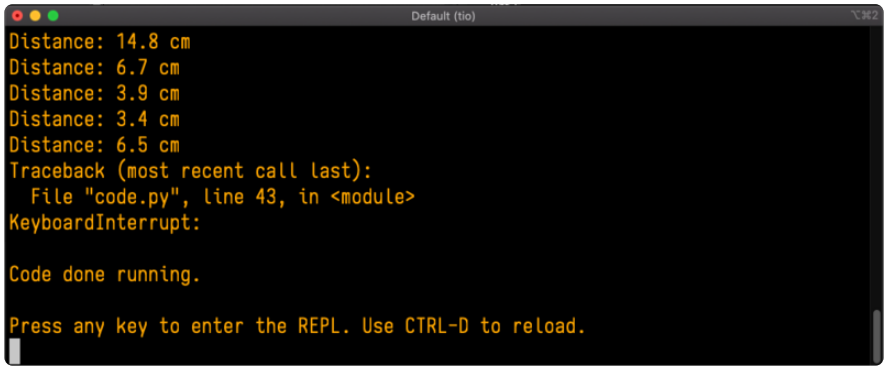
The other feature of the serial connection is the **R**ead-**E**valuate-**P**rint-**L**oop, or REPL. The REPL allows you to enter individual lines of code and have them run immediately. It's really handy if you're running into trouble with a particular program and can't figure out why. It's interactive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

Entering the REPL

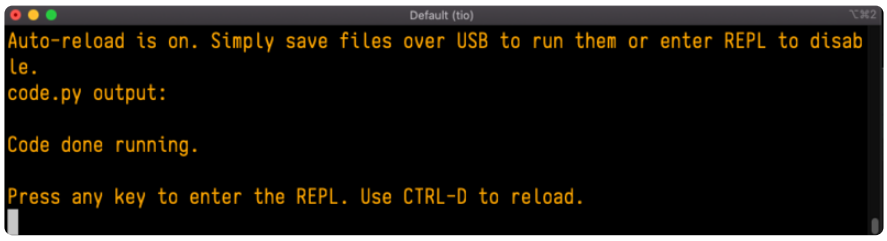
To use the REPL, you first need to be connected to the serial console. Once that connection has been established, you'll want to press **CTRL+C**.

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.

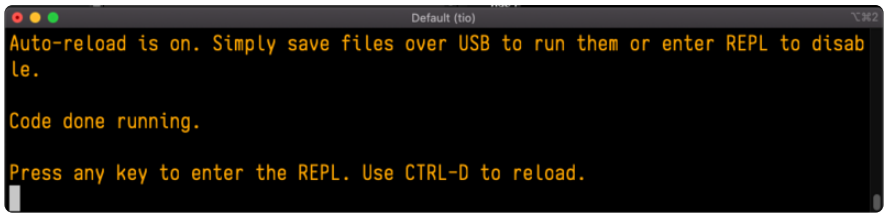
The `Traceback (most recent call last):` is telling you the last thing your board was doing before you pressed Ctrl + C and interrupted it. The `KeyboardInterrupt` is you pressing CTRL+C. This information can be handy when troubleshooting, but for now, don't worry about it. Just note that it is expected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

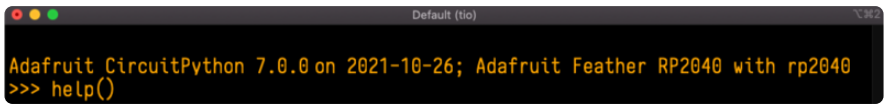
This is followed by the CircuitPython prompt.

```
>>>
```

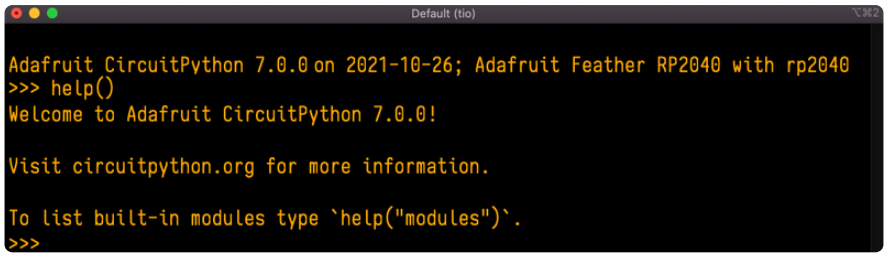
Interacting with the REPL

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.

```
>>> dir(board)
['__class__', '__name__', 'A0', 'A1', 'A2', 'A3', 'D0', 'D1', 'D10', 'D11', 'D12', 'D13',
'D24', 'D25', 'D4', 'D5', 'D6', 'D9', 'I2C', 'LED', 'MISO', 'MOSI', 'NEOPIXEL', 'RX', 'SCK
', 'SCL', 'SDA', 'SPI', 'TX', 'UART', 'board_id']
>>>
```

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.

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

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!

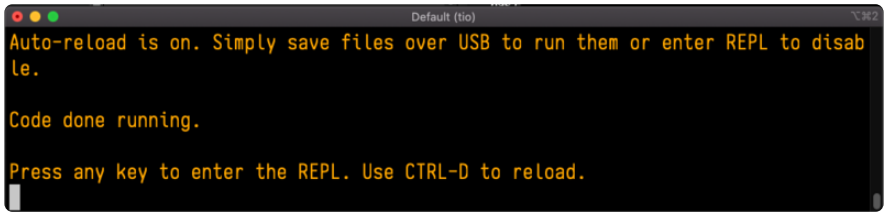
Text emphasized with a yellow exclamation:

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 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? () 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.

!	Text emphasized with a blue exclamation:	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:

```
1  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:

```
1  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())
```

> Text emphasized with a blue exclamation: 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.

> Text emphasized with a blue exclamation: 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

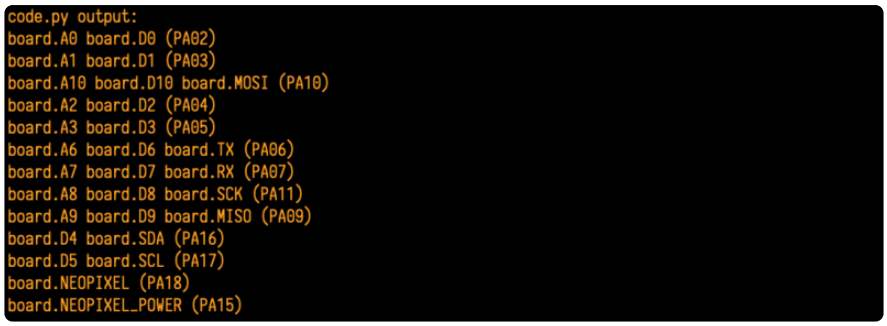
"""CircuitPython Essentials Pin Map Script"""
import microcontroller
import board
try:
    import cyw43  # raspberrypi
except ImportError:
    cyw43 = None

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)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Essentials/Pin_Map_Script/code.py

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



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 () and the Python-like modules included here ().
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 (), 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.

```
1 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.

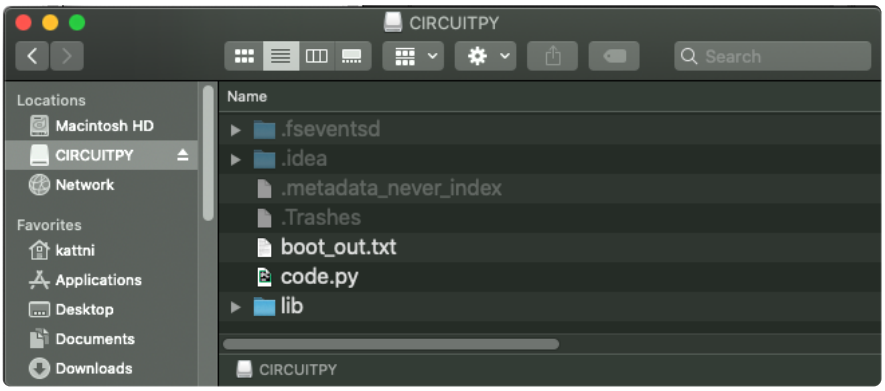
CircuitPython Libraries

Text emphasized with a red exclamation:

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 () 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.

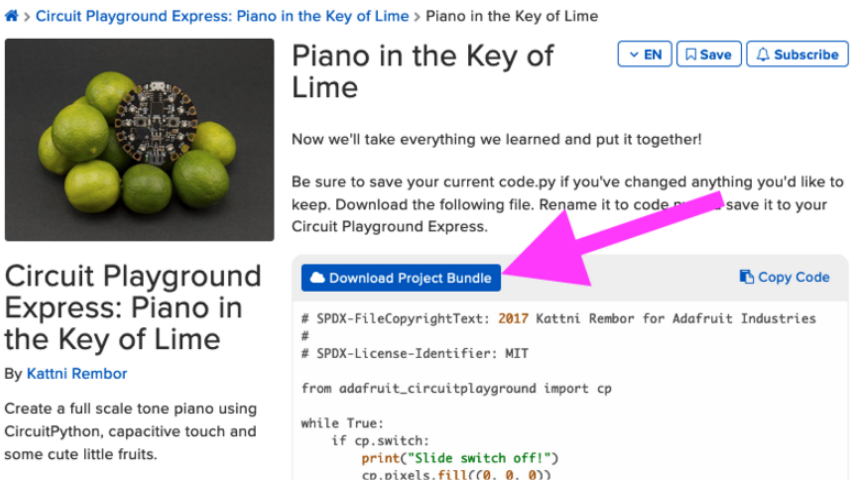
Either way, as you start to explore CircuitPython, you'll want to know how to get libraries on board.

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.

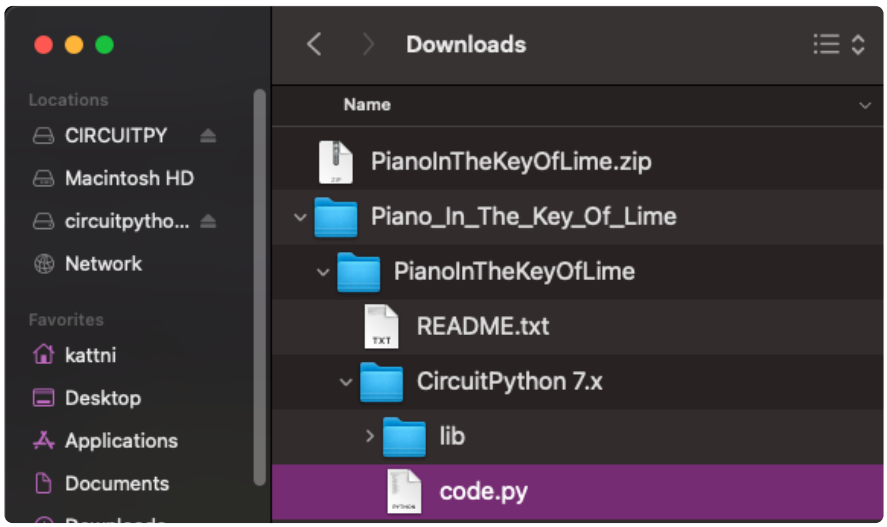
Text emphasized with a blue exclamation:	The Download Project Bundle button is only available on full demo code embedded from GitHub in a Learn guide. Code snippets will NOT have the button available.



Text emphasized with a yellow exclamation:

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.



Text emphasized with a blue exclamation:

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.

Text emphasized with a blue exclamation:

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.

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.

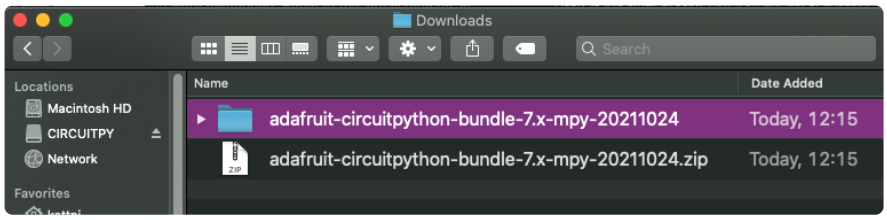
Downloading the CircuitPython Community Library Bundle

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.

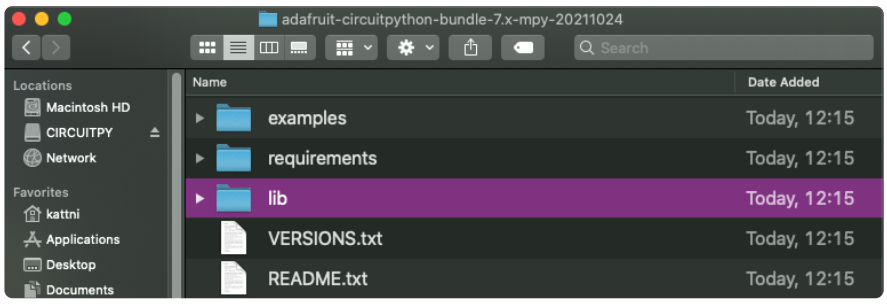
The link takes you to the latest release of the CircuitPython Community Library Bundle on GitHub. There are multiple versions of the bundle available. **Download the bundle version that matches your CircuitPython firmware version.** If you don't know the version, check the version info in **boot_out.txt** 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

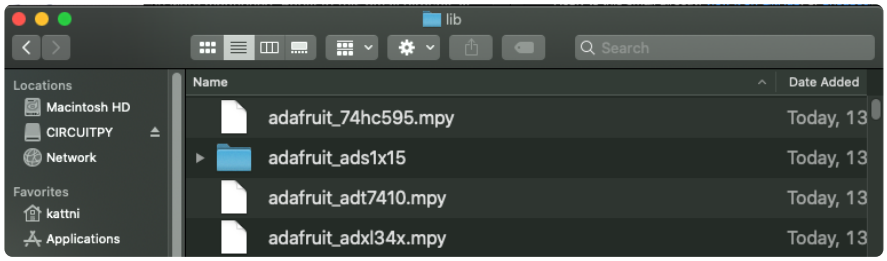
After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.



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.



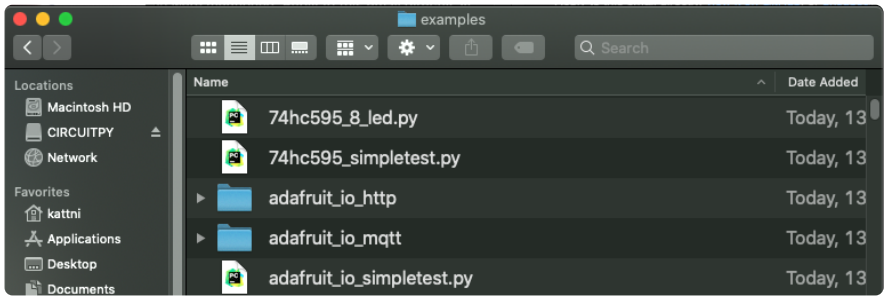
Now open the lib folder. When you open the folder, you'll see a large number of **.mpy** files, and folders.



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.

Text emphasized with a blue exclamation:

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.

```
1 import time
2 import board
3 import neopixel
4 import adafruit_lis3dh
```

```
5  import usb_hid
6  from adafruit_hid.consumer_control import ConsumerControl
7  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 () on The REPL page () 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.

```
>>> help("modules")
__main__          board             micropython       storage
_bleio            builtins          msgpack           struct
adafruit_bus_device                 busio             neopixel_write    supervisor
adafruit_pixelbuf collections       onewireio         synthio
aesio             countio           os                sys
alarm             digitalio         paralleldisplay   terminalio
analogio          displayio         pulseio           time
array             errno             pwmio             touchio
atexit            fontio            qrio              traceback
audiobusio        framebufferio     rainbowio         ulab
audiocore         gc                random            usb_cdc
audiomixer        getpass           re                usb_hid
audiomp3          imagecapture      rgbmatrix         usb_midi
audiopwmio        io                rotaryio          vectorio
binascii          json              rp2pio            watchdog
bitbangio         keypad            rtc
bitmaptools       math              sdcardio
bitops            microcontroller   sharpdisplay
```

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.

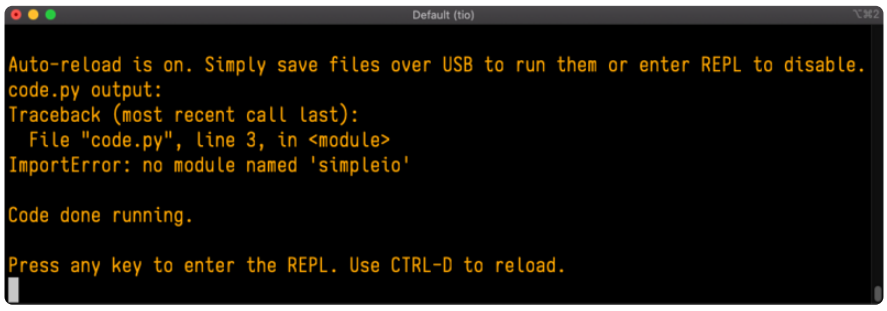
This demonstration will only return an error if you do not have the required library loaded into the **lib** folder on your **CIRCUITPY** drive.

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

```
1 import board
2 import time
3 import simpleio
```

```
4
5   led = simpleio.DigitalOut(board.LED)
6
7   while True:
8       led.value = True
9       time.sleep(0.5)
10      led.value = False
11      time.sleep(0.5)
```

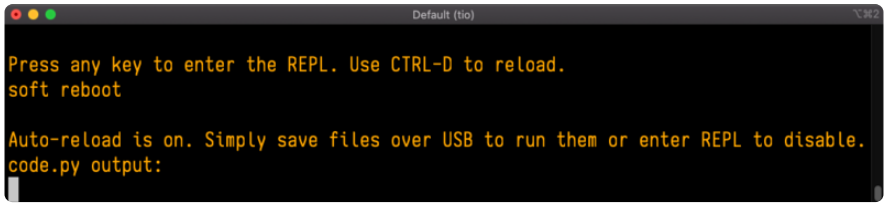
Save this file. Nothing happens to your board. Let's check the serial console to see what's going on.



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 ().

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 () 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 (). 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 () for a full list of functionality

---

CircuitPython Documentation

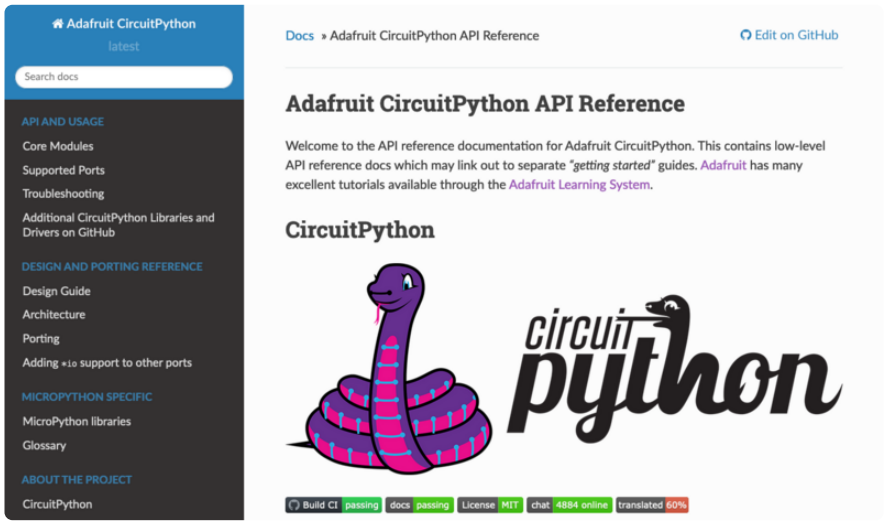
You've learned about the CircuitPython built-in modules and external libraries. You know that you can find the modules in CircuitPython, and the libraries in the Library Bundles. There are guides available that explain the basics of many of the modules and libraries. However, there's sometimes more capabilities 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 () and the Adafruit CircuitPython libraries ().

CircuitPython Core Documentation

The CircuitPython core documentation () 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!

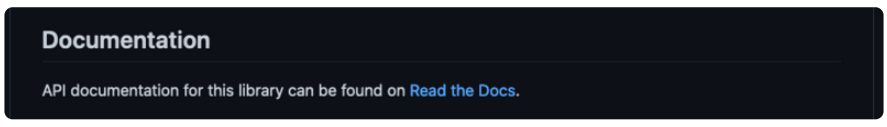
CircuitPython Library Documentation

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 (). 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 () 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.



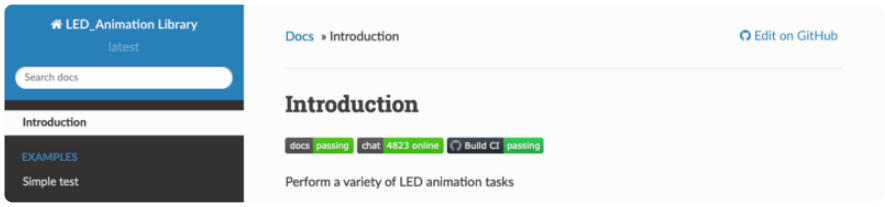
The second place is the **Documentation section** of the README. Scroll down to find it, and click on Read the Docs to get to the documentation.



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

Text emphasized with a blue exclamation:	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 () 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.



Text emphasized with a blue exclamation:

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 () 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.

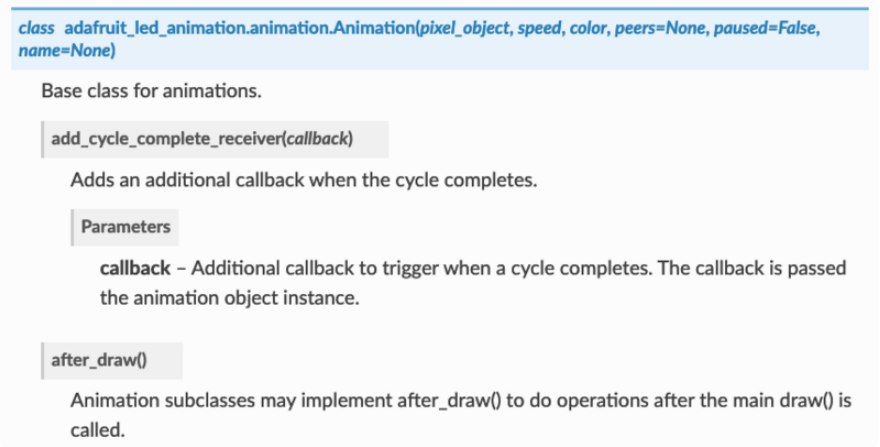
Click on the first item in the list to begin viewing the API Reference section.



Text emphasized with a blue exclamation:	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

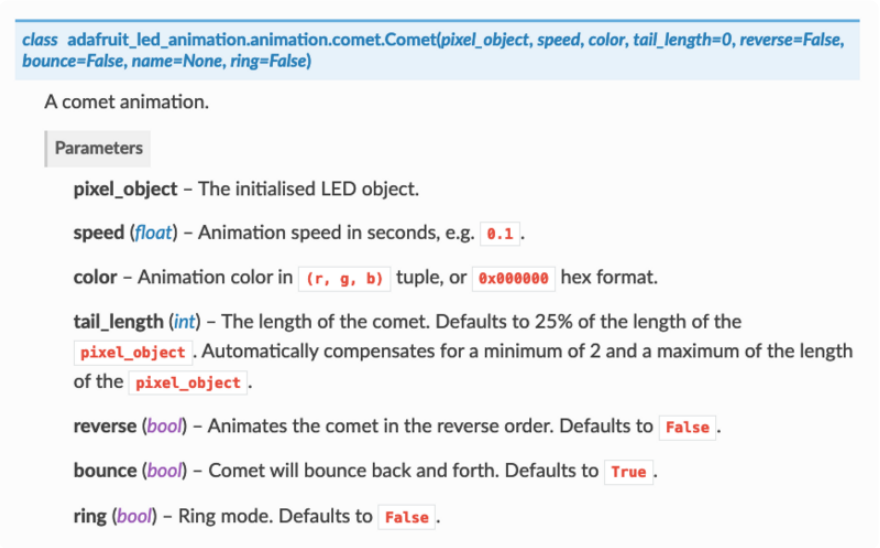
pixels = neopixel.NeoPixel(pixel_pin, pixel_num, brightness=0.5, auto_write=False)
```

```
26
27  comet = Comet(pixels, speed=0.02, color=JADE, tail_length=10, bounce=True)
28
29  while True:
30      comet.animate()
```

https://github.com/adafruit/Adafruit_CircuitPython_LED_Animation/blob/main/examples/led_animation_comet.py

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".

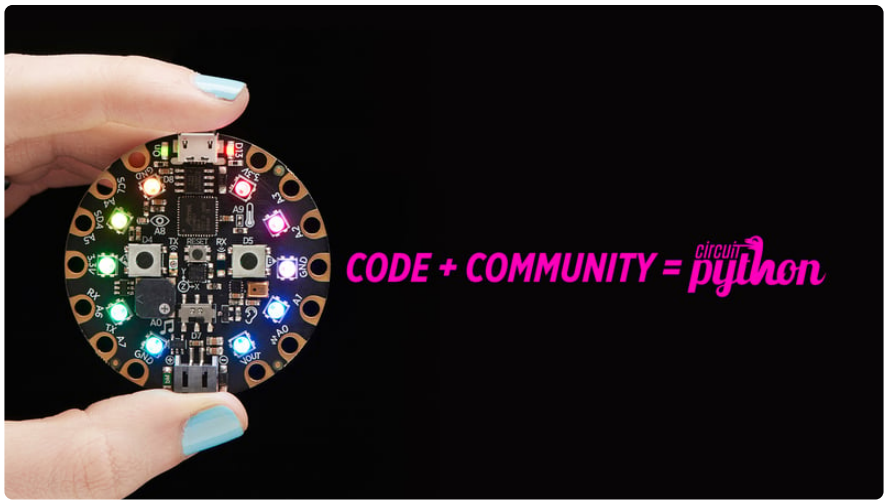
This type of information is available for any function you would set up in your code. If you need clarification on something, wonder whether there's more options available, or are simply interested in the details involved in the code you're writing, check out the documentation for the CircuitPython libraries!

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!

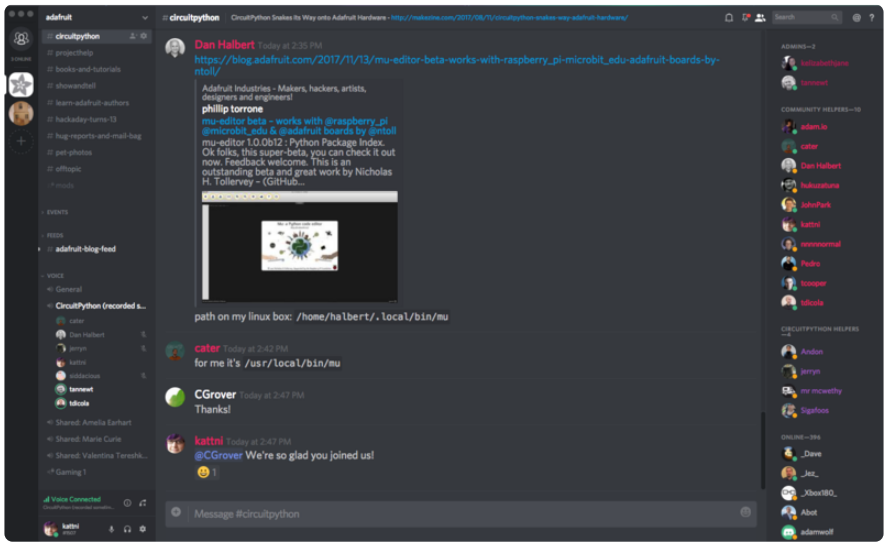
Welcome to the Community!



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.

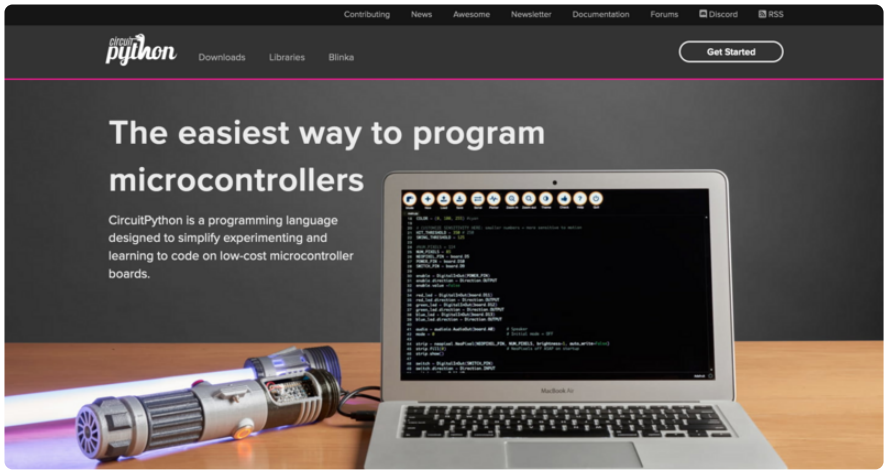
The help with CircuitPython channel is where to go with your CircuitPython questions. #help-with-circuitpython is there for new users and developers alike so feel free to ask a question or post a comment! Everyone of any experience level is welcome to join in on the conversation. Your contributions are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

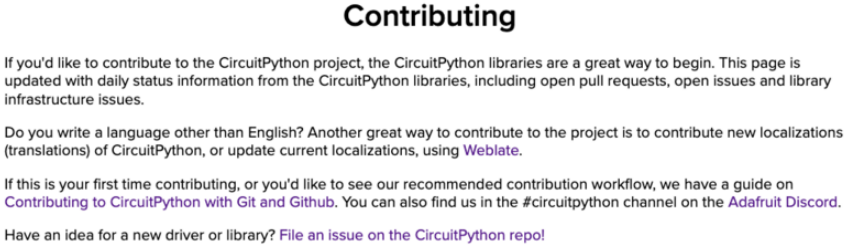
The Adafruit Discord is the 24x7x365 hackerspace that you can bring your granddaughter to.

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 (). Everything you need to get started with your new microcontroller and beyond is available. You can do things like download CircuitPython for your microcontroller () or download the latest CircuitPython Library bundle (), or check out which single board computers support Blinka (). 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 ().



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 (). 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**:

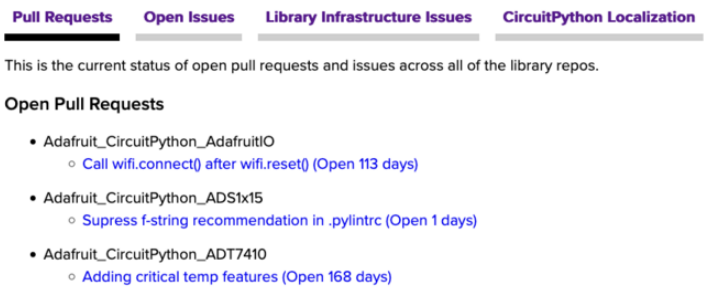


If you submit any contributions to the libraries, and do not see them reflected on the Contributing page, it could be that the job that checks for new updates hasn't yet run for today. Simply check back tomorrow!

Now, a look at the different options.

Pull Requests

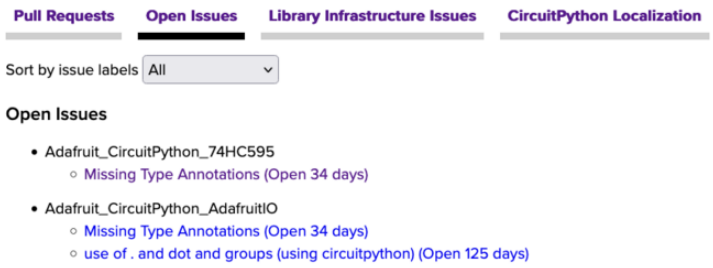
The first tab you'll find is a list of **open pull requests**.



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.

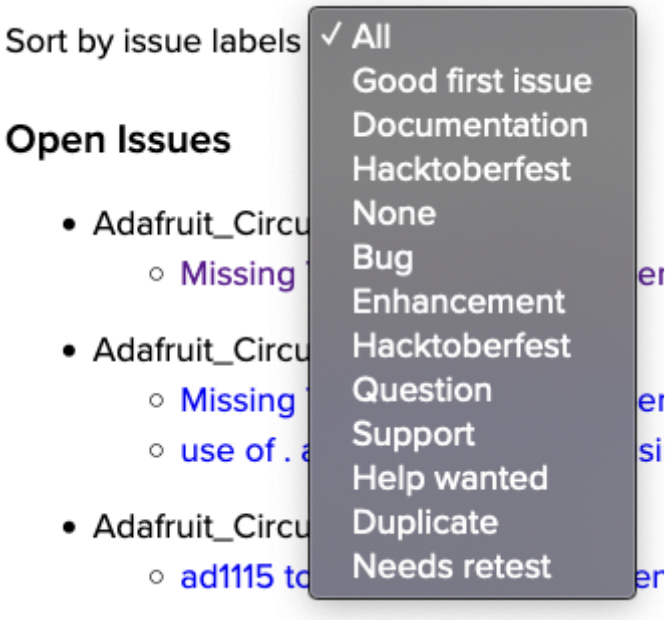
Open Issues

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

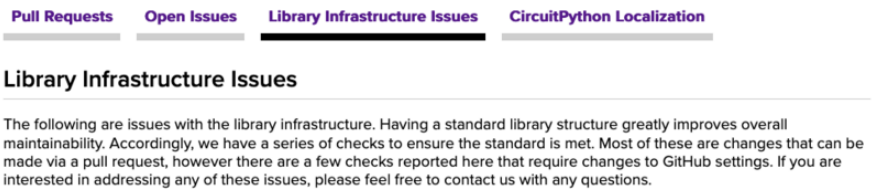
with that label are well defined, with a finite scope, and are intended to be easy for someone new to figure out.

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 () to walk you through the entire process. As well, there are always folks available on Discord () to answer questions.

Library Infrastructure Issues

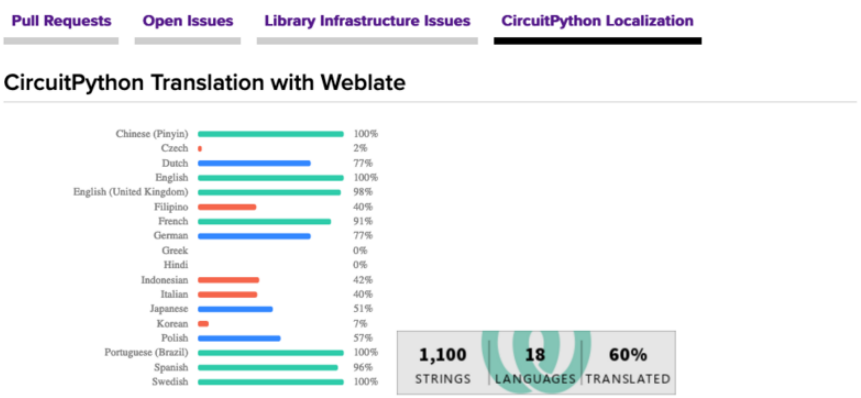
The third tab you'll find is a list of **library infrastructure issues**.



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

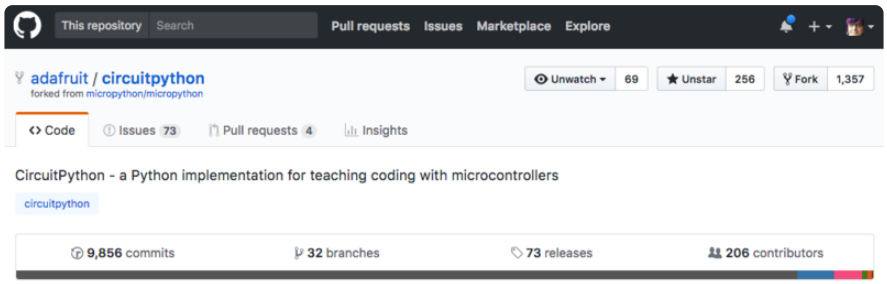
The fourth tab you'll find is the **CircuitPython Localization** tab.



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.

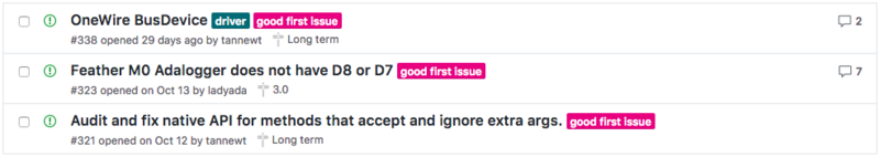
Regardless of your skill level, or how you want to contribute to the CircuitPython project, there is an opportunity available. The Contributing page () is an excellent place to start!

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 (), and the CircuitPython libraries (). If you need an account, visit https://github.com/ () 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 ()", and you'll find a list that includes issues labeled "good first issue ()". For the libraries, head over to the Contributing page Issues list (), and use the drop down menu to search for "good first issue ()". 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 ().



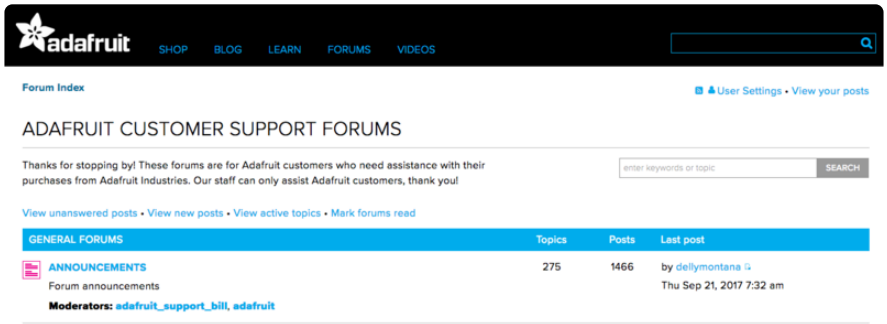
Already experienced and looking for a challenge? Checkout the rest of either issues list and you'll find plenty of ways to contribute. You'll find all sorts of things, from new driver requests, to library bugs, to core module updates. There's plenty of opportunities for everyone at any level!

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 (). 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!

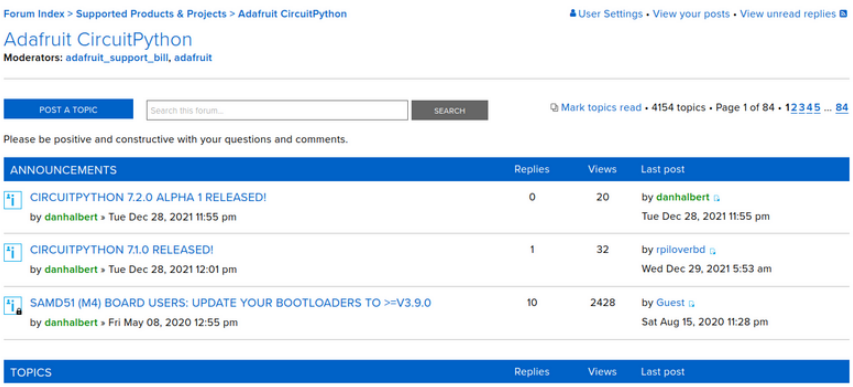
Adafruit Forums



The Adafruit Forums () 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.

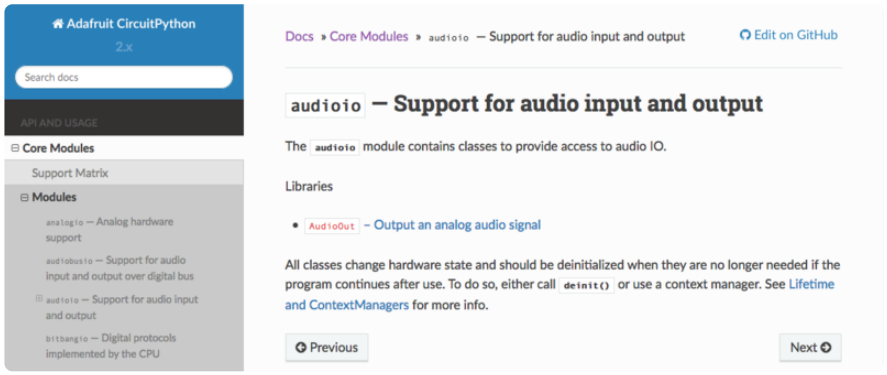
There are forum categories that cover all kinds of topics, including everything Adafruit. The Adafruit CircuitPython () category under "Supported Products & Projects" is the best place to post your CircuitPython questions.



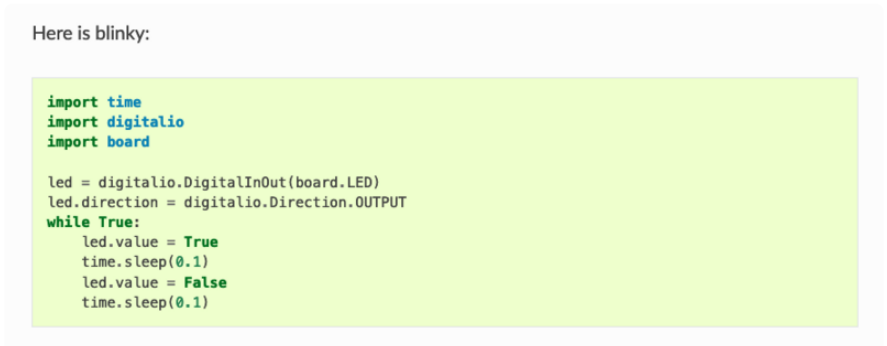
Be sure to include the steps you took to get to where you are. If it involves wiring, post a picture! If your code is giving you trouble, include your code in your post! These are great ways to make sure that there's enough information to help you with your issue.

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 () 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 () page!



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. On macOS, saves get slower for every swap or backup file an editor writes to CIRCUITPY. See below () to move them off the drive.

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** () is an editor that safely writes all changes (it's also our recommended editor!)
- **emacs** () is also an editor that will fully write files on save (). Set it up to keep backup and lock files off **CIRCUITPY**, as shown below ().
- Sublime Text **()** safely writes all changes
- Visual Studio Code **()** appears to safely write all changes
- **gedit** on Linux appears to safely write all changes
- IDLE (), in Python 3.8.1 or later, was fixed () to write all changes immediately
- Thonny () fully writes files on save
- Notepad++ () flushes files after writes, as of several years ago. In addition, you can change the path used for "Enable session snapshot and periodic backup" to write somewhere else than the CIRCUITPY drive. This will save space on CIRCUITPY and reduce writes to the drive.

Recommended only with particular settings or add-ons

- **vim** () / **vi** safely writes all changes. But set up **vim** to not write swapfiles () (.swp files: temporary records of your edits) to **CIRCUITPY**. Otherwise the swapfile writes trigger restarts of your program and make every save slower. Use the `.vimrc` settings below ().
- The PyCharm IDE **()** is safe if "Safe Write" is turned on in Settings->System Settings->Synchronization (true by default).
- If you are using **Atom** (), install the fsync-on-save package () or the language-circuitpython package () so that it will always write out all changes to files on **CIRCUITPY**.

- **SlickEdit** () works only if you add a macro to flush the disk ().

!	Text emphasized with a red exclamation:	The editors listed below are specifically NOT recommended!

Editors that are NOT recommended

- **notepad** (the default Windows editor) 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. Later versions do force out changes.
- **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!

Moving editor swap files off CIRCUITPY

On macOS, every swap, backup or lock file an editor writes to **CIRCUITPY** adds time to each save. Keep them on your computer instead.

vim

Make a swap folder in your home directory, then add these lines to `~/.vimrc`. Saving `code.py` went from 4.5 seconds to 1.1 seconds.

```
mkdir -p ~/.vim/swap
```

```
" keep swap files on the local disk, not next to the file (slow on CIRCUITPY)
set directory^=~/.vim/swap//
" CIRCUITPY: skip the temporary backup copy on save
autocmd BufReadPre,BufNewFile /Volumes/CIRCUITPY*/* setlocal nowritebackup
```

emacs

Make backup and auto-save folders in your home directory, then add these lines to `~/.emacs.d/init.el`. Saving went from 1.5 seconds to 1.1 seconds, with nothing extra written to **CIRCUITPY**.

```
mkdir -p ~/.emacs.d/backups ~/.emacs.d/auto-save
```

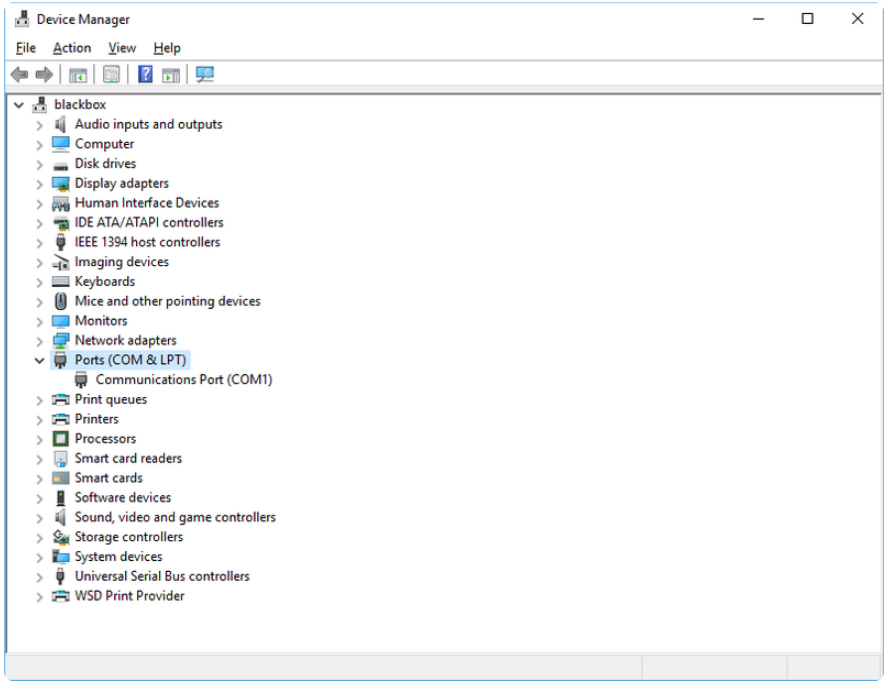
```
;; keep backup, auto-save and lock files off CIRCUITPY
(setq backup-directory-alist '(("." . "~/.emacs.d/backups/")))
(setq auto-save-file-name-transforms '((".*" "~/.emacs.d/auto-save/" t)))
(setq backup-by-copying t)
(setq create-lockfiles nil)
```

Advanced Serial Console on Windows

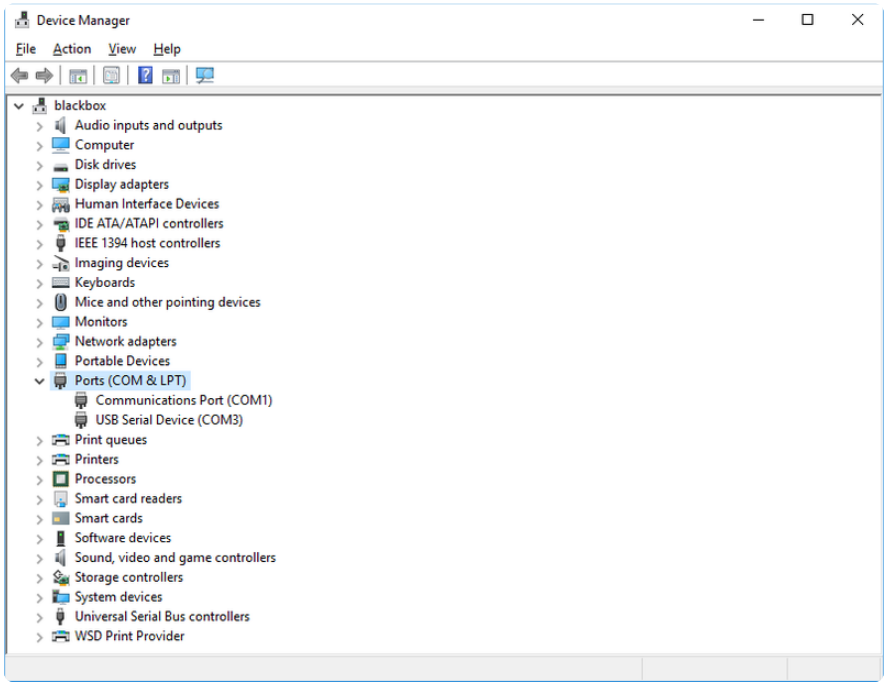
What's the COM?

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.

Windows Serial Port Terminal Programs

- Putty is a venerable serial port connection program. More details are below.
- Tera Term () is a nice terminal program. It will reconnect automatically after disconnections
- VSCode has a number of serial port extensions, such as Serial Monitor ().
- PyCharm has a Serial Port Monitor () plugin.

Install Putty

PuTTY is a well-known choice for connecting to serial ports on Windows.

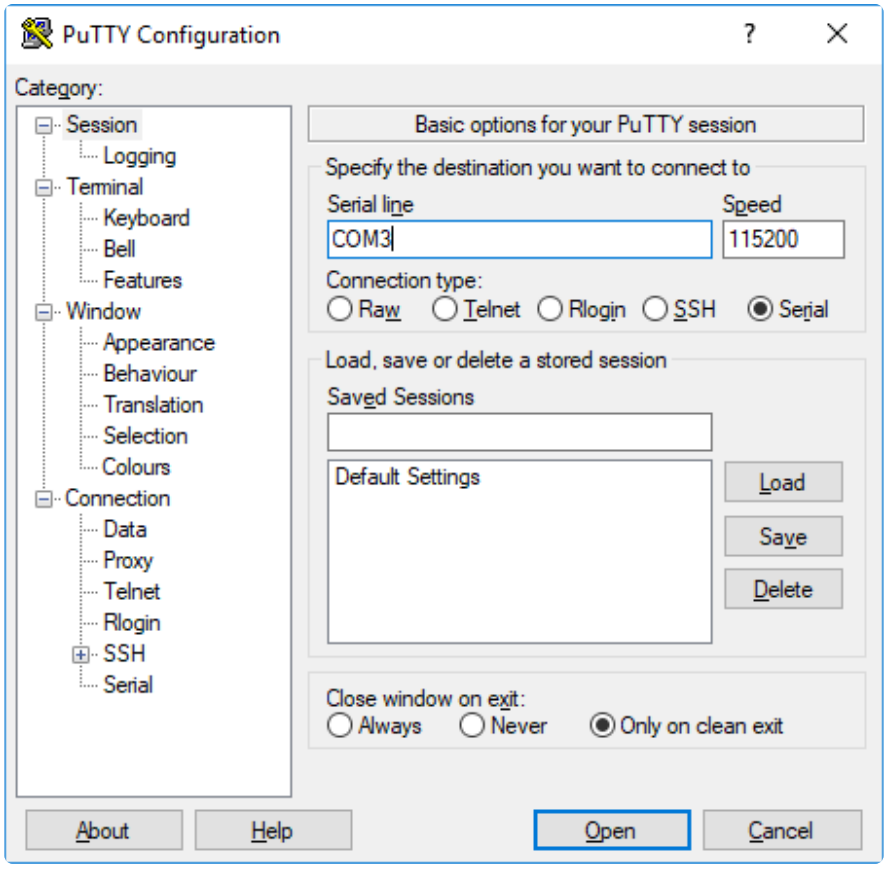
The first thing to do is download the latest version of PuTTY (). You'll want to download the Windows installer file. It is most likely that you'll need the 64-bit version. Download the file and install the program on your machine. If you run into issues, you

can try downloading the 32-bit version instead. However, the 64-bit version will work on most PCs.

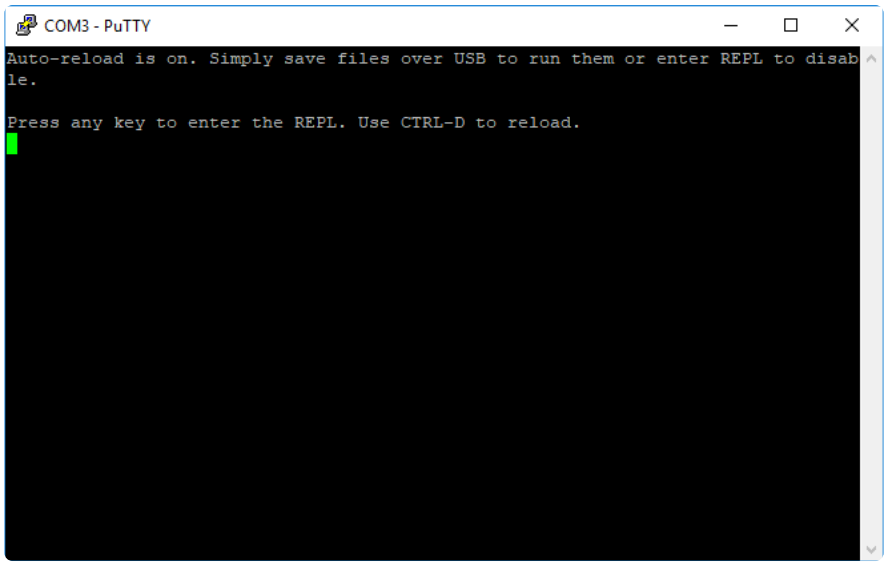
Now you need to open PuTTY.

- 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!

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 () for details. You will not need to install drivers on Mac, Linux or Windows 10 or 11.

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.

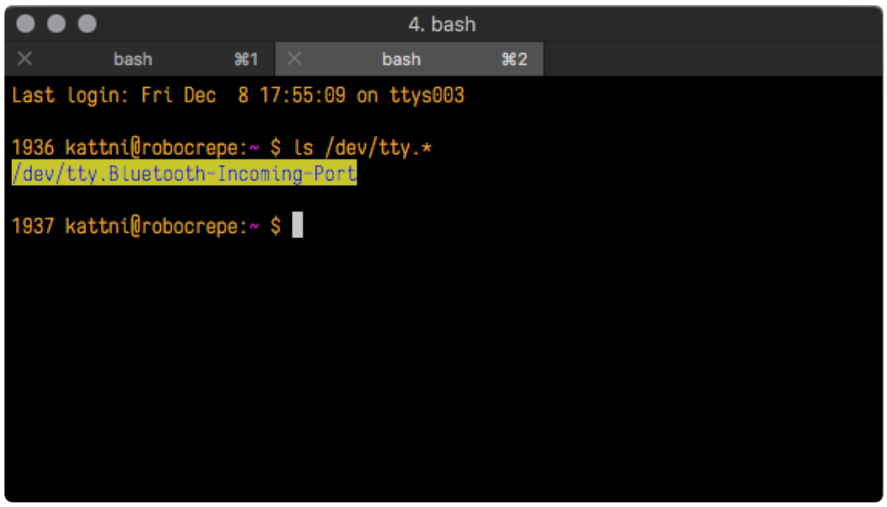
What's the Port?

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.*
```

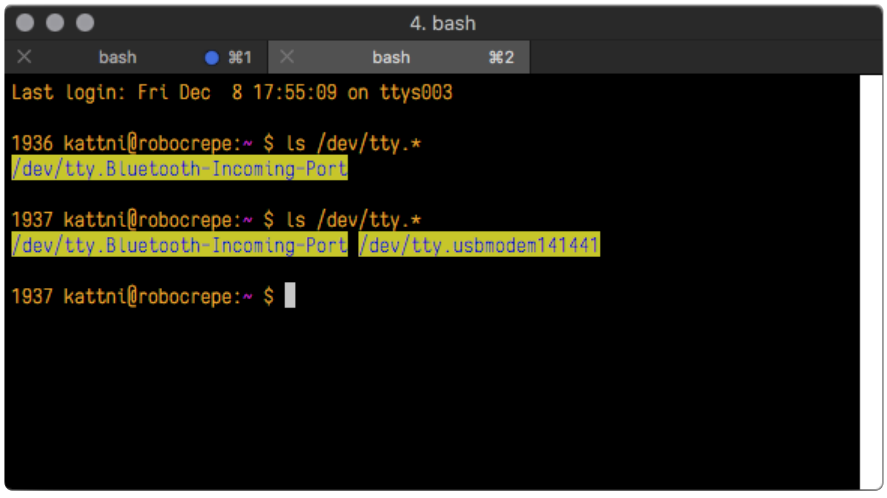
Each serial connection shows up in the `/dev/` directory. It has a name that starts with `tty.`. The command `ls` shows you a list of items in a directory. You can use `*` as a wildcard, to search for files that start with the same letters but end in something different. In this case, you're asking to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

!	Text emphasized with a red exclamation:	Using the `screen` terminal program can cause your CircuitPython program to hang when trying to print, if you exit `screen` after you've used it to connect.

macOS Serial Port Terminal Programs

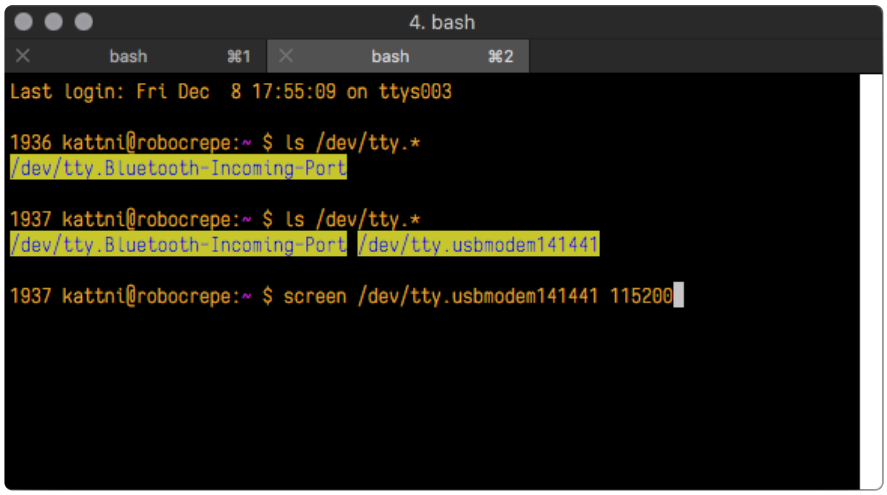
- `screen` is included with macOS. However, it's problematic because when it starts up, it enables the using DTR/RTS flow control signals and does not turn that off when it quits. This causes CircuitPython to block sending output when `screen` has exited, which will cause your program to stall until it is reconnected. See this issue () for a discussion.
- `tio ()` is a nice terminal program that works properly. You can install it with Homebrew ().
- VSCode has a number of serial port extensions, such as Serial Monitor ().
- PyCharm has a Serial Port Monitor () plugin.

Connect with screen

Despite the caveats above, if you can't download a better terminal program, you can use `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.

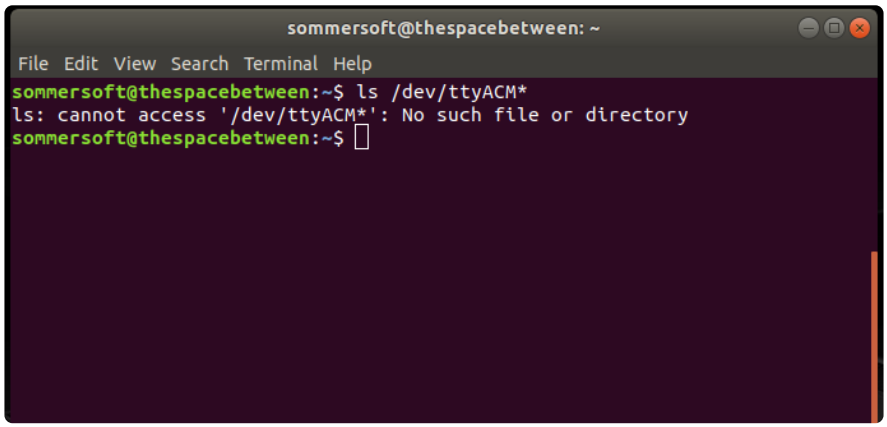
The easiest way to determine which port the board is using is to first check **without** the board plugged in. Open your terminal program and type the following:

```
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.

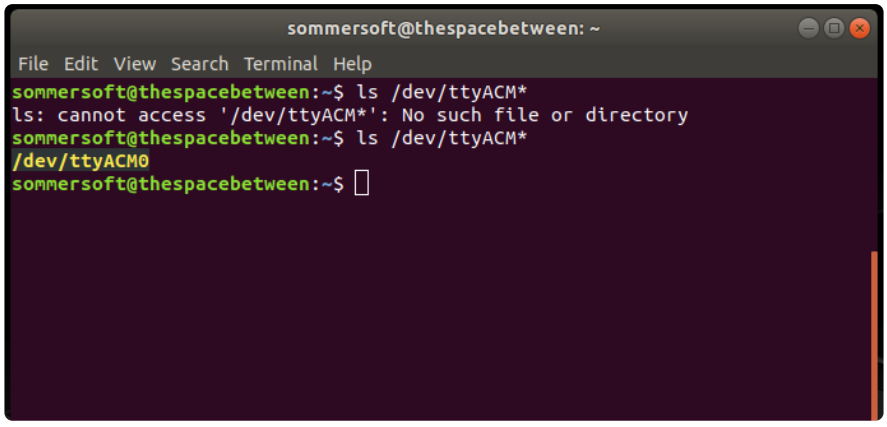
In the example below, the error is indicating that are no current serial connections starting with **ttyACM**.



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.

Linux Serial Port Terminal Programs

- `tio ()` is a nice terminal program that works properly. You can install it using your package manager.

- VSCode has a number of serial port extensions, such as Serial Monitor ().
- PyCharm has a Serial Port Monitor () plugin.

Connect with `tio`

Now that you know the name your board is using, you're ready connect to the serial console. Install `tio` using your package manager..

ype the following command, replacing `ttyACMx` with the name you found your board is using:

`tio /dev/ttyACMx`

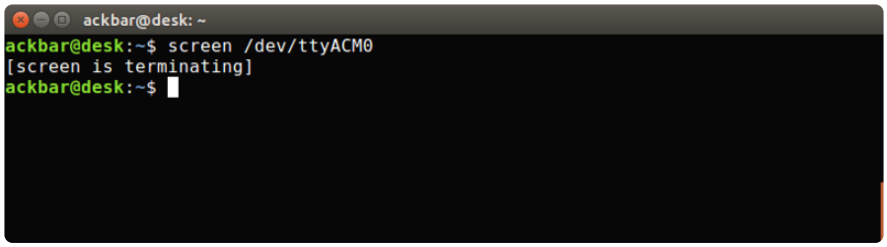
This will connect to the serial port and display a header like this:

```
$ tio /dev/ttyACM0
[11:07:00.578] tio v2.7
[11:07:00.578] Press ctrl-t q to quit
```

Permissions on Linux

If you try to run `tio` and it doesn't work, then you may be running into an issue with permissions. Your Linux distribution may not allow access to serial ports by defaut. You may see something like this; note the "permission denied".

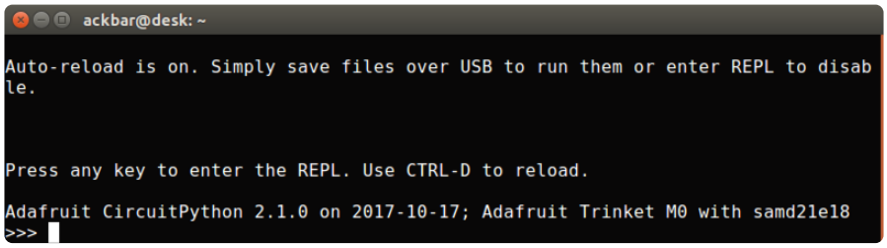
```
[11:13:42.754] tio v2.7
[11:13:42.754] Press ctrl-t q to quit
[11:13:42.754] Warning: Could not open tty device (Permission denied)
[11:13:42.754] Waiting for tty device..
```



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.

```
$ sudo tio /dev/ttyACM0
[sudo] password for smith:
```

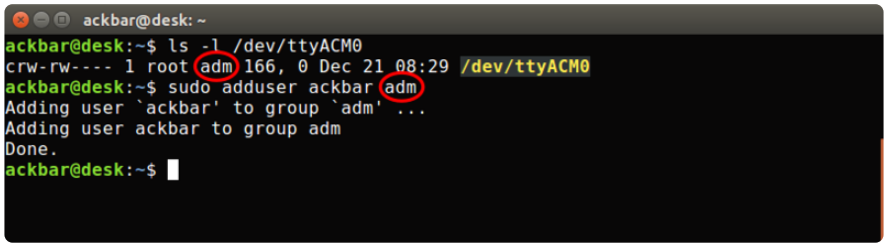
Once you enter your password, you should be in:



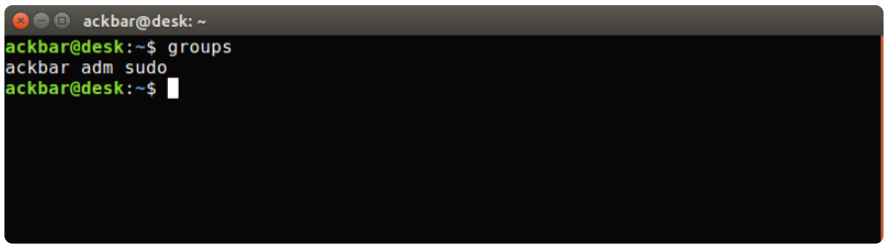
The second way is to add yourself to the user group associated with the hardware. To figure out what that group is, use the command `ls -l` as shown below. The group name is circled in red.

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**.

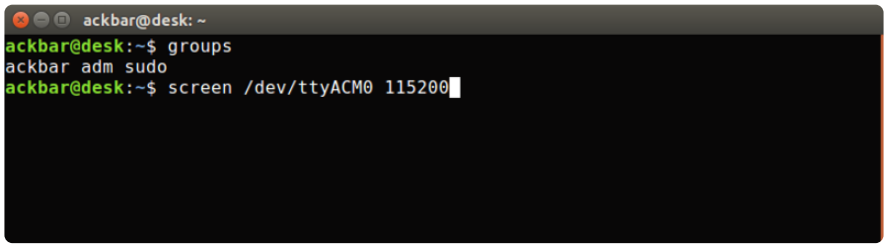
On Debian-based distributions, such as,Ubuntu the group will be **dialout**, not **adm**.



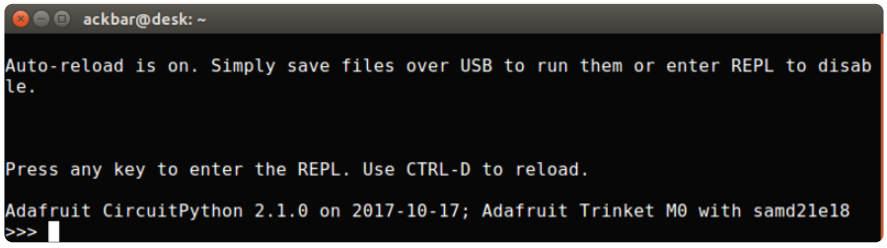
After you add yourself to the group, you'll need to logout and log back in, or in some cases, reboot your machine. After you log in again, verify that you have been added to the group using the command `groups`. If you are still not in the group, reboot and check again.



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



And you're in:



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 ()
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

Text emphasized with a red exclamation:

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 8.x or earlier. Where can I find compatible libraries?



We are no longer building or supporting the CircuitPython 8.x or earlier library bundles. We highly encourage you to update CircuitPython to the latest version **() and use** the current version of the libraries **()**. However, if for some reason you cannot update, here are the last available library bundles for older versions:

- 2.x bundle ()
- 3.x bundle ()
- 4.x bundle ()
- 5.x bundle ()
- 6.x bundle ()
- 7.x bundle ()
- 8.x bundle ()

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 () 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 ().



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



nRF52840, nRF52833, and **as of CircuitPython 9.1.0**, ESP32, ESP32-C3, and ESP32-S3 boards (with 8MB) 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.

Most Espressif boards with only 4MB of flash do not have enough room to include BLE in CircuitPython 9. Check the Module Support Matrix () to see if your board has support for `_bleio`. CircuitPython 10 is planned to support `_bleio` on Espressif boards with 4MB flash.

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 () or other NINA-FW-based co-processors. Some boards have this coprocessor on board, such as the PyPortal (). 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 ()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 () 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! ()

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 () for your version of CircuitPython.

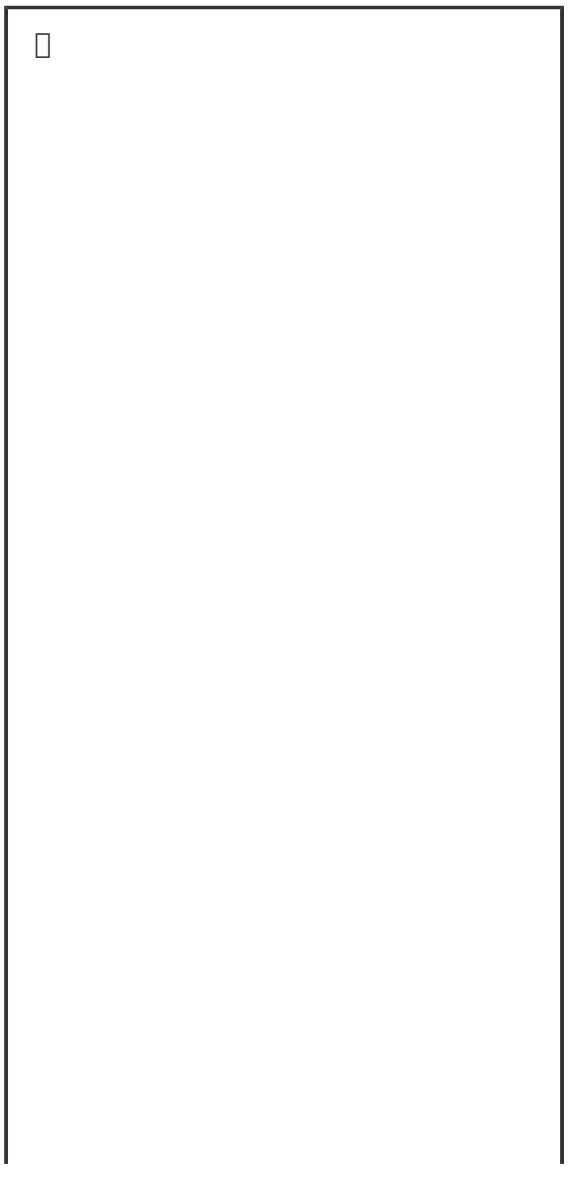
If that does not resolve your issue, try shortening your code. Shorten comments, remove extraneous or unneeded code, or any other clean up you can do to shorten your code. If you're using a lot of functions, you could try moving those into a separate library, creating a **.mpy** of that library, and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

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 (). 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.

On macOS and Linux, after you download **mpy-cross**, you must make the the file executable by doing `chmod +x name-of-the-mpy-cross-executable`.

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 ()!

As of CircuitPython 8.x we have started to support ESP32 and ESP32-C3 and have added a WiFi workflow for wireless coding! ()

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.

Text emphasized with a red exclamation:

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 **().**

You need to download the CircuitPython Library Bundle that matches your version of CircuitPython. **Please update CircuitPython and then** download the latest bundle **()**.

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 **()** **and use** the current version of the libraries **().** However, if for some reason you cannot update, links to the previous bundles are available in the FAQ ().

macOS Sonoma before 14.4: Errors Writing to CIRCUITPY
macOS 14.4 - 15.1: Slow Writes to CIRCUITPY

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

Below is a shell script to do this remount conveniently (courtesy @czei in GitHub ()). 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 and versions of macOS before Sequoia 15.2 did not have the problem above, but did take an inordinately long time to write to FAT drives of size 1GB or less (40 times longer than 2GB drives). As of macOS 15.2, writes are no longer very slow. This problem was tracked in CircuitPython GitHub issue 8918 ().

```
#!/bin/sh
#
# This works around bug where, by default,
# macOS 14.x before 14.4 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 ().

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 ()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 () 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 () for how to fix the problem.

Windows 10 or later

Did you install the Adafruit Windows Drivers package by mistake, or did you upgrade to Windows 10 or later with the driver package installed? You don't need to install this package on Windows 10 or 11for 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

Windows 7 and 8.1 have reached end of life. It is recommended () that you upgrade to Windows 10 or 11 if possible. Drivers are available for some older CircuitPython boards, but there are no plans to release drivers for newer boards.

Text emphasized with a blue exclamation:	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 newer boards. The boards work fine on Windows 10 and later.

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 () 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.
- **BitDefender anti-virus**
- **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.

boardnameBOOT on SAMD boards Does Not Appear or is Inaccessible on Linux or Chromebooks

UF2 bootloaders before version v4.0.0 on SAMD21 or SAMx5x boards had a bug that caused the BOOT drive to not show up or lock up when accessed. The fix is to update the bootloader. See the **Update the UF2 Bootloader** page in the Learn Guide for your board. For example, for the Circuit Playground Express, see this page ().

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

BitDefender anti-virus has been reported to block access to CIRCUITPY. You can set an exception for the drive letter.

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 () and the BOOT drive to reappear. It is not clear what causes this behavior.

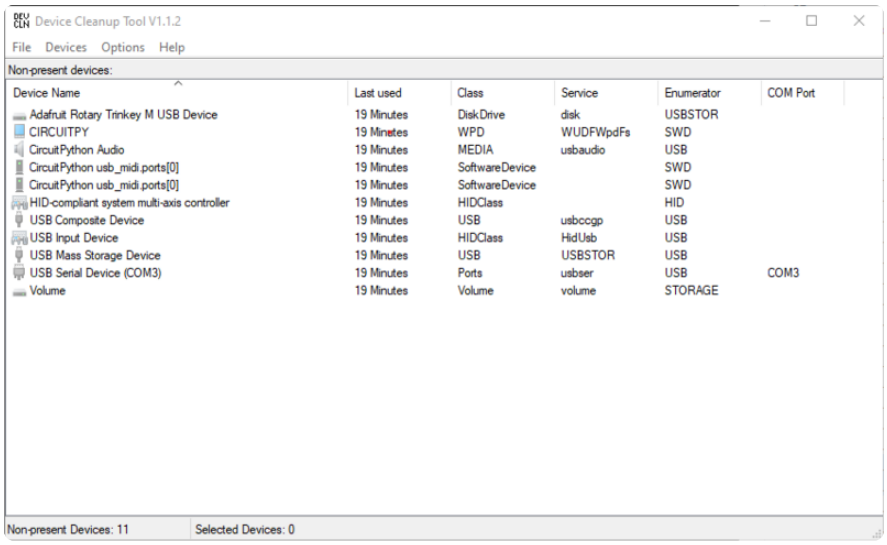
Samsung Magician can cause CIRCUITPY to disappear (reported here () and here ()).

"M105" Seen on Display, Crashes, Missing CIRCUITPY

The **Cura** 3D printing program sends 3D printing GCODE commands to unused serial ports to try to find 3D printers connected over serial. This causes a variety of problems. Disable (uncheck) **USB Printing** in Cura in the **Market** -> **Installed** menu, or uninstall Cura. For more information see this forum post (), this CircuitPython issue (), and this Cura issue ().

Device Errors or Problems on Windows

Windows can become confused about USB device installations. Try cleaning up your USB devices. Use Uwe Sieber's Device Cleanup Tool () (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

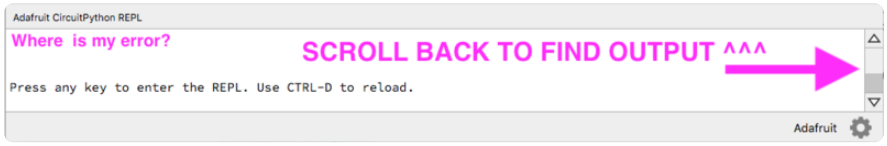
There are times when the serial console will accurately not display anything, such as, when no code is currently running, or when code with no serial output is already running before you open the console. However, if you find yourself in a situation where you feel it should be displaying something like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

```
1  Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
2  code.py output:
3  Traceback (most recent call last):
4    File "code.py", line 7
5  SyntaxError: invalid syntax
6
7
8
9  Press any key to enter the REPL. Use CTRL-D to reload.
10
```

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 " ()Acronis Managed Machine Service Mini" ().

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

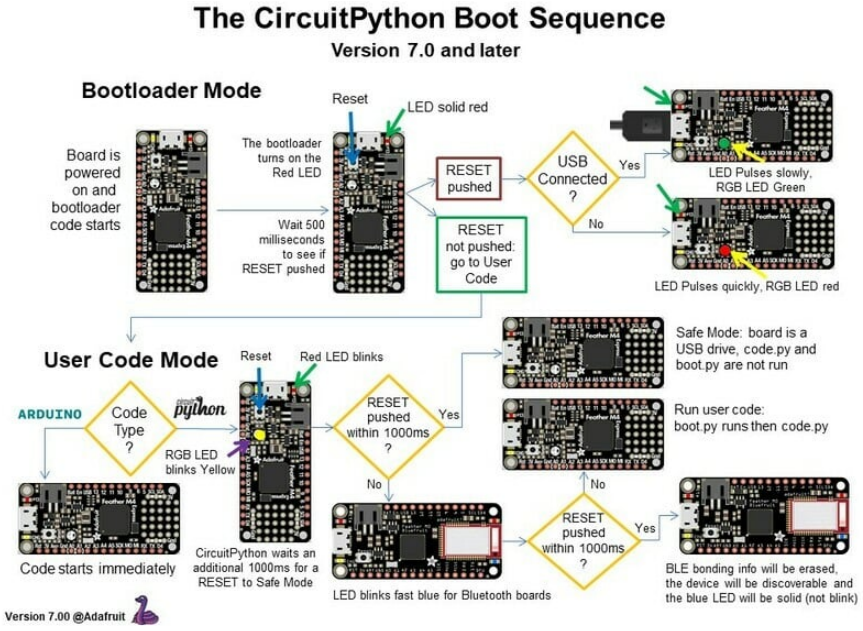
The status LED blinks were changed in CircuitPython 7.0.0 in order to save battery power and simplify the blinks. These blink patterns will occur on single color LEDs when the board does not have any RGB LEDs. Speed and blink count also vary for this reason.

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:

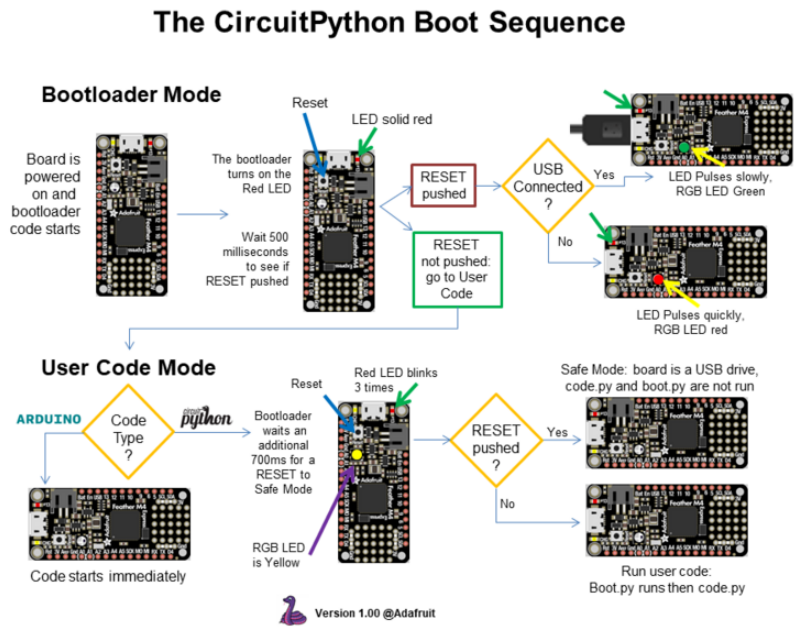
- steady **GREEN**: **code.py** (or **code.txt**, **main.py**, or **main.txt**) is running
- pulsing **GREEN**: **code.py** (etc.) has finished or does not exist
- steady **YELLOW** at start up: (4.0.0-alpha.5 and newer) CircuitPython is waiting for a reset to indicate that it should start in safe mode
- 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 ().

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 0.7 seconds. On some boards, the onboard status LED (highlighted in green above) will turn solid yellow during this

time. If you press reset during that 0.7 seconds, 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.

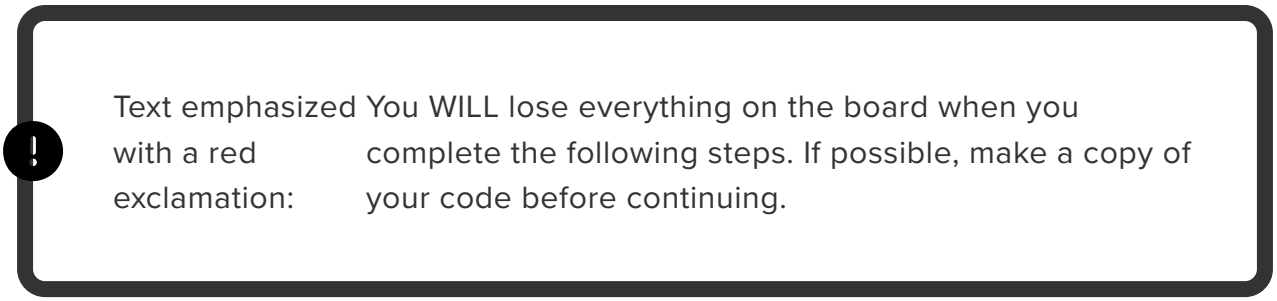
CircuitPython is in safe mode because you pressed the reset button during boot. Press again to

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.



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 () to do this.

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

```
>>> import storage
>>> 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.

!	Text emphasized with a yellow exclamation:	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:

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 () **.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 (). 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:

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 () **.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 () YYou'll also need to load your code and reinstall your libraries!

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

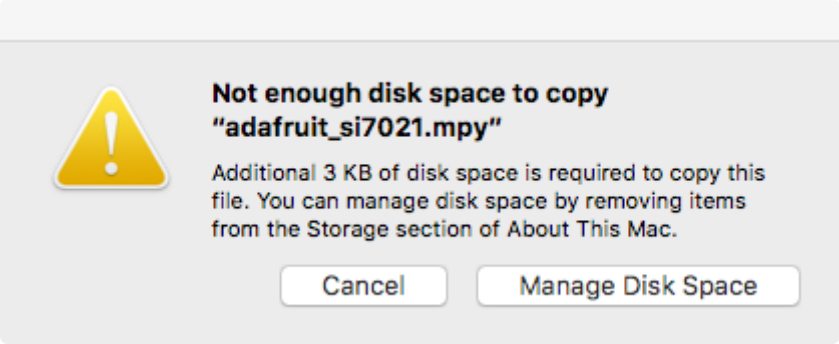
Any SAMD21-based microcontroller that does not have external flash available is considered a SAMD21 non-Express board. Non-Express boards that do **not** have a UF2 bootloader include the Feather M0 Basic Proto, Feather Adalogger, or the Arduino Zero.

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` (), 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 () 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
```

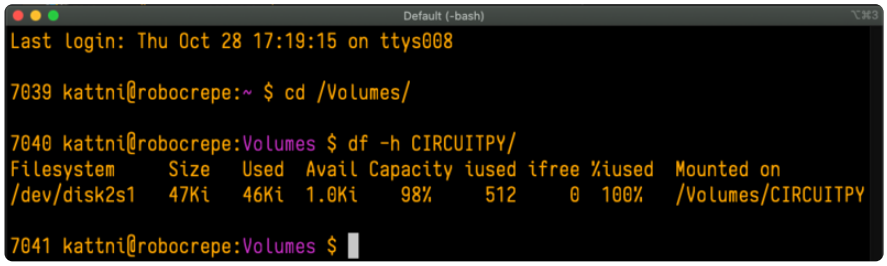
If you are copying to the **lib** folder, or another folder, make sure it exists before copying.

```
# 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/
```

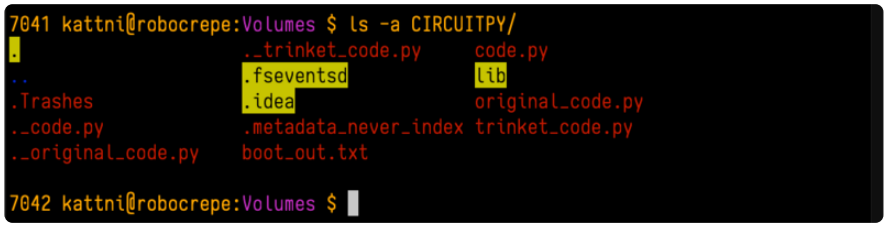
Other macOS Space-Saving Tips

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.



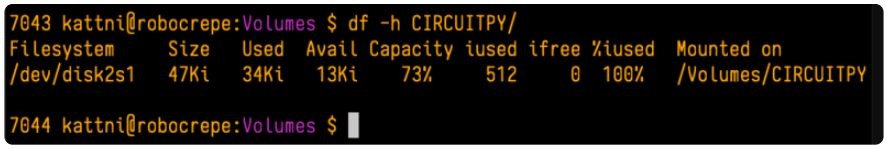
That's not very much space left! The next step is to show a list of the files currently on the **CIRCUITPY** drive, including the hidden files, using the `ls` command. You cannot use Finder to do this, you must do it via command line!



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.

For more information on safe mode and how to enter safe mode, see the Safe Mode section on this 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:

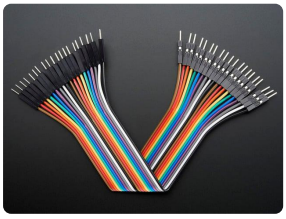


Featured

Tactile Switch Buttons (12mm square, 6mm tall) x 10 pack

Medium-sized clicky momentary switches are standard input "buttons" on electronic projects. These work best in a PCB but

https://www.adafruit.com/product/1119

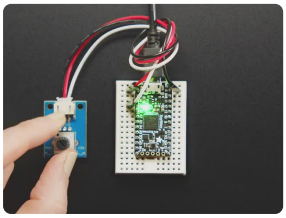


Featured

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

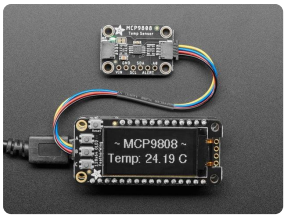


Featured

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

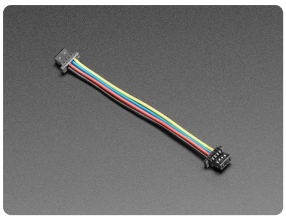


Featured

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



Featured

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

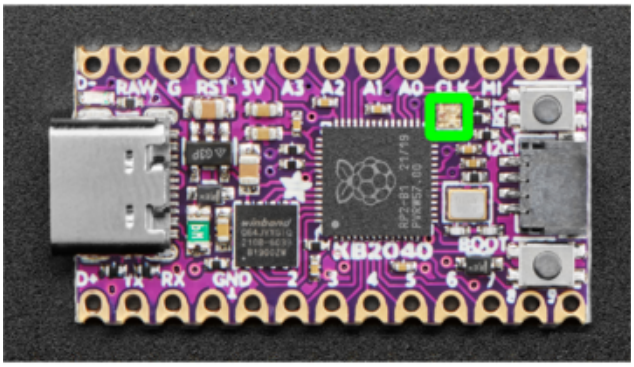
Blink

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. 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. Your board has a built-in NeoPixel LED that is great this example.

A NeoPixel is what Adafruit calls the WS281x family of addressable RGB LEDs. The built-in status LED on your board is a NeoPixel! 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!

Time to get blinky!

NeoPixel Location



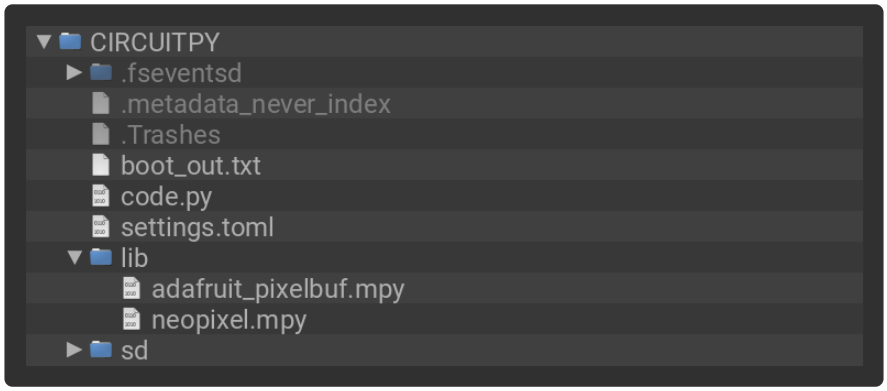
The **NeoPixel LED** is located under CLK on the board silk, by the right upper corner of the RP2040 microcontroller chip.

Blinking a NeoPixel LED

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/neopixel_blink_one_pixel/** 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:



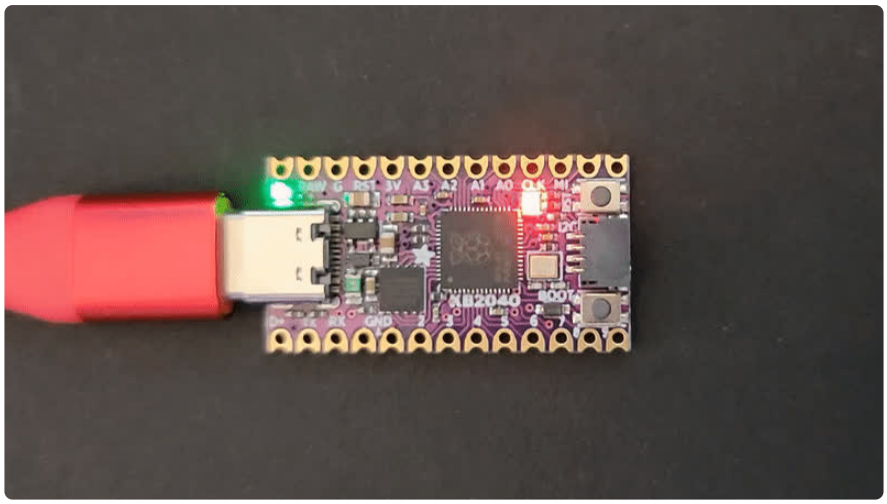

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython blink example for built-in NeoPixel LED"""
import time
import board
import neopixel

pixel = neopixel.NeoPixel(board.NEOPIXEL, 1)

while True:
    pixel.fill((255, 0, 0))
    time.sleep(0.5)
    pixel.fill((0, 0, 0))
    time.sleep(0.5)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Templates/neopixel_blink_one_pixel/code.py

The built-in NeoPixel LED begins blinking!



Text emphasized with a yellow exclamation:	If your NeoPixel does not start blinking, make sure you've copied all the necessary files and folders to the CIRCUITPY drive!

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

First you `import` three modules: `time`, `board` and `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`.

Finally, you create a `while True:` loop. This means all the code inside the loop will repeat indefinitely. Inside the loop, you "fill" the pixel with red using the RGB tuple `(255, 0, 0)`. (For more information on how RGB tuples work, see the next section!) 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 fills the pixel with "black", which turns it 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 a built-in NeoPixel LED using CircuitPython!

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.

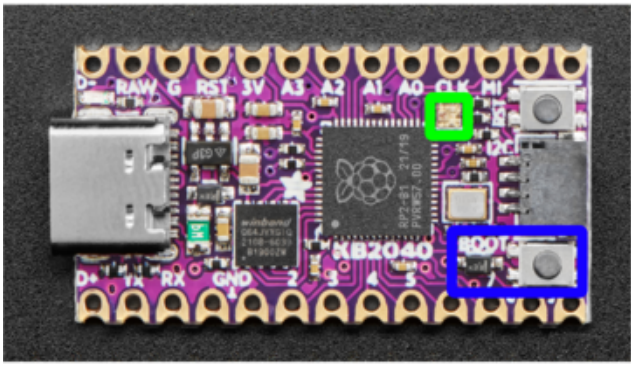
Common colors include:

- 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)

Digital Input

The CircuitPython `digitalio` module has many applications. You can easily set up a **digital input** such as a button to control the NeoPixel 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.

NeoPixel and Button



The **NeoPixel LED** is located under CLK on the board silk, by the right upper corner of the RP2040 microcontroller chip.
The **boot button**, labeled BOOT on the silk, is located in the lower right corner of the board.

Controlling the NeoPixel with a Button

To use the built-in NeoPixel on your board, you need to first install the NeoPixel library into the **lib** folder on your **CIRCUITPY** drive.

Then you need to update **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, and copy the **entire lib folder** and the **code.py file** to your **CIRCUITPY** drive.

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: Unlicense
"""
CircuitPython Digital Input example - Blinking a built-in NeoPixel LED using a button switch.
"""
import board
import digitalio
import neopixel

pixel = neopixel.NeoPixel(board.NEOPIXEL, 1)

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

while True:
    if not button.value:
        pixel.fill((255, 0, 0))
    else:
        pixel.fill((0, 0, 0))
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/Adafruit_KB2040/NeoPixel_Blink/code.py

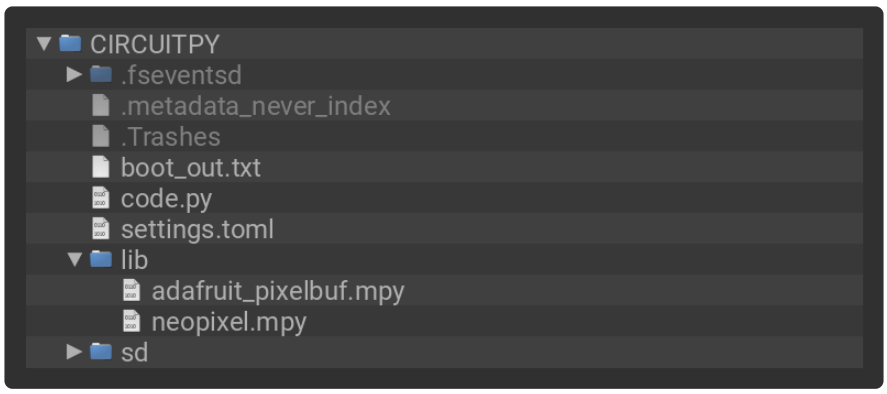
Your **CIRCUITPY** drive contents should resemble the image below.

You should have in **Root Folder** / of the **CIRCUITPY** drive:

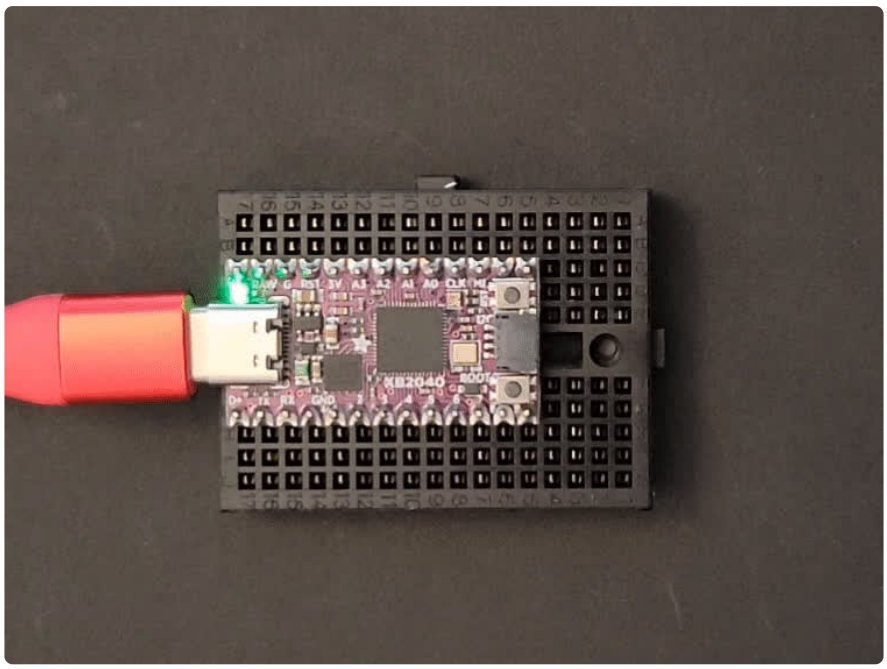
- **code.py**

And in the **lib** folder on your **CIRCUITPY** drive:

- **adafruit_pixelbuf.mpy**
- **neopixel.mpy**



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



Text emphasized with a yellow exclamation:

If your NeoPixel does not light up when you press the button, make sure you've copied all the necessary files and folders to the CIRCUITPY drive!

First you `import` two modules, `board` and `digitalio`, and one library, `neopixel`. This makes these modules available for use in your code.

Next, you set up the NeoPixel. 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 NeoPixels, `1`. You save this object to the variable `pixel`.

Then, you create a `digitalio.DigitalInOut()` object, provide it the button pin using the `board` module, and save it to the variable `button`. You tell the pin to act as an `INPUT` and provide a pull up.

Inside the loop, you check to see if the button is pressed, and if so, turn the NeoPixel red. Otherwise the NeoPixel is off.

That's all there is to controlling a NeoPixel LED with a button switch!

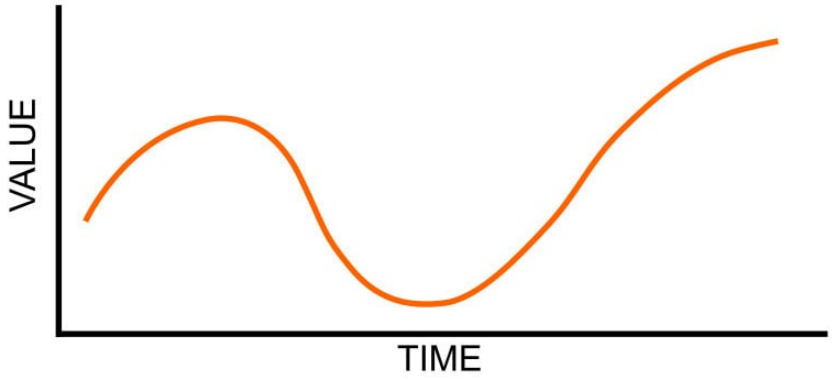
Analog In

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.



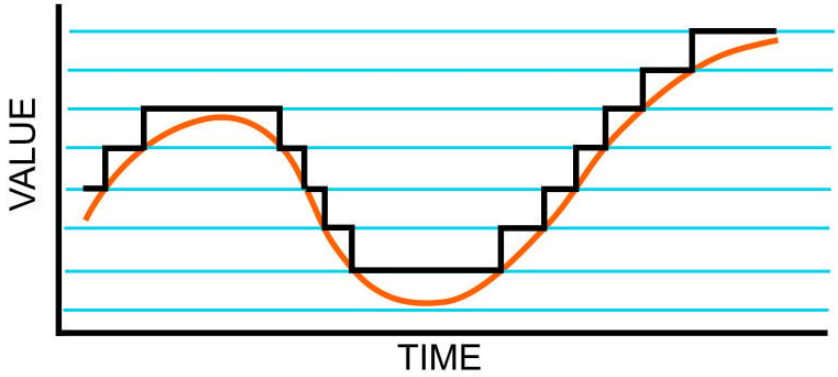
Analog signals are continuous values which means they can be an infinite number of different voltages. Think of analog signals like a floating point or fractional number, they can smoothly transiting to any in-between value like 1.8V, 1.81V, 1.801V, 1.8001V, 1.80001V and so forth to infinity.

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.

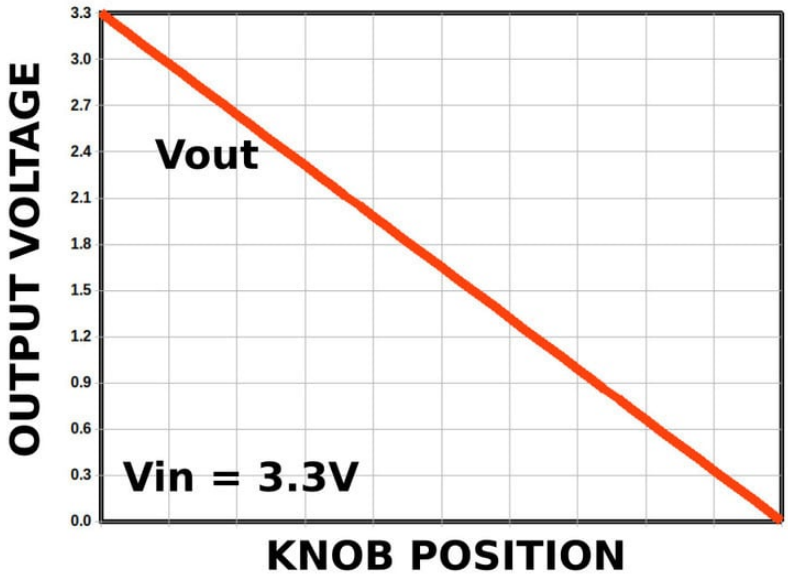


Once that analog signal has been converted by the ADC, the microcontroller can use those digital values any way you like!

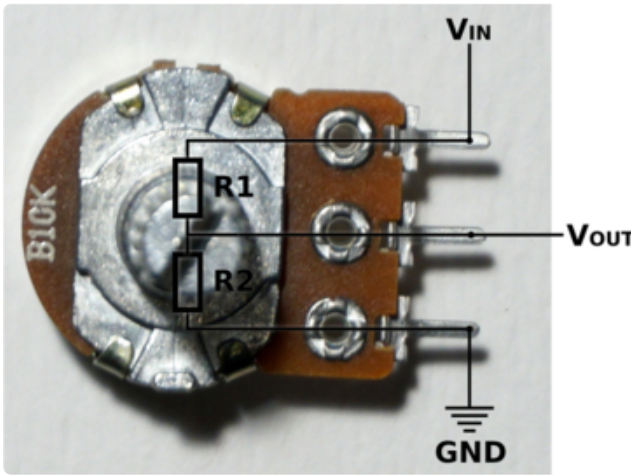
Potentiometers

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:

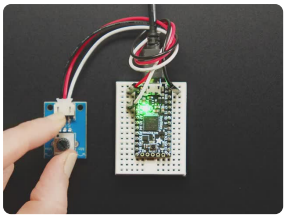


Connect **one outside pin** to **ground**
Connect **the other outside pin** to **voltage in (e.g. 3.3V)**
Connect **the middle pin** to **an analog pin (e.g. A0)**

Hardware

In addition to your microcontroller board, you will need the following hardware to follow along with this example.

Potentiometer



Featured

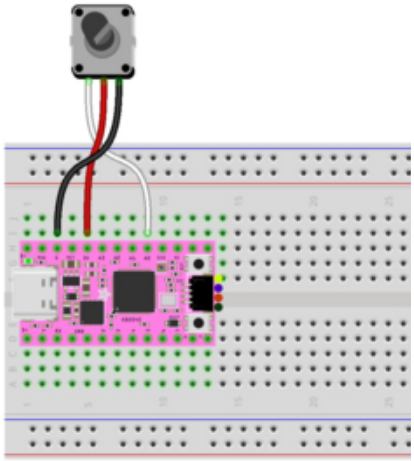
[STEMMA Wired Potentiometer Breakout Board - 10K ohm Linear]

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 **KB2040 A0**
Potentiometer center pin (red wire) to **KB2040 3.3V**
Potentiometer right pin (black wire) to **KB2040 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 () 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)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Templates/analog_pin_values/code.py

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 () 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)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Templates/analog_voltage_values/code.py

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.

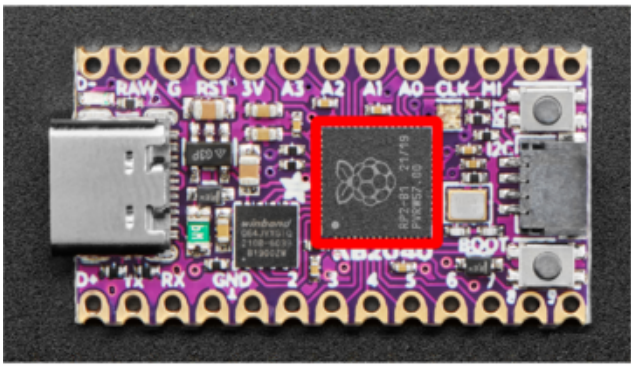
That's all there is to reading analog voltage values using CircuitPython!

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 large grey square towards the center of the board is the **RP2040 microcontroller**.

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 (), 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.

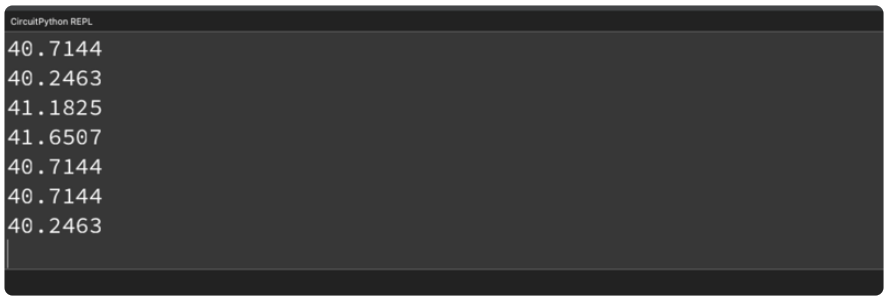
Your **CIRCUITPY** drive should now look similar to the following image:




```
# 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)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Templates/cpu_temperature/code.py



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.

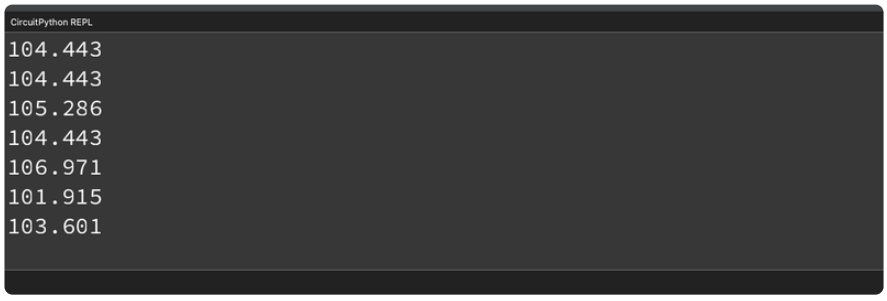
Your **CIRCUITPY** drive should now look similar to the following image:




```
# 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)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Templates/cpu_temperature_f/code.py



The CPU temperature in Fahrenheit is printed out to the serial console!

That's all there is to reading the CPU temperature using CircuitPython!

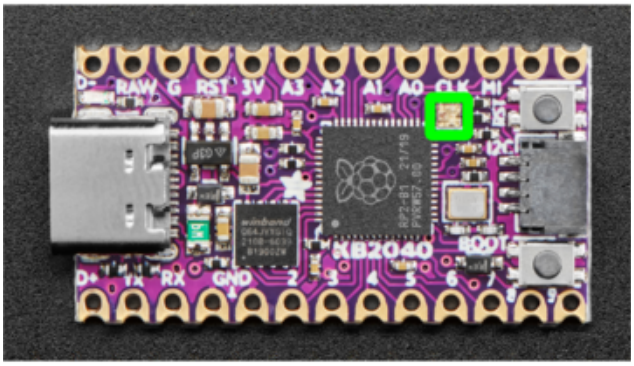
NeoPixel LED

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** is located under CLK on the board silk, by the right upper corner of the RP2040 microcontroller chip.

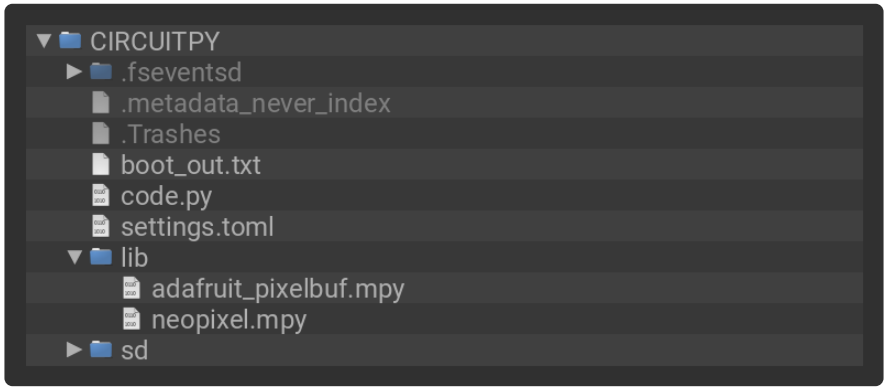
NeoPixel Color and Brightness

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

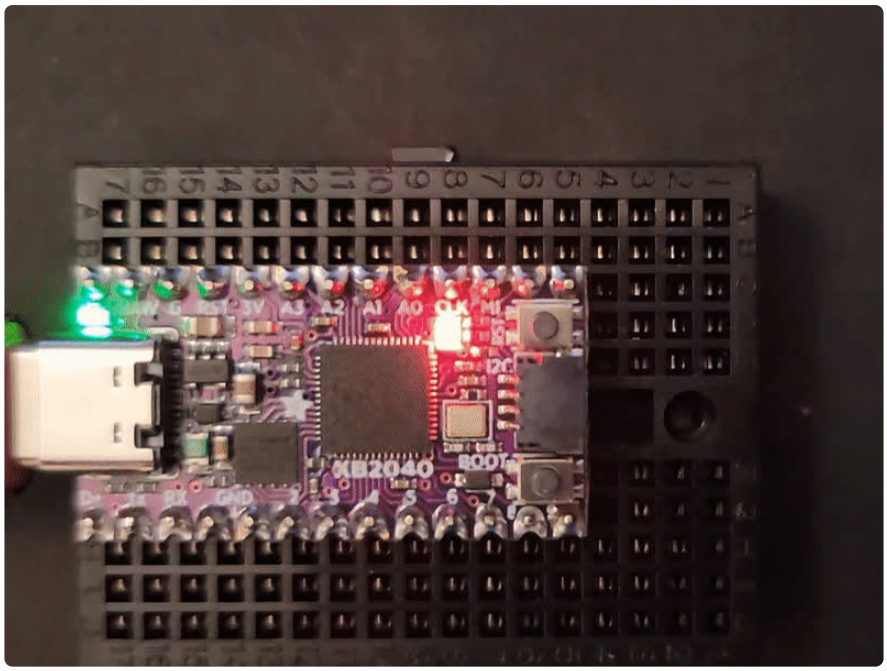
pixel = neopixel.NeoPixel(board.NEOPIXEL, 1)

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)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Templates/status_led_one_neopixel_rgb/code.py

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)`

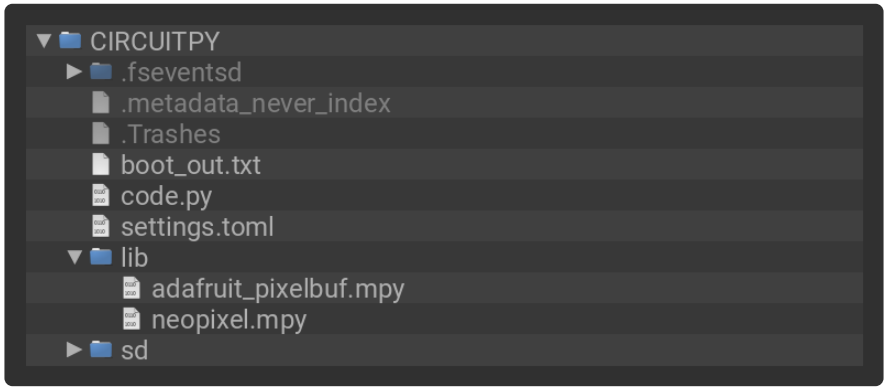
NeoPixel Rainbow

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

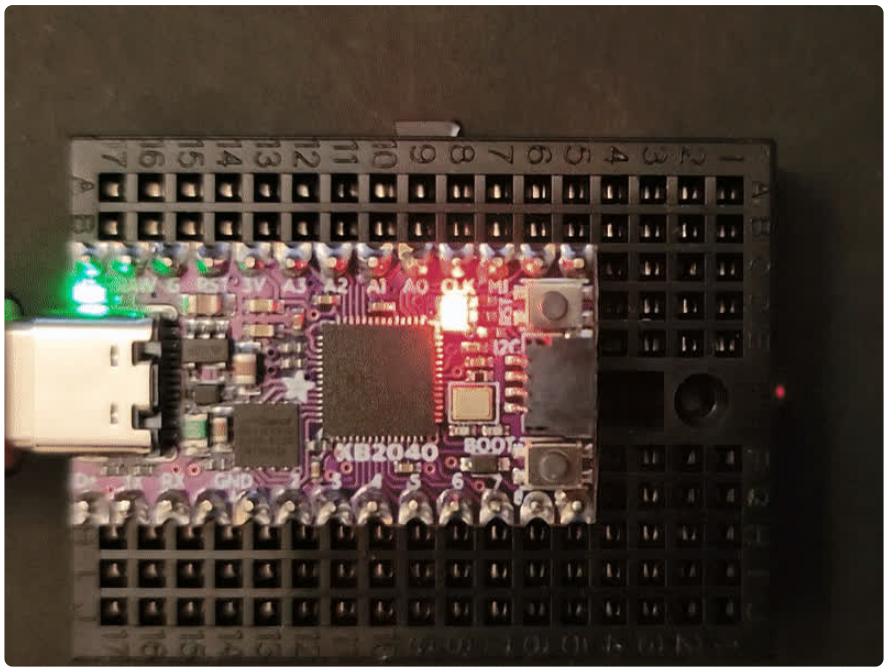
pixel = neopixel.NeoPixel(board.NEOPIXEL, 1)
pixel.brightness = 0.3


def rainbow(delay):
    for color_value in range(255):
        pixel[0] = colorwheel(color_value)
        time.sleep(delay)


while True:
    rainbow(0.02)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Templates/status_led_one_neopixel_rainbow/code.py

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.

Text emphasized with a red exclamation:	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

Text emphasized with a blue exclamation:	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: Unlicense
"""
CircuitPython Essentials Storage CP Filesystem boot.py file
"""
import board
import digitalio
import storage

```

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

# If the OBJECT_NAME is connected to ground, the filesystem is writable by CircuitPython
storage.remount("/", readonly=button.value)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/Adafruit_KB2040/Storage/boot.py

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.

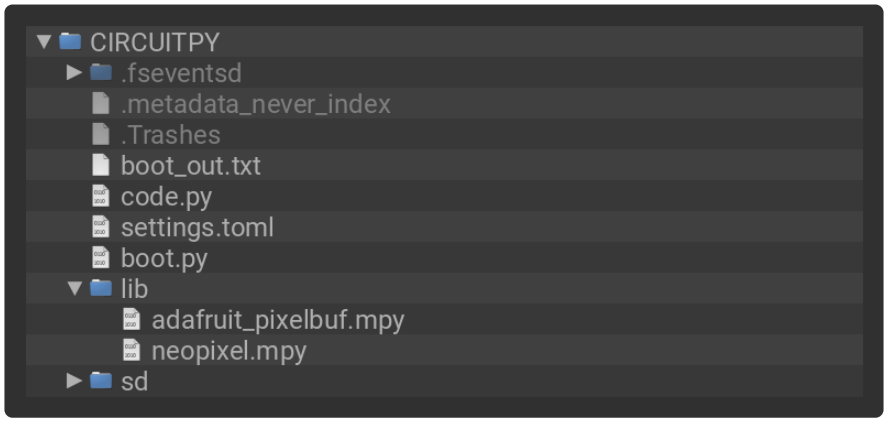
When the button is pressed, it returns `False`. The `readonly` argument in **boot.py** is set to the `value` of the button. When the `value=True`, the **CIRCUITPY** drive is read-only to CircuitPython (and writable by your computer). When the `value=False`, the **CIRCUITPY** drive is writable by CircuitPython (and read-only by your computer).

The code.py File

Save the following as **code.py** on your **CIRCUITPY** drive.

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 **Adafruit_KB2040/Storage/** 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: Unlicense
"""
CircuitPython Essentials Storage CP Filesystem code.py file
```

```
"""
import time
import board
import microcontroller
import neopixel

pixel = neopixel.NeoPixel(board.NEOPIXEL, 1)

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 NeoPixel on every write...
            pixel.fill((255, 0, 0))
            time.sleep(1)  # ...for one second.
            pixel.fill((0, 0, 0))  # 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 NeoPixel every half second.
    if e.args[0] == 28:  # If the file system is full...
        delay = 0.15  # ...blink the NeoPixel every 0.15 seconds!
    while True:
        pixel.fill((255, 0, 0))
        time.sleep(delay)
        pixel.fill((0, 0, 0))
        time.sleep(delay)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/Adafruit_KB2040/Storage/code.py

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

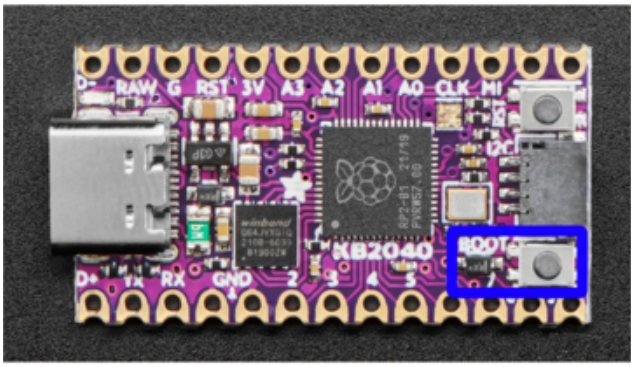
second, and then turn it off for the next nine seconds. Essentially, you blink the LED for one second every time the temperature is logged to the file which happens every ten seconds.

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 button is pressed when the board is powered on and boot.py is run. To begin logging the temperature, you must press the button.



The **boot button**, labeled BOOT on the silk, is in the lower right corner of the board.

While holding down the button, 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 red 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 button is pressed, 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.

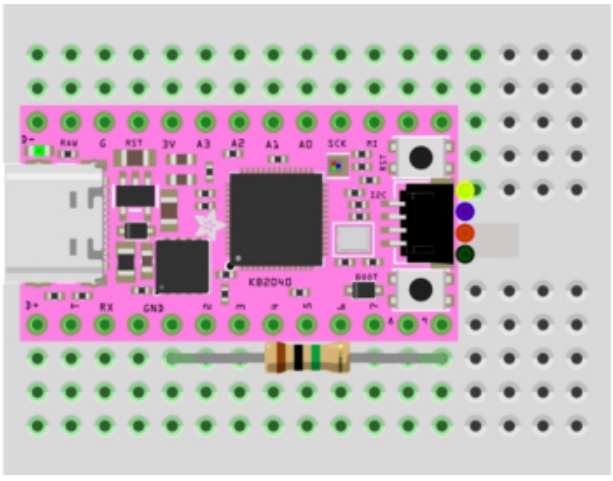
This section first covers using the `touchio` module to read touches on one pin. You'll learn how to setup the pin in your program, and read the touch status. Then, you'll learn how to read touches on multiple pins in a single example. Time to get started!

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.



1MΩ resistor from **KB2040 pin 9** to **KB2040 GND**

Reading Touch on the Pin

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 **Adafruit_KB2040/Capacitive_Touch_One_Pin/** 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 ().

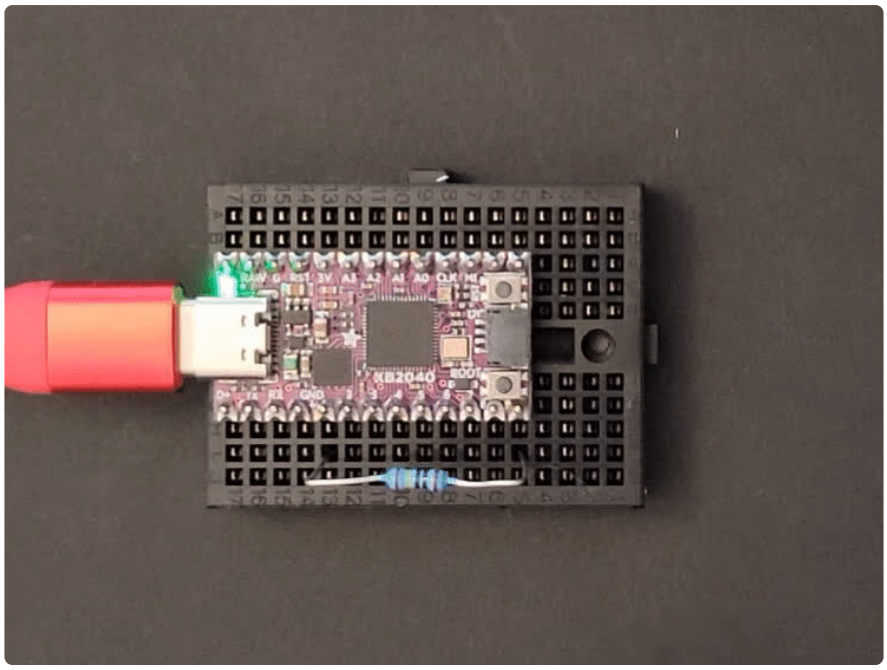
```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: Unlicense
"""
CircuitPython Capacitive Touch Pin Example - Print to the serial console when one pin is touched.
"""
import time
import board
import touchio

touch = touchio.TouchIn(board.D9)

while True:
    if touch.value:
        print("Pin touched!")
    time.sleep(0.1)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/Adafruit_KB2040/Capacitive_Touch_One_Pin/code.py

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.

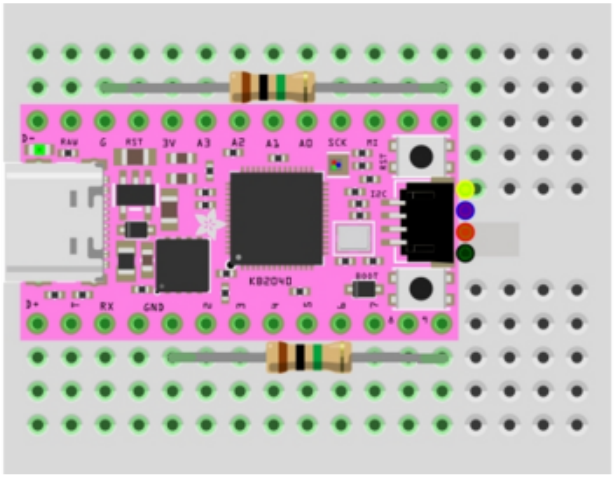
Inside the loop, you check to see if the pin is touched. If so, you print to the serial console. Finally, you include a `time.sleep()` to slow it down a bit so the output is readable.

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



1MΩ resistor from **KB2040 pin 9** to **KB2040 GND**
1MΩ resistor from **KB2040 pin 10** to **KB2040 GND**

Reading Touch on the Pins

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 **Adafruit_KB2040/Capacitive_Touch_Two_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 ().

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: Unlicense
"""
CircuitPython Capacitive Two Touch Pin Example - Print to the serial console when a pin is touched.
"""
import time
import board
import touchio
```

```
9
10  touch_one = touchio.TouchIn(board.D9)
11  touch_two = touchio.TouchIn(board.D10)
12
13  while True:
14      if touch_one.value:
15          print("Pin one touched!")
16      if touch_two.value:
17          print("Pin two touched!")
18      time.sleep(0.1)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/Adafruit_KB2040/Capacitive_Touch_Two_pins/code.py

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

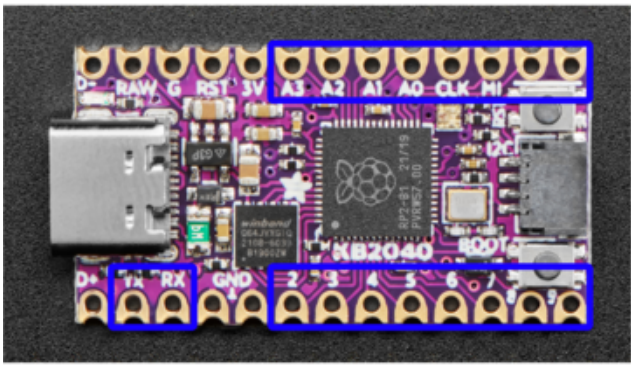

def get_pin_names():
    """
    Gets all unique pin names available in the board module, excluding a defined list.
    This list is not exhaustive, and depending on the order of the pins in the CircuitPython
    pin definition, some of the pins in the list may still show up in the script results.
    """
    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).
```

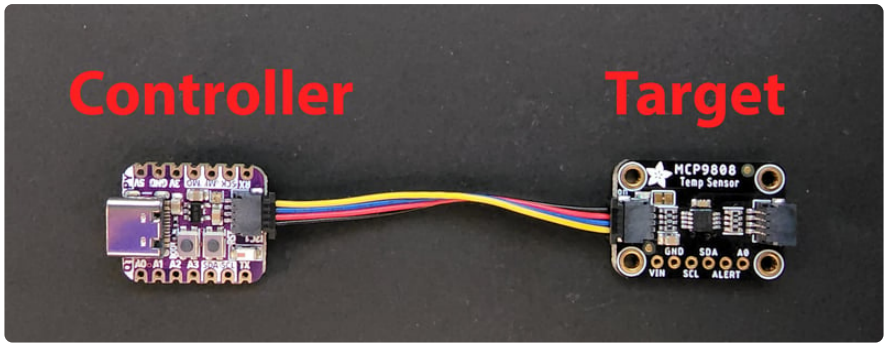
https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Templates/cap_touch_pin_script/code.py

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.



A0 - CircuitPython `board.A0`. Arduino `26`.
A1 - CircuitPython `board.A1`. Arduino `27`.
A2 - CircuitPython `board.A2`. Arduino `28`.
A3 - CircuitPython `board.A3`. Arduino `29`.
TX - CircuitPython `board.TX`. Arduino `0`.
RX - CircuitPython `board.RX`. Arduino `1`.
D10 - CircuitPython `board.D10`. Arduino `10`.
D2 - CircuitPython `board.D2`. Arduino `2`.
D3 - CircuitPython `board.D3`. Arduino `3`.
D4 - CircuitPython `board.D4`. Arduino `4`.
D5 - CircuitPython `board.D5`. Arduino `5`.
D6 - CircuitPython `board.D6`. Arduino `6`.
D7 - CircuitPython `board.D7`. Arduino `7`.
D8 - CircuitPython `board.D8`. Arduino `8`.
D9 - CircuitPython `board.D9`. Arduino `9`.
MISO - CircuitPython `board.MISO`. Arduino `20`.
MOSI - CircuitPython `board.MOSI`. Arduino `19`.
SCK - CircuitPython `board.SCK`. Arduino `18`.

I2C



The **I2C**, or inter-integrated circuit (), 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 ().

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.

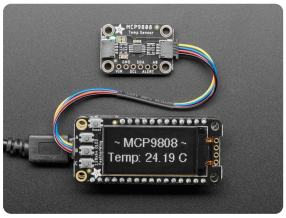
I2C and CircuitPython

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 (). (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.

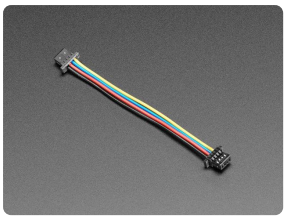


Featured

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



Featured

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...

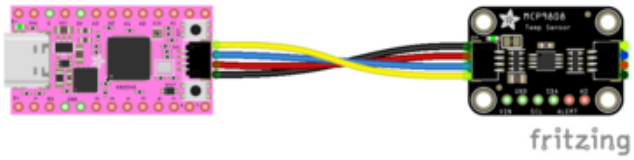
https://www.adafruit.com/product/4399

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()
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Templates/i2c_scan/code.py

```
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)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Templates/i2c_mcp9808/code.py

```
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
```

Text emphasized with a blue exclamation:	For the KB2040, you'll need to change the I2C setup to the commented out setup included in the code above.

The KB2040 STEMMA QT connector is available on `board.STEMMA_I2C()`. Comment out the current `i2c` setup line, and uncomment the the `i2c = board.STEMMA_I2C()` line to use with your board's STEMMA QT connector.

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

```

```
8
9   def is_hardware_i2c(scl, sda):
10      try:
11          p = busio.I2C(scl, sda)
12          p.deinit()
13          return True
14      except ValueError:
15          return False
16
17
18  def get_unique_pins():
19      exclude = [
20          getattr(board, p)
21          for p in [
22              # This is not an exhaustive list of unexposed pins. Your results
23              # may include other pins that you cannot easily connect to.
24              "NEOPIXEL",
25              "DOTSTAR_CLOCK",
26              "DOTSTAR_DATA",
27              "APA102_SCK",
28              "APA102_MOSI",
29              "LED",
30              "SWITCH",
31              "BUTTON",
32              "ACCELEROMETER_INTERRUPT",
33              "VOLTAGE_MONITOR",
34              "MICROPHONE_CLOCK",
35              "MICROPHONE_DATA",
36              "RFM_RST",
37              "RFM_CS",
38              "RFM_IO0",
39              "RFM_IO1",
40              "RFM_IO2",
41              "RFM_IO3",
42              "RFM_IO4",
43              "RFM_IO5",
44              "TFT_I2C_POWER",
45              "NEOPIXEL_POWER",
46          ]
47          if p in dir(board)
48      ]
49      pins = [
50          pin
51          for pin in [getattr(board, p) for p in dir(board)]
52          if isinstance(pin, Pin) and pin not in exclude
53      ]
54      unique = []
55      for p in pins:
56          if p not in unique:
57              unique.append(p)
58      return unique
59
60
61  for scl_pin in get_unique_pins():
62      for sda_pin in get_unique_pins():
63          if scl_pin is sda_pin:
```

```
64              continue
65          if is_hardware_i2c(scl_pin, sda_pin):
66              print("SCL pin:", scl_pin, "\t SDA pin:", sda_pin)
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/CircuitPython_Templates/i2c_find_pins/code.py

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.

Text emphasized with a green exclamation:	The output for the KB2040 is extremely long! The screenshot shows only the beginning. Run the script yourself to see the full output!



Text emphasized with a blue exclamation:	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.

Arduino IDE Setup

The Arduino Philhower core () 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.

Download and install it to your computer.

Once installed, open the Arduino IDE.

Adding the Philhower Board Manager URL

In the Arduino IDE, navigate to the **Preferences** window. You can access it through **File** > **Preferences** on Windows or Linux, or **Arduino** > **Preferences** on OS X.

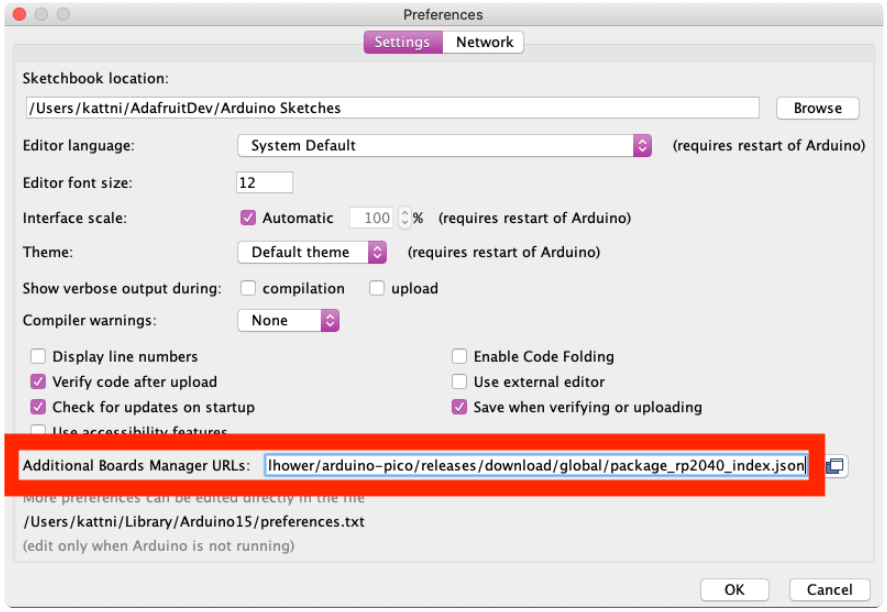
The **Preferences** window will open.

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
```

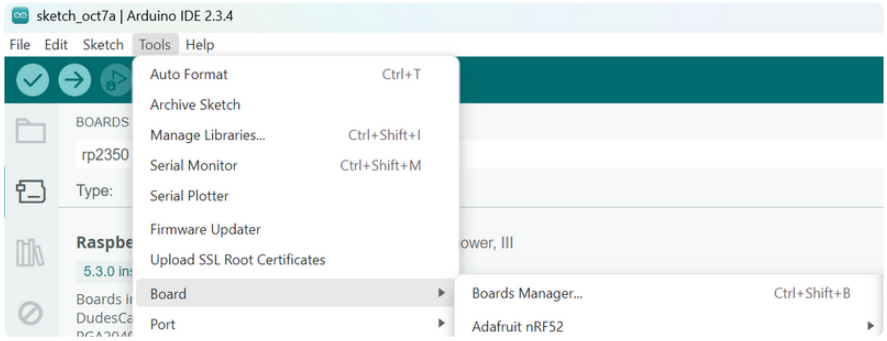
Add the URL to the the **Additional Boards Manager URLs** field (highlighted in red below).



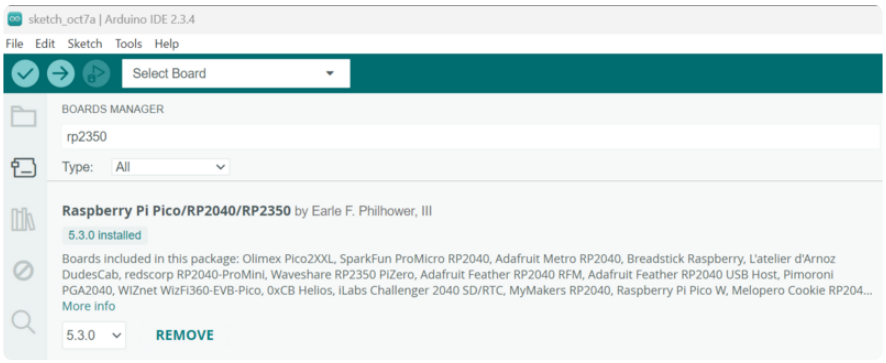
Click **OK** to save and close **Preferences**.

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/RP2350 by Earle F Philhower, III** entry. Click **Install** to install it. If it was previously installed, look to make sure you have the latest version.



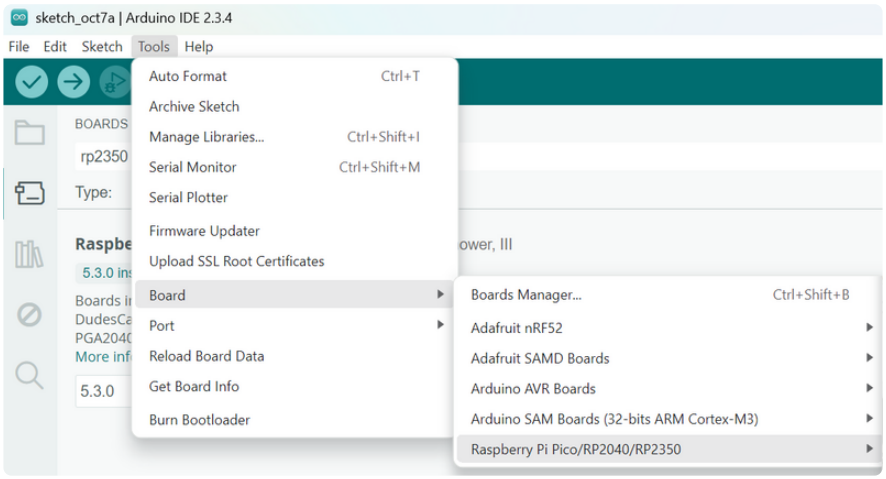
Text emphasized with a yellow exclamation:

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

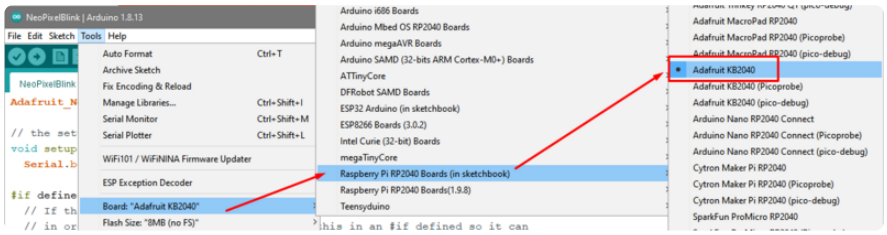
In the **Tools** > **Boards** menu, you should now see **Raspberry Pi RP2040 Boards** (possibly followed by a version number).

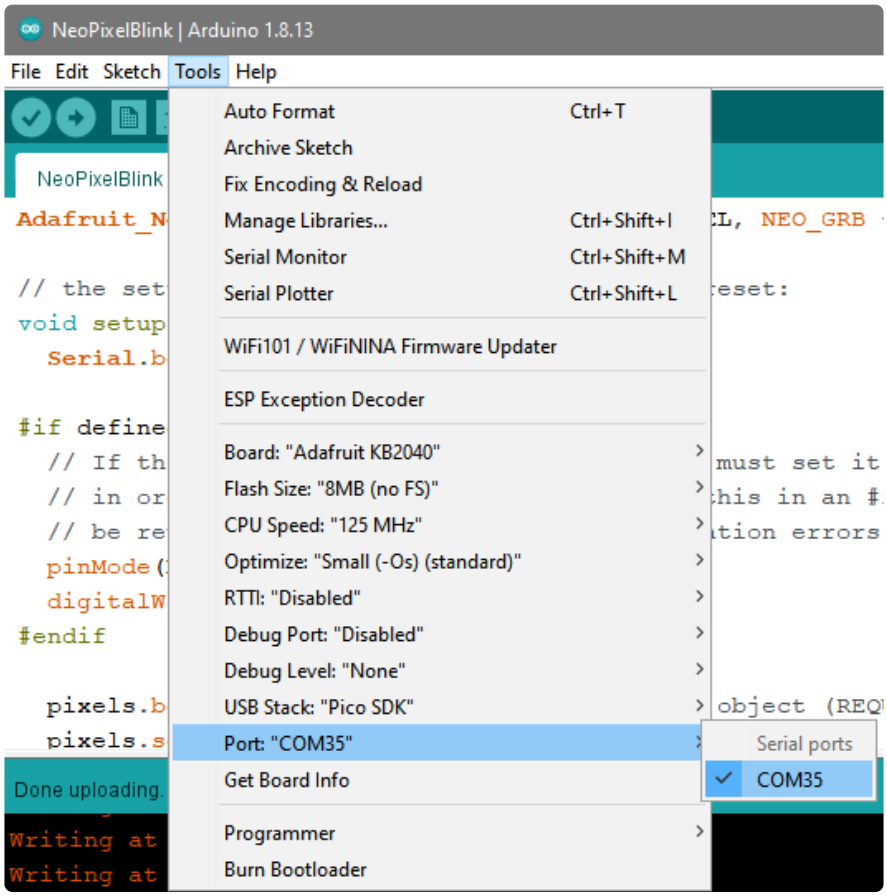


Navigate to the **Raspberry Pi Pico RP2040/RP2350** menu. You will see the available boards listed.

Text emphasized with a blue exclamation:

Look for Adafruit KB2040 in the RP2040 Boards collection!





Text emphasized with a red exclamation:

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.

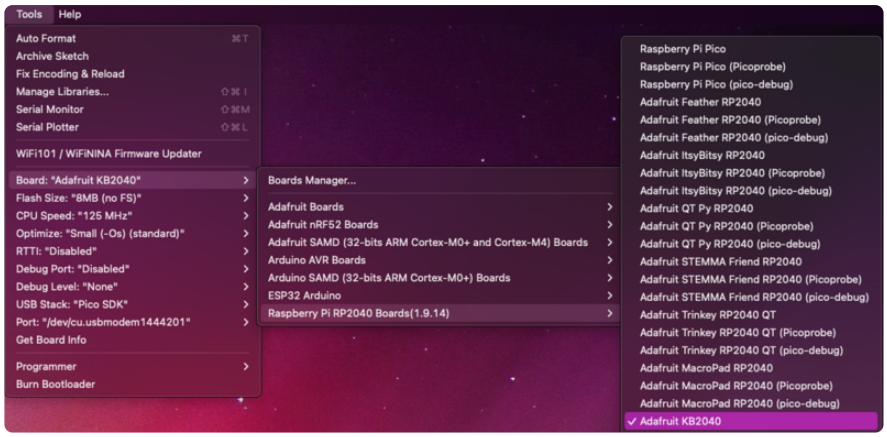
To find the Arduino pin name, check the PrettyPins diagram found on the Pinouts page (). Each GPIO pin in the diagram has a **GPIOx** pin name listed, where **x** is the pin number. The Arduino pin name is the number following **GPIO**. For example, **GPIO1** would be Arduino pin `1`.

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.

Once the menu has expanded, you will see three different versions the KB2040 available: **Adafruit KB2040**, **Adafruit KB2040 (Picoprobe)**, and **Adafruit KB2040 (pico-debug)**. Unless you are specifically familiar with the other two, always choose **Adafruit KB2040**.

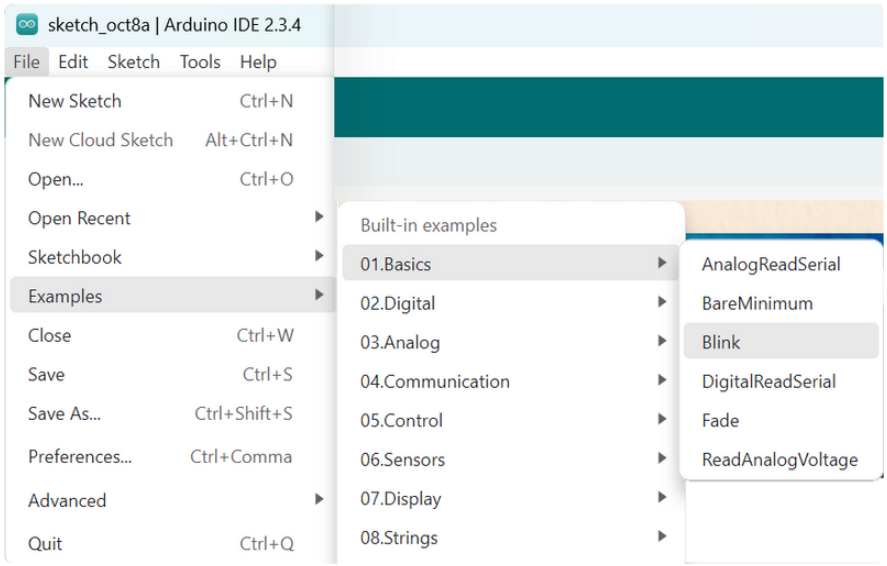
Choose **Adafruit KB2040** from the menu.



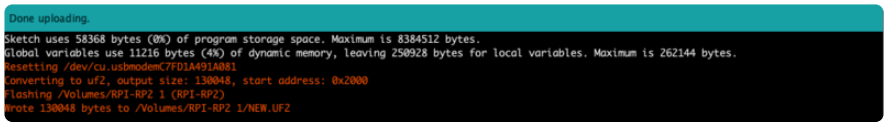
Load the Blink Sketch

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).

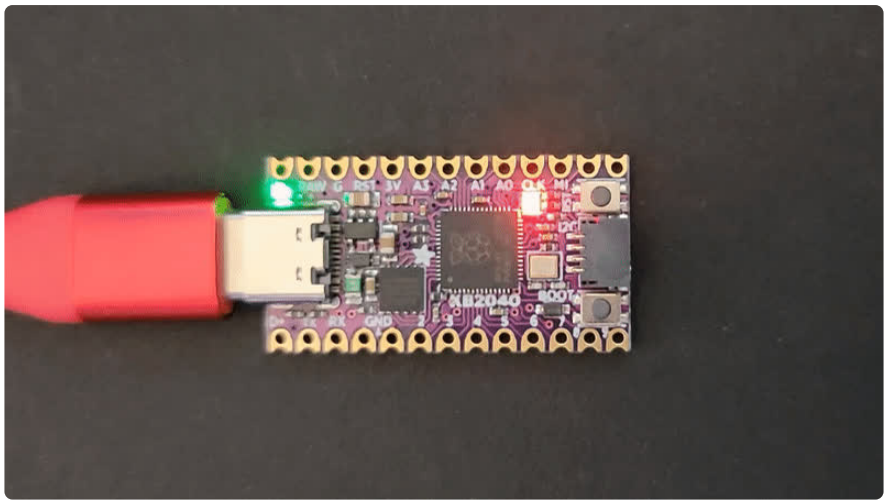
Now, click Upload on your sketch to try again.

Arduino NeoPixel 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!

Now traditionally you would use an onboard LED to make a blink occur. **However, this board does not have an onboard single-color LED**, so we will 'mimic' the same blink sketch but instead of using a digital output pin, we will use NeoPixel support to blink the onboard RGB LED!



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!

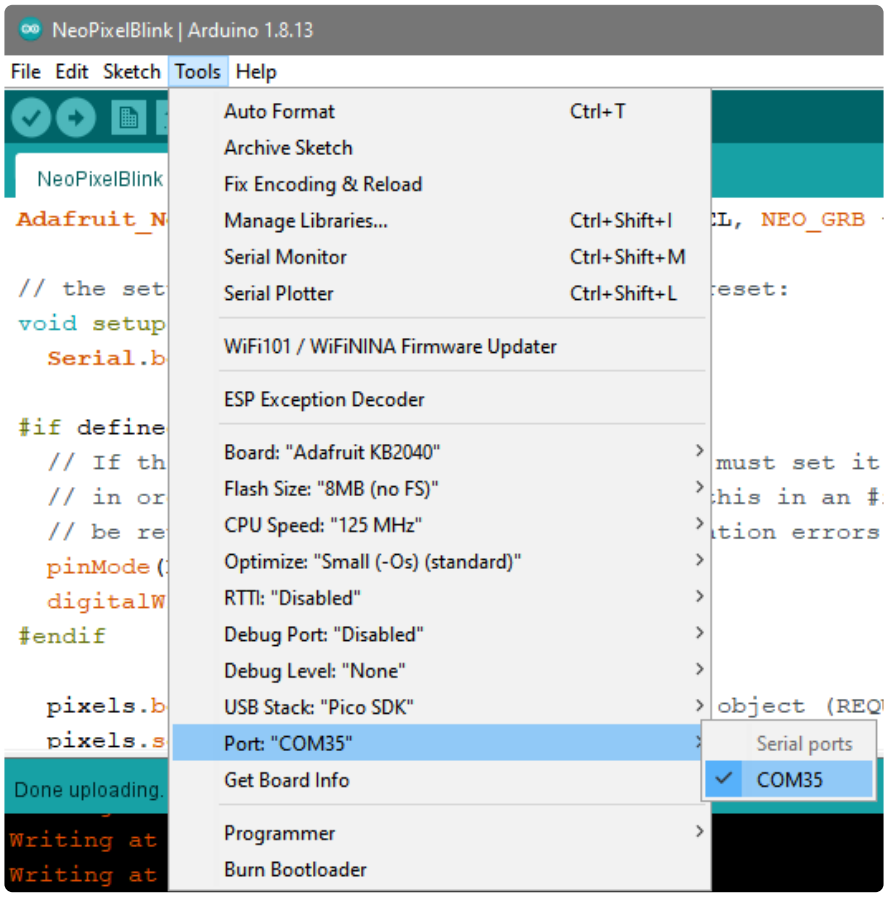
Text emphasized with a yellow exclamation:

The KB2040 has a green power LED next to the USB port, so you should make sure that it's lit up when plugged into USB. No drivers are required for the KB2040

Start up Arduino IDE and Select Board/Port

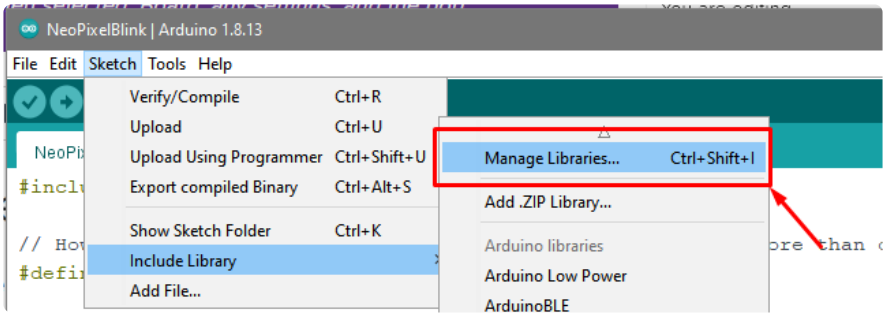
OK now you are prepared! Open the Arduino IDE on your computer. Now you have to tell the IDE what board you are using, and how you want to connect to it.

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.

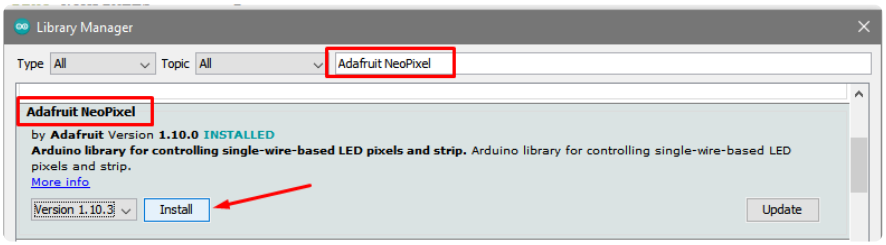


Install NeoPixel Library

Despite their popularity, NeoPixel RGB LEDs are not supported 'out of the box' in Arduino IDE! You will need to add support by installing the library. Good news it is very easy to do it. Go to the **Library Manager** here

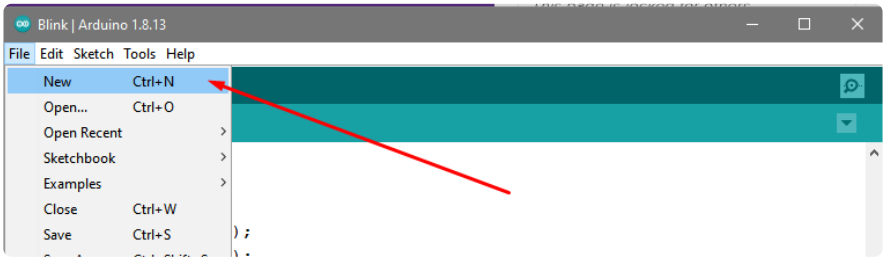


Search for and install the **Adafruit NeoPixel** library. It might not be first in the list so make sure you get the name matched up right!



New NeoPixel 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:

```
#include <Adafruit_NeoPixel.h>

```

```
// How many internal neopixels do we have? some boards have more than one!
#define NUMPIXELS        1

Adafruit_NeoPixel pixels(NUMPIXELS, PIN_NEOPIXEL, NEO_GRB + NEO_KHZ800);

// the setup routine runs once when you press reset:
void setup() {
  Serial.begin(115200);

#if defined(NEOPIXEL_POWER)
  // If this board has a power control pin, we must set it to output and high
  // in order to enable the NeoPixels. We put this in an #if defined so it can
  // be reused for other boards without compilation errors
  pinMode(NEOPIXEL_POWER, OUTPUT);
  digitalWrite(NEOPIXEL_POWER, HIGH);
#endif

  pixels.begin(); // INITIALIZE NeoPixel strip object (REQUIRED)
  pixels.setBrightness(20); // not so bright
}

// the loop routine runs over and over again forever:
void loop() {
  // say hi
  Serial.println("Hello!");

  // set color to red
  pixels.fill(0xFF0000);
  pixels.show();
  delay(500); // wait half a second

  // turn off
  pixels.fill(0x000000);
  pixels.show();
  delay(500); // wait half a second
}
```



Text emphasized with a blue exclamation:

Note that in this example, we are not only blinking the NeoPixel 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 `PIN_NEOPIXEL` rather than a number. That's because each board could have the built in NeoPixels on a different pin and this makes the code a little more portable!

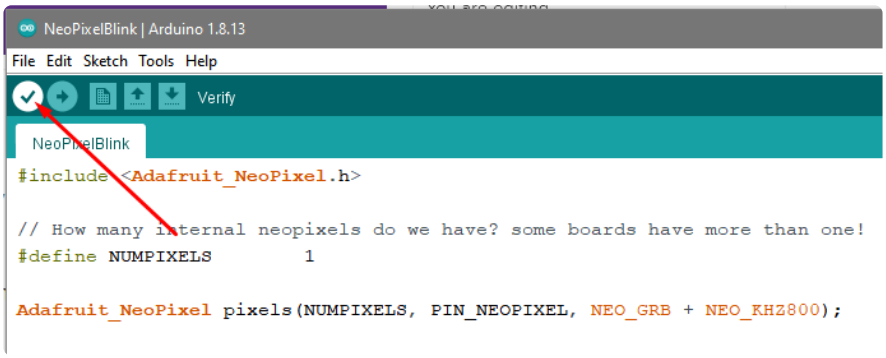
Text emphasized with a green exclamation:

On the KB2040 the NeoPixel is on GPIO pin #17

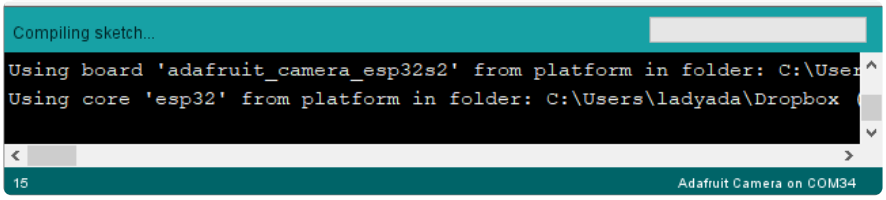
Verify (Compile) Sketch

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

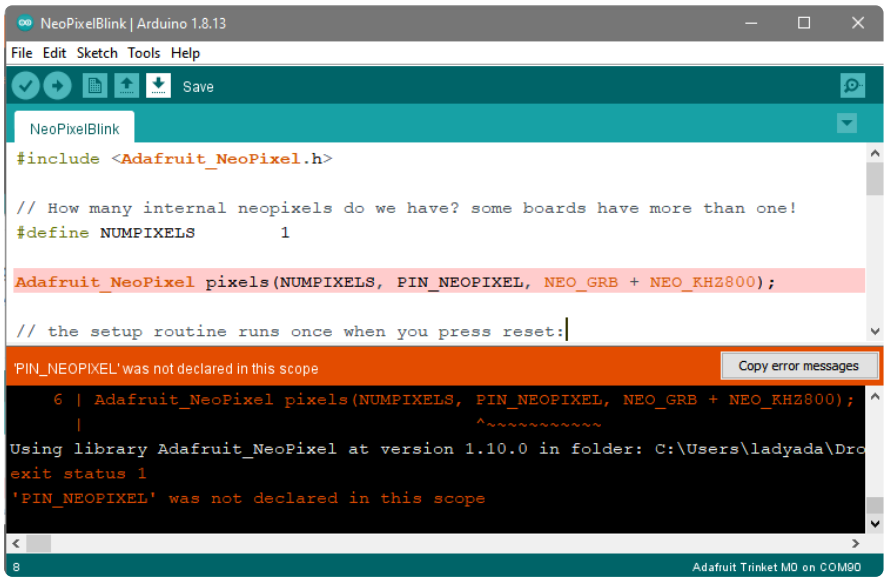


During verification/compilation, the computer will do a bunch of work to collect all the libraries and code and the results will appear in the bottom window of the IDE

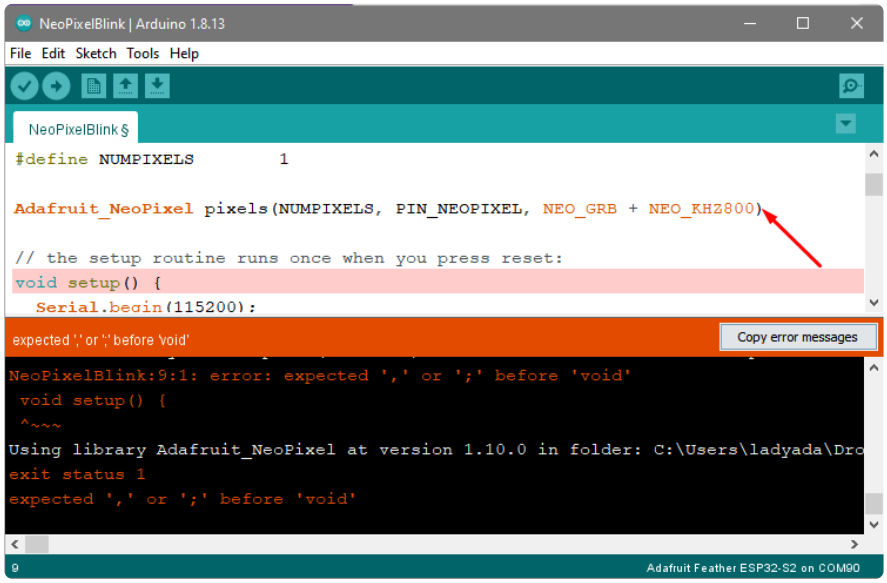


If something went wrong with compilation, you will get red warning/error text in the bottom window letting you know what the error was. It will also highlight the line with an error

For example, here I had the wrong board selected - and the selected board does not have a built in NeoPixel pin defined!



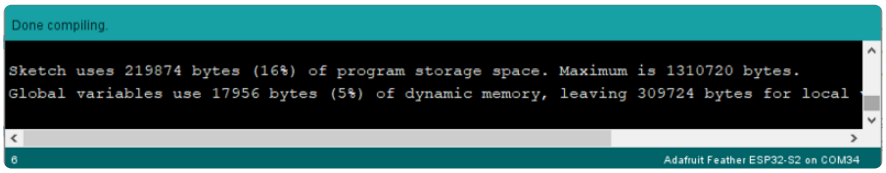
Here's another common error, in my haste I forgot to add a `;` at the end of a line. The compiler warns me that it's looking for one - note that the error is actually a few lines up!



Text emphasized with a blue exclamation:

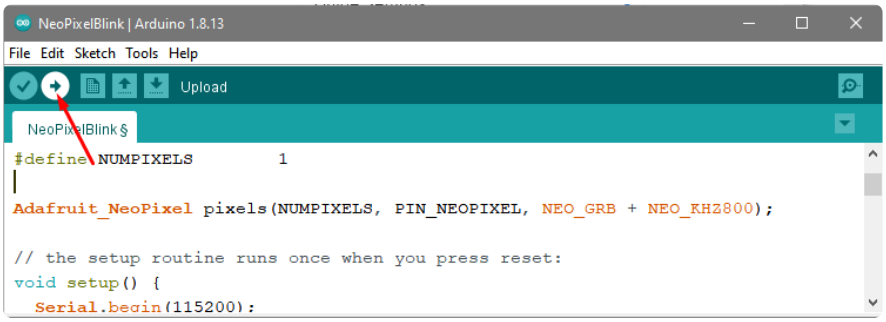
Turning on detailed compilation warnings and output can be very helpful sometimes - Its in Preferences under "Show Verbose Output During:" and check the Compilation button. If you ever need to get help from others, be sure to do this and then provide all the text that is output. It can assist in nailing 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

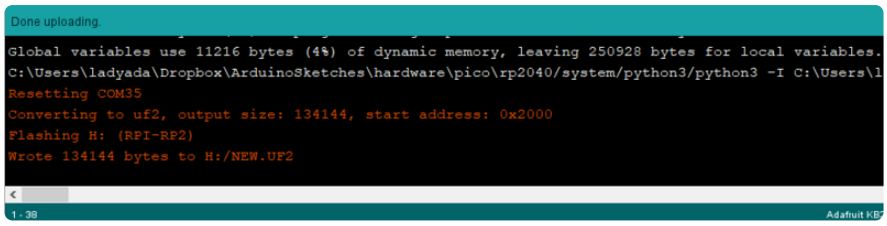
Once the code is verified/compiling cleanly you can upload it to your board. Click the **Upload** button



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.

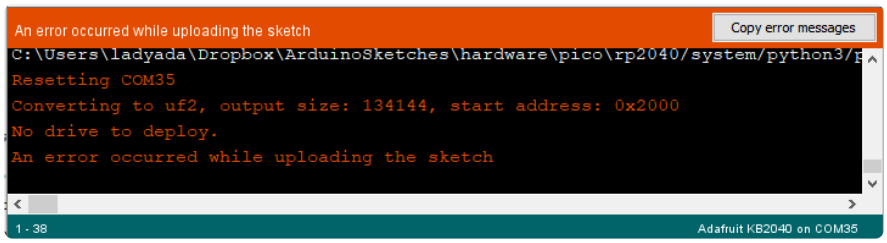
This is actually one of the hardest parts for beginners because it's where a lot of things can go wrong.

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:

```
Converting to uf2, output size: 134144, start address: 0x2000
No drive to deploy.
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.

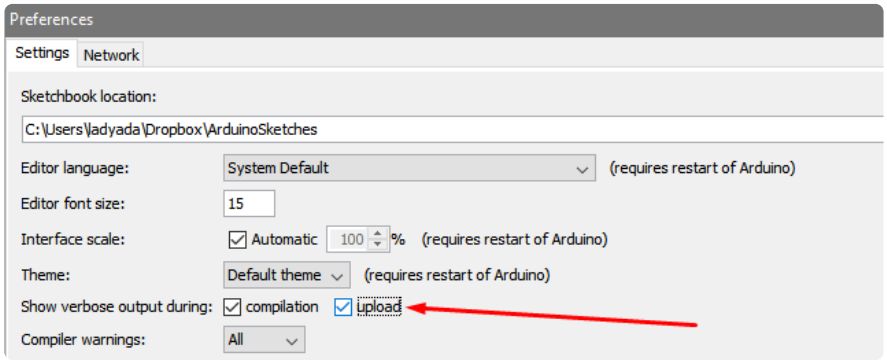
Modern chips often have 'native' USB - that means that there is no separate chip for USB interface. It's all in one! Great for cost savings, simplicity of design, reduced size and more control. However, it means the chip must be self-aware enough to be able to put itself into bootload/upload mode on 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.

Text emphasized with a yellow exclamation:

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! 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!

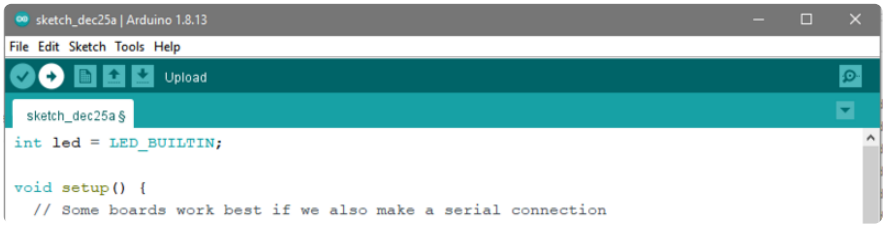
To enter the bootloader:

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.

Once you are in manual bootload mode, verify you see a new disk drive on your computer desktop named **RPI-RP2**



Now you can try uploading again!



This time, you should have success!

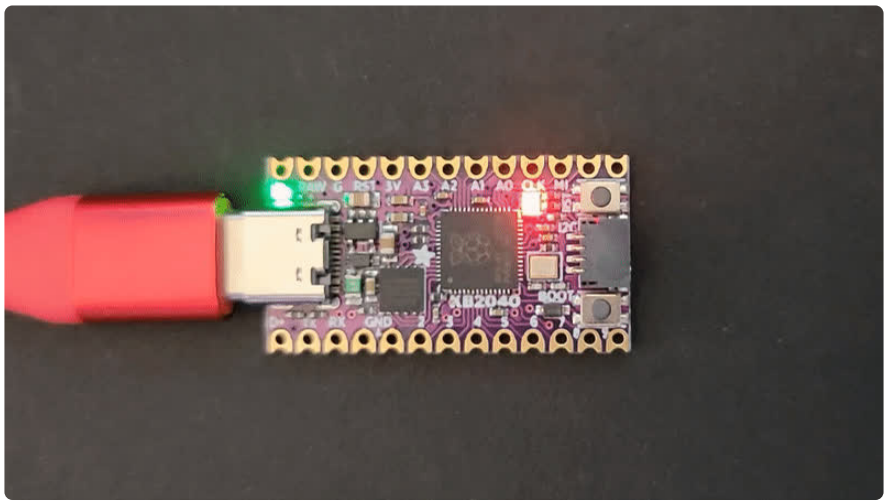
After uploading this way, be sure to **click the reset button** - it sort of makes sure that the board got a good reset and will come back to life nicely.

Text emphasized with a yellow exclamation:	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

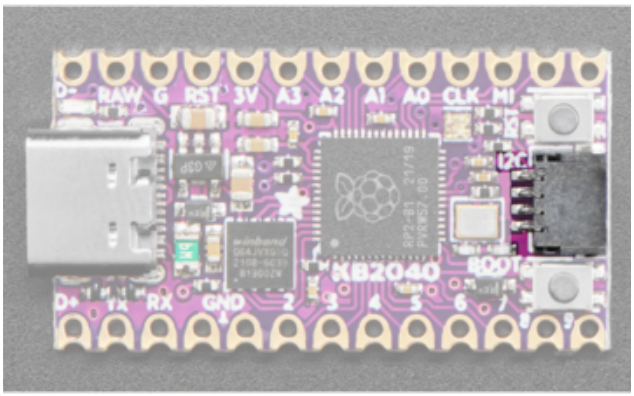
Common I2C Connectivity Issues

- **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)).

I2C Setup on the KB2040



The KB2040 has **TWO** I2C ports:

`Wire` is the default port, and is on the STEMMA QT port

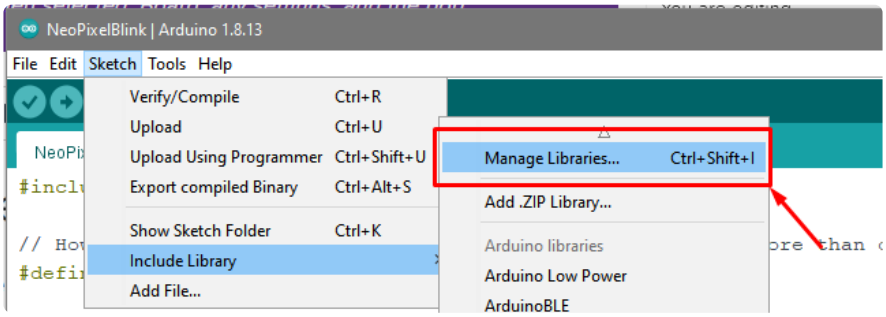
`Wire1` is the alternate/secondary port, and is available using pins 2 (SDA1) and 3 (SCL1)

Neither of the ports have pull up resistors built in! You will need to check your sensor has them or wire two pullup resistors yourself.

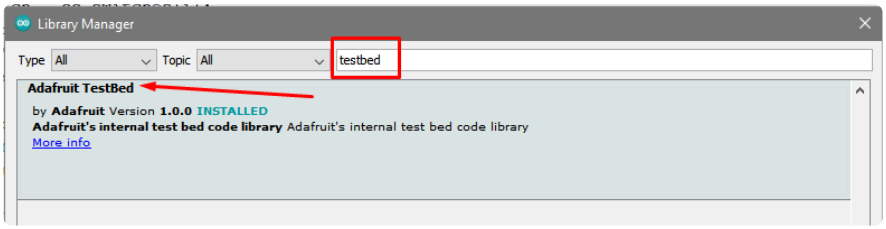
Perform an I2C scan!

Install TestBed Library

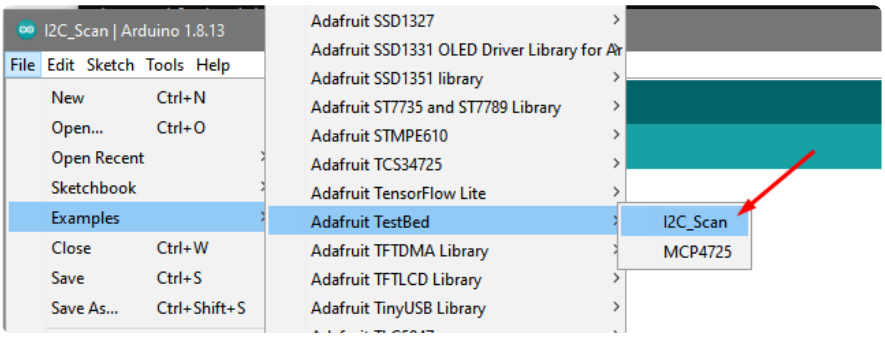
To scan I2C, the Adafruit TestBed library is used. This library and example just makes the scan a little easier to run because it takes care of some of the basics. You will need to add support by installing the library. Good news: it is very easy to do it. Go to the **Arduino Library Manager.**



Search for **TestBed** and install the **Adafruit TestBed** library



Now open up the I2C Scan example




```
1  // SPDX-FileCopyrightText: 2023 Carter Nelson for Adafruit Industries
2  //
3  // SPDX-License-Identifier: MIT
4  // ------------------------------------
5  // i2c_scanner
6  //
```

```
// Modified from https://playground.arduino.cc/Main/I2cScanner/
// --------------------------------------

#include <Wire.h>

// Set I2C bus to use: Wire, Wire1, etc.
#define WIRE Wire

void setup() {
  WIRE.begin();

  Serial.begin(9600);
  while (!Serial)
    delay(10);
  Serial.println("\nI2C Scanner");
}


void loop() {
  byte error, address;
  int nDevices;

  Serial.println("Scanning...");

  nDevices = 0;
  for(address = 1; address < 127; address++ )
  {
    // The i2c_scanner uses the return value of
    // the Write.endTransmisstion to see if
    // a device did acknowledge to the address.
    WIRE.beginTransmission(address);
    error = WIRE.endTransmission();

    if (error == 0)
    {
      Serial.print("I2C device found at address 0x");
      if (address<16)
        Serial.print("0");
      Serial.print(address,HEX);
      Serial.println("  !");

      nDevices++;
    }
    else if (error==4)
    {
      Serial.print("Unknown error at address 0x");
      if (address<16)
        Serial.print("0");
      Serial.println(address,HEX);
    }
  }
  if (nDevices == 0)
    Serial.println("No I2C devices found\n");
  else
    Serial.println("done\n");

```

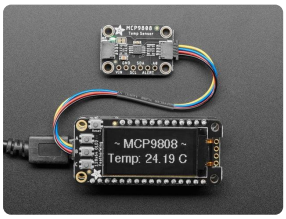
```
63    delay(5000);          // wait 5 seconds for next scan
64  }
```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/I2C_Scanners/arduino/i2c_scanner/i2c_scanner.ino

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.

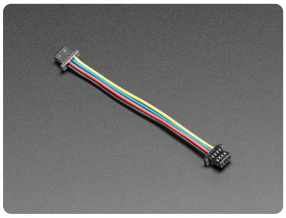


Featured

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



Featured

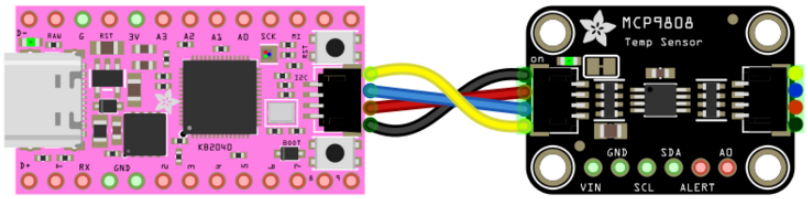
STEMMA QT / Qwiic JST SH 4-Pin Cable - 50mm Long

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.



Note that since the KB2040 has **TWO** I2C ports, this will make the sensor appear on the `Wire` port only (to have it on the `Wire1` secondary port, connect SDA/SCL to pins 2 and 3)

Now upload the scanning sketch to your microcontroller and open the serial port to see the output. You should see something like this:



The MCP9808 is appearing on the default port as address **0x18** as expected. There's nothing on the secondary port so there's not text there.

Factory Reset

The KB2040 microcontroller ships running a NeoPixel rainbow swirl example. 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.

!	Text emphasized with a red exclamation:	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.

Step 2. Enter RP2040 bootloader mode

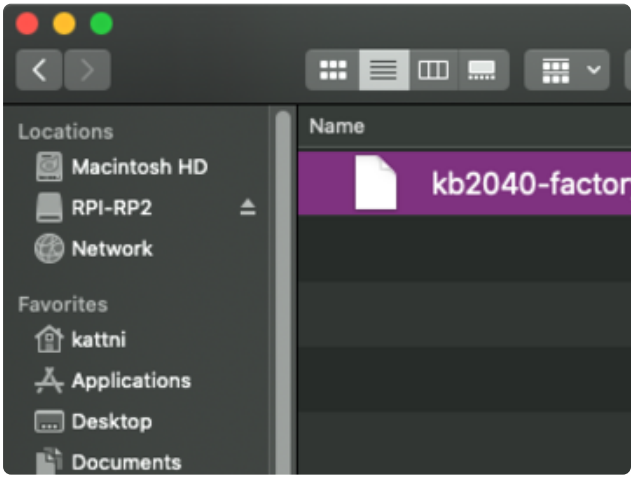
Entering the RP2040 bootloader is easy. Complete the following steps.

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!

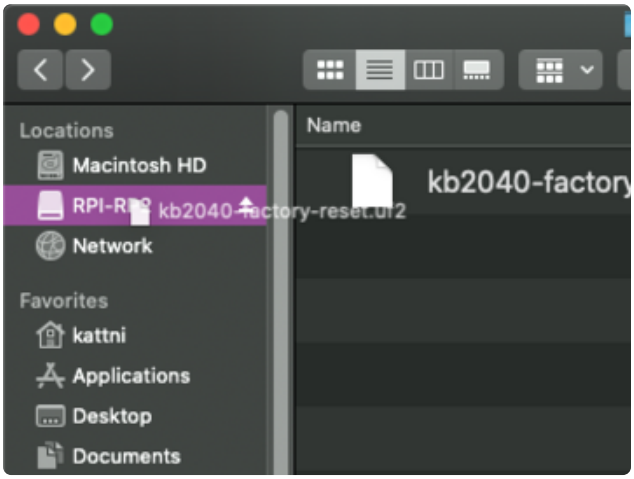
To enter the bootloader:

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.

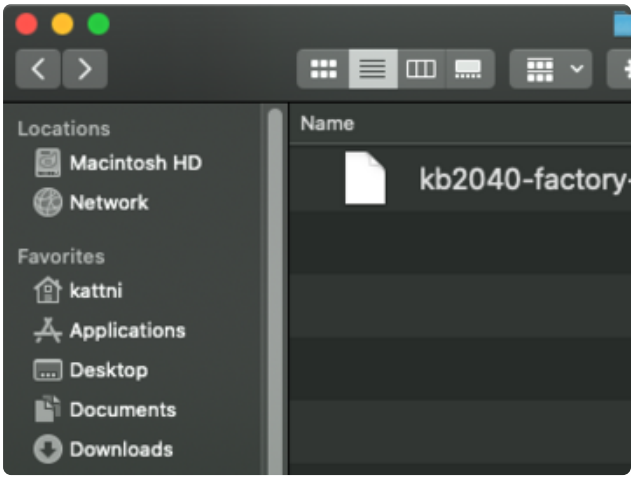
Step 3. Drag UF2 file to RPI-RP2



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 KB2040 will light up in a rainbow swirl.

You've successfully returned your board to a factory reset state!

Flash Resetting UF2

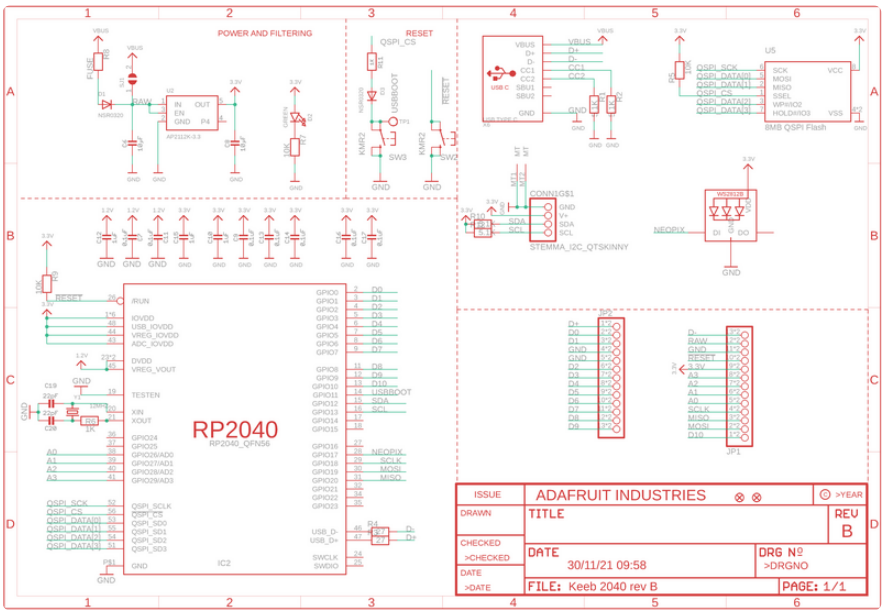
If your board ever gets into a really weird state and doesn't even show up when loading code, try loading this 'nuke' UF2 which will do a 'deep clean' on your Flash Memory. **You will lose all the files on the board**, but at least you'll be able to revive it! Download the file below, and follow the 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.

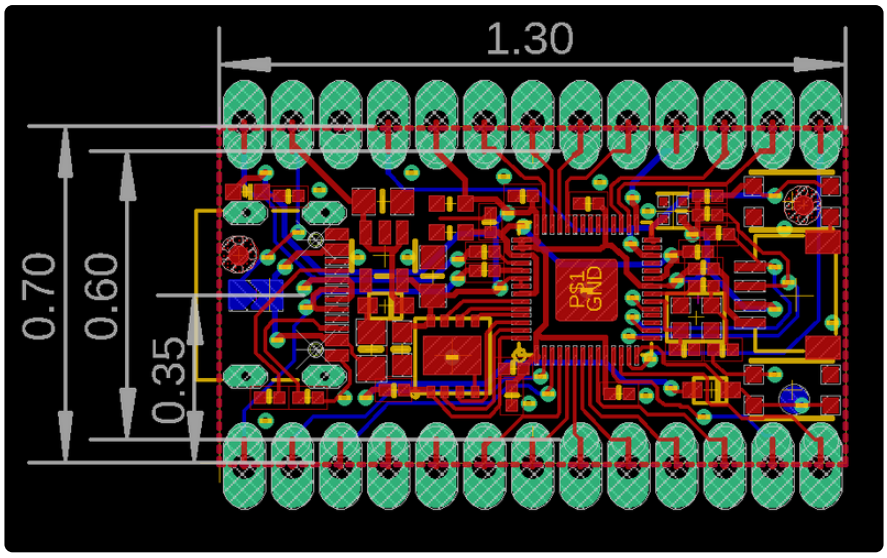
Downloads

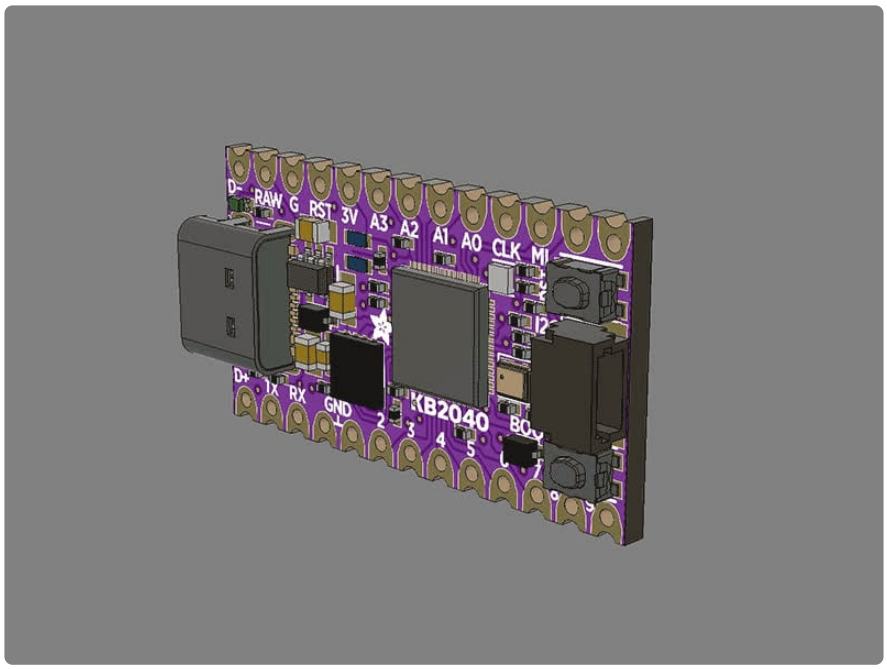
Files:

- RP2040 Datasheet ()
- EagleCAD PCB files on GitHub ()
- 3D Models on GitHub ()
- Fritzing object in the Adafruit Fritzing Library ()
- PrettyPins PDF on GitHub ()
- KB2040 PrettyPins SVG ()

Schematic and Fab Print




1.30
0.70
0.60
0.35
P-SL
GND


D-
RAW
G
RST
3V
A3
A2
A1
A0
CLK
D+
TX
RX
GND
KB2040