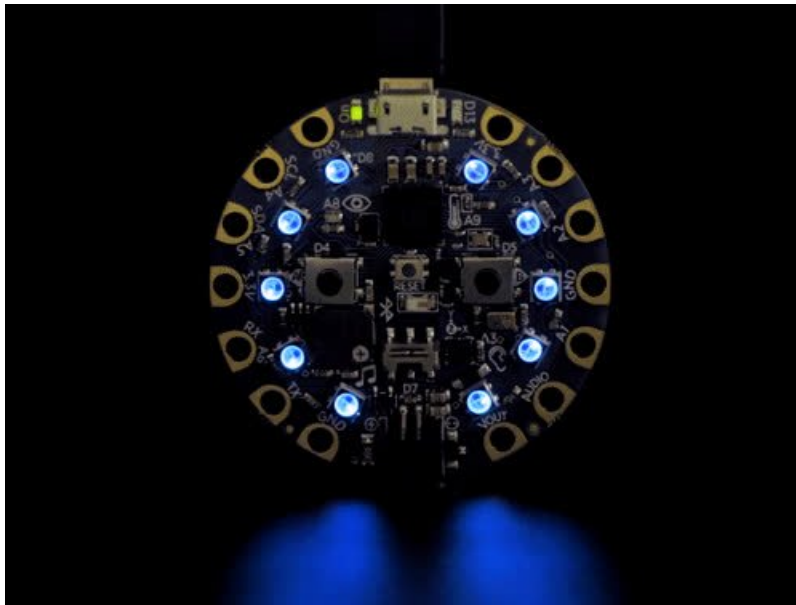




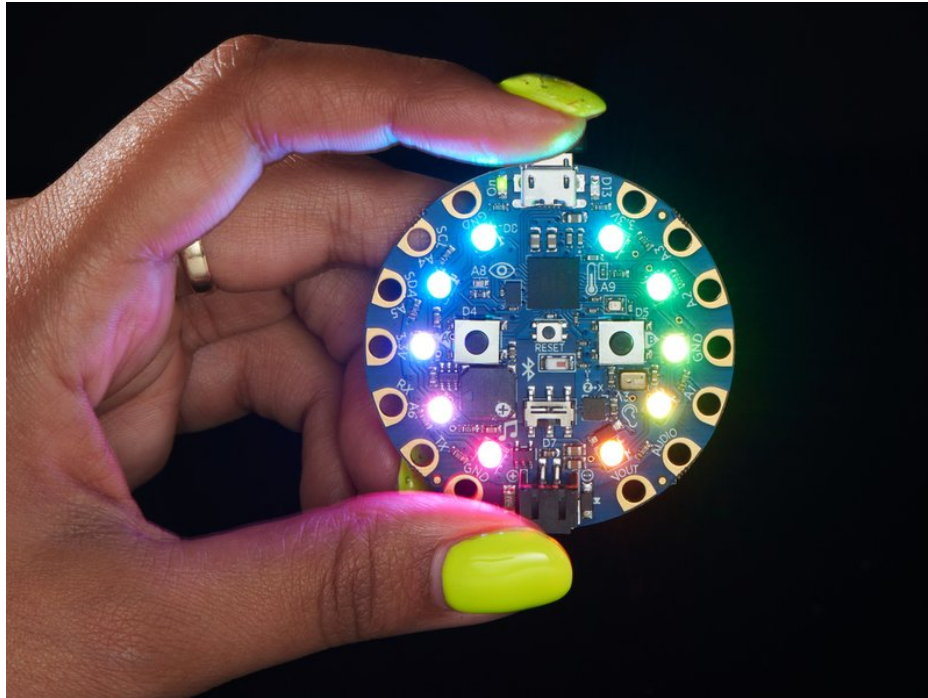
Adafruit Circuit Playground Bluefruit

Created by Kattni Rembor

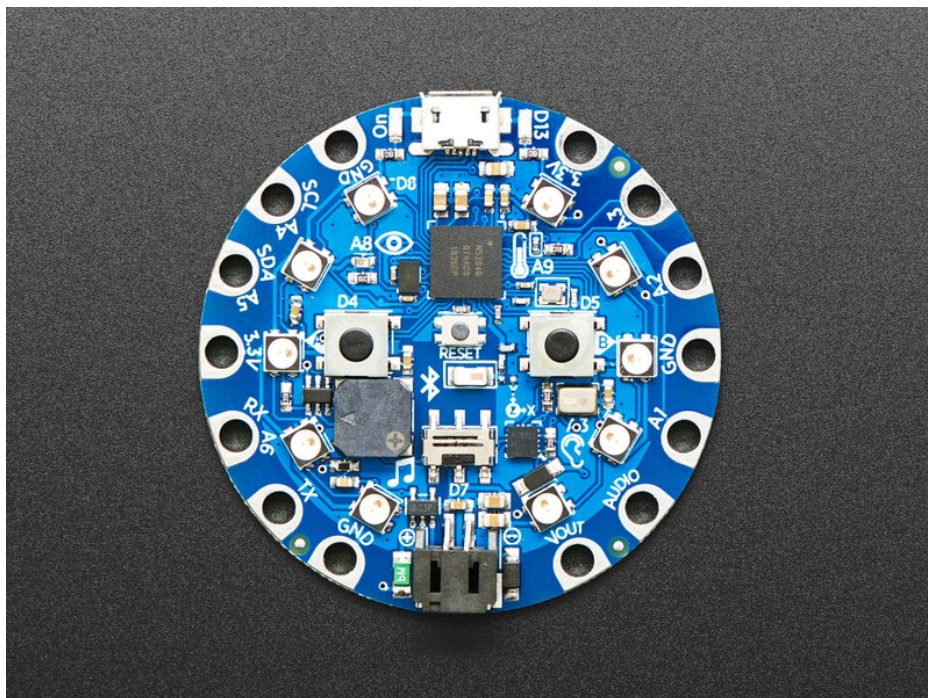


Last updated on 2020-05-15 03:55:15 PM EDT

Overview



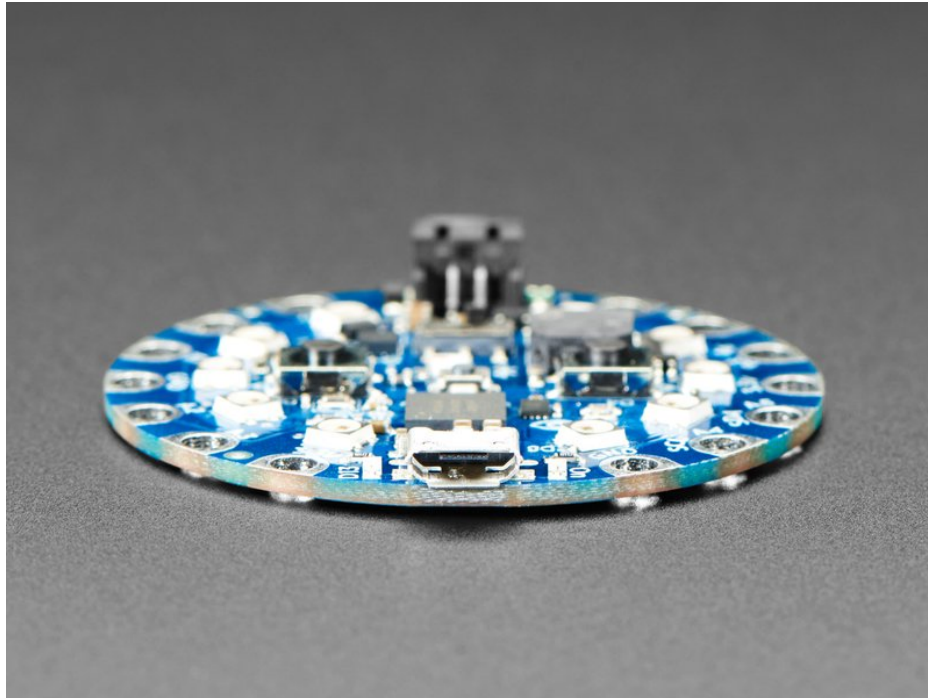
Circuit Playground Bluefruit is our third board in the Circuit Playground series, another step towards a perfect introduction to electronics and programming. We've taken the popular Circuit Playground Express and made it even better! Now the main chip is an nRF52840 microcontroller which is not only more powerful, but also comes with Bluetooth Low Energy support for wireless connectivity.



The board is round and has alligator-clip pads around it so you don't have to solder or sew to make it work. You can power it from USB, a AAA battery pack (<http://adafruit.it/727>), or with a Lipoly battery (for advanced users). Circuit

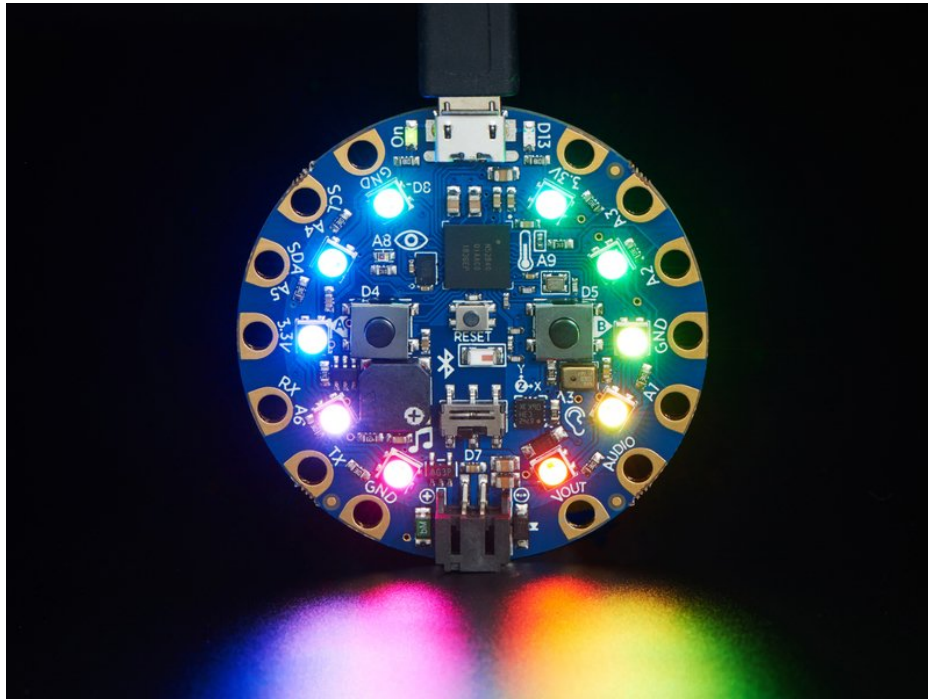
Playground Bluefruit has built-in USB support. Built in USB means you plug it in to program it and it just shows up, no special cable or adapter required. Just program your code into the board then take it on the go!

You can also use [MakeCode](https://adafru.it/C9N) (<https://adafru.it/C9N>)'s block-based GUI coding environment on this board.

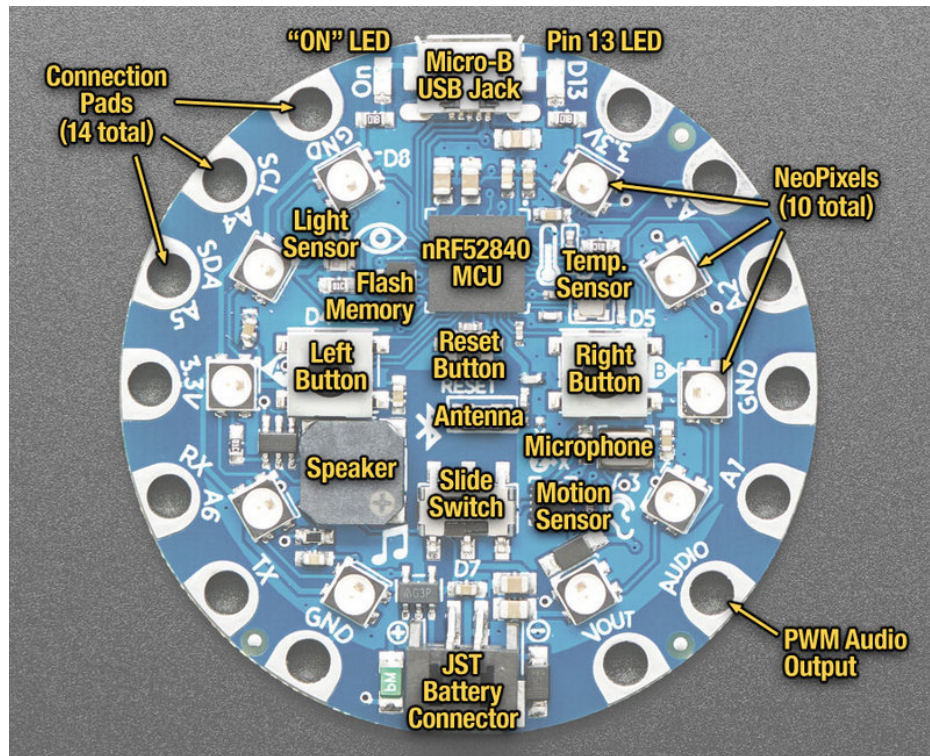


Here's some of the great goodies baked in to each Circuit Playground Bluefruit:

- 1 x nRF52840 Cortex M4 processor with Bluetooth Low Energy support
- 10 x mini NeoPixels, each one can display any color
- 1 x Motion sensor (LIS3DH triple-axis accelerometer with tap detection, free-fall detection)
- 1 x Temperature sensor (thermistor)
- 1 x Light sensor (phototransistor). Can also act as a color sensor and pulse sensor.
- 1 x Sound sensor (MEMS microphone)
- 1 x Mini speaker with class D amplifier (7.5mm magnetic speaker/buzzer)
- 2 x Push buttons, labeled A and B
- 1 x Slide switch
- 8 x alligator-clip friendly input/output pins
- Includes I2C, UART, 6 pins that can do analog inputs, multiple PWM outputs
- Green "ON" LED so you know its powered
- Red "#13" LED for basic blinking
- Reset button
- 2 MB of SPI Flash storage, used primarily with CircuitPython to store code and libraries.
- MicroUSB port for programming and debugging
- USB port can act like serial port, keyboard, mouse, joystick or MIDI!

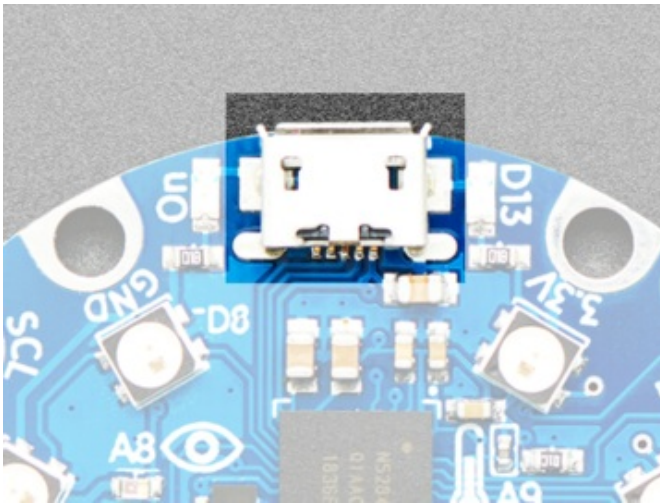


Guided Tour



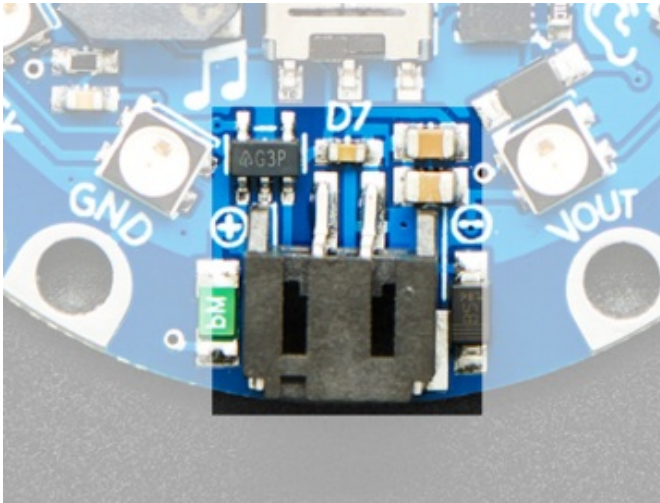
Let us take you on a tour of your Circuit Playground Bluefruit, which we'll shorten to **CPB**.

Power and Data



Micro B USB connector

This is at the top of the board. We went with the tried and true micro-B USB connector for power and/or USB communication (bootloader, serial, HID, etc). Use with any computer with a standard data/sync cable.

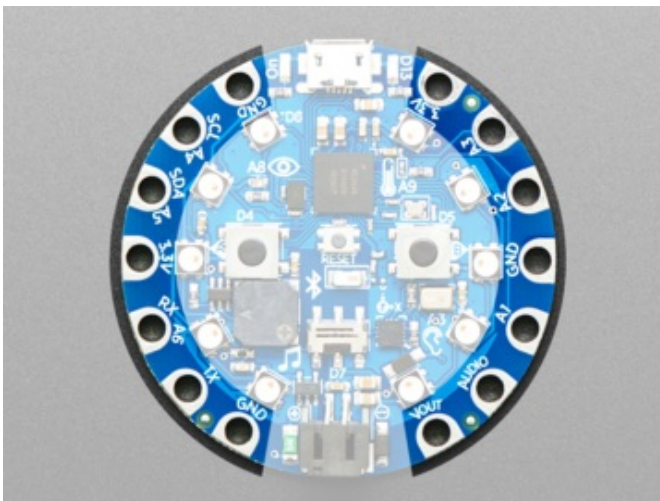


JST Battery Input

This is at the bottom of the board. You can take your CPB anywhere and power it from an external battery. This pin can take up 6V DC input, and has reverse-polarity, over-current and thermal protections. The circuitry inside will use either the battery input power or USB power, safely switching from one to the other. If both are connected, it will use whichever has the higher voltage. Works great with a Lithium Polymer battery or our 3xAAA battery packs with a JST connector on the end. There is no built in battery charging (so that you can use Alkaline *or* Lithium batteries safely)

Alligator/Croc Clip Pads

To make it super-easy to connect to the microcontroller, we have 14 connection pads. You can solder to them, use alligator/croc clips, sew with conductive thread, even use small metal screws!

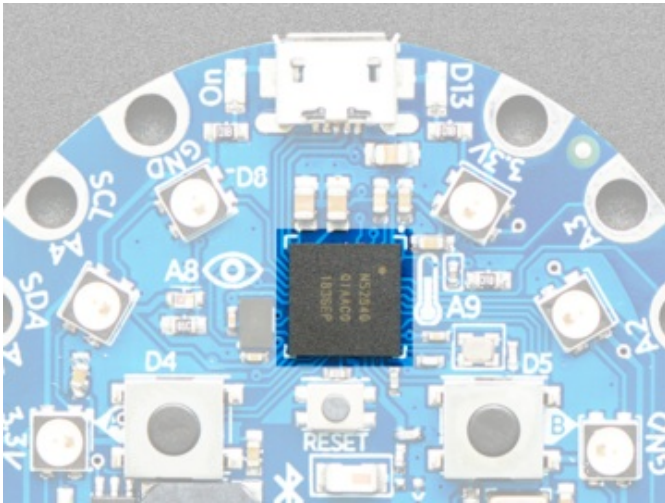


Of the 14 pads, you get a wide range of power pins, I2C, UART, Analog In, Digital In/Out, PWM, and Analog Out.

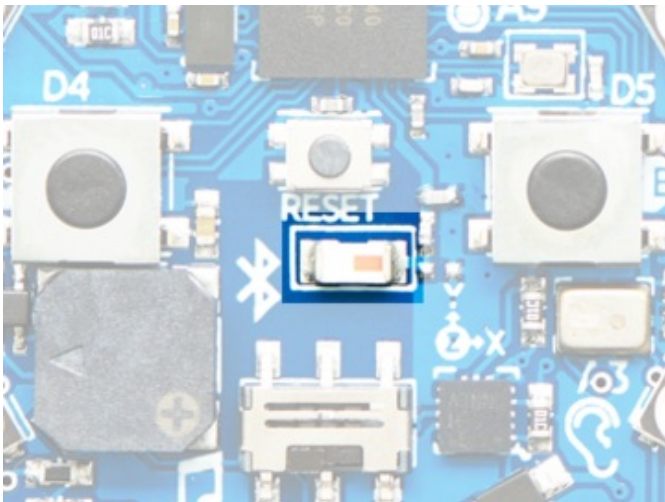
Some of them can even sense the touch of your finger!

See the next pinouts page for more details!

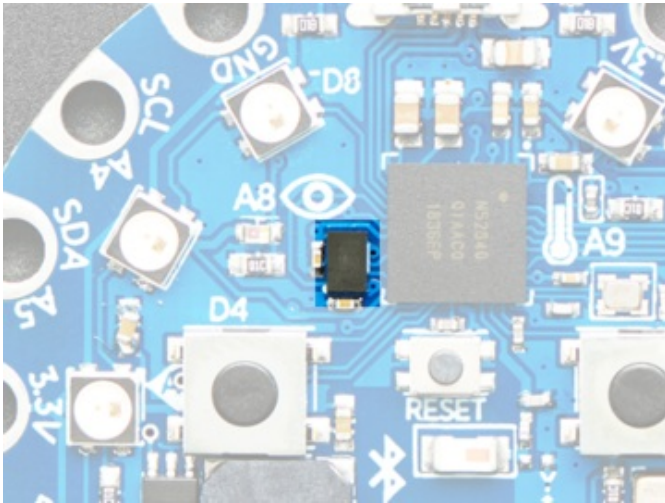
Microchips



The **brains** of the operation is the **nRF52840** Cortex M4 processor with Bluetooth Low Energy support. It sits at the top center and is what allows you to run CircuitPython or Arduino!

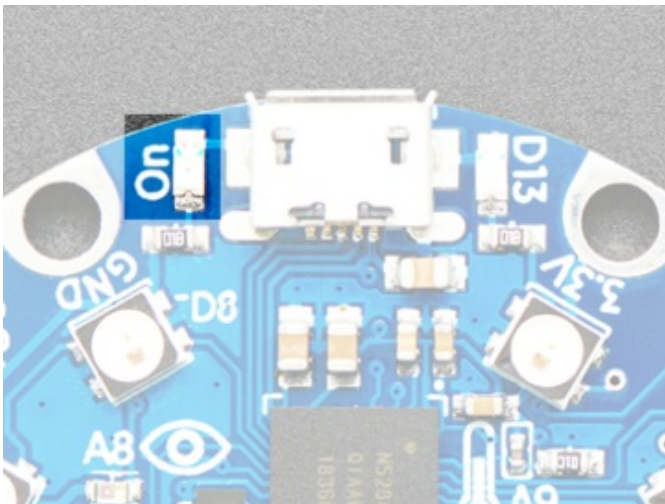


The **Bluetooth antenna** for the nRF52840 Bluetooth functionality is located in the center of the board. If you run into issues with Bluetooth range, make sure there's nothing near the antenna that might interfere, such as metallic surfaces!



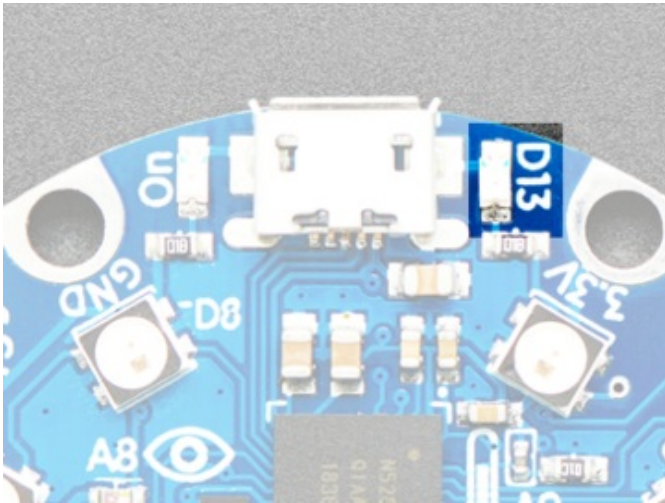
We have added a storage chip, called SPI Flash. This is a very, very small disk drive, only 2 MB large. You can use this in Arduino or CircuitPython to store files. In CircuitPython this is where all your code lives, and what you see when you use the **CIRCUITPY** drive on your computer.

LEDs



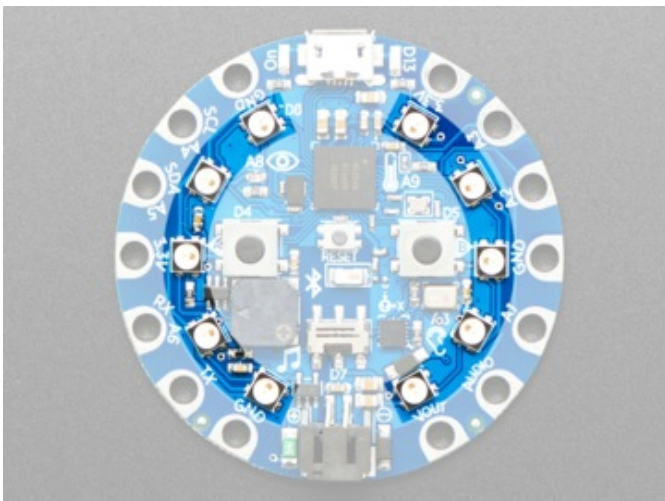
Green ON LED

To the left of the USB connector. This LED lets you know that the CPB is powered on. If it's lit, power is good! If it's dim, flickering or off, there's a power problem and you will have problems. You can't disable this light, but you *can* cover it with electrical tape if you want to make it black.



Red #13 LED

To the right of the USB connector. This LED does double duty. Its connected with a series resistor to the digital #13 GPIO pin. It pulses nicely when the CPB is in bootloader mode, and its also handy for when you want an indicator LED. Many first projects blink this LED to prove that programming worked.

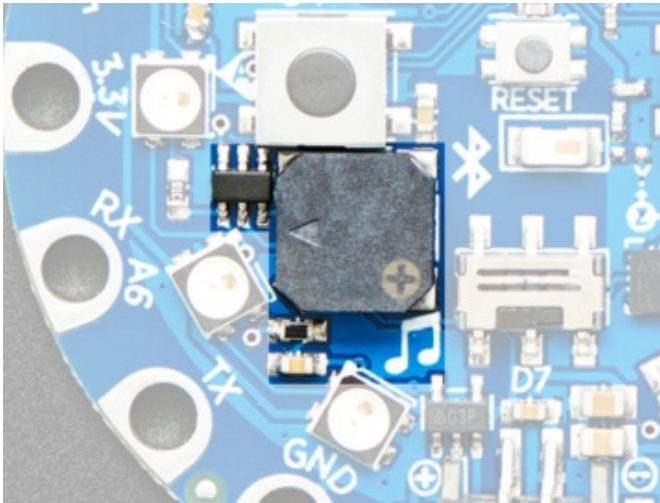


10 x Color NeoPixel LED

The ten LEDs surrounding the outer edge of the boards are all full color, RGB LEDs, each one can be set to any color in the rainbow. Great for beautiful lighting effects! The NeoPixels will also help you know when the bootloader is running (they will turn green) or if it failed to initialize USB when connected to a computer (they will turn red).

Speaker

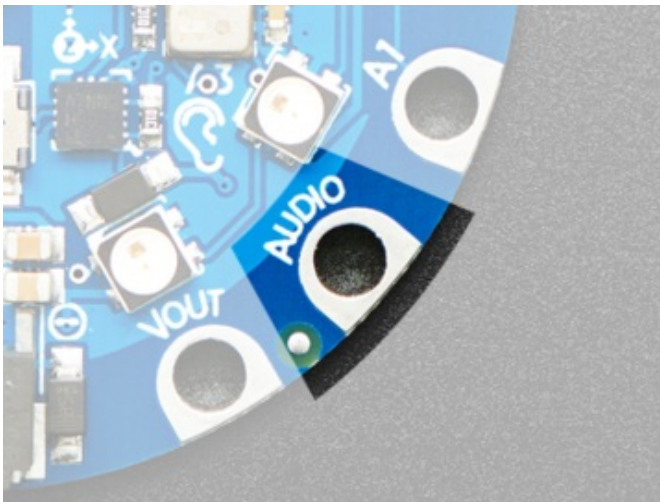
The CPB includes a speaker. It's not going to compete with your HiFi stereo, but it can play simple songs and tones.



The speaker is the squarish gray chunk on the bottom left of the board. There is a small class D amplifier connected to the speaker so it can get quite loud! Note: it won't sound good if too loud, so some experimentation may be necessary

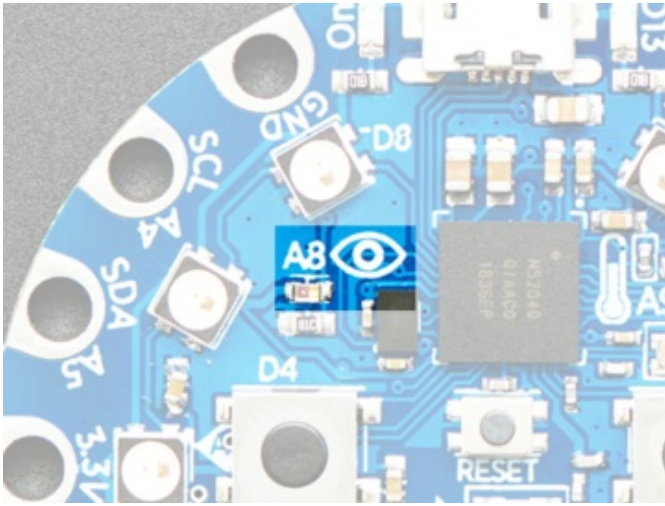
The amplifier is connected to the PWM output **AUDIO** pin -- this pin is also available on one of the connection pads in the lower right.

If you do not want the internal speaker to make noise, you can turn it off using the shutdown control on pin #11



Sensors

The Circuit Playground Bluefruit has a large number of sensor **inputs** that let you add all sorts of interactivity to your project.

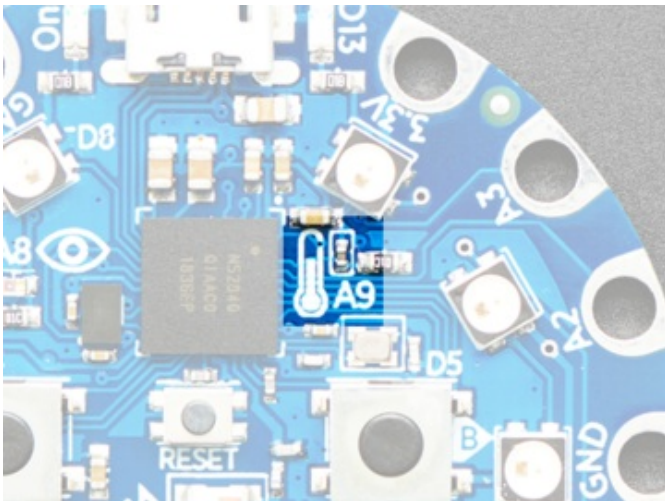


Light Sensor

There is an analog light sensor, [part number ALS-PT19](https://adafruit.it/tC2) (<https://adafruit.it/tC2>), in the top left part of the board. This can be used to detect ambient light, with similar spectral response to the human eye.

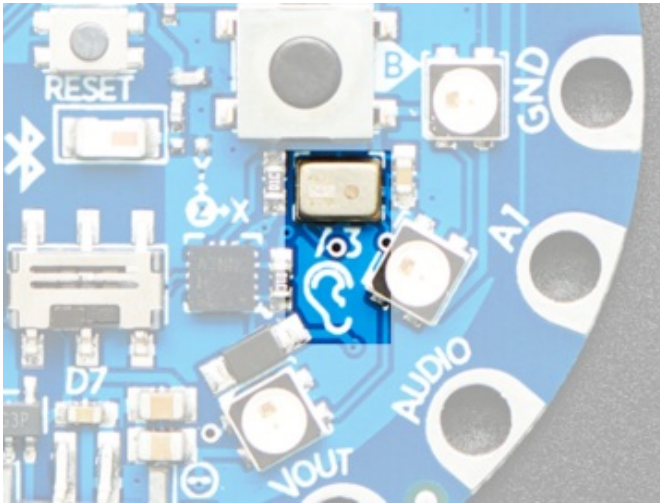
This sensor is connect to analog pin **A8** and will read between 0 and 1023 with higher values corresponding to higher light levels. A reading of about 300 is common for most indoor light levels.

With some clever code, you can use this as a color sensor or even a pulse sensor!



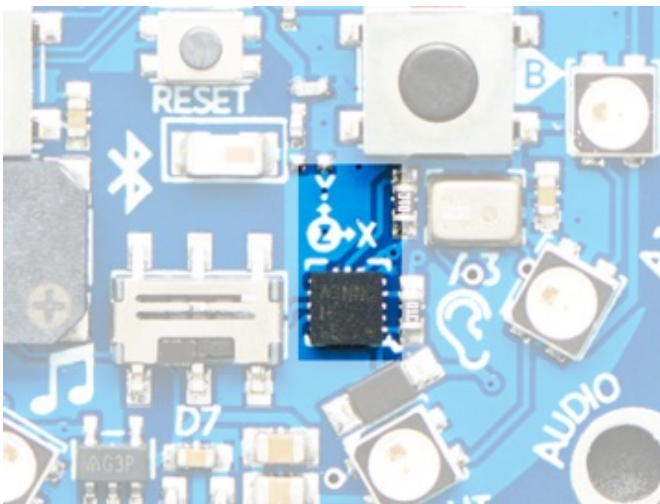
Temperature Sensor

There is an NTC thermistor (Murata NCP15XH103F03RC) that we use for temperature sensing. While it isn't an all-in-one temperature sensor, with linear output, it's easy to calculate the temperature based on the analog voltage on analog pin **A9**. There's a 10K resistor connected to it as a pull down.



Microphone Audio Sensor

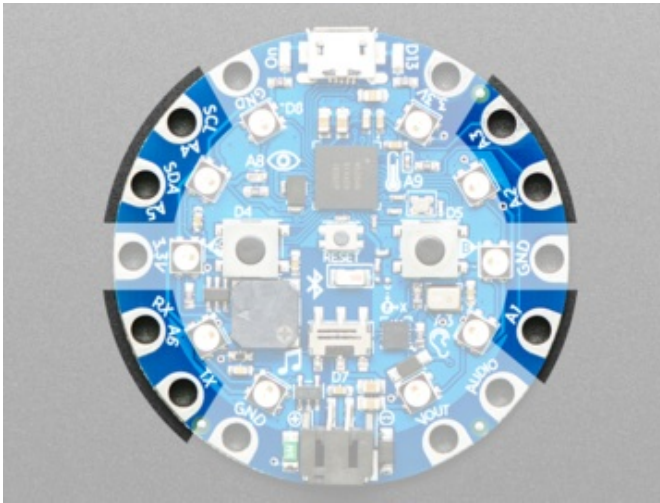
A MEMS microphone can be used to detect audio levels and even perform basic FFT functions. Instead of an analog microphone, that requires an external op-amp and level management, we've decided to go with a PDM microphone. This is a digital mic, and is a lot smaller and less expensive! You will have to use the CircuitPython/Arduino support libraries to read the audio volume, you cannot read it like an analog voltage



Motion Sensor

We can sense motion with an accelerometer. This sensor detects *acceleration* which means it can be used to detect when its being moved around, as well as gravitational pull in order to detect orientation.

The LIS3DH 3-axis XYZ accelerometer is in the dead center of the board and you can use it to detect tilt, gravity, motion, as well as 'tap' and 'double tap' strikes on the board. The LIS3DH is connected to an internal I2C pinset (not the same as the ones on the pads) and has an optional interrupt output on digital pin **D24**.

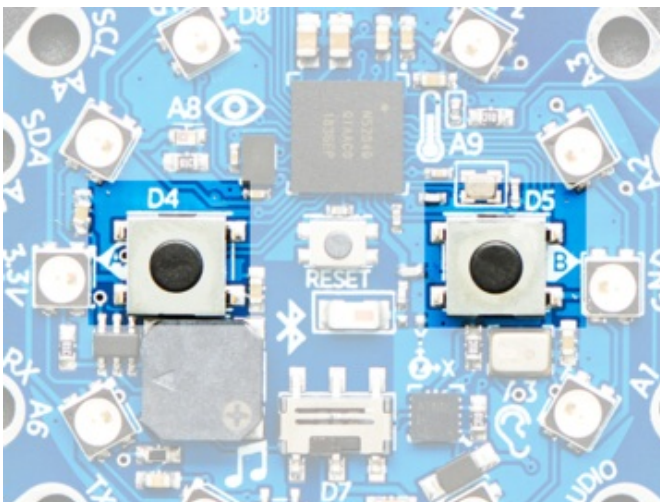


Capacitive Touch

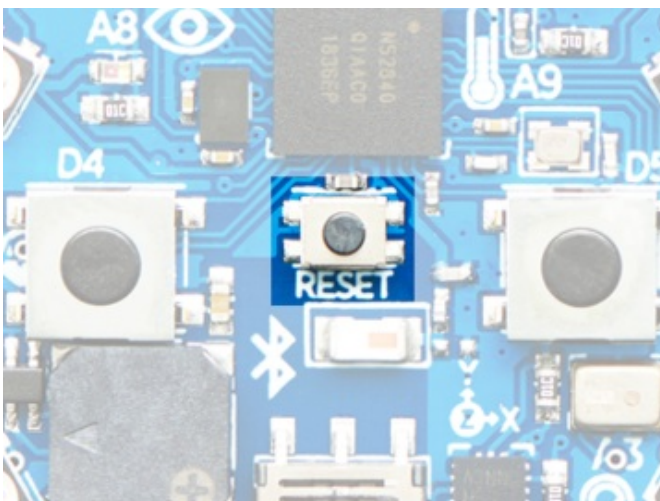
The CPB has capacitive touch capabilities. This is a great way to sense human touch without additional components. Even animals will work if it's directly touching their skin!

On the Bluefruit you get **seven** capacitive touch pads: **A1** - **A6** and **TX**. Capacitive touch is supported in both CircuitPython and Arduino!

Switches & Buttons

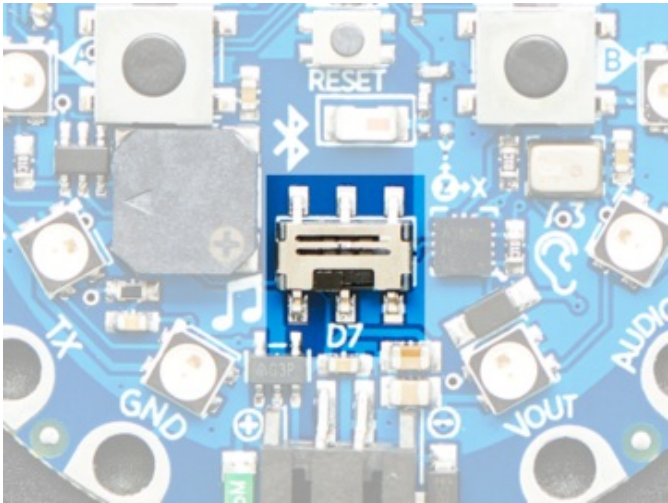


There are two large **A** and **B** buttons, connected to digital **D4** (Left) and **D5** (Right) each. These are unconnected when not pressed, and connected to 3.3V when pressed, so they read HIGH. Set the pins **D4** (**BUTTON_A** in CircuitPython) and **D5** (**BUTTON_B** in CircuitPython) to use an internal pull-down resistor when reading these pins so they will read LOW when not pressed.



This small button in the center of the board is for **Resetting** the board. You can use this button to restart or reset the CPB.

If using Arduino or CircuitPython, press this button once to reset, double-click to enter the bootloader manually.



There is a single slide switch near the center bottom of the Circuit Playground Bluefruit. It is connected to digital **D7**. The switch is unconnected when slid to the left and connected to ground when slid to the right. We set pin D7 to use an internal pull-up resistor so that the switch will read HIGH when slid to the left and LOW when slid to the right.

This is not an on-off switch, but you can use code to have this switch control how you want your project to behave

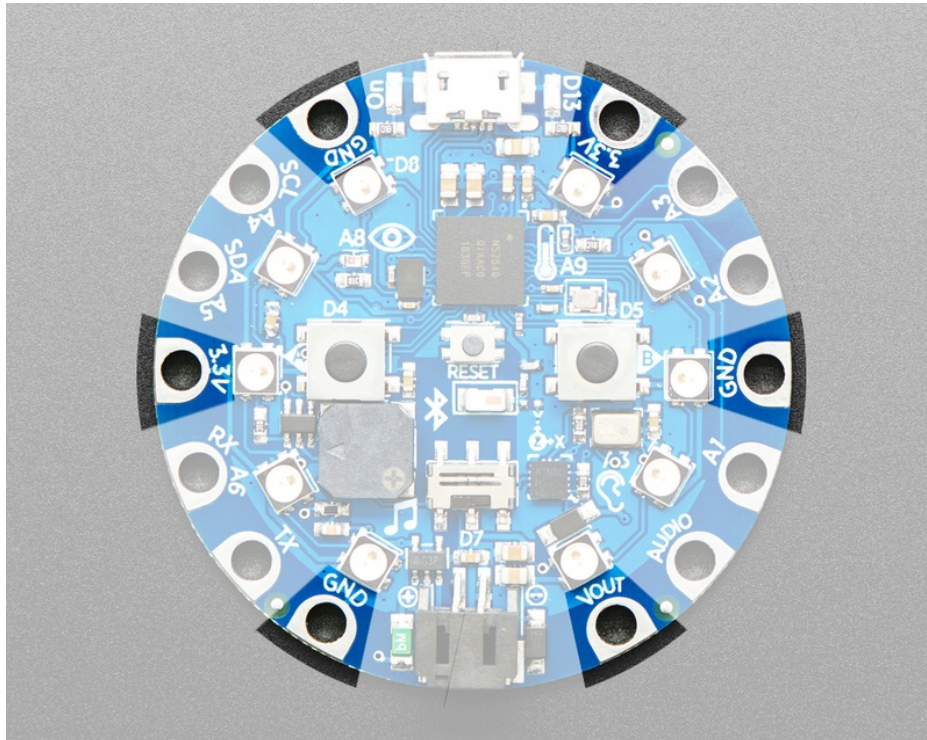
Note that you need to use an internal *pull-up* for the slide switch, but an internal *pull-down* for the push-buttons.

Pinouts

Despite having only 14 pads with 8 general purpose I/O pins available, there are a *lot* of possibilities with Circuit Playground Bluefruit. We went over all the internals in the last page. On this page we'll go through each pin/pad to explain what you can do with it.

Other than the Audio pad, no external I/O pads are shared with internal sensors/devices, so you do not need to worry about 'conflicting' pins or interactions!

Power Pads



There are 6 power pads available, equally spaced around the perimeter.

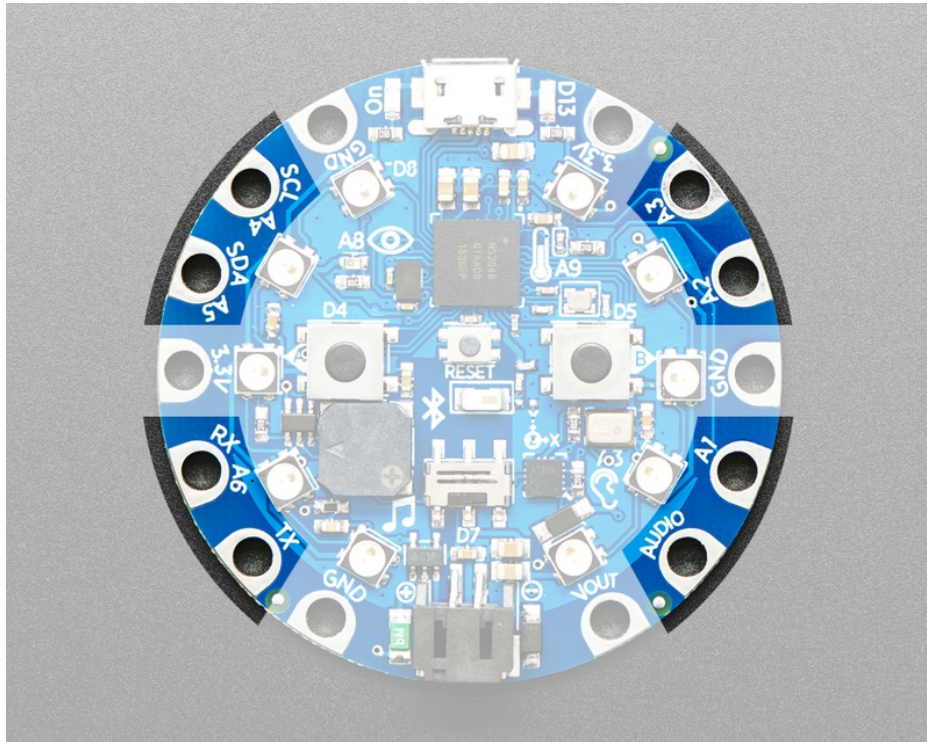
- **GND** - there are 3 x **Ground** pads. They are all connected together, and are all the signal/power ground connections
- **3.3V** - there are two **3.3 Volt output** pads. They are connected to the output of the onboard regulator. The regulator can provide about 500mA max, but that includes all the built in parts too! So you should roughly budget about 300mA available for your usage (450mA if you are not using the onboard NeoPixels)
- **Vout** - there is one **Voltage Output** pad. This is a special power pad, it will be connected to *either* the USB power or the battery input, whichever has the higher voltage. This output does not connect to the regulator so **you can draw as much current as your USB port / Battery can provide**. There is a resettable fuse on this pin, so you can draw about 500mA continuous, and 1 Amp peak before it will trip. If the fuse trips, just wait a minute and it will automatically reset

If you want to connect chips, sensors, and low power electronics that requires 3.3V clean power, use the **3.3V** pads.

If you want to connect servos, NeoPixels, DotStars or other high power electronics that are OK up to 5V, use the **Vout** pad.

Input/Output Pads

Next we will cover the 8 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 nRF52840. 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. In addition, A1-A6 can be used as analog inputs (12-bit ADC) (TX and Audio can not!). All but Audio can be used for capacitive touch. All pads can also be used as hardware interrupt inputs.

Each pad can provide up to ~20mA 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://adafruit.it/aUD>)

All of the GPIO pads 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.

Other than **Audio**, which is shared with the speaker, all of the pads are completely 'free' pins, they are not used by the USB connection, LEDs, sensors, etc so you never have to worry about interfering with them when programming.

Each Pin!

Let's start with **Audio** which is in the bottom right corner, and work our way counter-clockwise. Because the nRF52840 is flexible with PWM pins, you can make any of the pins PWM outputs

- **Audio** (a.k.a **D12**) - This is a designated pin that is OK with high speed PWM signal, so it's great for playing basic audio clips - it's also connected to the little speaker on board. It can be digital I/O, but if you do that it will interfere with the built-in speaker. This is the one pin that cannot be used for capacitive touch.
- **A1 / D6** - This pin can be digital I/O, or analog input and can be capacitive touch sensor

- **A2 / D9** - This pin can be digital I/O, or analog input and can be capacitive touch sensor
- **A3 / D10** - This pin can be digital I/O, or analog input and can be capacitive touch sensor
- **A4 / SCL / D3** - This pin can be digital I/O, or analog input. This pin is also the designated **I2C SCL** pin, and can be capacitive touch sensor
- **A5 / SDA / D2** - This pin can be digital I/O, or analog input. This pin is also the designated **I2C SDA** pin, and can be capacitive touch sensor
- **A6 / RX / D0** - This pin can be digital I/O, or analog Input. This pin has PWM output, **Serial Receive**, and can be capacitive touch sensor
- **TX / D1** - This pin can be digital I/O. This pin has PWM output, **Serial Transmit**, and can be capacitive touch sensor

Internally Used Pins!

These are the names of the pins that are used for built in sensors and such! CircuitPython has more user friendly names available as well for some pins with things like buttons and LEDs on them - these are included in parentheses where applicable. Both names will work in CircuitPython!

- **D4 (BUTTON_A)** - Left Button A
- **D5 (BUTTON_B)** - Right Button B
- **D7 (SLIDE_SWITCH)** - Slide Switch
- **D8 (NEOPIXEL)** - Built-in 10 NeoPixels
- **D12 / AUDIO (SPEAKER)** - Speaker analog output
- **D13** - Red LED
- **A8 (LIGHT)** - Light Sensor
- **A9 (TEMPERATURE)** - Temperature Sensor
- **D24** - PDM mic data
- **D25** - PDM mic clock
- **D26** - Internal I2C SCL for accelerometer
- **D27** - Accelerometer interrupt
- **D28** - Internal I2C SDA for accelerometer
- **D29 ~ D34** - QSPI FLASH chip pins

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.** There are 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.

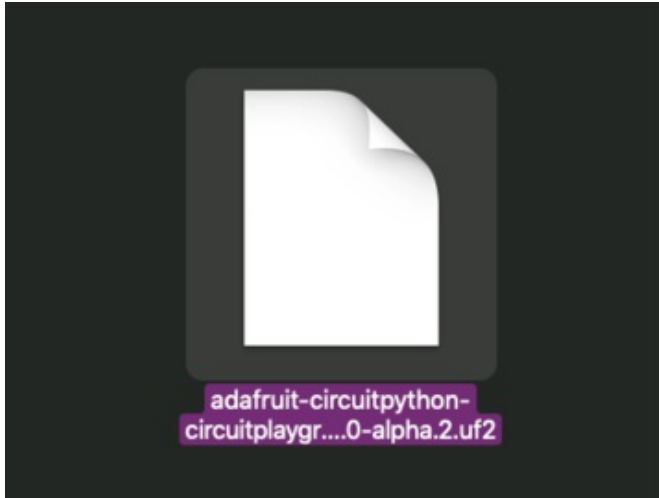
This is just the beginning. CircuitPython continues to evolve, and is constantly being updated. We welcome and encourage feedback from the community, and we incorporate this into how we are developing CircuitPython. That's the core of the open source concept. This makes CircuitPython better for you and everyone who uses it!

Install or Update CircuitPython

Follow this quick step-by-step to install or update CircuitPython on your Circuit Playground Bluefruit.

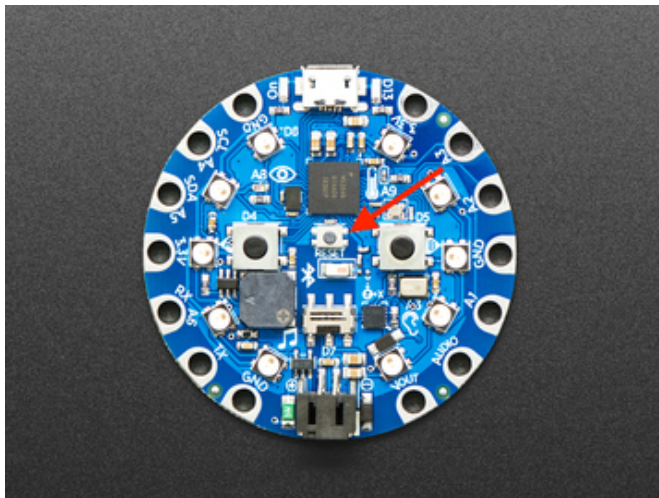
<https://adafru.it/FNK>

<https://adafru.it/FNK>



Click the link above and download the latest UF2 file

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

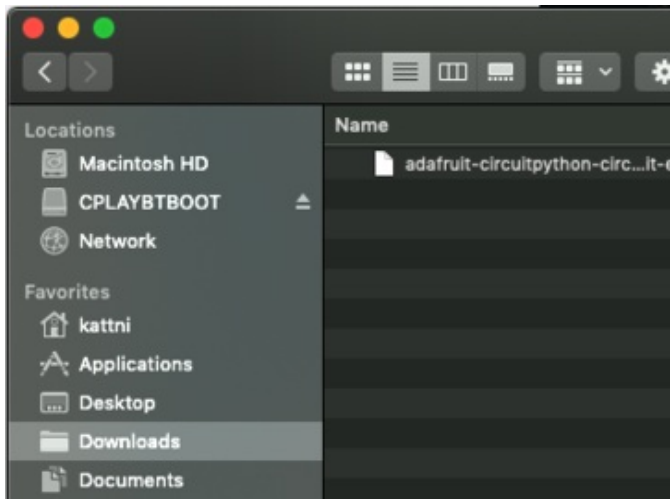


Plug your Circuit Playground Bluefruit into your computer using a known-good data-capable USB cable.

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.

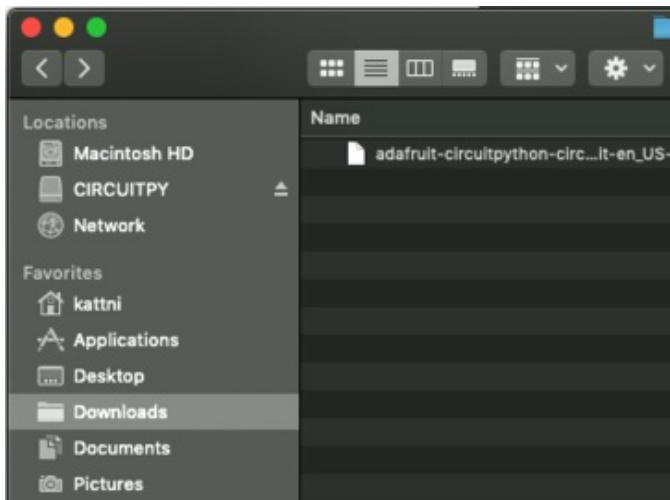
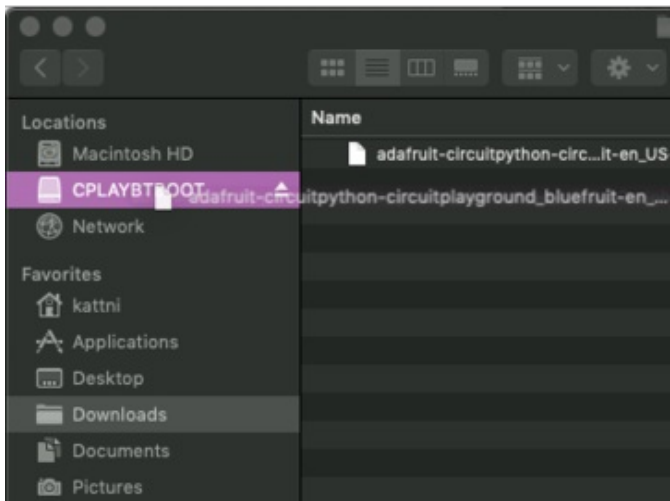
Double-click the small **Reset** button in the middle of the CPB (indicated by the red arrow in the image). The ten NeoPixel LEDs will all turn red, and then will all turn green. If they turn all red and stay red, check the USB cable, try another USB port, etc. The little red LED next to the USB connector will pulse 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! (If double-clicking doesn't do it, try a single-click!)



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

Drag the `adafruit_circuitpython_etc.uf2` file to **CPLAYBTBOOT**.



The LEDs will turn red. Then, the **CPLAYBTBOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

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

Circuit Playground Bluefruit CircuitPython Libraries

The Circuit Playground Bluefruit is packed full of features like Bluetooth and NeoPixel LEDs. Now that you have CircuitPython installed on your Circuit Playground Bluefruit, you'll need to install a base set of CircuitPython libraries to use the features of the board with CircuitPython.

Follow these steps to get the necessary libraries installed.

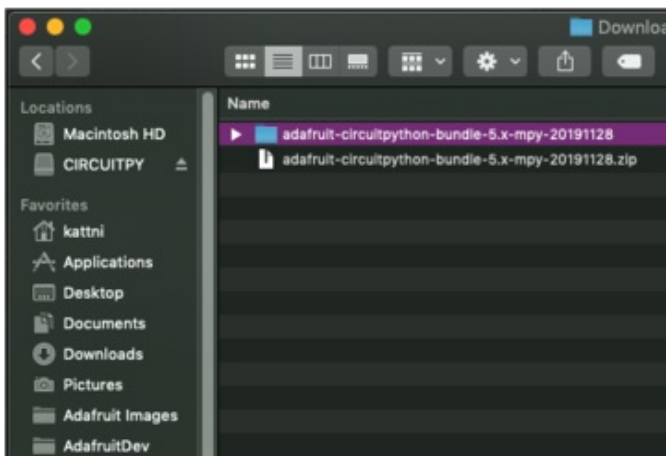
Installing CircuitPython Libraries on Circuit Playground Bluefruit

If you do not already have a **lib** folder on your **CIRCUITPY** drive, create one now.

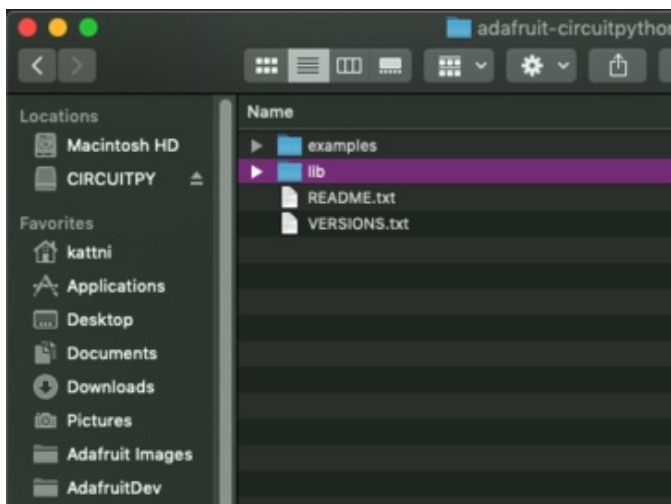
Then, download the CircuitPython library bundle that matches your version of CircuitPython from CircuitPython.org.

<https://adafru.it/ENC>

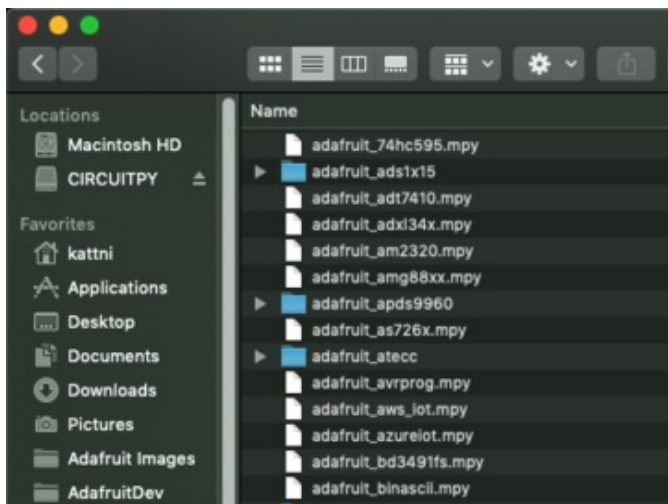
<https://adafru.it/ENC>



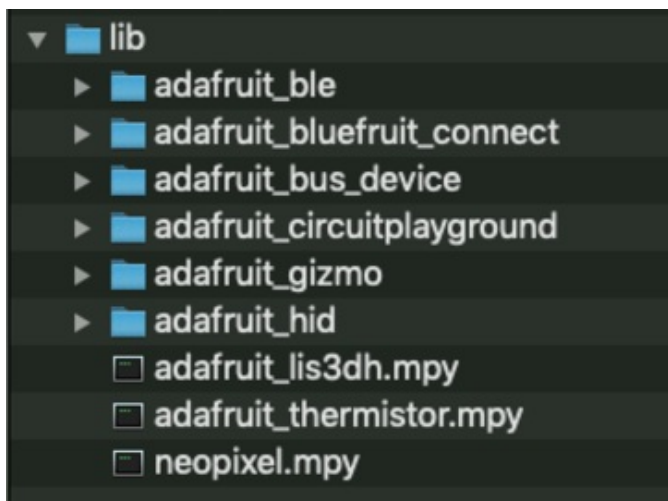
The bundle download as a .zip file. Extract the file. Open the resulting folder.



Open the **lib** folder found within.



Once inside, you'll find a lengthy list of folders and .mpy files. To install a CircuitPython library, you drag the file or folder from the **bundle lib** folder to the **lib** folder on your **CIRCUITPY** drive.



Copy the following folders and files from the **bundle lib** folder to the **lib** folder on your **CIRCUITPY** drive:

- adafruit_ble
- adafruit_bluefruit_connect
- adafruit_bus_device
- adafruit_circuitplayground
- adafruit_gizmo
- adafruit_hid
- adafruit_lis3dh.mpy
- adafruit_thermistor.mpy
- neopixel.mpy

Your lib folder should look like the image on the left.

Now you're all set to use CircuitPython with the features of the Circuit Playground Bluefruit!

Installing 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!)

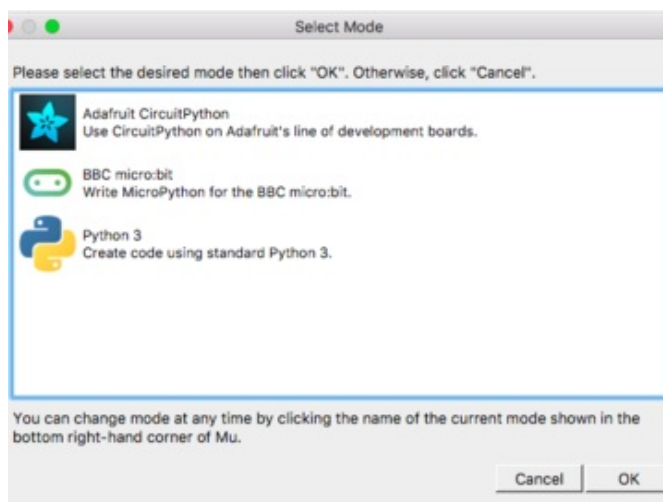
Download and Install Mu



Download Mu

from <https://codewith.mu> (<https://adafru.it/Be6>). Click the **Download** or **Start Here** links there for downloads and installation instructions. The website has a wealth of other information, including extensive tutorials and and how-to's.

Using Mu



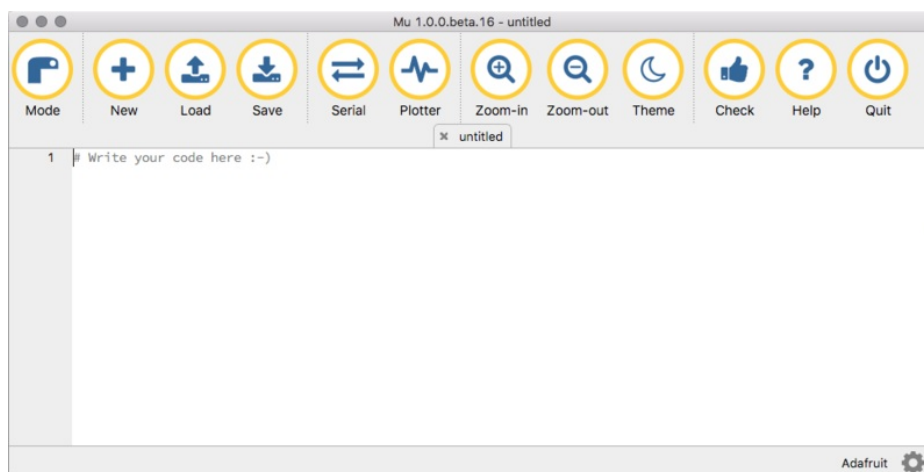
The first time you start Mu, you will be prompted to select your 'mode' - you can always change your mind later. For now please select **Adafruit!**

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 on that and then choose "Adafruit" in the dialog box that appears.



Mu attempts to auto-detect your board, so please plug in your CircuitPython device and make sure it shows up as a **CIRCUITPY** drive before starting Mu

Now you're ready to code! Lets keep going....



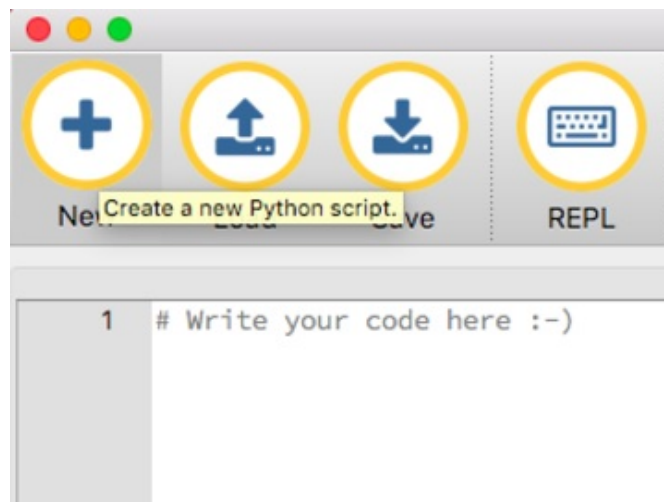
Creating and Editing Code

One of the best things about CircuitPython is how simple it is to get code up and running. In this section, we're going to cover 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. **We strongly recommend 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 basic text editors built into every operating system such as Notepad on Windows, TextEdit on Mac, and gedit on Linux. However, many of these editors don't write back changes immediately to files that you edit. That can cause problems when using CircuitPython. See the [Editing Code \(https://adafru.it/id3\)](https://adafru.it/id3) section below. If you want to skip that section for now, 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 is not a problem on MacOS.)

Creating Code



Open your editor, and create a new file. If you are using Mu, click the **New** button in the top left

Copy and paste the following code into your editor:

```
import board
import digitalio
import time

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

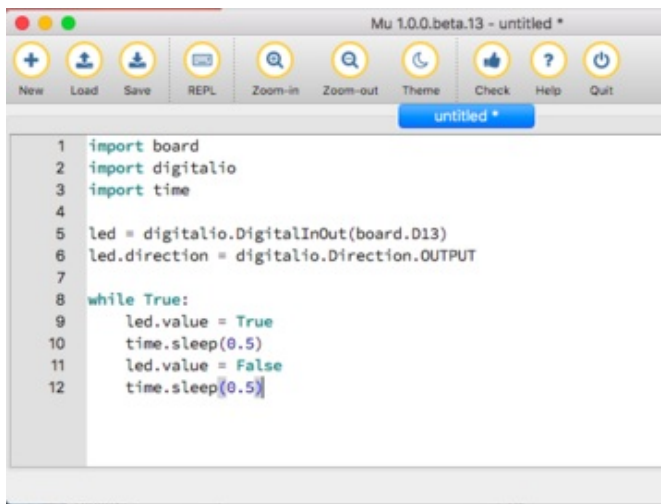
while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```



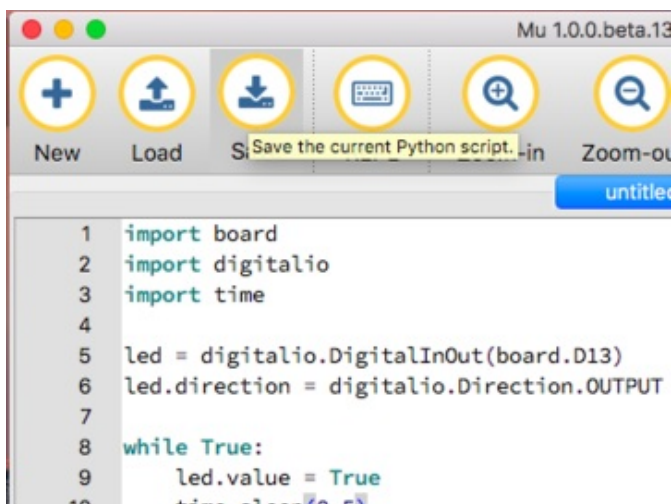
If you are using Adafruit CLUE, you will need to edit the code to use board.D17 as shown below!

For Adafruit CLUE, you'll need to use `board.D17` instead of `board.D13`. The rest of the code remains the same. Make the following change to the `led =` line:

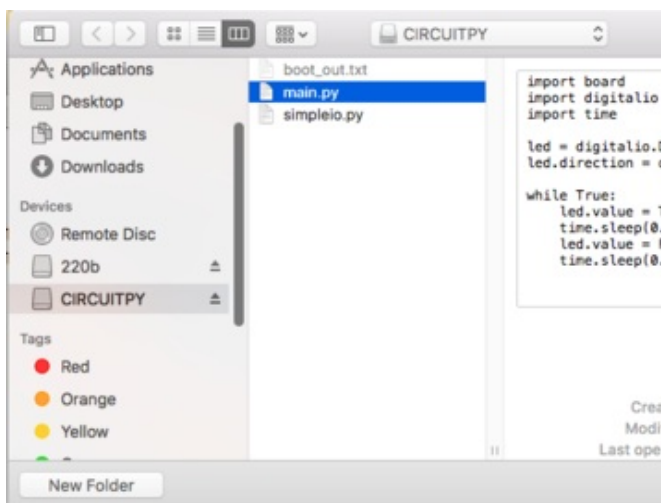
```
led = digitalio.DigitalInOut(board.D17)
```



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



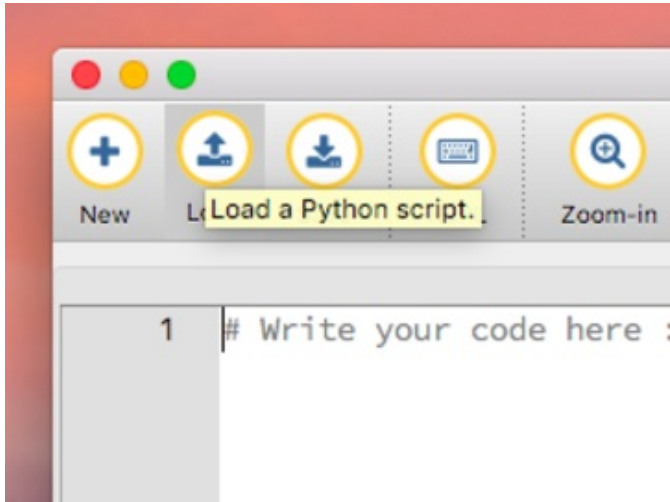
Save this file as `code.py` on your CIRCUITPY drive.



On each board you'll find a tiny red LED. It should now be blinking. Once per second

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

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 just one warning we have to give you before we continue...



Don't Click Reset or Unplug!

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

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

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

There are a few ways to avoid this:

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

Recommended editors:

- **mu** (<https://adafru.it/Be6>) is an editor that safely writes all changes (it's also our recommended editor!)
- **emacs** (<https://adafru.it/xNA>) is also an editor that will **fully write files on save** (<https://adafru.it/Be7>)
- **Sublime Text** (<https://adafru.it/xNB>) safely writes all changes
- **Visual Studio Code** (<https://adafru.it/Be9>) appears to safely write all changes
- **gedit** on Linux appears to safely write all changes
- **IDLE** (<https://adafru.it/IWB>), in Python 3.8.1 or later, **was fixed** (<https://adafru.it/IWD>) to write all changes

immediately

Recommended *only* with particular settings or with add-ons:

- **vim** (<https://adafru.it/ek9>) / **vi** safely writes all changes. But set up **vim** to not write **swapfiles** (<https://adafru.it/ELO>) (.swp files: temporary records of your edits) to CIRCUITPY. Run vim with **vim -n**, set the **no swapfile** option, or set the **directory** option to write swapfiles elsewhere. Otherwise the swapfile writes trigger restarts of your program.
- The **PyCharm IDE** (<https://adafru.it/xNC>) is safe if "Safe Write" is turned on in Settings->System Settings->Synchronization (true by default).
- If you are using **Atom** (<https://adafru.it/fMG>), install the **fsync-on-save package** (<https://adafru.it/E9m>) so that it will always write out all changes to files on **CIRCUITPY**.
- **SlickEdit** (<https://adafru.it/DdP>) works only if you **add a macro to flush the disk** (<https://adafru.it/ven>).

We *don't* recommend these editors:

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

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.

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

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

Back to Editing Code...

Now! Let's try editing the program you added to your board. Open your **code.py** file into your editor. We'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.D13)
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? Let's find out!

Exploring Your First CircuitPython Program

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

Here is the original code again:

```
import board
import digitalio
import time

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

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

Imports & Libraries

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

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

The **import** statements tell the board that you're going to use a particular library in your code. In this example, we imported three libraries: **board**, **digitalio**, and **time**. All three of these libraries are built into CircuitPython, so no separate files are needed. That's one of the things that makes this an excellent first example. You don't need anything extra to make it work! **board** gives you access to the *hardware on your board*, **digitalio** lets you *access that hardware as inputs/outputs* and **time** lets you pass time by 'sleeping'.

Setting Up The LED

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

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

Your board knows the red LED as **D13**. So, we initialise that pin, and we set it to output. We set **led** to equal the rest of that information so we don't have to type it all out again later in our code.

Loop-de-loops

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

Inside our loop, we have four items:

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

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

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

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

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

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

□ What if I don't have the loop?

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

The simplest loop would be:

```
while True:
    pass
```

And remember - you can press **<CTRL><C>** to exit the loop.

See also the [Behavior section in the docs](#).

More Changes

We 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 we suggest using **code.py** as 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 looks like this:

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

This line 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 print those too.

The serial console requires a terminal program. A terminal is a program that gives you a text-based interface to perform various tasks.



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, type this command at a shell:

```
sudo apt purge modemmanager
```

Are you using Mu?

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

Please note that Mu does yet not work with nRF52 or ESP8266-based CircuitPython boards, skip down to the next section for details on using a terminal program.



First, make sure your CircuitPython board is plugged in. If you are using Windows 7, make sure you installed the drivers (<https://adafru.it/Amd>).

Once in Mu, look for the **Serial** button in the menu and click it.



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 [Advanced Serial Console on Mac and Linux \(https://adafruit.it/AAI\)](https://adafruit.it/AAI) for details on how to add yourself to the right group.

Using Something Else?

If you're not using Mu to edit, are using ESP8266 or nRF52 CircuitPython, or if for some reason you are not a fan of the built in serial console, you can run the serial console as a separate program.

Windows requires you to download a terminal program, check out this page for more details (<https://adafruit.it/AAH>)

Mac and Linux both have one built in, though other options are available for download, check this page for more details (<https://adafruit.it/AAI>)

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, we'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:

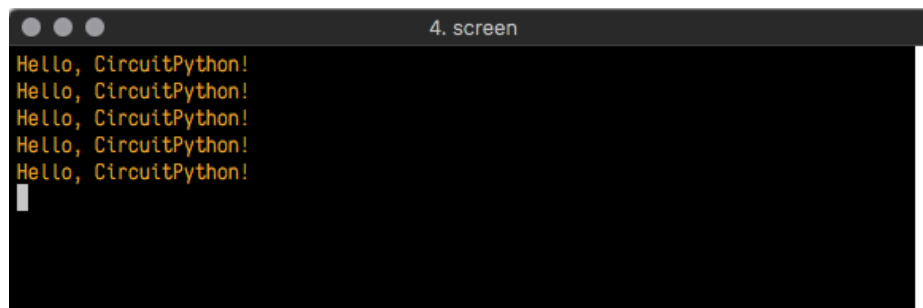
```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.D13)
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.



```
4. screen
Hello, CircuitPython!
Hello, CircuitPython!
Hello, CircuitPython!
Hello, CircuitPython!
Hello, CircuitPython!
```

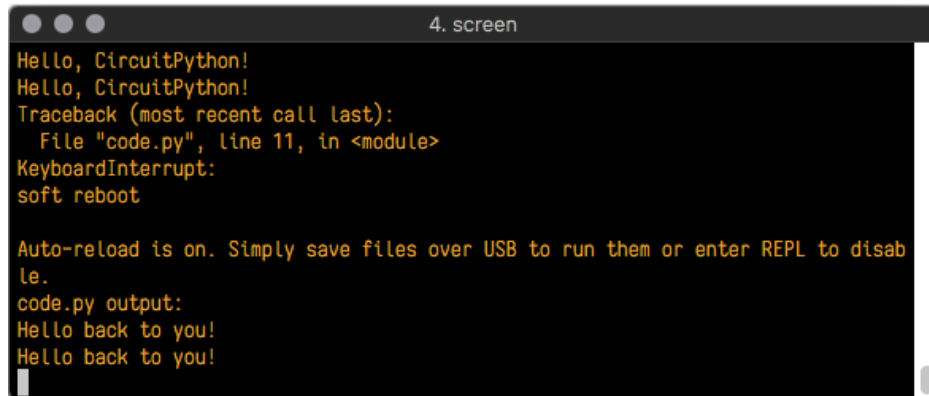
Excellent! Our print statement is showing up in our console! Try changing the printed text to something else.



```
code.py
1  import board
2  import digitalio
3  import time
4
5  led = digitalio.DigitalInOut(board.D13)
6  led.direction = digitalio.Direction.OUTPUT
7
8  while True:
9      print("Hello back to you!")
10     led.value = True
11     time.sleep(1)
12     led.value = False
13     time.sleep(1)
14
```

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!



```
4. screen
Hello, CircuitPython!
Hello, CircuitPython!
Traceback (most recent call last):
  File "code.py", line 11, in <module>
KeyboardInterrupt:
soft reboot

Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Hello back to you!
Hello back to you!
```

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 we can see how it is used.

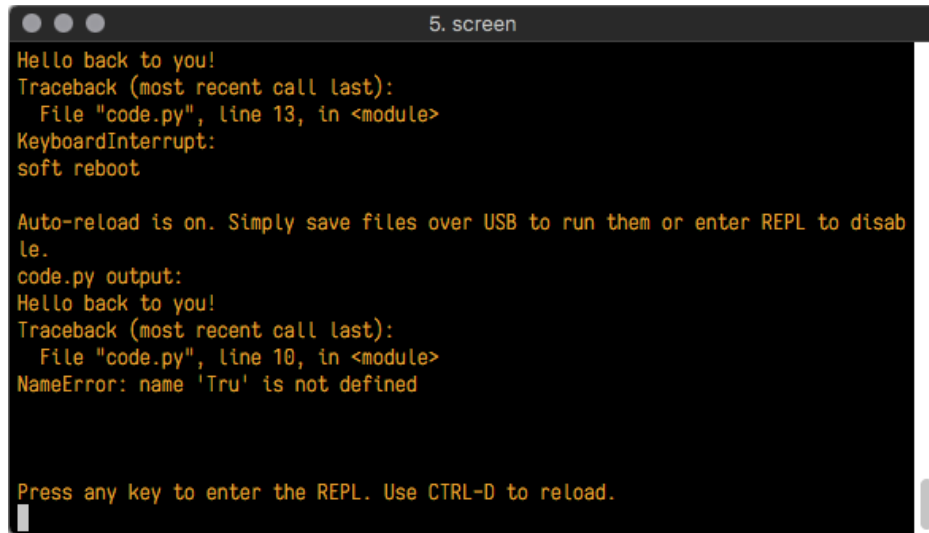
Delete the **e** at the end of **True** from the line **led.value = True** so that it says **led.value = Tru**



```
code.py
1 import board
2 import digitalio
3 import time
4
5 led = digitalio.DigitalInOut(board.D13)
6 led.direction = digitalio.Direction.OUTPUT
7
8 while True:
9     print("Hello back to you!")
10    led.value = Tru
11    time.sleep(1)
12    led.value = False
13    time.sleep(1)
14
```

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. We 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!



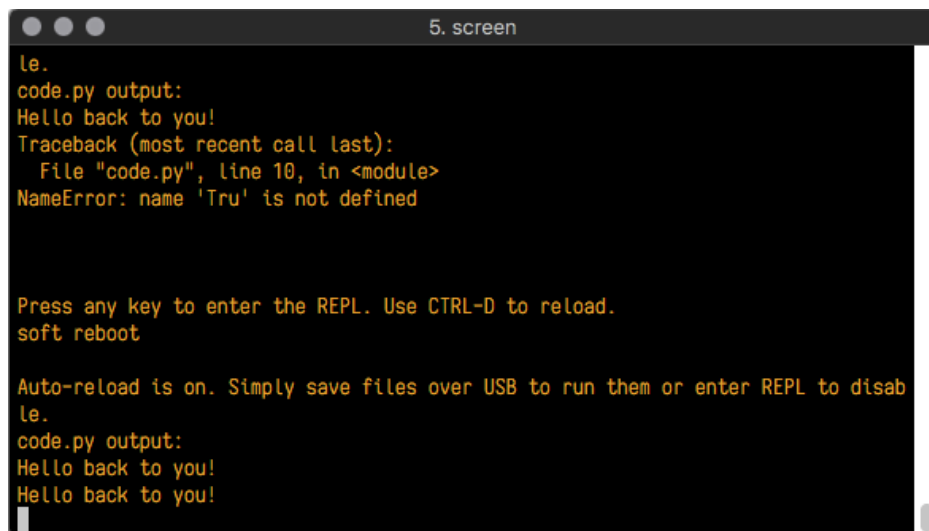
```
5. screen
Hello back to you!
Traceback (most recent call last):
  File "code.py", line 13, in <module>
KeyboardInterrupt:
soft reboot

Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Hello back to you!
Traceback (most recent call last):
  File "code.py", line 10, in <module>
NameError: name 'Tru' is not defined

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

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.



```
5. screen
le.
code.py output:
Hello back to you!
Traceback (most recent call last):
  File "code.py", line 10, in <module>
NameError: name 'Tru' is not defined

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

Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Hello back to you!
Hello back to you!
```

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. 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!

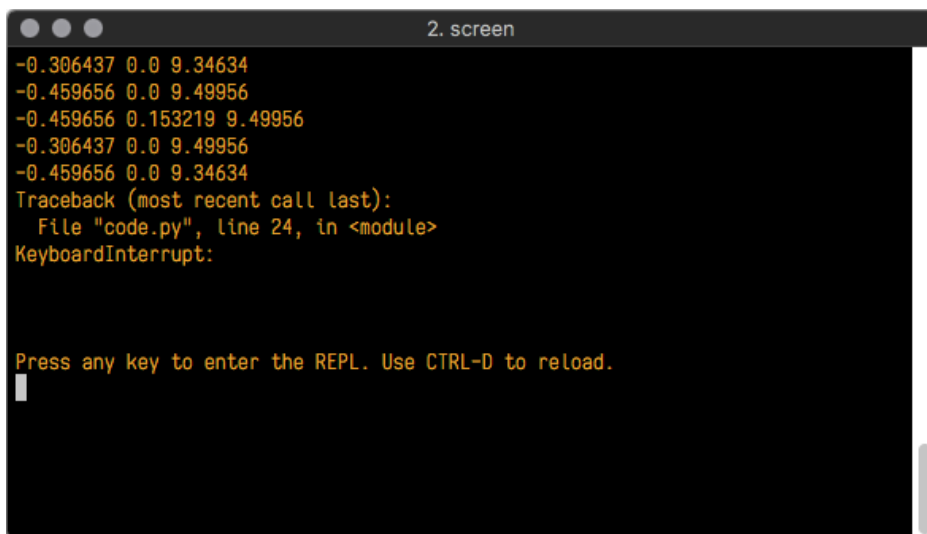
The REPL

The other feature of the serial connection is the **Read-Evaluate-Print-Loop**, 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.

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, 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.



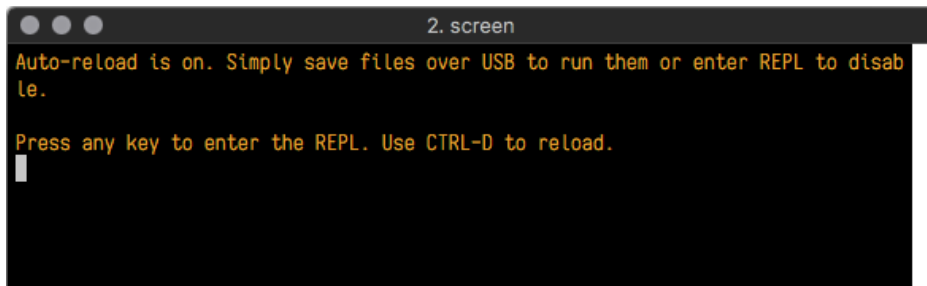
```

-0.306437 0.0 9.34634
-0.459656 0.0 9.49956
-0.459656 0.153219 9.49956
-0.306437 0.0 9.49956
-0.459656 0.0 9.34634
Traceback (most recent call last):
  File "code.py", line 24, in <module>
KeyboardInterrupt:

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

```

If there is no code running, you will enter the REPL immediately after pressing Ctrl + C. 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 disable.

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

```

Either way, once you press a key you'll see a **>>>** prompt welcoming you to the REPL!



```
Adafruit CircuitPython 2.1.0 on 2017-10-17; Adafruit CircuitPlayground Express w
ith samd21g18
>>> 
```

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 2.1.0 on 2017-10-17; Adafruit CircuitPlayground Express with samd21g18
```

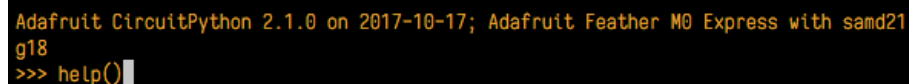
This line tells you the version of CircuitPython you're using and when it was released. Next, it gives you the type of board you're using and the type of microcontroller the board uses. Each part of this may be different for your board depending on the versions you're working with.

This is followed by the CircuitPython prompt.

```
>>>
```

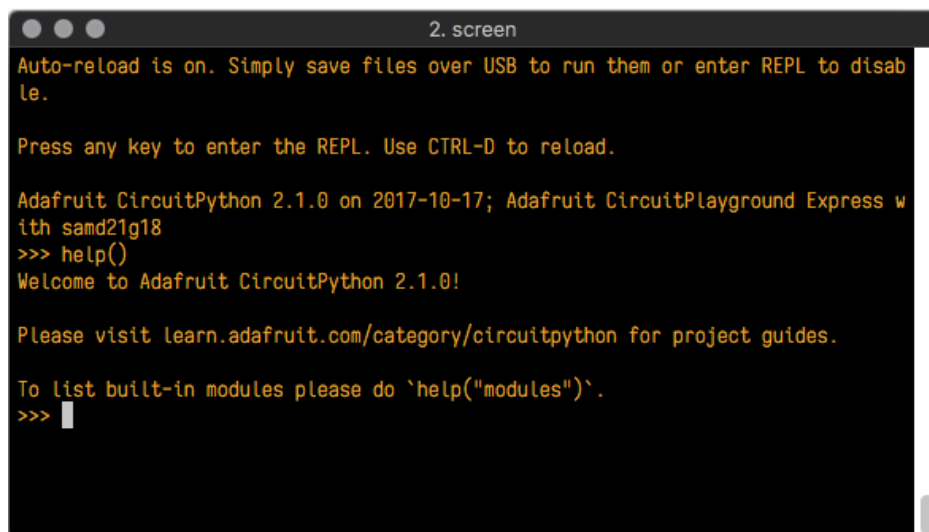
From this prompt you can run all sorts of commands and code. The first thing we'll do is run `help()`. This will tell us where to start exploring the REPL. To run code in the REPL, type it in next to the REPL prompt.

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



```
Adafruit CircuitPython 2.1.0 on 2017-10-17; Adafruit Feather M0 Express with samd21
g18
>>> help()
```

Then press enter. You should then see a message.



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

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

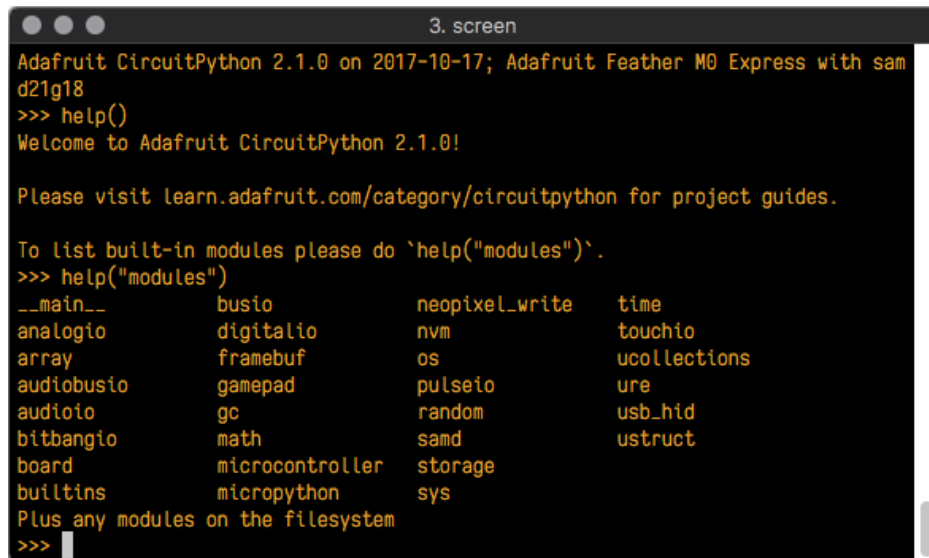
Adafruit CircuitPython 2.1.0 on 2017-10-17; Adafruit CircuitPlayground Express w
ith samd21g18
>>> help()
Welcome to Adafruit CircuitPython 2.1.0!

Please visit learn.adafruit.com/category/circuitpython for project guides.

To list built-in modules please do `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, please do `help("modules")`.` Remember the libraries 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.



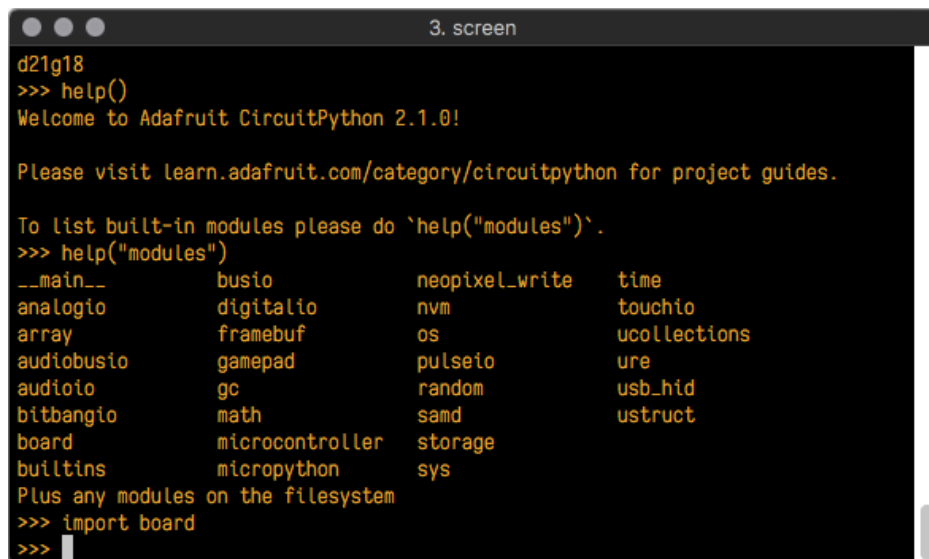
```
3. screen
Adafruit CircuitPython 2.1.0 on 2017-10-17; Adafruit Feather M0 Express with sam
d21g18
>>> help()
Welcome to Adafruit CircuitPython 2.1.0!

Please visit learn.adafruit.com/category/circuitpython for project guides.

To list built-in modules please do `help("modules")`.
>>> help("modules")
__main__      busio          neopixel_write  time
analogio      digitalio      nvm              touchio
array         framebuffer    os               ucollections
audiobusio    gamepad        pulseio          ure
audioio       gc             random           usb_hid
bitbangio     math           samd             ustruct
board         microcontroller storage
builtins      micropython    sys
Plus any modules on the filesystem
>>>
```

This is a list of all the core libraries built into CircuitPython. We discussed how `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.



```
3. screen
d21g18
>>> help()
Welcome to Adafruit CircuitPython 2.1.0!

Please visit learn.adafruit.com/category/circuitpython for project guides.

To list built-in modules please do `help("modules")`.
>>> help("modules")
__main__      busio          neopixel_write  time
analogio      digitalio      nvm              touchio
array         framebuffer    os               ucollections
audiobusio    gamepad        pulseio          ure
audioio       gc             random           usb_hid
bitbangio     math           samd             ustruct
board         microcontroller storage
builtins      micropython    sys
Plus any modules on the filesystem
>>> import board
>>>
```

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


```
3. screen

Please visit learn.adafruit.com/category/circuitpython for project guides.

To list built-in modules please do `help("modules")`.
>>> help("modules")
__main__      busio          neopixel_write  time
analogio      digitalio      nvm             touchio
array         framebuffer    os             uollections
audiobusio    gamepad        pulseio         ure
audioio       gc            random          usb_hid
bitbangio     math          samd            ustruct
board         microcontroller storage
builtins      micropython   sys
Plus any modules on the filesystem
>>> import board
>>> dir(board)
['A0', 'A1', 'A2', 'A3', 'A4', 'A5', 'SCK', 'MOSI', 'MISO', 'D0', 'RX', 'D1', 'TX',
 'SDA', 'SCL', 'D5', 'D6', 'D9', 'D10', 'D11', 'D12', 'D13', 'NEOPIXEL']
>>>
```

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 **D13** ? 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." We'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. As we said though, 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 libraries are available and explore those libraries.

Try typing more into the REPL to see what happens!

Returning to the serial console

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

You can return to the REPL at any time!

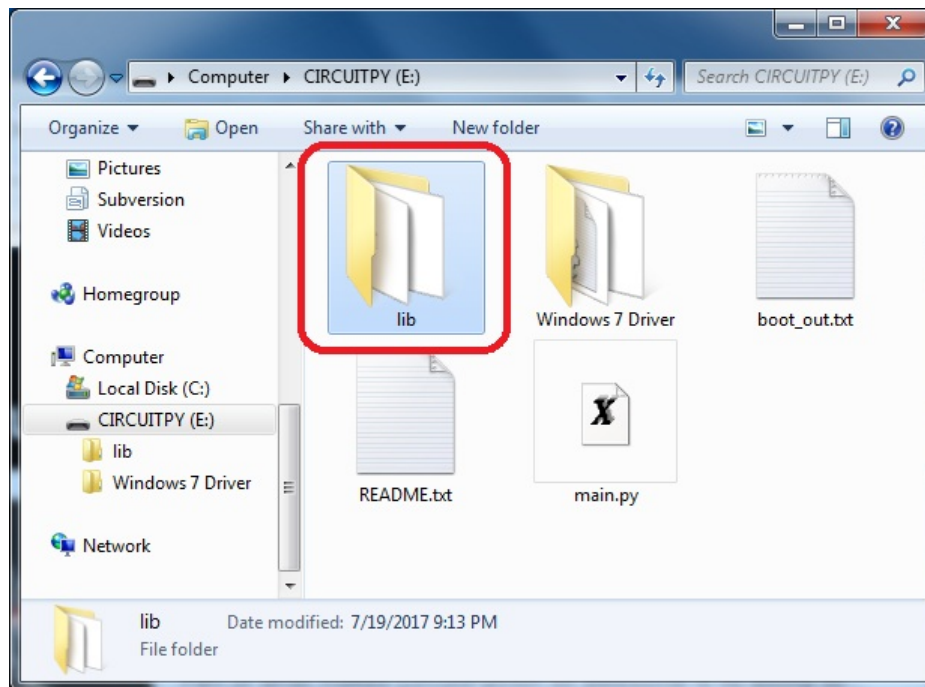
CircuitPython Libraries



As 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.

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 awesome 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 \(https://adafru.it/rar\)](https://adafru.it/rar) are a great reference for how it all should work. In Python terms, we 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, we provide a bundle full of our libraries.

Our bundle and releases 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.

Installing the CircuitPython Library Bundle

We're constantly updating and improving our libraries, so we don't (at this time) ship our CircuitPython boards with the full library bundle. Instead, you can find example code in the guides for your board that depends on external libraries. Some of these libraries may be available from us at Adafruit, some may be written by community members!

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

You can grab the latest Adafruit CircuitPython Bundle release by clicking the button below.

Note: Match up the bundle version with the version of CircuitPython you are running - 3.x library for running any version of CircuitPython 3, 4.x for running any version of CircuitPython 4, etc. If you mix libraries with major CircuitPython versions, you will most likely get errors due to changes in library interfaces possible during major version changes.

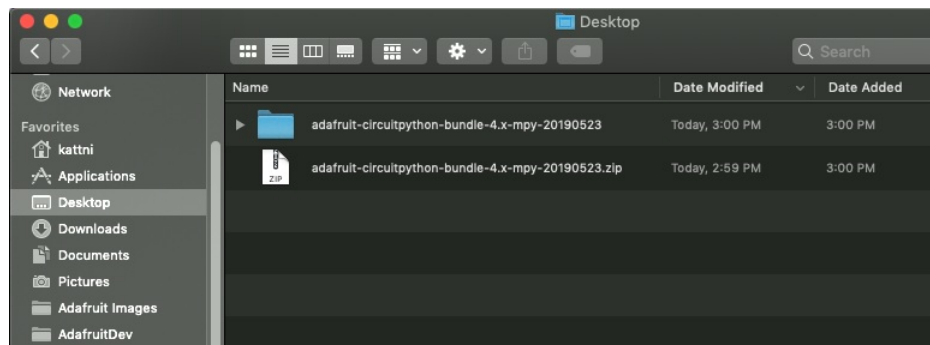
<https://adafru.it/ENC>

<https://adafru.it/ENC>

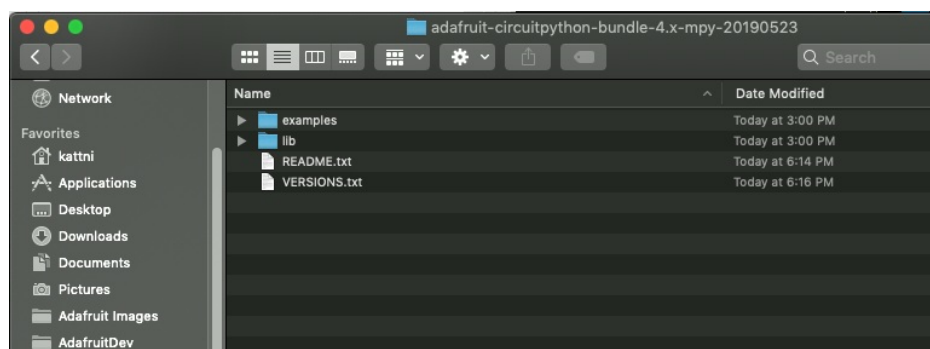
If you need another version, [you can also visit the bundle release page \(https://adafru.it/Ayy\)](https://adafru.it/Ayy) which will let you select exactly what version you're looking for, as well as information about changes.

Either way, download the version that matches your CircuitPython firmware version. If you don't know the version, look at the initial prompt in the CircuitPython REPL, which reports the version. For example, if you're running v4.0.1, download the 4.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.

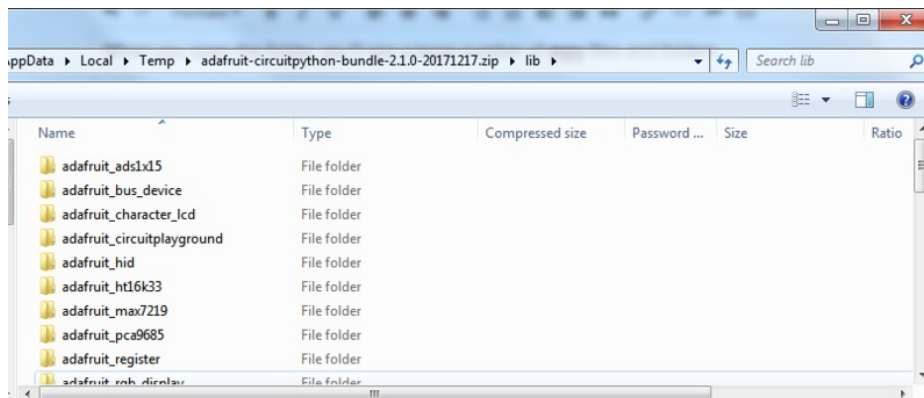
After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.



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.



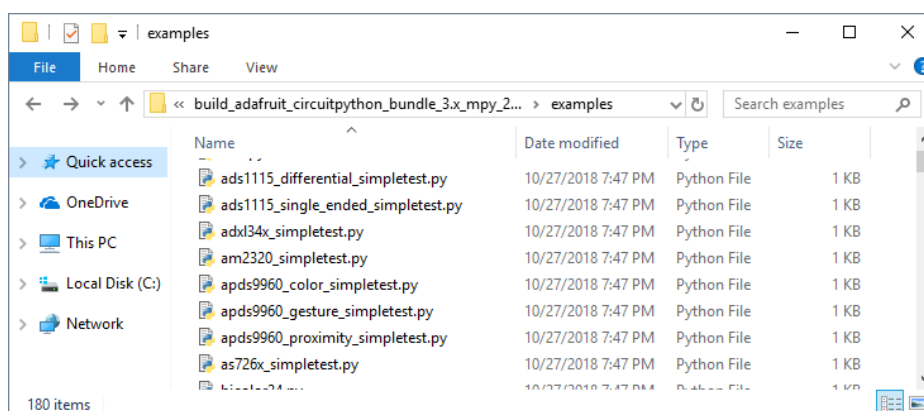
Now open the lib folder. When you open the folder, you'll see a large number of **mpy** files and folders



Example Files

All example files from each library are now included in the bundles, 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 you'll want to create a **lib** folder on your **CIRCUITPY** drive. Open the drive, right click, choose the option to create a new folder, and call it **lib**. 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**.

This also applies to example files. They are only supplied as raw **.py** files, so they may need to be converted to **.mpy** using the **mpy-cross** utility if you encounter **MemoryErrors**. This is discussed in the [CircuitPython Essentials Guide \(https://adafru.it/CTw\)](https://adafru.it/CTw). Usage is the same as described above in the Express Boards section. Note: If you do not place examples in a separate folder, you would remove the examples from the **import** statement.

Example: **ImportError** Due to Missing Library

If you choose to load libraries as you need them, you may write up code that tries to use a library you haven't yet loaded. We're going to 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.

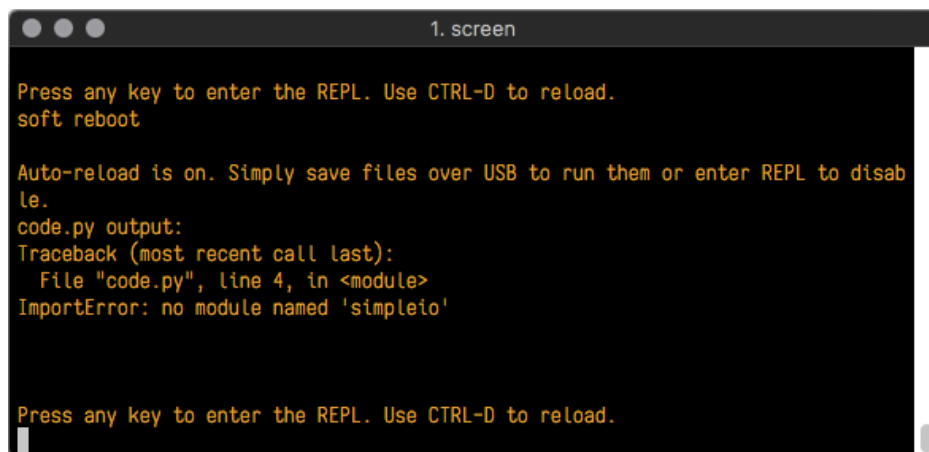
Let's use a modified version of the blinky example.

```
import board
import time
import simpleio

led = simpleio.DigitalOut(board.D13)

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.



```
1. screen

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

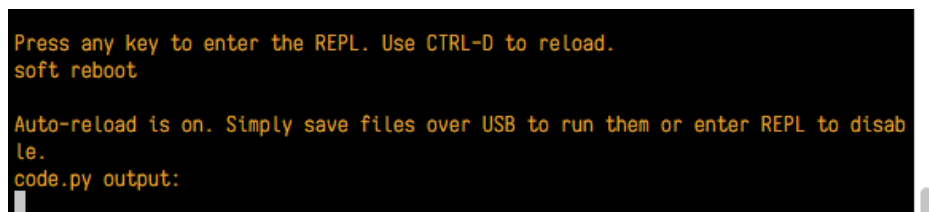
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Traceback (most recent call last):
  File "code.py", line 4, in <module>
    ImportError: no module named 'simpleio'

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

We have an **ImportError**. It says there is **no module named 'simpleio'**. That's the one we just included in our 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 we're looking for! Follow the steps above to load an individual library file.

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



```
1. screen

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

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

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 a Trinket M0 or Gemma M0, you'll want to follow the same steps in the example above to install libraries as you need them. You don't always need to wait for an `ImportError` as you probably know what library you added to your code. Simply open the `lib` folder you downloaded, find the library you need, and drag it to the `lib` folder on your **CIRCUITPY** drive.

You may end up running out of space on your Trinket M0 or Gemma M0 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 them in the Troubleshooting page in the Learn guides for your board.

Updating CircuitPython Libraries/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.

Frequently Asked Questions

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



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.

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>).

□ Is ESP8266 or ESP32 supported in CircuitPython? Why not?

We are dropping ESP8266 support as of 4.x - For more information please read about it here!

<https://learn.adafruit.com/welcome-to-circuitpython/circuitpython-for-esp8266>

□ How do I connect to the Internet with CircuitPython?

If you'd like to add WiFi support, check out our [guide on ESP32/ESP8266 as a co-processor](#).

□ Is there asyncio support in CircuitPython

We do not have asyncio support in CircuitPython at this time

📌 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!](#)

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 (REPL).

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 \(https://adafruit.it/uap\)](https://adafruit.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 be because the memory gets fragmented differently depending on allocation order and the size of objects. Loading `.mpy` files uses less memory so it's 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 the CircuitPython 2.x version of `mpy-cross` for your operating system from the [CircuitPython Releases page \(https://adafru.it/tBa\)](https://adafru.it/tBa) under the latest 2.x version.

You can build `mpy-cross` for CircuitPython 3.x by cloning the [CircuitPython GitHub repo \(https://adafru.it/tB7\)](https://adafru.it/tB7), and running `make` in the `circuitpython/mpy-cross/` directory. Then run `./mpy-cross path/to/foo.py` to create a `foo.mpy` in the same directory as the original file.

How do I check how much memory I have free?

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

Will give you the number of bytes available for use.

Does CircuitPython support interrupts?

No. CircuitPython does not currently support interrupts. We do not have an estimated time for when they will be included.

Does Feather M0 support WINC1500?

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

Can AVR's such as ATmega328 or ATmega2560 run CircuitPython?

No.

Commonly Used Acronyms

CP or CPy = [CircuitPython \(https://adafru.it/cpy-welcome\)](https://adafru.it/cpy-welcome)

CPC = [Circuit Playground Classic \(https://adafru.it/ncE\)](https://adafru.it/ncE)

CPX = [Circuit Playground Express \(https://adafru.it/wpF\)](https://adafru.it/wpF)

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 \(https://adafru.it/Amd\)](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, Feather M0 Basic, and other non-Express Feather M0 variants.

Non-Express Boards: Gemma and Trinket

CircuitPython runs nicely on the Gemma M0 or Trinket 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 playback 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. 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 $\pm 3.4e38$. The smallest magnitude that can be represented with full

accuracy is about $\pm 1.7\text{e-}38$, though numbers as small as $\pm 5.6\text{e-}45$ 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). (<https://adafru.it/zwA>)

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 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>).

CPLAYBOOT, TRINKETBOOT, FEATHERBOOT, or GEMMABOOT Drive Not Present

You may have a different board.

Only Adafruit Express boards and the Trinket M0 and Gemma M0 boards ship with the **UF2 bootloader** (<https://adafru.it/zbX>) installed. 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.

Windows 10

Did you install the Adafruit Windows Drivers package by mistake? 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.

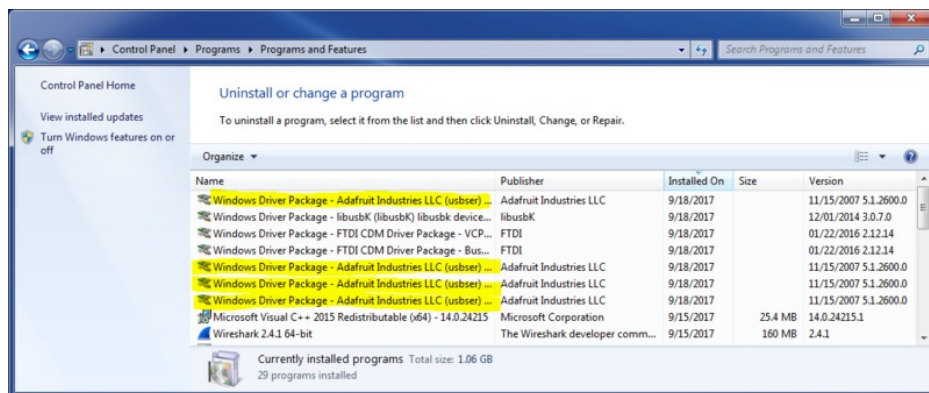
Windows 7

Version 2.0.0.0 or later of the Adafruit Windows Drivers will fix the missing `boardnameBOOT` drive problem on Windows 7. To resolve this, first uninstall the old versions of the drivers:

- Unplug any boards. In **Uninstall or Change a Program (Control Panel->Programs->Uninstall a program)**, uninstall everything named "Windows Driver Package - Adafruit Industries LLC ...".

MacOS

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

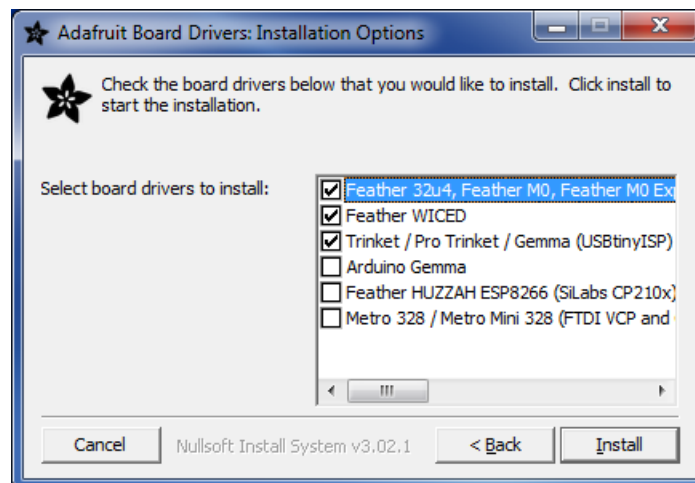


- Now install the new 2.3.0.0 (or higher) Adafruit Windows Drivers Package:

<https://adafru.it/ABO>

<https://adafru.it/ABO>

- When running the installer, you'll be shown a list of drivers to choose from. You can check and uncheck the boxes to choose which drivers to install.



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

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

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

On Windows, several third-party programs we know of 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**: We have seen 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

Kaspersky anti-virus can block the appearance of the **CIRCUITPY** drive. We haven't yet figured out a settings change 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.

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.

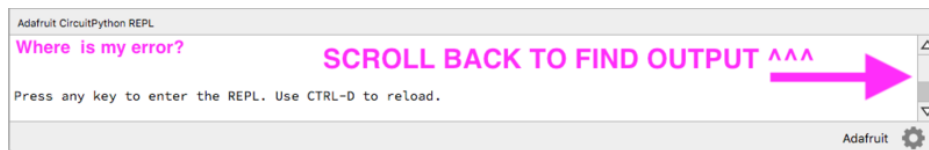
Depending on the size of your screen or Mu window, when you open the serial console, the serial console panel may be very small. This can be a problem. A basic CircuitPython error takes 10 lines to display!

```
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.

CircuitPython RGB Status Light

Nearly all Adafruit 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. They do NOT indicate any status while running CircuitPython.

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.

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 2.x and 3.x, as well as between 1.x and 2.x.

So, for instance, if you upgraded to CircuitPython 3.x from 2.x you'll need to download a newer version of the library that triggered the error on **import**. They are all available in the [Adafruit bundle \(https://adafru.it/y8E\)](https://adafru.it/y8E).

Make sure to download a version with 2.0.0 or higher in the filename if you're using CircuitPython version 2.2.4, and the version with 3.0.0 or higher in the filename if you're using CircuitPython version 3.0.

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.

First check - have you used Arduino to program your board? If so, CircuitPython is no longer able to provide the USB services. Reset the board so you get a **boardnameBOOT** drive rather than a **CIRCUITPY** drive, copy the latest version of CircuitPython (**.uf2**) back to the board, then Reset. This may restore **CIRCUITPY** functionality.

If still broken - 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.

In this situation, 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.

Easiest Way: Use `storage.erase_filesystem()`

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

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

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

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

Old Way: For the Circuit Playground Express, Feather M0 Express, and Metro M0 Express:

If you can't get to the REPL, or you're running a version of CircuitPython before 2.3.0, and you don't want to upgrade, you can do this.

1. Download the correct erase file:

<https://adafru.it/AdI>

<https://adafru.it/AdI>

<https://adafru.it/AdJ>

<https://adafru.it/AdJ>

<https://adafru.it/EVK>

<https://adafru.it/EVK>

<https://adafru.it/AdK>

<https://adafru.it/AdK>

<https://adafru.it/EoM>

<https://adafru.it/EoM>

<https://adafru.it/DjD>

<https://adafru.it/DjD>

<https://adafru.it/DBA>

<https://adafru.it/DBA>

<https://adafru.it/Eca>

<https://adafru.it/Eca>

<https://adafru.it/Gnc>

<https://adafru.it/Gnc>

<https://adafru.it/GAN>

<https://adafru.it/GAN>

<https://adafru.it/GAO>

<https://adafru.it/GAO>

<https://adafru.it/Jat>

<https://adafru.it/Jat>

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 onboard NeoPixel will turn yellow or blue, indicating the erase has started.
5. After approximately 15 seconds, the mainboard NeoPixel 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/Amd>) `.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 install your libraries and code!

Old Way: For Non-Express Boards with a UF2 bootloader (Gemma M0, Trinket M0):

If you can't get to the REPL, or you're running a version of CircuitPython before 2.3.0, and you don't want to upgrade, you can do this.

1. Download the erase file:

<https://adafru.it/AdL>

<https://adafru.it/AdL>

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/Amd>) `.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>) You'll also need to install your libraries and code!

Old Way: For non-Express Boards without a UF2 bootloader (Feather M0 Basic Proto, Feather Adalogger, Arduino Zero):

If you are running a version of CircuitPython before 2.3.0, and you don't want to upgrade, or you can't get to the REPL, you can do this.

Just 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 Non-Express 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 couple ways to free up space.

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. Its ~12KiB or so.

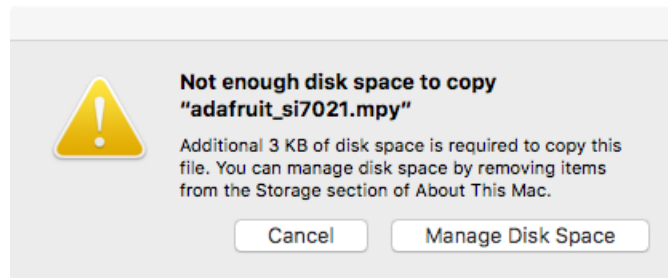
Delete something!

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.

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, we recommend that 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 we're counting bytes.

MacOS loves to add extra 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 \(https://adafru.it/u1c\)](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 `foo.mpy` file to the board use a command like:

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

(Replace `foo.mpy` with the name of the file you want to copy.) Or to copy a folder and all of its child files/folders 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 foo.mpy /Volumes/CIRCUITPY/lib
# This is safer, and will complain if a lib folder does not exist.
cp -X foo.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 list the amount of space used on the `CIRCUITPY` drive with the `df` command:

```
1. bash
X bash %1 X bash %2 X bash %3
(venv) tannewt@shallan:/Volumes $ df -h /Volumes/CIRCUITPY/
Filesystem      Size  Used Avail Capacity iused ifree %used  Mounted on
/dev/disk3s1    59Ki  54Ki  5.5Ki   91%    128     0  100%  /Volumes/CIRCUITPY
(venv) tannewt@shallan:/Volumes $ ls -a CIRCUITPY/
./              ._Trashes*    boot_out.txt*
../             ._original_code.py* code.py*
.TemporaryItems/ .fseventsd/   lib/
.Trashes/        README.txt*   original_code.py*
._.TemporaryItems* Windows 7 Driver/
(venv) tannewt@shallan:/Volumes $
```

Lets remove the `._` files first.

```
1. bash
X bash %1 X bash %2 X bash %3
(venv) tannewt@shallan:/Volumes $ df -h /Volumes/CIRCUITPY/
Filesystem      Size  Used Avail Capacity iused ifree %used  Mounted on
/dev/disk3s1    59Ki  54Ki  5.5Ki   91%    128     0  100%  /Volumes/CIRCUITPY
(venv) tannewt@shallan:/Volumes $ ls -a CIRCUITPY/
./              ._Trashes*    boot_out.txt*
../             ._original_code.py* code.py*
.TemporaryItems/ .fseventsd/   lib/
.Trashes/        README.txt*   original_code.py*
._.TemporaryItems* Windows 7 Driver/
(venv) tannewt@shallan:/Volumes $ rm CIRCUITPY/._*
(venv) tannewt@shallan:/Volumes $ df -h /Volumes/CIRCUITPY/
Filesystem      Size  Used Avail Capacity iused ifree %used  Mounted on
/dev/disk3s1    59Ki  42Ki  18Ki   71%    128     0  100%  /Volumes/CIRCUITPY
(venv) tannewt@shallan:/Volumes $ ls -a CIRCUITPY/
./              .Trashes/     Windows 7 Driver/ lib/
../             .fseventsd/   boot_out.txt*   original_code.py*
.TemporaryItems/ README.txt*    code.py*
(venv) tannewt@shallan:/Volumes $
```

Whoa! We have 13Ki more than before! This space can now be used for libraries and code!

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/re-install CircuitPython *whenever you want!* Heck, you can change your mind every day!

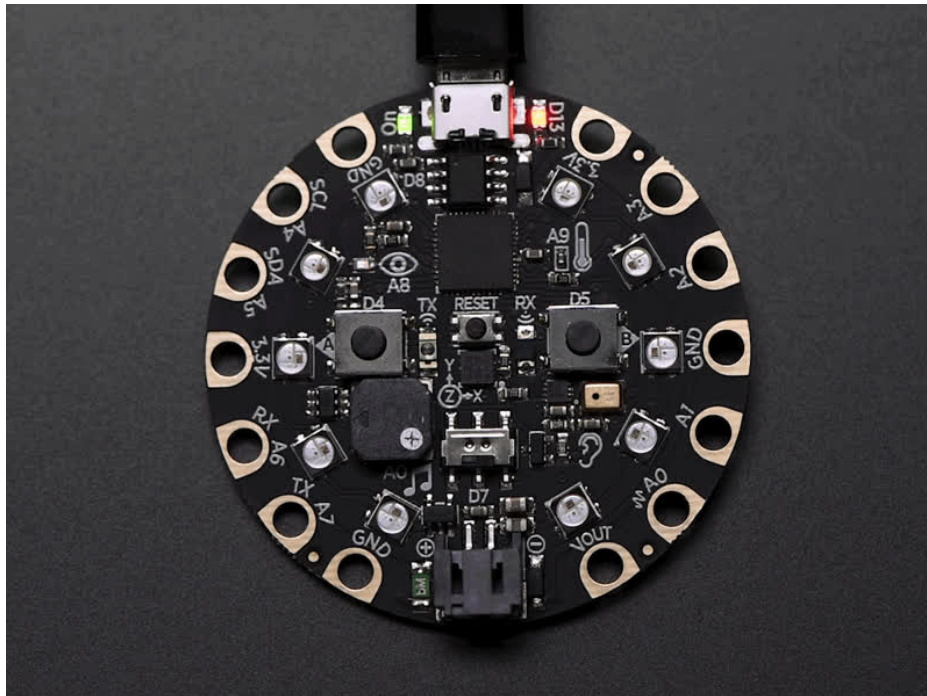
Backup Your Code

Before uninstalling CircuitPython, don't forget to make a backup of the code you have on the little disk drive. That means your **main.py** or **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.

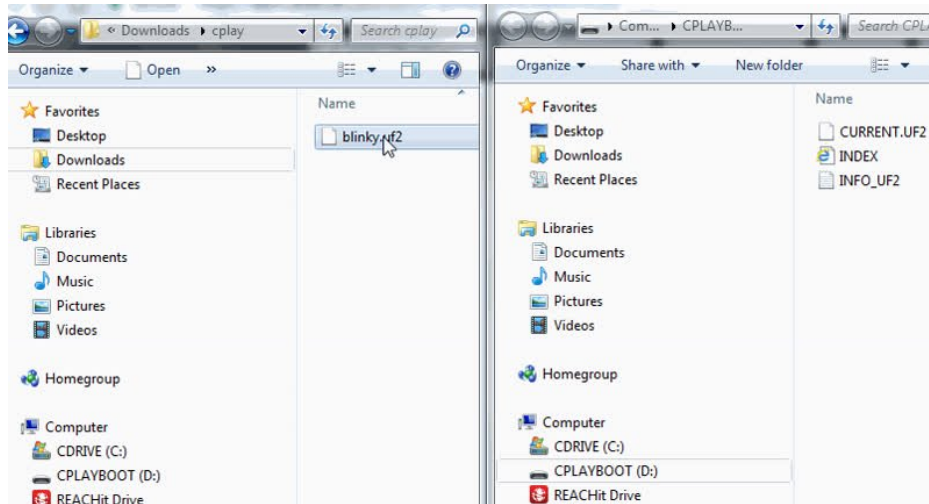
Moving Circuit Playground Express to MakeCode

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 **...BOOT** drive.



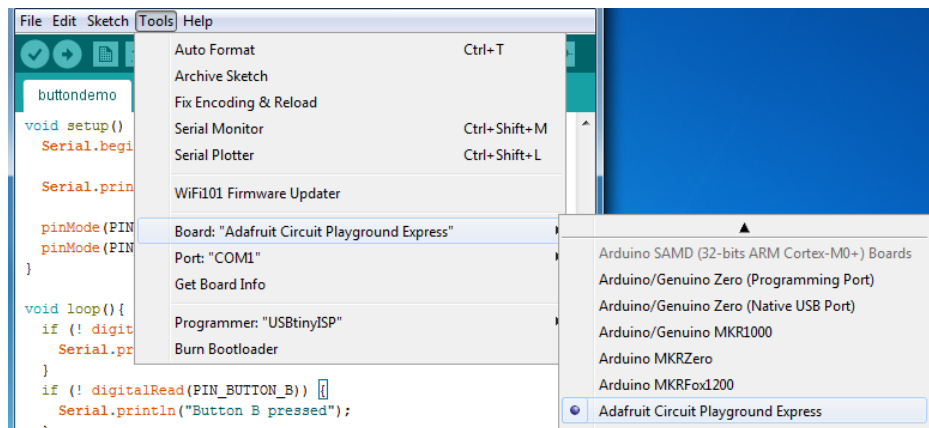
Your MakeCode is now running and CircuitPython has been removed. Going forward you only have to **single click** the reset button

Moving to Arduino

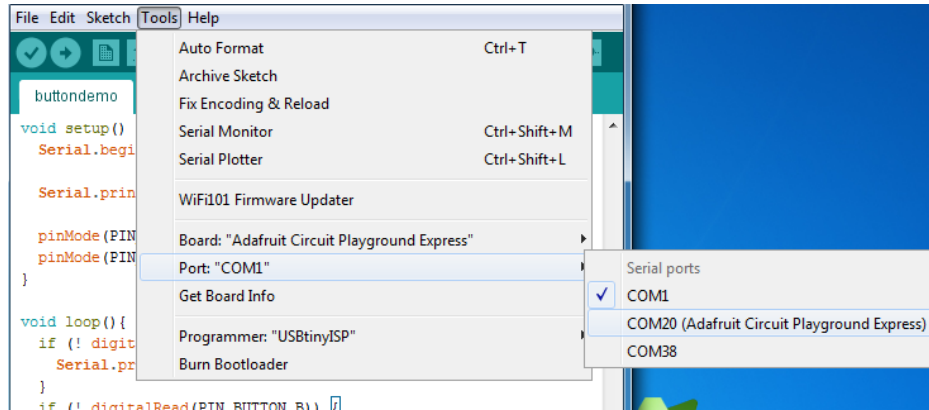
If you want to change your firmware to Arduino, it's also pretty easy.

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

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



Select the correct matching Port:



Create a new simple Blink sketch example:

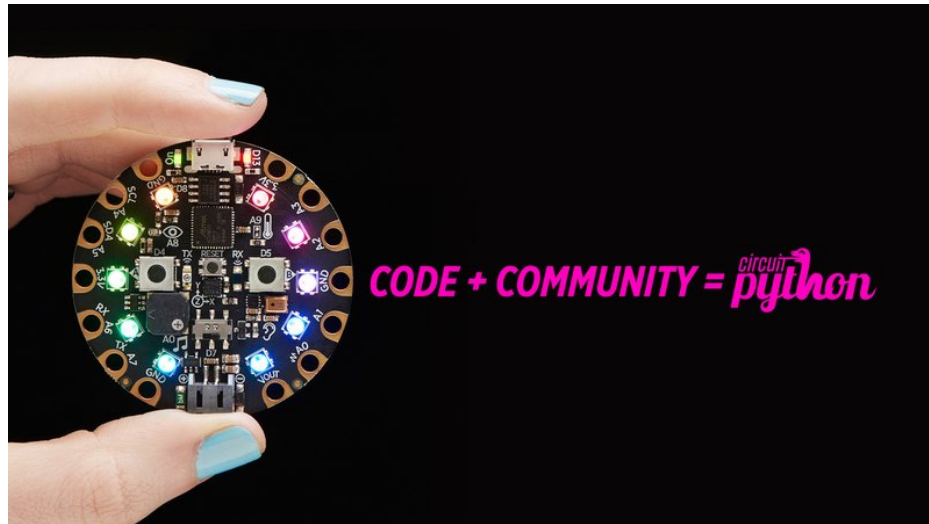
```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// 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

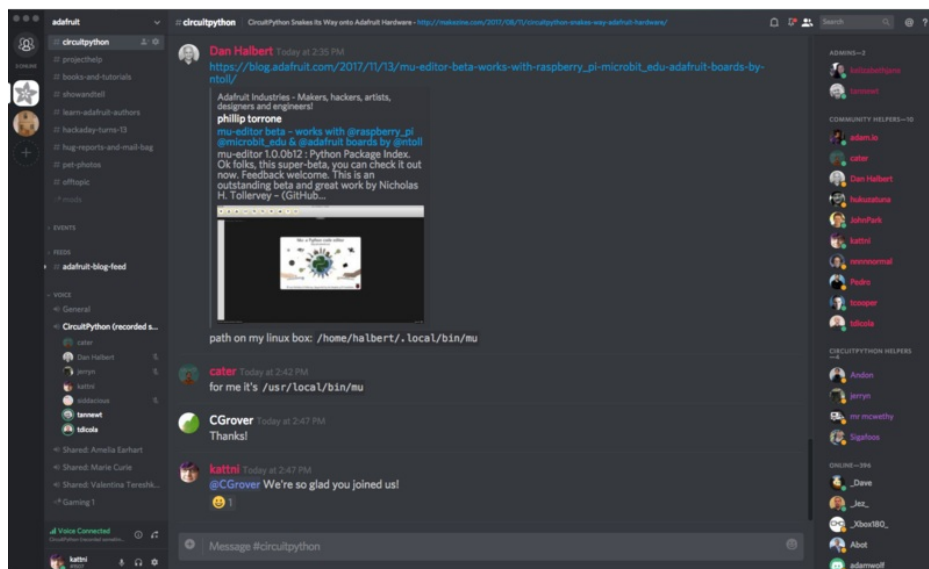
Welcome to the Community!



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. It doesn't matter whether this is your first microcontroller board or you're a computer engineer, you have something important to offer the Adafruit CircuitPython community. We're going to highlight 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 #projecthelp channel for assistance with your current project or help coming up with ideas for your next one. There's the #showandtell 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.

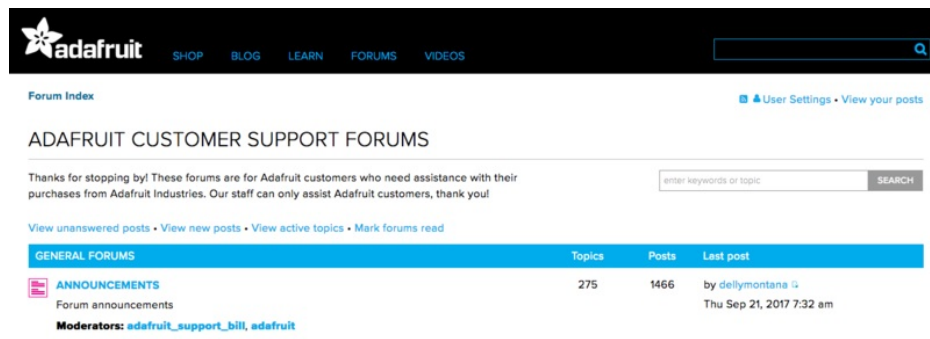
The CircuitPython channel is where to go with your CircuitPython questions. #circuitpython is there for new users and developers alike so feel free to ask a question or post a comment! Everyone of any experience level is welcome to join in on the conversation. We'd love to hear what you have to say!

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. We're looking forward to meeting you!

Adafruit Forums



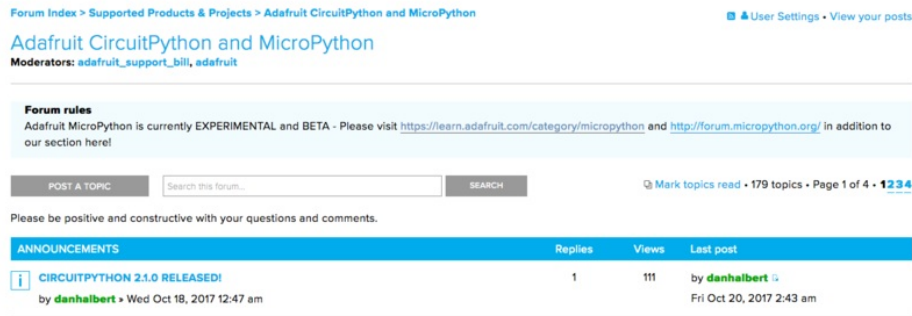
The screenshot shows the Adafruit Customer Support Forums homepage. At the top is the Adafruit logo and navigation links: SHOP, BLOG, LEARN, FORUMS, and VIDEOS. Below the navigation bar is a search bar and links for "User Settings" and "View your posts". The main heading is "ADAFRUIT CUSTOMER SUPPORT FORUMS". A welcome message states: "Thanks for stopping by! These forums are for Adafruit customers who need assistance with their purchases from Adafruit Industries. Our staff can only assist Adafruit customers, thank you!". Below this is a search bar with the placeholder "enter keywords or topic" and a "SEARCH" button. A navigation bar includes links: "View unanswered posts", "View new posts", "View active topics", and "Mark forums read". The main content area is titled "GENERAL FORUMS" and contains a table with columns "Topics", "Posts", and "Last post". The first row is for "ANNOUNCEMENTS" with 275 topics, 1466 posts, and a last post by "dellymontana" on "Thu Sep 21, 2017 7:32 am". Below the table, it lists moderators: "adafruit_support_bill, adafruit".

GENERAL FORUMS	Topics	Posts	Last post
ANNOUNCEMENTS Forum announcements Moderators: adafruit_support_bill , adafruit	275	1466	by dellymontana ↗ Thu Sep 21, 2017 7:32 am

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

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

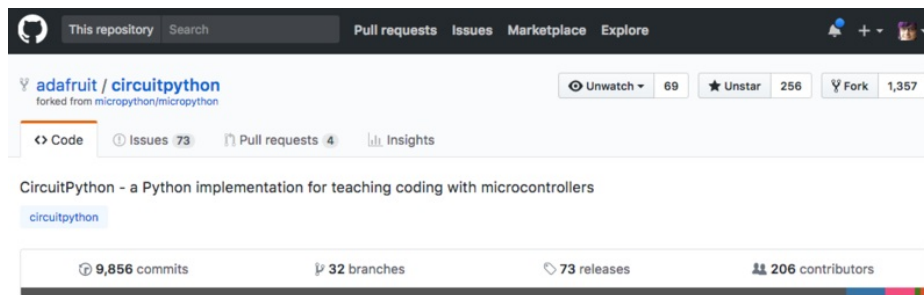
There are forum categories that cover all kinds of topics, including everything Adafruit. The [Adafruit CircuitPython and MicroPython \(https://adafru.it/xXA\)](https://adafru.it/xXA) category under "Supported Products & Projects" is the best place to post your CircuitPython questions.



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.

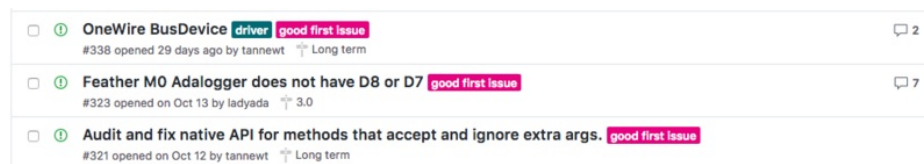
You might think you're just getting started, but you definitely know something that someone else doesn't. The great thing about the forums is that you can help others too! Everyone is welcome and encouraged to provide constructive feedback to any of the posted questions. This is an excellent way to contribute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

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 building CircuitPython. GitHub is the best source of ways to contribute to [CircuitPython \(https://adafru.it/tB7\)](https://adafru.it/tB7) itself. 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. Head over to [adafruit/circuitpython \(https://adafru.it/tB7\)](https://adafru.it/tB7) on GitHub, click on "Issues (<https://adafru.it/Bee>)", and you'll find a list that includes issues labeled "good first issue (<https://adafru.it/Bef>)". These are things we've 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.



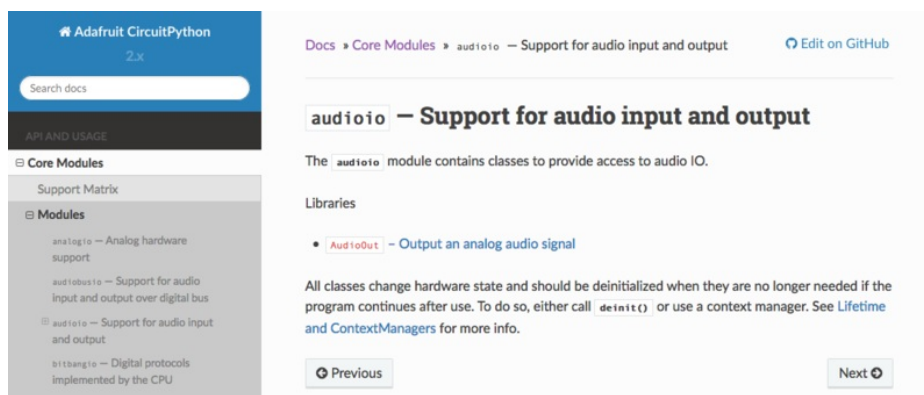
Already experienced and looking for a challenge? Checkout the rest of the issues list and you'll find plenty of ways to contribute. You'll find everything from new driver requests to core module updates. There's plenty of opportunities for everyone at any level!

When working with CircuitPython, you may find problems. If you find a bug, that's great! We love bugs! Posting a detailed issue to GitHub is an invaluable way to contribute to improving CircuitPython. 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 current and beta releases is a very important part of contributing CircuitPython. We can't possibly find all the problems ourselves! We 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!

ReadTheDocs



[ReadTheDocs \(https://adafru.it/Beg\)](https://adafru.it/Beg) is an excellent resource for a more in depth look at CircuitPython. This is where you'll find things like API documentation and details about core modules. There is also a Design Guide that includes contribution guidelines for CircuitPython.

RTD gives you access to a low level look at CircuitPython. There are details about each of the [core modules \(https://adafru.it/Beh\)](https://adafru.it/Beh). Each module lists the available libraries. Each module library page lists the available parameters and an explanation for each. In many cases, you'll find quick code examples to help you understand how the modules and parameters work, however it won't have detailed explanations like the Learn Guides. If you want help understanding what's going on behind the scenes in any CircuitPython code you're writing, ReadTheDocs is there to help!

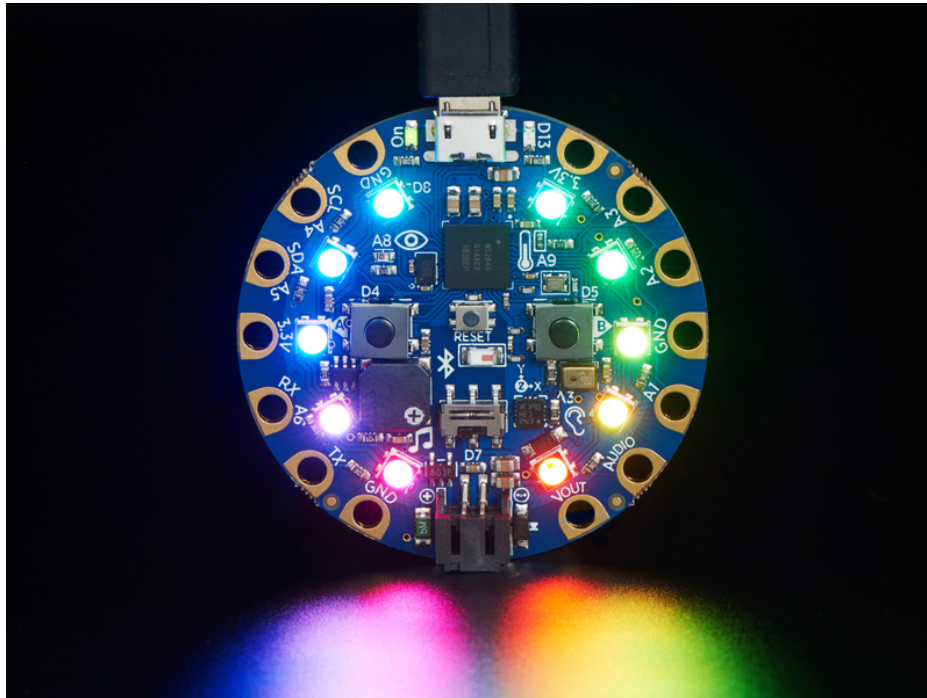
Here is blinky:

```
import digitalio
from board import *
import time

led = digitalio.DigitalInOut(D13)
led.direction = digitalio.Direction.OUTPUT
while True:
    led.value = True
    time.sleep(0.1)
    led.value = False
    time.sleep(0.1)
```

CircuitPython Made Easy

CircuitPython Made Easy (<https://adafru.it/BQj>)



Here are examples of some of the many things you can do with the Circuit Playground Bluefruit with CircuitPython!



Many of the following examples are shown using Circuit Playground Express. The code works exactly the same way on the Circuit Playground Bluefruit. Simply copy the code and follow along with your Circuit Playground Bluefruit!

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.

Things 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 \(https://adafru.it/BfQ\)](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



Though the following example uses the Circuit Playground Express to demonstrate, the code works exactly the same way with the Circuit Playground Bluefruit. Simply copy the code and follow along with your Circuit Playground Bluefruit!

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

This quick-start example shows how you can use one of the Circuit Playground Express buttons as an *input* to control another digital *output* - the built in LED

Copy and paste the code block into **code.py** using your favorite text editor, and save the file, to run the demo

```
# Circuit Playground digitalio example

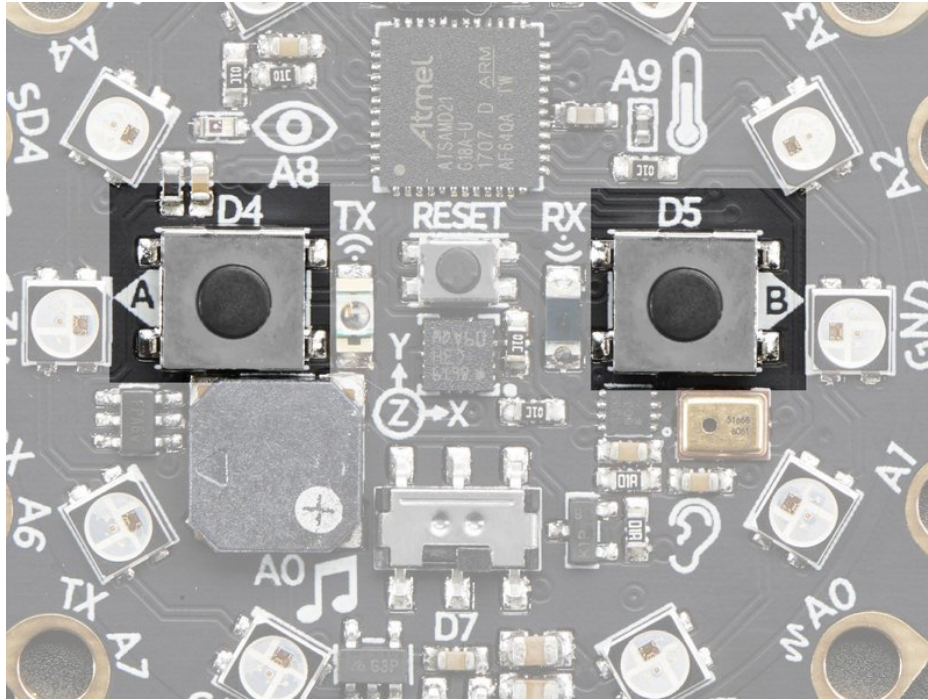
import time
import board
import digitalio

led = digitalio.DigitalInOut(board.D13)
led.switch_to_output()

button = digitalio.DigitalInOut(board.BUTTON_A)
button.switch_to_input(pull=digitalio.Pull.DOWN)

while True:
    if button.value: # button is pushed
        led.value = True
    else:
        led.value = False

    time.sleep(0.01)
```



We're using the built-in pushbuttons in this example because it's very easy to get started, but you can use ALL KINDS of different buttons and switches, even homemade ones such as tinfoil or pennies, as digital inputs connected to the Digital IO pads!

Note that we made the code a little less 'pythonic' than necessary, the if/then could be replaced with a simple `led.value = not button.value` but I wanted to make it super clear how to test the inputs. When the interpreter gets to evaluating `button.value` that is when it will read the digital input.

Press Button A (the one on the left), and the onboard red LED will turn on!

Note that on the M0/SAMD based CircuitPython boards, at least, you can also have internal pullups with `Pull.UP` when using external buttons, but the built in buttons require `Pull.DOWN`.

Maybe you're setting up your own external button with pullup or pulldown resistor. If you want to turn off the internal pullup/pulldown simply include `button.switch_to_input()`.

Going Beyond the Lesson!

It's time to flex your new learnings and try something different!!

Experiment 1

See if you can adjust your code so that you use Button B instead of Button A.

It only takes a small change to switch buttons. If you get stuck, click on the blurry text below to reveal a hint and then the answer:

You need to change one single, solitary letter!


```
button = DigitalInOut(board.BUTTON_B)
```

Experiment 2

Perhaps you want to be sure there are no accidental illuminations of the red LED! Make it so that BOTH buttons must be pressed in order to light the red LED.

Hints:

You'll need to declare a variable for the second button, just as you did with the first. You'll also need to set it up as an input, with pull down resistance.

It's a good idea to rename the original **button** variable to **buttonA**, and the new set to **buttonB**.

To check both buttons in the 'if' statement, you'll use an 'and' to string together both value checks.

```
if buttonA.value == True and buttonB.value ==  
True:
```

Experiment 3

Try testing the slide switch instead of the buttons. For the slide switch you need to use **Pull.UP** instead of **Pull.DOWN**.

Hints:

```
switch = DigitalInOut(board.SLIDE_SWITCH)  
switch.direction = Direction.INPUT  
switch.pull = Pull.UP
```

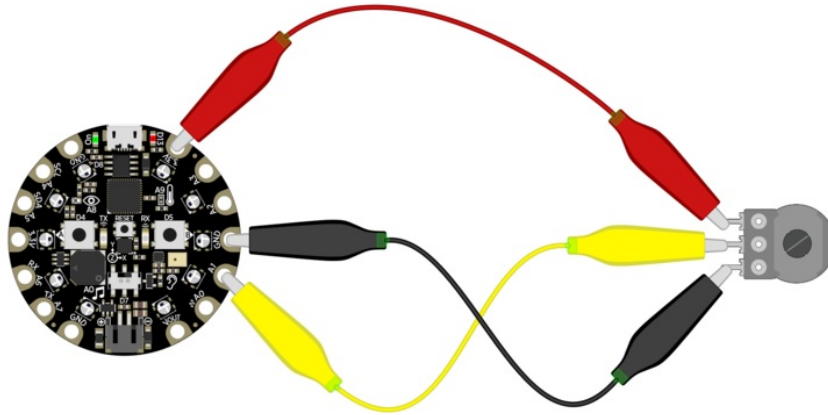
```
if switch.value is True: # switch is slid to the  
left
```

CircuitPython Analog In



Though the following example uses the Circuit Playground Express to demonstrate, the code works exactly the same way with the Circuit Playground Bluefruit. Simply copy the code and follow along with your Circuit Playground Bluefruit!

This quick-start example shows how you can read the analog voltage of a potentiometer connected to the Circuit Playground Express.



First, connect your potentiometer to the Circuit Playground Express using three alligator clip leads, as shown. The connections are:

- Left pot connection to **3.3V**
- Center pot (wiper) to **A1**
- Right pot connection to **GND**



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 the Circuit Playground Express on A1.

Copy and paste the code block into `code.py` using your favorite code editor, and save the file, to run the demo.

```
# Circuit Playground AnalogIn
# Reads the analog voltage level from a 10k potentiometer connected to GND, 3.3V, and pin A1
# and prints the results to the serial console.

import time
import board
import analogio

analogin = analogio.AnalogIn(board.A1)

def getVoltage(pin): # helper
    return (pin.value * 3.3) / 65536

while True:
    print("Analog Voltage: %f" % getVoltage(analogin))
    time.sleep(0.1)
```

Creating an Analog Input

`analogin = analogio.AnalogIn(board.A1)` creates an object named `analogin` which is connected to the A1 pad on the Circuit Playground Express.



You can use many of different kinds of external analog sensors connected to the Analog IO pads, such as distance sensors, flex sensors, and more!!

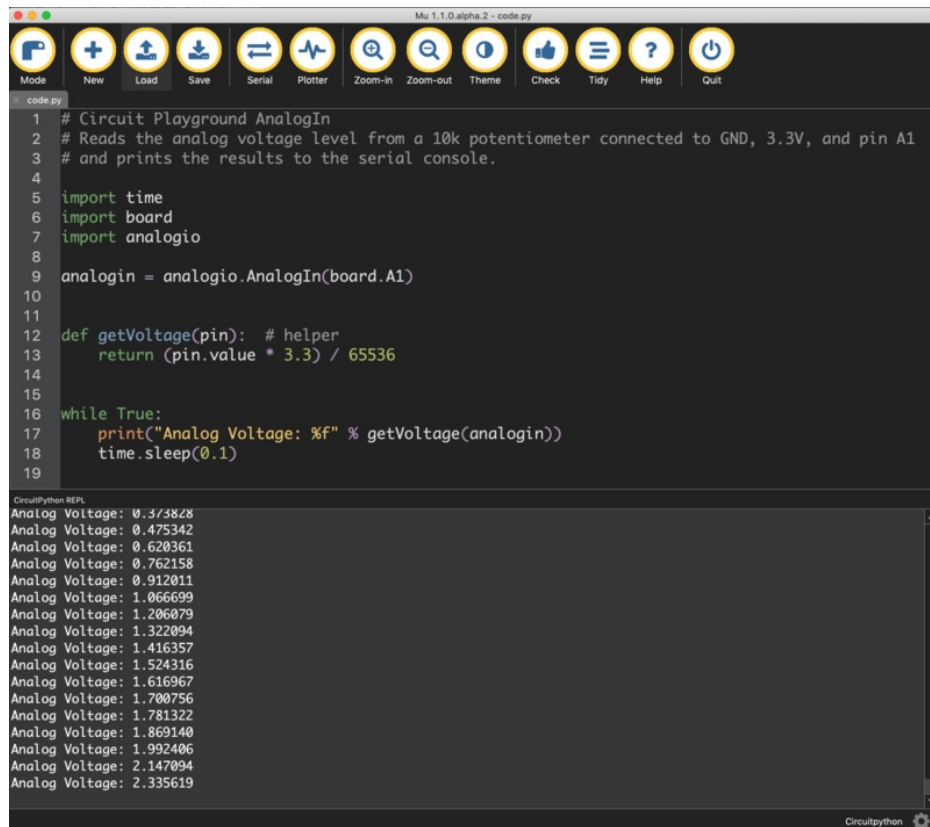
GetVoltage Helper

`getVoltage(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 will just print out the voltage as a floating point value (the `%f` indicates to print as floating point) by calling `getVoltage` on each of our analog object, in this case the potentiometer.

If you connect to the serial console, you'll see the voltage printed out. Try turning the knob of the potentiometer to see the voltage change!



The screenshot shows the Mu Python IDE interface. The top toolbar contains icons for Mode, New, Load, Save, Serial, Plotter, Zoom-in, Zoom-out, Theme, Check, Tidy, Help, and Quit. The code editor displays a Python script for reading an analog voltage level from a 10k potentiometer connected to GND, 3.3V, and pin A1. The script imports time, board, and analogio modules, initializes an analog input on pin A1, and defines a helper function getVoltage(pin) to convert the raw value to a voltage. A while loop prints the voltage every 0.1 seconds. The serial console at the bottom shows the output of the program, displaying a series of analog voltage readings from 0.373828V to 2.335619V.

```
1 # Circuit Playground AnalogIn
2 # Reads the analog voltage level from a 10k potentiometer connected to GND, 3.3V, and pin A1
3 # and prints the results to the serial console.
4
5 import time
6 import board
7 import analogio
8
9 analogin = analogio.AnalogIn(board.A1)
10
11
12 def getVoltage(pin): # helper
13     return (pin.value * 3.3) / 65536
14
15
16 while True:
17     print("Analog Voltage: %f" % getVoltage(analogin))
18     time.sleep(0.1)
19
```

CircuitPython REPL

```
Analog Voltage: 0.373828
Analog Voltage: 0.475342
Analog Voltage: 0.620361
Analog Voltage: 0.762158
Analog Voltage: 0.912011
Analog Voltage: 1.066699
Analog Voltage: 1.206079
Analog Voltage: 1.322094
Analog Voltage: 1.416357
Analog Voltage: 1.524316
Analog Voltage: 1.616967
Analog Voltage: 1.700756
Analog Voltage: 1.781322
Analog Voltage: 1.869140
Analog Voltage: 1.992406
Analog Voltage: 2.147094
Analog Voltage: 2.335619
```

Circuitpython

CircuitPython Servo

In order to use servos, we take advantage of `pulseio`. Now, in theory, you could just use the raw `pulseio` 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 \(https://adafru.it/zdx\)](https://adafru.it/zdx)! If you need help installing the library, check out the [CircuitPython Libraries page \(https://adafru.it/ABU\)](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



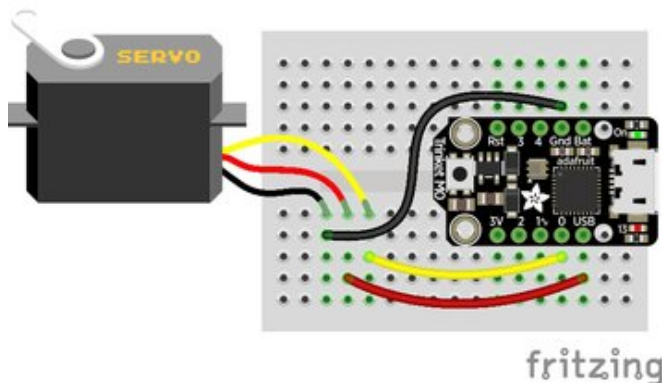
Servos will only work on PWM-capable pins! Check your board details to verify which pins have PWM outputs.

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.

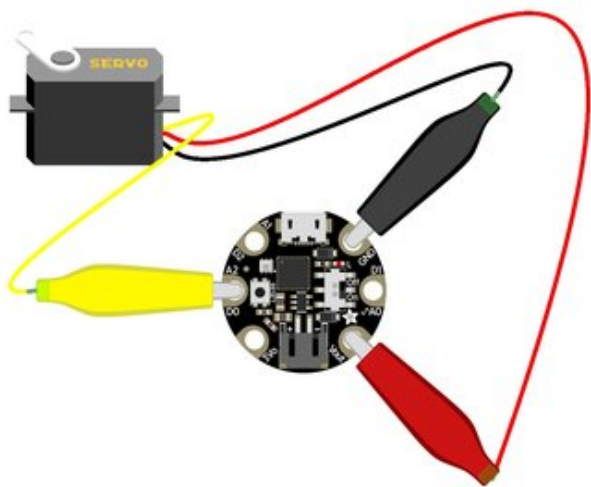
Connect the servo's **red** power wire to 5V power, USB power is good for a servo or two. For more than that, you'll need an external battery pack. Do not use 3.3V for powering a servo!

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.

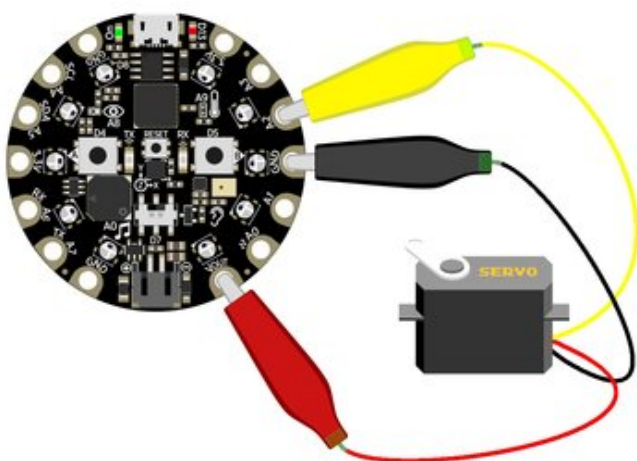


For example, to wire a servo to **Trinket**, connect the ground wire to **GND**, the power wire to **USB**, and the signal wire to **0**.

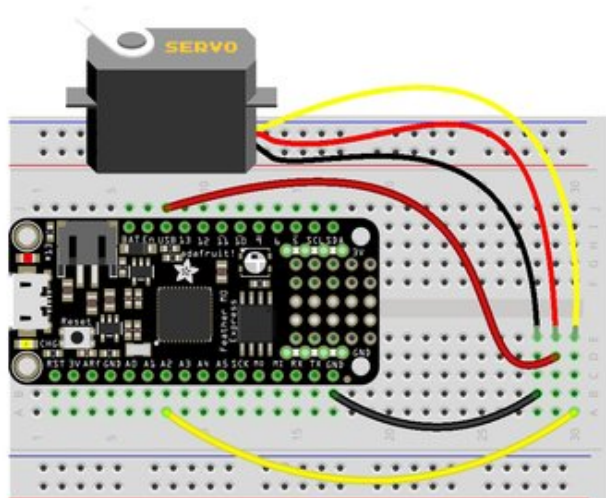
Remember, **A2** on Trinket is labeled "**0**".



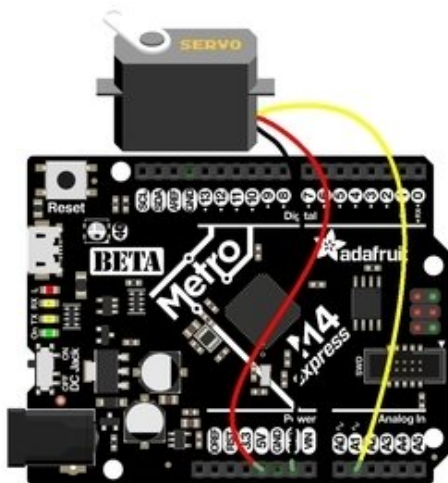
For **Gemma**, use jumper wire alligator clips to connect the ground wire to **GND**, the power wire to **VOUT**, and the signal wire to **A2**.



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 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 **A1**.

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:

```
import time
import board
import pulseio
from adafruit_motor import servo

# create a PWMOut object on Pin A2.
pwm = pulseio.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.


```

# Continuous Servo Test Program for CircuitPython
import time
import board
import pulseio
from adafruit_motor import servo

# create a PWMOut object on Pin A2.
pwm = pulseio.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://adafruit.com/guides/circuitpython/servo) (<https://adafruit.com/guides/circuitpython/servo>)!

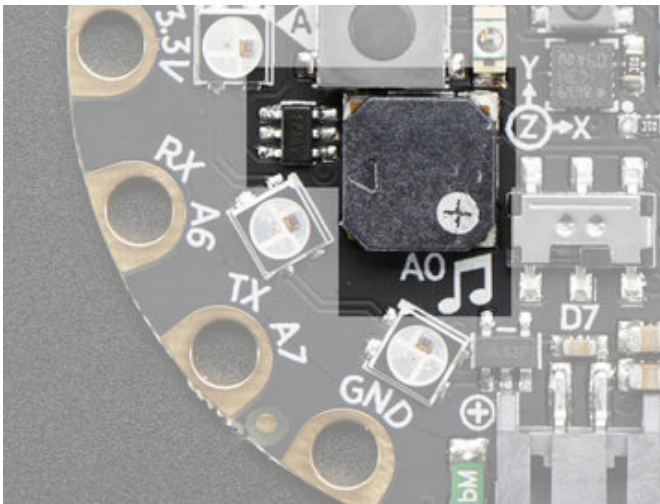
CircuitPython Audio Out



Though the following example uses the Circuit Playground Express to demonstrate, the code works exactly the same way with the Circuit Playground Bluefruit. Simply copy the code and follow along with your Circuit Playground Bluefruit!

The Circuit Playground Express has some nice built in audio output capabilities.

There are **two** ways to get audio output, one is via the small built in speaker. The other is by using alligator clips to connect a headphone or powered speaker to the **A0/AUDIO** pin.

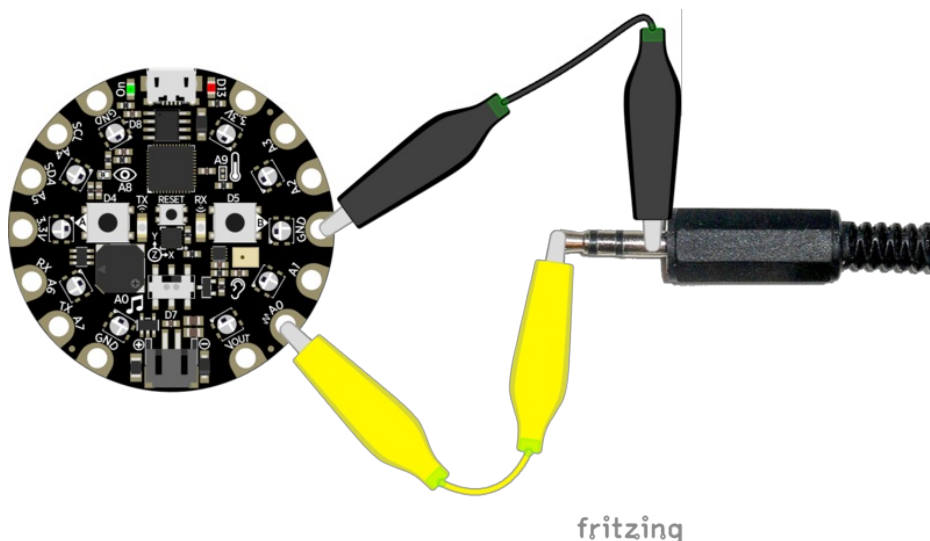


The speaker is over here, its small but can make some loud sounds! You can **ENABLE** or disable the speaker. If you disable the speaker, audio will only come out the **A0/AUDIO** pin. If you enable the speaker, audio will come out from both!

If you want to connect a speaker or headphones, use two alligator clips and connect **GND** to the sleeve of the headphone, and **A0/AUDIO** to the tip.



The A0/AUDIO pin cannot drive a speaker directly, please only connect headphones, or powered speakers!



Basic Tones

We can start by making simple tones. We will play sine waves. We first generate a single period of a sine wave in python, with the `math.sin` function, and stick it into `sine_wave` .

Then we enable the speaker by setting the `SPEAKER_ENABLE` pin to be an output and `True` .

We can create the audio object with this line that sets the output pin and the sine wave sample object and give it the sample array

```
audio = AudioOut(board.SPEAKER)
sine_wave_sample = RawSample(sine_wave)
```

Finally you can run `audio.play()` - if you only want to play the sample once, call as is. If you want it to *loop* the sample, which we definitely do so its one long tone, pass in `loop=True`

You can then do whatever you like, the tone will play in the background until you call `audio.stop()`

```

import time
import array
import math
import board
import digitalio

try:
    from audiocore import RawSample
except ImportError:
    from audioio 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!

FREQUENCY = 440 # 440 Hz middle 'A'
SAMPLERATE = 8000 # 8000 samples/second, recommended!

# Generate one period of sine wav.
length = SAMPLERATE // FREQUENCY
sine_wave = array.array("H", [0] * length)
for i in range(length):
    sine_wave[i] = int(math.sin(math.pi * 2 * i / 18) * (2 ** 15) + 2 ** 15)

# Enable the speaker
speaker_enable = digitalio.DigitalInOut(board.SPEAKER_ENABLE)
speaker_enable.direction = digitalio.Direction.OUTPUT
speaker_enable.value = True

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

# A single sine wave sample is hundredths of a second long. If you set loop=False, it will play
# a single instance of the sample (a quick burst of sound) and then silence for the rest of the
# duration of the time.sleep(). If loop=True, it will play the single instance of the sample
# continuously for the duration of the time.sleep().
audio.play(sine_wave_sample, loop=True) # Play the single sine_wave sample continuously...
time.sleep(1) # for the duration of the sleep (in seconds)
audio.stop() # and then stop.

```

Playing Audio Files

Tones are lovely but lets play some music! You can drag-and-drop audio files onto the **CIRCUITPY** drive and then play them with a Python command

Here's the two files we're going to play:

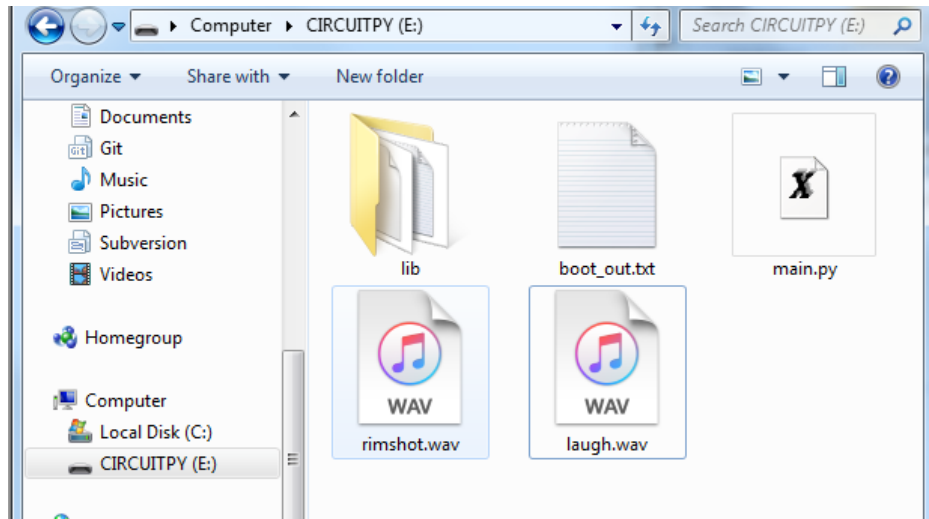
<https://adafru.it/zFK>

<https://adafru.it/zFK>

<https://adafru.it/zFL>

<https://adafru.it/zFL>

Click the green buttons to download the wave files, and save them onto your **CIRCUITPY** drive, alongside your **code.py** and **lib** files



This is the example code we'll be using

```

import board
import digitalio

try:
    from audiocore import WaveFile
except ImportError:
    from audioio 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!

# Enable the speaker
spkrenable = digitalio.DigitalInOut(board.SPEAKER_ENABLE)
spkrenable.direction = digitalio.Direction.OUTPUT
spkrenable.value = True

# Make the 2 input buttons
buttonA = digitalio.DigitalInOut(board.BUTTON_A)
buttonA.direction = digitalio.Direction.INPUT
buttonA.pull = digitalio.Pull.DOWN

buttonB = digitalio.DigitalInOut(board.BUTTON_B)
buttonB.direction = digitalio.Direction.INPUT
buttonB.pull = digitalio.Pull.DOWN

# The two files assigned to buttons A & B
audiofiles = ["rimshot.wav", "laugh.wav"]

def play_file(filename):
    print("Playing file: " + filename)
    wave_file = open(filename, "rb")
    with WaveFile(wave_file) as wave:
        with AudioOut(board.SPEAKER) as audio:
            audio.play(wave)
            while audio.playing:
                pass
    print("Finished")

while True:
    if buttonA.value:
        play_file(audiofiles[0])
    if buttonB.value:
        play_file(audiofiles[1])

```

This example creates two input buttons using the onboard buttons, then has a helper function that will:

1. open a file on the disk drive with `wave_file = open(filename, "rb")`
2. create the wave file object with `with WaveFile(wave_file) as wave:`
3. create the audio playback object with `with AudioOut(board.SPEAKER) as audio:`
4. and finally play it until its done:

```
audio.play(wave)
while audio.playing:
    pass
```

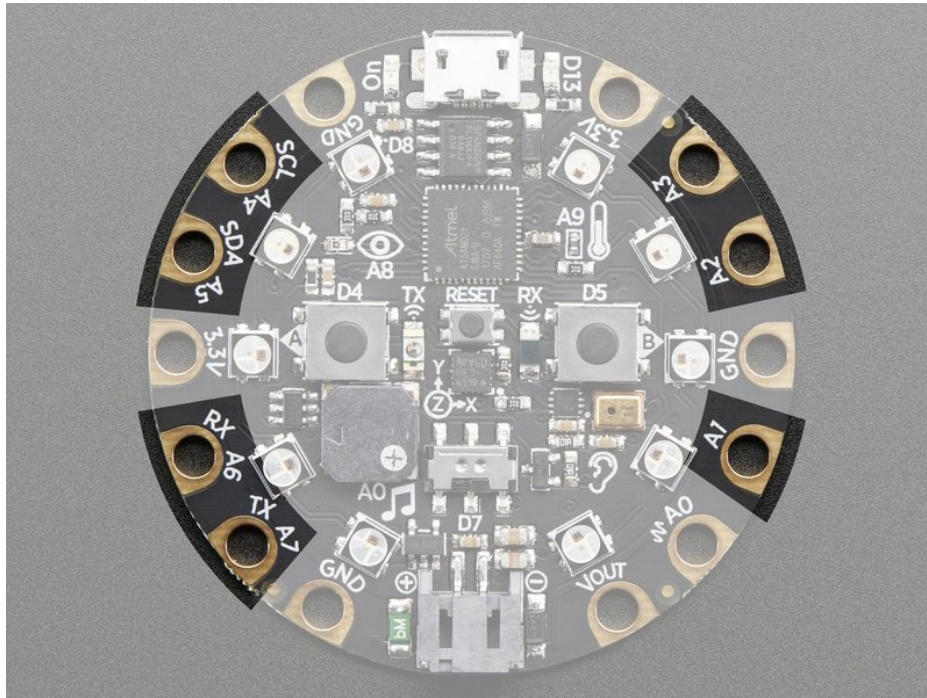
Upload the code then try pressing the two buttons one at a time to create your own laugh track!

If you want to use your own sound files, you can! Record, sample, remix, or simply download files from a sound file sight, such as freesample.org. Then, to make sure you have the files converted to the proper specifications, [check out this guide here \(https://adafru.it/BvU\)](https://adafru.it/BvU) that'll show you how! Spoiler alert: you'll need to make a small, 22Khz (or lower), 16 bit PCM, mono .wav file!

CircuitPython Cap Touch



Though the following example uses the Circuit Playground Express to demonstrate, the code works exactly the same way with the Circuit Playground Bluefruit. Simply copy the code and follow along with your Circuit Playground Bluefruit!



This quick-start example shows how you can read the capacitive touch sensors built into on seven of the Circuit Playground Express pads (pad **A0/Audio** is not a capacitive touch pad).

Copy and paste the code block into `code.py` using your favorite code editor, and save the file, to run the demo

```
# Circuit Playground Capacitive Touch

import time
import board
import touchio

touch_A1 = touchio.TouchIn(board.A1)
touch_A2 = touchio.TouchIn(board.A2)
touch_A3 = touchio.TouchIn(board.A3)
touch_A4 = touchio.TouchIn(board.A4)
touch_A5 = touchio.TouchIn(board.A5)
touch_A6 = touchio.TouchIn(board.A6)
touch_TX = touchio.TouchIn(board.TX)

while True:
    if touch_A1.value:
        print("A1 touched!")
    if touch_A2.value:
        print("A2 touched!")
    if touch_A3.value:
        print("A3 touched!")
    if touch_A4.value:
        print("A4 touched!")
    if touch_A5.value:
        print("A5 touched!")
    if touch_A6.value:
        print("A6 touched!")
    if touch_TX.value:
        print("TX touched!")

    time.sleep(0.01)
```

You can open up the serial console, then touch each touch pad to see the touches detected and printed out.

Creating an capacitive touch input

Pads **A1 - A6** and **TX** can be used as capacitive TouchIn devices:

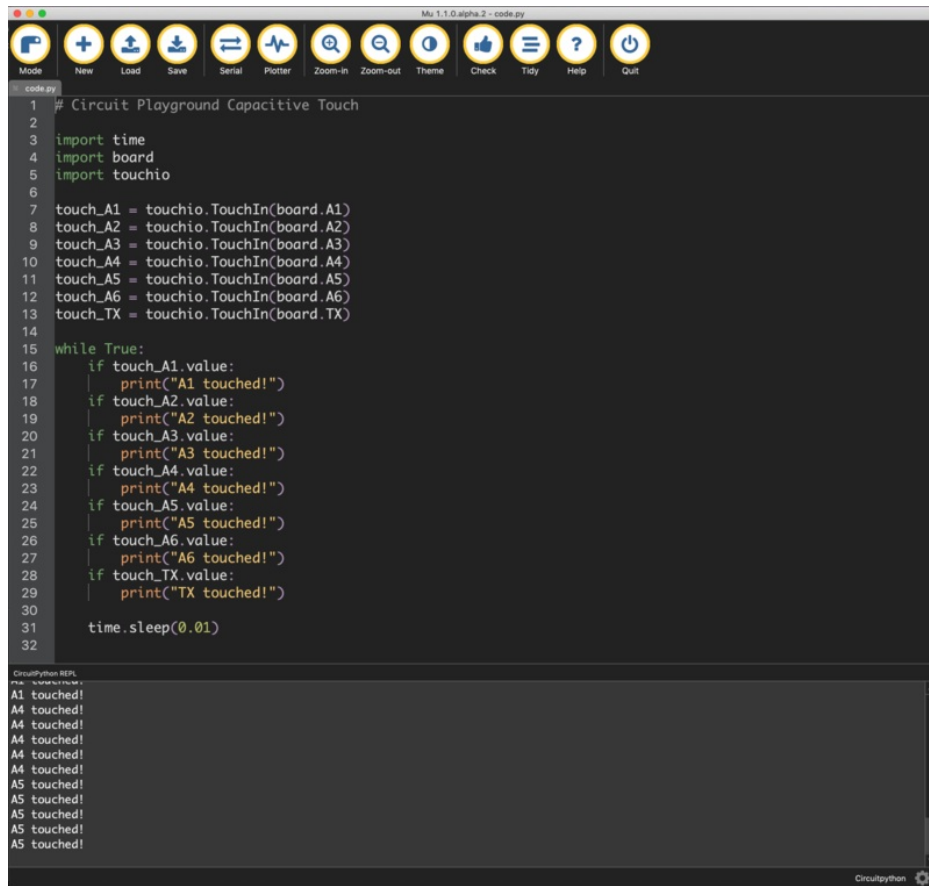
```
touch_A1 = touchio.TouchIn(board.A1)
touch_A2 = touchio.TouchIn(board.A2)
touch_A3 = touchio.TouchIn(board.A3)
touch_A4 = touchio.TouchIn(board.A4)
touch_A5 = touchio.TouchIn(board.A5)
touch_A6 = touchio.TouchIn(board.A6)
touch_TX = touchio.TouchIn(board.TX)
```

This code creates seven objects, one connected to each of the cap touch pads.

Main Loop

The main loop checks each sensor one after the other, to determine if it has been touched. If `touch_A1.value` returns True, that means that that pad, **A1**, detected a touch. For each pad, if it has been touched, a message will print.

A small sleep delay is added at the end so the loop doesn't run *too* fast. You may want to change the delay from 0.1 seconds to 0 seconds to slow it down or increase it to speed it up.



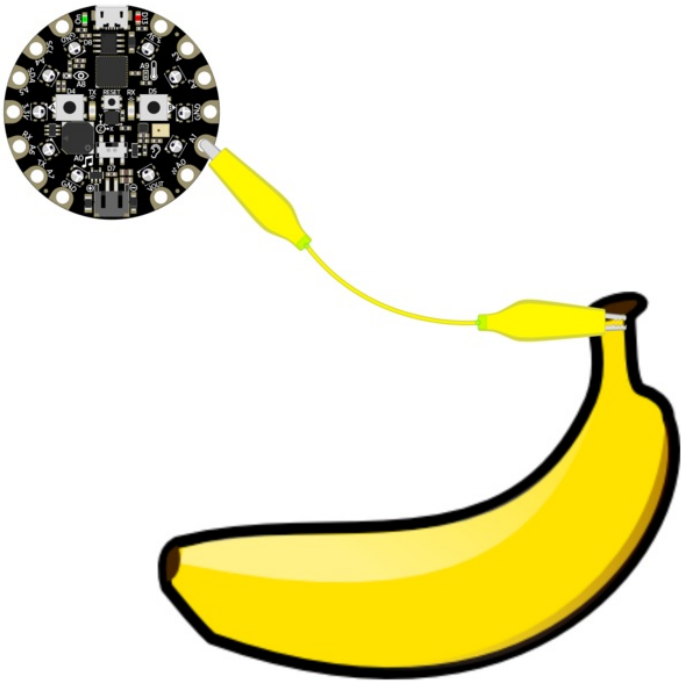
The screenshot shows the Circuit Playground software interface. The top toolbar includes icons for Mode, New, Load, Save, Serial, Plotter, Zoom-in, Zoom-out, Theme, Check, Tidy, Help, and Quit. The main code editor displays the following Python code:

```
1 # Circuit Playground Capacitive Touch
2
3 import time
4 import board
5 import touchio
6
7 touch_A1 = touchio.TouchIn(board.A1)
8 touch_A2 = touchio.TouchIn(board.A2)
9 touch_A3 = touchio.TouchIn(board.A3)
10 touch_A4 = touchio.TouchIn(board.A4)
11 touch_A5 = touchio.TouchIn(board.A5)
12 touch_A6 = touchio.TouchIn(board.A6)
13 touch_TX = touchio.TouchIn(board.TX)
14
15 while True:
16     if touch_A1.value:
17         print("A1 touched!")
18     if touch_A2.value:
19         print("A2 touched!")
20     if touch_A3.value:
21         print("A3 touched!")
22     if touch_A4.value:
23         print("A4 touched!")
24     if touch_A5.value:
25         print("A5 touched!")
26     if touch_A6.value:
27         print("A6 touched!")
28     if touch_TX.value:
29         print("TX touched!")
30
31     time.sleep(0.01)
32
```

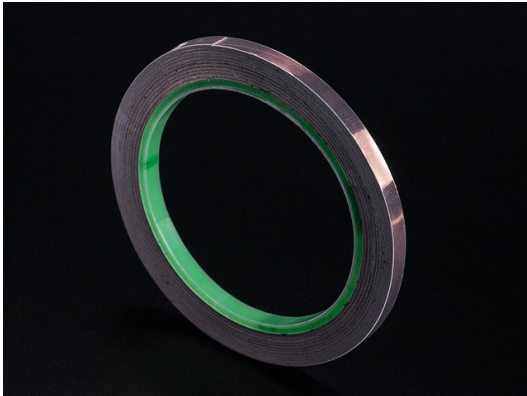
The bottom REPL window shows the output of the code:

```
CircuitPython REPL
>
A1 touched!
A4 touched!
A4 touched!
A4 touched!
A4 touched!
A4 touched!
A5 touched!
A5 touched!
A5 touched!
A5 touched!
A5 touched!
```

Note that no extra hardware is required, you can touch the pads directly, but you may want to attach alligator clips or foil tape to metallic or conductive objects. Try silverware, fruit or other food, liquid, aluminum foil, and items around your desk!



You may need to restart your code/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-signals or not enough, hit that reset button!



Copper Foil Tape with Conductive Adhesive - 6mm x 15 meter roll

OUT OF STOCK

Out Of Stock



Copper Foil Tape with Conductive Adhesive - 25mm x 15 meter roll

\$19.95
IN STOCK

Add To Cart



Small Alligator Clip Test Lead (set of 12)

\$3.95
IN STOCK

Add To Cart

Capacitive Touch and the Audio Pin on Circuit Playground Bluefruit

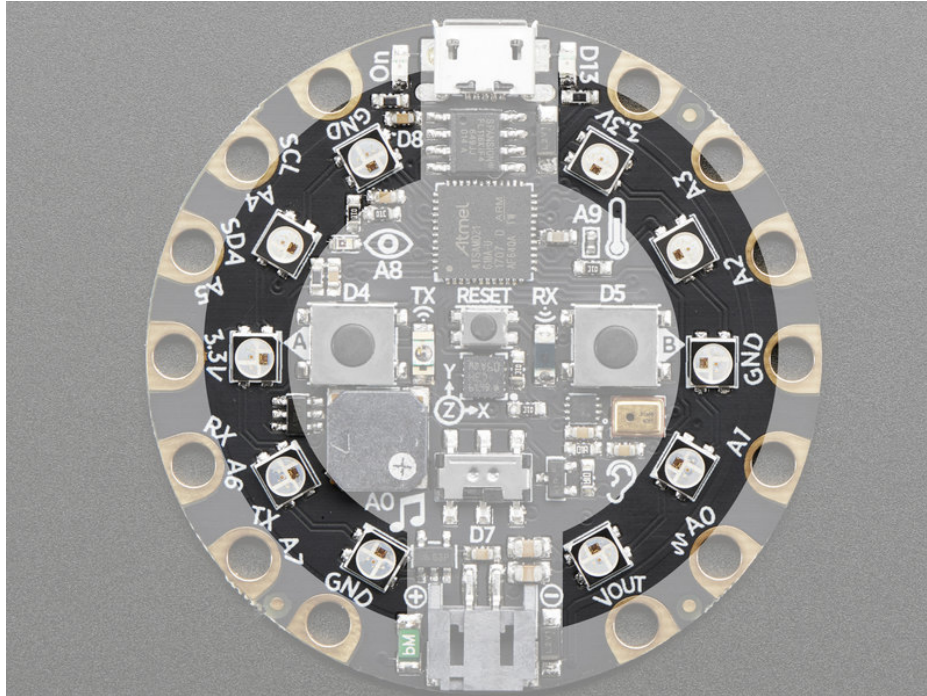
On the Circuit Playground Bluefruit, if you touch any of the touch pads at the same time as touching the Audio pin, you may hear a clicking or buzzing coming from the speaker. This is due to how the capacitive touch on the Bluefruit works. If you run into this and wish to avoid it, you can turn the speaker off using code by including the following in your **code.py**:

```
import digitalio

speaker = digitalio.DigitalInOut(board.SPEAKER_ENABLE)
speaker.switch_to_output(value=False)
```



Though the following example uses the Circuit Playground Express to demonstrate, the code works exactly the same way with the Circuit Playground Bluefruit. Simply copy the code and follow along with your Circuit Playground Bluefruit!



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 pixels with brightness control (e.g. setting `brightness=0.2` to set it to 20% brightness) and 1000 pixels without (e.g. not setting `brightness` at all or setting `brightness=1.0` in object creation). That's because to adjust the brightness we have to dynamically re-create the datastream each write.

Here's an example with a lot of different visual effects you can check out. You'll need the **neopixel.mpy** library file if you don't have it yet! (<https://adafru.it/ENC>)

```
# Circuit Playground NeoPixel
import time
import board
import neopixel

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10, brightness=0.2, auto_write=False)

# choose which demos to play
# 1 means play, 0 means don't!
color_chase_demo = 1
flash_demo = 1
rainbow_demo = 1
rainbow_cycle_demo = 1
```



```

def wheel(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)
    if pos < 85:
        return (255 - pos * 3, pos * 3, 0)
    if pos < 170:
        pos -= 85
        return (0, 255 - pos * 3, pos * 3)
    pos -= 170
    return (pos * 3, 0, 255 - pos * 3)

def color_chase(color, wait):
    for i in range(10):
        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(10):
            rc_index = (i * 256 // 10) + j * 5
            pixels[i] = wheel(rc_index & 255)
        pixels.show()
        time.sleep(wait)

def rainbow(wait):
    for j in range(255):
        for i in range(len(pixels)):
            idx = int(i + j)
            pixels[i] = wheel(idx & 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)
WHITE = (255, 255, 255)
OFF = (0, 0, 0)

while True:
    if color_chase_demo:
        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)
        color_chase(OFF, 0.1)

```



```

    color_chase(0.01, 0.1)

if flash_demo:
    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)
    pixels.fill(WHITE)
    pixels.show()
    time.sleep(1)

if rainbow_cycle_demo:
    rainbow_cycle(0.05) # Increase the number to slow down the rainbow.

if rainbow_demo:
    rainbow(0.05) # Increase the number to slow down the rainbow.

```

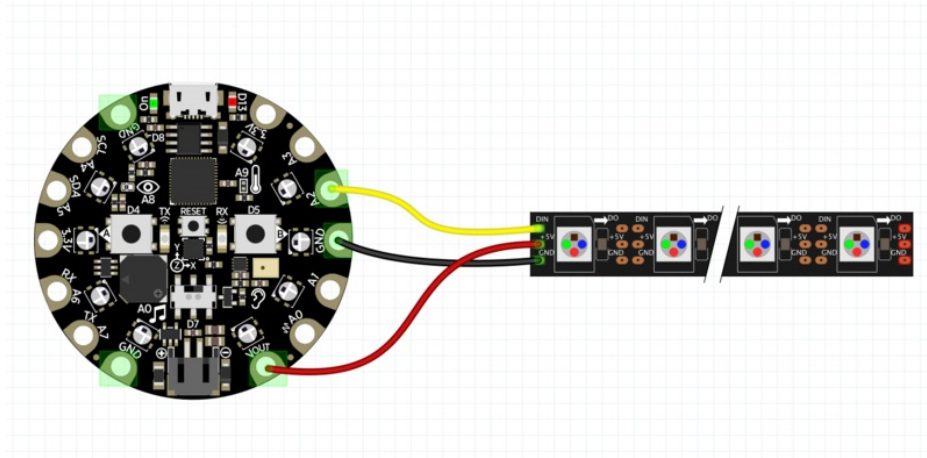
The NeoPixel object's argument list requires the pin you'll use (any pin can be used) and the number of pixels. There are two optional arguments, `brightness` (range from 0 off to 1.0 full brightness) and `auto_write`. `auto_write` defaults to `True` when not set. When `auto_write` is set to `True`, every change is immediately written to the strip of pixels, which is easier to use but *way* slower. if you set `auto_write=False` then you will have to call `pixels.show()` when you want to actually write color data out.

You can easily set colors by indexing into the location `pixels[n] = (red, green, blue)`. For example, `pixels[0] = (100, 0, 0)` will set the first pixel to a medium-brightness red, and `pixels[2] = (0, 255, 0)` will set the third pixel to bright green. Then, if you have `auto_write=False` don't forget to call `pixels.show()` !



You aren't limited to the on-board NeoPixels -- externally connected NeoPixels can be driven by any Digital IO pin.

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 the power source. On the Circuit Playground Express this is the **Vout** pad - that pad has direct power from USB or BAT (battery), depending on which is higher voltage.



Verify the wiring on your strip or device - plugging into the 'DOUT' side is a common mistake! Wire up NeoPixels only while the Circuit Playground Express is not on, to avoid possible damage!

If the power to the NeoPixels is $> 5.5\text{V}$ you may have some difficulty driving some strips, in which case you may need to lower the voltage to $4.5\text{--}5\text{V}$ or use a level shifter



We have a ton more information on general purpose NeoPixel know-how at our NeoPixel UberGuide <https://learn.adafruit.com/adafruit-neopixel-uberguide>

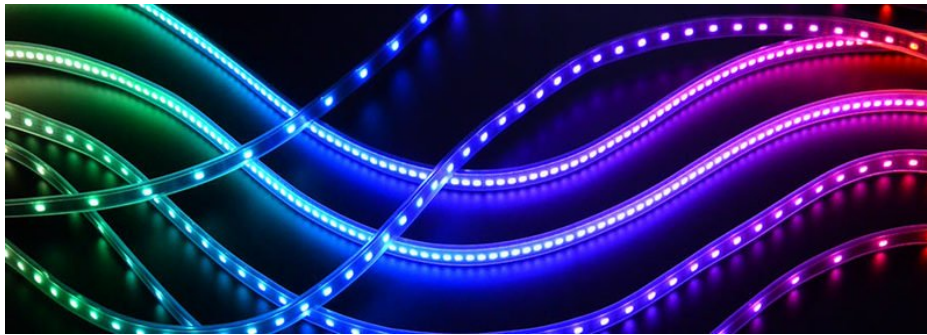
CircuitPython DotStar

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 \(https://adafru.it/y8E\)](https://adafru.it/y8E). If you need help installing the library, check out the [CircuitPython Libraries page \(https://adafru.it/ABU\)](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!

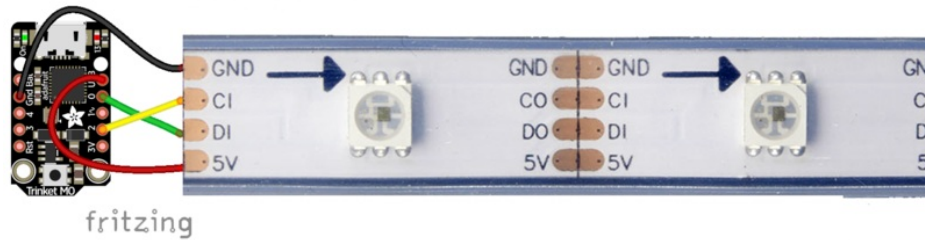
For powering the pixels from the board, the 3.3V regulator output can handle about 500mA peak which is about 50 pixels with 'average' use. If you want really bright lights and a lot of pixels, we recommend powering direct from an external power source.

- 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.

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. Copy and paste the code into **code.py** using your favorite editor, and save the file.

```
# CircuitPython demo - Dotstar
import time
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 wheel(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)
    if pos < 85:
        return (255 - pos * 3, pos * 3, 0)
    if pos < 170:
        pos -= 85
        return (0, 255 - pos * 3, pos * 3)
    pos -= 170
    return (pos * 3, 0, 255 - pos * 3)

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()
```

```

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] = wheel(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 \(https://adafru.it/Bel\)](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 \(https://adafru.it/Bel\)](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!


```
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.

You'll need the `adafruit_bus_device` library folder if you don't already have it in your `/lib` folder! You can get it from the [CircuitPython Library Bundle \(https://adafru.it/y8E\)](https://adafru.it/y8E). If you need help installing the library, check out the [CircuitPython Libraries page \(https://adafru.it/ABU\)](https://adafru.it/ABU).

Copy and paste the code into `code.py` using your favorite editor, and save the file.

```
# CircuitPython Demo - USB/Serial echo

import board
import busio
import digitalio

led = digitalio.DigitalInOut(board.D13)
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
```

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.

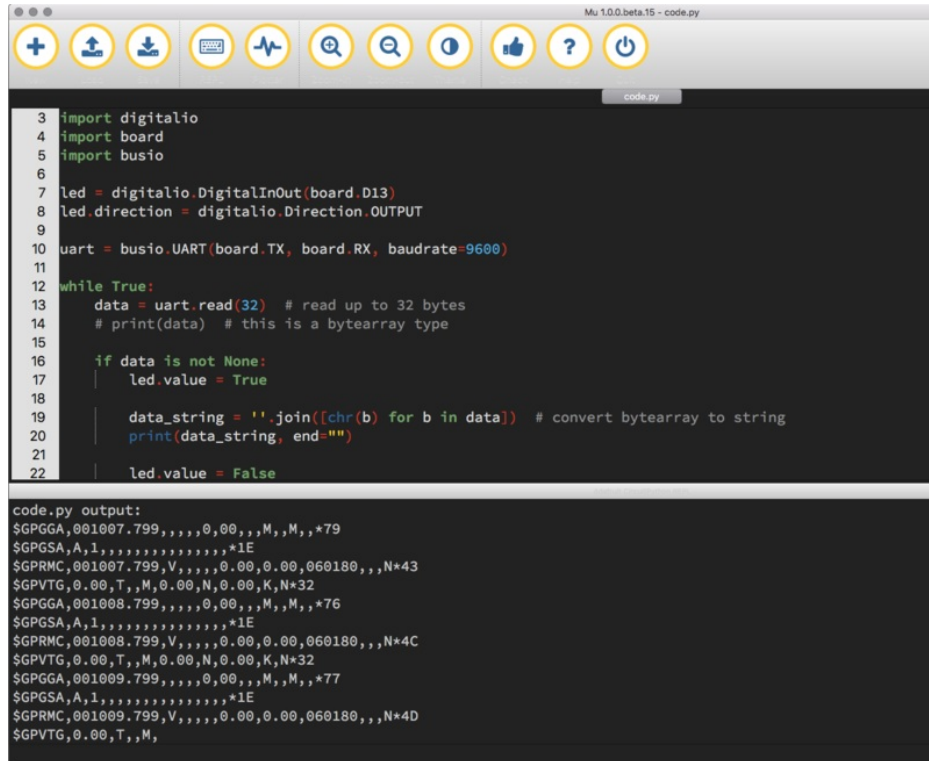
Once the object is created you read data in with `read(numbytes)` where you can specify the max number of bytes. It will return a byte array type object if anything was received already. Note it will always return immediately because there is an internal buffer! So read as much data as you can 'digest'.

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:



The screenshot shows the Mu Python IDE interface. The code editor contains the following Python code:

```
3 import digitalio
4 import board
5 import busio
6
7 led = digitalio.DigitalInOut(board.D13)
8 led.direction = digitalio.Direction.OUTPUT
9
10 uart = busio.UART(board.TX, board.RX, baudrate=9600)
11
12 while True:
13     data = uart.read(32) # read up to 32 bytes
14     # print(data) # this is a bytearray type
15
16     if data is not None:
17         led.value = True
18
19         data_string = ''.join([chr(b) for b in data]) # convert bytearray to string
20         print(data_string, end="")
21
22     led.value = False
```

The output window below the code editor shows the following text:

```
code.py output:
$GPGGA,001007.799,,,0.00,,M,,M,,*79
$GPGSA,A,1,,,,,,,,,,,,,*1E
$GPRMC,001007.799,V,,,0.00,0.00,060180,,,N*43
$GPVTG,0.00,T,,M,0.00,N,0.00,K,N*32
$GPGGA,001008.799,,,0.00,,M,,M,,*76
$GPGSA,A,1,,,,,,,,,,,,,*1E
$GPRMC,001008.799,V,,,0.00,0.00,060180,,,N*4C
$GPVTG,0.00,T,,M,0.00,N,0.00,K,N*32
$GPGGA,001009.799,,,0.00,,M,,M,,*77
$GPGSA,A,1,,,,,,,,,,,,,*1E
$GPRMC,001009.799,V,,,0.00,0.00,060180,,,N*4D
$GPVTG,0.00,T,,M,
```



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 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.

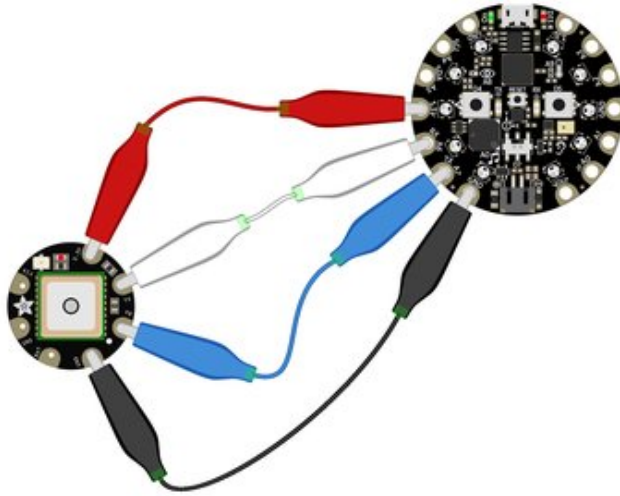
We've included diagrams show you how to connect the GPS to your board. In these diagrams, the wire colors match the same pins on each board.

- 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!

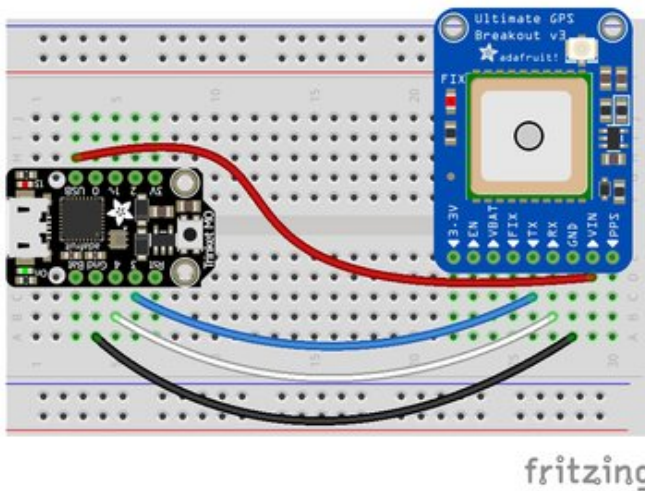


Watch out! A common mixup with UART serial is that RX on one board connects to TX on the other! However, sometimes boards have RX labeled TX and vice versa. So, you'll want to start with RX connected to TX, but if that doesn't work, try the other way around!



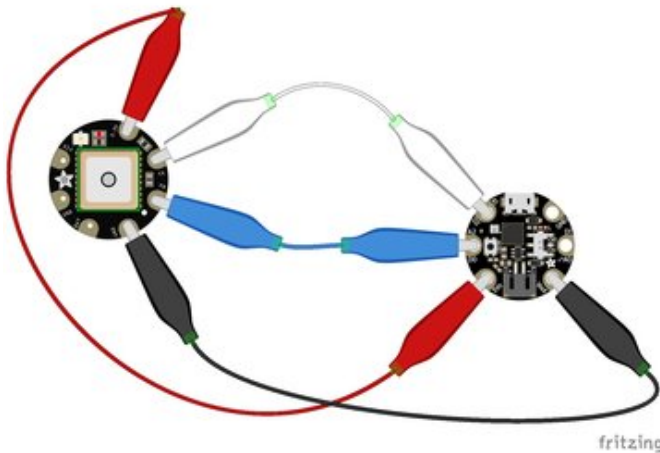
Circuit Playground Express and Circuit Playground Bluefruit

- 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.



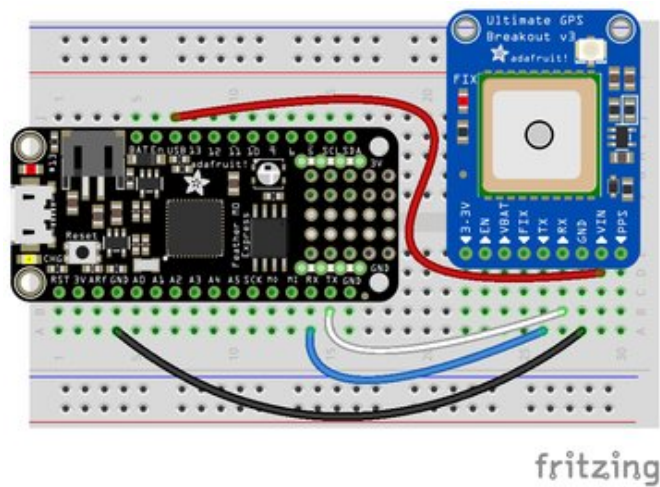
Trinket M0

- 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.



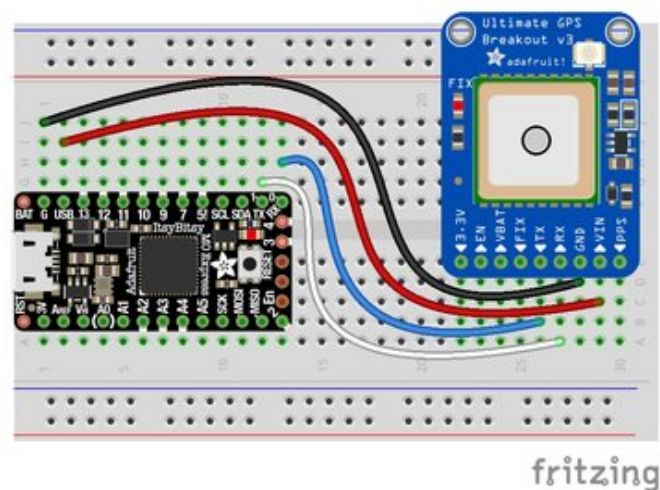
Gemma M0

- Connect **3V** 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.



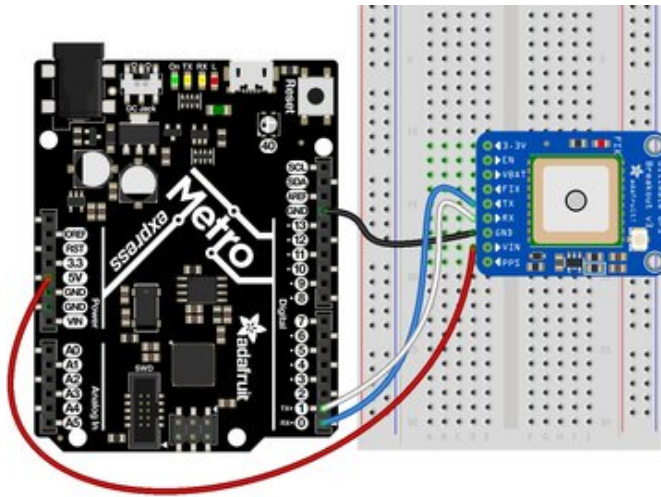
Feather M0 Express and Feather M4 Express

- 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.

All you need to do is copy this file to your board, rename it **code.py**, connect to the serial console and check out the output! The results print out a nice handy list of RX and TX pin pairs that you can use.

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)

```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
RX pin: board.D2      TX pin: board.D0
RX pin: board.D4      TX pin: board.D0
RX pin: board.D3      TX pin: board.D0
RX pin: board.D13     TX pin: board.D0
RX pin: board.D0      TX pin: board.D4
RX pin: board.D2      TX pin: board.D4
RX pin: board.D3      TX pin: board.D4
RX pin: board.D0      TX pin: board.D13
RX pin: board.D2      TX pin: board.D13
RX pin: board.D3      TX pin: board.D13
```



```

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
        else:
            if is_hardware_uart(tx_pin, rx_pin):
                print("RX pin:", rx_pin, "\t TX pin:", tx_pin)
            else:
                pass

```

Trinket M0: Create UART before I2C

On the Trinket M0 (only), if you are using both `busio.UART` and `busio.I2C`, you must create the UART object first, e.g.:

```

>>> import board, busio
>>> uart = busio.UART(board.TX, board.RX)
>>> i2c = busio.I2C(board.SCL, board.SDA)

```

Creating `busio.I2C` first does not work:

```

>>> import board, busio
>>> i2c = busio.I2C(board.SCL, board.SDA)
>>> uart = busio.UART(board.TX, board.RX)
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 \(https://adafru.it/uap\)](https://adafru.it/uap). (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 TSL2561, a common, low-cost light sensor. While the exact code we're running is specific to the TSL2561 the overall process is the same for just about any I2C sensor or device.

You'll need the `adafruit_tsl2561.mpy` library and `adafruit_bus_device` library folder if you don't already have it in your `/lib` folder! You can get it from the [CircuitPython Library Bundle \(https://adafru.it/y8E\)](https://adafru.it/y8E). If you need help installing the library, check out the [CircuitPython Libraries page \(https://adafru.it/ABU\)](https://adafru.it/ABU).

These examples will use the TSL2561 lux sensor Flora and 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 TSL2561 to your board.

For Gemma M0 and Circuit Playground Express, you can use use alligator clips to connect to the Flora TSL2561 Lux Sensor.

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 TSL2561 Lux Sensor breakout board.

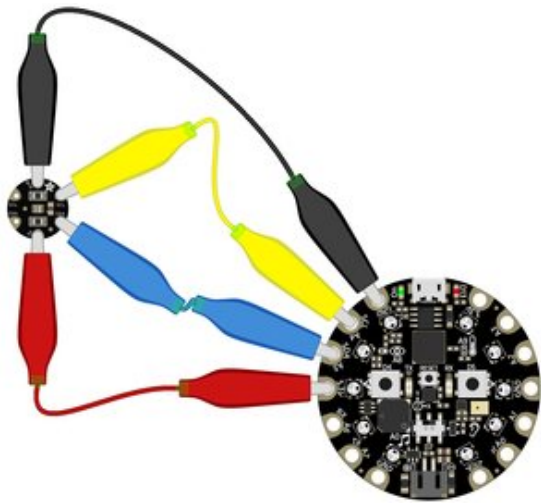
We've included diagrams show you how to connect the TSL2561 to your board. In these diagrams, the wire colors match the same pins on each board.

- The **black** wire connects between the **ground** pins.
- The **red** wire connects between the **power** pins on the TSL2561 and your board.
- The **yellow** wire connects from **SCL** on the TSL2561 to **SCL** on your board.
- The **blue** wire connects from **SDA** on the TSL2561 to **SDA** on your board.

Check out the list below for a diagram of your specific board!

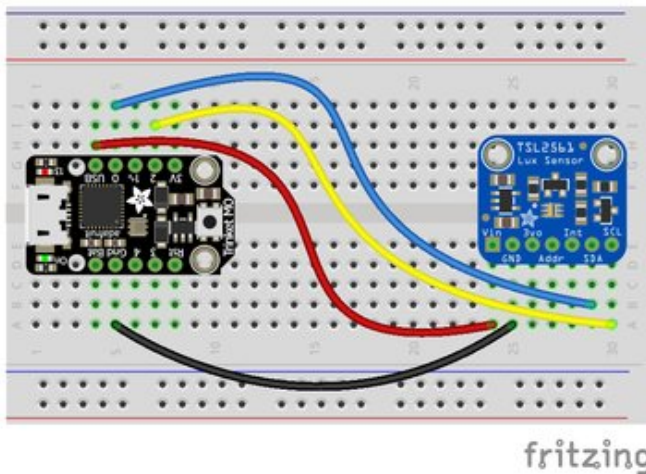


Be aware that the Adafruit microcontroller boards do not have I2C pullup resistors built in! All of the Adafruit breakouts do, but if you're building your own board or using a non-Adafruit breakout, you must add 2.2K-10K ohm pullups on both SDA and SCL to the 3.3V.



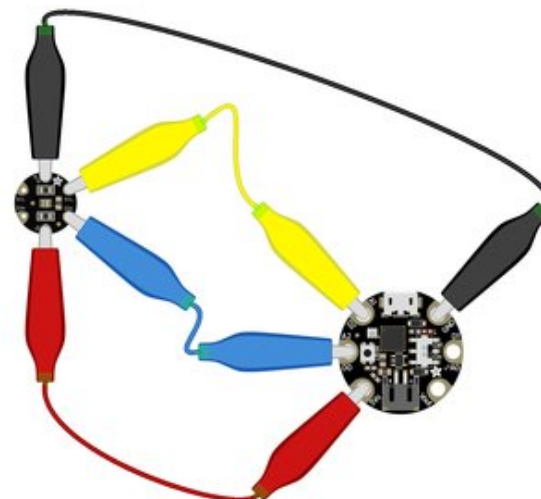
Circuit Playground Express and Circuit Playground Bluefruit

- Connect **3.3v** on your CPX to **3.3v** on your TSL2561.
- Connect **GND** on your CPX to **GND** on your TSL2561.
- Connect **SCL/A4** on your CPX to **SCL** on your TSL2561.
- Connect **SDA/A5** on your CPX to **SDA** on your TSL2561.



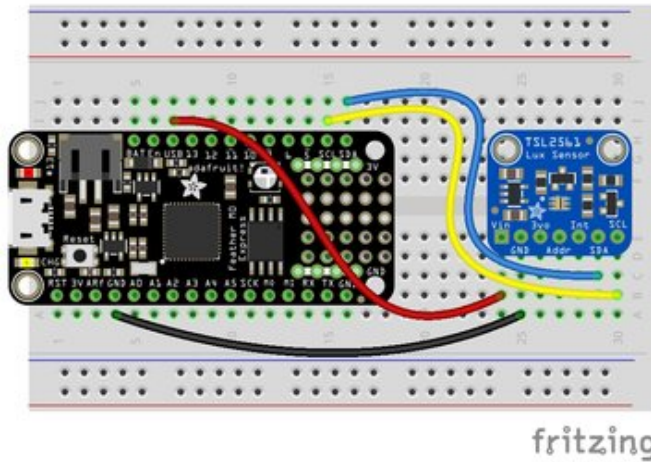
Trinket M0

- Connect **USB** on the Trinket to **VIN** on the TSL2561.
- Connect **Gnd** on the Trinket to **GND** on the TSL2561.
- Connect **D2** on the Trinket to **SCL** on the TSL2561.
- Connect **D0** on the Trinket to **SDA** on the TSL2561.



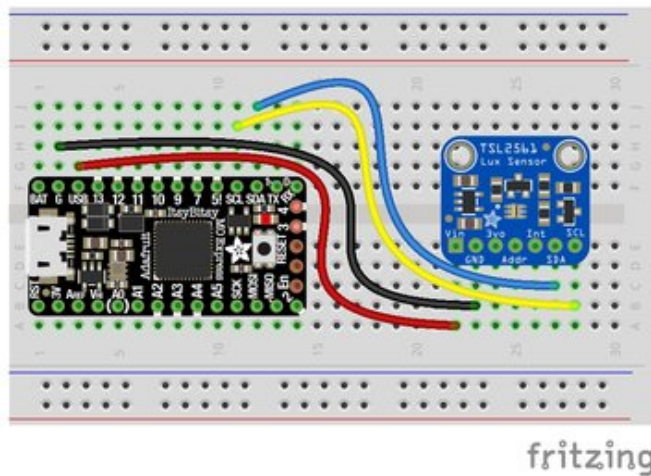
Gemma M0

- Connect **3v0** on the Gemma to **3V** on the TSL2561.
- Connect **GND** on the Gemma to **GND** on the TSL2561.
- Connect **A1/D2** on the Gemma to **SCL** on the TSL2561.
- Connect **A2/D0** on the Gemma to **SDA** on the TSL2561.



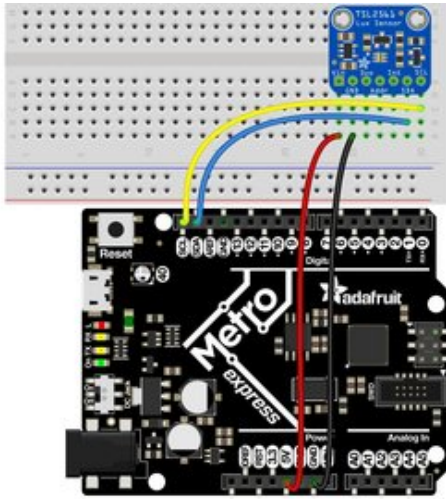
Feather M0 Express and Feather M4 Express

- Connect **USB** on the Feather to **VIN** on the TSL2561.
- Connect **GND** on the Feather to **GND** on the TSL2561.
- Connect **SCL** on the Feather to **SCL** on the TSL2561.
- Connect **SDA** on the Feather to **SDA** on the TSL2561.



ItsyBitsy M0 Express and ItsyBitsy M4 Express

- Connect **USB** on the ItsyBitsy to **VIN** on the TSL2561
- Connect **G** on the ItsyBitsy to **GND** on the TSL2561.
- Connect **SCL** on the ItsyBitsy to **SCL** on the TSL2561.
- Connect **SDA** on the ItsyBitsy to **SDA** on the TSL2561.



Metro M0 Express and Metro M4 Express

- Connect **5V** on the Metro to **VIN** on the TSL2561.
- Connect **GND** on the Metro to **GND** on the TSL2561.
- Connect **SCL** on the Metro to **SCL** on the TSL2561.
- Connect **SDA** on the Metro to **SDA** on the TSL2561.

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.

Copy and paste the code into **code.py** using your favorite editor, and save the file.

```
# CircuitPython demo - I2C scan

import time

import board
import busio

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

while not i2c.try_lock():
    pass

while True:
    print("I2C addresses found:", [hex(device_address)
                                   for device_address in i2c.scan()])
    time.sleep(2)
```

First we create the **i2c** object and provide the I2C pins, **board.SCL** and **board.SDA**.

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 TSL2561 which has a 7-bit I2C address of 0x39. The result for this sensor is `I2C addresses found: ['0x39']`. 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!

Copy and paste the code into **code.py** using your favorite editor, and save the file.

```
# CircuitPython Demo - I2C sensor

import time

import adafruit_tsl2561
import board
import busio

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

# 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()

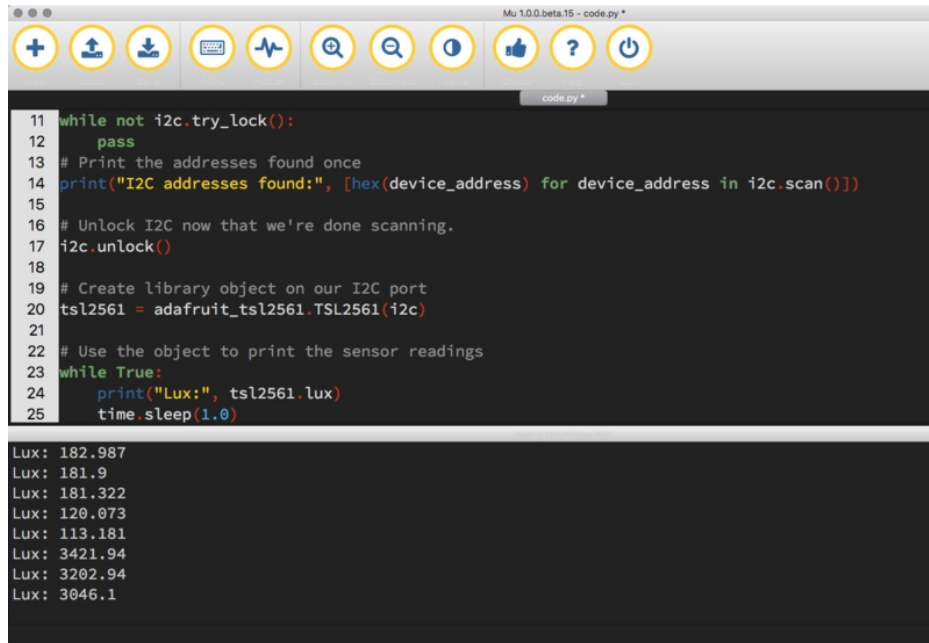
# Create library object on our I2C port
tsl2561 = adafruit_tsl2561.TSL2561(i2c)

# Use the object to print the sensor readings
while True:
    print("Lux:", tsl2561.lux)
    time.sleep(1.0)
```

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 `tsl2561` 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!



```
Mu 1.0.0.beta.15 - code.py *
+ [upload] [download] [terminal] [serial] [help] [info] [like] [question] [power]

code.py *
11 while not i2c.try_lock():
12     pass
13 # Print the addresses found once
14 print("I2C addresses found:", [hex(device_address) for device_address in i2c.scan()])
15
16 # Unlock I2C now that we're done scanning.
17 i2c.unlock()
18
19 # Create library object on our I2C port
20 tsl2561 = adafruit_tsl2561.TSL2561(i2c)
21
22 # Use the object to print the sensor readings
23 while True:
24     print("Lux:", tsl2561.lux)
25     time.sleep(1.0)

Lux: 182.987
Lux: 181.9
Lux: 181.322
Lux: 120.073
Lux: 113.181
Lux: 3421.94
Lux: 3202.94
Lux: 3046.1
```

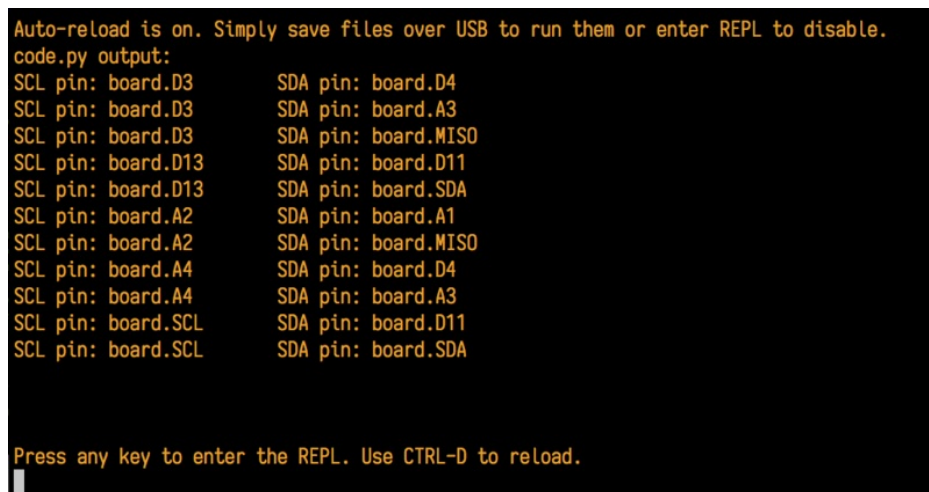
Where's my I2C?

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.

All you need to do is copy this file to your board, rename it **code.py**, connect to the serial console and check out the output! The results print out a nice handy list of SCL and SDA pin pairs that you can use.

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 \(https://adafruit.it/Ben\)](https://adafruit.it/Ben).



```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
SCL pin: board.D3      SDA pin: board.D4
SCL pin: board.D3      SDA pin: board.A3
SCL pin: board.D3      SDA pin: board.MISO
SCL pin: board.D13     SDA pin: board.D11
SCL pin: board.D13     SDA pin: board.SDA
SCL pin: board.A2      SDA pin: board.A1
SCL pin: board.A2      SDA pin: board.MISO
SCL pin: board.A4      SDA pin: board.D4
SCL pin: board.A4      SDA pin: board.A3
SCL pin: board.SCL     SDA pin: board.D11
SCL pin: board.SCL     SDA pin: board.SDA

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

```

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
        else:
            if is_hardware_I2C(scl_pin, sda_pin):
                print("SCL pin:", scl_pin, "\t SDA pin:", sda_pin)
            else:
                pass

```

CircuitPython HID Keyboard



Though the following example uses the Circuit Playground Express to demonstrate, the code works exactly the same way with the Circuit Playground Bluefruit. Simply copy the code and follow along with your Circuit Playground Bluefruit!



One of the things we baked into CircuitPython is 'HID' control - Keyboard and Mouse capabilities. This means a Circuit Playground Express can act like a keyboard device and press keys, or a mouse and have it move the mouse 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!

You'll need to copy the **adafruit_hid** module from the library bundle which include Keyboard, Keycode and Mouse support (<https://adafru.it/ENC>)

Then try running this example code which will set the Circuit Playground Express **Button_A** and **Button_B** as HID keyboard "keys".



This example has been updated for version 4+ of the CircuitPython HID library. On the Circuit Playground Express this library is built into CircuitPython. So, please use the latest version of CircuitPython as well. (At least 5.0.0-beta.3)

```
# Circuit Playground HID Keyboard

import time

import board
import usb_hid
from adafruit_hid.keyboard import Keyboard
from adafruit_hid.keyboard_layout_us import KeyboardLayoutUS
from adafruit_hid.keycode import Keycode
from digitalio import DigitalInOut, Direction, Pull
```



```

# A simple neat keyboard demo in CircuitPython

# The button pins we'll use, each will have an internal pulldown
buttonpins = [board.BUTTON_A, board.BUTTON_B]
# our array of button objects
buttons = []
# The keycode sent for each button, will be paired with a control key
buttonkeys = [Keycode.A, "Hello World!\n"]
controlkey = Keycode.SHIFT

# the keyboard object!
# sleep for a bit to avoid a race condition on some systems
time.sleep(1)
kbd = Keyboard(usb_hid.devices)
# we're americans :)
layout = KeyboardLayoutUS(kbd)

# make all pin objects, make them inputs with pulldowns
for pin in buttonpins:
    button = DigitalInOut(pin)
    button.direction = Direction.INPUT
    button.pull = Pull.DOWN
    buttons.append(button)

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

print("Waiting for button presses")

while True:
    # check each button
    # when pressed, the LED will light up,
    # when released, the keycode or string will be sent
    # this prevents rapid-fire repeats!
    for button in buttons:
        if button.value: # pressed?
            i = buttons.index(button)
            print("Button #%d Pressed" % i)

            # turn on the LED
            led.value = True

            while button.value:
                pass # wait for it to be released!
            # type the keycode or string
            k = buttonkeys[i] # get the corresponding keycode or string
            if isinstance(k, str):
                layout.write(k)
            else:
                kbd.press(controlkey, k) # press...
                kbd.release_all() # release!

            # turn off the LED
            led.value = False

    time.sleep(0.01)

```

Press Button A or Button B to have the keypresses sent.

The Keyboard and Layout object are created, we only have US right now (if you make other layouts please submit a GitHub pull request!)

```
# the keyboard object!
kbd = Keyboard(usb_hid.devices)
# we're americans :)
layout = KeyboardLayoutUS(kbd)
```

Then you can send key-down's with `kbd.press(keycode, ...)` You can have up to 6 keycode presses at once. Note that these are **keycodes** so if you want to send a capital A, you need both SHIFT and A. Don't forget to call `kbd.release_all()` soon after or you'll have a stuck key which is really annoying!

You can also send full strings, with `layout.write("Hello World!\n")` - it will use the layout to determine the keycodes to press.



For more detail check out the documentation at <https://circuitpython.readthedocs.io/projects/hid/en/latest/>

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 \(https://adafru.it/Bec\)](https://adafru.it/Bec), and [enter the REPL \(https://adafru.it/Awz\)](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!

```
Adafruit CircuitPython 2.2.4 on 2018-03-07; Adafruit Metro M0 Express with samd21g18
>>> import microcontroller
>>> microcontroller.cpu.temperature
21.8071
>>> 
```

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!

```
>>> microcontroller.cpu.temperature * (9 / 5) + 32
70.8655
>>> 
```



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 Storage

CircuitPython boards show up as as USB drive, allowing you to edit code directly on the board. You've been doing this for a while. By now, maybe you've wondered, "Can I write data *from CircuitPython* to the storage drive to act as a datalogger?" The answer is **yes**!

However, it is a little tricky. You need to add some special code to **boot.py**, not just **code.py**. That's because you have to set the filesystem to be read-only when you need to edit code to the disk from your computer, and set it to writeable when you want the CircuitPython core to be able to write.



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!)

The following is your new **boot.py**. Copy and paste the code into **boot.py** using your favorite editor. You may need to create a new 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("/", switch.value)
```

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.



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.

The following is your new **code.py**. Copy and paste the code into **code.py** using your favorite editor.

```

import time

import board
import digitalio
import microcontroller

led = digitalio.DigitalInOut(board.D13)
led.switch_to_output()

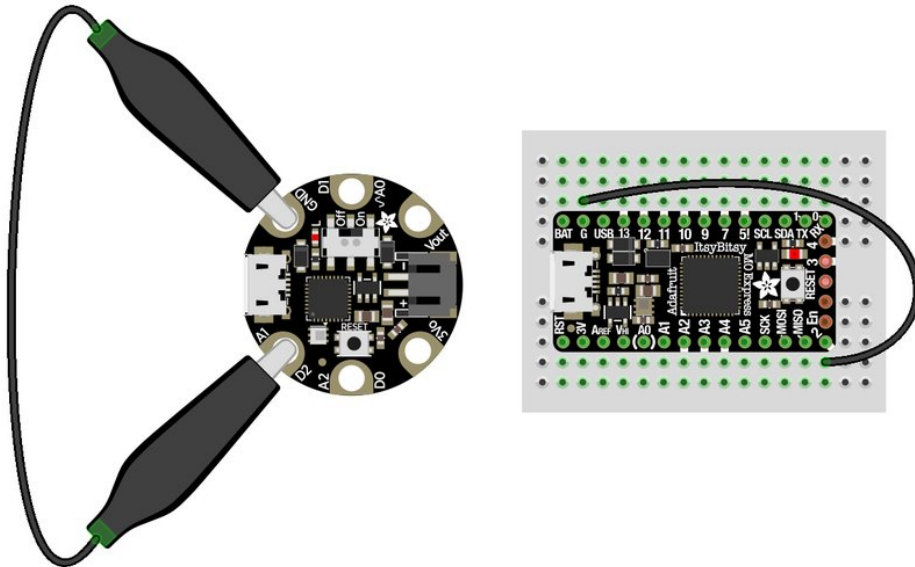
try:
    with open("/temperature.txt", "a") as fp:
        while True:
            temp = microcontroller.cpu.temperature
            # do the C-to-F conversion here if you would like
            fp.write('{0:f}\n'.format(temp))
            fp.flush()
            led.value = not led.value
            time.sleep(1)
except OSError as e:
    delay = 0.5
    if e.args[0] == 28:
        delay = 0.25
    while True:
        led.value = not led.value
        time.sleep(delay)

```

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 \(https://adafruit.it/Bes\)](https://adafruit.it/Bes). Instead of wiring up a switch, however, you'll be connecting the pin directly to ground with alligator clips or jumper wires.



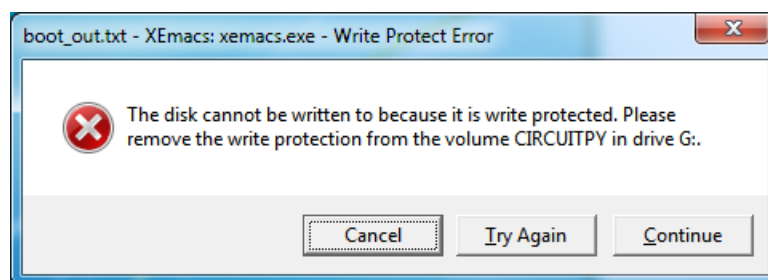
fritzing



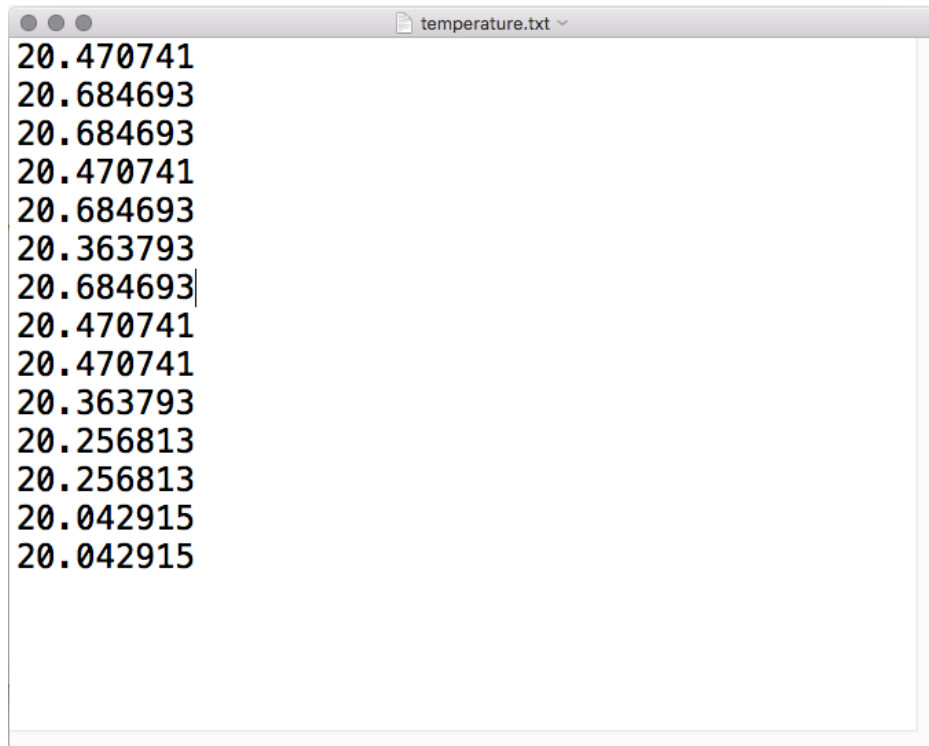
boot.py only runs on first boot of the device, not if you re-load the serial console with ctrl+D or if you save a file. You must EJECT the USB drive, then physically press the reset button!

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.



```
20.470741
20.684693
20.684693
20.470741
20.684693
20.363793
20.684693
20.470741
20.470741
20.363793
20.256813
20.256813
20.042915
20.042915
```

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://adafruit.com/blog/2017/07/20/cpu-temperature-logging-with-circuitpython/). (<https://adafruit.com/blog/2017/07/20/cpu-temperature-logging-with-circuitpython/>) If you'd like more details, check it out!

Playground Temperature

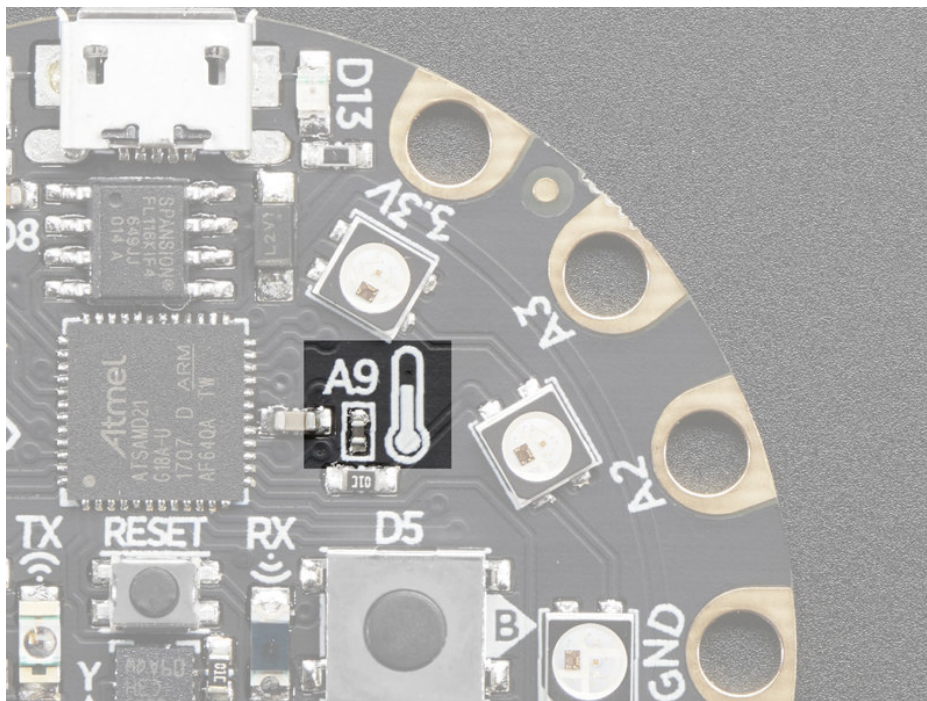


Though the following example uses the Circuit Playground Express to demonstrate, the code works exactly the same way with the Circuit Playground Bluefruit. Simply copy the code and follow along with your Circuit Playground Bluefruit!

But wait! There's more -- the Circuit Playground Express can also tell the temperature!

How, you ask? With a built-in thermistor. This little sensor is a thermally sensitive resistor, meaning its resistance changes based on temperature.

We can access its readings in CircuitPython by importing the `adafruit_thermistor` library, and then using the `board.TEMPERATURE` pin to read the thermistor value.



Copy the code below in to a new file, then save it onto the board as `main.py`. Then, open up a REPL session and you'll see the temperature readings in both Celsius and Fahrenheit.


```

# Circuit Playground Temperature
# Reads the on-board temperature sensor and prints the value

import time

import adafruit_thermistor
import board

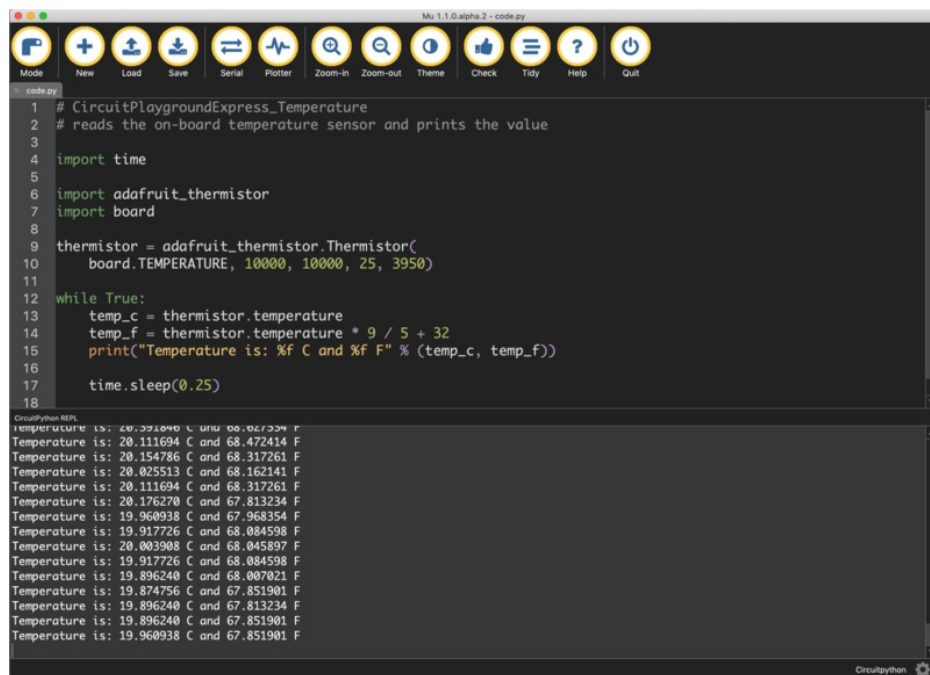
thermistor = adafruit_thermistor.Thermistor(
    board.TEMPERATURE, 10000, 10000, 25, 3950)

while True:
    temp_c = thermistor.temperature
    temp_f = thermistor.temperature * 9 / 5 + 32
    print("Temperature is: %f C and %f F" % (temp_c, temp_f))

    time.sleep(0.25)

```

Try placing your finger over the sensor (you'll see a thermometer icon on the board) and watch the readings change.



The screenshot shows the CircuitPython IDE interface. The top toolbar includes icons for Mode, New, Load, Save, Serial, Plotter, Zoom-in, Zoom-out, Theme, Check, Tidy, Help, and Quit. The code editor displays the same Python code as shown in the first block. The bottom console window shows the output of the program, which prints temperature readings in both Celsius and Fahrenheit every 0.25 seconds. The readings fluctuate slightly, reflecting the change in temperature when a finger is placed over the sensor.

```

CircuitPython REPL
1 Temperature is: 20.111694 C and 68.472414 F
2 Temperature is: 20.154786 C and 68.317261 F
3 Temperature is: 20.025513 C and 68.162141 F
4 Temperature is: 20.111694 C and 68.317261 F
5 Temperature is: 20.176270 C and 67.813234 F
6 Temperature is: 19.960938 C and 67.968354 F
7 Temperature is: 19.917726 C and 68.045898 F
8 Temperature is: 20.003908 C and 68.045897 F
9 Temperature is: 19.917726 C and 68.045898 F
10 Temperature is: 19.896240 C and 68.007021 F
11 Temperature is: 19.874756 C and 67.851901 F
12 Temperature is: 19.896240 C and 67.813234 F
13 Temperature is: 19.896240 C and 67.851901 F
14 Temperature is: 19.960938 C and 67.851901 F

```

Playground Light Sensor

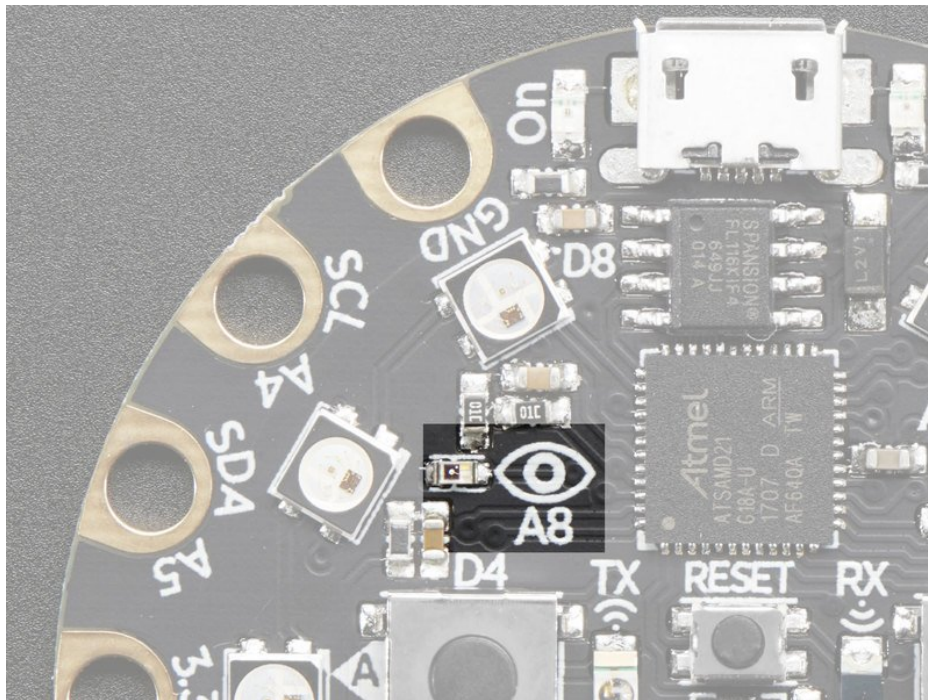


Though the following example uses the Circuit Playground Express to demonstrate, the code works exactly the same way with the Circuit Playground Bluefruit. Simply copy the code and follow along with your Circuit Playground Bluefruit!

The Circuit Playground Express can see you! OK, not really. That would be creepy.

But, it can sense light and dark, as well as colors and even your pulse!!

The Light Sensor in the upper left of the board (look for the eye icon) is a phototransistor. Here's how to use it as a light sensor:



```

# Circuit Playground Light Sensor
# Reads the on-board light sensor and graphs the brightness with NeoPixels

import time
import board
import neopixel
import analogio
import simpleio

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10, brightness=.05, auto_write=False)
pixels.fill((0, 0, 0))
pixels.show()

light = analogio.AnalogIn(board.LIGHT)

while True:
    # light value remapped to pixel position
    peak = simpleio.map_range(light.value, 2000, 62000, 0, 9)
    print(light.value)
    print(int(peak))

    for i in range(0, 9, 1):
        if i <= peak:
            pixels[i] = (0, 255, 0)
        else:
            pixels[i] = (0, 0, 0)
    pixels.show()

    time.sleep(0.01)

```

Copy and paste that code into a text editor and then save it to your Circuit Playground Express as **code.py**.

The code reads the light sensor and then lights up the NeoPixels like a bar graph depending on the light level. Try waving your hand over it, or shining it with a flashlight to see it change!

Playground Drum Machine



Though the following example uses the Circuit Playground Express to demonstrate, the code works exactly the same way with the Circuit Playground Bluefruit. Simply copy the code and follow along with your Circuit Playground Bluefruit!

A wise man once said, *"Nothing sounds better than an Eight-O-Eight."*

(That wise man was Adam Horovitz of the Beastie Boys.)

The 808 to which Ad-Rock was referring is the Roland TR-808 drum machine. Let's build a little Circuit Playground Express tribute to the venerable 808! Instead of a full-blown drum pattern sequencer, we'll just focus on the machine's pads which are used for finger drumming to play back sampled drum sounds.

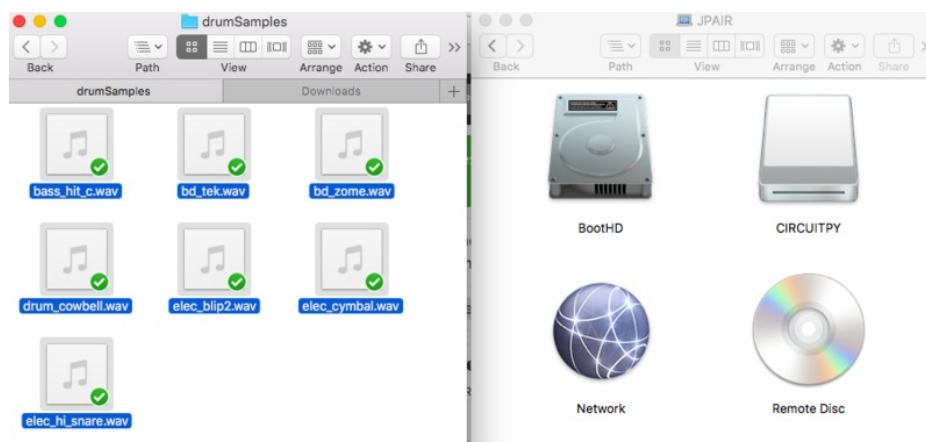
We can use the capacitive touch pads on the Circuit Playground Express as triggers, and small .wav files for our drum sounds!

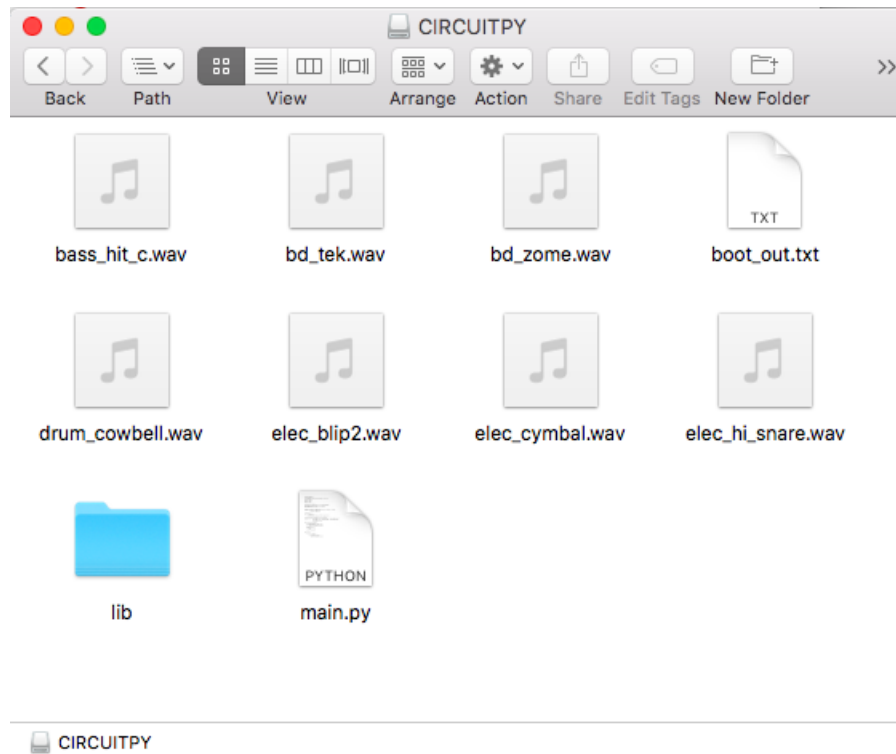
First, download the .zip file below, which contains all of the drum samples we'll be using. Save the file to your desktop or somewhere else easy to find, and then unzip it.

<https://adafru.it/zHc>

<https://adafru.it/zHc>

You can plug in your Circuit Playground Express, and then drag the drum files onto it. It shows up as the CIRCUITPY drive.





Now, it's time to code the Circuit Playground Express! Copy the code shown below, and then paste it into your code editor of choice. Save the file as **code.py** on your CIRCUITPY drive.

```

# Circuit Playground 808 Drum machine
import time
import board
import touchio
import digitalio

try:
    from audiocore import WaveFile
except ImportError:
    from audioio 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!

bpm = 120 # Beats per minute, change this to suit your tempo

# Enable the speaker
speaker_enable = digitalio.DigitalInOut(board.SPEAKER_ENABLE)
speaker_enable.direction = digitalio.Direction.OUTPUT
speaker_enable.value = True

# Make the input capacitive touchpads
capPins = (board.A1, board.A2, board.A3, board.A4, board.A5,
           board.A6, board.TX)

touchPad = []
for i in range(7):
    touchPad.append(touchio.TouchIn(capPins[i]))

# The seven files assigned to the touchpads
audiofiles = ["bd_tek.wav", "elec_hi_snare.wav", "elec_cymbal.wav",
              "elec_blip2.wav", "bd_zome.wav", "bass_hit_c.wav",
              "drum_cowbell.wav"]

audio = AudioOut(board.SPEAKER)

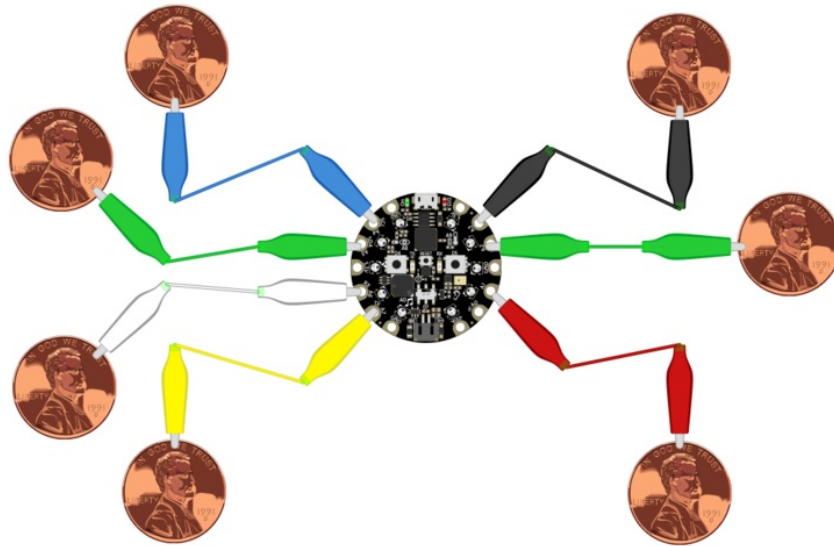
def play_file(filename):
    print("playing file " + filename)
    file = open(filename, "rb")
    wave = WaveFile(file)
    audio.play(wave)
    time.sleep(bpm / 960) # Sixteenth note delay

while True:
    for i in range(7):
        if touchPad[i].value:
            play_file(audiofiles[i])

```

Try it out! When you tap the capacitive pads, the corresponding drum sample is triggered!!

Things are a bit cramped, admittedly, so you can try adding foil, copper tape, alligator clips, etc. in order to increase the surface area and physical space you have for your drumming!



Capacitance is calibrated at startup, so you may need to reset the board after attaching leads to the pads!

If you want to use your own sound files, you can! Record, sample, remix, or simply download files from a sound file sight, such as freesample.org. Then, to make sure you have the files converted to the proper specifications, [check out this guide here \(https://adafruit.com/blog/2017/01/11/adafruit-circuit-playground-bluefruit/\)](https://adafruit.com/blog/2017/01/11/adafruit-circuit-playground-bluefruit/) that'll show you how! Spoiler alert: you'll need to make a small, 22Khz (or lower), 16 bit PCM, mono .wav file!

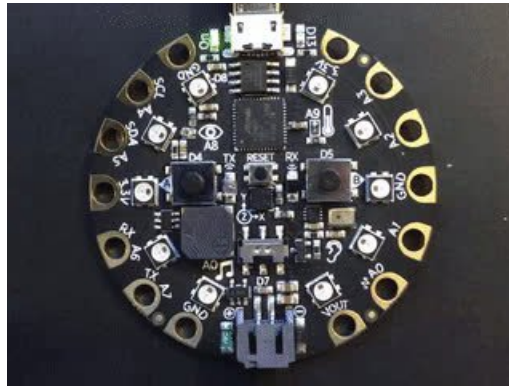
Want to listen to your Drum Machine at body movin' volumes? No problem! Hook up an $\frac{1}{8}$ " (<https://adafruit.com/blog/2017/01/11/adafruit-circuit-playground-bluefruit/>) or $\frac{1}{4}$ " [phono output \(https://adafruit.com/blog/2017/01/11/adafruit-circuit-playground-bluefruit/\)](https://adafruit.com/blog/2017/01/11/adafruit-circuit-playground-bluefruit/) to the **GND** and **A0/Audio** pads, then plug in to an amp!! I tried it on a small guitar amp and it sounds great.

Playground Sound Meter



Though the following example uses the Circuit Playground Express to demonstrate, the code works exactly the same way with the Circuit Playground Bluefruit. Simply copy the code and follow along with your Circuit Playground Bluefruit!

Use the microphone on your Circuit Playground Express to measure sound levels and display them on a VU-meter-like display!



The program is below. There are many settings that you can change to make the readings more or less sensitive and the display more or less jumpy. Try changing `CURVE` to be 4 or 1 or 10 or -2 and see what happens.

The program samples audio for a short time and then computes the energy in the sample (corresponding to volume) using a [Root-Mean-Square](https://adafru.it/Bf5) (RMS) computation. You could try varying the sample size if you like.

The `log_scale()` function varies the number of NeoPixels lit in an exponential way, because sound levels can vary by many factors of 10 between loud and soft. Try varying how it's computed to see what happens.

The program takes one sample when it first starts to set a "quiet" sound level. If that doesn't work for you, set `input_floor` to be a fixed number. If the meter seems too sensitive, try changing `input_ceiling = input_floor + 500` to be `input_ceiling = input_floor + 2000` or higher. Or go the other way.

You can also change the colors. Try different ways of computing `volume_color(i)` for more of a rainbow effect, or make it a constant if you don't like changing colors.

```
# The MIT License (MIT)
#
# Copyright (c) 2017 Dan Halbert for Adafruit Industries
# Copyright (c) 2017 Kattni Rembor, Tony DiCola for Adafruit Industries
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
```



```

# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.

# Circuit Playground Sound Meter

import array
import math
import audiobusio
import board
import neopixel

# Color of the peak pixel.
PEAK_COLOR = (100, 0, 255)
# Number of total pixels - 10 build into Circuit Playground
NUM_PIXELS = 10

# Exponential scaling factor.
# Should probably be in range -10 .. 10 to be reasonable.
CURVE = 2
SCALE_EXPONENT = math.pow(10, CURVE * -0.1)

# Number of samples to read at once.
NUM_SAMPLES = 160

# Restrict value to be between floor and ceiling.
def constrain(value, floor, ceiling):
    return max(floor, min(value, ceiling))

# Scale input_value between output_min and output_max, exponentially.
def log_scale(input_value, input_min, input_max, output_min, output_max):
    normalized_input_value = (input_value - input_min) / \
        (input_max - input_min)
    return output_min + \
        math.pow(normalized_input_value, SCALE_EXPONENT) \
        * (output_max - output_min)

# Remove DC bias before computing RMS.
def normalized_rms(values):
    minbuf = int(mean(values))
    samples_sum = sum(
        float(sample - minbuf) * (sample - minbuf)
        for sample in values
    )
    return math.sqrt(samples_sum / len(values))

def mean(values):
    return sum(values) / len(values)

```

```

def volume_color(volume):
    return 200, volume * (255 // NUM_PIXELS), 0

# Main program

# Set up NeoPixels and turn them all off.
pixels = neopixel.NeoPixel(board.NEOPIXEL, NUM_PIXELS, brightness=0.1, auto_write=False)
pixels.fill(0)
pixels.show()

mic = audiobusio.PDMIn(board.MICROPHONE_CLOCK, board.MICROPHONE_DATA,
                        sample_rate=16000, bit_depth=16)

# Record an initial sample to calibrate. Assume it's quiet when we start.
samples = array.array('H', [0] * NUM_SAMPLES)
mic.record(samples, len(samples))
# Set lowest level to expect, plus a little.
input_floor = normalized_rms(samples) + 10
# OR: used a fixed floor
# input_floor = 50

# You might want to print the input_floor to help adjust other values.
# print(input_floor)

# Corresponds to sensitivity: lower means more pixels light up with lower sound
# Adjust this as you see fit.
input_ceiling = input_floor + 500

peak = 0
while True:
    mic.record(samples, len(samples))
    magnitude = normalized_rms(samples)
    # You might want to print this to see the values.
    # print(magnitude)

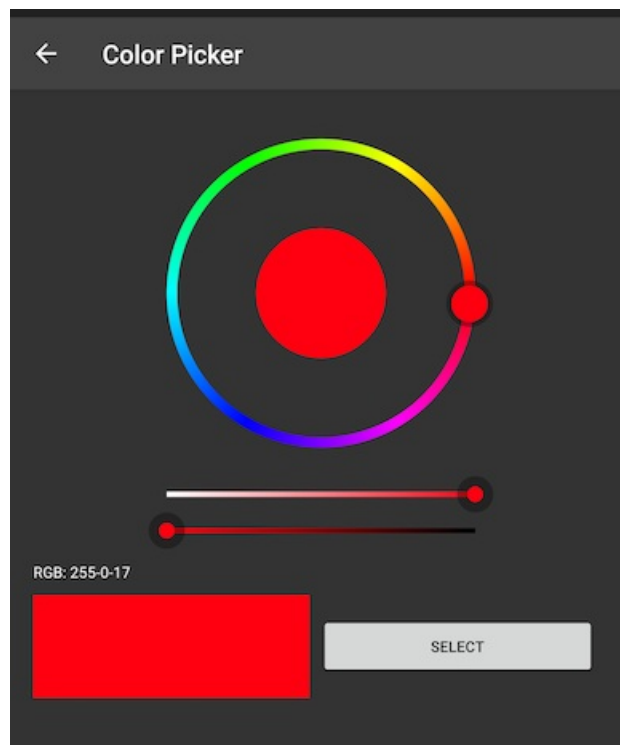
    # Compute scaled logarithmic reading in the range 0 to NUM_PIXELS
    c = log_scale(constrain(magnitude, input_floor, input_ceiling),
                  input_floor, input_ceiling, 0, NUM_PIXELS)

    # Light up pixels that are below the scaled and interpolated magnitude.
    pixels.fill(0)
    for i in range(NUM_PIXELS):
        if i < c:
            pixels[i] = volume_color(i)
    # Light up the peak pixel and animate it slowly dropping.
    if c >= peak:
        peak = min(c, NUM_PIXELS - 1)
    elif peak > 0:
        peak = peak - 1
    if peak > 0:
        pixels[int(peak)] = PEAK_COLOR
    pixels.show()

```

Playground Color Picker

You can use your Circuit Playground Bluefruit with the [Adafruit Bluefruit LE Connect](https://adafru.it/GcN) (<https://adafru.it/GcN>) mobile app to control the NeoPixel RGB LEDs on the CPB!



The Code

Plug your Circuit Playground Bluefruit into your computer, and save the following as **code.py** on the **CIRCUITPY** drive:

```
# CircuitPython NeoPixel Color Picker Example

import board
import neopixel
from adafruit_ble import BLERadio
from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
from adafruit_ble.services.nordic import UARTService
from adafruit_bluefruit_connect.packet import Packet
from adafruit_bluefruit_connect.color_packet import ColorPacket

ble = BLERadio()
uart_server = UARTService()
advertisement = ProvideServicesAdvertisement(uart_server)

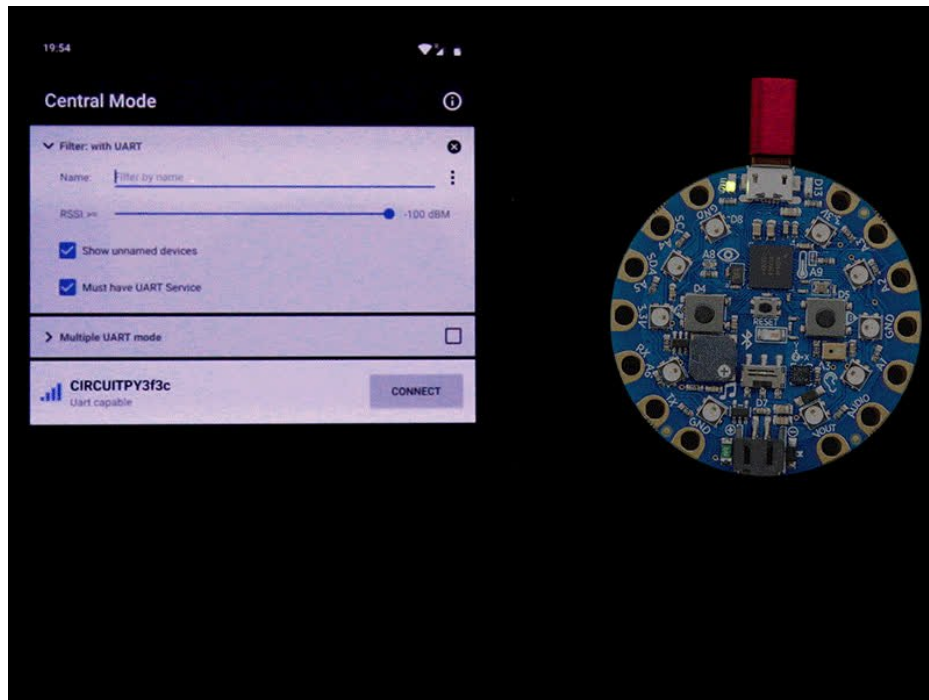
pixels = neopixel.NeoPixel(board.NEOPIXEL, 10, brightness=0.1)

while True:
    # Advertise when not connected.
    ble.start_advertising(advertisement)
    while not ble.connected:
        pass
    ble.stop_advertising()

    while ble.connected:
        packet = Packet.from_stream(uart_server)
        if isinstance(packet, ColorPacket):
            print(packet.color)
            pixels.fill(packet.color)
```

Connect to your board through the Adafruit Bluefruit LE Connect mobile app. If you need assistance, [check out this page on installing and using the app \(https://adafru.it/GcN\)](https://adafru.it/GcN).

Once connected, from the device menu, tap on Controller, then Color Picker. Choose a color from the dial and tap Select (Android) or Send selected color (iOS). The LEDs will light up in the color you chose!



Playground Bluetooth Plotter

The Circuit Playground Bluefruit has a built in light sensor that returns a light value and a temperature sensor that returns the temperature in degrees Celsius. The Adafruit Bluefruit LE Connect mobile app has a built in plotter function that you can use to plot any numerical information. This page will show you how to plot the light and temperature data from the Circuit Playground Bluefruit in the Bluefruit LE Connect app!

The Code

Plug your Circuit Playground Bluefruit into your computer, and save the following as **code.py** on the **CIRCUITPY** drive:

```
# CircuitPython Bluefruit LE Connect Plotter Example

import time
import board
import analogio
import adafruit_thermistor
from adafruit_ble import BLERadio
from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
from adafruit_ble.services.nordic import UARTService

ble = BLERadio()
uart_server = UARTService()
advertisement = ProvideServicesAdvertisement(uart_server)

thermistor = adafruit_thermistor.Thermistor(board.TEMPERATURE, 10000, 10000, 25, 3950)
light = analogio.AnalogIn(board.LIGHT)

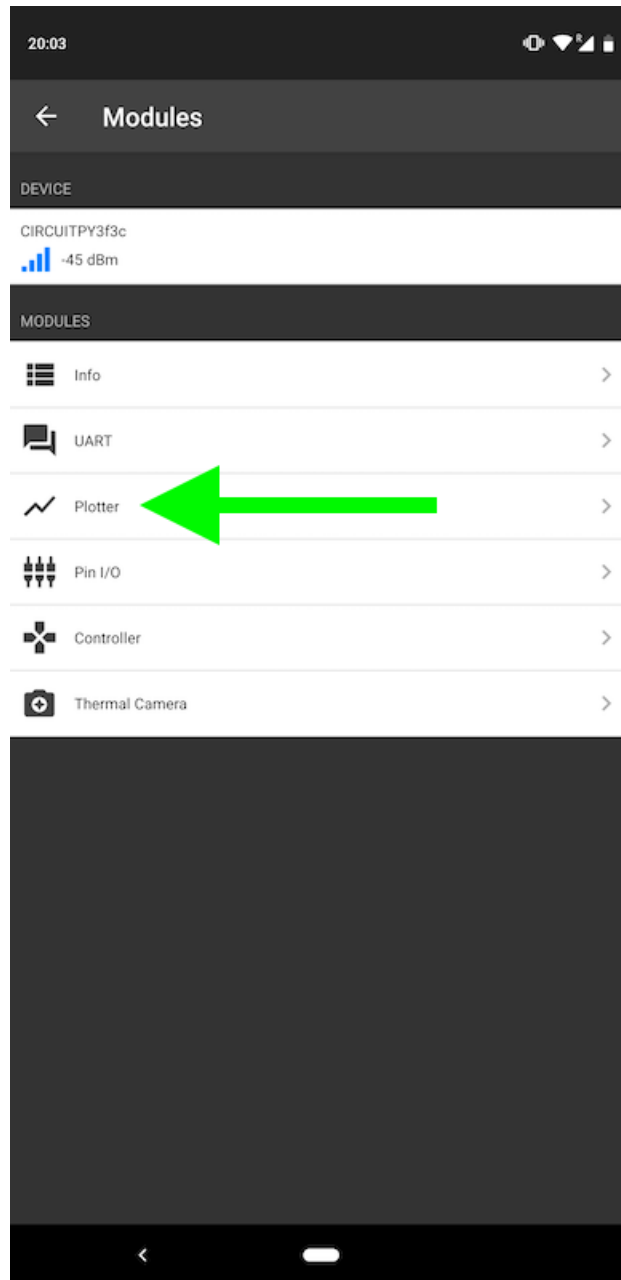
def scale(value):
    """Scale the light sensor values from 0-65535 (AnalogIn range)
    to 0-50 (arbitrarily chosen to plot well with temperature)"""
    return value / 65535 * 50

while True:
    # Advertise when not connected.
    ble.start_advertising(advertisement)
    while not ble.connected:
        pass
    ble.stop_advertising()

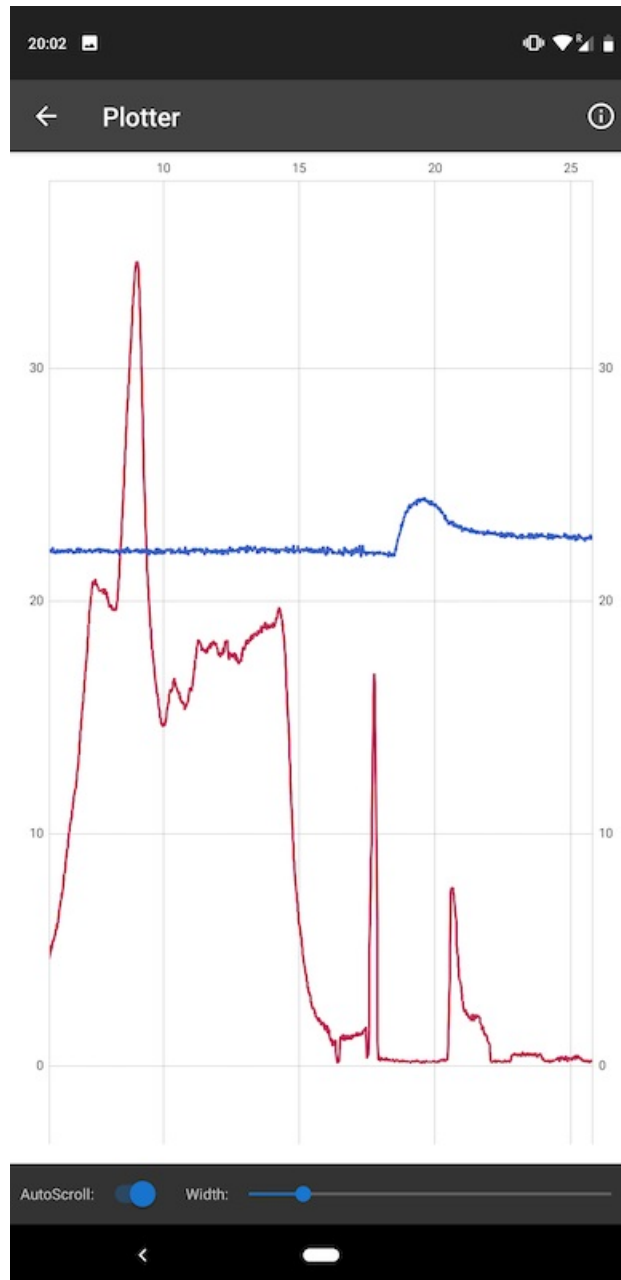
    while ble.connected:
        print((scale(light.value), thermistor.temperature))
        uart_server.write("{}{}\n".format(scale(light.value), thermistor.temperature))
        time.sleep(0.1)
```

Connect to your board through the Adafruit Bluefruit LE Connect mobile app. If you need assistance, check out [the Bluefruit LE Connect Basics page in the Getting Started guide \(https://adafru.it/F-x\)](https://adafru.it/F-x).

Once connected, tap Plotter.



Your data should start plotting automatically. Try shining a light towards your Circuit Playground Bluefruit to see the light value line change. Try placing your finger over the thermistor (towards the top-right, labeled A9, next to the picture of a thermometer) to see the temperature value line change.



That's all there is to plotting numerical data with the Circuit Playground Bluefruit and the Adafruit Bluefruit LE Connect mobile app!

Arduino Support Setup

You can install the Adafruit Bluefruit nRF52 BSP (Board Support Package) in two steps:



nRF52 support requires at least Arduino IDE version 1.8.6! Please make sure you have an up to date version before proceeding with this guide!

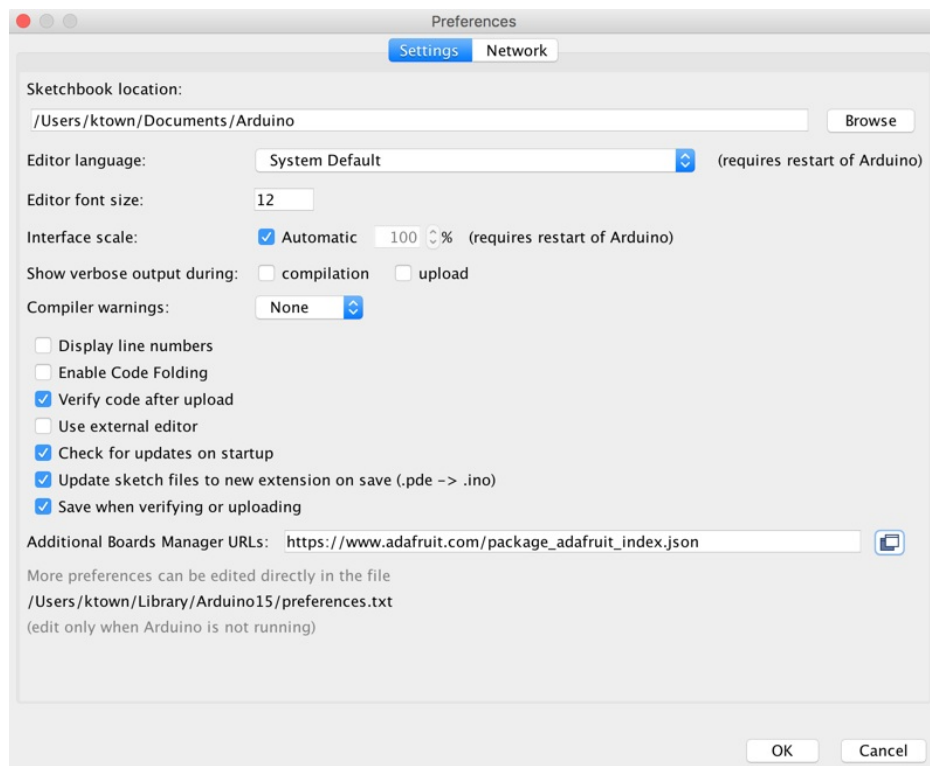


Please consult the FAQ section at the bottom of this page if you run into any problems installing or using this BSP!

1. BSP Installation

Recommended: Installing the BSP via the Board Manager

- Download and install the Arduino IDE (<https://adafru.it/fvm>) (At least v1.8)
- Start the Arduino IDE
- Go into Preferences
- Add https://www.adafruit.com/package_adafruit_index.json as an 'Additional Board Manager URL' (see image below)

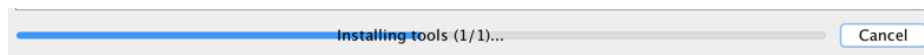


- Restart the Arduino IDE
- Open the **Boards Manager** option from the **Tools -> Board** menu and install '**Adafruit nRF52 by Adafruit**' (see image below)



It will take up to a few minutes to finish installing the cross-compiling toolchain and tools associated with this BSP.

The delay during the installation stage shown in the image below is normal, please be patient and let the installation terminate normally:



- Once the BSP is installed, select '**Adafruit Bluefruit nRF52832 Feather**' (for the nRF52 Feather) of '**Adafruit Bluefruit nRF52840 Feather Express**' (for the nRF52840 Feather) from the **Tools** -> **Board** menu, which will update your system config to use the right compiler and settings for the nRF52:



2. LINUX ONLY: adafruit-nrfutil Tool Installation

adafruit-nrfutil (<https://adafru.it/Cau>) is a **modified version** of Nordic's **nrfutil** (<https://adafru.it/vaG>), which is used to flash boards using the built in serial bootloader. It is originally written for python2, but have been migrated to python3 and renamed to **adafruit-nrfutil** since BSP version 0.8.5.



This step is only required on Linux, pre-built binaries of **adafruit-nrfutil** for Windows and MacOS are already included in the BSP. That should work out of the box for most setups.

Install python3 if it is not installed in your system already

```
$ sudo apt-get install python3
```

Then run the following command to install the tool from PyPi

```
$ pip3 install --user adafruit-nrfutil
```

Add pip3 installation dir to your **PATH** if it is not added already. Make sure **adafruit-nrfutil** can be executed in terminal by running

```
$ adafruit-nrfutil version
adafruit-nrfutil version 0.5.3.post12
```

3. Update the bootloader (nRF52832 ONLY)

To keep up with Nordic's SoftDevice advances, you will likely need to update your bootloader if you are using the original nRF52832 based **Bluefruit nRF52 Feather** boards.

Follow this link for instructions on how to do that



This step ISN'T required for the newer nRF52840 Feather Express, which has a different bootloader entirely!

<https://adafru.it/Dsx>

<https://adafru.it/Dsx>

Advanced Option: Manually Install the BSP via 'git'

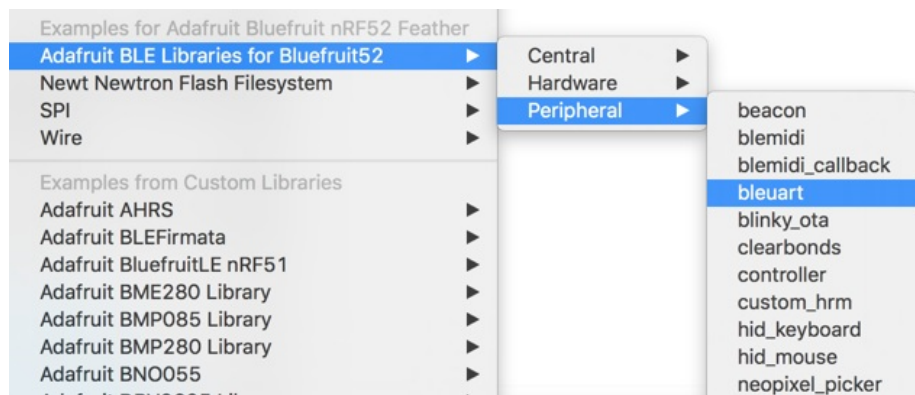
If you wish to do any development against the core codebase (generate pull requests, etc.), you can also optionally install the Adafruit nRF52 BSP manually using 'git', as described below:

Adafruit nRF52 BSP via git (for core development and PRs only)

1. Install BSP via Board Manager as above to install compiler & tools.
2. Delete the core folder **nrf52** installed by Board Manager in Arduino15, depending on your OS. It could be
macOS: `~/Library/Arduino15/packages/adafruit/hardware/nrf52`
Linux: `~/arduino15/packages/adafruit/hardware/nrf52`
Windows: `%APPDATA%\Local\Arduino15\packages\adafruit\hardware\nrf52`
3. Go to the sketchbook folder on your command line, which should be one of the following:
macOS: `~/Documents/Arduino`
Linux: `~/Arduino`
Windows: `~/Documents/Arduino`
4. Create a folder named **hardware/Adafruit**, if it does not exist, and change directories into it.
5. Clone the **Adafruit_nRF52_Arduino** (<https://adafru.it/vaF>) repo in the folder described in step 2:
`git clone git@github.com:adafruit/Adafruit_nRF52_Arduino.git`
6. This should result in a final folder name like
`~/Documents/Arduino/hardware/Adafruit/Adafruit_nRF52_Arduino` (macOS).
7. Restart the Arduino IDE

Arduino BLE Examples

There are numerous examples available for the Bluefruit nRF52/nRF52840 Feathers in the **Examples** menu of the nRF52 BSP, and these are always up to date. You're first stop looking for example code should be there:



Example Source Code

The latest example source code is always available and visible on Github, and the public git repository should be considered the definitive source of example code for this board.

<https://adafru.it/vaK>

<https://adafru.it/vaK>

Documented Examples

To help explain some common use cases for the nRF52 BLE API, feel free to consult the example documentation in this section of the learning guide:

- **Advertising: Beacon** - Shows how to use the BLEBeacon helper class to configure your Bluefruit nRF52 Feather as a beacon
- **BLE UART: Controller** - Shows how to use the **Controller** utility in our Bluefruit LE Connect apps to send basic data between your peripheral and your phone or tablet.
- **Custom: HRM** - Shows how to defined and work with a custom GATT Service and Characteristic, using the officially adopted Heart Rate Monitor (HRM) service as an example.
- **BLE Pin I/O** (StandardFirmataBLE) Shows how to control Pin I/O of nRF52 with Firmata protocol

Advertising: Beacon

This example shows how you can use the BLEBeacon helper class and advertising API to configure your Bluefruit nRF52 board as a 'Beacon'.

Complete Code

```
/*
*****
This is an example for our nRF52 based Bluefruit LE modules

Pick one up today in the adafruit shop!

Adafruit invests time and resources providing this open source code,
please support Adafruit and open-source hardware by purchasing
products from Adafruit!

MIT license, check LICENSE for more information
All text above, and the splash screen below must be included in
any redistribution
*****
#include <bluefruit.h>

// Beacon uses the Manufacturer Specific Data field in the advertising
// packet, which means you must provide a valid Manufacturer ID. Update
// the field below to an appropriate value. For a list of valid IDs see:
// https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers
// 0x004C is Apple (for example)
#define MANUFACTURER_ID 0x004C

// AirLocate UUID: E2C56DB5-DFFB-48D2-B060-D0F5A71096E0
uint8_t beaconUuid[16] =
{
  0xE2, 0xC5, 0x6D, 0xB5, 0xDF, 0xFB, 0x48, 0xD2,
  0xB0, 0x60, 0xD0, 0xF5, 0xA7, 0x10, 0x96, 0xE0,
};

// A valid Beacon packet consists of the following information:
// UUID, Major, Minor, RSSI @ 1M
BLEBeacon beacon(beaconUuid, 0x0000, 0x0000, -54);

void setup()
{
  Serial.begin(115200);
  while ( !Serial ) delay(10); // for nrf52840 with native usb

  Serial.println("Bluefruit52 Beacon Example");
  Serial.println("-----\n");

  Bluefruit.begin();

  // off Blue LED for lowest power consumption
  Bluefruit.autoConnLed(false);
  Bluefruit.setTxPower(0); // Check bluefruit.h for supported values
  Bluefruit.setName("Bluefruit52");

  // Manufacturer ID is required for Manufacturer Specific Data
  Bluefruit.setManufacturerData(MANUFACTURER_ID, beaconUuid, 16);
}

void loop()
{
  beacon.advertise();
}
```

```

beacon.setManufacturer(MANUFACTURER_ID);

// Setup the advertising packet
startAdv();

Serial.println("Broadcasting beacon, open your beacon app to test");

// Suspend Loop() to save power, since we didn't have any code there
suspendLoop();
}

void startAdv(void)
{
  // Advertising packet
  // Set the beacon payload using the BLEBeacon class populated
  // earlier in this example
  Bluefruit.Advertising.setBeacon(beacon);

  // Secondary Scan Response packet (optional)
  // Since there is no room for 'Name' in Advertising packet
  Bluefruit.ScanResponse.addName();

  /* Start Advertising
   * - Enable auto advertising if disconnected
   * - Timeout for fast mode is 30 seconds
   * - Start(timeout) with timeout = 0 will advertise forever (until connected)
   *
   * Apple Beacon specs
   * - Type: Non connectable, undirected
   * - Fixed interval: 100 ms -> fast = slow = 100 ms
   */
  //Bluefruit.Advertising.setType(BLE_GAP_ADV_TYPE_ADV_NONCONN_IND);
  Bluefruit.Advertising.restartOnDisconnect(true);
  Bluefruit.Advertising.setInterval(160, 160); // in unit of 0.625 ms
  Bluefruit.Advertising.setFastTimeout(30); // number of seconds in fast mode
  Bluefruit.Advertising.start(0); // 0 = Don't stop advertising after n seconds
}

void loop()
{
  // loop is already suspended, CPU will not run loop() at all
}

```

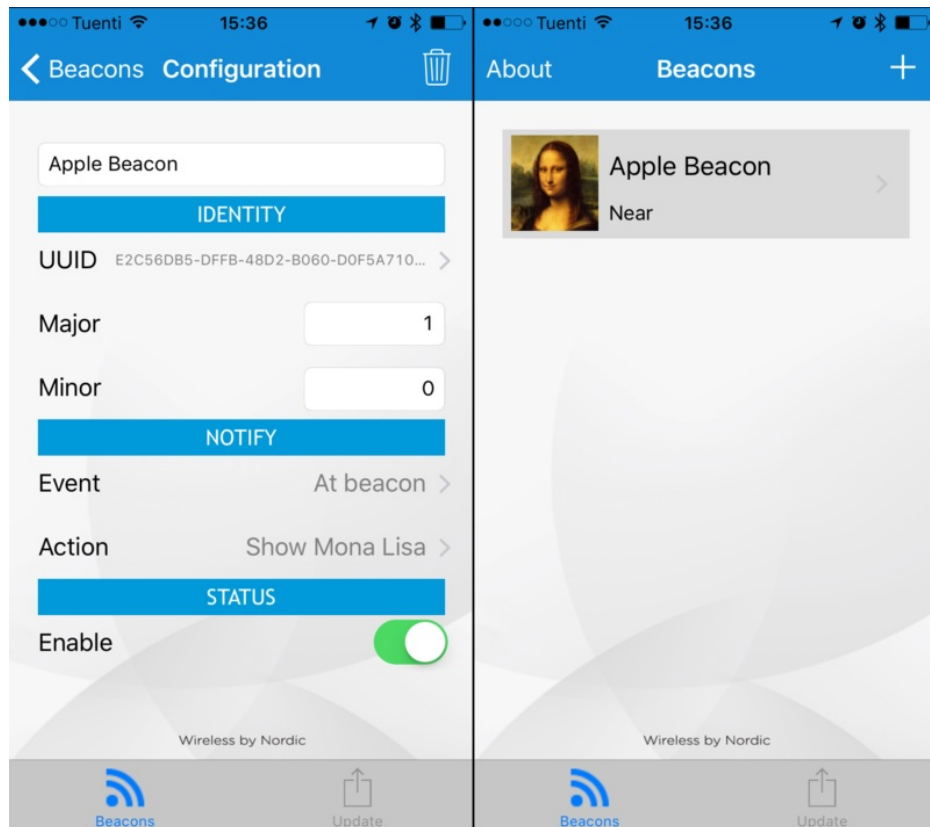
Output

You can use the nRF Beacons application from Nordic Semiconductors to test this sketch:

- nRF Beacons for iOS (<https://adafru.it/vaC>)
- nRF Beacons for Android (<https://adafru.it/vaD>)

Make sure that you set the UUID, Major and Minor values to match the sketch above, and then run the sketch at the same time as the nRF Beacons application.

With the default setup you should see a Mona Lisa icon when the beacon is detected. If you don't see this, double check the UUID, Major and Minor values to be sure they match exactly.



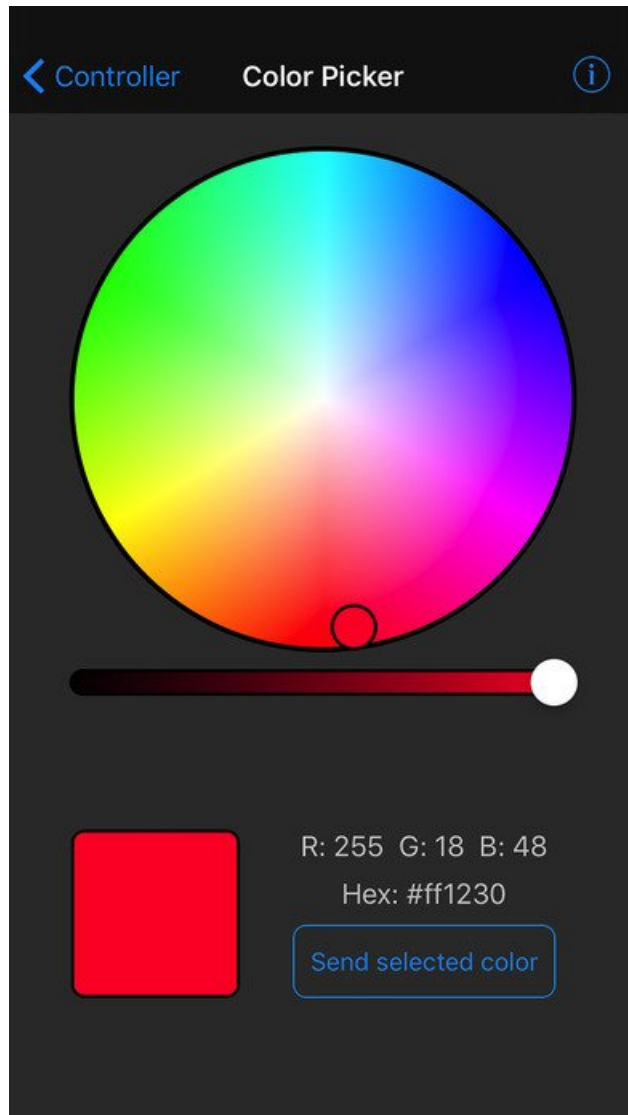
BLE UART: Controller

This examples shows you you can use the BLEUart helper class and the Bluefruit LE Connect applications to send based keypad and sensor data to your nRF52.

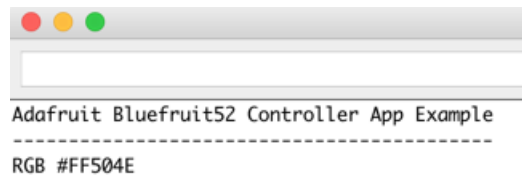
Setup

In order to use this sketch, you will need to open Bluefruit LE Connect on your mobile device using our free [iOS \(https://adafru.it/f4H\)](https://adafru.it/f4H), [Android \(https://adafru.it/f4G\)](https://adafru.it/f4G) or [OS X \(https://adafru.it/o9F\)](https://adafru.it/o9F) applications.

- Load the **Controller** example sketch (<https://adafru.it/vaN>) in the Arduino IDE
- Compile the sketch and flash it to your nRF52 based Feather
- Once you are done uploading, open the **Serial Monitor** (Tools > Serial Monitor)
- Open the **Bluefruit LE Connect** application on your mobile device
- Connect to the appropriate target (probably 'Bluefruit52')
- Once connected switch to the **Controller** application inside the app
- Enable an appropriate control surface. The **Color Picker** control surface is shown below, for example (screen shot taken from the iOS application):



As you change the color (or as other data becomes available) you should receive the data on the nRF52, and see it in the Serial Monitor output:



Complete Code

```

/*****
This is an example for our nRF52 based Bluefruit LE modules

Pick one up today in the adafruit shop!

Adafruit invests time and resources providing this open source code,
please support Adafruit and open source hardware by purchasing
*****/

```

please support Adafruit and open-source hardware by purchasing products from Adafruit!

MIT license, check LICENSE for more information

All text above, and the splash screen below must be included in any redistribution

*****/

```
#include <bluefruit.h>

// OTA DFU service
BLEDFU bleDFU;

// UART over BLE service
BLEUART bleUART;

// Function prototypes for packetparser.cpp
uint8_t readPacket (BLEUART *ble_uart, uint16_t timeout);
float   parseFloat (uint8_t *buffer);
void    printHex   (const uint8_t * data, const uint32_t numBytes);

// Packet buffer
extern uint8_t packetbuffer[];

void setup(void)
{
    Serial.begin(115200);
    while ( !Serial ) delay(10);    // for nrf52840 with native usb

    Serial.println(F("Adafruit Bluefruit52 Controller App Example"));
    Serial.println(F("-----"));

    Bluefruit.begin();
    Bluefruit.setTxPower(4);    // Check bluefruit.h for supported values
    Bluefruit.setName("Bluefruit52");

    // To be consistent OTA DFU should be added first if it exists
    bleDFU.begin();

    // Configure and start the BLE UART service
    bleUART.begin();

    // Set up and start advertising
    startAdv();

    Serial.println(F("Please use Adafruit Bluefruit LE app to connect in Controller mode"));
    Serial.println(F("Then activate/use the sensors, color picker, game controller, etc!"));
    Serial.println();
}

void startAdv(void)
{
    // Advertising packet
    Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);
    Bluefruit.Advertising.addTxPower();

    // Include the BLE UART (AKA 'NUS') 128-bit UUID
    Bluefruit.Advertising.addService(bleUART);

    // Secondary Scan Response packet (optional)
```

```

// Since there is no room for 'Name' in Advertising packet
Bluefruit.ScanResponse.addName();

/* Start Advertising
 * - Enable auto advertising if disconnected
 * - Interval: fast mode = 20 ms, slow mode = 152.5 ms
 * - Timeout for fast mode is 30 seconds
 * - Start(timeout) with timeout = 0 will advertise forever (until connected)
 *
 * For recommended advertising interval
 * https://developer.apple.com/library/content/qa/qa1931/_index.html
 */
Bluefruit.Advertising.restartOnDisconnect(true);
Bluefruit.Advertising.setInterval(32, 244); // in unit of 0.625 ms
Bluefruit.Advertising.setFastTimeout(30); // number of seconds in fast mode
Bluefruit.Advertising.start(0); // 0 = Don't stop advertising after n seconds
}

/*****
 *!
 @brief Constantly poll for new command or response data
 */
*****/
void loop(void)
{
    // Wait for new data to arrive
    uint8_t len = readPacket(&bleuart, 500);
    if (len == 0) return;

    // Got a packet!
    // printHex(packetbuffer, len);

    // Color
    if (packetbuffer[1] == 'C') {
        uint8_t red = packetbuffer[2];
        uint8_t green = packetbuffer[3];
        uint8_t blue = packetbuffer[4];
        Serial.print("RGB #");
        if (red < 0x10) Serial.print("0");
        Serial.print(red, HEX);
        if (green < 0x10) Serial.print("0");
        Serial.print(green, HEX);
        if (blue < 0x10) Serial.print("0");
        Serial.println(blue, HEX);
    }

    // Buttons
    if (packetbuffer[1] == 'B') {
        uint8_t buttnum = packetbuffer[2] - '0';
        boolean pressed = packetbuffer[3] - '0';
        Serial.print("Button "); Serial.print(buttnum);
        if (pressed) {
            Serial.println(" pressed");
        } else {
            Serial.println(" released");
        }
    }

    // GPS Location
    if (packetbuffer[1] == 'L') {

```

```

if (packetbuffer[1] == 'L') {
  float lat, lon, alt;
  lat = parsefloat(packetbuffer+2);
  lon = parsefloat(packetbuffer+6);
  alt = parsefloat(packetbuffer+10);
  Serial.print("GPS Location\t");
  Serial.print("Lat: "); Serial.print(lat, 4); // 4 digits of precision!
  Serial.print('\t');
  Serial.print("Lon: "); Serial.print(lon, 4); // 4 digits of precision!
  Serial.print('\t');
  Serial.print(alt, 4); Serial.println(" meters");
}

// Accelerometer
if (packetbuffer[1] == 'A') {
  float x, y, z;
  x = parsefloat(packetbuffer+2);
  y = parsefloat(packetbuffer+6);
  z = parsefloat(packetbuffer+10);
  Serial.print("Accel\t");
  Serial.print(x); Serial.print('\t');
  Serial.print(y); Serial.print('\t');
  Serial.print(z); Serial.println();
}

// Magnetometer
if (packetbuffer[1] == 'M') {
  float x, y, z;
  x = parsefloat(packetbuffer+2);
  y = parsefloat(packetbuffer+6);
  z = parsefloat(packetbuffer+10);
  Serial.print("Mag\t");
  Serial.print(x); Serial.print('\t');
  Serial.print(y); Serial.print('\t');
  Serial.print(z); Serial.println();
}

// Gyroscope
if (packetbuffer[1] == 'G') {
  float x, y, z;
  x = parsefloat(packetbuffer+2);
  y = parsefloat(packetbuffer+6);
  z = parsefloat(packetbuffer+10);
  Serial.print("Gyro\t");
  Serial.print(x); Serial.print('\t');
  Serial.print(y); Serial.print('\t');
  Serial.print(z); Serial.println();
}

// Quaternions
if (packetbuffer[1] == 'Q') {
  float x, y, z, w;
  x = parsefloat(packetbuffer+2);
  y = parsefloat(packetbuffer+6);
  z = parsefloat(packetbuffer+10);
  w = parsefloat(packetbuffer+14);
  Serial.print("Quat\t");
  Serial.print(x); Serial.print('\t');
  Serial.print(y); Serial.print('\t');
  Serial.print(z); Serial.print('\t');

```

```

    Serial.print(w); Serial.println();
  }
}

```

You will also need the following helper class in a file called **packetParser.cpp**:

```

#include <string.h>
#include <Arduino.h>
#include <bluefruit.h>

#define PACKET_ACC_LEN           (15)
#define PACKET_GYRO_LEN         (15)
#define PACKET_MAG_LEN          (15)
#define PACKET_QUAT_LEN         (19)
#define PACKET_BUTTON_LEN       (5)
#define PACKET_COLOR_LEN        (6)
#define PACKET_LOCATION_LEN     (15)

//   READ_BUFSIZE           Size of the read buffer for incoming packets
#define READ_BUFSIZE           (20)

/* Buffer to hold incoming characters */
uint8_t packetbuffer[READ_BUFSIZE+1];

/*****
 *!
 *brief Casts the four bytes at the specified address to a float
 */
/*****/
float parsefloat(uint8_t *buffer)
{
  float f;
  memcpy(&f, buffer, 4);
  return f;
}

/*****
 *!
 *brief Prints a hexadecimal value in plain characters
 *param data      Pointer to the byte data
 *param numBytes  Data length in bytes
 */
/*****/
void printHex(const uint8_t * data, const uint32_t numBytes)
{
  uint32_t szPos;
  for (szPos=0; szPos < numBytes; szPos++)
  {
    Serial.print(F("0x"));
    // Append leading 0 for small values
    if (data[szPos] <= 0xF)
    {
      Serial.print(F("0"));
      Serial.print(data[szPos] & 0xf, HEX);
    }
    else
    {

```

```

    {
        Serial.print(data[szPos] & 0xff, HEX);
    }
    // Add a trailing space if appropriate
    if ((numBytes > 1) && (szPos != numBytes - 1))
    {
        Serial.print(F(" "));
    }
}
Serial.println();
}

/*****
/*!
    @brief  Waits for incoming data and parses it
*/
*****/
uint8_t readPacket(BLEUart *ble_uart, uint16_t timeout)
{
    uint16_t origtimeout = timeout, replyidx = 0;

    memset(packetbuffer, 0, READ_BUFSIZE);

    while (timeout--) {
        if (replyidx >= 20) break;
        if ((packetbuffer[1] == 'A') && (replyidx == PACKET_ACC_LEN))
            break;
        if ((packetbuffer[1] == 'G') && (replyidx == PACKET_GYRO_LEN))
            break;
        if ((packetbuffer[1] == 'M') && (replyidx == PACKET_MAG_LEN))
            break;
        if ((packetbuffer[1] == 'Q') && (replyidx == PACKET_QUAT_LEN))
            break;
        if ((packetbuffer[1] == 'B') && (replyidx == PACKET_BUTTON_LEN))
            break;
        if ((packetbuffer[1] == 'C') && (replyidx == PACKET_COLOR_LEN))
            break;
        if ((packetbuffer[1] == 'L') && (replyidx == PACKET_LOCATION_LEN))
            break;

        while (ble_uart->available()) {
            char c = ble_uart->read();
            if (c == '!') {
                replyidx = 0;
            }
            packetbuffer[replyidx] = c;
            replyidx++;
            timeout = origtimeout;
        }

        if (timeout == 0) break;
        delay(1);
    }

    packetbuffer[replyidx] = 0; // null term

    if (!replyidx) // no data or timeout
        return 0;
    if (packetbuffer[0] != '!') // doesn't start with '!' packet beginning
        return 0;
}

```

```

// check checksum!
uint8_t xsum = 0;
uint8_t checksum = packetbuffer[replyidx-1];

for (uint8_t i=0; i<replyidx-1; i++) {
    xsum += packetbuffer[i];
}
xsum = ~xsum;

// Throw an error message if the checksum's don't match
if (xsum != checksum)
{
    Serial.print("Checksum mismatch in packet : ");
    printHex(packetbuffer, replyidx+1);
    return 0;
}

// checksum passed!
return replyidx;
}

```

Custom: HRM

The `BLEService` and `BLECharacteristic` classes can be used to implement any custom or officially adopted BLE service of characteristic using a set of basic properties and callback handlers.

The example below shows how to use these classes to implement the [Heart Rate Monitor](https://adafru.it/vaO) (<https://adafru.it/vaO>) service, as defined by the Bluetooth SIG.

HRM Service Definition

UUID: [0x180D](https://adafru.it/vaO) (<https://adafru.it/vaO>)

Characteristic Name	UUID	Requirement	Properties
Heart Rate Measurement	0x2A37	Mandatory	Notify
Body Sensor Location	0x2A38	Optional	Read
Heart Rate Control Point	0x2A39	Conditional	Write

Only the first characteristic is mandatory, but we will also implement the optional **Body Sensor Location** characteristic. Heart Rate Control Point won't be used in this example to keep things simple.

Implementing the HRM Service and Characteristics

The core service and the first two characteristics can be implemented with the following code:

First, define the `BLEService` and `BLECharacteristic` variables that will be used in your project:

```
/* HRM Service Definitions
 * Heart Rate Monitor Service: 0x180D
 * Heart Rate Measurement Char: 0x2A37
 * Body Sensor Location Char: 0x2A38
 */
BLEService hrms = BLEService(UUID16_SVC_HEART_RATE);
BLECharacteristic hrmc = BLECharacteristic(UUID16_CHR_HEART_RATE_MEASUREMENT);
BLECharacteristic bslc = BLECharacteristic(UUID16_CHR_BODY_SENSOR_LOCATION);
```

Then you need to 'populate' those variables with appropriate values. For simplicity sake, you can define a custom function for your service where all of the code is placed, and then just call this function once in the 'setup' function:

```
void setupHRM(void)
{
  // Configure the Heart Rate Monitor service
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.service.heart_rate.xml
  // Supported Characteristics:
  // Name                UUID      Requirement Properties
  // -----
  // Heart Rate Measurement 0x2A37  Mandatory   Notify
  // Body Sensor Location   0x2A38  Optional    Read
  // Heart Rate Control Point 0x2A39  Conditional  Write      <-- Not used here
  hrms.begin();

  // Note: You must call begin() on the BLEService before calling begin() on
```



```

// Note: you must call .begin() on the BLEService before calling .begin() on
// any characteristic(s) within that service definition.. Calling .begin() on
// a BLECharacteristic will cause it to be added to the last BLEService that
// was 'begin()'ed!

// Configure the Heart Rate Measurement characteristic
// See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.heart_rate_measurement.xml
// Permission = Notify
// Min Len    = 1
// Max Len    = 8
// B0         = UINT8 - Flag (MANDATORY)
// b5:7       = Reserved
// b4         = RR-Internal (0 = Not present, 1 = Present)
// b3         = Energy expended status (0 = Not present, 1 = Present)
// b1:2       = Sensor contact status (0+1 = Not supported, 2 = Supported but contact not detected, 3
= Supported and detected)
// b0         = Value format (0 = UINT8, 1 = UINT16)
// B1         = UINT8 - 8-bit heart rate measurement value in BPM
// B2:3       = UINT16 - 16-bit heart rate measurement value in BPM
// B4:5       = UINT16 - Energy expended in joules
// B6:7       = UINT16 - RR Internal (1/1024 second resolution)
hrmc.setProperties(CHR_PROPS_NOTIFY);
hrmc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
hrmc.setFixedLen(2);
hrmc.setCccdWriteCallback(cccd_callback); // Optionally capture CCCD updates
hrmc.begin();
uint8_t hrmdata[2] = { 0b00000110, 0x40 }; // Set the characteristic to use 8-bit values, with the
sensor connected and detected
hrmc.notify(hrmdata, 2); // Use .notify instead of .write!

// Configure the Body Sensor Location characteristic
// See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.body_sensor_location.xml
// Permission = Read
// Min Len    = 1
// Max Len    = 1
// B0         = UINT8 - Body Sensor Location
// 0          = Other
// 1          = Chest
// 2          = Wrist
// 3          = Finger
// 4          = Hand
// 5          = Ear Lobe
// 6          = Foot
// 7:255     = Reserved
bslc.setProperties(CHR_PROPS_READ);
bslc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
bslc.setFixedLen(1);
bslc.begin();
bslc.write8(2); // Set the characteristic to 'Wrist' (2)
}

```

Service + Characteristic Setup Code Analysis

1. The first thing to do is to call **.begin()** on the BLEService (**hrms** above). Since the UUID is set in the object declaration at the top of the sketch, there is normally nothing else to do with the BLEService instance.



You **MUST** call `.begin()` on the `BLEService` before adding any `BLECharacteristics`. Any `BLECharacteristic` will automatically be added to the last `BLEService` that was `.begin()`'ed!

2. Next, you can configure the **Heart Rate Measurement** characteristic (`hrmc` above). The values that you set for this will depend on the characteristic definition, but for convenience sake we've documented the key information in the comments in the code above.

- `'hrmc.setProperties(CHR_PROPS_NOTIFY);'` - This sets the `PROPERTIES` value for the characteristic, which determines how the characteristic can be accessed. In this case, the Bluetooth SIG has defined the characteristic as **Notify**, which means that the peripheral will receive a request ('notification') from the Central when the Central wants to receive data using this characteristic.
- `'hrmc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);'` - This sets the security for the characteristic, and should normally be set to the values used in this example.
- `'hrmc.setFixedLen(2);'` - This tells the Bluetooth stack how many bytes the characteristic contains (normally a value between 1 and 20). In this case, we will use a fixed size of two bytes, so we call `.setFixedLen`. If the characteristic has a variable length, you would need to set the max size via `.setMaxLen`.
- `'hrmc.setCccdWriteCallback(cccd_callback);'` - This optional code sets the callback that will be fired when the CCCD record is updated by the central. This is relevant because the characteristic is setup with the `NOTIFY` property. When the Central sets to 'Notify' bit, it will write to the CCCD record, and you can capture this write even in the CCCD callback and turn the sensor on, for example, allowing you to save power by only turning the sensor on (and back off) when it is or isn't actually being used. For the implementation of the CCCD callback handler, see the full sample code at the bottom of this page.
- `'hrmc.begin();'` - Once all of the properties have been set, you must call `.begin()` which will add the characteristic definition to the last `BLEService` that was `.begin()`'ed'.

3. Optionally set an initial value for the characteristic(s), such as the following code that populates `'hrmc'` with a correct values, indicating that we are providing 8-bit heart rate monitor values, that the Body Sensor Location characteristic is present, and setting the first heart rate value to `0x04`:



Note that we use `.notify()` in the example above instead of `.write()`, since this characteristic is setup with the `NOTIFY` property which needs to be handled in a slightly different manner than other characteristics.

```
// Set the characteristic to use 8-bit values, with the sensor connected and detected
uint8_t hrmdata[2] = { 0b00000110, 0x40 };

// Use .notify instead of .write!
hrmc.notify(hrmdata, 2);
```

The CCCD callback handler has the following signature:

```

void cccd_callback(uint16_t conn_hdl, BLECharacteristic* chr, uint16_t cccd_value)
{
    // Display the raw request packet
    Serial.print("CCCD Updated: ");
    //Serial.printBuffer(request->data, request->len);
    Serial.print(cccd_value);
    Serial.println("");

    // Check the characteristic this CCCD update is associated with in case
    // this handler is used for multiple CCCD records.
    if (chr->uuid == htmc.uuid) {
        if (chr->indicateEnabled(conn_hdl)) {
            Serial.println("Temperature Measurement 'Indicate' enabled");
        } else {
            Serial.println("Temperature Measurement 'Indicate' disabled");
        }
    }
}
}

```

4. Repeat the same procedure for any other BLECharacteristics in your service.

Full Sample Code

The full sample code for this example can be seen below:

```

/*****
This is an example for our nRF52 based Bluefruit LE modules

Pick one up today in the adafruit shop!

Adafruit invests time and resources providing this open source code,
please support Adafruit and open-source hardware by purchasing
products from Adafruit!

MIT license, check LICENSE for more information
All text above, and the splash screen below must be included in
any redistribution
*****/
#include <bluefruit.h>

/* HRM Service Definitions
 * Heart Rate Monitor Service: 0x180D
 * Heart Rate Measurement Char: 0x2A37
 * Body Sensor Location Char: 0x2A38
 */
BLEService hrms = BLEService(UUID16_SVC_HEART_RATE);
BLECharacteristic hrmc = BLECharacteristic(UUID16_CHR_HEART_RATE_MEASUREMENT);
BLECharacteristic bslc = BLECharacteristic(UUID16_CHR_BODY_SENSOR_LOCATION);

BLEDis bledis; // DIS (Device Information Service) helper class instance
BLEBas blebas; // BAS (Battery Service) helper class instance

uint8_t bps = 0;

void setup()
{
    Serial.begin(115200);

```

```

while ( !Serial ) delay(10);  // for nrf52840 with native usb

Serial.println("Bluefruit52 HRM Example");
Serial.println("-----\n");

// Initialise the Bluefruit module
Serial.println("Initialise the Bluefruit nRF52 module");
Bluefruit.begin();

// Set the advertised device name (keep it short!)
Serial.println("Setting Device Name to 'Feather52 HRM'");
Bluefruit.setName("Bluefruit52 HRM");

// Set the connect/disconnect callback handlers
Bluefruit.Periph.setConnectCallback(connect_callback);
Bluefruit.Periph.setDisconnectCallback(disconnect_callback);

// Configure and Start the Device Information Service
Serial.println("Configuring the Device Information Service");
bledis.setManufacturer("Adafruit Industries");
bledis.setModel("Bluefruit Feather52");
bledis.begin();

// Start the BLE Battery Service and set it to 100%
Serial.println("Configuring the Battery Service");
blebas.begin();
blebas.write(100);

// Setup the Heart Rate Monitor service using
// BLEService and BLECharacteristic classes
Serial.println("Configuring the Heart Rate Monitor Service");
setupHRM();

// Setup the advertising packet(s)
Serial.println("Setting up the advertising payload(s)");
startAdv();

Serial.println("Ready Player One!!!");
Serial.println("\nAdvertising");
}

void startAdv(void)
{
  // Advertising packet
  Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);
  Bluefruit.Advertising.addTxPower();

  // Include HRM Service UUID
  Bluefruit.Advertising.addService(hrms);

  // Include Name
  Bluefruit.Advertising.addName();

  /* Start Advertising
   * - Enable auto advertising if disconnected
   * - Interval:  fast mode = 20 ms, slow mode = 152.5 ms
   * - Timeout for fast mode is 30 seconds
   * - Start(timeout) with timeout = 0 will advertise forever (until connected)
   *
   * For recommended advertising interval
  */
}

```

```

    // Recommended advertising interval
    * https://developer.apple.com/library/content/qa/qa1931/_index.html
    */
    Bluefruit.Advertising.restartOnDisconnect(true);
    Bluefruit.Advertising.setInterval(32, 244);    // in unit of 0.625 ms
    Bluefruit.Advertising.setFastTimeout(30);      // number of seconds in fast mode
    Bluefruit.Advertising.start(0);                // 0 = Don't stop advertising after n seconds
}

void setupHRM(void)
{
    // Configure the Heart Rate Monitor service
    // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.service.heart_rate.xml
    // Supported Characteristics:
    // Name                                UUID      Requirement Properties
    // -----
    // Heart Rate Measurement              0x2A37    Mandatory    Notify
    // Body Sensor Location                 0x2A38    Optional     Read
    // Heart Rate Control Point             0x2A39    Conditional  Write        <-- Not used here
    hrms.begin();

    // Note: You must call .begin() on the BLEService before calling .begin() on
    // any characteristic(s) within that service definition.. Calling .begin() on
    // a BLECharacteristic will cause it to be added to the last BLEService that
    // was 'begin()'ed!

    // Configure the Heart Rate Measurement characteristic
    // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.heart_rate_measurement.xml
    // Properties = Notify
    // Min Len    = 1
    // Max Len    = 8
    // B0         = UINT8 - Flag (MANDATORY)
    // b5:7       = Reserved
    // b4         = RR-Interval (0 = Not present, 1 = Present)
    // b3         = Energy expended status (0 = Not present, 1 = Present)
    // b1:2       = Sensor contact status (0+1 = Not supported, 2 = Supported but contact not detected, 3
= Supported and detected)
    // b0         = Value format (0 = UINT8, 1 = UINT16)
    // B1         = UINT8 - 8-bit heart rate measurement value in BPM
    // B2:3       = UINT16 - 16-bit heart rate measurement value in BPM
    // B4:5       = UINT16 - Energy expended in joules
    // B6:7       = UINT16 - RR Interval (1/1024 second resolution)
    hrmc.setProperties(CHR_PROPS_NOTIFY);
    hrmc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
    hrmc.setFixedLen(2);
    hrmc.setCccdWriteCallback(cccd_callback); // Optionally capture CCCD updates
    hrmc.begin();
    uint8_t hrmdata[2] = { 0b00000110, 0x40 }; // Set the characteristic to use 8-bit values, with the
sensor connected and detected
    hrmc.write(hrmdata, 2);

    // Configure the Body Sensor Location characteristic
    // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.body_sensor_location.xml
    // Properties = Read
    // Min Len    = 1
    // Max Len    = 1
    // B0         = UINT8 - Body Sensor Location

```

```

//      0      = Other
//      1      = Chest
//      2      = Wrist
//      3      = Finger
//      4      = Hand
//      5      = Ear Lobe
//      6      = Foot
//      7:255 = Reserved
bslc.setProperties(CHR_PROPS_READ);
bslc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
bslc.setFixedLen(1);
bslc.begin();
bslc.write8(2);    // Set the characteristic to 'Wrist' (2)
}

void connect_callback(uint16_t conn_handle)
{
    // Get the reference to current connection
    BLEConnection* connection = Bluefruit.Connection(conn_handle);

    char central_name[32] = { 0 };
    connection->getPeerName(central_name, sizeof(central_name));

    Serial.print("Connected to ");
    Serial.println(central_name);
}

/**
 * Callback invoked when a connection is dropped
 * @param conn_handle connection where this event happens
 * @param reason is a BLE_HCI_STATUS_CODE which can be found in ble_hci.h
 */
void disconnect_callback(uint16_t conn_handle, uint8_t reason)
{
    (void) conn_handle;
    (void) reason;

    Serial.print("Disconnected, reason = 0x"); Serial.println(reason, HEX);
    Serial.println("Advertising!");
}

void cccd_callback(uint16_t conn_hdl, BLECharacteristic* chr, uint16_t cccd_value)
{
    // Display the raw request packet
    Serial.print("CCCD Updated: ");
    //Serial.printBuffer(request->data, request->len);
    Serial.print(cccd_value);
    Serial.println("");

    // Check the characteristic this CCCD update is associated with in case
    // this handler is used for multiple CCCD records.
    if (chr->uuid == hrmc.uuid) {
        if (chr->notifyEnabled(conn_hdl)) {
            Serial.println("Heart Rate Measurement 'Notify' enabled");
        } else {
            Serial.println("Heart Rate Measurement 'Notify' disabled");
        }
    }
}
}

```

```

void loop()
{
  digitalToggle(LED_RED);

  if ( Bluefruit.connected() ) {
    uint8_t hrmdata[2] = { 0b00000110, bps++ };          // Sensor connected, increment BPS value

    // Note: We use .notify instead of .write!
    // If it is connected but CCCD is not enabled
    // The characteristic's value is still updated although notification is not sent
    if ( hrmc.notify(hrmdata, sizeof(hrmdata)) ){
      Serial.print("Heart Rate Measurement updated to: "); Serial.println(bps);
    }else{
      Serial.println("ERROR: Notify not set in the CCCD or not connected!");
    }
  }

  // Only send update once per second
  delay(1000);
}

```

Bluefruit LE Connect



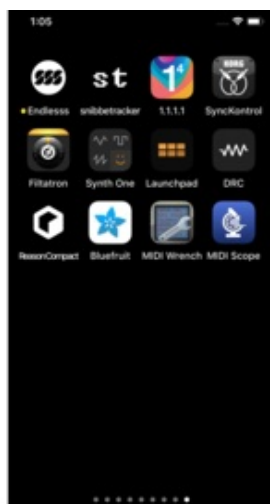
Adafruit Bluefruit LE Connect 
 Adafruit Industries
 ★★★★★ 4.1, 8 Ratings
 Free

Screenshots [iPhone](#) [iPad](#) [Apple Watch](#)



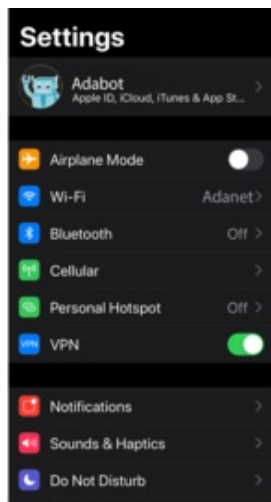
The **Bluefruit LE Connect** app provides iOS devices with a variety of tools to communicate with Bluefruit LE devices, such as the **Circuit Playground Bluefruit!** These tools cover basic communication and info reporting as well as more project specific uses such as remote button control and a NeoPixel color picker.

The iOS app is a [free download from Apple's App Store \(https://adafru.it/ddu\)](https://adafru.it/ddu). As of this writing, it requires iOS 11.3 or later and works on the **iPhone**, **iPad**, and **iPod Touch**.



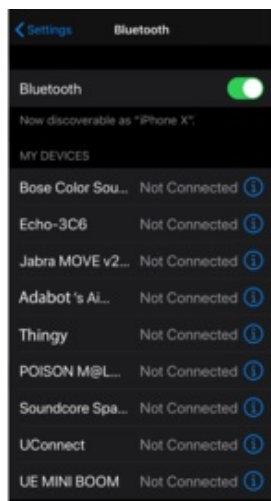
Install Bluefruit LE

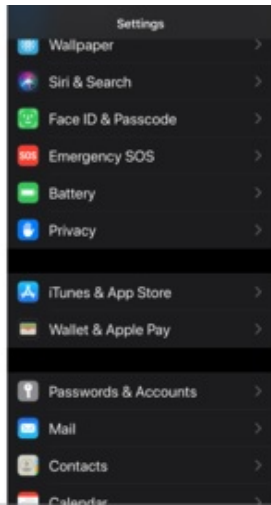
The first step is to install the app on your device.



Enable Bluetooth

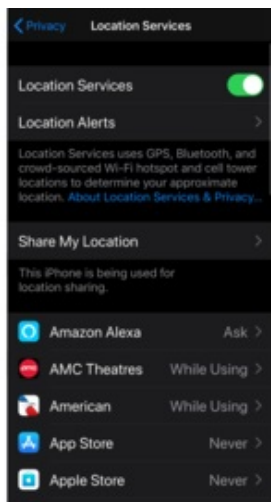
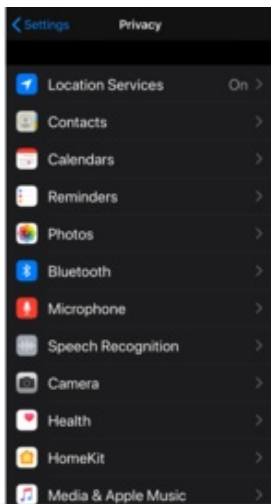
If Bluetooth is disabled on your device, enable it by going to **Setting > Bluetooth** on your iOS device and then turning it on.





Enable Location Services

If you plan to use the app to send location/GPS data to Bluefruit LE, enable Location Services. Enable it on iOS using **Settings->Privacy->Location Services**.



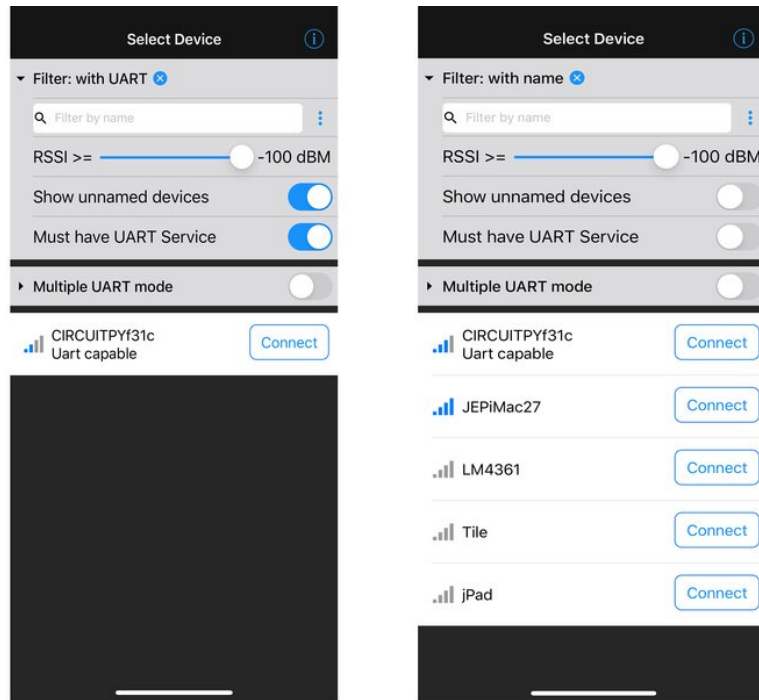
Scan for Devices

Launch the app now -- it will automatically begin to scan the airwaves for Bluetooth LE devices. These are presented in a list at the bottom of the page.

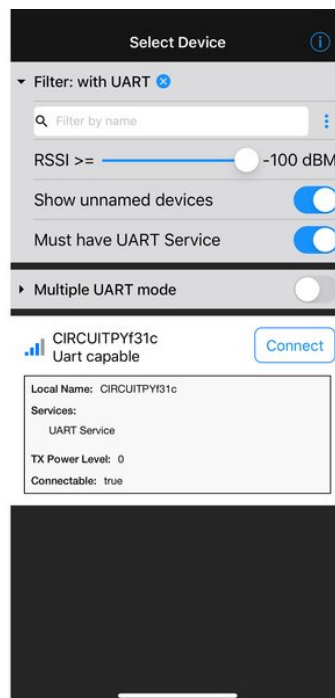
Notice, you can use the **Must have UART Service** filter to prevent BLE devices from showing up that can't work with

the app.

- To refresh the list and start a new scan, simply swipe down on the current list.
- Each device's signal strength is displayed in the left side of its row.



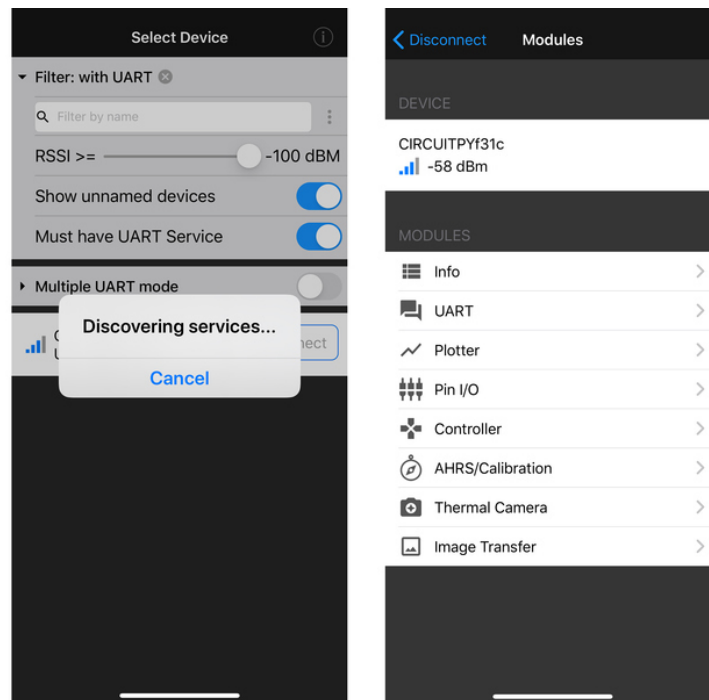
If you tap on the device entry (not on **Connect**), you'll see more detail about a particular device:



Connect

Tap the **Connect** button on the UART capable device you wish to use. The app will connect to the Circuit Playground

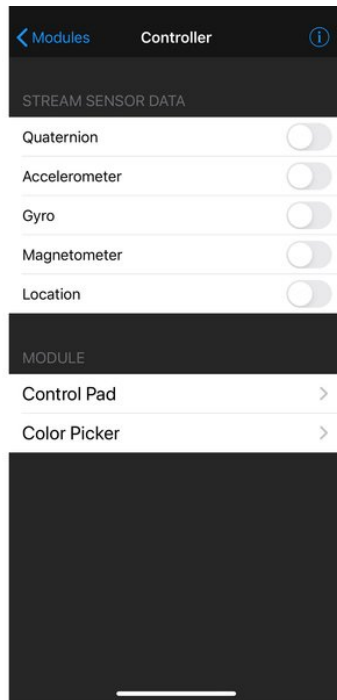
Bluefruit! Now, you'll be presented with the Device name and signal strength, and a number of different Modules you can use.



Controller Module

Click on the **Controller** module. You'll see a number of different sensor data streaming options. Enabling these will allow you to send data from your phone, such as the **Accelerometer** data or **Location** data, directly to your Circuit Playground Bluefruit!

The two modules on this page that can send data to the Circuit Playground Bluefruit are the **Control Pad** and **Color Picker**.

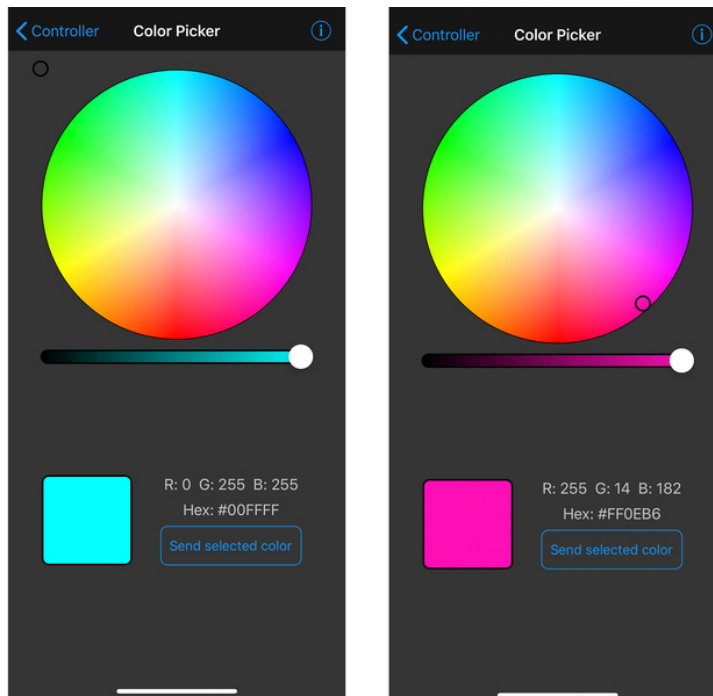


Color Picker

Click on the **Color Picker**. Now, you can dial in the hue, saturation, and value of a color using the color wheel and value slider.

Follow this page (<https://adafru.it/GcO>) for setting up the CPB with the color picker code.

Press the **Send selected color** button and your color values will be sent to the Circuit Playground Bluefruit to adjust its NeoPixels!



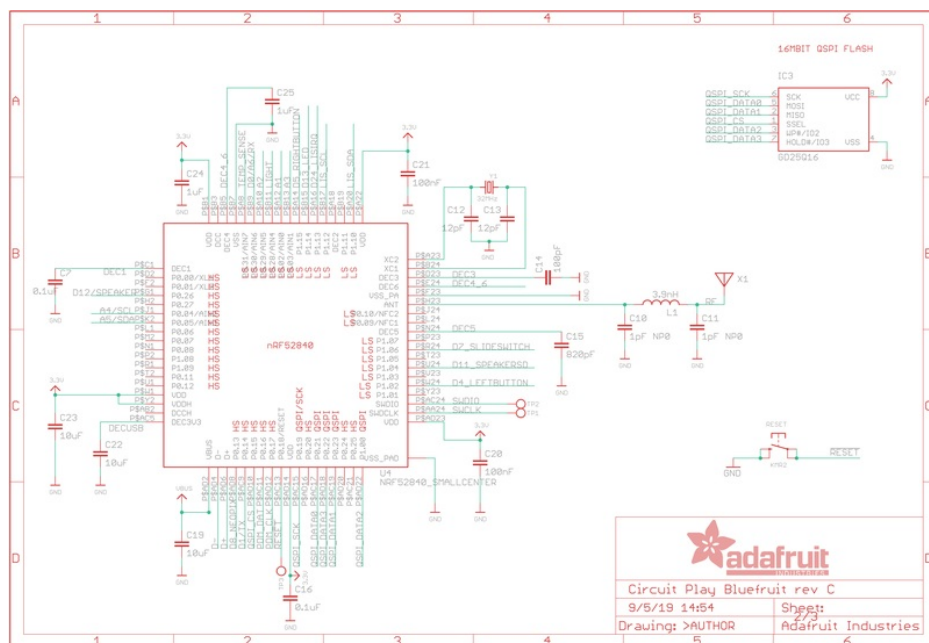
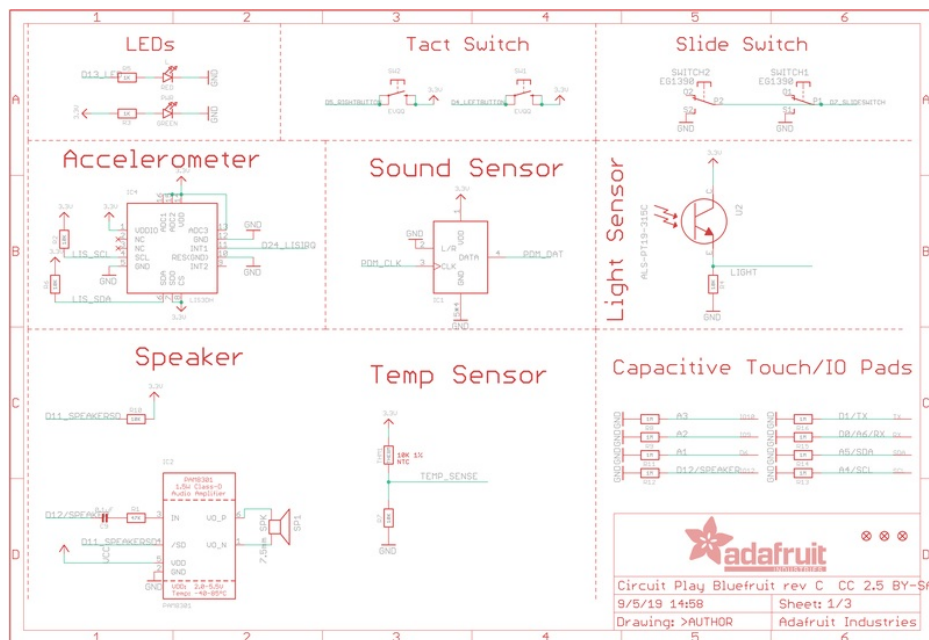
The app provides many other features with the additional modules. Have a look at the [Bluefruit LE Connect for iOS and Android standalone guide \(https://adafru.it/GcP\)](https://adafru.it/GcP) for an explanation of each feature.

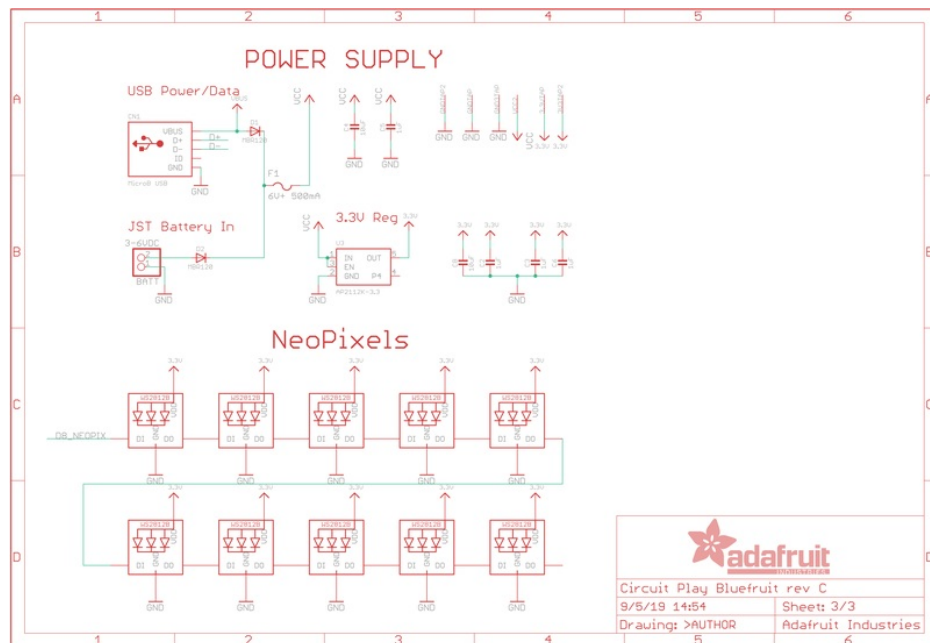
Downloads

Files:

- Datasheet for Nordic nRF52840 (<https://adafru.it/FKw>)
- Nordic InfoCenter for further documentation (<https://adafru.it/FKx>)
- EagleCAD files for Circuit Playground Bluefruit on GitHub (<https://adafru.it/FKy>)
- 3D Models on GitHub (<https://adafru.it/G4D>)
- Fritzing object in the Adafruit Fritzing Library (<https://adafru.it/FKz>)

Schematic for Circuit Playground Bluefruit





Fab print of Circuit Playground Bluefruit



