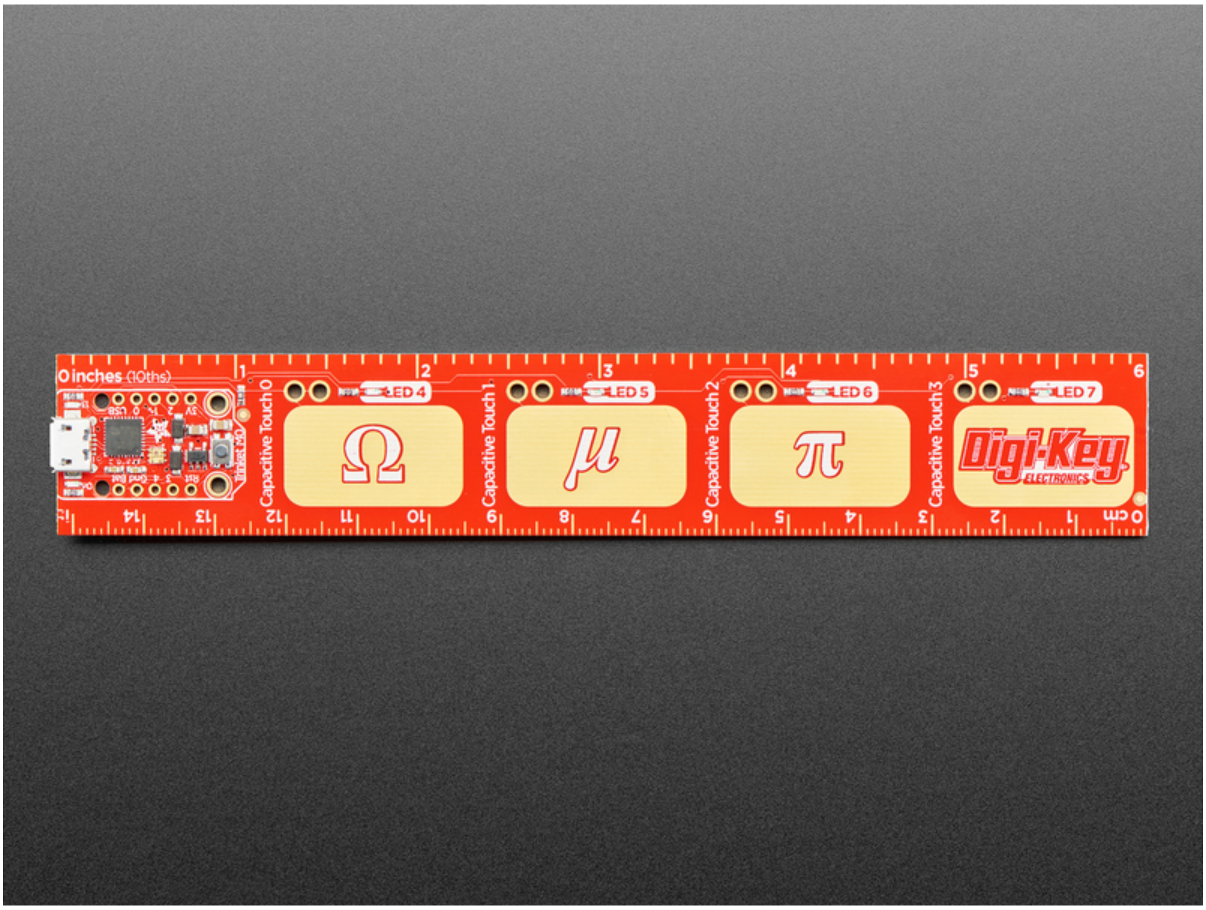
Adafruit PyRuler

Created by Kattni Rembor

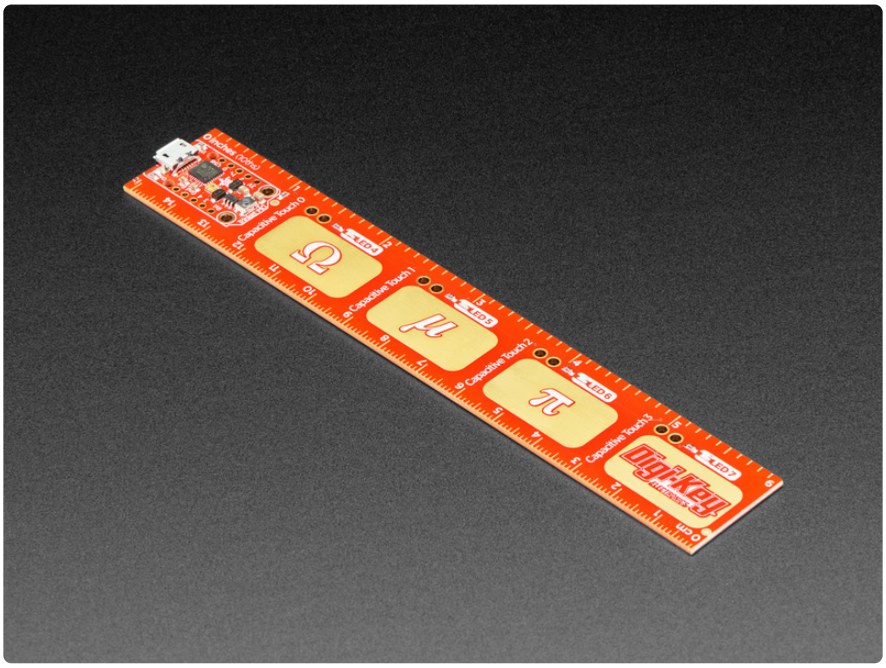


https://learn.adafruit.com/adafruit-pyruler

Last updated on 2024-04-04 10:40:08 AM EDT

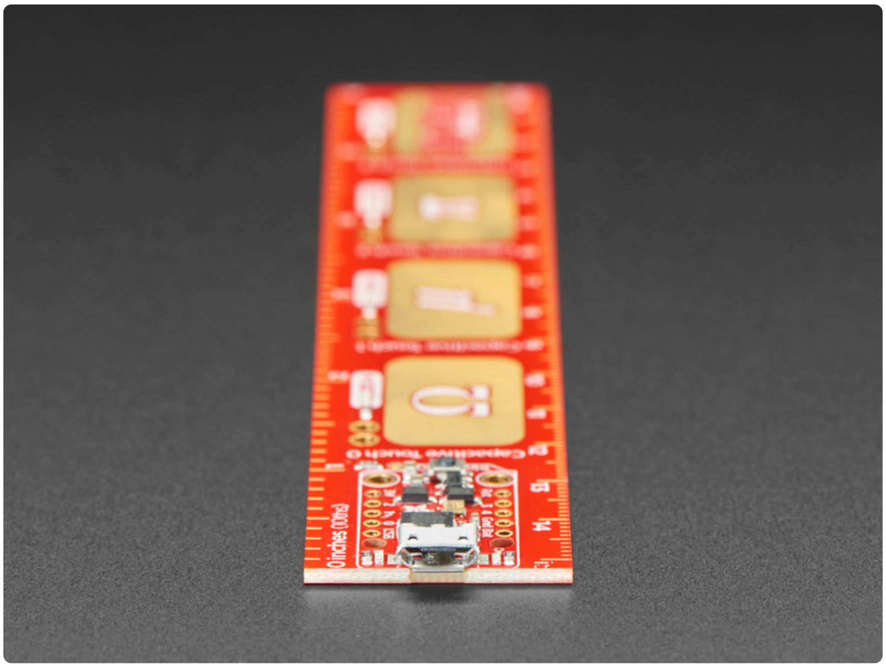
Table of Contents

Overview



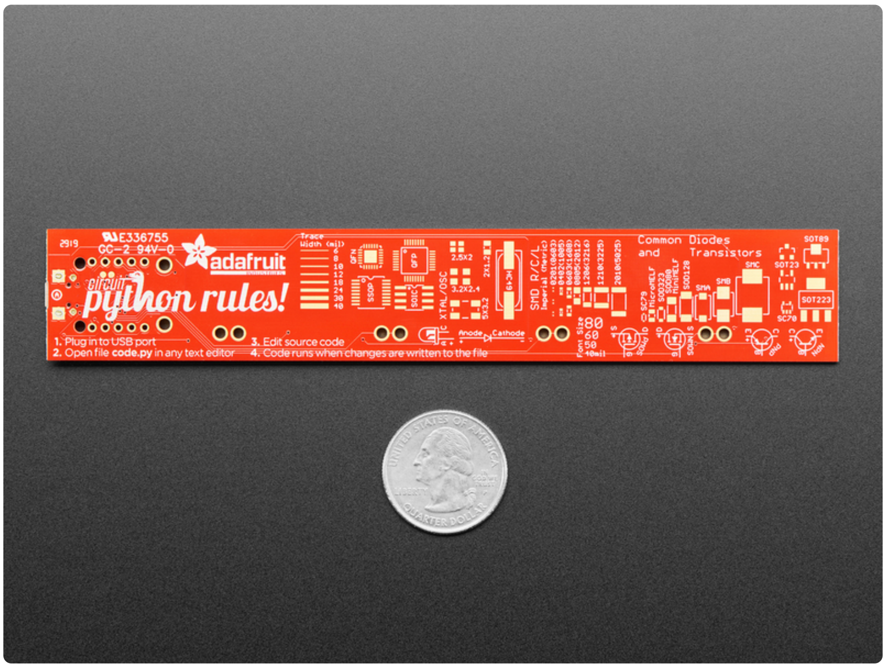
The first time you soldered up a surface mount component you may have been surprised "these are really small parts!" and there's dozens of different names too! QFN, TDFN, SOIC, SOP, J-Lead, what do they mean and how can you tell how big they are? Now you can have a reference board at your fingertips, with this snazzy PCB reference ruler.

Measuring approx 1" x 6", this standard-thickness FR4, gold plated ruler has the most common component packages you'll encounter. It also has font size guide, and a trace-width diagram.



That's not all, it's even a fully featured microcontroller board! Embedded in the end is a Trinket M0, our little Cortex M0+ development board, **and in addition, there's 4 capacitive touch pads with matching LEDs** that our code will turn into a specialized engineer keyboard. We're always needing to type Ω and μ but we can never memorize the complex key-commands necessary. Thanks to CircuitPython, it's super-easy to make a touch keyboard to solve this for you. Plug in the ruler into your computer, if it's your first time using it, you'll need to open up the **code.py** file and set the Keyboard mode to "True". Now when you touch the pads, you'll get a Ω, μ, π or, when the Digi-Key logo is touched, the URL for Digi-Key's Python on Hardware guide.

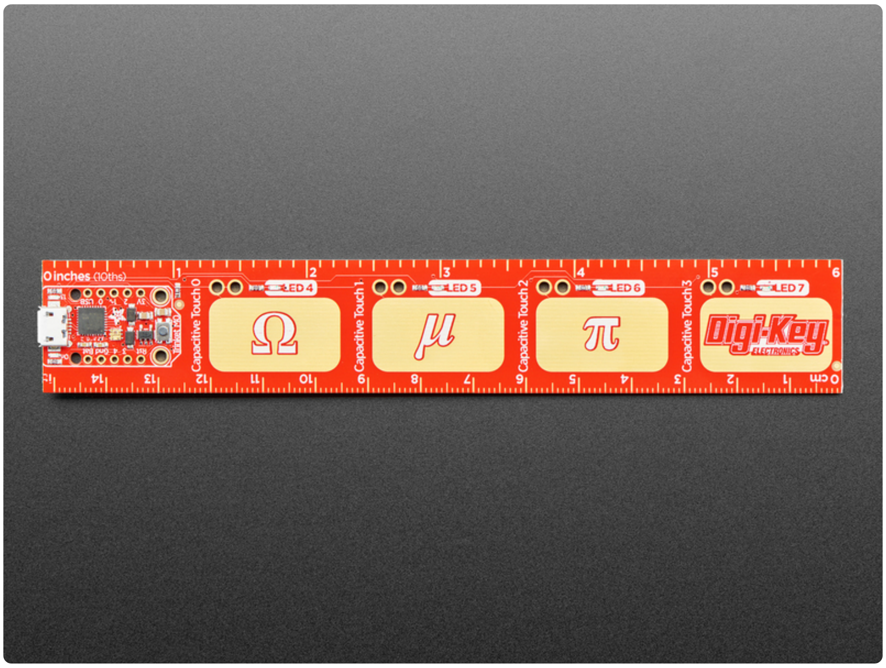
The PyRuler was designed as a great introduction to CircuitPython. While you can use it with the Arduino IDE, we are shipping it with CircuitPython on board. When you plug it in, it will show up as a very small disk drive with **code.py** on it. Edit **code.py** with your favorite text editor to build your project using Python, the most popular programming language. No installs, IDE or compiler needed, so you can use it on any computer, even ChromeBooks or computers you can't install software on. When you're done, unplug the PyRuler and your code will go with you. Please check out the Trinket M0 CircuitPython guide for a list of capabilities and quick-start code examples (https://adafru.it/ABS) - CircuitPython is easier to code but not as low-level as Arduino.



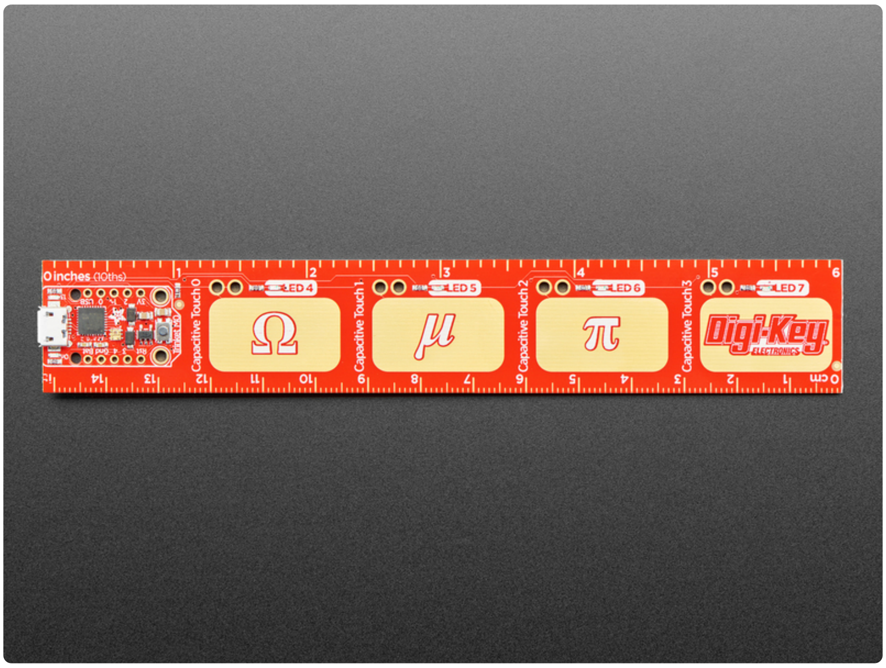
Here are some of the updates you can look forward to when using PyRuler

- ATSAMD21E18 32-bit Cortex M0+ - 256KB Flash, 32 KB RAM, 48 MHz 32 bit processor

- Native USB supported by every OS - can be used in Arduino or CircuitPython as USB serial console, Keyboard/Mouse HID, even a little disk drive for storing Python scripts.
- Can be used with the Arduino IDE or CircuitPython
- Four capacitive touch pads.
- Lots of LEDS - Built in green ON LED, red pin #13 LED, RGB DotStar LED, **plus red/yellow/green/blue** matching LEDs for each capacitive touch pad
- 5 GPIO header pins are available and are not shared with USB or the touch pads/LEDs - so you can use them for whatever you like! Digital I/O with pullup/down. 3 ADCs, 1 DAC, 2 PWM, 3 extra captouch sensors
- Can drive NeoPixels or DotStars on any pins, with enough memory to drive 8000+ pixels. [DMA-NeoPixel support on one pin](https://adafru.it/xYD) (https://adafru.it/xYD)so you can drive pixels without having to spend any processor time on it.
- Native hardware SPI, I2C and Serial available on two pads so you can connect to any I2C or Serial device with true hardware support (no annoying bit-banging). You can have either one SPI device or both I2C and Serial.
- Reset switch for starting your project code over
- Power with either USB or external output (such as a battery) - it'll automatically switch over



Pinouts



Capacitive Touch Pins

There are four capacitive touch pads on the PyRuler!

- **Capacitive Touch 0** - This is the the Ω capacitive touch pad. It is addressable at pin **CAP0**.
- **Capacitive Touch 1** - This is the the µ capacitive touch pad. It is addressable at pin **CAP1**.
- **Capacitive Touch 2** - This is the the π capacitive touch pad. It is addressable at pin **CAP2**.
- **Capacitive Touch 3** - This is the the Digi-Key Electronics capacitive touch pad. It is addressable at pin **CAP3**.

LEDs

There are 5 LEDs on the PyRuler.

- **LED 4** - This red LED is located above the Ω capacitive touch pad. It can be addressed at pin **LED4**.
- **LED 5** - This yellow LED is located above the µ capacitive touch pad. It can be addressed at pin **LED5**.

- **LED 6** -This green LED is located above the π capacitive touch pad. It can be addressed at pin **LED6**.
- **LED 7** - This blue LED is located above the Digi-Key Electronics capacitive touch pad. It can be addressed at pin **LED7**.
- **D13 / red status LED** - Located next to the USB connector on the left end of the PyRuler. The red status LED can be addressed using pin D13.
- **RGB DotStar status LED** - Located under the Adafruit symbol inside the white outline on the left end of the PyRuler. Addressable at **APA102_MOSI** as the data pin and **APA102_SCK** as the clock pin.

Power Pins

About half of the pins on the Trinket M0 are related to power in and out: **3V**, **USB**, **BAT** and **GND**

- **BAT** - This is a voltage **INPUT** pin, you can use it to connect a battery or other external power to the Trinket. It has a Schottkey protection diode so it is completely separate from the USB power input/output. You can put 3V-6V into this pin and it will be regulated down by the 3V regulator
- **USB** - This is a voltage **OUTPUT** or **INPUT** pin - it is connected directly to the micro USB port +5V pin, so if you are powering over usb, this pin will give you 5V out at 500mA+. Or if you are using the Trinket as a USB host or you have a good reason, you can put 5V into this pin and it will back-power the USB port.
- **3V** - This is the **3.3V OUTPUT** pad from the voltage regulator. It can provide up to 500mA at a steady 3.3V. Good for sensors or small LEDs or other 3V devices.
- **GND** is the common ground pin, used for logic and power. It is connected to the USB ground and the power regulator, etc. This is the pin you'll want to use for any and all ground connections

Input/Output Pins

Next we will cover the 5 GPIO (General Purpose Input Output) pins! For reference, you may want to also check out the datasheet-reference in the downloads section for the core ATSAMD21E18 pins. We picked pins that have a lot of capabilities.

Common to all pads

All the GPIO pads can be used as digital inputs, digital outputs, for LEDs, buttons and switches. All pads can also be used as hardware interrupts inputs.

Each pad can provide up to ~7mA of current. Don't connect a motor or other high-power component directly to the pins! Instead, use a transistor to power the DC motor on/off (https://adafru.it/aUD)

On a Trinket M0, the GPIO are 3.3V output level, and should not be used with 5V inputs. In general, most 5V devices are OK with 3.3V output though.

The five pins are completely 'free' pins, they are not used by the USB connection, LEDs, DotStar, etc so you never have to worry about the USB interface interfering with them when programming

Unique pad capabilities

- **Digital #0 / A2** - this is connected to **PA08** on the ATSAMD21. This pin can be used as a digital I/O with selectable pullup or pulldown, analog input (use 'A2'), PWM output, and is also used for I2C data (SDA)
- **Digital #1 / A0** - this is connected to **PA02** on the ATSAMD21. This pin can be used as a digital I/O with selectable pullup or pulldown, capacitive touch, analog input (use 'A0'), and true analog (10-bit DAC) output. It cannot be used as PWM output.
- **Digital #2 / A1** - this is connected to **PA09** on the ATSAMD21. This pin can be used as a digital I/O with selectable pullup or pulldown, analog input (use 'A1'), PWM output, and is also used for I2C clock (SCL), and hardware SPI MISO
- **Digital #3 / A3** - this is connected to **PA07** on the ATSAMD21. This pin can be used as a digital I/O with selectable pullup or pulldown, analog input (use 'A3'), capacitive touch, PWM output, and is also used for UART RX (**Serial1** in Arduino), and hardware SPI SCK
- **Digital #4 / A4** - this is connected to **PA06** on the ATSAMD21. This pin can be used as a digital I/O with selectable pullup or pulldown, analog input (use 'A4'), capacitive touch, PWM output, and is also used for UART TX (**Serial1** in Arduino), and hardware SPI MOSI

Other Pads!

- **Digital #7** - You can't see this pin but it is connected to the internal RGB DotStar data in pin
- **Digital #8** - You can't see this pin but it is connected to the internal RGB DotStar clock in pin
- **Digital #13** - You can't see this pin but it is connected to the little red status LED

Usage

The PyRuler is basically a Trinket M0 with extra LEDs and capacitive touch pads.

This demo program will let you turn the capacitive touch pads into a little keyboard. Three of the pads have a 'true' capacitive touch hardware interface, with the fourth we DIY a capacitive interface - we recommend using `read_caps` which will do all the nitty gritty for you.

The HID keyboard library is what emits keypresses. Note that Mac and Windows have different ways of typing special characters so you will have to set a `WINDOWS_COMPUTER` variable to match the host computer used!

```
import os
import board
from digitalio import DigitalInOut, Direction
import time
import touchio

# Set this to True to turn the touchpads into a keyboard
ENABLE_KEYBOARD = False

WINDOWS = "W"
MAC = "M"
LINUX = "L"  # and Chrome OS

# Set your computer type to one of the above
OS = WINDOWS

# Used if we do HID output, see below
if ENABLE_KEYBOARD:
    import usb_hid
    from adafruit_hid.keyboard import Keyboard
    from adafruit_hid.keycode import Keycode
    from adafruit_hid.keyboard_layout_us import KeyboardLayoutUS
    kbd = Keyboard(usb_hid.devices)
    layout = KeyboardLayoutUS(kbd)

#print(dir(board), os.uname()) # Print a little about ourselves

led = DigitalInOut(board.D13)
led.direction = Direction.OUTPUT

touches = [DigitalInOut(board.CAP0)]
for p in (board.CAP1, board.CAP2, board.CAP3):
```

```
    touches.append(touchio.TouchIn(p))

leds = []
for p in (board.LED4, board.LED5, board.LED6, board.LED7):
    led = DigitalInOut(p)
    led.direction = Direction.OUTPUT
    led.value = True
    time.sleep(0.25)
    leds.append(led)
for led in leds:
    led.value = False


cap_touches = [False, False, False, False]

def read_caps():
    t0_count = 0
    t0 = touches[0]
    t0.direction = Direction.OUTPUT
    t0.value = True
    t0.direction = Direction.INPUT
    # funky idea but we can 'diy' the one non-hardware captouch device by hand
    # by reading the drooping voltage on a tri-state pin.
    t0_count = t0.value + t0.value + t0.value + t0.value + t0.value + \
               t0.value + t0.value + t0.value + t0.value + t0.value + \
               t0.value + t0.value + t0.value + t0.value + t0.value
    cap_touches[0] = t0_count > 2
    cap_touches[1] = touches[1].raw_value > 3000
    cap_touches[2] = touches[2].raw_value > 3000
    cap_touches[3] = touches[3].raw_value > 3000
    return cap_touches

def type_alt_code(code):
    kbd.press(Keycode.ALT)
    for c in str(code):
        if c == '0':
            keycode = Keycode.KEYPAD_ZERO
        elif '1' <= c <= '9':
            keycode = Keycode.KEYPAD_ONE + ord(c) - ord('1')
        else:
            raise RuntimeError("Only number codes permitted!")
        kbd.press(keycode)
        kbd.release(keycode)
    kbd.release_all()

while True:
    caps = read_caps()
    print(caps)
    # light up the matching LED
    for i,c in enumerate(caps):
        leds[i].value = c
    if caps[0]:
        if ENABLE_KEYBOARD:
            if OS == WINDOWS:
                type_alt_code(234)
            elif OS == MAC:
                kbd.send(Keycode.ALT, Keycode.Z)
            elif OS == LINUX:
                kbd.press(Keycode.CONTROL, Keycode.SHIFT)
                kbd.press(Keycode.U)
                kbd.release_all()
                kbd.send(Keycode.TWO)
                kbd.send(Keycode.ONE)
                kbd.send(Keycode.TWO)
                kbd.send(Keycode.SIX)
                kbd.send(Keycode.ENTER)
    if caps[1]:
        if ENABLE_KEYBOARD:
            if OS == WINDOWS:
```

```
                type_alt_code(230)
            elif OS == MAC:
                kbd.send(Keycode.ALT, Keycode.M)
            elif OS == LINUX:
                kbd.press(Keycode.CONTROL, Keycode.SHIFT)
                kbd.press(Keycode.U)
                kbd.release_all()
                kbd.send(Keycode.ZERO)
                kbd.send(Keycode.THREE)
                kbd.send(Keycode.B)
                kbd.send(Keycode.C)
                kbd.send(Keycode.ENTER)
    if caps[2]:
        if ENABLE_KEYBOARD:
            if OS == WINDOWS:
                type_alt_code(227)
            elif OS == MAC:
                kbd.send(Keycode.ALT, Keycode.P)
            elif OS == LINUX:
                kbd.press(Keycode.CONTROL, Keycode.SHIFT)
                kbd.press(Keycode.U)
                kbd.release_all()
                kbd.send(Keycode.ZERO)
                kbd.send(Keycode.THREE)
                kbd.send(Keycode.C)
                kbd.send(Keycode.ZERO)
                kbd.send(Keycode.ENTER)
    if caps[3]:
        if ENABLE_KEYBOARD:
            layout.write('https://www.digikey.com/python\n')
    time.sleep(0.1)
```

What is CircuitPython?

CircuitPython is a programming language designed to simplify experimenting and learning to program on low-cost microcontroller boards. It makes getting started easier than ever with no upfront desktop downloads needed. Once you get your board set up, open any text editor, and get started editing code. It's that simple.

CircuitPython is based on Python

Python is the fastest growing programming language. It's taught in schools and universities. It's a high-level programming language which means it's designed to be easier to read, write and maintain. It supports modules and packages which means it's easy to reuse your code for other projects. It has a built in interpreter which means there are no extra steps, like compiling, to get your code to work. And of course, Python is Open Source Software which means it's free for anyone to use, modify or improve upon.

CircuitPython adds hardware support to all of these amazing features. If you already have Python knowledge, you can easily apply that to using CircuitPython. If you have no previous experience, it's really simple to get started!

Why would I use CircuitPython?

CircuitPython is designed to run on microcontroller boards. A microcontroller board is a board with a microcontroller chip that's essentially an itty-bitty all-in-one computer. The board you're holding is a microcontroller board! CircuitPython is easy to use because all you need is that little board, a USB cable, and a computer with a USB connection. But that's only the beginning.

Other reasons to use CircuitPython include:

- **You want to get up and running quickly.** Create a file, edit your code, save the file, and it runs immediately. There is no compiling, no downloading and no uploading needed.
- **You're new to programming.** CircuitPython is designed with education in mind. It's easy to start learning how to program and you get immediate feedback from the board.
- **Easily update your code.** Since your code lives on the disk drive, you can edit it whenever you like, you can also keep multiple files around for easy experimentation.
- **The serial console and REPL.** These allow for live feedback from your code and interactive programming.
- **File storage.** The internal storage for CircuitPython makes it great for data-logging, playing audio clips, and otherwise interacting with files.

- **Strong hardware support.** CircuitPython has builtin support for microcontroller hardware features like digital I/O pins, hardware buses (UART, I2C, SPI), audio I/O, and other capabilities. There are also many libraries and drivers for sensors, breakout boards and other external components.
- **It's Python!** Python is the fastest-growing programming language. It's taught in schools and universities. CircuitPython is almost-completely compatible with Python. It simply adds hardware support.

This is just the beginning. CircuitPython continues to evolve, and is constantly being updated. Adafruit welcomes and encourages feedback from the community, and incorporate it into the development of CircuitPython. That's the core of the open source concept. This makes CircuitPython better for you and everyone who uses it!

CircuitPython

CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. The trinket M0 is the second board that comes pre-loaded with CircuitPython. Simply copy and edit files on the **CIRCUITPY** drive to iterate.

Your PyRuler already comes with CircuitPython but maybe there's a new version, or you overwrote your Trinket M0 with Arduino code! In that case, see the below for how to reinstall or update CircuitPython. Otherwise you can skip this and go straight to the next page!

If you have already plugged in your board, start by ejecting or "safely remove" the CIRCUITPY drive. This is a good practice to get into. Always eject before unplugging or resetting your board!

Set up CircuitPython Quick Start!

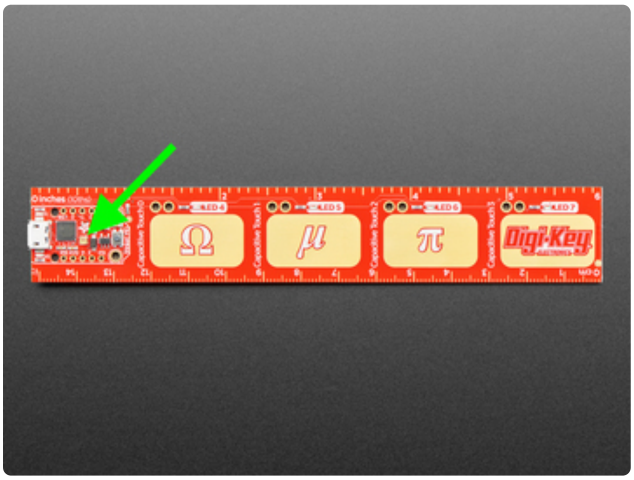
Follow this quick step-by-step for super-fast Python power :)

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

https://adafru.it/Fst

Click the link above and download the latest UF2 file.

Download and save it to your desktop (or wherever is handy).

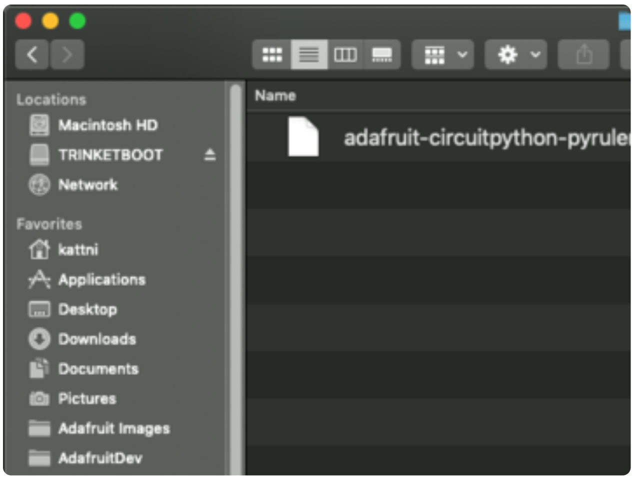


Plug your PyRuler into your computer using a known-good USB cable.

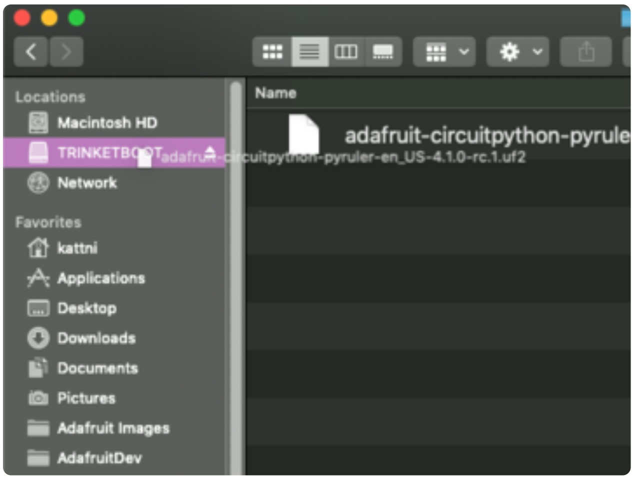
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Double-click the small **Reset** button next to the Trinket M0 name printed on your board, and you will see the Dotstar RGB LED, noted by the green arrow in the image, turn green. If it turns red, check the USB cable, try another USB port, etc. **Note:** The little LED above the USB connector will be red - this is ok!

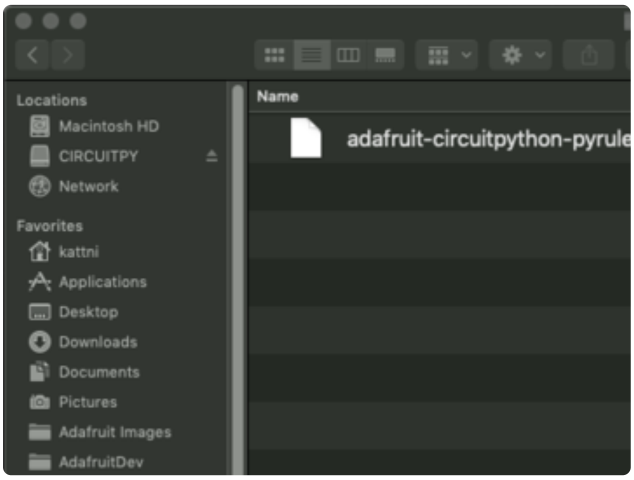
If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!



You will see a new disk drive appear called **TRINKETBOOT**.



Drag the **adafruit_circuitpython_etc.uf2** file to **TRINKETBOOT**



The red LED will flash. Then, the **TRINKETBOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

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

Further Information

For more detailed info on installing CircuitPython, check out Installing CircuitPython (https://adafru.it/Amd).

Installing the Mu Editor

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

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

Download and Install Mu

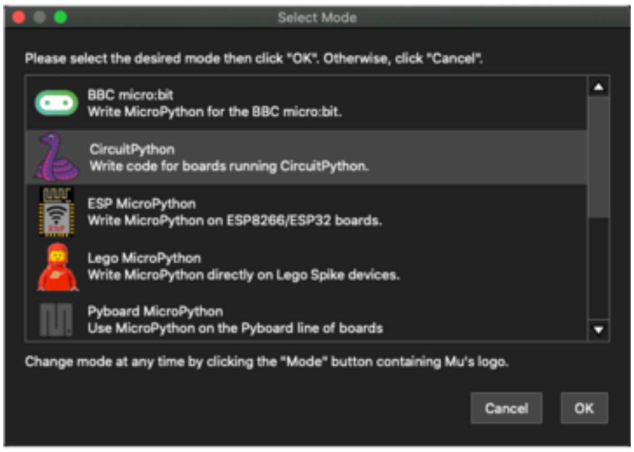
Download Mu from https://codewith.mu (https://adafru.it/Be6).

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

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

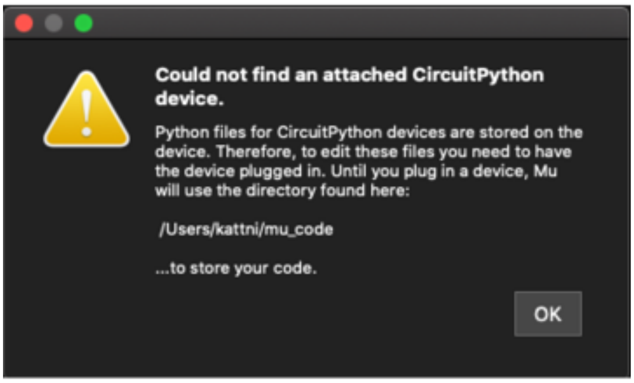
Windows users: due to the nature of MSI installers, please remove old versions of Mu before installing the latest version.

Starting Up Mu



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

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



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

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

Using Mu

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

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

Creating and Editing Code

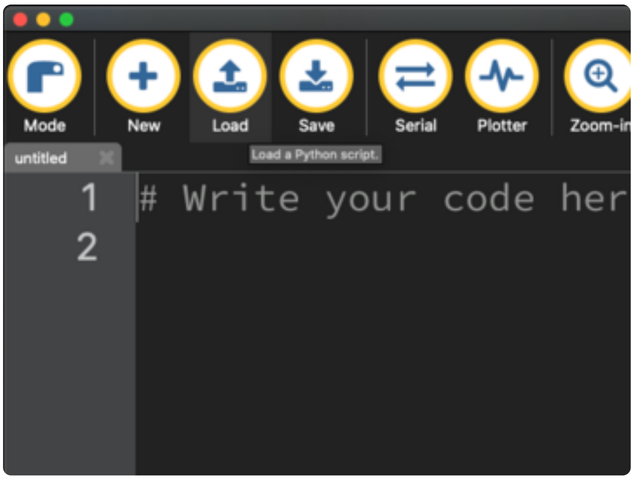
One of the best things about CircuitPython is how simple it is to get code up and running. This section covers how to create and edit your first CircuitPython program.

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

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

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

Creating Code



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

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

Copy and paste the following code into your editor:

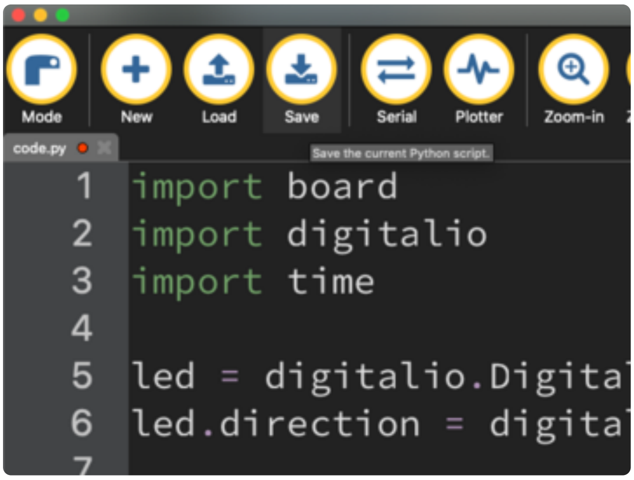
```
import board
import digitalio
import time

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

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

The KB2040, QT Py , 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 (https://adafru.it/UDU).

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

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



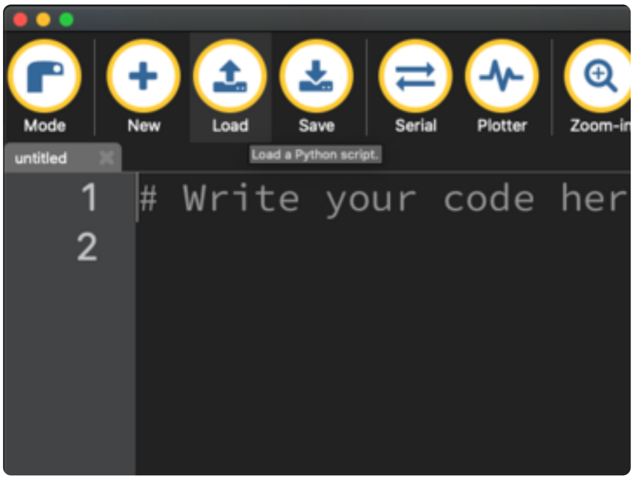
Save the **code.py** file on your **CIRCUITPY** drive.

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

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

On most boards you'll find a tiny red LED. On the ItsyBitsy nRF52840, you'll find a tiny blue LED. On QT Py M0, QT Py RP2040, 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...

Don't click reset or unplug your board!

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

There are a couple of ways to avoid filesystem corruption.

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

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

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

2. Eject or Sync the Drive After Writing

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

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

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

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

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

Back to Editing Code...

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

```
import board
import digitalio
import time

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

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

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

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

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

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

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

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

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

Naming Your Program File

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

Connecting to the Serial Console

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

`print("Hello, world!")`

This line in your code.py would result in:

`Hello, world!`

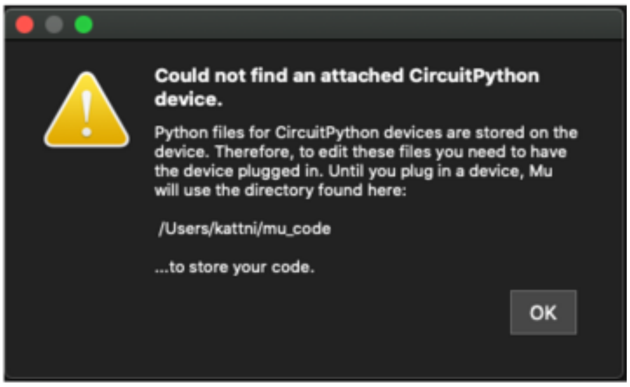
However, these print statements need somewhere to display. That's where the serial console comes in!

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

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

Are you using Mu?

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



First, make sure your CircuitPython board is plugged in.

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

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

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

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

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

Serial Console Issues or Delays on Linux

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

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

```
sudo apt purge modemmanager
```

Setting Permissions on Linux

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

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

```
sudo adduser $USER dialout
```

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

Using Something Else?

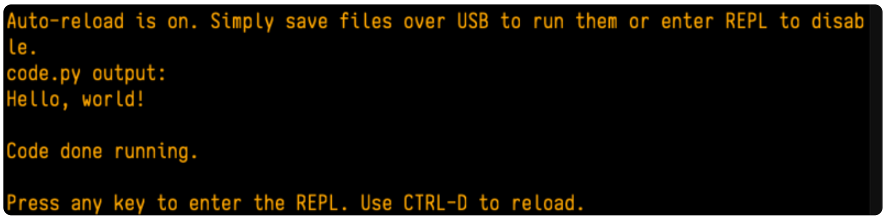
If you're not using Mu to edit, are using or if for some reason you are not a fan of its built in serial console, you can run the serial console from a separate program.

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

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

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

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



Interacting with the Serial Console

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

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

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

```
import board
import digitalio
import time

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

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

```
    time.sleep(1)
    led.value = False
    time.sleep(1)
```

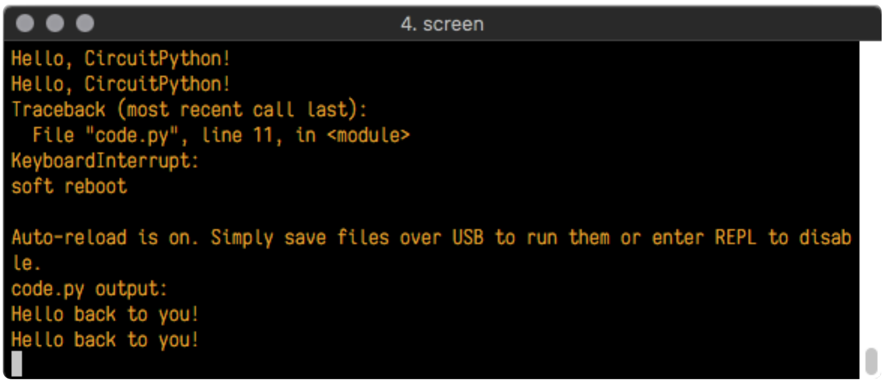
Save your file.

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

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

```
import board
import digitalio
import time

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

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

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



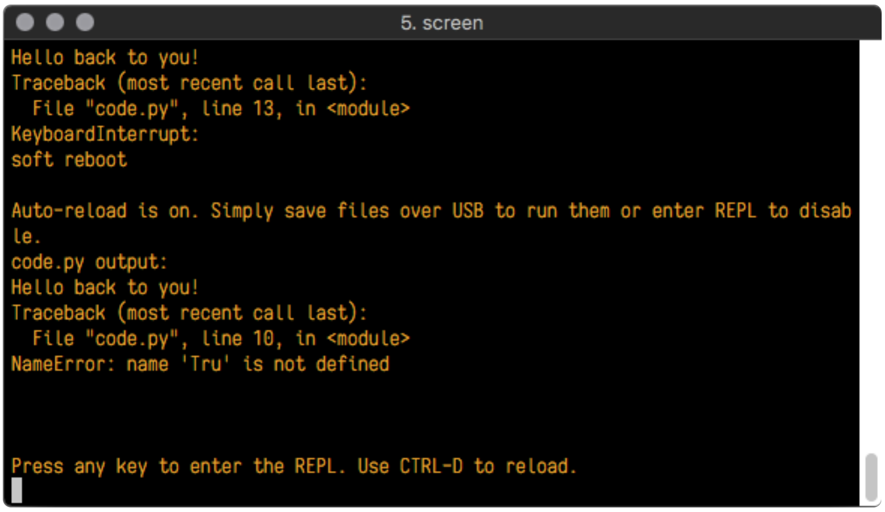
The `Traceback (most recent call last):` is telling you the last thing your board was doing before you saved your file. This is normal behavior and will happen every time the board resets. This is really handy for troubleshooting. Let's introduce an error so you can see how it is used.

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

```
import board
import digitalio
import time

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

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

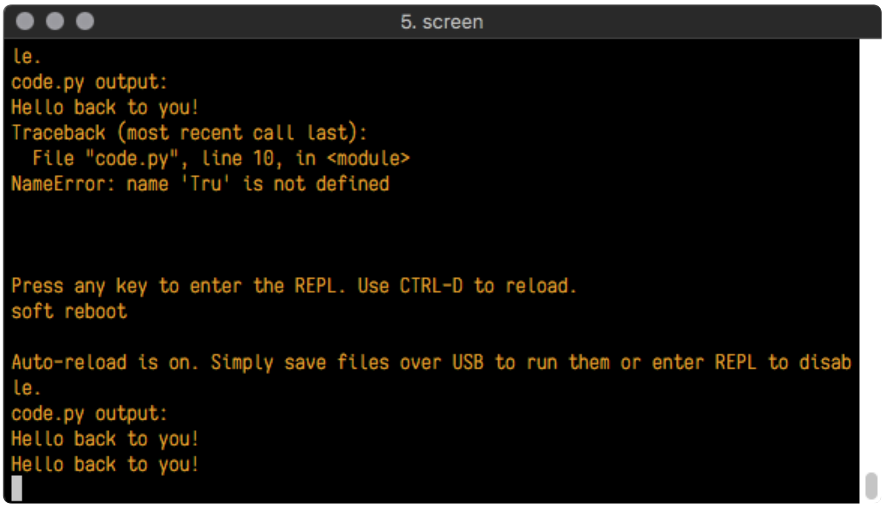
Save your file. You will notice that your red LED will stop blinking, and you may have a colored status LED blinking at you. This is because the code is no longer correct and can no longer run properly. You need to fix it!

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



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

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



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

The serial console will display any output generated by your code. Some sensors, such as a humidity sensor or a thermistor, receive data and you can use print statements to display that information. You can also use print statements for troubleshooting, which is called "print debugging". Essentially, if your code isn't working, and you want to know where it's failing, you can put print statements in various places to see where it stops printing.

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

The REPL

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

Entering the REPL

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

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

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

```
Distance: 14.8 cm
Distance: 6.7 cm
Distance: 3.9 cm
Distance: 3.4 cm
Distance: 6.5 cm
Traceback (most recent call last):
  File "code.py", line 43, in <module>
KeyboardInterrupt:

Code done running.

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

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.

```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disab
le.
code.py output:

Code done running.

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

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.

```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disab
le.

Code done running.

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

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

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

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.

```
Adafruit CircuitPython 7.0.0 on 2021-10-26; Adafruit Feather RP2040 with rp2040
>>> help()
```

Then press enter. You should then see a message.

```
Adafruit CircuitPython 7.0.0 on 2021-10-26; Adafruit Feather RP2040 with rp2040
>>> help()
Welcome to Adafruit CircuitPython 7.0.0!

Visit circuitpython.org for more information.

To list built-in modules type `help("modules")`.
>>>
```

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.

```
>>> 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
Plus any modules on the filesystem
>>>
```

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.

```
>>> import board
>>>
```

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!

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!

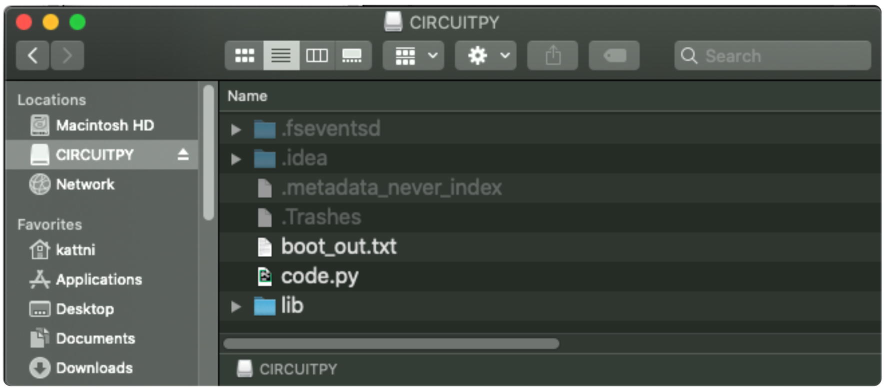
```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disab
le.

Code done running.

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

CircuitPython Libraries

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

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

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



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

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

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

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

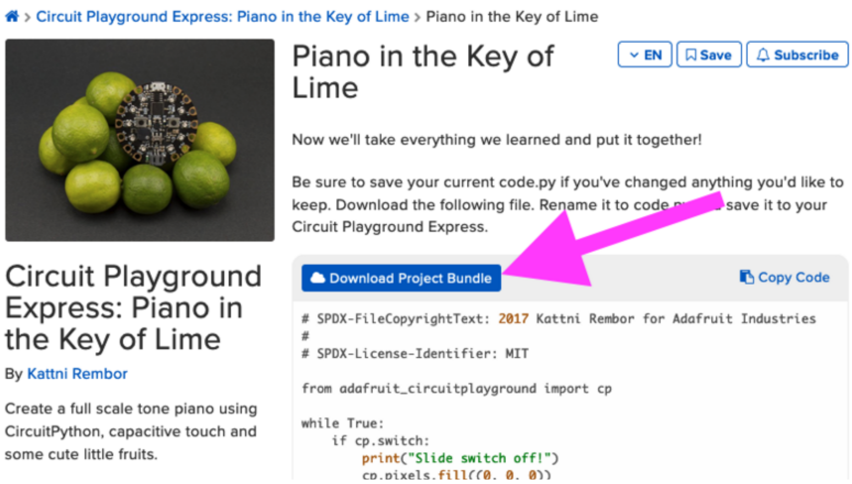
Either way, as you start to explore CircuitPython, you'll want to know how to get libraries on board.

The Adafruit Learn Guide Project Bundle

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

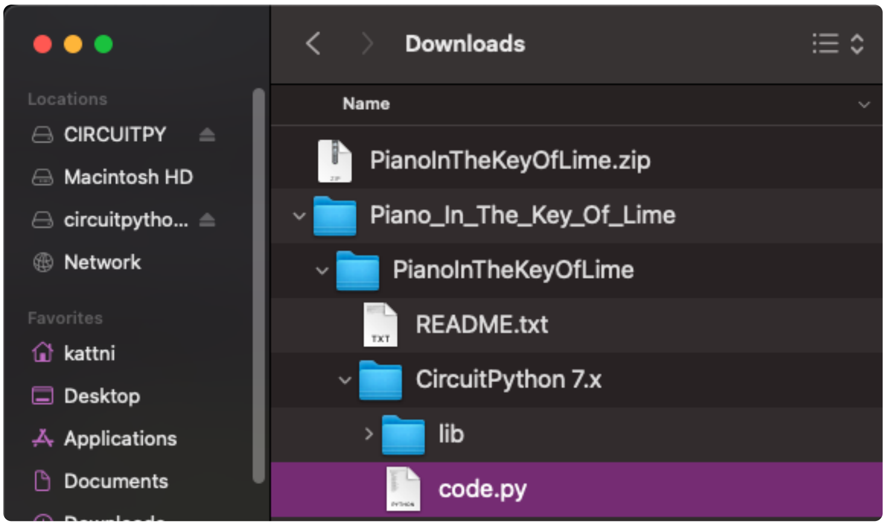
The first step is to find the Download Project Bundle button in the guide you're working on.

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



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

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



The Piano in the Key of Lime guide was chosen as an example. That guide is specific to Circuit Playground Express, and cannot be used on all boards. Do not expect to download that exact bundle and have it work on your non-CPX microcontroller.

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

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

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

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

The Adafruit CircuitPython Library Bundle

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

Downloading the Adafruit CircuitPython Library Bundle

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

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

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

https://adafru.it/ENC

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

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

The CircuitPython Community Library Bundle

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

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

Downloading the CircuitPython Community Library Bundle

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

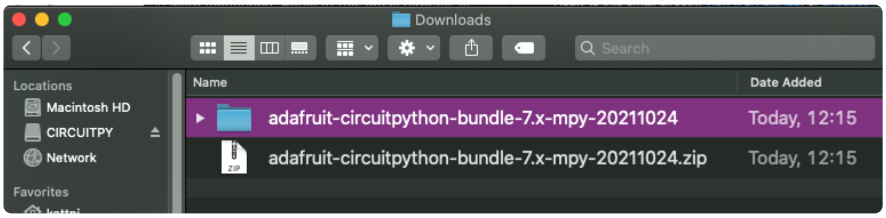
Click for the latest CircuitPython Community Library Bundle release

https://adafru.it/VCn

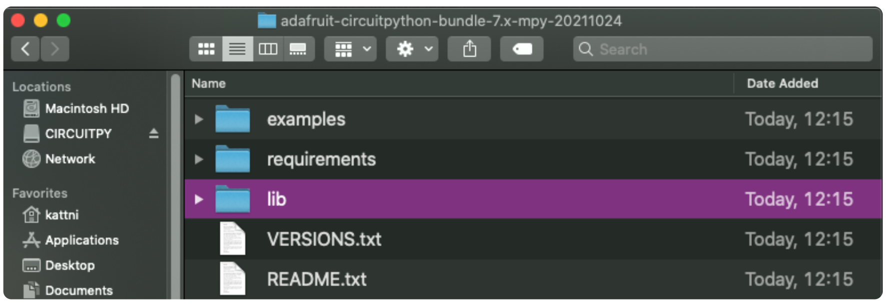
The link takes you to the latest release of the CircuitPython Community Library Bundle on GitHub. There are multiple versions of the bundle available. **Download the bundle version that matches your CircuitPython firmware version.** If you don't know the version, check the version info in **boot_out.txt**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

Understanding the Bundle

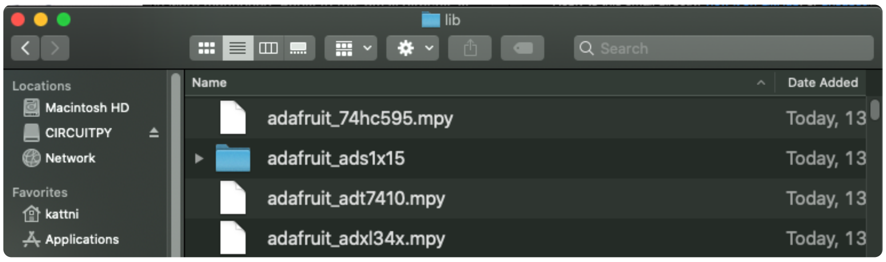
After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.



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



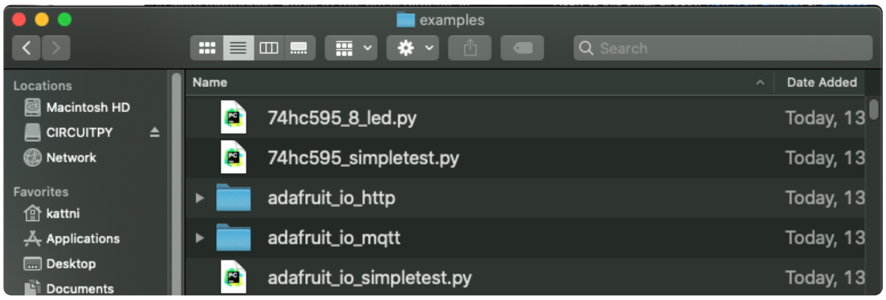
Now open the lib folder. When you open the folder, you'll see a large number of **.mpy** files, and folders.



Example Files

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

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



Copying Libraries to Your Board

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

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

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

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

Understanding Which Libraries to Install

You now know how to load libraries on to your CircuitPython-compatible microcontroller board. You may now be wondering, how do you know which libraries you need to install? Unfortunately, it's not always straightforward. Fortunately, there is an obvious place to start, and a relatively simple way to figure out the rest. First up: the best place to start.

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

- `import library_or_module`

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

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

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

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

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

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

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

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

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

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

```
>>> 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.

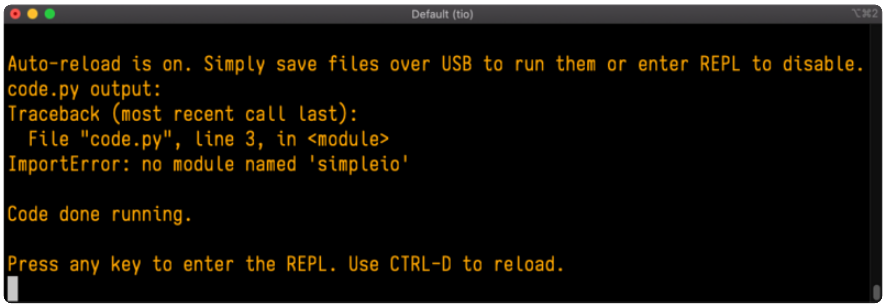
This demonstration will only return an error if you do not have the required library loaded into the **lib** folder on your **CIRCUITPY** drive**.**

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

```
import board
import time
import simpleio
```

```
led = simpleio.DigitalOut(board.LED)

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

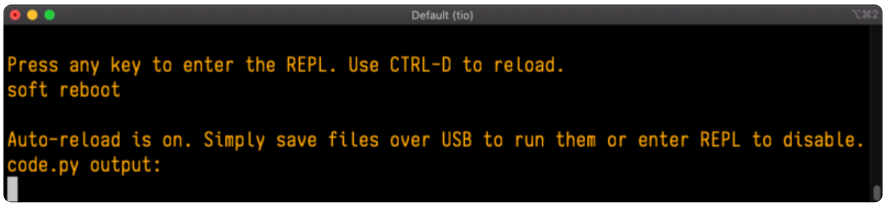
Save this file. Nothing happens to your board. Let's check the serial console to see what's going on.



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

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

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



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

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

Library Install on Non-Express Boards

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

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

Updating CircuitPython Libraries and Examples

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

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

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

CircUp CLI Tool

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

---

Frequently Asked Questions

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

What are some common acronyms to know?

CP or CPy = CircuitPython (https://adafru.it/KJD)
CPC = Circuit Playground Classic (http://adafru.it/3000) (does not run CircuitPython)

CPX = Circuit Playground Express (http://adafru.it/3333)
CPB = Circuit Playground Bluefruit (http://adafru.it/4333)

Using Older Versions

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

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

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

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

Python Arithmetic

Does CircuitPython support floating-point numbers?

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

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

Does CircuitPython support long integers, like regular Python?

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

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

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

Wireless Connectivity

How do I connect to the Internet with CircuitPython?

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

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

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

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

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

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

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

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

Are there other ways to communicate by radio with CircuitPython?

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

Asyncio and Interrupts

Is there asyncio support in CircuitPython?

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

Does CircuitPython support interrupts?

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

Status RGB LED

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

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

Memory Issues

What is a MemoryError?

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

What do I do when I encounter a MemoryError?

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

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

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

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

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

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

How can I create my own .mpy files?

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

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

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

How do I check how much memory I have free?

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

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

Unsupported Hardware

Is ESP8266 or ESP32 supported in CircuitPython? Why not?

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

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

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

Does Feather M0 support WINC1500?

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

Can AVRs such as ATmega328 or ATmega2560 run CircuitPython?

No.

Troubleshooting

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

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

Always Run the Latest Version of CircuitPython and Libraries

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

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

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

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

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

macOS Sonoma 14.x: Disk Errors Writing to CIRCUITPY

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

Here is a shell script to do this remount conveniently (courtesy @czei in GitHub (https://adafru.it/18ea)). Copy the code here into a file named, say, **remount-CIRCUITPY.sh**. Place the file in a directory on your PATH, or in some other convenient place.

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

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

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

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

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

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

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

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

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

Bootloader (boardnameBOOT) Drive Not Present

You may have a different board.

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

MakeCode

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

macOS

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

Windows 10

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

Windows 7 or 8.1

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

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

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

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

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

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

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

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

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

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

CIRCUITPY Drive Does Not Appear or Disappears Quickly

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

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

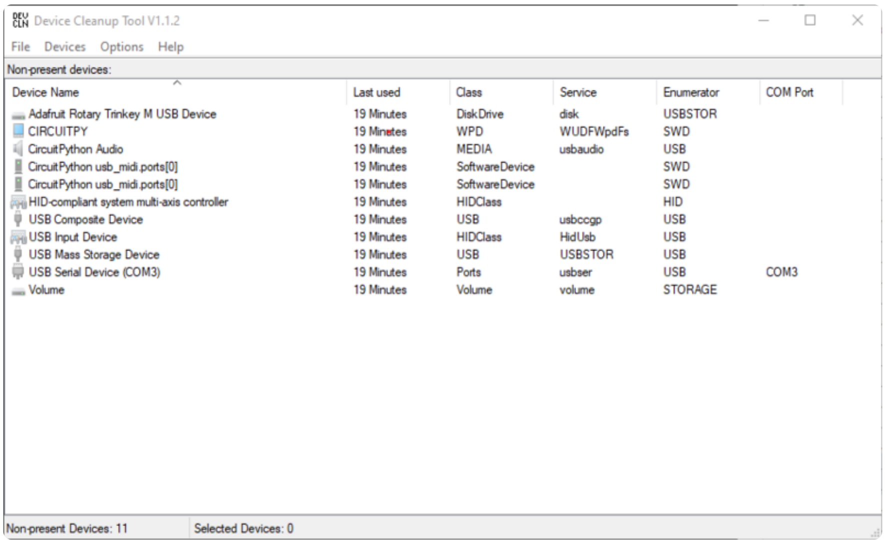
Sophos Endpoint security software can cause CIRCUITPY to disappear (https://adafru.it/ELr) and the BOOT drive to reappear. It is not clear what causes this behavior.

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

Device Errors or Problems on Windows

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

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



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

Serial Console in Mu Not Displaying Anything

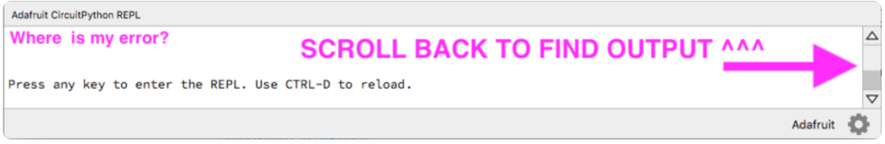
There are times when the serial console will accurately not display anything, such as, when no code is currently running, or when code with no serial output is already running before you open the console. However, if you find yourself in a situation where you feel it should be displaying something like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

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

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



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

More complex errors take even more lines!

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



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

code.py Restarts Constantly

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

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

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

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

```
import supervisor

supervisor.runtime.autoreload = False
```

CircuitPython RGB Status Light

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

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

CircuitPython 7.0.0 and Later

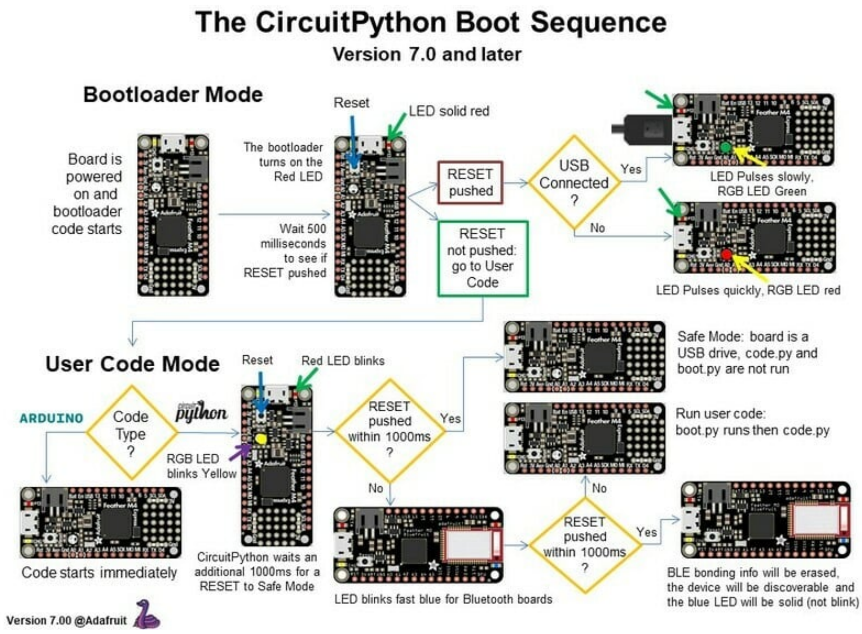
The status LED blinks were changed in CircuitPython 7.0.0 in order to save battery power and simplify the blinks. These blink patterns will occur on single color LEDs when the board does not have any RGB LEDs. Speed and blink count also vary for this reason.

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

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

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

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



CircuitPython 6.3.0 and earlier

Here's what the colors and blinking mean:

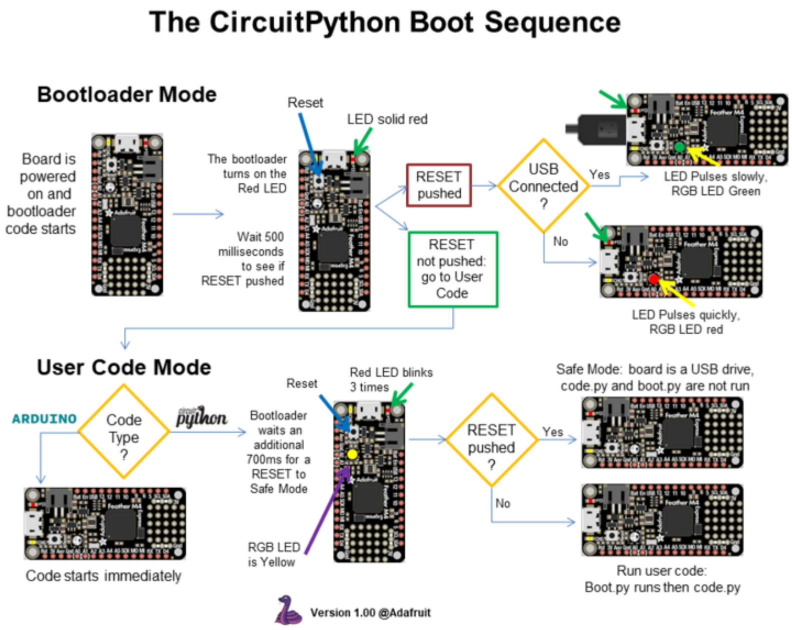
- steady **GREEN**: **code.py** (or **code.txt**, **main.py**, or **main.txt**) is running
- pulsing **GREEN**: **code.py** (etc.) has finished or does not exist
- steady **YELLOW** at start up: (4.0.0-alpha.5 and newer) CircuitPython is waiting for a reset to indicate that it should start in safe mode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

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

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

These are followed by flashes indicating the line number, including place value. **WHITE** flashes are thousands' place, **BLUE** are hundreds' place, **YELLOW** are tens' place, and **CYAN** are one's place. So for example, an error on line 32 would flash

YELLOW three times and then **CYAN** two times. Zeroes are indicated by an extra-long dark gap.



Serial console showing `ValueError: Incompatible .mpy file`

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

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

CIRCUITPY Drive Issues

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

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

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

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

Safe Mode

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

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

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

Entering Safe Mode in CircuitPython 7.x and Later

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

Entering Safe Mode in CircuitPython 6.x

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

In Safe Mode

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

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

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

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

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

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

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

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

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

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

To erase CIRCUITPY: `storage.erase_filesystem()`

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

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

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

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

Erase CIRCUITPY Without Access to the REPL

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

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

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

For the specific boards listed below:

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

1. Download the correct erase file:

Circuit Playground Express

https://adafru.it/AdI

Feather M0 Express

https://adafru.it/AdJ

Feather M4 Express

https://adafru.it/EVK

Metro M0 Express

https://adafru.it/AdK

Metro M4 Express QSPI Eraser

https://adafru.it/EoM

Trellis M4 Express (QSPI)

https://adafru.it/DjD

Grand Central M4 Express (QSPI)

https://adafru.it/DBA

PyPortal M4 Express (QSPI)

https://adafru.it/Eca

Circuit Playground Bluefruit (QSPI)

https://adafru.it/Gnc

Monster M4SK (QSPI)

https://adafru.it/GAN

PyBadge/PyGamer QSPI Eraser.UF2

https://adafru.it/GAO

CLUE_Flash_Erase.UF2

https://adafru.it/Jat

Matrix_Portal_M4_(QSPI).UF2

https://adafru.it/Q5B

RP2040 boards (flash_nuke.uf2)

https://adafru.it/18ed

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

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

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

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

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

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

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

1. Download the erase file:

SAMD21 non-Express Boards

https://adafru.it/VB-

2. Double-click the reset button on the board to bring up the **boardnameBOOT** drive.
3. Drag the erase **.uf2** file to the **boardnameBOOT** drive.
4. The boot LED will start flashing again, and the **boardnameBOOT** drive will

reappear.

5. Drag the appropriate latest release CircuitPython (https://adafru.it/Em8) **.uf2** file to the boardnameBOOT drive.

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

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

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

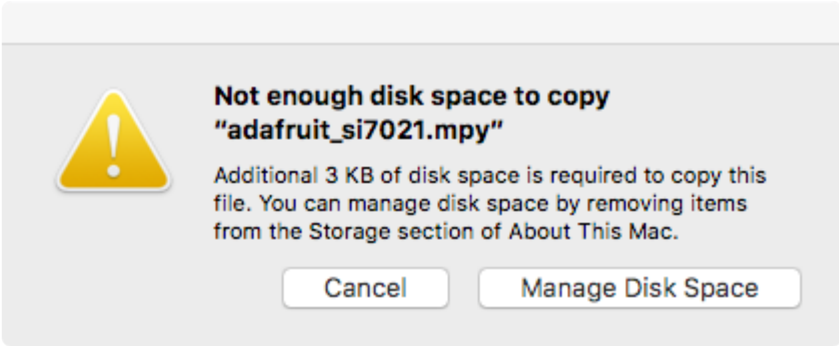
Any SAMD21-based microcontroller that does not have external flash available is considered a SAMD21 non-Express board. Non-Express boards that do **not** have a UF2 bootloader include the Feather M0 Basic Proto, Feather Adalogger, or the Arduino Zero.

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

Running Out of File Space on SAMD21 Non-Express Boards

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

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



Delete something!

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

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

Use tabs

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

On MacOS?

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

Prevent & Remove MacOS Hidden Files

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

```
ls -l /Volumes
```

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

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

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

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

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

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

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

Copy Files on MacOS Without Creating Hidden Files

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

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

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

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

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

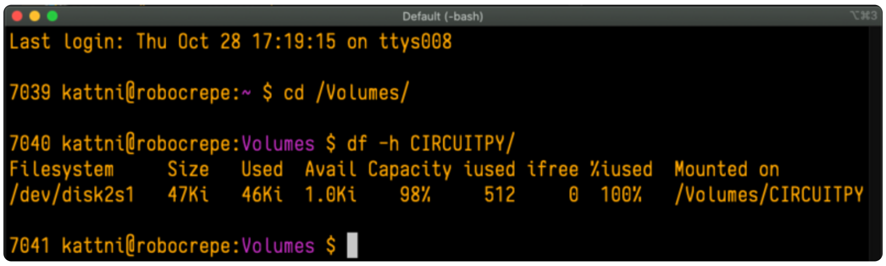
```
cp -rX folder_to_copy /Volumes/CIRCUITPY
```

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

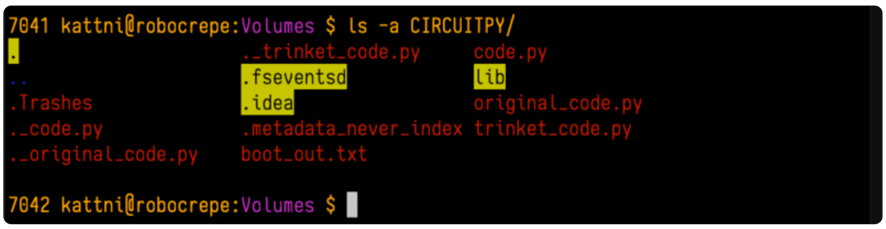
```
# if lib does not exist, you'll create a file named lib !
cp -X file_name.mpy /Volumes/CIRCUITPY/lib
# This is safer, and will complain if a lib folder does not exist.
cp -X file_name.mpy /Volumes/CIRCUITPY/lib/
```

Other MacOS Space-Saving Tips

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



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



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

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

```
7043 kattni@robocrepe:Volumes $ df -h CIRCUITPY/
Filesystem     Size   Used  Avail Capacity iused ifree %iused  Mounted on
/dev/disk2s1   47Ki   34Ki   13Ki    73%     512     0  100%   /Volumes/CIRCUITPY

7044 kattni@robocrepe:Volumes $
```

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

Device Locked Up or Boot Looping

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

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

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

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

Refer to the diagrams above for boot sequence details.

"Uninstalling" CircuitPython

A lot of our boards can be used with multiple programming languages. For example, the Circuit Playground Express can be used with MakeCode, Code.org CS Discoveries, CircuitPython and Arduino.

Maybe you tried CircuitPython and want to go back to MakeCode or Arduino? Not a problem. You can always remove or reinstall CircuitPython whenever you want! Heck, you can change your mind every day!

There is nothing to uninstall. CircuitPython is "just another program" that is loaded onto your board. You simply load another program (Arduino or MakeCode) and it will overwrite CircuitPython.

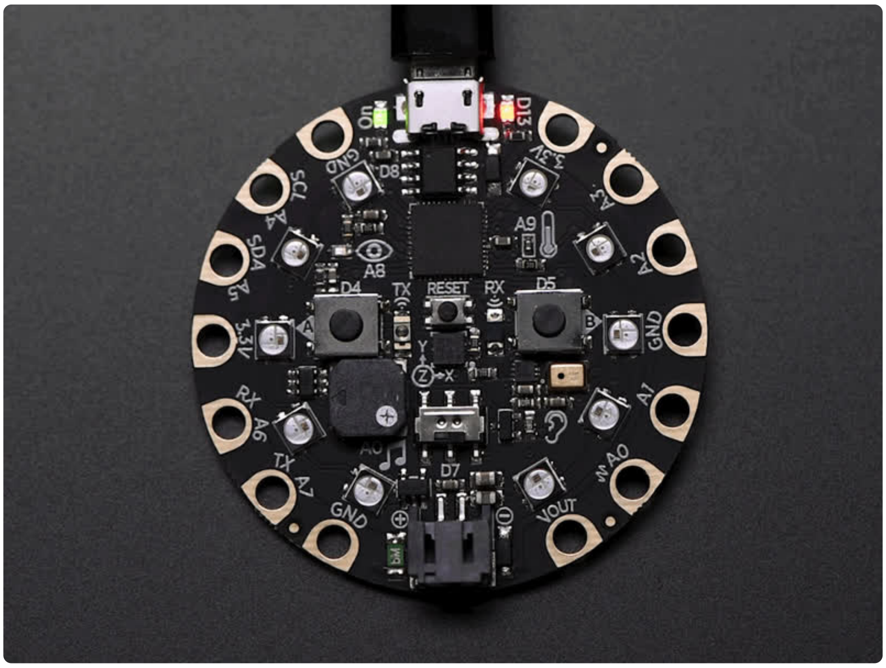
Backup Your Code

Before replacing CircuitPython, don't forget to make a backup of the code you have on the **CIRCUITPY** drive. That means your **code.py** any other files, the **lib** folder etc. You may lose these files when you remove CircuitPython, so backups are key! Just drag the files to a folder on your laptop or desktop computer like you would with any USB drive.

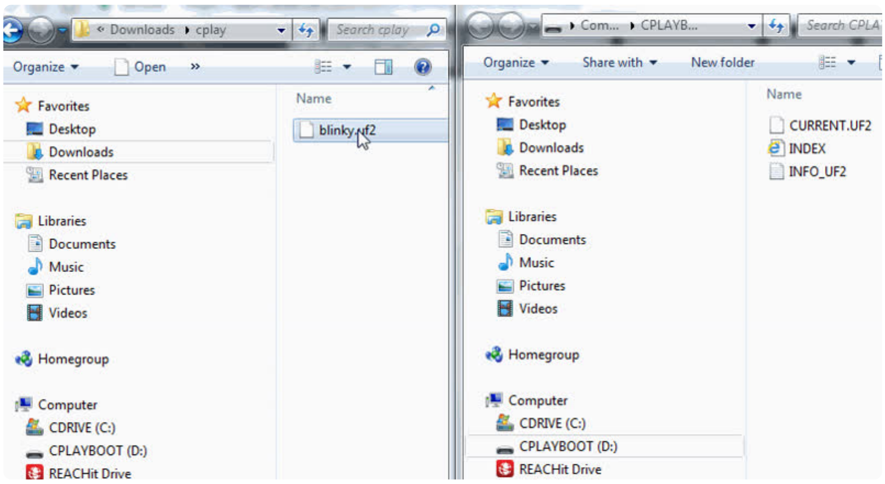
Moving Circuit Playground Express to MakeCode

On the Circuit Playground Express (**this currently does NOT apply to Circuit Playground Bluefruit**), if you want to go back to using MakeCode, it's really easy. Visit makecode.adafruit.com (https://adafru.it/wpC) and find the program you want to upload. Click Download to download the **.uf2** file that is generated by MakeCode.

Now double-click your CircuitPython board until you see the onboard LED(s) turn green and the **...BOOT** directory shows up.



Then find the downloaded MakeCode **.uf2** file and drag it to the **CPLAYBOOT** drive.



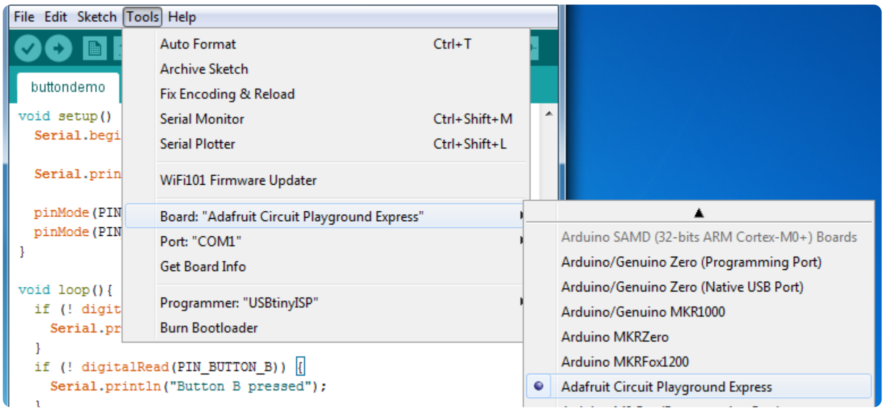
Your MakeCode is now running and CircuitPython has been removed. Going forward you only have to **single click** the reset button to get to **CPLAYBOOT**. This is an idiosyncrasy of MakeCode.

Moving to Arduino

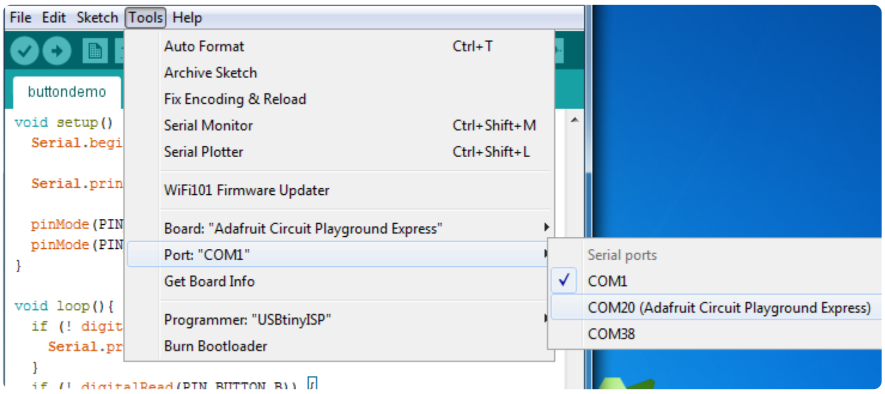
If you want to use Arduino instead, you just use the Arduino IDE to load an Arduino program. Here's an example of uploading a simple "Blink" Arduino program, but you don't have to use this particular program.

Start by plugging in your board, and double-clicking reset until you get the green onboard LED(s).

Within Arduino IDE, select the matching board, say Circuit Playground Express.



Select the correct matching Port:



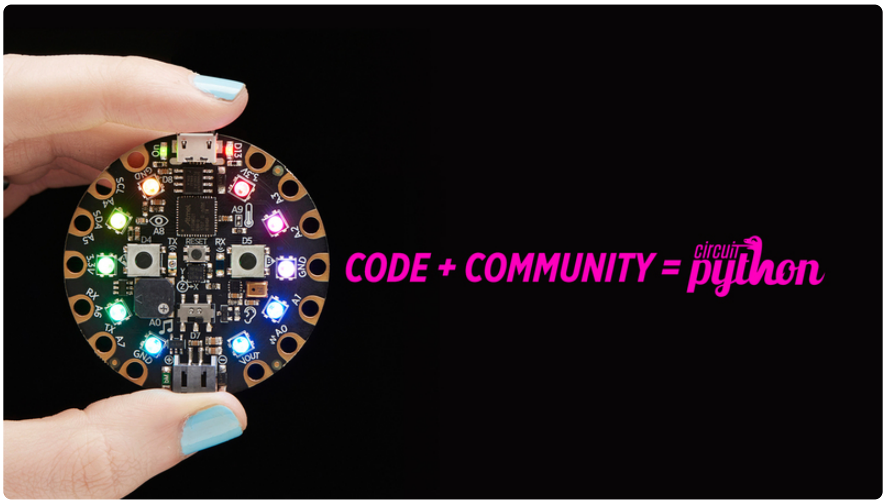
Create a new simple Blink sketch example:

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}
```

Make sure the LED(s) are still green, then click **Upload** to upload Blink. Once it has uploaded successfully, the serial Port will change so **re-select the new Port**!

Once Blink is uploaded you should no longer need to double-click to enter bootloader mode. Arduino will automatically reset when you upload.

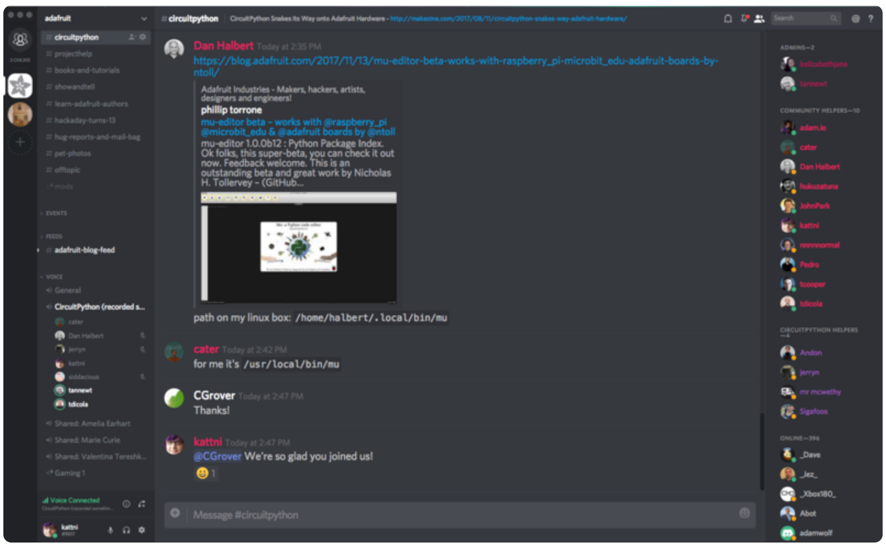
Welcome to the Community!



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

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

Adafruit Discord



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

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

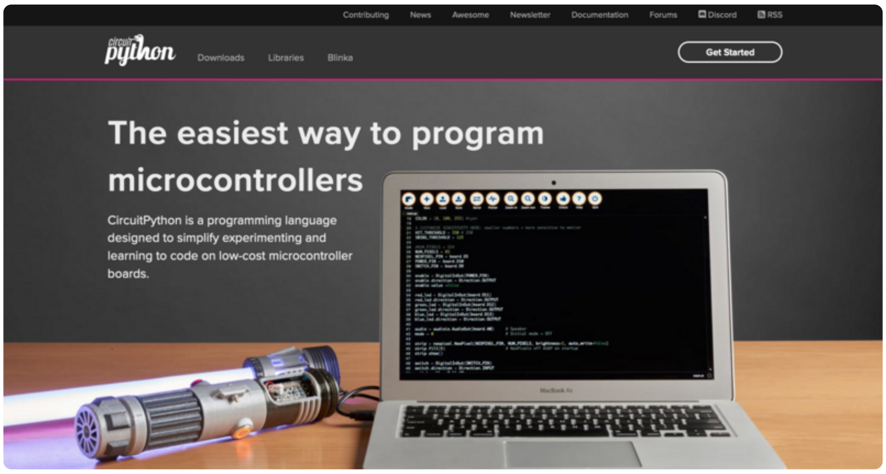
The help with CircuitPython channel is where to go with your CircuitPython questions. #help-with-circuitpython is there for new users and developers alike so feel free to ask a question or post a comment! Everyone of any experience level is welcome to join in on the conversation. Your contributions are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

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

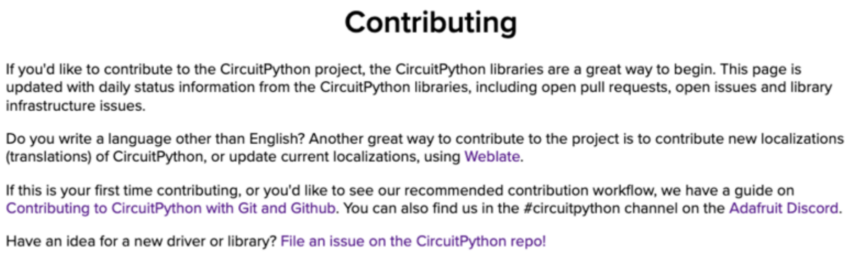
The Adafruit Discord is the 24x7x365 hackerspace that you can bring your granddaughter to.

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

CircuitPython.org



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



Contributing

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

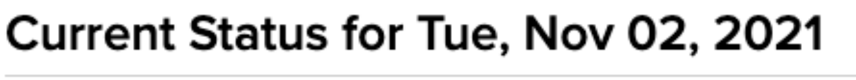
Do you write a language other than English? Another great way to contribute to the project is to contribute new localizations (translations) of CircuitPython, or update current localizations, using Weblate.

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

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

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

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

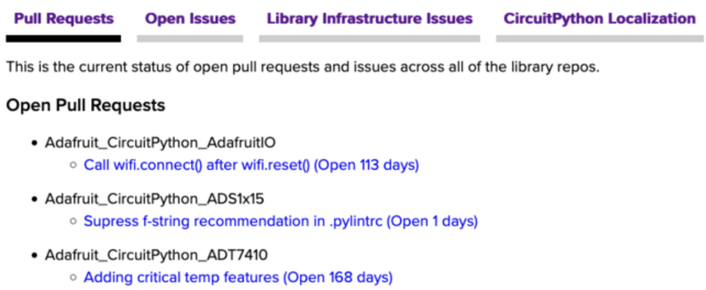


If you submit any contributions to the libraries, and do not see them reflected on the Contributing page, it could be that the job that checks for new updates hasn't yet run for today. Simply check back tomorrow!

Now, a look at the different options.

Pull Requests

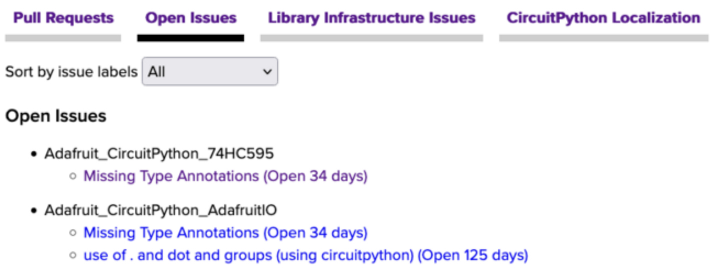
The first tab you'll find is a list of **open pull requests**.



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

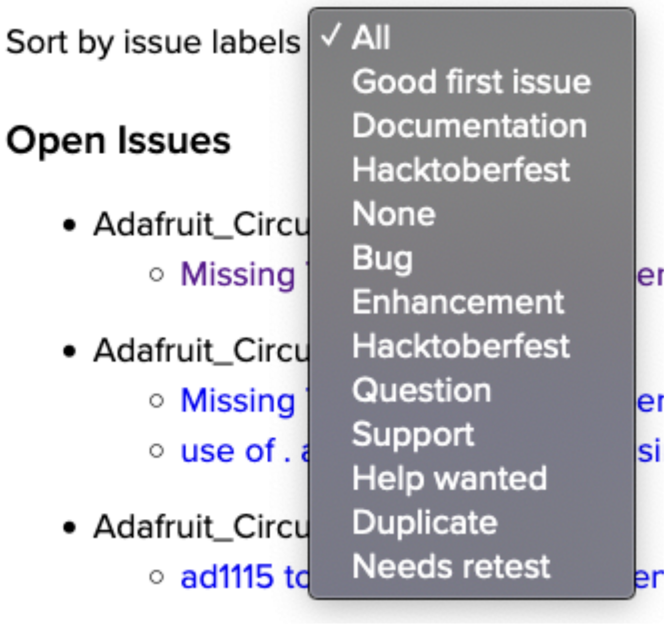
Open Issues

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



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

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



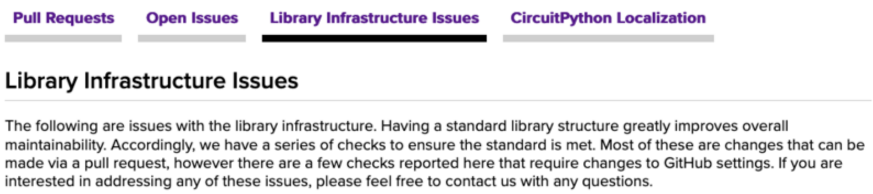
If you're new to everything, new to contributing to open source, or new to contributing to the CircuitPython project, you can choose "Good first issue". Issues with that label are well defined, with a finite scope, and are intended to be easy for someone new to figure out.

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

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

Library Infrastructure Issues

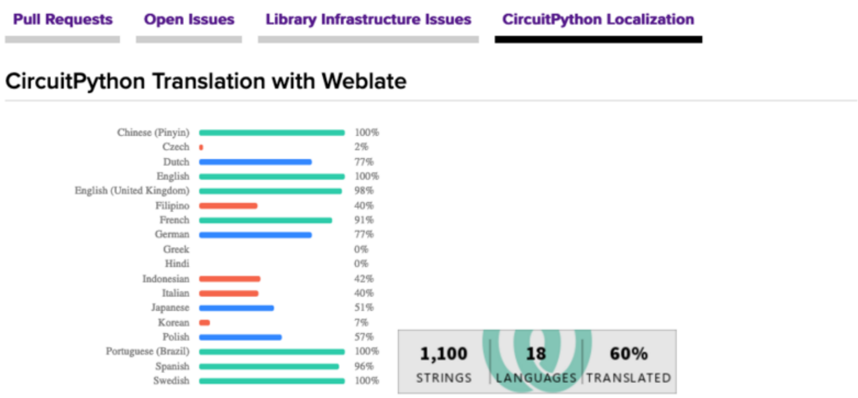
The third tab you'll find is a list of **library infrastructure issues**.



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

CircuitPython Localization

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

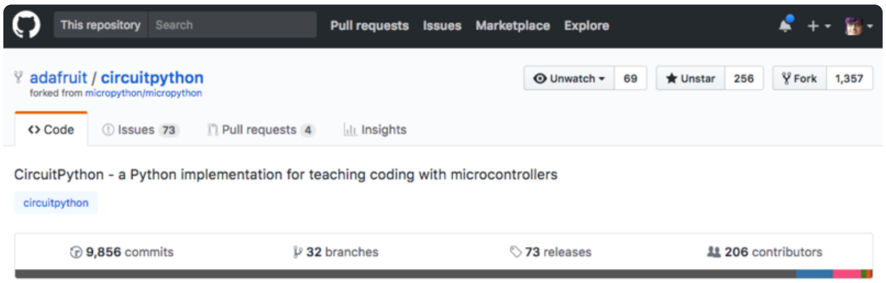


If you speak another language, you can help translate CircuitPython! The translations apply to informational and error messages that are within the CircuitPython core. It means that folks who do not speak English have the opportunity to have these messages shown to them in their own language when using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

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

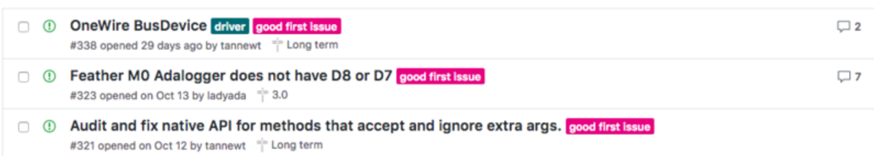
Regardless of your skill level, or how you want to contribute to the CircuitPython project, there is an opportunity available. The Contributing page (https://adafru.it/VD7) is an excellent place to start!

Adafruit GitHub



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

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



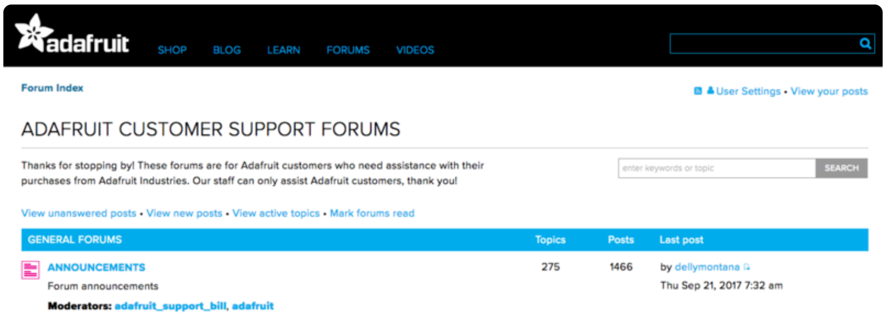
Already experienced and looking for a challenge? Checkout the rest of either issues list and you'll find plenty of ways to contribute. You'll find all sorts of things, from new driver requests, to library bugs, to core module updates. There's plenty of opportunities for everyone at any level!

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

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

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

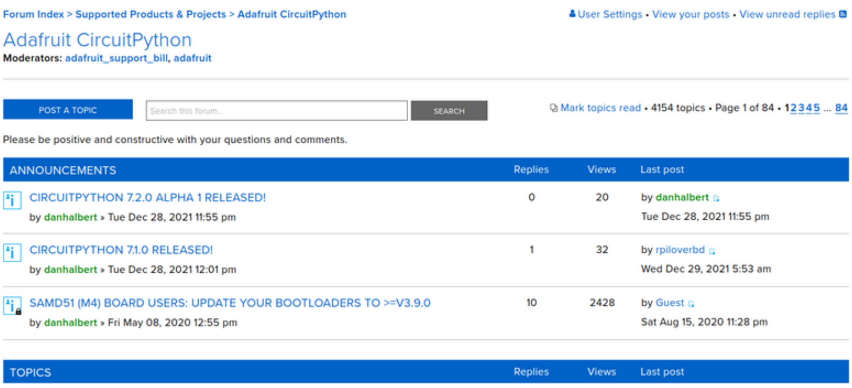
Adafruit Forums



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

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

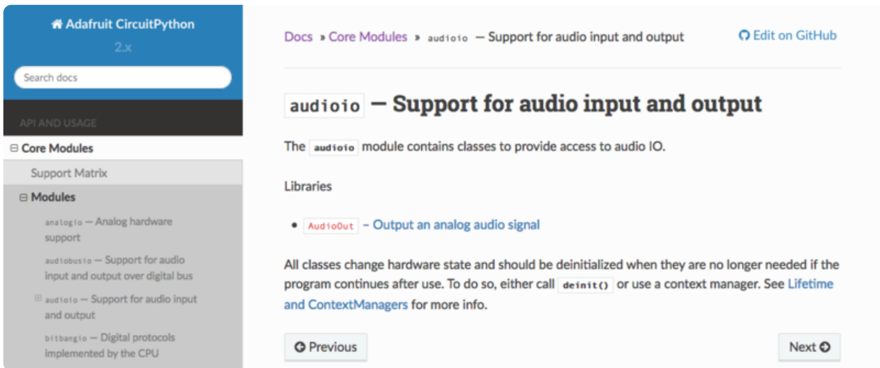
There are forum categories that cover all kinds of topics, including everything Adafruit. The [Adafruit CircuitPython](https://adafru.it/xXA) (https://adafru.it/xXA) category under "Supported Products & Projects" is the best place to post your CircuitPython questions.



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

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

Read the Docs



[Read the Docs](https://adafru.it/Beg) (https://adafru.it/Beg) is a an excellent resource for a more detailed look at the CircuitPython core and the CircuitPython libraries. This is where you'll find things like API documentation and example code. For an in depth look at viewing and understanding Read the Docs, check out the CircuitPython Documentation (https://adafru.it/VFx) page!

Here is blinky:

```
import time
import digitalio
import board

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT
while True:
    led.value = True
    time.sleep(0.1)
    led.value = False
    time.sleep(0.1)
```

CircuitPython Essentials

You've gone through the Welcome to CircuitPython guide (https://adafru.it/cpy-welcome). You've already gotten everything setup, and you've gotten CircuitPython running. Great! Now what? CircuitPython Essentials!

There are a number of core modules built into CircuitPython and commonly used libraries available. This guide will introduce you to these and show you an example of how to use each one.

Each section will present you with a piece of code designed to work with different boards, and explain how to use the code with each board. These examples work with any board designed for CircuitPython, including **Circuit Playground Express**, **Trinket M0**, **Gemma M0**, **QT Py**, **ItsyBitsy M0 Express**, **ItsyBitsy M4 Express, Feather M0 Express, Feather M4 Express, Metro M4 Express, Metro M0 Express, Trellis M4 Express, and Grand Central M4 Express**.

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.

Let's get started learning the CircuitPython Essentials!

CircuitPython Pins and Modules

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

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

CircuitPython Pins

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

`import board`

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

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

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

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

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

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

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

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

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

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

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

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

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

I2C, SPI, and UART

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

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

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

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

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

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

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

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

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

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

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

What Are All the Available Names?

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

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

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

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


```
# SPDX-FileCopyrightText: 2020 anecdata for Adafruit Industries
# SPDX-FileCopyrightText: 2021 Neradoc for Adafruit Industries
# SPDX-FileCopyrightText: 2021-2023 Kattni Rembor for Adafruit Industries
# SPDX-FileCopyrightText: 2023 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

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

board_pins = []
for pin in dir(microcontroller.pin):
    if (isinstance(getattr(microcontroller.pin, pin), microcontroller.Pin) or
        (cyw43 and isinstance(getattr(microcontroller.pin, pin), cyw43.CywPin))):
```

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

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

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

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

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

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

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

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

Microcontroller Pin Names

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

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

CircuitPython Built-In Modules

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

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

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

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

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

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

CircuitPython Built-Ins

CircuitPython comes 'with the kitchen sink' - a lot of the things you know and love about classic Python 3 (sometimes called CPython) already work. There are a few things that don't but we'll try to keep this list updated as we add more capabilities!

This is not an exhaustive list! It's simply some of the many features you can use.

Thing That Are Built In and Work

Flow Control

All the usual `if`, `elif`, `else`, `for`, `while` work just as expected.

Math

`import math` will give you a range of handy mathematical functions.

```
>>> dir(math)
['__name__', 'e', 'pi', 'sqrt', 'pow', 'exp', 'log', 'cos', 'sin',
'tan', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'copysign', 'fabs',
'floor', 'fmod', 'frexp', 'ldexp', 'modf', 'isfinite', 'isinf',
'isnan', 'trunc', 'radians', 'degrees']
```

CircuitPython supports 30-bit wide floating point values so you can use `int` and `float` whenever you expect.

Tuples, Lists, Arrays, and Dictionaries

You can organize data in `()`, `[]`, and `{}` including strings, objects, floats, etc.

Classes, Objects and Functions

We use objects and functions extensively in our libraries so check out one of our many examples like this MCP9808 library (https://adafru.it/BfQ) for class examples.

Lambdas

Yep! You can create function-functions with `lambda` just the way you like em:

```
>>> g = lambda x: x**2
>>> g(8)
64
```

Random Numbers

To obtain random numbers:

```
import random
```

`random.random()` will give a floating point number from `0` to `1.0`.

`random.randint(min, max)` will give you an integer number between `min` and `max`.

CircuitPython Digital In & Out

The first part of interfacing with hardware is being able to manage digital inputs and outputs. With CircuitPython, it's super easy!

This example shows how to use both a digital input and output. You can use a switch input with pullup resistor (built in) to control a digital output - the built in red LED.

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Essentials/CircuitPython_Digitial_In_Out/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Digital In Out example"""
import time
import board
from digitalio import DigitalInOut, Direction, Pull

# LED setup.
led = DigitalInOut(board.LED)
# For QT Py M0. QT Py M0 does not have a D13 LED, so you can connect an external
LED instead.
# led = DigitalInOut(board.SCK)
led.direction = Direction.OUTPUT

# For Gemma M0, Trinket M0, Metro M0 Express, ItsyBitsy M0 Express, Itsy M4
Express, QT Py M0
switch = DigitalInOut(board.D2)
# switch = DigitalInOut(board.D5)  # For Feather M0 Express, Feather M4 Express
# switch = DigitalInOut(board.D7)  # For Circuit Playground Express
switch.direction = Direction.INPUT
switch.pull = Pull.UP

while True:
    # We could also do "led.value = not switch.value"!
    if switch.value:
        led.value = False
    else:
        led.value = True

    time.sleep(0.01)  # debounce delay
```

Note that we made the code a little less "Pythonic" than necessary. The `if/else` block could be replaced with a simple `led.value = not switch.value` but we wanted to make it super clear how to test the inputs. The interpreter will read the digital input when it evaluates `switch.value`.

For **Gemma M0, Trinket M0, Metro M0 Express, Metro M4 Express, ItsyBitsy M0 Express, ItsyBitsy M4 Express**, no changes to the initial example are needed.

Note: To "comment out" a line, put a # and a space before it. To "uncomment" a line, remove the # + space from the beginning of the line.

For **Feather M0 Express and Feather M4 Express**, comment out `switch = DigitalInOut(board.D2)` (and/or `switch = DigitalInOut(board.D7)` depending on what changes you already made), and uncomment `switch = DigitalInOut(board.D5)`.

For **Circuit Playground Express**, you'll need to comment out `switch = DigitalInOut(board.D2)` (and/or `switch = DigitalInOut(board.D5)` depending on what changes you already made), and uncomment `switch = DigitalInOut(board.D7)`.

> QT Py M0 does not have a little red LED built in. Therefore, you must connect an external LED for this example to work. See below for a wiring diagram illustrating how to connect an external LED to a QT Py M0.

For **QT Py M0**, you'll need to comment out `led = DigitalInOut(board.LED)` and uncomment `led = DigitalInOut(board.SCK)`. The switch code remains the same.

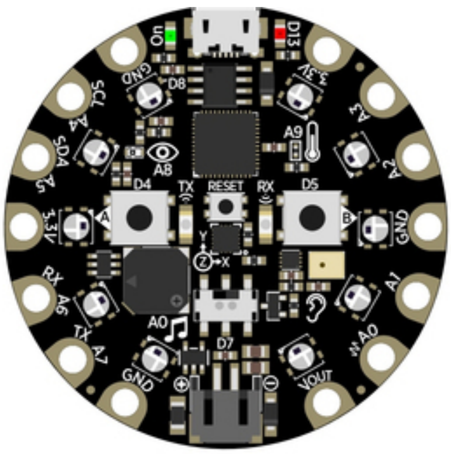
To find the pin or pad suggested in the code, see the list below. For the boards that require wiring, wire up a switch (also known as a tactile switch, button or push-button), following the diagram for guidance. Press or slide the switch, and the onboard red LED will turn on and off.

Note that on the M0/SAMD based CircuitPython boards, at least, you can also have internal pulldowns with **Pull.DOWN** and if you want to turn off the pullup/pulldown just assign **switch.pull** = **None.**

Find the pins!

The list below shows each board, explains the location of the Digital pin suggested for use as input, and the location of the D13 LED.

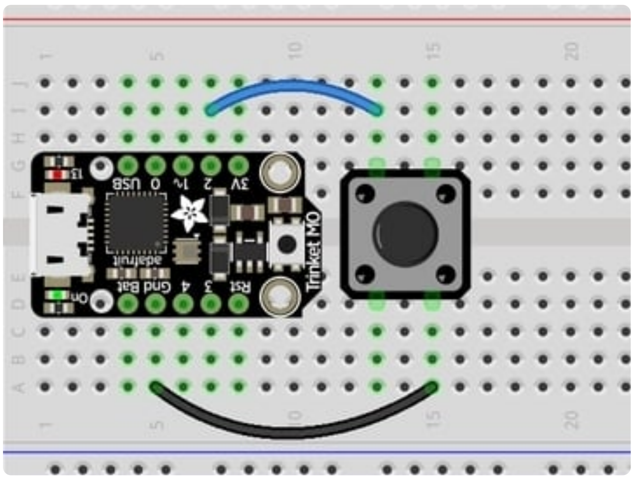
Circuit Playground Express



We're going to use the switch, which is pin D7, and is located between the battery connector and the reset switch on the board. The LED is labeled D13 and is located next to the USB micro port.

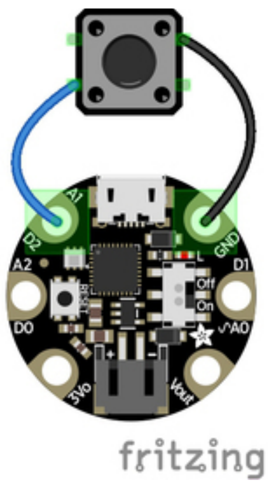
To use D7, comment out the current pin setup line, and uncomment the line labeled for Circuit Playground Express. See the details above!

Trinket M0



D2 is connected to the blue wire, labeled "2", and located between "3V" and "1" on the board. The LED is labeled "13" and is located next to the USB micro port.

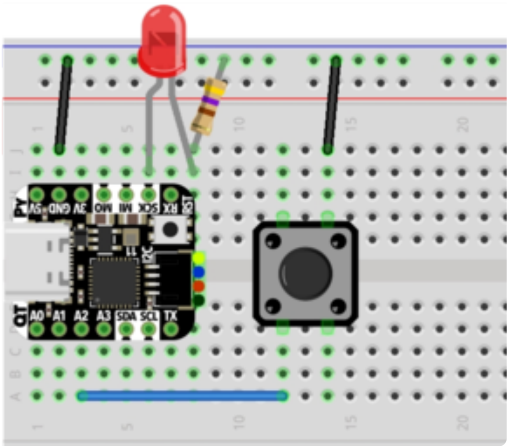
Gemma M0



D2 is an alligator-clip-friendly pad labeled both "D2" and "A1", shown connected to the blue wire, and is next to the USB micro port. The LED is located next to the "GND" label on the board, above the "On/Off" switch.

Use alligator clips to connect your switch to your Gemma M0!

QT Py M0



D2 is labeled A2, shown connected to the blue wire, and is near the USB port between A1 and A3.

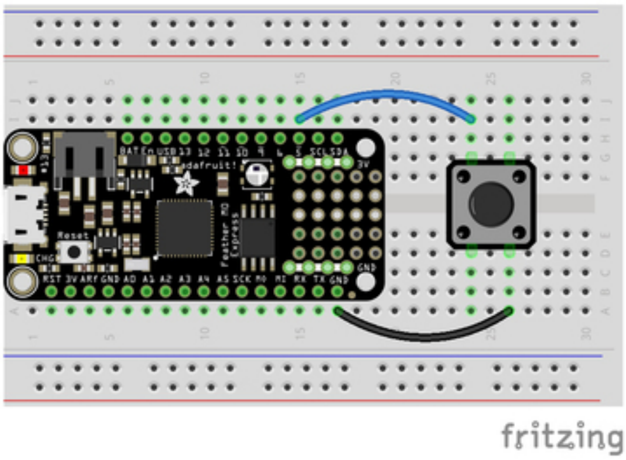
There is no little red LED built-in to the QT Py M0. Therefore, you must connect an external LED for this example to work.

To wire up an external LED:

LED + to **QT Py SCK**
LED - to **470Ω resistor**
470Ω resistor to **QT Py GND**
The button and the LED share the same GND pin.

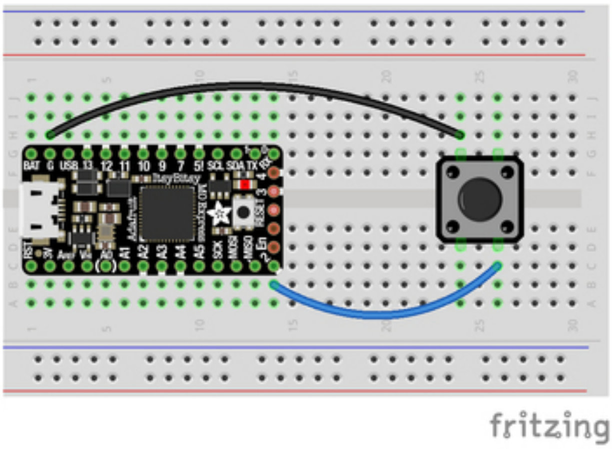
To use the external LED, comment out the current LED setup line, and uncomment the line labeled for QT Py M0. See the details above!

Feather M0 Express and Feather M4 Express



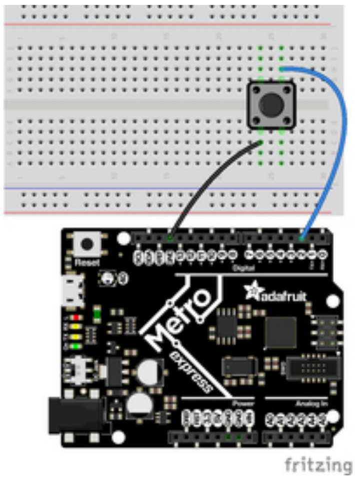
D5 is labeled "5" and connected to the blue wire on the board. The LED is labeled "#13" and is located next to the USB micro port.

To use D5, comment out the current pin setup line, and uncomment the line labeled for Feather M0 Express. See the details above!



ItsyBitsy M0 Express and ItsyBitsy M4 Express

D2 is labeled "2", located between the "MISO" and "EN" labels, and is connected to the blue wire on the board. The LED is located next to the reset button between the "3" and "4" labels on the board.



Metro M0 Express and Metro M4 Express

D2 is located near the top left corner, and is connected to the blue wire. The LED is labeled "L" and is located next to the USB micro port.

Read the Docs

For a more in-depth look at what `digitalio` can do, check out [the `DigitalInOut` page in Read the Docs](https://adafru.it/C4c) (https://adafru.it/C4c).

CircuitPython Analog In

This example shows you how you can read the analog voltage on the A1 pin on your board.

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/CircuitPython_Analogin/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Analog In example"""
import time
import board
from analogio import AnalogIn

analog_in = AnalogIn(board.A1)


def get_voltage(pin):
    return (pin.value * 3.3) / 65536


while True:
    print((get_voltage(analog_in),))
    time.sleep(0.1)
```

Make sure you're running the latest CircuitPython! If you are not, you may run into an error: "AttributeError: 'module' object has no attribute 'A1'". If you receive this error, first make sure you're running the latest version of CircuitPython!

Creating the analog input

`analog1in = AnalogIn(board.A1)`

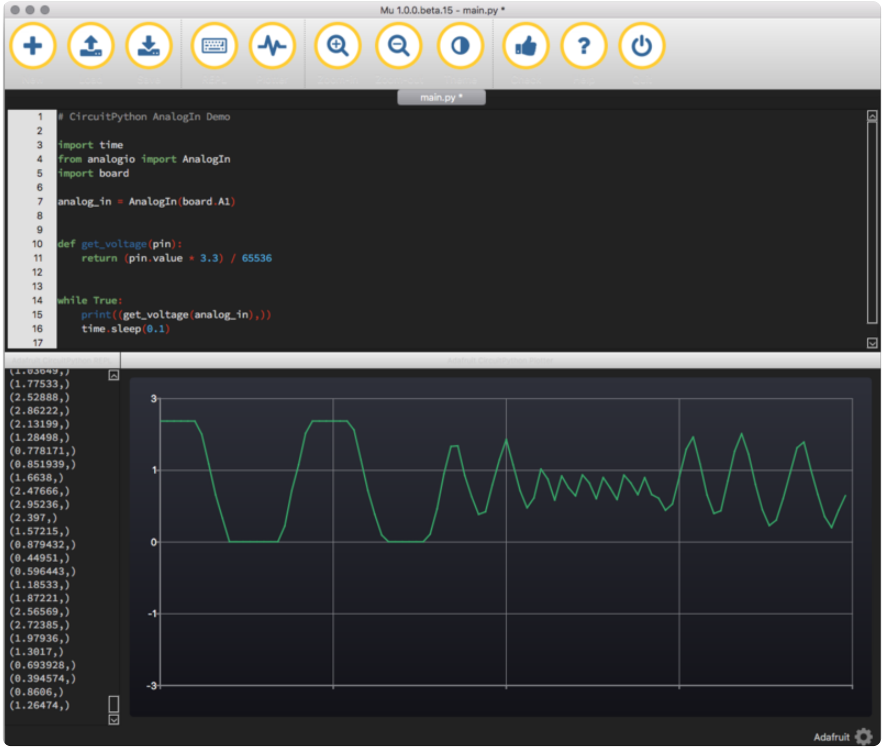
Creates an object and connects the object to A1 as an analog input.

`get_voltage` Helper

`get_voltage(pin)` is our little helper program. By default, analog readings will range from 0 (minimum) to 65535 (maximum). This helper will convert the 0-65535 reading from `pin.value` and convert it a 0-3.3V voltage reading.

Main Loop

The main loop is simple. It `prints` out the voltage as floating point values by calling `get_voltage` on our analog object. Connect to the serial console to see the results.



Changing It Up

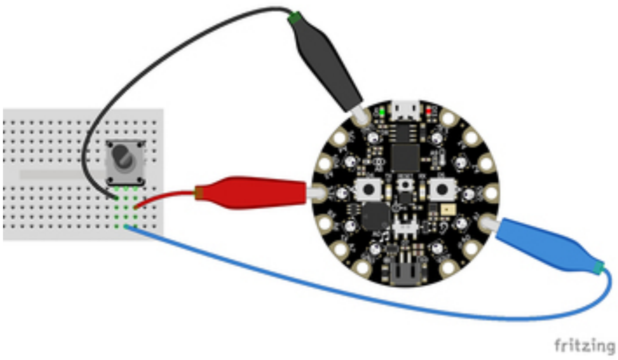
By default the pins are floating so the voltages will vary. While connected to the serial console, try touching a wire from **A1** to the **GND** pin or **3Vo** pin to see the voltage change.

You can also add a potentiometer to control the voltage changes. From the potentiometer to the board, connect the **left pin** to **ground**, the **middle pin** to **A1**, and the **right pin** to **3V**. If you're using Mu editor, you can see the changes as you rotate the potentiometer on the plotter like in the image above! (Click the Plotter icon at the top of the window to open the plotter.)

> When you turn the knob of the potentiometer, the wiper rotates left and right, increasing or decreasing the resistance. This, in turn, changes the analog voltage level that will be read by your board on A1.

Wire it up

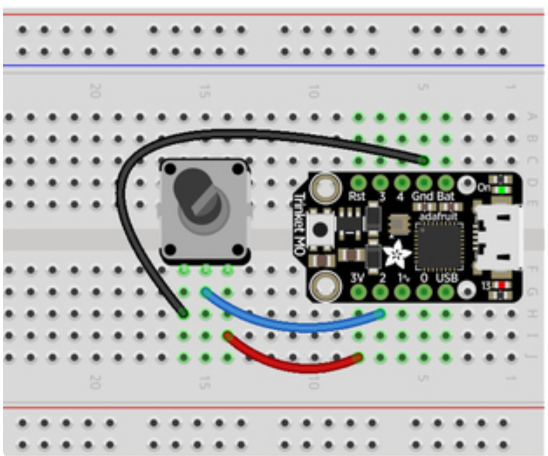
The list below shows wiring diagrams to help find the correct pins and wire up the potentiometer, and provides more information about analog pins on your board!



Circuit Playground Express

A1 is located on the right side of the board. There are multiple ground and 3V pads (pins).

Your board has 7 analog pins that can be used for this purpose. For the full list, see the pinout page (https://adafru.it/AM9) on the main guide.

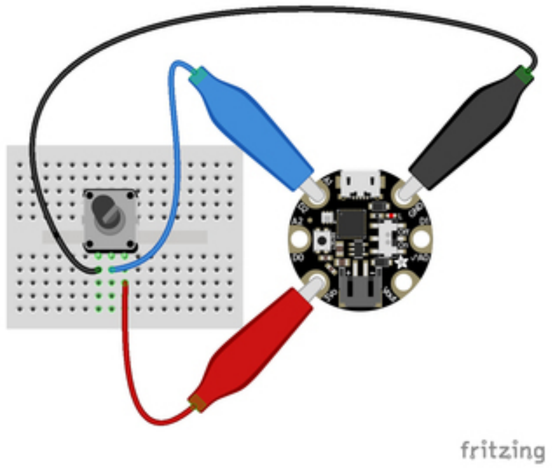


Trinket M0

A1 is labeled as 2! It's located between "1~" and "3V" on the same side of the board as the little red LED. Ground is located on the opposite side of the board. 3V is located next to 2, on the same end of the board as the reset button.

You have 5 analog pins you can use. For the full list, see the pinouts page (https://adafru.it/AMd) on the main guide.

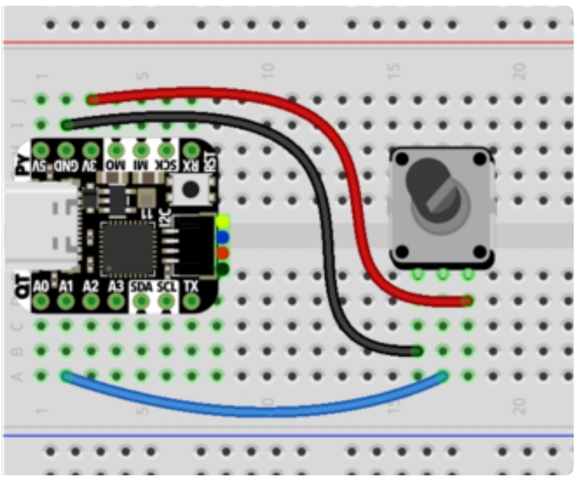
Gemma M0



A1 is located near the top of the board of the board to the left side of the USB Micro port. Ground is on the other side of the USB port from A1. 3V is located to the left side of the battery connector on the bottom of the board.

Your board has 3 analog pins. For the full list, see the [pinout page (https://adafru.it/AMa)](https://adafru.it/AMa) on the main guide.

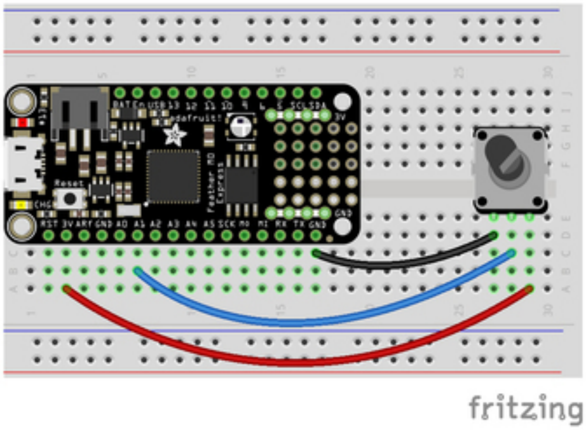
QT Py M0



A1, shown connected to the blue wire, is near the USB port between A0 and A2. Ground is on the opposite side of the QT Py, near the USB port, between 3V and 5V. 3V is the next pin, between GND and MO.

Your board has 10 analog pins. For the full list, see the [pinouts page (https://adafru.it/OeY)](https://adafru.it/OeY) in the main guide.

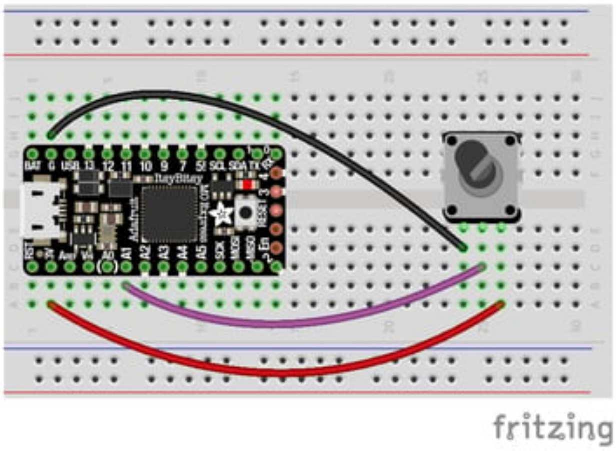
Feather M0 Express and Feather M4 Express



A1 is located along the edge opposite the battery connector. There are multiple ground pins. 3V is located along the same edge as A1, and is next to the reset button.

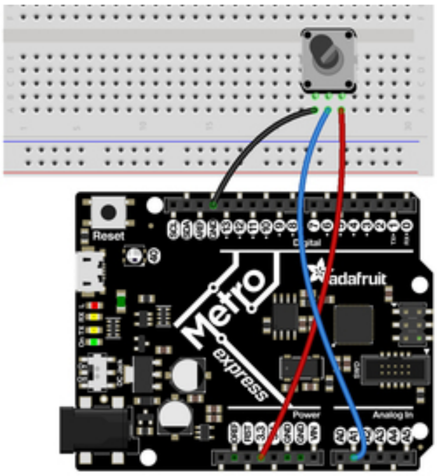
Your board has 6 analog pins you can use. For the full list, see the pinouts page (https://adafru.it/AMc) on the main guide.

ItsyBitsy M0 Express and ItsyBitsy M4 Express



A1 is located in the middle of the board, near the "A" in "Adafruit". Ground is labled "G" and is located next to "BAT", near the USB Micro port. 3V is found on the opposite side of the USB port from Ground, next to RST.

You have 6 analog pins you can use. For a full list, see the pinouts page (https://adafru.it/BMg) on the main guide.



A1 is located on the same side of the board as the barrel jack. There are multiple ground pins available. 3V is labeled "3.3" and is located in the center of the board on the same side as the barrel jack (and as A1).

Your **Metro M0 Express** board has 6 analog pins you can use. For the full list, see the pinouts page (https://adafru.it/AMb) on the main guide.

Your **Metro M4 Express** board has 6 analog pins you can use. For the full list, see the pinouts page (https://adafru.it/B1O) on the main guide.

Reading Analog Pin Values

The `get_voltage()` helper used in the potentiometer example above reads the raw analog pin value and converts it to a voltage level. You can, however, directly read an analog pin value in your code by using `pin.value`. For example, to simply read the raw analog pin value from the potentiometer, you would run the following code:

```
import time
import board
from analogio import AnalogIn

analog_in = AnalogIn(board.A1)

while True:
    print(analog_in.value)
    time.sleep(0.1)
```

This works with any analog pin or input. Use the `<pin_name>.value` to read the raw value and utilise it in your code.

CircuitPython Analog Out

This example shows you how you can use the DAC (Digital to Analog Converter).

Many chips do not have a DAC, including Espressif ESP32-S3, nRF52840, and various others. If `AnalogOut` is not available, its use will raise `NotImplementedError.` Check the board guide for your board.

A0 is the only true analog output on the M0 boards. No other pins do true analog output!

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/CircuitPython_AnalogOut/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Analog Out example"""
import board
from analogio import AnalogOut

analog_out = AnalogOut(board.A0)

while True:
    # Count up from 0 to 65535, with 64 increment
    # which ends up corresponding to the DAC's 10-bit range
    for i in range(0, 65535, 64):
        analog_out.value = i
```

Creating an analog output

`analog_out = AnalogOut(A0)`

Creates an object `analog_out` and connects the object to **A0**, the only DAC pin available on both the M0 and the M4 boards. (The M4 has two, A0 and A1.)

Setting the analog output

The DAC on the SAMD21 is a 10-bit output, from 0-3.3V. So in theory you will have a resolution of 0.0032 Volts per bit. To allow CircuitPython to be general-purpose enough that it can be used with chips with anything from 8 to 16-bit DACs, the DAC takes a 16-bit value and divides it down internally.

For example, writing 0 will be the same as setting it to 0 - 0 Volts out.

Writing 5000 is the same as setting it to 5000 / 64 = 78, and 78 / 1024 * 3.3V = 0.25V output.

Writing 65535 is the same as 1023 which is the top range and you'll get 3.3V output

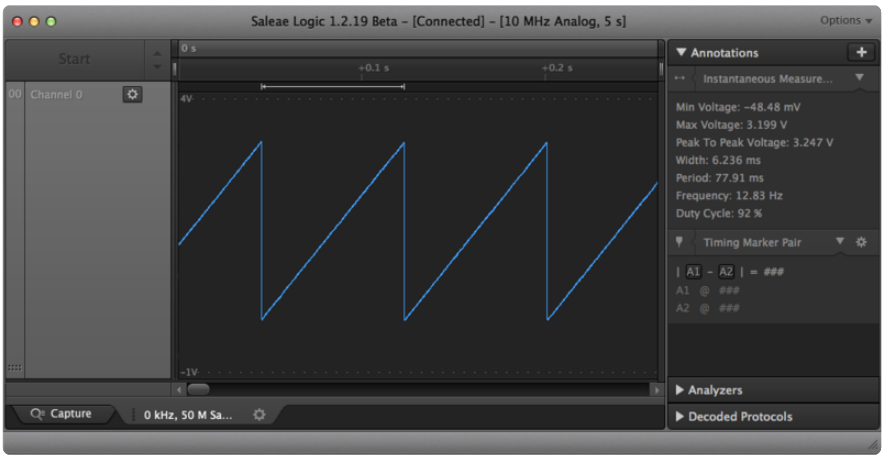
Main Loop

The main loop is fairly simple, it goes through the entire range of the DAC, from 0 to 65535, but increments 64 at a time so it ends up clicking up one bit for each of the 10-bits of range available.

CircuitPython is not terribly fast, so at the fastest update loop you'll get 4 Hz. The DAC isn't good for audio outputs as-is.

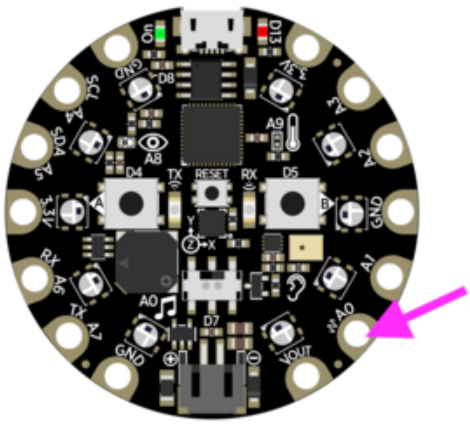
Express boards like the Circuit Playground Express, Metro M0 Express, ItsyBitsy M0 Express, ItsyBitsy M4 Express, Metro M4 Express, Feather M4 Express, or Feather M0 Express have more code space and can perform audio playback capabilities via the DAC. QT Py M0, Gemma M0 and Trinket M0 cannot!

Check out the Audio Out section of this guide (https://adafru.it/BRj) for examples!



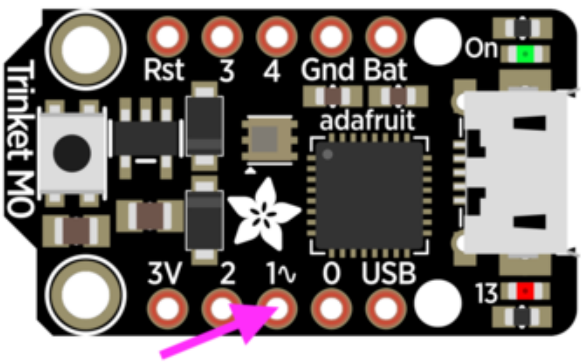
Find the pin

Use the diagrams below to find the A0 pin marked with a magenta arrow!



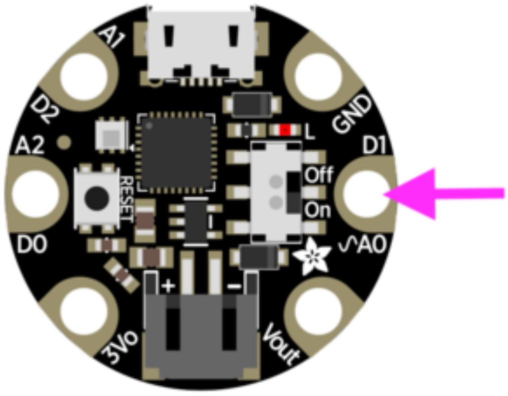
Circuit Playground Express

A0 is located between VOUT and A1 near the battery port.



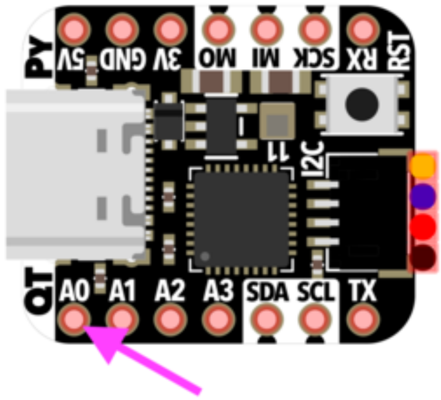
Trinket M0

A0 is labeled "1~" on Trinket! A0 is located between "0" and "2" towards the middle of the board on the same side as the red LED.



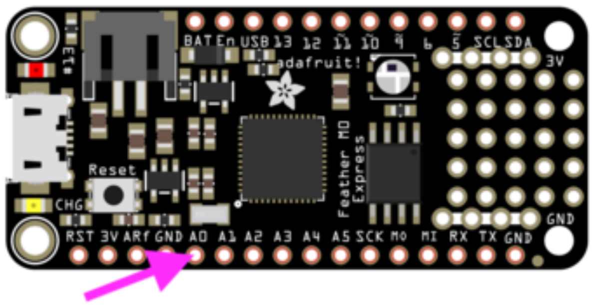
Gemma M0

A0 is located in the middle of the right side of the board next to the On/Off switch.



QT Py M0

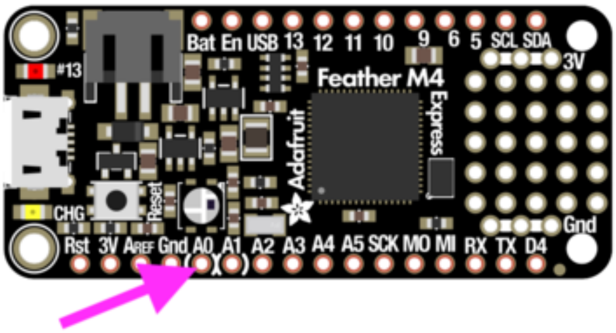
A0 is located next to the USB port, by the "QT" label on the board silk.



Feather M0 Express

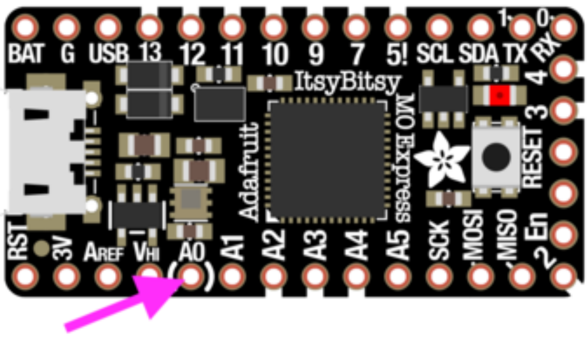
A0 is located between GND and A1 on the opposite side of the board from the battery connector, towards the end with the Reset button.

Feather M4 Express



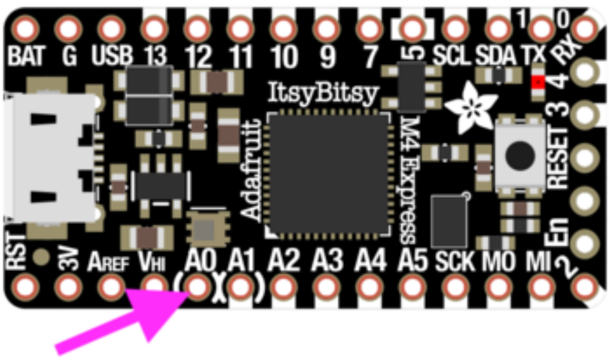
A0 is located between GND and A1 on the opposite side of the board from the battery connector, towards the end with the Reset button, and the pin pad has left and right white parenthesis markings around it

ItsyBitsy M0 Express

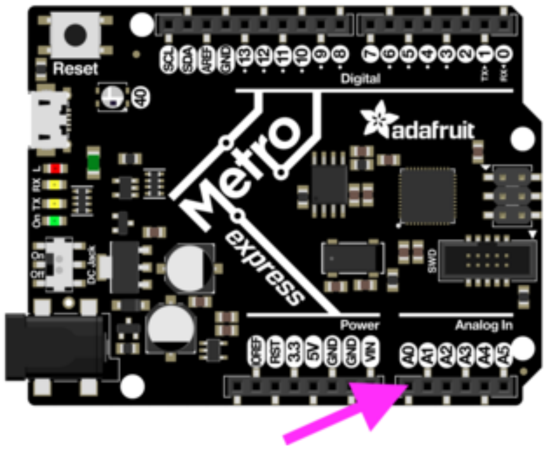


A0 is located between VHI and A1, near the "A" in "Adafruit", and the pin pad has left and right white parenthesis markings around it.

ItsyBitsy M4 Express

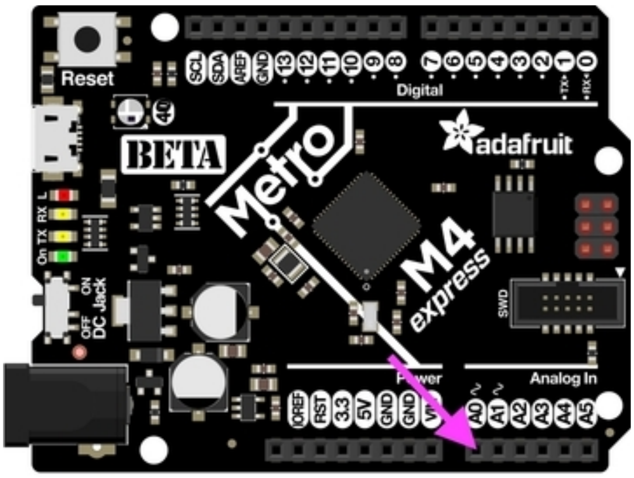


A0 is located between VHI and A1, and the pin pad has left and right white parenthesis markings around it.



Metro M0 Express

A0 is between VIN and A1, and is located along the same side of the board as the barrel jack adapter towards the middle of the headers found on that side of the board.



Metro M4 Express

A0 is between VIN and A1, and is located along the same side of the board as the barrel jack adapter towards the middle of the headers found on that side of the board.

On the Metro M4 Express, there are TWO true analog outputs: A0 and A1.

CircuitPython Audio Out

CircuitPython comes with `audioio`, which provides built-in audio output support. You can play generated tones. You can also play, pause and resume wave files. You can have 3V-peak-to-peak analog output or I2S digital output. In this page we will show using analog output.

This is great for all kinds of projects that require sound, like a tone piano or anything where you'd like to add audio effects!

QT Py M0, Hallowing M0, Trinket M0 and Gemma M0 do not support audioio! You must use an M0 Express, M4 Express, nRF52840 etc board for this.

The first example will show you how to generate a tone and play it using a button. The second example will show you how to play, pause, and resume a wave file using a button to resume. Both will play the audio through an audio jack. The default volume on both of these examples is painfully high through headphones. So, we've added a potentiometer and included some code in the tone generation example to control volume.

In our code, we'll use pin A0 for our audio output, as this is the only DAC pin available on every Express board. The M0 Express boards have audio output on A0. The M4 Express boards have two audio output pins, A0 and A1, however we'll be using only A0 in this guide.

Play a Tone

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/CircuitPython_Audio_Out_Tone/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Audio Out tone example"""
import time
import array
import math
import board
import digitalio
from audiocore import RawSample

try:
    from audioio import AudioOut
except ImportError:
    try:
        from audiopwmio import PWMAudioOut as AudioOut
    except ImportError:
        pass  # not always supported by every board!

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

```
tone_volume = 0.1  # Increase this to increase the volume of the tone.
frequency = 440  # Set this to the Hz of the tone you want to generate.
length = 8000 // frequency
sine_wave = array.array("H", [0] * length)
for i in range(length):
    sine_wave[i] = int((1 + math.sin(math.pi * 2 * i / length)) * tone_volume * (2
** 15 - 1))

audio = AudioOut(board.A0)
sine_wave_sample = RawSample(sine_wave)

while True:
    if not button.value:
        audio.play(sine_wave_sample, loop=True)
        time.sleep(1)
        audio.stop()
```

First we create the button object, assign it to pin `A1`, and set it as an input with a pull-up. Even though the button switch involves `digitalio`, we're using an A-pin so that the same setup code will work across all the boards.

Since the default volume was incredibly high, we included a `tone_volume` variable in the sine wave code. You can use the code to control the volume by increasing or decreasing this number to increase or decrease the volume. You can also control volume with the potentiometer by rotating the knob.

To set the frequency of the generated tone, change the number assigned to the `frequency` variable to the Hz of the tone you'd like to generate.

Then, we generate one period of a sine wave with the `math.sin` function, and assign it to `sine_wave`.

Next, we create the audio object, and assign it to pin `A0`.

We create a sample of the sine wave by using `RawSample` and providing the `sine_wave` we created.

Inside our loop, we check to see if the button is pressed. The button has two states `True` and `False`. The `button.value` defaults to the `True` state when not pressed. So, to check if it has been pressed, we're looking for the `False` state. So, we check to see `if not button.value` which is the equivalent of `not True`, or `False`.

Once the button is pressed, we `play` the sample we created and we loop it. The `time.sleep(1)` tells it to loop (play) for 1 second. Then we `stop` it after 1 second is up. You can increase or decrease the length of time it plays by increasing or decreasing the number of seconds provided to `time.sleep()`. Try changing it from `1` to `0.5`. Now try changing it to `2`. You can change it to whatever works for you!

That's it!

Play a Wave File

You can use any supported wave file you like. CircuitPython supports **mono or stereo**, at **22 KHz sample rate** (or less) and **16-bit** WAV format. The M0 boards support ONLY MONO. The reason for mono is that there's only one analog output on those boards! The M4 boards support stereo as they have two outputs. The 22 KHz or less because the circuitpython can't handle more data than that (and also it will not sound much better) and the DAC output is 10-bit so anything over 16-bit will just take up room without better quality.

Since the WAV file must fit on the CircuitPython file system, it must be under 2 MB.

CircuitPython does not support OGG. Just WAV and MP3!

We have a detailed guide on how to generate WAV files here (https://adafru.it/s8f).

We've included the one we used here. Download it and copy it to your board.

StreetChicken.wav

https://adafru.it/BQF

We're going to play the wave file for 6 seconds, pause it, wait for a button to be pressed, and then resume the file to play through to the end. Then it loops back to the beginning and starts again! Let's take a look.

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/CircuitPython_Audio_Out_Wave/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Audio Out WAV example"""
import time
import board
import digitalio
from audiocore import WaveFile

try:
    from audioio import AudioOut
except ImportError:
    try:
        from audiopwmio import PWMAudioOut as AudioOut
    except ImportError:
        pass  # not always supported by every board!

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

wave_file = open("StreetChicken.wav", "rb")
wave = WaveFile(wave_file)
audio = AudioOut(board.A0)

while True:
    audio.play(wave)

    # This allows you to do other things while the audio plays!
    t = time.monotonic()
    while time.monotonic() - t < 6:
        pass

    audio.pause()
    print("Waiting for button press to continue!")
    while button.value:
        pass
    audio.resume()
    while audio.playing:
        pass
    print("Done!")
```

First we create the button object, assign it to pin `A1`, and set it as an input with a pull-up.

Next we then open the file, `"StreetChicken.wav"` as a readable binary and store the file object in `wave_file` which is what we use to actually read audio from: `wave_file = open("StreetChicken.wav", "rb")`.

Now we will ask the audio playback system to load the wave data from the file `wave = audiocore.WaveFile(wave_file)` and finally request that the audio is played through the A0 analog output pin `audio = audioio.AudioOut(board.A0)`.

The audio file is now ready to go, and can be played at any time with `audio.play(wave)`!

Inside our loop, we start by playing the file.

Next we have the block that tells the code to wait 6 seconds before pausing the file. We chose to go with using `time.monotonic()` because it's **non-blocking** which means you can do other things while the file is playing, like control servos or NeoPixels! At any given point in time, `time.monotonic()` is equal to the number seconds since your board was last power-cycled. (The soft-reboot that occurs with the auto-reload when you save changes to your CircuitPython code, or enter and exit the REPL, does not start it over.) When it is called, it returns a number with a decimal. When you assign `time.monotonic()` to a variable, that variable is equal to the number of seconds that `time.monotonic()` was equal to at the moment the variable was assigned. You can then call it again and subtract the variable from `time.monotonic()` to get the amount of time that has passed. For more details, check out [this example](https://adafru.it/BIT) (https://adafru.it/BIT).

So, we assign `t = time.monotonic()` to get a starting point. Then we say `pass`, or "do nothing" until the difference between `t` and `time.monotonic()` is greater than `6` seconds. In other words, continue playing until 6 seconds passes. Remember, you can add in other code here to do other things while you're playing audio for 6 seconds.

Then we `pause` the audio and `print` to the serial console, `"Waiting for button press to continue!"`

Now we're going to wait for a button press in the same way we did for playing the generated tone. We're saying `while button.value`, or while the button is returning `True`, `pass`. Once the button is pressed, it returns `False`, and this tells the code to continue.

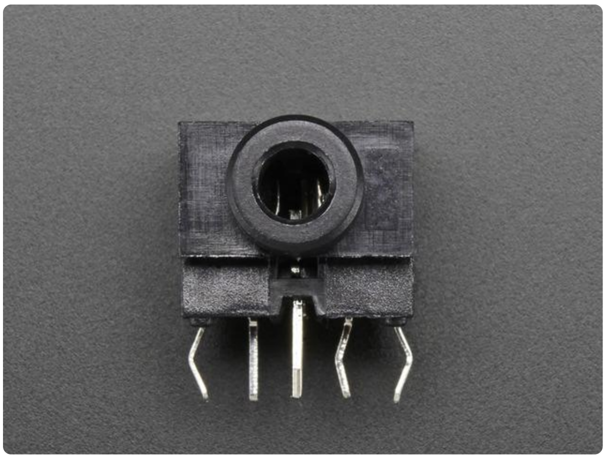
Once the button is pressed, we `resume` playing the file. We tell it to finish playing saying `while audio.playing: pass`.

Finally, we `print` to the serial console, `"Done!"`

You can do this with any supported wave file, and you can include all kinds of things in your project while the file is playing. Give it a try!

Wire It Up

Along with your **microcontroller board**, we're going to be using:



[Breadboard-Friendly 3.5mm Stereo Headphone Jack](https://www.adafruit.com/product/1699)

Pipe audio in or out of your project with this very handy breadboard-friendly audio jack. It's a stereo jack with disconnect-switches on Left and Right channels as well as a center...

https://www.adafruit.com/product/1699

[Tactile Switch Buttons (12mm square, 6mm tall) x 10 pack](https://www.adafruit.com/product/1119)

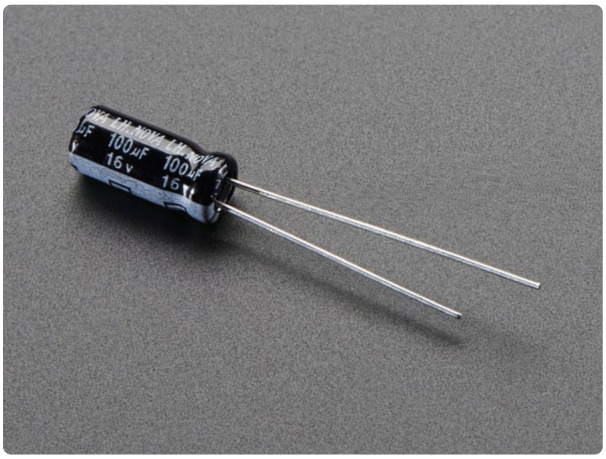
Medium-sized clicky momentary switches are standard input "buttons" on electronic projects. These work best in a PCB but

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

[Panel Mount 10K potentiometer (Breadboard Friendly)](https://www.adafruit.com/product/562)

This potentiometer is a two-in-one, good in a breadboard or with a panel. It's a fairly standard linear taper 10K ohm potentiometer, with a grippy shaft. It's smooth and easy...

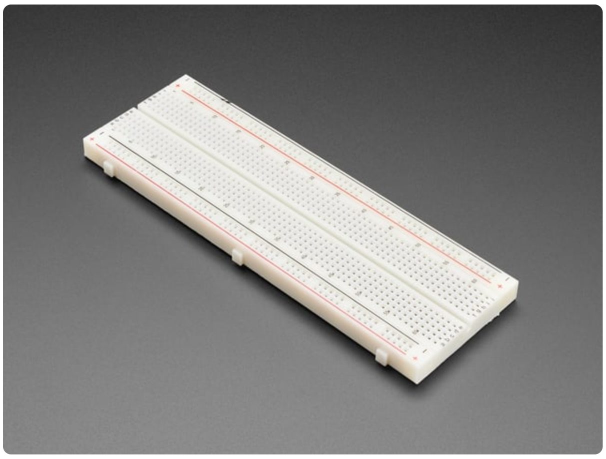
https://www.adafruit.com/product/562



[100uF 16V Electrolytic Capacitors - Pack of 10](https://www.adafruit.com/product/2193)

We like capacitors so much we made a kids' show about them. ...

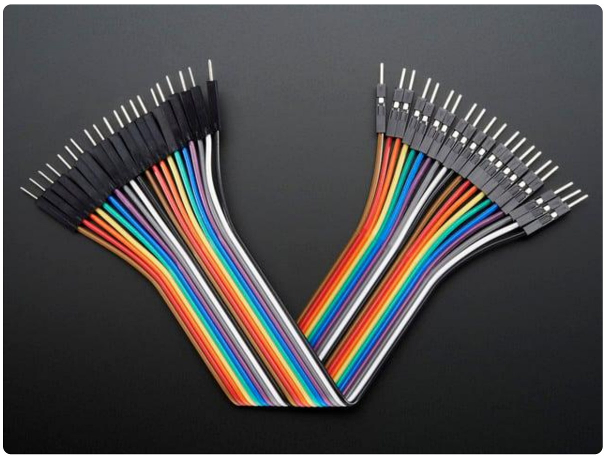
https://www.adafruit.com/product/2193



[Full Sized Premium Breadboard - 830 Tie Points](https://www.adafruit.com/product/239)

This is a 'full-size' premium quality breadboard, 830 tie points. Good for small and medium projects. It's 2.2" x 7" (5.5 cm x 17 cm) with a standard double-strip...

https://www.adafruit.com/product/239

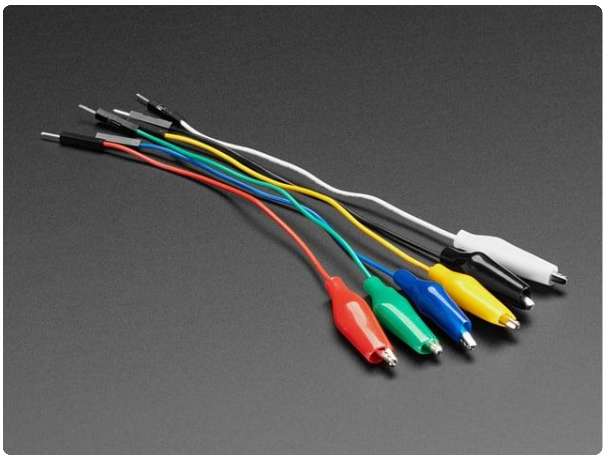


Premium Male/Male Jumper Wires - 20 x 6" (150mm)

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

And to make it easier to wire up the Circuit Playground Express:



Small Alligator Clip to Male Jumper Wire Bundle - 6 Pieces

When working with unusual non-header-friendly surfaces, these handy cables will be your best friends! No longer will you have long, cumbersome strands of alligator clips. These...

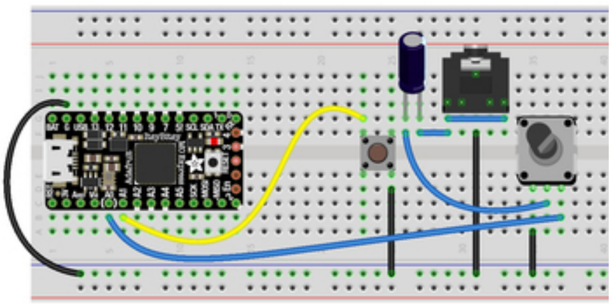
https://www.adafruit.com/product/3448

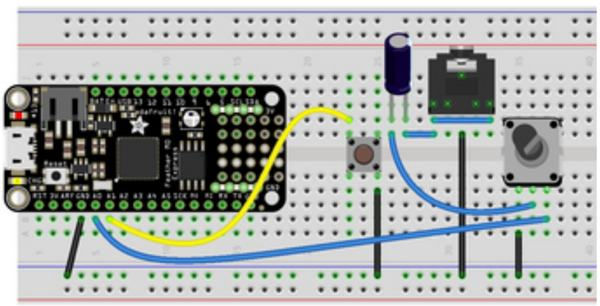
Button switches with four pins are really two pairs of pins. When wiring up a button switch with four pins, the easiest way to verify that you're wiring up the correct pins is to **wire up opposite corners of the button switch**. Then there's no chance that you'll accidentally wire up the same pin twice.

Here are the steps you're going to follow to wire up these components:

- Connect the **ground pin on your board** to a **ground rail on the breadboard** because you'll be connecting all three components to ground.
- Connect **one pin on the button switch** to **pin A1 on your board**, and the **opposite pin on the button switch** to the **ground rail on the breadboard**.
- Connect the **left and right pin on the audio jack to each other**.
- Connect the **center pin on the audio jack** to the **ground rail on the breadboard**.
- Connect the **left pin** to the **negative side of a 100uF capacitor.**
- Connect the **positive side of the capacitor** to the **center pin on the potentiometer**.
- Connect the **right pin on the potentiometer** to **pin A0 on your board**.
- Connect the **left pin of the potentiometer** to **the ground rail on the breadboard.**

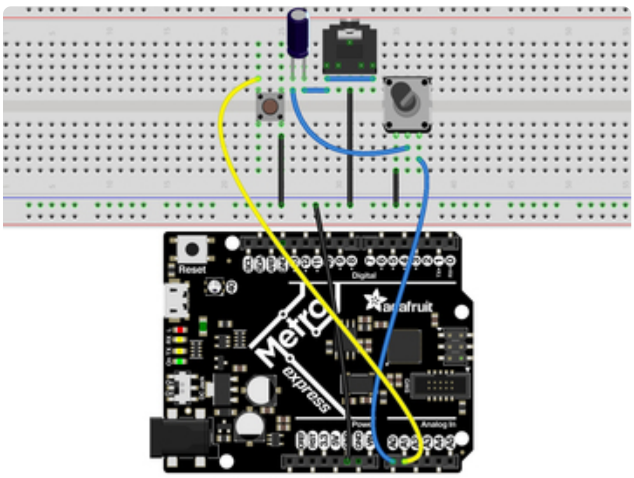
The list below shows wiring diagrams to help with finding the correct pins and wiring up the different components. The ground wires are black. The wire for the button switch is yellow. The wires involved with audio are blue.

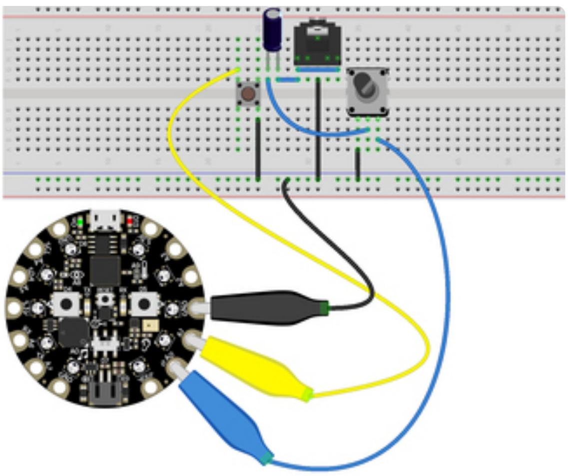




Wiring is the same for the M4 versions of the boards as it is for the M0 versions. Follow the same image for both.

Use a breadboard to make your wiring neat and tidy!





Circuit Playground Express is wired electrically the same as the ItsyBitsy/Feather/Metro above but we use alligator clip to jumper wires instead of plain jumpers

CircuitPython PWM

Your board has `pwmio` support, which means you can PWM LEDs, control servos, beep piezos, and manage "pulse train" type devices like DHT22 and Infrared.

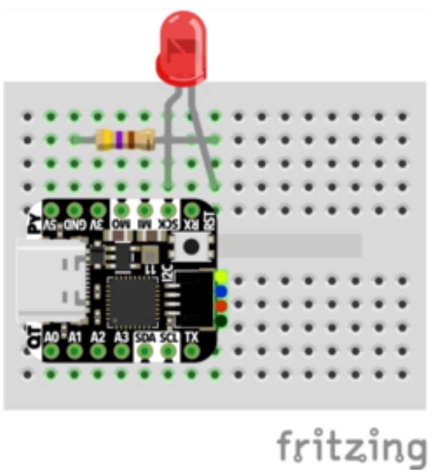
Nearly every pin has PWM support! For example, all ATSAMD21 board have an **A0** pin which is 'true' analog out and does not have PWM support.

PWM with Fixed Frequency

This example will show you how to use PWM to fade the little red LED on your board.

> The QT Py M0 does not have a little red LED. Therefore, you must connect an external LED and edit this example for it to work. Follow the wiring diagram and steps below to run this example on QT Py M0.

The following illustrates how to connect an external LED to a QT Py M0.



LED + to **QT Py SCK**
LED - to **470Ω resistor**
470Ω resistor to **QT Py GND**

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/CircuitPython_PWM/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials: PWM with Fixed Frequency example."""
import time
import board
import pwmio

# LED setup for most CircuitPython boards:
led = pwmio.PWMOut(board.LED, frequency=5000, duty_cycle=0)
# LED setup for QT Py M0:
# led = pwmio.PWMOut(board.SCK, frequency=5000, duty_cycle=0)

while True:
    for i in range(100):
        # PWM LED up and down
        if i < 50:
            led.duty_cycle = int(i * 2 * 65535 / 100)  # Up
        else:
            led.duty_cycle = 65535 - int((i - 50) * 2 * 65535 / 100)  # Down
        time.sleep(0.01)
```

Remember: To "comment out" a line, put a # and a space before it. To "uncomment" a line, remove the # + space from the beginning of the line.

To use with QT Py M0, you must comment out `led = pwmio.PWMOut(board.LED, frequency=5000, duty_cycle=0)` and uncomment `led = pwmio.PWMOut(board.SCK, frequency=5000, duty_cycle=0)`. Your setup lines should look like this for the example to work with QT Py M0:

```
# LED setup for most CircuitPython boards:
# led = pwmio.PWMOut(board.LED, frequency=5000, duty_cycle=0)
# LED setup for QT Py M0:
led = pwmio.PWMOut(board.SCK, frequency=5000, duty_cycle=0)
```

Create a PWM Output

```
led = pwmio.PWMOut(board.LED, frequency=5000, duty_cycle=0)
```

Since we're using the onboard LED, we'll call the object `led`, use `pwmio.PWMOut` to create the output and pass in the `D13` LED pin to use.

Main Loop

The main loop uses `range()` to cycle through the loop. When the range is below 50, it PWMs the LED brightness up, and when the range is above 50, it PWMs the brightness down. This is how it fades the LED brighter and dimmer!

The `time.sleep()` is needed to allow the PWM process to occur over a period of time. Otherwise it happens too quickly for you to see!

PWM Output with Variable Frequency

Fixed frequency outputs are great for pulsing LEDs or controlling servos. But if you want to make some beeps with a piezo, you'll need to vary the frequency.

The following example uses `pwmio` to make a series of tones on a piezo.

To use with any of the M0 boards, no changes to the following code are needed.

> Remember: To "comment out" a line, put a # and a space before it. To "uncomment" a line, remove the # + space from the beginning of the line.

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/CircuitPython_PWM_Piezo/** 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:

To use with the Metro M4 Express, ItsyBitsy M4 Express or the Feather M4 Express, you must comment out the `piezo = pwmio.PWMOut(board.A2, duty_cycle=0, frequency=440, variable_frequency=True)` line and uncomment the `piezo = pwmio.PWMOut(board.A1, duty_cycle=0, frequency=440, variable_frequency=True)` line. A2 is not a supported PWM pin on the M4 boards!

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials PWM with variable frequency piezo example"""
import time
import board
import pwmio

# For the M0 boards:
piezo = pwmio.PWMOut(board.A2, duty_cycle=0, frequency=440, variable_frequency=True)

# For the M4 boards:
# piezo = pwmio.PWMOut(board.A1, duty_cycle=0, frequency=440,
variable_frequency=True)

while True:
    for f in (262, 294, 330, 349, 392, 440, 494, 523):
        piezo.frequency = f
        piezo.duty_cycle = 65535 // 2  # On 50%
        time.sleep(0.25)  # On for 1/4 second
        piezo.duty_cycle = 0  # Off
        time.sleep(0.05)  # Pause between notes
    time.sleep(0.5)
```

The following example uses a nice little helper in the `simpleio` library that makes a tone for you on a piezo with a single command.

To use with any of the M0 boards, no changes to the following code are needed.

To use with the Metro M4 Express, ItsyBitsy M4 Express or the Feather M4 Express, you must comment out the `simpleio.tone(board.A2, f, 0.25)` line and uncomment the `simpleio.tone(board.A1, f, 0.25)` line. A2 is not a supported PWM pin on the M4 boards!

Installing Project Code

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_Essentials/CircuitPython_PWM_Piezo_simpleio/** 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: 2017 Limor Fried for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials PWM piezo simpleio example"""
import time
import board
import simpleio

while True:
    for f in (262, 294, 330, 349, 392, 440, 494, 523):
        # For the M0 boards:
        simpleio.tone(board.A2, f, 0.25)  # on for 1/4 second
        # For the M4 boards:
        # simpleio.tone(board.A1, f, 0.25)  # on for 1/4 second
        time.sleep(0.05)  # pause between notes
    time.sleep(0.5)
```

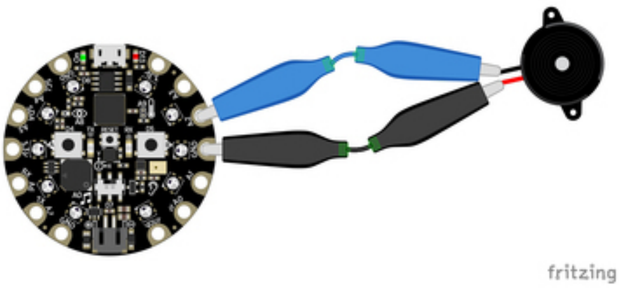
As you can see, it's much simpler!

Wire it up

Use the diagrams below to help you wire up your piezo. Attach one leg of the piezo to pin **A2** on the M0 boards or **A1** on the M4 boards, and the other leg to **ground**. It doesn't matter which leg is connected to which pin. They're interchangeable!

Circuit Playground Express

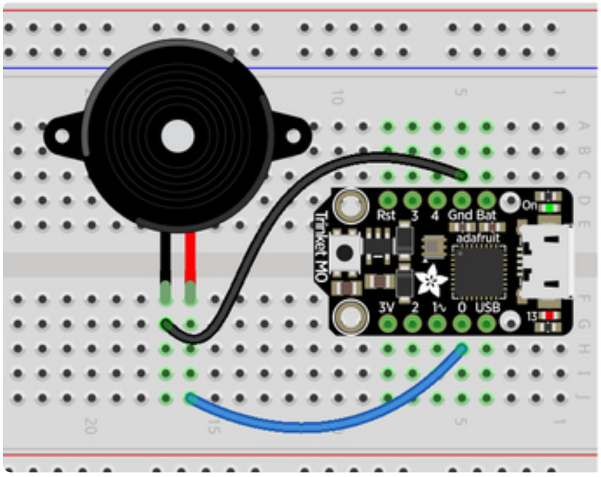
Use alligator clips to attach **A2** and any one of the **GND** to different legs of the piezo.



CPX has PWM on the following pins: A1, A2, A3, A6, RX, LIGHT, A8, TEMPERATURE, A9, BUTTON_B, D5, SLIDE_SWITCH, D7, D13, REMOTEIN, IR_RX, REMOTEOUT, IR_TX, IR_PROXIMITY, MICROPHONE_CLOCK, MICROPHONE_DATA, ACCELEROMETER_INTERRUPT, ACCELEROMETER_SDA, ACCELEROMETER_SCL, SPEAKER_ENABLE.

There is NO PWM on: A0, SPEAKER, A4, SCL, A5, SDA, A7, TX, BUTTON_A, D4, NEOPIXEL, D8, SCK, MOSI, MISO, FLASH_CS.

Trinket M0



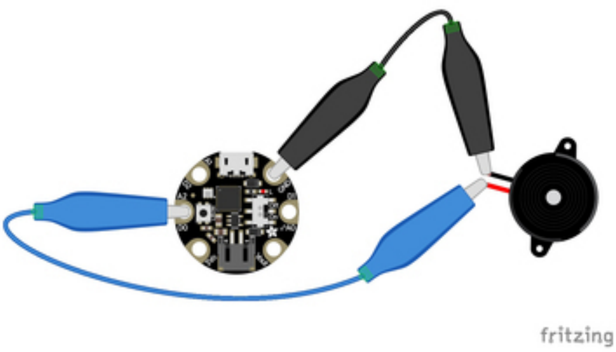
Note: A2 on Trinket is also labeled Digital "0"!

Use jumper wires to connect **GND** and **D0** to different legs of the piezo.

Trinket has PWM available on the following pins: D0, A2, SDA, D2, A1, SCL, MISO, D4, A4, TX, MOSI, D3, A3, RX, SCK, D13, APA102_MOSI, APA102_SCK.

There is NO PWM on: A0, D1.

Gemma M0

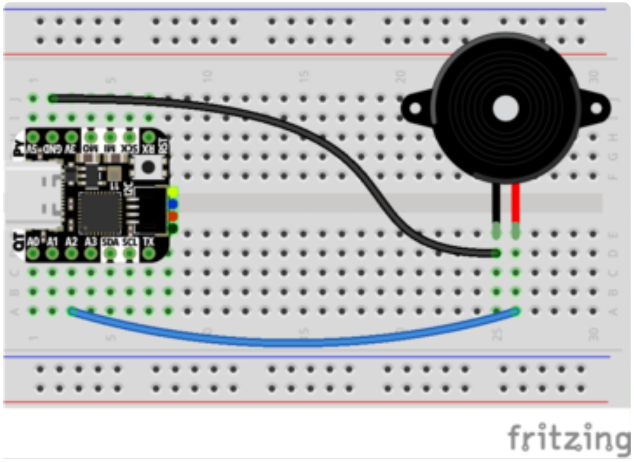


Use alligator clips to attach **A2** and **GND** to different legs on the piezo.

Gemma has PWM available on the following pins: A1, D2, RX, SCL, A2, D0, TX, SDA, L, D13, APA102_MOSI, APA102_SCK.

There is NO PWM on: A0, D1.

QT Py M0

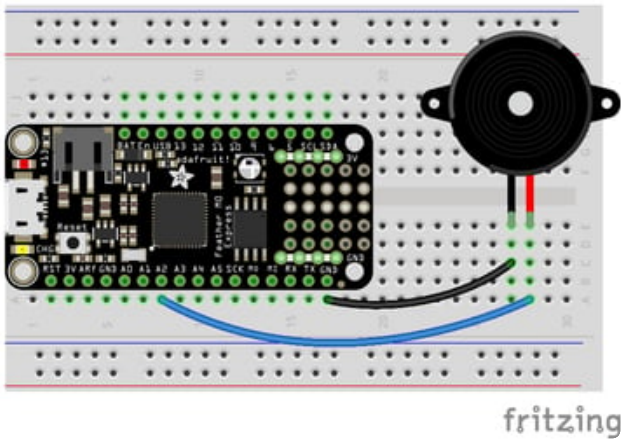


Use jumper wires to attach **A2** and **GND** to different legs of the piezo.

The QT Py M0 has PWM on the following pins: A2, A3, A6, A7, A8, A9, A10, D2, D3, D4, D5, D6, D7, D8, D9, D10, SCK, MISO, MOSI, NEOPIXEL, RX, TX, SCL, SDA.

There is NO A0, A1, D0, D1, NEOPIXEL_POWER.

Feather M0 Express

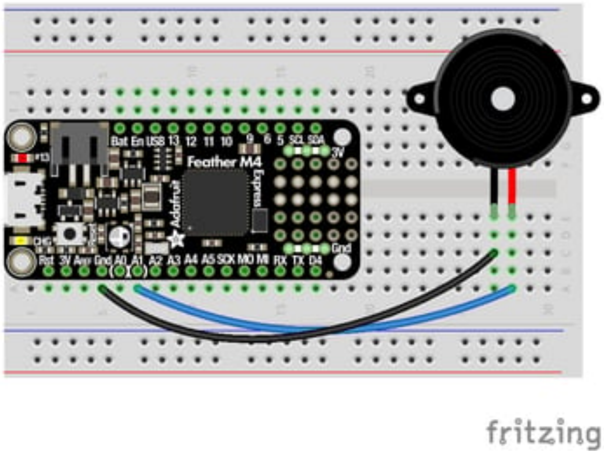


Use jumper wires to attach **A2** and one of the two **GND** to different legs of the piezo.

Feather M0 Express has PWM on the following pins: A2, A3, A4, SCK, MOSI, MISO, D0, RX, D1, TX, SDA, SCL, D5, D6, D9, D10, D11, D12, D13, NEOPIXEL.

There is NO PWM on: A0, A1, A5.

Feather M4 Express



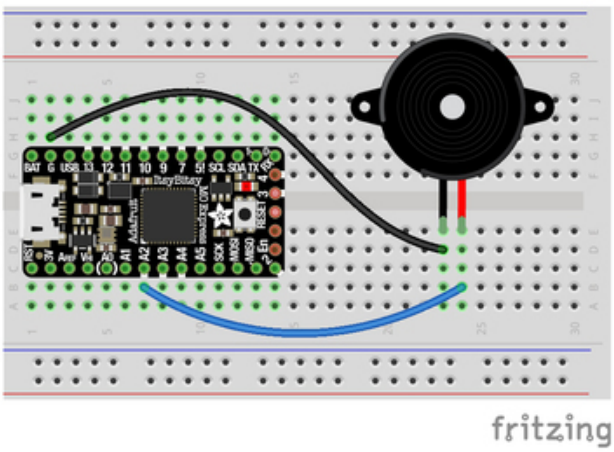
Use jumper wires to attach **A1** and one of the two **GND** to different legs of the piezo.

To use A1, comment out the current pin setup line, and uncomment the line labeled for the M4 boards. See the details above!

Feather M4 Express has PWM on the following pins: A1, A3, SCK, D0, RX, D1, TX, SDA, SCL, D4, D5, D6, D9, D10, D11, D12, D13.

There is NO PWM on: A0, A2, A4, A5, MOSI, MISO.

ItsyBitsy M0 Express

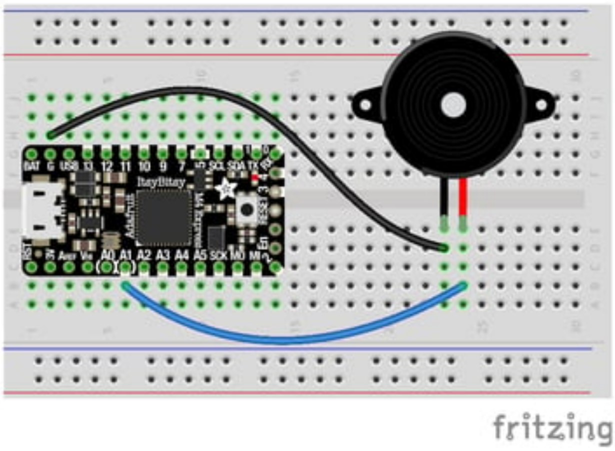


Use jumper wires to attach **A2** and **G** to different legs of the piezo.

ItsyBitsy M0 Express has PWM on the following pins: D0, RX, D1, TX, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, L, A2, A3, A4, MOSI, MISO, SCK, SCL, SDA, APA102_MOSI, APA102_SCK.

There is NO PWM on: A0, A1, A5.

ItsyBitsy M4 Express



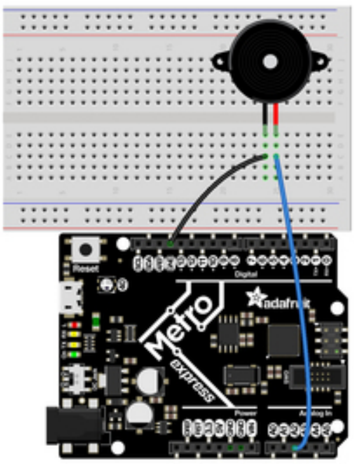
Use jumper wires to attach **A1** and **G** to different legs of the piezo.

To use A1, comment out the current pin setup line, and uncomment the line labeled for the M4 boards. See the details above!

ItsyBitsy M4 Express has PWM on the following pins: A1, D0, RX, D1, TX, D2, D4, D5, D7, D9, D10, D11, D12, D13, SDA, SCL.

There is NO PWM on: A2, A3, A4, A5, D3, SCK, MOSI, MISO.

Metro M0 Express

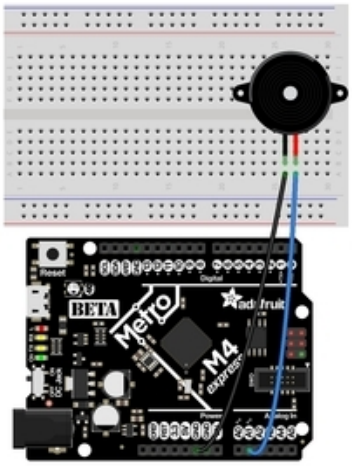


Use jumper wires to connect **A2** and any one of the **GND** to different legs on the piezo.

Metro M0 Express has PWM on the following pins: A2, A3, A4, D0, RX, D1, TX, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, SDA, SCL, NEOPIXEL, SCK, MOSI, MISO.

There is NO PWM on: A0, A1, A5, FLASH_CS.

Metro M4 Express



Use jumper wires to connect **A1** and any one of the **GND** to different legs on the piezo.

To use A1, comment out the current pin setup line, and uncomment the line labeled for the M4 boards. See the details above!

Metro M4 Express has PWM on: A1, A5, D0, RX, D1, TX, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, SDA, SCK, MOSI, MISO

There is No PWM on: A0, A2, A3, A4, SCL, AREF, NEOPIXEL, LED_RX, LED_TX.

Where's My PWM?

Want to check to see which pins have PWM yourself? We've written this handy script! It attempts to setup PWM on every pin available, and lets you know which ones work and which ones don't. Check it 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_Essentials/PWM_Test_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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials PWM pin identifying script"""
import board
import pwmio

for pin_name in dir(board):
    pin = getattr(board, pin_name)
    try:
        p = pwmio.PWMOut(pin)
        p.deinit()
        print("PWM on:", pin_name)  # Prints the valid, PWM-capable pins!
    except ValueError:  # This is the error returned when the pin is invalid.
        print("No PWM on:", pin_name)  # Prints the invalid pins.
    except RuntimeError:  # Timer conflict error.
        print("Timers in use:", pin_name)  # Prints the timer conflict pins.
    except TypeError:  # Error returned when checking a non-pin object in
dir(board).
        pass  # Passes over non-pin objects in dir(board).
```

CircuitPython Servo

In order to use servos, we take advantage of `pwmio`. Now, in theory, you could just use the raw `pwmio` calls to set the frequency to 50 Hz and then set the pulse widths. But we would rather make it a little more elegant and easy!

So, instead we will use `adafruit_motor` which manages servos for you quite nicely! `adafruit_motor` is a library so be sure to grab it from the library bundle if you have not yet (https://adafru.it/ENC)! If you need help installing the library, check out the CircuitPython Libraries page (https://adafru.it/ABU).

Servos come in two types:

- A **standard hobby servo** - the horn moves 180 degrees (90 degrees in each direction from zero degrees).
- A **continuous servo** - the horn moves in full rotation like a DC motor. Instead of an angle specified, you set a throttle value with 1.0 being full forward, 0.5 being half forward, 0 being stopped, and -1 being full reverse, with other values between.

Servo Wiring

Servos will only work on PWM-capable pins! Check your board details to verify which pins have PWM outputs.

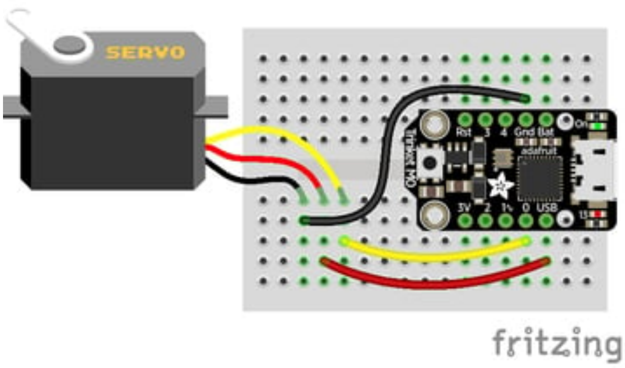
The connections for a servo are the same for standard servos and continuous rotation servos.

Connect the servo's **brown** or **black** ground wire to ground on the CircuitPython board.

Connect the servo's **red** power wire to 5V power. USB power can in some cases good for a servo or two. But some USB ports supply limited current, and operating a servo from the USB 5V line may cause a power brownout and board crash.

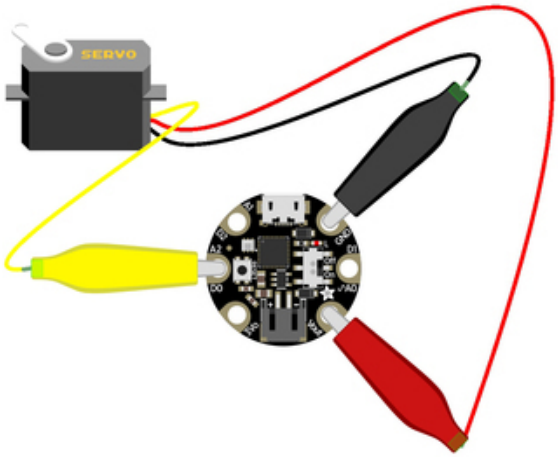
For more servos, you'll need an external battery pack or external power supply. Do not use 3.3V for powering a servo!

Connect the servo's **yellow** or **white** signal wire to the control/data pin, in this case **A1 or A2** but you can use any PWM-capable pin.

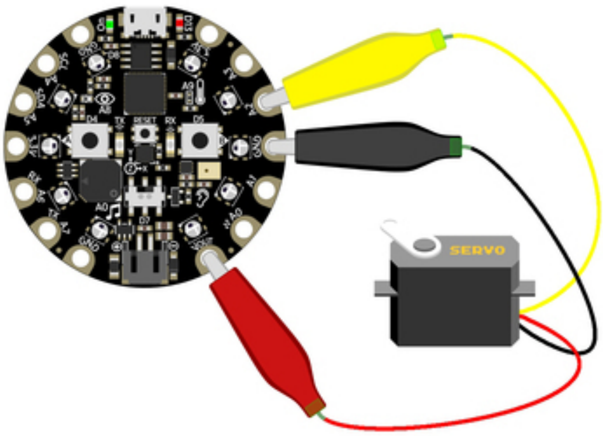


For example, to wire a servo to **Trinket**, connect the ground wire to **GND**, the power wire to **USB**, and the signal wire to **0**.

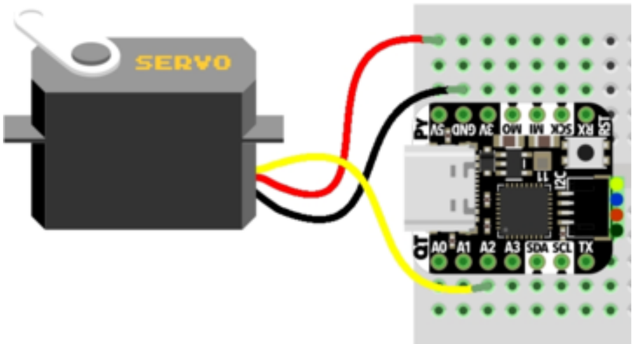
Remember, **A2 on Trinket is labeled "0"**.



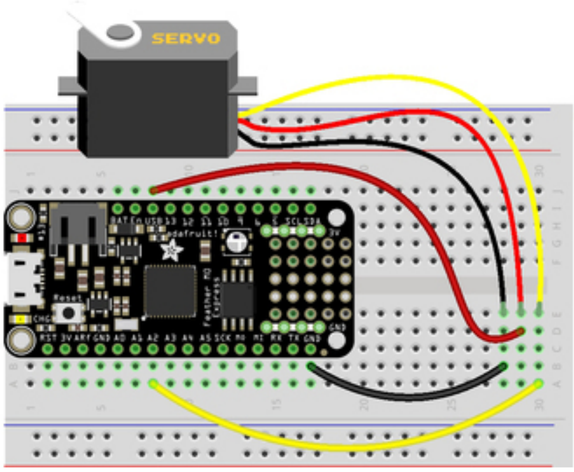
For **Gemma**, use jumper wire alligator clips to connect the ground wire to **GND**, the power wire to **VOUT**, and the signal wire to **A2**.



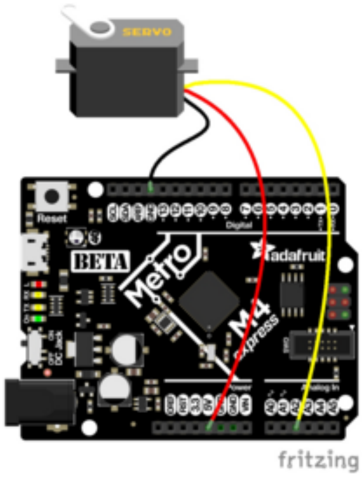
For **Circuit Playground Express and Circuit Playground Bluefruit**, use jumper wire alligator clips to connect the ground wire to **GND**, the power wire to **VOUT**, and the signal wire to **A2**.



For **QT Py M0**, connect the ground wire to **GND**, the power wire to **5V**, and the signal wire to **A2**.



For boards like **Feather M0 Express**, **ItsyBitsy M0 Express** and **Metro M0 Express**, connect the ground wire to any **GND**, the power wire to **USB or 5V**, and the signal wire to **A2**.



For the **Metro M4 Express**, **ItsyBitsy M4 Express** and the **Feather M4 Express**, connect the ground wire to any **G or GND**, the power wire to **USB or 5V**, and the signal wire to **A2**.

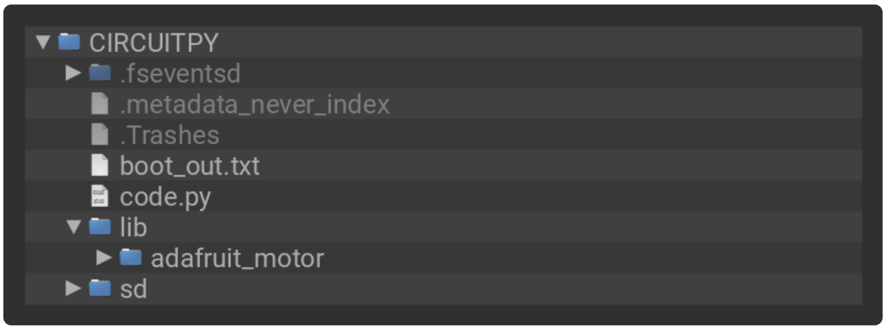
Standard Servo Code

Here's an example that will sweep a servo connected to pin **A2** from 0 degrees to 180 degrees (-90 to 90 degrees) and back.

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_Essentials/CircuitPython_Servo/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Servo standard servo example"""
import time
import board
import pwmio
from adafruit_motor import servo

# create a PWMOut object on Pin A2.
pwm = pwmio.PWMOut(board.A2, duty_cycle=2 ** 15, frequency=50)

# Create a servo object, my_servo.
my_servo = servo.Servo(pwm)

while True:
    for angle in range(0, 180, 5):  # 0 - 180 degrees, 5 degrees at a time.
        my_servo.angle = angle
        time.sleep(0.05)
    for angle in range(180, 0, -5): # 180 - 0 degrees, 5 degrees at a time.
        my_servo.angle = angle
        time.sleep(0.05)
```

Continuous Servo Code

There are two differences with Continuous Servos vs. Standard Servos:

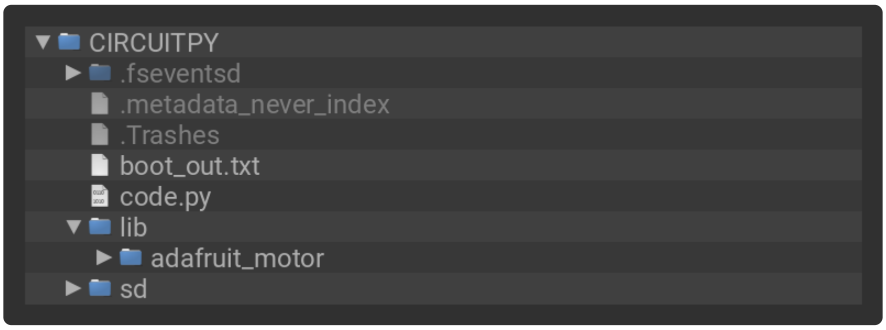
1. The `servo` object is created like `my_servo = servo.ContinuousServo(pwm)` instead of `my_servo = servo.Servo(pwm)`
2. Instead of using `myservo.angle`, you use `my_servo.throttle` using a throttle value from 1.0 (full on) to 0.0 (stopped) to -1.0 (full reverse). Any number between would be a partial speed forward (positive) or reverse (negative). This is very similar to standard DC motor control with the **adafruit_motor** library.

This example runs full forward for 2 seconds, stops for 2 seconds, runs full reverse for 2 seconds, then stops for 4 seconds.

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_Essentials/CircuitPython_Continuous_Servo/** 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: 2019 Anne Barela for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Servo continuous rotation servo example"""
import time
import board
import pwmio
from adafruit_motor import servo

# create a PWMOut object on Pin A2.
pwm = pwmio.PWMOut(board.A2, frequency=50)

# Create a servo object, my_servo.
my_servo = servo.ContinuousServo(pwm)

while True:
    print("forward")
    my_servo.throttle = 1.0
    time.sleep(2.0)
    print("stop")
    my_servo.throttle = 0.0
    time.sleep(2.0)
    print("reverse")
    my_servo.throttle = -1.0
    time.sleep(2.0)
    print("stop")
    my_servo.throttle = 0.0
    time.sleep(4.0)
```

Pretty simple!

Note that we assume that 0 degrees is 0.5ms and 180 degrees is a pulse width of 2.5ms. That's a bit wider than the official 1-2ms pulse widths. If you have a servo that has a different range you can initialize the `servo` object with a different `min_pulse` and `max_pulse`. For example:

```
my_servo = servo.Servo(pwm, min_pulse = 500, max_pulse = 2500)
```

For more detailed information on using servos with CircuitPython, check out the CircuitPython section of the servo guide (https://adafru.it/Bei)!

CircuitPython Cap Touch

Nearly all CircuitPython boards provide capacitive touch capabilities. This means each board has at least one pin that works as an input when you touch it! For SAMD21 (M0) boards, the capacitive touch is done in hardware, so no external resistors, capacitors or ICs required. On SAMD51 (M4), nRF52840, and some other boards, Adafruit uses a software solution: you will need to add a 1M (1 megaohm) resistor from the pin to ground.

On the Circuit Playground Bluefruit (nrf52840) board, the necessary resistors are already on the board, so you don't need to add them.

This example will show you how to use a capacitive touch pin on your board.

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/CircuitPython_CapTouch/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Capacitive Touch example"""
import time
import board
import touchio

touch_pad = board.A0  # Will not work for Circuit Playground Express!
# touch_pad = board.A1  # For Circuit Playground Express
```

```
touch = touchio.TouchIn(touch_pad)

while True:
    if touch.value:
        print("Touched!")
    time.sleep(0.05)
```

Create the Touch Input

First, we assign the variable `touch_pad` to a pin. The example uses **A0**, so we assign `touch_pad = board.A0`. You can choose any touch capable pin from the list below if you'd like to use a different pin. Then we create the touch object, name it `touch` and attach it to `touch_pad`.

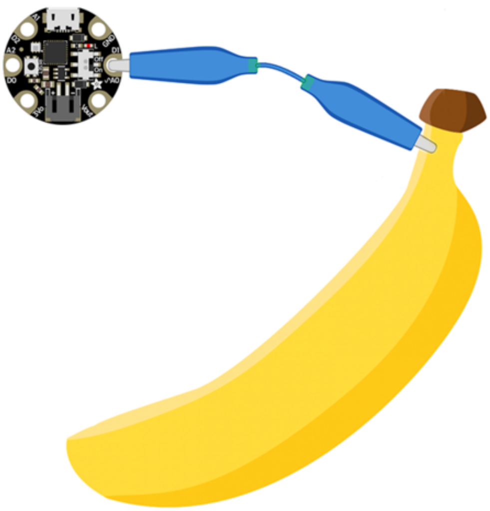
To use with Circuit Playground Express, comment out `touch_pad = board.A0` and uncomment `touch_pad = board.A1`.

Main Loop

Next, we create a loop that checks to see if the pin is touched. If it is, it `prints` to the serial console. Connect to the serial console to see the printed results when you touch the pin!

Remember: To "comment out" a line, put a # and a space before it. To "uncomment" a line, remove the # + space from the beginning of the line.

No extra hardware is required, because you can touch the pin directly. However, you may want to attach alligator clips or copper tape to metallic or conductive objects. Try metal flatware, fruit or other foods, liquids, aluminum foil, or other items lying around your desk!



You may need to reload your code or restart your board after changing the attached item because the capacitive touch code "calibrates" based on what it sees when it first starts up. So if you get too many touch responses or not enough, reload your code through the serial console or eject the board and tap the reset button!

Find the Pin(s)

Your board may have more touch capable pins beyond A0. We've included a list below that helps you find A0 (or A1 in the case of CPX) for this example, identified by the magenta arrow. This list also includes information about any other pins that work for touch on each board!

To use the other pins, simply change the number in **A0** to the pin you want to use. For example, if you want to use **A3** instead, your code would start with `touch_pad = board.A3`.

If you would like to use more than one pin at the same time, your code may look like the following. If needed, you can modify this code to include pins that work for your board.

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/CircuitPython_CapTouch_2Pins/** 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:

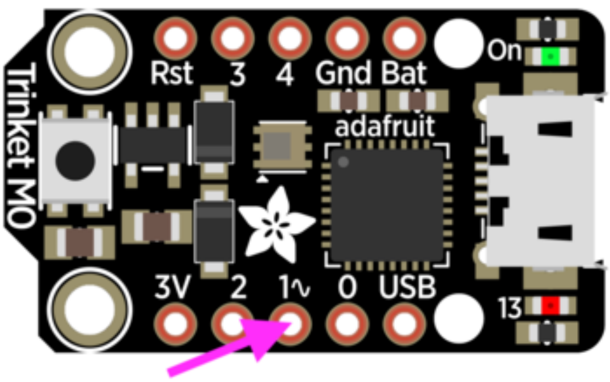

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Capacitive Touch on two pins example. Does not work on
Trinket M0!"""
import time
import board
import touchio

touch_A1 = touchio.TouchIn(board.A1)  # Not a touch pin on Trinket M0!
```

```
touch_A2 = touchio.TouchIn(board.A2)  # Not a touch pin on Trinket M0!

while True:
    if touch_A1.value:
        print("Touched A1!")
    if touch_A2.value:
        print("Touched A2!")
    time.sleep(0.05)
```

This example does NOT work for Trinket M0! You must change the pins to use with this board. This example only works with Gemma, Circuit Playground Express, Feather M0 Express, Metro M0 Express and ItsyBitsy M0 Express.

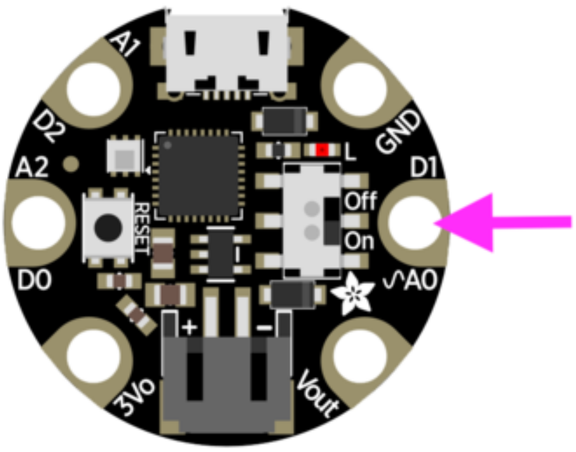
Use the list below to find out what pins you can use with your board. Then, try adding them to your code and have fun!

Trinket M0



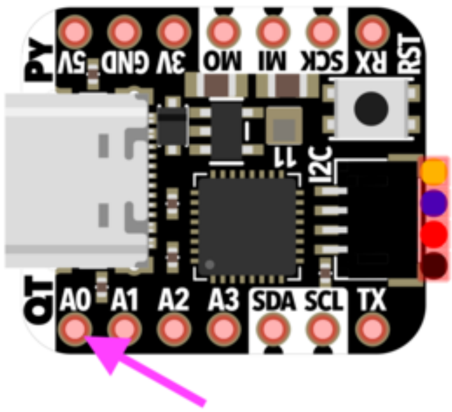
There are three touch capable pins on Trinket: **A0**, **A3**, and **A4**.

Remember, **A0 is labeled "1~" on Trinket M0!**



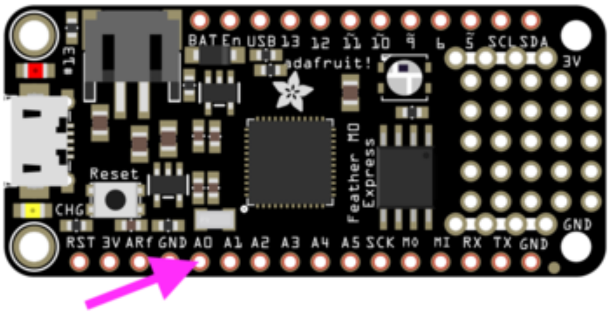
Gemma M0

There are three pins on Gemma, in the form of alligator-clip-friendly pads, that work for touch input: **A0**, **A1** and **A2**.



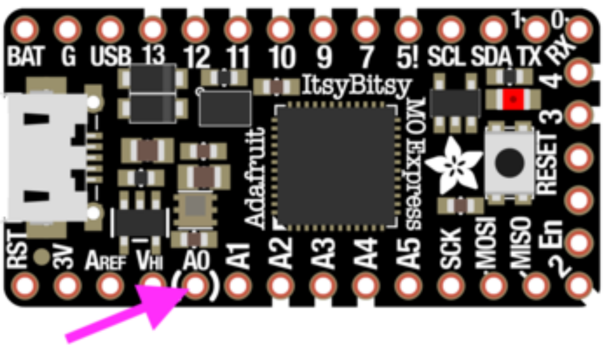
QT Py M0

There are six pins on QT Py that work for touch input: **A0 - A3, TX,** and **RX**.



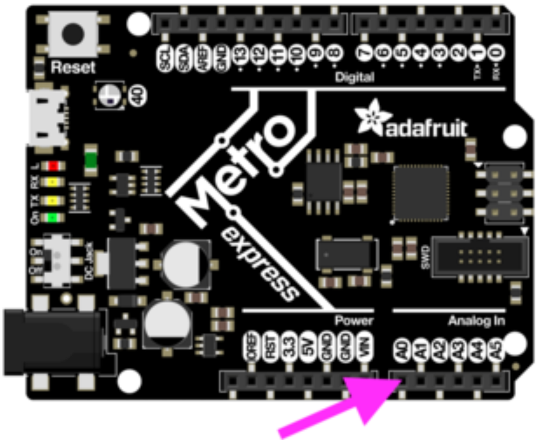
Feather M0 Express

There are 6 pins on the Feather that have touch capability: **A0 - A5**.



ItsyBitsy M0 Express

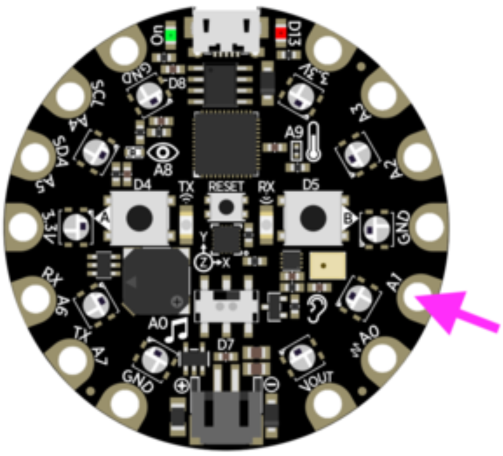
There are 6 pins on the ItsyBitsy that have touch capability: **A0 - A5**.



Metro M0 Express

There are 6 pins on the Metro that have touch capability: **A0 - A5**.

Circuit Playground Express

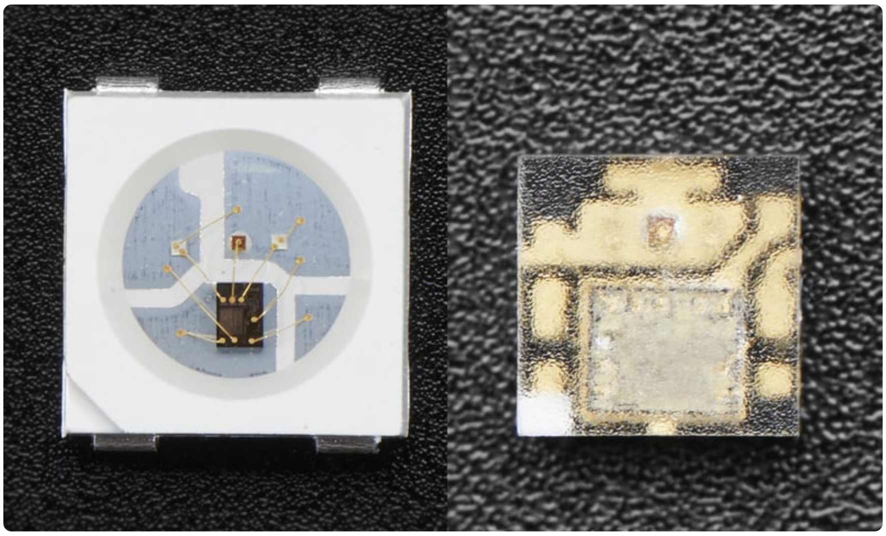


Circuit Playground Express has seven touch capable pins! You have **A1 - A7** available, in the form of alligator-clip-friendly pads. See the CPX guide Cap Touch section (https://adafru.it/ANC) for more information on using these pads for touch!

Remember: A0 does **NOT** have touch capabilities on CPX.

CircuitPython Internal RGB LED

Every board has a built in RGB LED. You can use CircuitPython to control the color and brightness of this LED. There are two different types of internal RGB LEDs: DotStar (https://adafru.it/kDg) and NeoPixel (https://adafru.it/Bej). This section covers both and explains which boards have which LED.

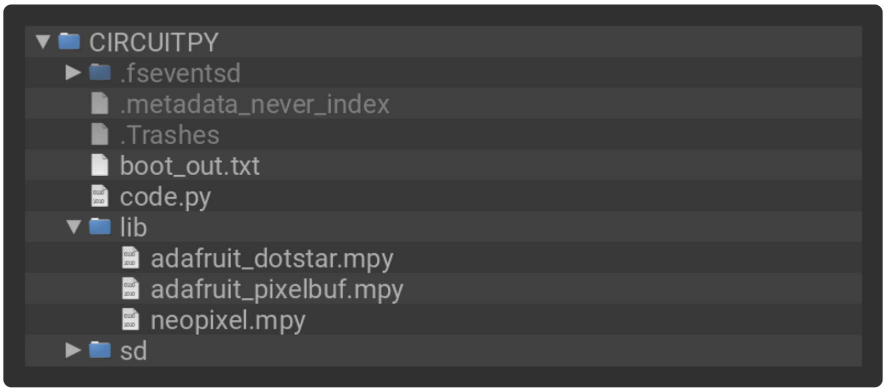


The first example will show you how to change the color and brightness of the internal RGB LED.

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_Essentials/CircuitPython_Internal_RGB_LED_colors/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Internal RGB LED red, green, blue example"""
import time
import board

if hasattr(board, "APA102_SCK"):
```

```
    import adafruit_dotstar

    led = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1)
else:
    import neopixel

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

led.brightness = 0.3

while True:
    led[0] = (255, 0, 0)
    time.sleep(0.5)
    led[0] = (0, 255, 0)
    time.sleep(0.5)
    led[0] = (0, 0, 255)
    time.sleep(0.5)
```

Create the LED

First, we create the LED object and attach it to the correct pin or pins. In the case of a NeoPixel, there is only one pin necessary, and we have called it `NEOPIXEL` for easier use. In the case of a DotStar, however, there are two pins necessary, and so we use the pin names `APA102_MOSI` and `APA102_SCK` to get it set up. Since we're using the single onboard LED, the last thing we do is tell it that there's only `1` LED!

Trinket M0, **Gemma M0**, **ItsyBitsy M0 Express**, and **ItsyBitsy M4 Express** each have an onboard **Dotstar** LED, so **no changes are needed** to the initial version of the example.

> Remember: To "comment out" a line, put a # and a space before it. To "uncomment" a line, remove the # + space from the beginning of the line.

QT Py M0, Feather M0 Express, Feather M4 Express, Metro M0 Express, Metro M4 Express, and Circuit Playground Express each have an onboard **NeoPixel** LED, so you must **comment out** `import adafruit_dotstar` **and** `led = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1)`, and **uncomment** `import neopixel` **and** `led = neopixel.NeoPixel(board.NEOPIXEL, 1)`.

Brightness

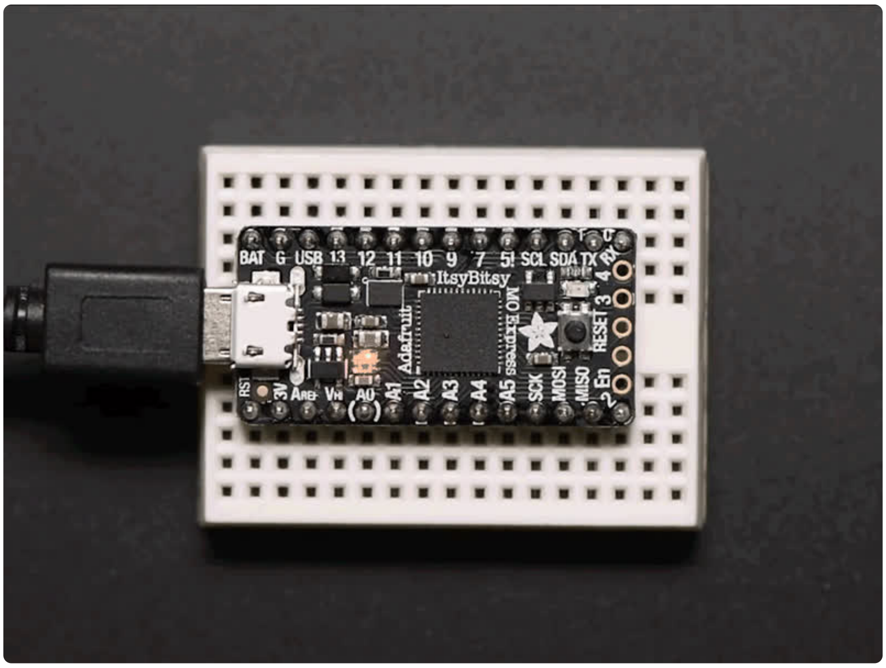
To set the brightness you simply use the `brightness` attribute. Brightness is set with a number between `0` and `1`, representative of a percent from 0% to 100%. So, `led.brightness = (0.3)` sets the LED brightness to 30%. The default brightness is `1` or 100%, and at it's maximum, the LED is blindingly bright! You can set it lower if you choose.

Main Loop

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 the 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.

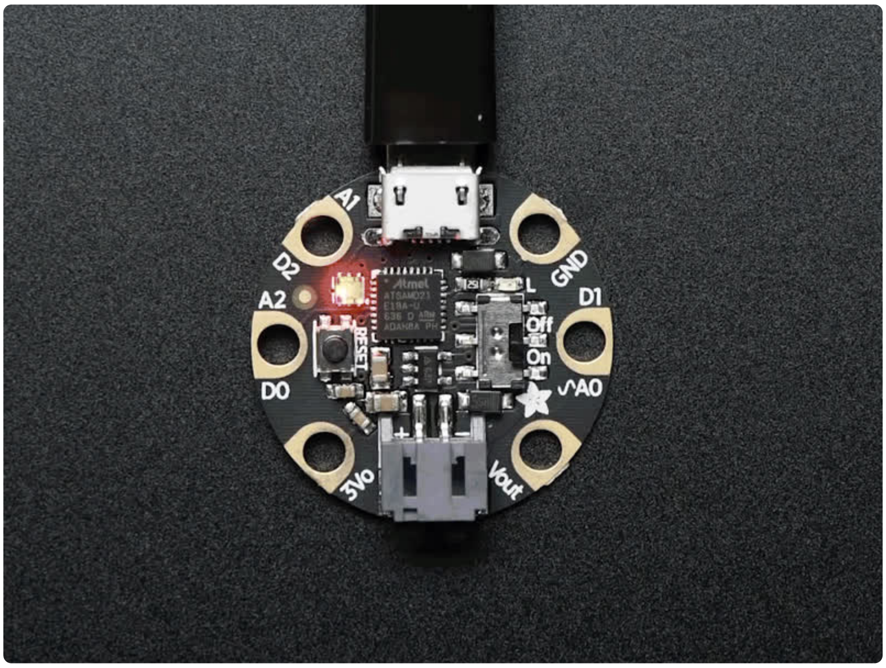
The main loop is quite simple. It sets the first LED to **red** using `(255, 0, 0)`, then **green** using `(0, 255, 0)`, and finally **blue** using `(0, 0, 255)`. Next, we give it a `time.sleep()` so it stays each color for a period of time. We chose `time.sleep(0.5)`, or half a second. Without the `time.sleep()` it'll flash really quickly and the colors will be difficult to see!

Note that we set `led[0]`. This means the first, and in the case of most of the boards, the only LED. In CircuitPython, counting starts at 0. So the first of any object, list, etc will be `0`!



Try changing the numbers in the tuples to change your LED to any color of the rainbow. Or, you can add more lines with different color tuples to add more colors to the sequence. Always add the `time.sleep()`, but try changing the amount of time to create different cycle animations!

Making Rainbows (Because Who Doesn't Love 'Em!)



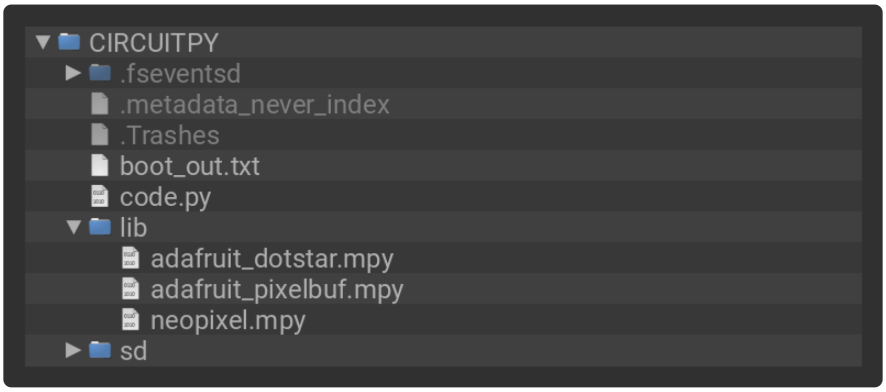
Coding a rainbow effect involves a little math and a helper function called `colorwheel`. For details about how wheel works, see [this explanation here](https://adafru.it/Bek) (https://adafru.it/Bek)!

The last example shows how to do a rainbow animation on the internal RGB 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_Essentials/CircuitPython_Internal_RGB_LED_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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Internal RGB LED rainbow example"""
import time
import board
from rainbowio import colorwheel

if hasattr(board, "APA102_SCK"):
    import adafruit_dotstar

    led = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1)
else:
    import neopixel

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

led.brightness = 0.3

i = 0
while True:
    i = (i + 1) % 256  # run from 0 to 255
    led.fill(colorwheel(i))
    time.sleep(0.01)
```

We add the `colorwheel` function in after setup but before our main loop.

And right before our main loop, we assign the variable `i = 0`, so it's ready for use inside the loop.

The main loop contains some math that cycles `i` from `0` to `255` and around again repeatedly. We use this value to cycle `colorwheel()` through the rainbow!

The `time.sleep()` determines the speed at which the rainbow changes. Try a higher number for a slower rainbow or a lower number for a faster one!

Circuit Playground Express Rainbow

Note that here we use `led.fill` instead of `led[0]`. This means it turns on all the LEDs, which in the current code is only one. So why bother with `fill`? Well, you may have a Circuit Playground Express, which as you can see has TEN NeoPixel LEDs built in. The examples so far have only turned on the first one. If you'd like to do a rainbow on all ten LEDs, change the `1` in:

`led = neopixel.NeoPixel(board.NEOPIXEL, 1)`

to `10` so it reads:

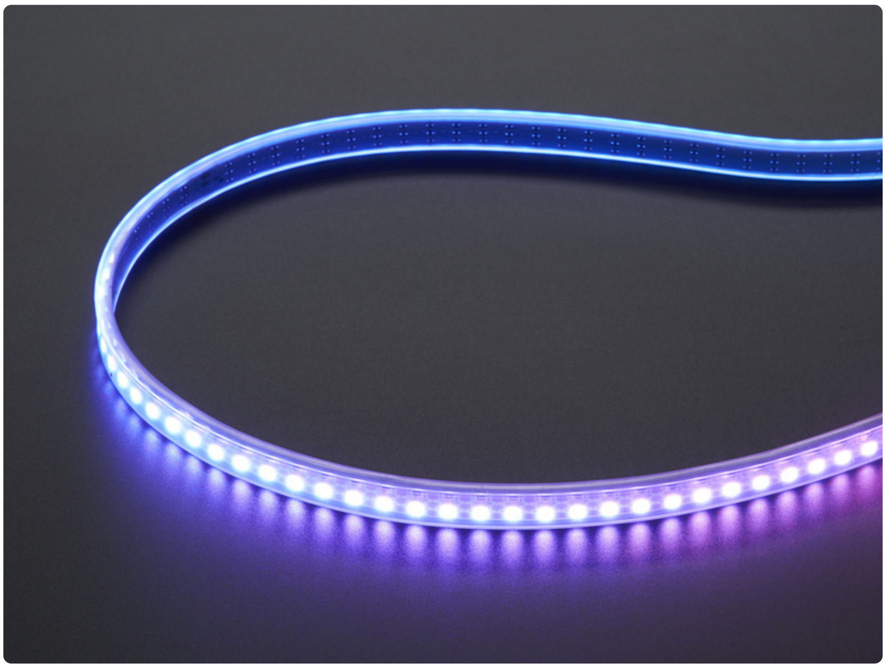
`led = neopixel.NeoPixel(board.NEOPIXEL, 10)`.

This tells the code to look for 10 LEDs instead of only 1. Now save the code and watch the rainbow go! You can make the same `1` to `10` change to the previous examples as well, and use `led.fill` to light up all the LEDs in the colors you chose! For more details, check out the NeoPixel section of the CPX guide (https://adafru.it/Bem)!

CircuitPython NeoPixel

NeoPixels are a revolutionary and ultra-popular way to add lights and color to your project. These stranded RGB lights have the controller inside the LED, so you just push the RGB data and the LEDs do all the work for you. They're a perfect match for CircuitPython!

You can drive 300 NeoPixel LEDs with brightness control (set `brightness=1.0` in object creation) and 1000 LEDs without. That's because to adjust the brightness we have to dynamically recreate the data-stream each write.



Wiring It Up

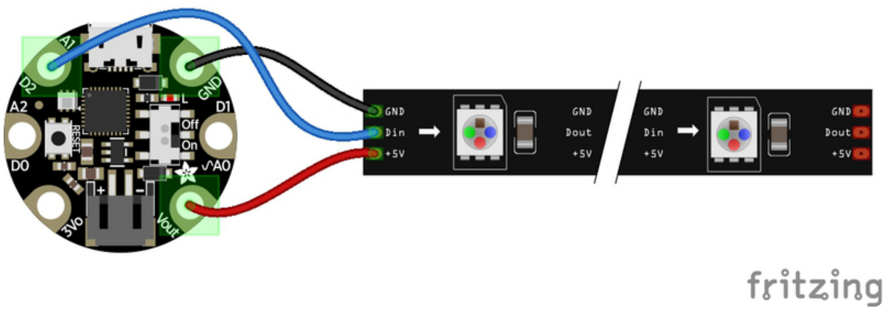
You'll need to solder up your NeoPixels first. Verify your connection is on the **DATA INPUT** or **DIN** side. Plugging into the DATA OUT or DOUT side is a common mistake! The connections are labeled and some formats have arrows to indicate the direction the data must flow.

For powering the pixels from the board, the 3.3V regulator output can handle about 500mA peak which is about 50 pixels with 'average' use. If you want really bright lights and a lot of pixels, we recommend powering direct from an external power source.

- On Gemma M0 and Circuit Playground Express this is the **Vout** pad - that pad has direct power from USB or the battery, depending on which is higher voltage.
- On Trinket M0, Feather M0 Express, Feather M4 Express, ItsyBitsy M0 Express and ItsyBitsy M4 Express the **USB** or **BAT** pins will give you direct power from the USB port or battery.
- On Metro M0 Express and Metro M4 Express, use the **5V** pin regardless of whether it's powered via USB or the DC jack.
- On QT Py M0, use the **5V** pin.

If the power to the NeoPixels is greater than 5.5V you may have some difficulty driving some strips, in which case you may need to lower the voltage to 4.5-5V or use a level shifter.

Do not use the VIN pin directly on Metro M0 Express or Metro M4 Express! The voltage can reach 9V and this can destroy your NeoPixels!



Note that the wire ordering on your NeoPixel strip or shape may not exactly match the diagram above. Check the markings to verify which pin is DIN, 5V and GND

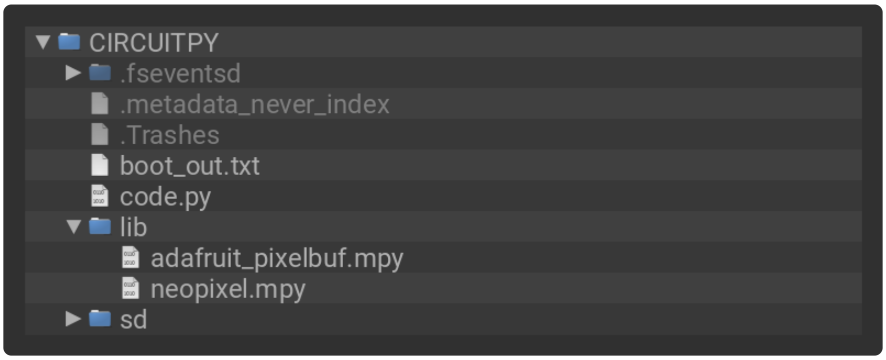
The Code

This example includes multiple visual effects.

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_Essentials/CircuitPython_NeoPixel/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials NeoPixel example"""
import time
import board
from rainbowio import colorwheel
import neopixel

pixel_pin = board.A1
num_pixels = 8

pixels = neopixel.NeoPixel(pixel_pin, num_pixels, brightness=0.3, auto_write=False)


def color_chase(color, wait):
    for i in range(num_pixels):
        pixels[i] = color
        time.sleep(wait)
        pixels.show()
    time.sleep(0.5)


def rainbow_cycle(wait):
    for j in range(255):
        for i in range(num_pixels):
```

```
            rc_index = (i * 256 // num_pixels) + j
            pixels[i] = colorwheel(rc_index & 255)
        pixels.show()
        time.sleep(wait)


RED = (255, 0, 0)
YELLOW = (255, 150, 0)
GREEN = (0, 255, 0)
CYAN = (0, 255, 255)
BLUE = (0, 0, 255)
PURPLE = (180, 0, 255)

while True:
    pixels.fill(RED)
    pixels.show()
    # Increase or decrease to change the speed of the solid color change.
    time.sleep(1)
    pixels.fill(GREEN)
    pixels.show()
    time.sleep(1)
    pixels.fill(BLUE)
    pixels.show()
    time.sleep(1)

    color_chase(RED, 0.1)  # Increase the number to slow down the color chase
    color_chase(YELLOW, 0.1)
    color_chase(GREEN, 0.1)
    color_chase(CYAN, 0.1)
    color_chase(BLUE, 0.1)
    color_chase(PURPLE, 0.1)

    rainbow_cycle(0)  # Increase the number to slow down the rainbow
```

Create the LED

The first thing we'll do is create the LED object. The NeoPixel object has two required arguments and two optional arguments. You are required to set the pin you're using to drive your NeoPixels and provide the number of pixels you intend to use. You can optionally set `brightness` and `auto_write`.

NeoPixels can be driven by any pin. We've chosen **A1**. To set the pin, assign the variable `pixel_pin` to the pin you'd like to use, in our case `board.A1`.

To provide the number of pixels, assign the variable `num_pixels` to the number of pixels you'd like to use. In this example, we're using a strip of `8`.

We've chosen to set `brightness=0.3`, or 30%.

By default, `auto_write=True`, meaning any changes you make to your pixels will be sent automatically. Since `True` is the default, if you use that setting, you don't need to include it in your LED object at all. We've chosen to set `auto_write=False`. If you set `auto_write=False`, you must include `pixels.show()` each time you'd like to send data to your pixels. This makes your code more complicated, but it can make your LED animations faster!

NeoPixel Helpers

Next we've included a few helper functions to create the super fun visual effects found in this code. First is `wheel()` which we just learned with the [Internal RGB LED](https://adafru.it/Bel) (https://adafru.it/Bel). Then we have `color_chase()` which requires you to provide a `color` and the amount of time in seconds you'd like between each step of the chase. Next we have `rainbow_cycle()`, which requires you to provide the mount of time in seconds you'd like the animation to take. Last, we've included a list of variables for our colors. This makes it much easier if to reuse the colors anywhere in the code, as well as add more colors for use in multiple places. Assigning and using RGB colors is explained in [this section of the CircuitPython Internal RGB LED page](https://adafru.it/Bel) (https://adafru.it/Bel).

Main Loop

Thanks to our helpers, our main loop is quite simple. We include the code to set every NeoPixel we're using to red, green and blue for 1 second each. Then we call `color_chase()`, one time for each `color` on our list with `0.1` second delay between setting each subsequent LED the same color during the chase. Last we call `rainbow_cycle(0)`, which means the animation is as fast as it can be. Increase both of those numbers to slow down each animation!

Note that the longer your strip of LEDs, the longer it will take for the animations to complete.

> We have a ton more information on general purpose NeoPixel know-how at our NeoPixel UberGuide https://learn.adafruit.com/adafruit-neopixel-uberguide

NeoPixel RGBW

NeoPixels are available in RGB, meaning there are three LEDs inside, red, green and blue. They're also available in RGBW, which includes four LEDs, red, green, blue and white. The code for RGBW NeoPixels is a little bit different than RGB.

If you run RGB code on RGBW NeoPixels, approximately 3/4 of the LEDs will light up and the LEDs will be the incorrect color even though they may appear to be changing.

This is because NeoPixels require a piece of information for each available color (red, green, blue and possibly white).

Therefore, RGB LEDs require three pieces of information and RGBW LEDs require FOUR pieces of information to work. So when you create the LED object for RGBW LEDs, you'll include `pixel_order=(1, 0, 2, 3)`, which sets the pixel order and indicates four pieces of information involved.

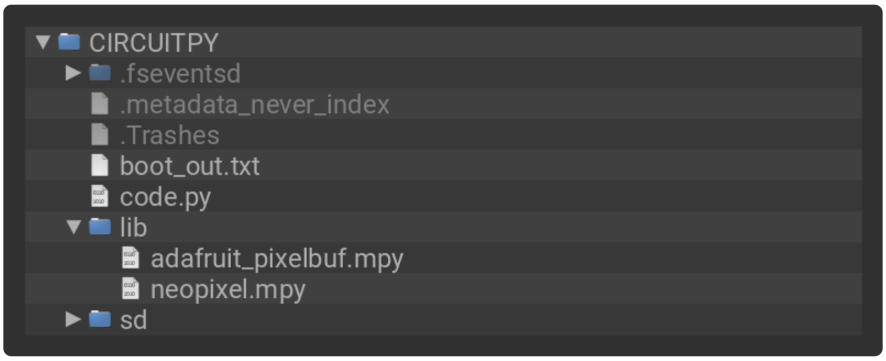
Then, you must include an extra number in every color tuple you create. For example, red will be `(255, 0, 0, 0)`. This is how you send the fourth piece of information. Check out the example below to see how our NeoPixel code looks for using with RGBW LEDs!

The Code

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_Essentials/CircuitPython_NeoPixel_RGBW/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials NeoPixel RGBW example"""
import time
import board
import neopixel

pixel_pin = board.A1
```

```
num_pixels = 8

pixels = neopixel.NeoPixel(pixel_pin, num_pixels, brightness=0.3, auto_write=False,
                           pixel_order=(1, 0, 2, 3))


def colorwheel(pos):
    # Input a value 0 to 255 to get a color value.
    # The colours are a transition r - g - b - back to r.
    if pos < 0 or pos > 255:
        return (0, 0, 0, 0)
    if pos < 85:
        return (255 - pos * 3, pos * 3, 0, 0)
    if pos < 170:
        pos -= 85
        return (0, 255 - pos * 3, pos * 3, 0)
    pos -= 170
    return (pos * 3, 0, 255 - pos * 3, 0)


def color_chase(color, wait):
    for i in range(num_pixels):
        pixels[i] = color
        time.sleep(wait)
        pixels.show()
    time.sleep(0.5)


def rainbow_cycle(wait):
    for j in range(255):
        for i in range(num_pixels):
            rc_index = (i * 256 // num_pixels) + j
            pixels[i] = colorwheel(rc_index & 255)
        pixels.show()
        time.sleep(wait)


RED = (255, 0, 0, 0)
YELLOW = (255, 150, 0, 0)
GREEN = (0, 255, 0, 0)
CYAN = (0, 255, 255, 0)
BLUE = (0, 0, 255, 0)
PURPLE = (180, 0, 255, 0)

while True:
    pixels.fill(RED)
    pixels.show()
    # Increase or decrease to change the speed of the solid color change.
    time.sleep(1)
    pixels.fill(GREEN)
    pixels.show()
    time.sleep(1)
    pixels.fill(BLUE)
    pixels.show()
    time.sleep(1)

    color_chase(RED, 0.1)  # Increase the number to slow down the color chase
    color_chase(YELLOW, 0.1)
    color_chase(GREEN, 0.1)
    color_chase(CYAN, 0.1)
    color_chase(BLUE, 0.1)
    color_chase(PURPLE, 0.1)

    rainbow_cycle(0)  # Increase the number to slow down the rainbow
```

Read the Docs

For a more in depth look at what `neopixel` can do, check out NeoPixel on Read the Docs (https://adafru.it/C5m).

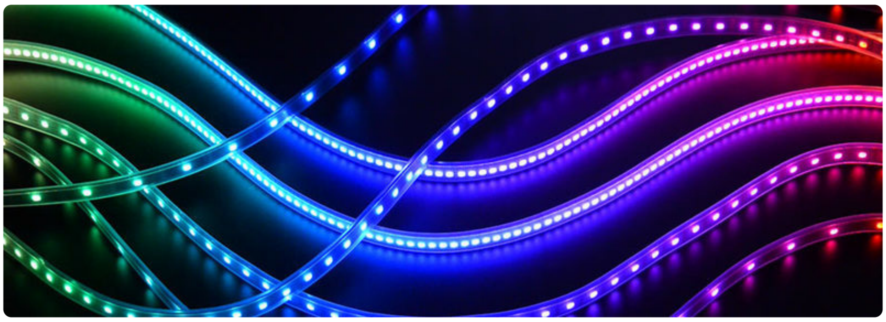
CircuitPython DotStar

DotStars use two wires, unlike NeoPixel's one wire. They're very similar but you can write to DotStars much faster with hardware SPI and they have a faster PWM cycle so they are better for light painting.

Any pins can be used **but** if the two pins can form a hardware SPI port, the library will automatically switch over to hardware SPI. If you use hardware SPI then you'll get 4 MHz clock rate (that would mean updating a 64 pixel strand in about 500uS - that's 0.0005 seconds). If you use non-hardware SPI pins you'll drop down to about 3KHz, 1000 times as slow!

You can drive 300 DotStar LEDs with brightness control (set `brightness=1.0` in object creation) and 1000 LEDs without. That's because to adjust the brightness we have to dynamically recreate the data-stream each write.

You'll need the **adafruit_dotstar.mpy** library if you don't already have it in your /lib folder! You can get it from the CircuitPython Library Bundle (https://adafru.it/ENC). If you need help installing the library, check out the CircuitPython Libraries page (https://adafru.it/ABU).



Wire It Up

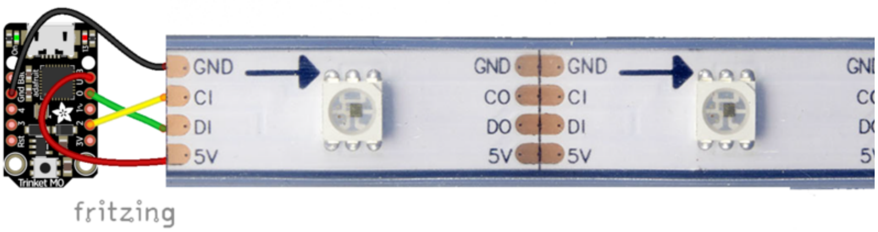
You'll need to solder up your DotStars first. Verify your connection is on the **DATA INPUT** or **DI** and **CLOCK INPUT** or **CI** side. Plugging into the DATA OUT/DO or CLOCK OUT/CO side is a common mistake! The connections are labeled and some formats have arrows to indicate the direction the data must flow. Always verify your wiring with a visual inspection, as the order of the connections can differ from strip to strip!

For powering the pixels from the board, the 3.3V regulator output can handle about 500mA peak which is about 50 pixels with 'average' use. If you want really bright lights and a lot of pixels, we recommend powering direct from an external power source.

- On Gemma M0 and Circuit Playground Express this is the **Vout** pad - that pad has direct power from USB or the battery, depending on which is higher voltage.
- On Trinket M0, Feather M0 Express, Feather M4 Express, ItsyBitsy M0 Express and ItsyBitsy M4 Express the **USB** or **BAT** pins will give you direct power from the USB port or battery.
- On Metro M0 Express and Metro M4 Express, use the **5V** pin regardless of whether it's powered via USB or the DC jack.
- On QT Py M0, use the **5V** pin.

If the power to the DotStars is greater than 5.5V you may have some difficulty driving some strips, in which case you may need to lower the voltage to 4.5-5V or use a level shifter.

Do not use the VIN pin directly on Metro M0 Express or Metro M4 Express! The voltage can reach 9V and this can destroy your DotStars!



Note that the wire ordering on your DotStar strip or shape may not exactly match the diagram above. Check the markings to verify which pin is DIN, CIN, 5V and GND

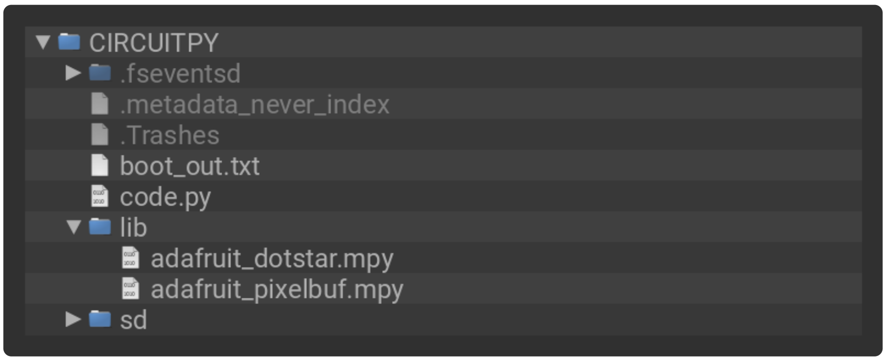
The Code

This example includes multiple visual effects.

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_Essentials/CircuitPython_DotStar/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials DotStar example"""
import time
from rainbowio import colorwheel
import adafruit_dotstar
import board

num_pixels = 30
pixels = adafruit_dotstar.DotStar(board.A1, board.A2, num_pixels, brightness=0.1,
auto_write=False)


def color_fill(color, wait):
    pixels.fill(color)
    pixels.show()
    time.sleep(wait)


def slice_alternating(wait):
    pixels[::2] = [RED] * (num_pixels // 2)
    pixels.show()
    time.sleep(wait)
    pixels[1::2] = [ORANGE] * (num_pixels // 2)
    pixels.show()
    time.sleep(wait)
    pixels[::2] = [YELLOW] * (num_pixels // 2)
    pixels.show()
    time.sleep(wait)
    pixels[1::2] = [GREEN] * (num_pixels // 2)
    pixels.show()
    time.sleep(wait)
    pixels[::2] = [TEAL] * (num_pixels // 2)
    pixels.show()
    time.sleep(wait)
    pixels[1::2] = [CYAN] * (num_pixels // 2)
    pixels.show()
```

```
    time.sleep(wait)
    pixels[::2] = [BLUE] * (num_pixels // 2)
    pixels.show()
    time.sleep(wait)
    pixels[1::2] = [PURPLE] * (num_pixels // 2)
    pixels.show()
    time.sleep(wait)
    pixels[::2] = [MAGENTA] * (num_pixels // 2)
    pixels.show()
    time.sleep(wait)
    pixels[1::2] = [WHITE] * (num_pixels // 2)
    pixels.show()
    time.sleep(wait)


def slice_rainbow(wait):
    pixels[::6] = [RED] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)
    pixels[1::6] = [ORANGE] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)
    pixels[2::6] = [YELLOW] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)
    pixels[3::6] = [GREEN] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)
    pixels[4::6] = [BLUE] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)
    pixels[5::6] = [PURPLE] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)


def rainbow_cycle(wait):
    for j in range(255):
        for i in range(num_pixels):
            rc_index = (i * 256 // num_pixels) + j
            pixels[i] = colorwheel(rc_index & 255)
        pixels.show()
        time.sleep(wait)


RED = (255, 0, 0)
YELLOW = (255, 150, 0)
ORANGE = (255, 40, 0)
GREEN = (0, 255, 0)
TEAL = (0, 255, 120)
CYAN = (0, 255, 255)
BLUE = (0, 0, 255)
PURPLE = (180, 0, 255)
MAGENTA = (255, 0, 20)
WHITE = (255, 255, 255)

while True:
    # Change this number to change how long it stays on each solid color.
    color_fill(RED, 0.5)
    color_fill(YELLOW, 0.5)
    color_fill(ORANGE, 0.5)
    color_fill(GREEN, 0.5)
    color_fill(TEAL, 0.5)
    color_fill(CYAN, 0.5)
    color_fill(BLUE, 0.5)
    color_fill(PURPLE, 0.5)
    color_fill(MAGENTA, 0.5)
    color_fill(WHITE, 0.5)
```

```
    # Increase or decrease this to speed up or slow down the animation.
    slice_alternating(0.1)

    color_fill(WHITE, 0.5)

    # Increase or decrease this to speed up or slow down the animation.
    slice_rainbow(0.1)

    time.sleep(0.5)

    # Increase this number to slow down the rainbow animation.
    rainbow_cycle(0)
```

We've chosen pins A1 and A2, but these are not SPI pins on all boards. DotStars respond faster when using hardware SPI!

Create the LED

The first thing we'll do is create the LED object. The DotStar object has three required arguments and two optional arguments. You are required to set the pin you're using for data, set the pin you'll be using for clock, and provide the number of pixels you intend to use. You can optionally set `brightness` and `auto_write`.

DotStars can be driven by any two pins. We've chosen **A1** for clock and **A2** for data. To set the pins, include the pin names at the beginning of the object creation, in this case `board.A1` and `board.A2`.

To provide the number of pixels, assign the variable `num_pixels` to the number of pixels you'd like to use. In this example, we're using a strip of `72`.

We've chosen to set `brightness=0.1`, or 10%.

By default, `auto_write=True`, meaning any changes you make to your pixels will be sent automatically. Since `True` is the default, if you use that setting, you don't need to include it in your LED object at all. We've chosen to set `auto_write=False`. If you set `auto_write=False`, you must include `pixels.show()` each time you'd like to send data to your pixels. This makes your code more complicated, but it can make your LED animations faster!

DotStar Helpers

We've included a few helper functions to create the super fun visual effects found in this code.

First is `wheel()` which we just learned with the Internal RGB LED (https://adafru.it/Bel). Then we have `color_fill()` which requires you to provide a `color` and the length of time you'd like it to be displayed. Next, are `slice_alternating()`, `slice_rainbow()`, and `rainbow_cycle()` which require you to provide the amount of time in seconds you'd between each step of the animation.

Last, we've included a list of variables for our colors. This makes it much easier if to reuse the colors anywhere in the code, as well as add more colors for use in multiple places. Assigning and using RGB colors is explained in this section of the CircuitPython Internal RGB LED page (https://adafru.it/Bel).

The two slice helpers utilise a nifty feature of the DotStar library that allows us to use math to light up LEDs in repeating patterns. `slice_alternating()` first lights up the even number LEDs and then the odd number LEDs and repeats this back and forth. `slice_rainbow()` lights up every sixth LED with one of the six rainbow colors until the strip is filled. Both use our handy color variables. This slice code only works when the total number of LEDs is divisible by the slice size, in our case 2 and 6. DotStars come in strips of 30, 60, 72, and 144, all of which are divisible by 2 and 6. In the event that you cut them into different sized strips, the code in this example may not work without modification. However, as long as you provide a total number of LEDs that is divisible by the slices, the code will work.

Main Loop

Our main loop begins by calling `color_fill()` once for each `color` on our list and sets each to hold for `0.5` seconds. You can change this number to change how fast each color is displayed. Next, we call `slice_alternating(0.1)`, which means there's a 0.1 second delay between each change in the animation. Then, we fill the strip white to create a clean backdrop for the rainbow to display. Then, we call `slice_rainbow(0.1)`, for a 0.1 second delay in the animation. Last we call `rainbow_cycle(0)`, which means it's as fast as it can possibly be. Increase or decrease either of these numbers to speed up or slow down the animations!

Note that the longer your strip of LEDs is, the longer it will take for the animations to complete.

We have a ton more information on general purpose DotStar know-how at our DotStar UberGuide https://learn.adafruit.com/adafruit-dotstar-leds

Is it SPI?

We explained at the beginning of this section that the LEDs respond faster if you're using hardware SPI. On some of the boards, there are HW SPI pins directly available in the form of MOSI and SCK. However, hardware SPI is available on more than just those pins. But, how can you figure out which? Easy! We wrote a handy script.

We chose pins **A1** and **A2** for our example code. To see if these are hardware SPI on the board you're using, copy and paste the code into **code.py** using your favorite editor, and save the file. Then connect to the serial console to see the results.

To check if other pin combinations have hardware SPI, change the pin names on the line reading: `if is_hardware_SPI(board.A1, board.A2):` to the pins you want to use. Then, check the results in the serial console. Super simple!

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/SPI_Test_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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Hardware SPI pin verification script"""
import board
import busio


def is_hardware_spi(clock_pin, data_pin):
    try:
        p = busio.SPI(clock_pin, data_pin)
        p.deinit()
        return True
    except ValueError:
        return False
```

```
# Provide the two pins you intend to use.
if is_hardware_spi(board.A1, board.A2):
    print("This pin combination is hardware SPI!")
else:
    print("This pin combination isn't hardware SPI.")
```

Read the Docs

For a more in depth look at what `dotstar` can do, check out [DotStar on Read the Docs](https://adafru.it/C4d) (https://adafru.it/C4d).

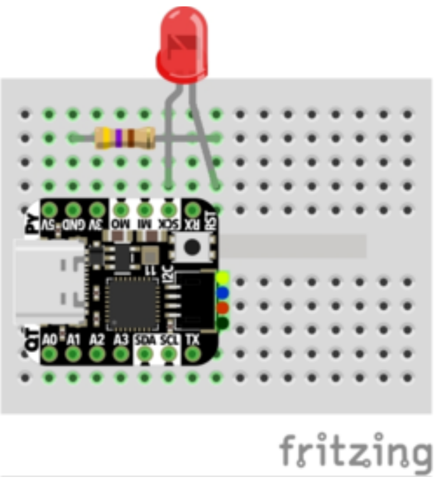
CircuitPython UART Serial

In addition to the USB-serial connection you use for the REPL, there is also a hardware UART you can use. This is handy to talk to UART devices like GPSs, some sensors, or other microcontrollers!

This quick-start example shows how you can create a UART device for communicating with hardware serial devices.

To use this example, you'll need something to generate the UART data. We've used a GPS! Note that the GPS will give you UART data without getting a fix on your location. You can use this example right from your desk! You'll have data to read, it simply won't include your actual location.

> The QT Py M0 does not have a little red LED. Therefore, you must connect an external LED and edit this example for it to work. Follow the wiring diagram and steps below to run this example on QT Py M0.



LED + to **QT Py SCK**
LED - to **470Ω resistor**
470Ω resistor to **QT Py GND**

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/CircuitPython_UART/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials UART Serial example"""
import board
import busio
import digitalio

# For most CircuitPython boards:
led = digitalio.DigitalInOut(board.LED)
# For QT Py M0:
# led = digitalio.DigitalInOut(board.SCK)
led.direction = digitalio.Direction.OUTPUT

uart = busio.UART(board.TX, board.RX, baudrate=9600)

while True:
    data = uart.read(32)  # read up to 32 bytes
    # print(data)  # this is a bytearray type

    if data is not None:
        led.value = True

        # convert bytearray to string
        data_string = ''.join([chr(b) for b in data])
        print(data_string, end="")

        led.value = False
```

Note: To "comment out" a line, put a # and a space before it. To "uncomment" a line, remove the # + space from the beginning of the line.

For **QT Py M0**, you'll need to comment out `led = DigitalInOut(board.LED)` and uncomment `led = DigitalInOut(board.SCK)`. The UART code remains the same.

The Code

First we create the UART object. We provide the pins we'd like to use, `board.TX` and `board.RX`, and we set the `baudrate=9600`. While these pins are labeled on most of the boards, be aware that RX and TX are not labeled on Gemma, and are labeled on the bottom of Trinket. See the diagrams below for help with finding the correct pins on your board.

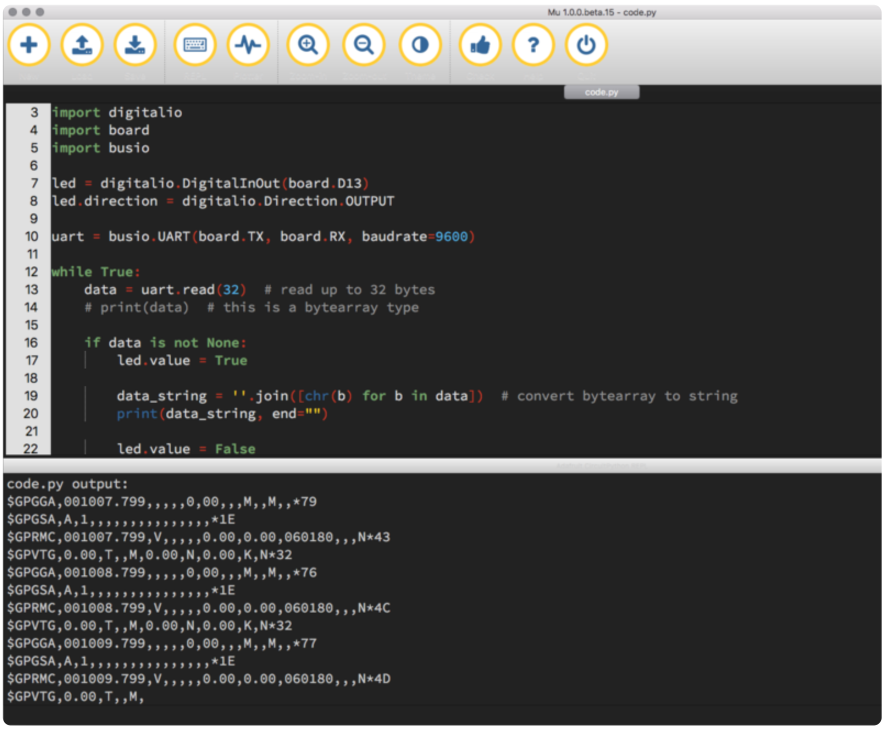
Once the object is created you read data in with `read(numbytes)` where you can specify the max number of bytes. It will return a byte array type object if anything was received already. Note it will always return immediately because there is an internal buffer! So read as much data as you can 'digest'.

If there is no data available, `read()` will return `None`, so check for that before continuing.

The data that is returned is in a byte array, if you want to convert it to a string, you can use this handy line of code which will run `chr()` on each byte:

```
datastr = ''.join([chr(b) for b in data]) # convert bytearray to string
```

Your results will look something like this:



For more information about the data you're reading and the Ultimate GPS, check out the Ultimate GPS guide: https://learn.adafruit.com/adafruit-ultimate-gps

Wire It Up

You'll need a couple of things to connect the GPS to your board.

For Gemma M0 and Circuit Playground Express, you can use use alligator clips to connect to the Flora Ultimate GPS Module.

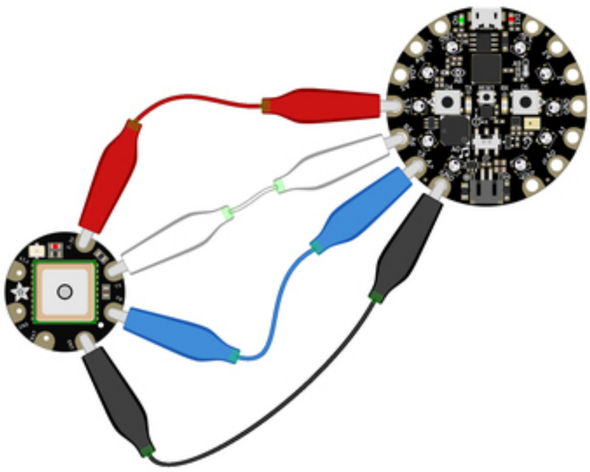
For Trinket M0, Feather M0 Express, Metro M0 Express and ItsyBitsy M0 Express, you'll need a breadboard and jumper wires to connect to the Ultimate GPS Breakout.

We've included diagrams show you how to connect the GPS to your board. In these diagrams, the wire colors match the same pins on each board.

- The **black** wire connects between the **ground** pins.
- The **red** wire connects between the **power** pins on the GPS and your board.
- The **blue** wire connects from **TX** on the GPS to **RX** on your board.
- The **white** wire connects from **RX** on the GPS to **TX** on your board.

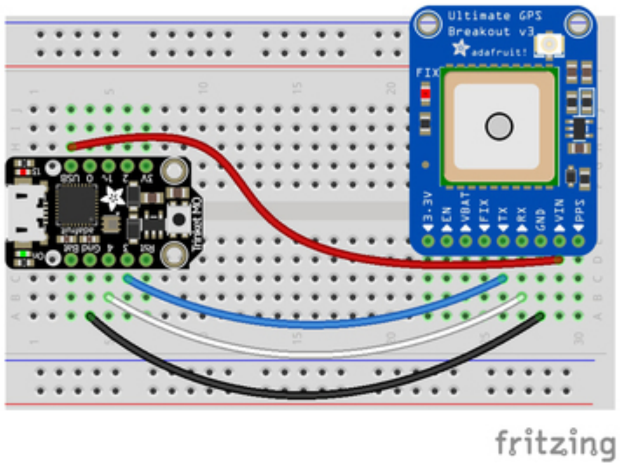
Check out the list below for a diagram of your specific board!

Watch out! A common mixup with UART serial is that RX on one board connects to TX on the other! However, sometimes boards have RX labeled TX and vice versa. So, you'll want to start with RX connected to TX, but if that doesn't work, try the other way around!



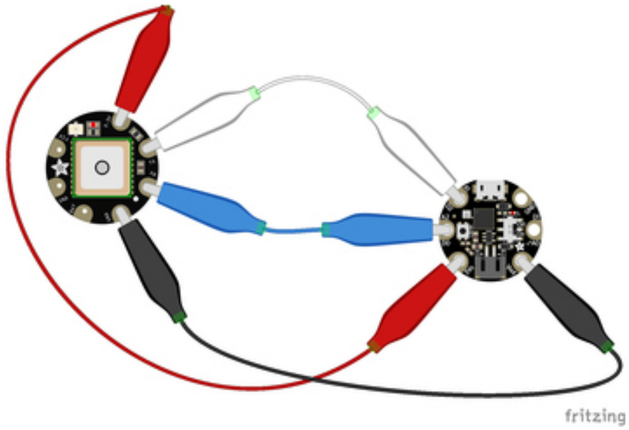
Circuit Playground Express and Circuit Playground Bluefruit

Connect **3.3v** on your CPX to **3.3v** on your GPS.
Connect **GND** on your CPX to **GND** on your GPS.
Connect **RX/A6** on your CPX to **TX** on your GPS.
Connect **TX/A7** on your CPX to **RX** on your GPS.



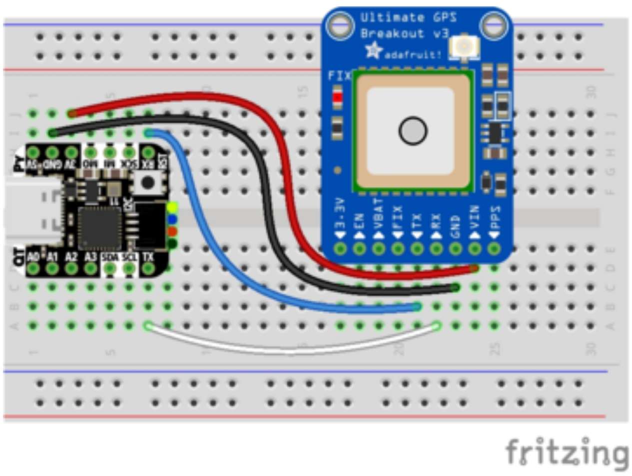
Trinket M0

Connect **USB** on the Trinket to **VIN** on the GPS.
Connect **Gnd** on the Trinket to **GND** on the GPS.
Connect **D3** on the Trinket to **TX** on the GPS.
Connect **D4** on the Trinket to **RX** on the GPS.



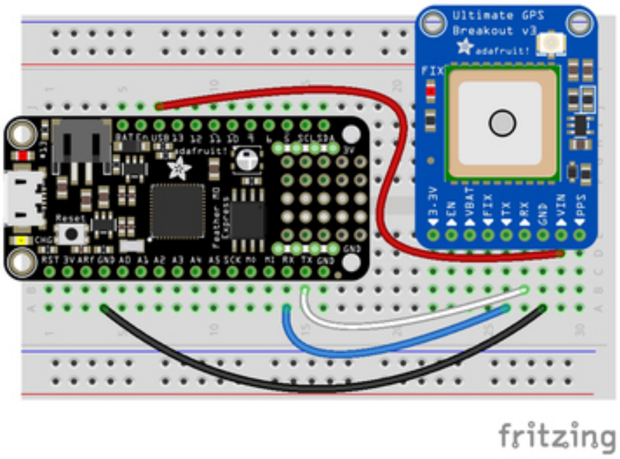
Gemma M0

Connect **3vo** on the Gemma to **3.3v** on the GPS.
Connect **GND** on the Gemma to **GND** on the GPS.
Connect **A1/D2** on the Gemma to **TX** on the GPS.
Connect **A2/D0** on the Gemma to **RX** on the GPS.



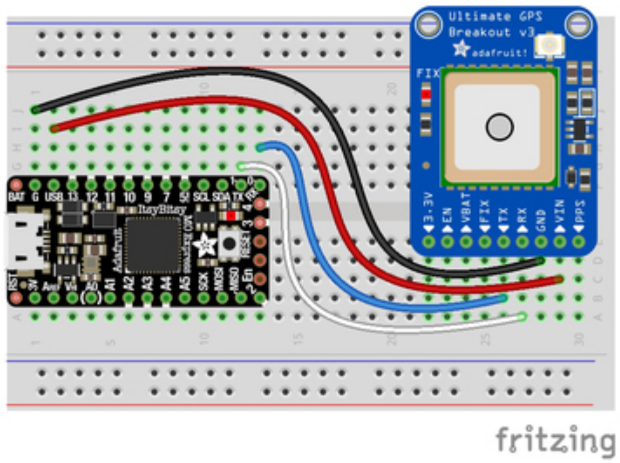
QT Py M0

Connect **3V** on the QT Py to **VIN** on the GPS.
Connect **GND** on the QT Py to **GND** on the GPS.
Connect **RX** on the QT Py to **TX** on the GPS.
Connect **TX** on the QT Py to **RX** on the GPS.



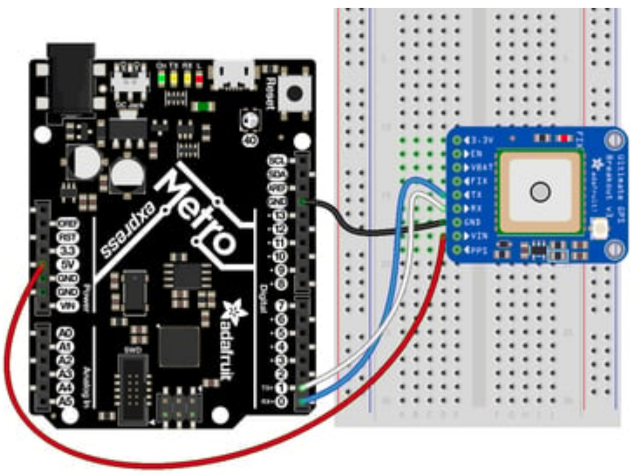
Feather M0 Express and Feather M4 Express

Connect **USB** on the Feather to **VIN** on the GPS.
Connect **GND** on the Feather to **GND** on the GPS.
Connect **RX** on the Feather to **TX** on the GPS.
Connect **TX** on the Feather to **RX** on the GPS.



ItsyBitsy M0 Express and ItsyBitsy M4 Express

Connect **USB** on the ItsyBitsy to **VIN** on the GPS
Connect **G** on the ItsyBitsy to **GND** on the GPS.
Connect **RX/0** on the ItsyBitsy to **TX** on the GPS.
Connect **TX/1** on the ItsyBitsy to **RX** on the GPS.



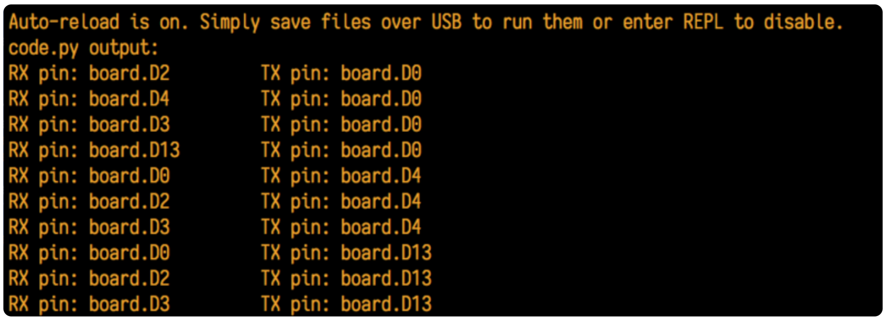
Metro M0 Express and Metro M4 Express

Connect **5V** on the Metro to **VIN** on the GPS.
Connect **GND** on the Metro to **GND** on the GPS.
Connect **RX/D0** on the Metro to **TX** on the GPS.
Connect **TX/D1** on the Metro to **RX** on the GPS.

Where's my UART?

On the SAMD21, we have the flexibility of using a wide range of pins for UART. Compare this to some chips like the ESP8266 with fixed UART pins. The good news is you can use many but not all pins. Given the large number of SAMD boards we have, its impossible to guarantee anything other than the labeled 'TX' and 'RX'. So, if you want some other setup, or multiple UARTs, how will you find those pins? Easy! We've written a handy script.

These are the results from a Trinket M0, your output may vary and it might be very long. [For more details about UARTs and SERCOMs check out our detailed guide here](https://adafru.it/Ben) (https://adafru.it/Ben)



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/UART_Test_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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials UART possible pin-pair identifying script"""
import board
import busio
from microcontroller import Pin


def is_hardware_uart(tx, rx):
    try:
        p = busio.UART(tx, rx)
        p.deinit()
        return True
    except ValueError:
        return False


def get_unique_pins():
```

```
    exclude = ['NEOPIXEL', 'APA102_MOSI', 'APA102_SCK']
    pins = [pin for pin in [
        getattr(board, p) for p in dir(board) if p not in exclude]
            if isinstance(pin, Pin)]
    unique = []
    for p in pins:
        if p not in unique:
            unique.append(p)
    return unique


for tx_pin in get_unique_pins():
    for rx_pin in get_unique_pins():
        if rx_pin is tx_pin:
            continue
        if is_hardware_uart(tx_pin, rx_pin):
            print("RX pin:", rx_pin, "\t TX pin:", tx_pin)
```

Trinket M0: Create UART before I2C

On the Trinket M0 (only), if you are using both UART and I2C, you must create the UART object first, e.g.:

```
&gt;&gt;&gt; import board
&gt;&gt;&gt; uart = board.UART()   # Uses pins 4 and 3 for TX and RX, baudrate 9600.
&gt;&gt;&gt; i2c = board.I2C()     # Uses pins 2 and 0 for SCL and SDA.

# or alternatively,
```

Creating the I2C object first does not work:

```
&gt;&gt;&gt; import board
&gt;&gt;&gt; i2c = board.I2C()     # Uses pins 2 and 0 for SCL and SDA.
&gt;&gt;&gt; uart = board.UART()   # Uses pins 4 and 3 for TX and RX, baudrate 9600.
Traceback (most recent call last):
File "", line 1, in
ValueError: Invalid pins
```

CircuitPython I2C

I2C is a 2-wire protocol for communicating with simple sensors and devices, meaning it uses two connections for transmitting and receiving data. There are many I2C devices available and they're really easy to use with CircuitPython. We have libraries available for many I2C devices in the library bundle (https://adafru.it/ENC). (If you don't see the sensor you're looking for, keep checking back, more are being written all the time!)

In this section, we're going to do is learn how to scan the I2C bus for all connected devices. Then we're going to learn how to interact with an I2C device.

We'll be using the Adafruit TSL2591 (http://adafru.it/1980), a common, low-cost light sensor. While the exact code we're running is specific to the TSL2591 the overall process is the same for just about any I2C sensor or device.

These examples will use the TSL2591 lux sensor breakout. The first thing you'll want to do is get the sensor connected so your board has I2C to talk to.

Wire It Up

You'll need a couple of things to connect the TSL2591 to your board. The TSL2591 comes with STEMMA QT / QWIIC connectors on it, which makes it super simple to wire it up. No further soldering required!

For Gemma M0, Circuit Playground Express and Circuit Playground Bluefruit, you can use use the STEMMA QT to alligator clips cable (http://adafru.it/4398) to connect to the TSL2591.

For Trinket M0, Feather M0 and M4 Express, Metro M0 and M4 Express and ItsyBitsy M0 and M4 Express, you'll need a breadboard and STEMMA QT to male jumper wires cable (http://adafru.it/4209) to connect to the TSL2591.

For QT Py M0, you'll need a STEMMA QT cable (http://adafru.it/4210) to connect to the TSL2591.

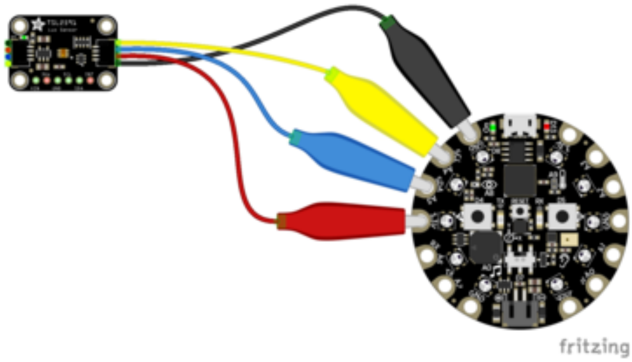
We've included diagrams show you how to connect the TSL2591 to your board. In these diagrams, the wire colors match the STEMMA QT cables and connect to the same pins on each board.

- The **black** wire connects from **GND** on the TSL2591 to **ground** on your board.
- The **red** wire connects from **VIN** on the TSL2591 to **power** on your board.
- The **yellow** wire connects from **SCL** on the TSL2591 to **SCL** on your board.
- The **blue** wire connects from **SDA** on the TSL2591 to **SDA** on your board.

Check out the list below for a diagram of your specific board!

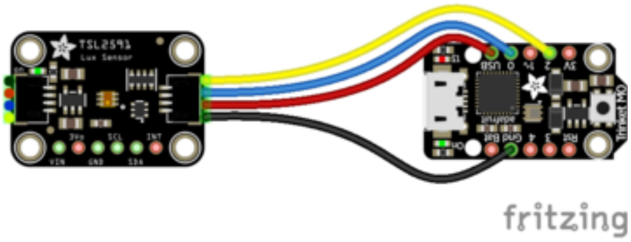
Be aware that the Adafruit microcontroller boards do not have I2C pullup resistors built in! All of the Adafruit breakouts do, but if you're building your own board or using a non-Adafruit breakout, you must add 2.2K-10K ohm pullups on both SDA and SCL to the 3.3V.

Circuit Playground Express and Circuit Playground Bluefruit



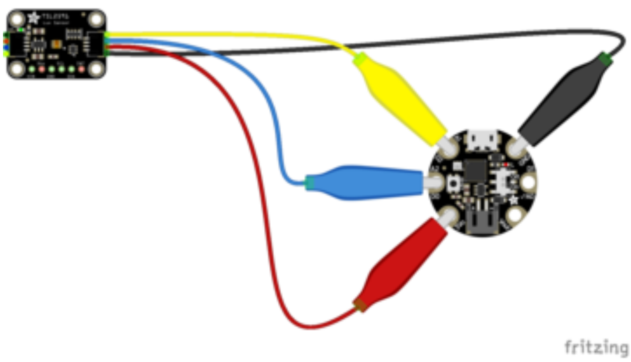
Connect **3.3v** on your CPX to **3.3v** on your TSL2591.
Connect **GND** on your CPX to **GND** on your TSL2591.
Connect **SCL/A4** on your CPX to **SCL** on your TSL2591.
Connect **SDL/A5** on your CPX to **SDA** on your TSL2591.

Trinket M0



Connect **USB** on the Trinket to **VIN** on the TSL2591.
Connect **Gnd** on the Trinket to **GND** on the TSL2591.
Connect **D2** on the Trinket to **SCL** on the TSL2591.
Connect **D0** on the Trinket to **SDA** on the TSL2591.

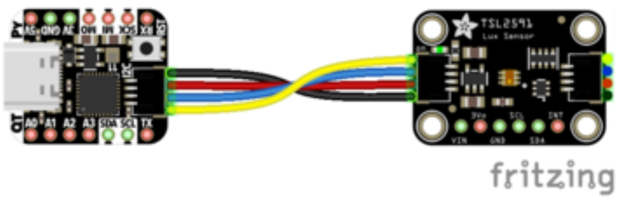
Gemma M0



Connect **3vo** on the Gemma to **3V** on the TSL2591.
Connect **GND** on the Gemma to **GND** on the TSL2591.
Connect **A1/D2** on the Gemma to **SCL** on the TSL2591.
Connect **A2/D0** on the Gemma to **SDA** on the TSL2591.

QT Py M0

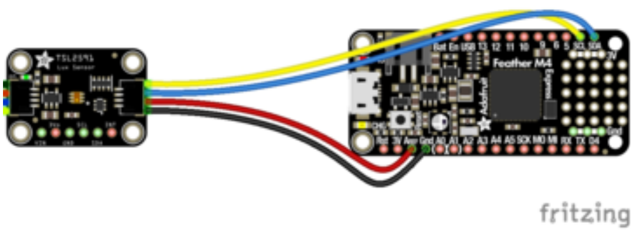
If using the STEMMA QT cable:



Connect the **STEMMA QT cable** from **the connector on the QT Py** to **the connector on the TSL2591.**
Alternatively, if using a breadboard:

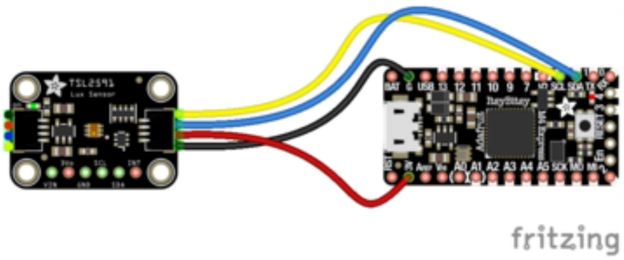
Connect **3V** on the QT Py to **VIN** on the TSL2591.
Connect **GND** on the QT Py to **GND** on the TSL2591.
Connect **SCL** on the QT Py to **SCL** on the TSL2591.
Connect **SDA** on the QT Py to **SDA** on the TSL2591.

Feather M0 Express and Feather M4 Express



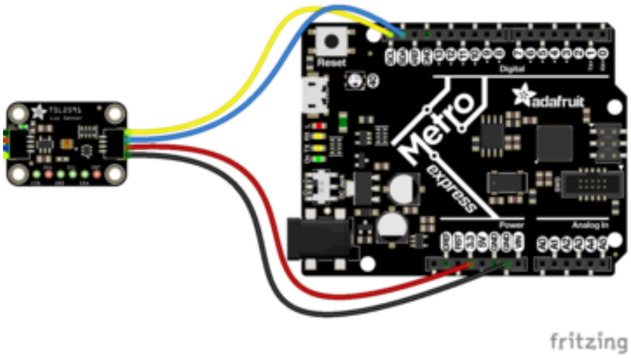
Connect **USB** on the Feather to **VIN** on the TSL2591.
Connect **GND** on the Feather to **GND** on the TSL2591.
Connect **SCL** on the Feather to **SCL** on the TSL2591.
Connect **SDA** on the Feather to **SDA** on the TSL2591.

ItsyBitsy M0 Express and ItsyBitsy M4 Express



Connect **USB** on the ItsyBitsy to **VIN** on the TSL2591
Connect **G** on the ItsyBitsy to **GND** on the TSL2591.
Connect **SCL** on the ItsyBitsy to **SCL** on the TSL2591.
Connect **SDA** on the ItsyBitsy to **SDA** on the TSL2591.

Metro M0 Express and Metro M4 Express



Connect **5V** on the Metro to **VIN** on the TSL2591.
Connect **GND** on the Metro to **GND** on the TSL2591.
Connect **SCL** on the Metro to **SCL** on the TSL2591.
Connect **SDA** on the Metro to **SDA** on the TSL2591.

Find Your Sensor

The first thing you'll want to do after getting the sensor wired up, is make sure it's wired correctly. We're going to do an I2C scan to see if the board is detected, and if it is, print out its I2C address.

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/CircuitPython_I2C_Scan/** 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: 2017 Limor Fried for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython I2C Device Address Scan"""
# If you run this and it seems to hang, try manually unlocking
# your I2C bus from the REPL with
#  >>> import board
#  >>> board.I2C().unlock()

import time
import board

# To use default I2C bus (most boards)
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.SCL1, board.SDA1)  # QT Py RP2040 STEMMA connector
# 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()
```

First we 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, we need to lock the I2C down so the only thing accessing it is the code. So next we include a loop that waits until I2C is locked and then continues on to the scan function.

Last, we have the loop that runs the actual scan, `i2c_scan()`. Because I2C typically refers to addresses in hex form, we've included 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. We've connected the TSL2591 which has a 7-bit I2C address of 0x29. The result for this sensor is `I2C addresses found: ['0x29']`. If no addresses are returned, refer back to the wiring diagrams to make sure you've wired up your sensor correctly.

I2C Sensor Data

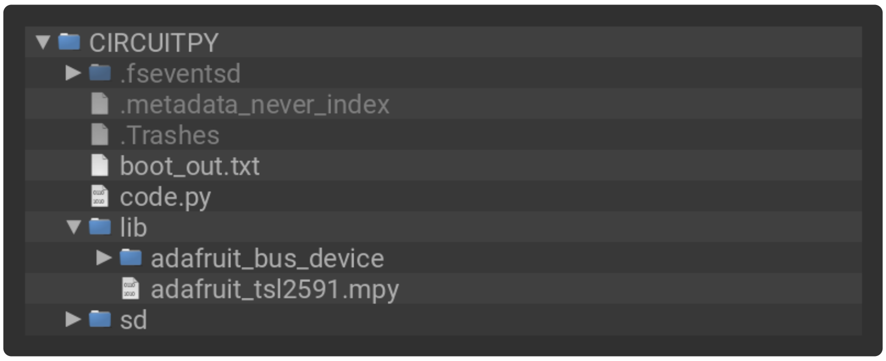
Now we know for certain that our sensor is connected and ready to go. Let's find out how to get the data from our sensor!

Installing Project Code

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_Essentials/CircuitPython_I2C_TSL2591/** 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:



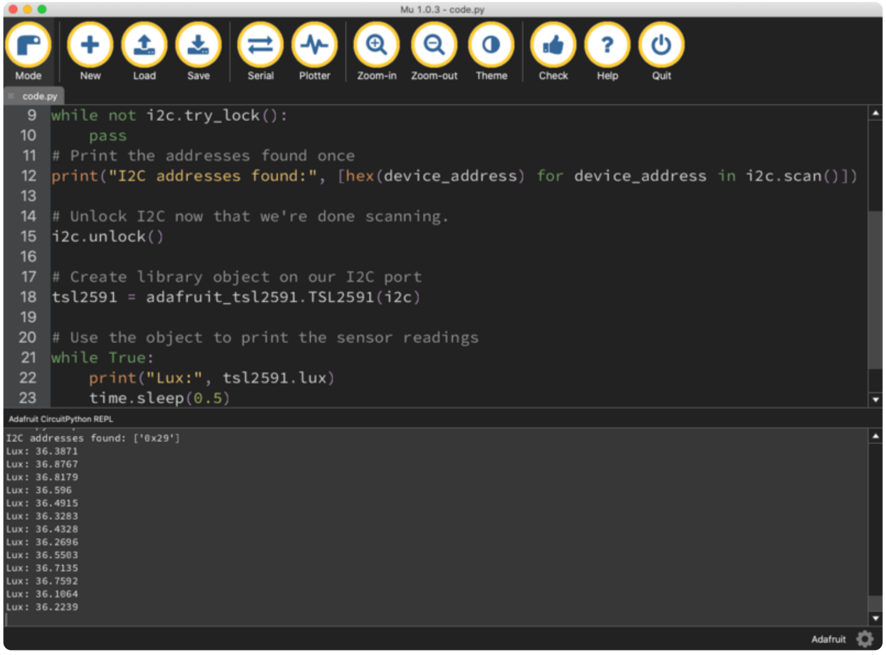

```
# SPDX-FileCopyrightText: 2017 Limor Fried for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials I2C sensor example using TSL2591"""
```

```
import time
import board
import adafruit_tsl2591

i2c = board.I2C()  # uses board.SCL and board.SDA
# i2c = board.STEMMA_I2C()  # For using the built-in STEMMA QT connector on a
microcontroller

# Lock the I2C device before we try to scan
while not i2c.try_lock():
    pass
# Print the addresses found once
print("I2C addresses found:", [hex(device_address) for device_address in
i2c.scan()])

# Unlock I2C now that we're done scanning.
i2c.unlock()

# Create library object on our I2C port
tsl2591 = adafruit_tsl2591.TSL2591(i2c)

# Use the object to print the sensor readings
while True:
    print("Lux:", tsl2591.lux)
    time.sleep(0.5)
```

This code begins the same way as the scan code. We've included the scan code so you have verification that your sensor is wired up correctly and is detected. It prints the address once. After the scan, we unlock I2C with `i2c_unlock()` so we can use the sensor for data.

We create our sensor object using the sensor library. We call it `tsl2591` and provide it the `i2c` object.

Then we have a simple loop that prints out the lux reading using the sensor object we created. We add a `time.sleep(1.0)`, so it only prints once per second. Connect to the serial console to see the results. Try shining a light on it to see the results change!

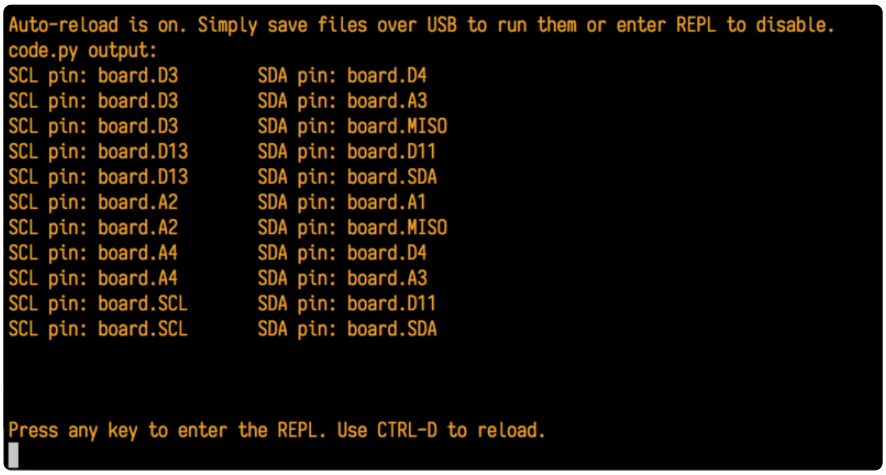


Where's my I2C?

On the SAMD21, SAMD51 and nRF52840, we have the flexibility of using a wide range of pins for I2C. On the nRF52840, any pin can be used for I2C! Some chips, like the ESP8266, require using bitbangio, but can also use any pins for I2C. There's some other chips that may have fixed I2C pin.

The good news is you can use many but not all pins. Given the large number of SAMD boards we have, its impossible to guarantee anything other than the labeled 'SDA' and 'SCL'. So, if you want some other setup, or multiple I2C interfaces, how will you find those pins? Easy! We've written a handy script.

These are the results from an ItsyBitsy M0 Express. Your output may vary and it might be very long. For more details about I2C and SERCOMs, check out our detailed guide here (https://adafru.it/Ben).



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/I2C_Test_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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials I2C possible pin-pair identifying script"""
import board
import busio
from microcontroller import Pin


def is_hardware_I2C(scl, sda):
    try:
        p = busio.I2C(scl, sda)
        p.deinit()
        return True
    except ValueError:
        return False
    except RuntimeError:
        return True


def get_unique_pins():
    exclude = ['NEOPIXEL', 'APA102_MOSI', 'APA102_SCK']
    pins = [pin for pin in [
        getattr(board, p) for p in dir(board) if p not in exclude]
            if isinstance(pin, Pin)]
    unique = []
    for p in pins:
        if p not in unique:
            unique.append(p)
    return unique


for scl_pin in get_unique_pins():
    for sda_pin in get_unique_pins():
        if scl_pin is sda_pin:
            continue
        if is_hardware_I2C(scl_pin, sda_pin):
            print("SCL pin:", scl_pin, "\t SDA pin:", sda_pin)
```

CircuitPython HID Keyboard and Mouse

One of the things we baked into CircuitPython is 'HID' (**H**uman **I**nterface **D**evice) control - that means keyboard and mouse capabilities. This means your CircuitPython board can act like a keyboard device and press key commands, or a mouse and have it move the mouse pointer around and press buttons. This is really handy because even if you cannot adapt your software to work with hardware, there's almost always a keyboard interface - so if you want to have a capacitive touch interface for a game, say, then keyboard emulation can often get you going really fast!

This section walks you through the code to create a keyboard or mouse emulator. First we'll go through an example that uses pins on your board to emulate keyboard input. Then, we will show you how to wire up a joystick to act as a mouse, and cover the code needed to make that happen.

CircuitPython Keyboard Emulator

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.

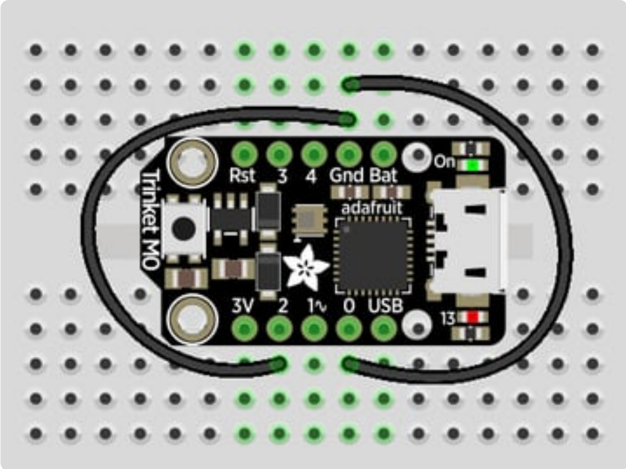
Thankfully, we can do this in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Essentials/CircuitPython_HID_Keyboard/** 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: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials HID Keyboard example"""
import time
```

```
import board
import digitalio
import usb_hid
from adafruit_hid.keyboard import Keyboard
from adafruit_hid.keyboard_layout_us import KeyboardLayoutUS
from adafruit_hid.keycode import Keycode

# A simple neat keyboard demo in CircuitPython

# The pins we'll use, each will have an internal pullup
keypress_pins = [board.A1, board.A2]
# Our array of key objects
key_pin_array = []
# The Keycode sent for each button, will be paired with a control key
keys_pressed = [Keycode.A, "Hello World!\n"]
control_key = Keycode.SHIFT

# The keyboard object!
time.sleep(1)  # Sleep for a bit to avoid a race condition on some systems
keyboard = Keyboard(usb_hid.devices)
keyboard_layout = KeyboardLayoutUS(keyboard)  # We're in the US :)

# Make all pin objects inputs with pullups
for pin in keypress_pins:
    key_pin = digitalio.DigitalInOut(pin)
    key_pin.direction = digitalio.Direction.INPUT
    key_pin.pull = digitalio.Pull.UP
    key_pin_array.append(key_pin)

# For most CircuitPython boards:
led = digitalio.DigitalInOut(board.LED)
# For QT Py M0:
# led = digitalio.DigitalInOut(board.SCK)
led.direction = digitalio.Direction.OUTPUT

print("Waiting for key pin...")

while True:
    # Check each pin
    for key_pin in key_pin_array:
        if not key_pin.value:  # Is it grounded?
            i = key_pin_array.index(key_pin)
            print("Pin #%d is grounded." % i)

            # Turn on the red LED
            led.value = True

            while not key_pin.value:
                pass  # Wait for it to be ungrounded!
            # "Type" the Keycode or string
            key = keys_pressed[i]  # Get the corresponding Keycode or string
            if isinstance(key, str):  # If it's a string...
                keyboard_layout.write(key)  # ...Print the string
            else:  # If it's not a string...
                keyboard.press(control_key, key)  # "Press"...
                keyboard.release_all()  # ..."Release"!

            # Turn off the red LED
            led.value = False

    time.sleep(0.01)
```

Connect pin **A1** or **A2** to ground, using a wire or alligator clip, then disconnect it to send the key press "A" or the string "Hello world!"



This wiring example shows A1 and A2 connected to ground.

Remember, on Trinket, A1 and A2 are labeled 2 and 0! On other boards, you will have A1 and A2 labeled as expected.

Create the Objects and Variables

First, we assign some variables for later use. We create three arrays assigned to variables: `keypress_pins`, `key_pin_array`, and `keys_pressed`. The first is the pins we're going to use. The second is empty because we're going to fill it later. The third is what we would like our "keyboard" to output - in this case the letter "A" and the phrase, "Hello world!". We create our last variable assigned to `control_key` which allows us to later apply the shift key to our keypress. We'll be using two keypresses, but you can have up to six keypresses at once.

Next `keyboard` and `keyboard_layout` objects are created. We only have US right now (if you make other layouts please submit a GitHub pull request!). The `time.sleep(1)` avoids an error that can happen if the program gets run as soon as the board gets plugged in, before the host computer finishes connecting to the board.

Then we take the pins we chose above, and create the pin objects, set the direction and give them each a pullup. Then we apply the pin objects to `key_pin_array` so we can use them later.

Next we set up the little red LED to so we can use it as a status light.

The last thing we do before we start our loop is `print`, "Waiting for key pin..." so you know the code is ready and waiting!

The Main Loop

Inside the loop, we check each pin to see if the state has changed, i.e. you connected the pin to ground. Once it changes, it prints, "Pin # grounded." to let you know the ground state has been detected. Then we turn on the red LED. The code waits for the state to change again, i.e. it waits for you to unground the pin by disconnecting the wire attached to the pin from ground.

Then the code gets the corresponding keys pressed from our array. If you grounded and ungrounded A1, the code retrieves the keypress `a`, if you grounded and ungrounded A2, the code retrieves the string, `"Hello world!"`

If the code finds that it's retrieved a string, it prints the string, using the `keyboard_layout` to determine the keypresses. Otherwise, the code prints the keypress from the `control_key` and the keypress "a", which result in "A". Then it calls `keyboard.release_all()`. You always want to call this soon after a keypress or you'll end up with a stuck key which is really annoying!

Instead of using a wire to ground the pins, you can try wiring up buttons like we did in CircuitPython Digital In & Out (https://adafru.it/Beo). Try altering the code to add more pins for more keypress options!

Non-US Keyboard Layouts

The code above uses KeyboardLayoutUS. If you would like to emulate a non-US keyboard, a number of other keyboard layout classes are available (https://adafru.it/UYD).

CircuitPython Mouse Emulator

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_Essentials/CircuitPython_HID_Mouse/** 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:

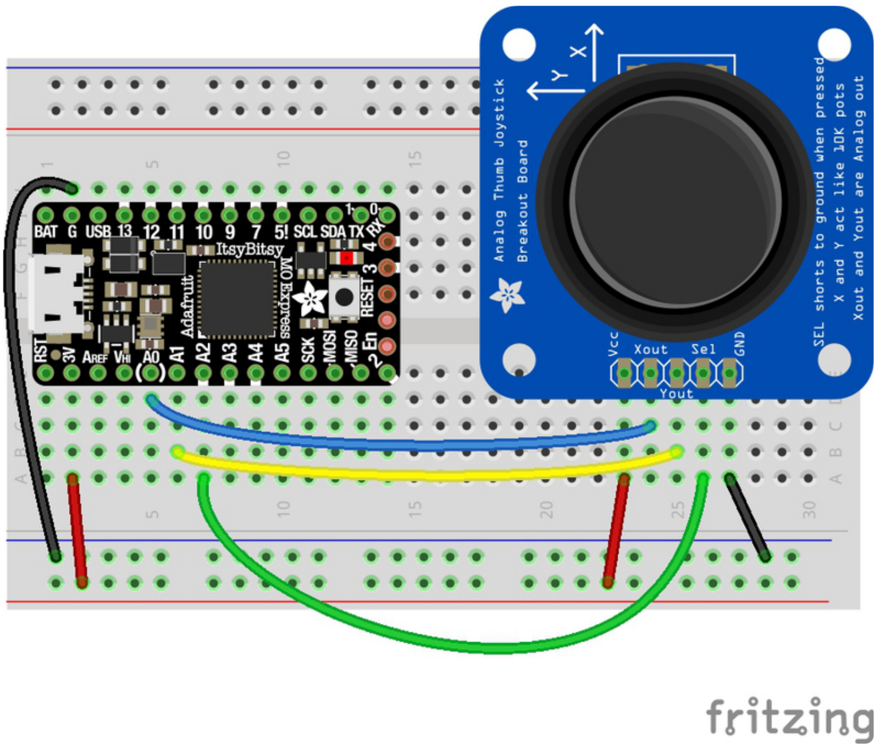

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials HID Mouse example"""
import time
import analogio
import board
import digitalio
import usb_hid
from adafruit_hid.mouse import Mouse

mouse = Mouse(usb_hid.devices)

x_axis = analogio.AnalogIn(board.A0)
y_axis = analogio.AnalogIn(board.A1)
select = digitalio.DigitalInOut(board.A2)
select.direction = digitalio.Direction.INPUT
select.pull = digitalio.Pull.UP

pot_min = 0.00
pot_max = 3.29
step = (pot_max - pot_min) / 20.0


def get_voltage(pin):
    return (pin.value * 3.3) / 65536


def steps(axis):
    """ Maps the potentiometer voltage range to 0-20 """
    return round((axis - pot_min) / step)


while True:
    x = get_voltage(x_axis)
    y = get_voltage(y_axis)

    if select.value is False:
        mouse.click(Mouse.LEFT_BUTTON)
        time.sleep(0.2)  # Debounce delay

    if steps(x) > 11.0:
        # print(steps(x))
        mouse.move(x=1)
    if steps(x) < 9.0:
        # print(steps(x))
        mouse.move(x=-1)

    if steps(x) > 19.0:
        # print(steps(x))
        mouse.move(x=8)
    if steps(x) < 1.0:
        # print(steps(x))
        mouse.move(x=-8)
```

```
    if steps(y) > 11.0:
        # print(steps(y))
        mouse.move(y=-1)
    if steps(y) < 9.0:
        # print(steps(y))
        mouse.move(y=1)

    if steps(y) > 19.0:
        # print(steps(y))
        mouse.move(y=-8)
    if steps(y) < 1.0:
        # print(steps(y))
        mouse.move(y=8)
```

For this example, we've wired up a 2-axis thumb joystick with a select button. We use this to emulate the mouse movement and the mouse left-button click. To wire up this joytick:

- Connect **VCC** on the joystick to the **3V** on your board. Connect **ground** to **ground**.
- Connect **Xout** on the joystick to pin **A0** on your board.
- Connect **Yout** on the joystick to pin **A1** on your board.
- Connect **Sel** on the joystick to pin **A2** on your board.

Remember, Trinket's pins are labeled differently. Check the [Trinket Pinouts page (https://adafru.it/AMd)](https://adafru.it/AMd) to verify your wiring.



To use this demo, simply move the joystick around. The mouse will move slowly if you move the joystick a little off center, and more quickly if you move it as far as it goes. Press down on the joystick to click the mouse. Awesome! Now let's take a look at the code.

Create the Objects and Variables

First we create the mouse object.

Next, we set `x_axis` and `y_axis` to pins `A0` and `A1`. Then we set `select` to `A2`, set it as input and give it a pullup.

The x and y axis on the joystick act like 2 potentiometers. We'll be using them just like we did in [CircuitPython Analog In](https://adafru.it/Bep) (https://adafru.it/Bep). We set `pot_min` and `pot_max` to be the minimum and maximum voltage read from the potentiometers. We assign `step = (pot_max - pot_min) / 20.0` to use in a helper function.

CircuitPython HID Mouse Helpers

First we have the `get_voltage()` helper so we can get the correct readings from the potentiometers. Look familiar? We [learned about it in Analog In](https://adafru.it/Bep) (https://adafru.it/Bep).

Second, we have `steps(axis)`. To use it, you provide it with the axis you're reading. This is where we're going to use the `step` variable we assigned earlier. The potentiometer range is 0-3.29. This is a small range. It's even smaller with the joystick because the joystick sits at the center of this range, 1.66, and the + and - of each axis is above and below this number. Since we need to have thresholds in our code, we're going to map that range of 0-3.29 to while numbers between 0-20.0 using this helper function. That way we can simplify our code and use larger ranges for our thresholds instead of trying to figure out tiny decimal number changes.

Main Loop

First we assign `x` and `y` to read the voltages from `x_axis` and `y_axis`.

Next, we check to see when the state of the select button is `False`. It defaults to `True` when it is not pressed, so if the state is `False`, the button has been pressed. When it's pressed, it sends the command to click the left mouse button. The `time.sleep(0.2)` prevents it from reading multiple clicks when you've only clicked once.

Then we use the `steps()` function to set our mouse movement. There are two sets of two `if` statements for each axis. Remember that `10` is the center step, as we've mapped the range `0-20`. The first set for each axis says if the joystick moves 1 step off center (left or right for the x axis and up or down for the y axis), to move the mouse the appropriate direction by 1 unit. The second set for each axis says if the joystick is moved to the lowest or highest step for each axis, to move the mouse the appropriate direction by 8 units. That way you have the option to move the mouse slowly or quickly!

To see what `step` the joystick is at when you're moving it, uncomment the `print` statements by removing the `# ` from the lines that look like `# print(steps(x))`, and connecting to the serial console to see the output. Consider only uncommenting one set at a time, or you end up with a huge amount of information scrolling very quickly, which can be difficult to read!

For more detail check out the documentation at https://circuitpython.readthedocs.io/projects/hid/en/latest/

CircuitPython Storage

CircuitPython-compatible microcontrollers show up as a **CIRCUITPY** drive when plugged into your computer, allowing you to edit code directly on the board. Perhaps you've wondered whether or not you can write data from CircuitPython directly to the board to act as a data logger. The answer is **yes**!

The `storage` module in CircuitPython enables you to write code that allows CircuitPython to write data to the **CIRCUITPY** drive. This process requires you to include a **boot.py** file on your **CIRCUITPY** drive, along side your **code.py** file.

The **boot.py** file is special - the code within it is executed when CircuitPython starts up, either from a hard reset or powering up the board. It is not run on soft reset, for example, if you reload the board from the serial console or the REPL. This is in contrast to the code within **code.py**, which is executed after CircuitPython is already running.

The **CIRCUITPY** drive is typically writable by your computer; this is what allows you to edit your code directly on the board. The reason you need a **boot.py** file is that you have to set the filesystem to be read-only by your computer to allow it to be writable by CircuitPython. This is because CircuitPython cannot write to the filesystem at the same time as your computer. Doing so can lead to filesystem corruption and loss of all content on the drive, so CircuitPython is designed to only allow one at at time.

You can only have either your computer edit the CIRCUITPY drive files, or CircuitPython. You cannot have both write to the drive at the same time. (Bad Things Will Happen so we do not allow you to do it!)

boot.py

```
# SPDX-FileCopyrightText: 2017 Limor Fried for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Storage logging boot.py file"""
import board
import digitalio
import storage

# For Gemma M0, Trinket M0, Metro M0/M4 Express, ItsyBitsy M0/M4 Express
switch = digitalio.DigitalInOut(board.D2)

# For Feather M0/M4 Express
# switch = digitalio.DigitalInOut(board.D5)

# For Circuit Playground Express, Circuit Playground Bluefruit
# switch = digitalio.DigitalInOut(board.D7)

switch.direction = digitalio.Direction.INPUT
switch.pull = digitalio.Pull.UP

# If the switch pin is connected to ground CircuitPython can write to the drive
storage.remount("/", readonly=switch.value)
```

The `storage.remount()` command has a `readonly` keyword argument. **This argument refers to the read/write state of CircuitPython.** It does NOT refer to the read/write state of your computer.

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

For **Gemma M0, Trinket M0, Metro M0 Express, Metro M4 Express, ItsyBitsy M0 Express and ItsyBitsy M4 Express**, no changes to the initial code are needed.

For **Feather M0 Express and Feather M4 Express**, comment out `switch = digitalio.DigitalInOut(board.D2)`, and uncomment `switch = digitalio.DigitalInOut(board.D5)`.

For **Circuit Playground Express and Circuit Playground Bluefruit**, comment out `switch = digitalio.DigitalInOut(board.D2)`, and uncomment `switch = digitalio.DigitalInOut(board.D7)`. Remember, D7 is the onboard slide switch, so there's no extra wires or alligator clips needed.

On the Circuit Playground Express or Circuit Playground Bluefruit, the switch is in the right position (closer to the ear icon on the silkscreen) it returns `False`, and the **CIRCUITPY** drive will be writable by CircuitPython. If the switch is in the left position (closer to the music icon on the silkscreen), it returns `True`, and the **CIRCUITPY** drive will be writable by your computer.

Remember: To "comment out" a line, put a # and a space before it. To "uncomment" a line, remove the # + space from the beginning of the line.

Installing Project Code

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/CircuitPython_Logger/** 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: 2017 Limor Fried for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Storage logging example"""
import time
import board
import digitalio
import microcontroller

# For most CircuitPython boards:
led = digitalio.DigitalInOut(board.LED)
# For QT Py M0:
# led = digitalio.DigitalInOut(board.SCK)
```

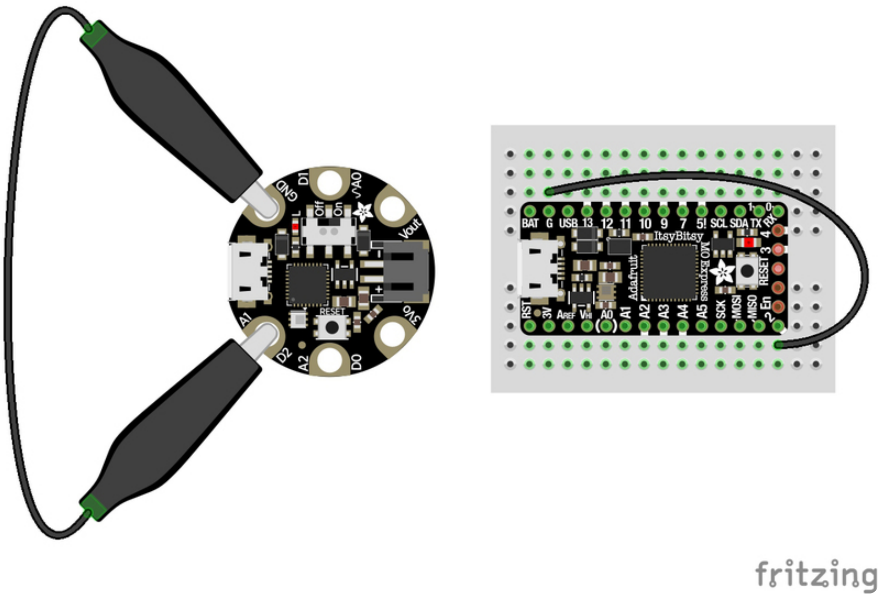
```
led.switch_to_output()

try:
    with open("/temperature.txt", "a") as fp:
        while True:
            temp = microcontroller.cpu.temperature
            # do the C-to-F conversion here if you would like
            fp.write('{0:f}\n'.format(temp))
            fp.flush()
            led.value = not led.value
            time.sleep(1)
except OSError as e:  # Typically when the filesystem isn't writeable...
    delay = 0.5  # ...blink the LED every half second.
    if e.args[0] == 28:  # If the file system is full...
        delay = 0.25  # ...blink the LED faster!
    while True:
        led.value = not led.value
        time.sleep(delay)
```

The filesystem will NOT automatically be set to read-only after you copy these files over! You'll still be able to edit files on CIRCUITPY after saving this boot.py.

Logging the Temperature

The way **boot.py** works is by checking to see if the pin you specified in the switch setup in your code is connected to a ground pin. If it is, it changes the read-write state of the file system, so the CircuitPython core can begin logging the temperature to the board.

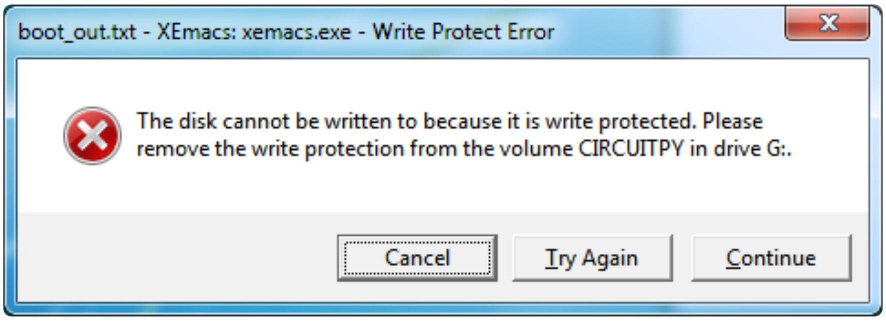
For help finding the correct pins, see the wiring diagrams and information in the Find the Pins section of the CircuitPython Digital In & Out guide (https://adafru.it/Bes). Instead of wiring up a switch, however, you'll be connecting the pin directly to ground with alligator clips or jumper wires.



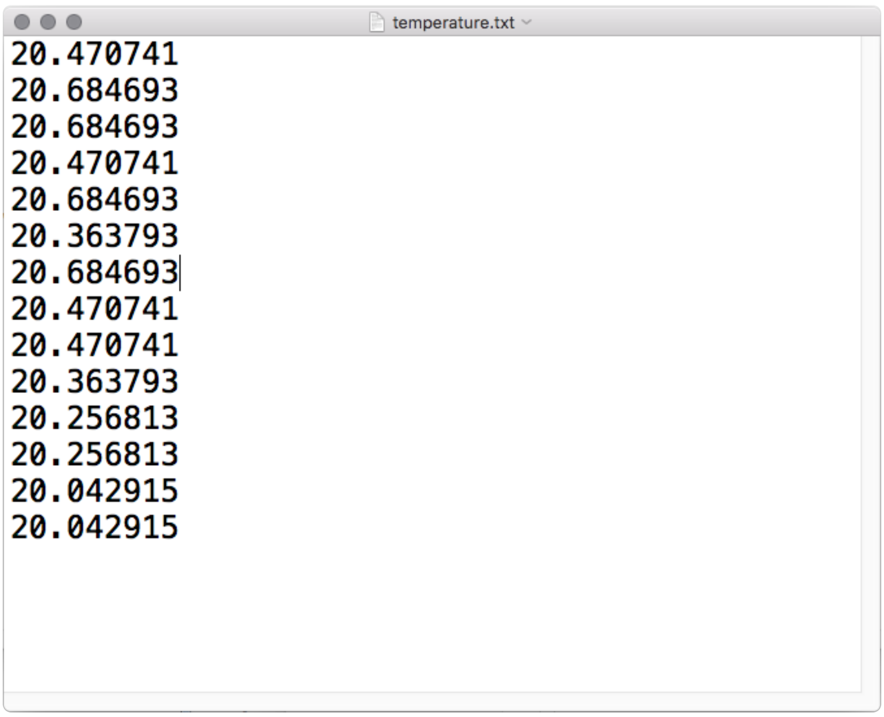
boot.py only runs on first boot of the device, not if you re-load the serial console with ctrl+D or if you save a file. You must EJECT the USB drive, then physically press the reset button!

Once you copied the files to your board, eject it and unplug it from your computer. If you're using your Circuit Playground Express, all you have to do is make sure the switch is to the right. Otherwise, use alligator clips or jumper wires to connect the chosen pin to ground. Then, plug your board back into your computer.

You will not be able to edit code on your **CIRCUITPY** drive anymore!



The red LED should blink once a second and you will see a new **temperature.txt** file on **CIRCUITPY**.



This file gets updated once per second, but you won't see data come in live. Instead, when you're ready to grab the data, eject and unplug your board. For CPX, move the switch to the left, otherwise remove the wire connecting the pin to ground. Now it will be possible for you to write to the filesystem from your computer again, but it will not be logging data.

We have a more detailed guide on this project available here: CPU Temperature Logging with CircuitPython (https://adafru.it/zuF). If you'd like more details, check it out!

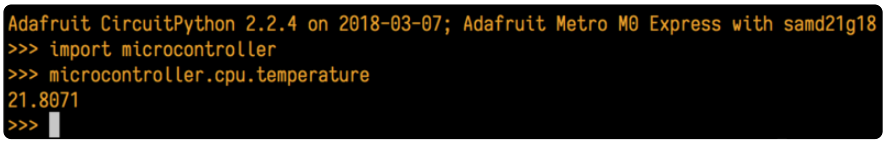
CircuitPython CPU Temp

There is a CPU temperature sensor built into every ATSAMD21, ATSAMD51 and nRF52840 chips. CircuitPython makes it really simple to read the data from this sensor. This works on the Adafruit CircuitPython boards it's built into the microcontroller used for these boards.

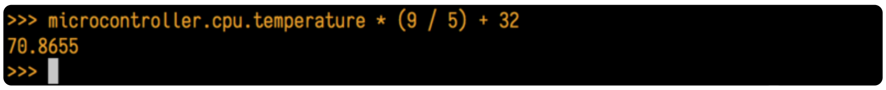
The data is read using two simple commands. We're going to enter them in the REPL. Plug in your board, connect to the serial console (https://adafru.it/Bec), and enter the REPL (https://adafru.it/Awz). Then, enter the following commands into the REPL:

```
import microcontroller
microcontroller.cpu.temperature
```

That's it! You've printed the temperature in Celsius to the REPL. Note that it's not exactly the ambient temperature and it's not super precise. But it's close!



If you'd like to print it out in Fahrenheit, use this simple formula: Celsius * (9/5) + 32. It's super easy to do math using CircuitPython. Check it out!



Note that the temperature sensor built into the nRF52840 has a resolution of 0.25 degrees Celsius, so any temperature you print out will be in 0.25 degree increments.

CircuitPython Expectations

As we continue to develop CircuitPython and create new releases, we 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 we continue to develop CircuitPython and create new releases, we will stop supporting older releases. **You need to update to the latest CircuitPython (https://adafru.it/Em8).**

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

As we release new versions of CircuitPython, we 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 3.x or 2.x, where can I find compatible libraries?

We are no longer building or supporting the CircuitPython 2.x and 3.x library bundles. We highly encourage you to update CircuitPython to the latest version (https://adafru.it/Em8) and use the current version of the libraries (https://adafru.it/ENC). However, if for some reason you cannot update, you can find the last available 2.x build here (https://adafru.it/FJA) and the last available 3.x build here (https://adafru.it/FJB).

Switching Between CircuitPython and Arduino

Many of the CircuitPython boards also run Arduino. But how do you switch between the two? Switching between CircuitPython and Arduino is easy.

If you're currently running Arduino and would like to start using CircuitPython, follow the steps found in Welcome to CircuitPython: Installing CircuitPython (https://adafru.it/Amd).

If you're currently running CircuitPython and would like to start using Arduino, plug in your board, and then load your Arduino sketch. If there are any issues, you can double tap the reset button to get into the bootloader and then try loading your sketch. Always backup any files you're using with CircuitPython that you want to save as they could be deleted.

That's it! It's super simple to switch between the two.

The Difference Between Express And Non-Express Boards

We often reference "Express" and "Non-Express" boards when discussing CircuitPython. What does this mean?

Express refers to the inclusion of an extra 2MB flash chip on the board that provides you with extra space for CircuitPython and your code. This means that we're able to include more functionality in CircuitPython and you're able to do more with your code on an Express board than you would on a non-Express board.

Express boards include Circuit Playground Express, ItsyBitsy M0 Express, Feather M0 Express, Metro M0 Express and Metro M4 Express.

Non-Express boards include Trinket M0, Gemma M0, QT Py, Feather M0 Basic, and other non-Express Feather M0 variants.

Non-Express Boards: Gemma, Trinket, and QT Py

CircuitPython runs nicely on the Gemma M0, Trinket M0, or QT Py M0 but there are some constraints

Small Disk Space

Since we use the internal flash for disk, and that's shared with runtime code, its limited! Only about 50KB of space.

No Audio or NVM

Part of giving up that FLASH for disk means we couldn't fit everything in. There is, at this time, no support for hardware audio playpack or NVM 'eeprom'. Modules `audioio` and `bitbangio` are not included. For that support, check out the Circuit Playground Express or other Express boards.

However, I2C, UART, capacitive touch, NeoPixel, DotStar, PWM, analog in and out, digital IO, logging storage, and HID do work! Check the CircuitPython Essentials for examples of all of these.

Differences Between CircuitPython and MicroPython

For the differences between CircuitPython and MicroPython, check out the [CircuitPython documentation](https://adafru.it/Bvz) (https://adafru.it/Bvz).

Differences Between CircuitPython and Python

Python (also known as CPython) is the language that MicroPython and CircuitPython are based on. There are many similarities, but there are also many differences. This is a list of a few of the differences.

Python Libraries

Python is advertised as having "batteries included", meaning that many standard libraries are included. Unfortunately, for space reasons, many Python libraries are not available. So for instance while we wish you could `import numpy`, `numpy` isn't available (look for the `ulab` library for similar functions to `numpy` which works on many microcontroller boards). So you may have to port some code over yourself!

Integers in CircuitPython

On the non-Express boards, integers can only be up to 31 bits long. Integers of unlimited size are not supported. The largest positive integer that can be represented is 2^{30}-1, 1073741823, and the most negative integer possible is -2^{30}, -1073741824.

As of CircuitPython 3.0, Express boards have arbitrarily long integers as in Python.

Floating Point Numbers and Digits of Precision for Floats in CircuitPython

Floating point numbers are single precision in CircuitPython (not double precision as in Python). The largest floating point magnitude that can be represented is about +/-3.4e38. The smallest magnitude that can be represented with full accuracy is about +/-1.7e-38, though numbers as small as +/-5.6e-45 can be represented with reduced accuracy.

CircuitPython's floats have 8 bits of exponent and 22 bits of mantissa (not 24 like regular single precision floating point), which is about five or six decimal digits of precision.

Differences between MicroPython and Python

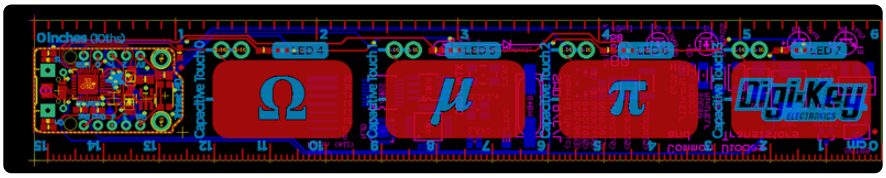
For a more detailed list of the differences between CircuitPython and Python, you can look at the MicroPython documentation. We keep up with MicroPython stable releases, so check out the core 'differences' they document here. (https://adafru.it/zwA)

Downloads

Files

- ATSAMD21 Datasheet (https://adafru.it/xZe)
- Webpage for the ATSAMD21E18 (main chip used) (https://adafru.it/xZf)
- EagleCAD files on GitHub (https://adafru.it/Fsu)
- Fritzing Object in Adafruit Fritzing Library (https://adafru.it/Fsv)
- Default files shipped with the board including default example code.py (https://adafru.it/Fyg)

Fab Print



Schematic