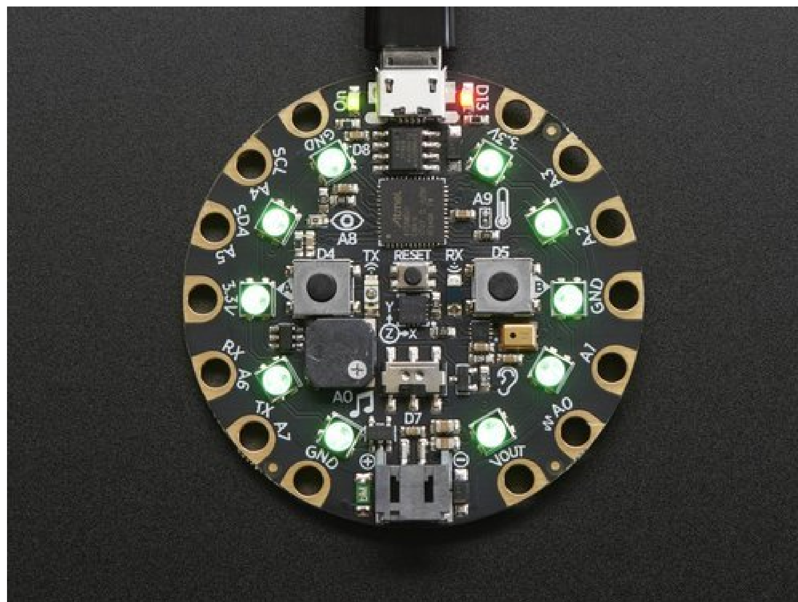
Adafruit Circuit Playground Express

Created by lady ada



Last updated on 2021-08-19 10:42:50 AM EDT

Guide Contents

Guide Contents 2
Overview 9
Classic vs. Express 12
How to tell if you have a Classic 12
How to tell if you have an Express 12
Guided Tour 14
Power and Data 14
- Micro B USB connector 14
- JST Battery Input 15
Alligator/Croc Clip Pads 15
Microchips 15
LEDs 16
- Green ON LED 16
- Red #13 LED 16
- 10 x Color NeoPixel LED 17
Speaker 17
Sensors 18
- Light Sensor 19
- Temperature Sensor 19
- Microphone Audio Sensor 20
- Motion Sensor 20
- Capacitive Touch 20
Switches & Buttons 21
Infrared Receive/Transmit & Proximity 21
Pinouts 23
Power Pads 23
Input/Output Pads 24
- Common to all pads 25
Each Pin! 25
Internally Used Pins! 26
Update the Bootloader (Mac Users Especially!) 27
- Updating Your Circuit Playground Express Bootloader (see below for other boards) 27
- Oh no, I updated MacOS already and I can't see CPLAYBOOT! 29
- Upgrading Other Boards 29
Windows Driver Installation 30
Manual Driver Installation 32
Code.org CSD 33
Step 1. Connect to USB 33
Step 2. Press RESET to get into bootloader mode 33
Step 3. Copy over Firmata firmware 34
MakeCode 36
What is MakeCode? 37
Editing Blocks 39
- Blinky! 39
Downloading and Flashing 41
- Step 1: Bootloader mode 41

Step 2: Compile and Download 42
Running MakeCode that's Already Loaded 44
Saving and Sharing 45
Extracting your code from the Circuit Playground 45
Sharing 45
Editing JavaScript 47
Apps 48
Windows Store 48
Node.JS 48
Other Good Stuff 49
GitHub packages 49
We are Open Source on GitHub 49
We have crowd-sourced translations 49
MakeCode and Windows 10 50
The MakeCode App for Windows 10 50
Installation 50
See More MakeCode Projects 52
What is CircuitPython? 53
CircuitPython is based on Python 53
Why would I use CircuitPython? 53
CircuitPython 55
Install or update CircuitPython! 55
Further Information 57
Installing Mu Editor 58
Download and Install Mu 58
Using Mu 58
Creating and Editing Code 60
Creating Code 60
Editing Code 62
Your code changes are run as soon as the file is done saving. 63
1. Use an editor that writes out the file completely when you save it. 63
2. Eject or Sync the Drive After Writing 64
Oh No I Did Something Wrong and Now The CIRCUITPY Drive Doesn't Show Up!!! 64
Back to Editing Code... 65
Exploring Your First CircuitPython Program 66
Imports & Libraries 66
Setting Up The LED 66
Loop-de-loops 67
What Happens When My Code Finishes Running? 67
What if I don't have the loop? 68
More Changes 69
Naming Your Program File 69
Connecting to the Serial Console 70
Are you using Mu? 70
Setting Permissions on Linux 71
Using Something Else? 72
Interacting with the Serial Console 73
The REPL 77
Returning to the serial console 80
CircuitPython Libraries 81

Installing the CircuitPython Library Bundle 82
Example Files 83
Copying Libraries to Your Board 84
Example: ImportError Due to Missing Library 84
Library Install on Non-Express Boards 85
Updating CircuitPython Libraries/Examples 86
Frequently Asked Questions 87
I have to continue using an older version of CircuitPython; where can I find compatible libraries? 87
Is ESP8266 or ESP32 supported in CircuitPython? Why not? 87
How do I connect to the Internet with CircuitPython? 88
Is there asyncio support in CircuitPython? 89
My RGB NeoPixel/DotStar LED is blinking funny colors - what does it mean? 90
What is a MemoryError? 91
What do I do when I encounter a MemoryError? 91
Can the order of my import statements affect memory? 92
How can I create my own .mpy files? 92
How do I check how much memory I have free? 92
Does CircuitPython support interrupts? 92
Does Feather M0 support WINC1500? 93
Can AVRs such as ATmega328 or ATmega2560 run CircuitPython? 93
Commonly Used Acronyms 93
Troubleshooting 94
Always Run the Latest Version of CircuitPython and Libraries 94
I have to continue using CircuitPython 5.x, 4.x, 3.x or 2.x, where can I find compatible libraries? 94
CPLAYBOOT, TRINKETBOOT, FEATHERBOOT, or GEMMABOOT Drive Not Present 94
You may have a different board. 95
MakeCode 95
MacOS 95
Windows 10 95
Windows 7 or 8.1 95
Windows Explorer Locks Up When Accessing boardnameBOOT Drive 96
Copying UF2 to boardnameBOOT Drive Hangs at 0% Copied 97
CIRCUITPY Drive Does Not Appear 97
Windows 7 and 8.1 Problems 97
Serial Console in Mu Not Displaying Anything 97
CircuitPython RGB Status Light 98
ValueError: Incompatible .mpy file. 99
CIRCUITPY Drive Issues 99
Easiest Way: Use storage.erase_filesystem() 100
Old Way: For the Circuit Playground Express, Feather M0 Express, and Metro M0 Express: 100
Old Way: For Non-Express Boards with a UF2 bootloader (Gemma M0, Trinket M0): 102
Old Way: For non-Express Boards without a UF2 bootloader (Feather M0 Basic Proto, Feather Adalogger, Arduino Zero): 102
Running Out of File Space on Non-Express Boards 102
Delete something! 102
Use tabs 103
MacOS loves to add extra files. 103
Prevent & Remove MacOS Hidden Files 103
Copy Files on MacOS Without Creating Hidden Files 104
Other MacOS Space-Saving Tips 105
Device locked up or boot looping 106

Uninstalling CircuitPython 107
- Backup Your Code 107

Moving Circuit Playground Express to MakeCode 107

Moving to Arduino 108

Welcome to the Community! 111
- Adafruit Discord 111
- Adafruit Forums 112
- Adafruit Github 113
- ReadTheDocs 114

CircuitPython Made Easy 116

CircuitPython Playground 117

CircuitPython Pins and Modules 118

CircuitPython Pins 118
- import board 118
- I2C, SPI, and UART 119
- What Are All the Available Names? 120
- Microcontroller Pin Names 121

CircuitPython Built-In Modules 122

CircuitPython Built-Ins 123

Thing That Are Built In and Work 123
- Flow Control 123
- Math 123
- Tuples, Lists, Arrays, and Dictionaries 123
- Classes, Objects and Functions 123
- Lambdas 123
- Random Numbers 124

CircuitPython Digital In & Out 125
- Going Beyond the Lesson! 126
  - Experiment 1 126
  - Experiment 2 127

CircuitPython Analog In 128
- Creating an Analog Input 129
- GetVoltage Helper 129
- Main Loop 129

CircuitPython Analog Out 131
- Creating an analog output 131
- Setting the analog output 131
- Main Loop 131
- Find the pin 132

CircuitPython PWM 136
- PWM with Fixed Frequency 136
- Create a PWM Output 137
- Main Loop 137
- PWM Output with Variable Frequency 138
- Wire it up 139
- Where's My PWM? 143

CircuitPython Servo 145
- Servo Wiring 145
- Standard Servo Code 148
- Continuous Servo Code 148

CircuitPython Audio Out 150

Basic Tones 151

Playing Audio Files 152
CircuitPython Cap Touch 156
- Creating an capacitive touch input 157
- Main Loop 157
- Capacitive Touch and the Audio Pin on Circuit Playground Bluefruit 160

CircuitPython NeoPixel 161
CircuitPython DotStar 165
- Wire It Up 165
- The Code 166
- Create the LED 169
- DotStar Helpers 169
- Main Loop 170
- Is it SPI? 170
- Read the Docs 171

CircuitPython UART Serial 172
- The Code 173
- Wire It Up 174
- Where's my UART? 177
- Trinket M0: Create UART before I2C 178

CircuitPython I2C 180
- Wire It Up 180
- Find Your Sensor 183
- I2C Sensor Data 184
- Where's my I2C? 185

CircuitPython HID Keyboard 187
CircuitPython HID Keyboard and Mouse 190
- CircuitPython Keyboard Emulator 190
- Create the Objects and Variables 192
- The Main Loop 192

CircuitPython Mouse Emulator 193
- Create the Objects and Variables 195
- CircuitPython HID Mouse Helpers 195
- Main Loop 196

CircuitPython CPU Temp 197
CircuitPython Storage 198
- Logging the Temperature 200

CircuitPython Expectations 203
Always Run the Latest Version of CircuitPython and Libraries 203
I have to continue using CircuitPython 3.x or 2.x, where can I find compatible libraries? 203
Switching Between CircuitPython and Arduino 203
The Difference Between Express And Non-Express Boards 204
Non-Express Boards: Gemma, Trinket, and QT Py 204
- Small Disk Space 204
- No Audio or NVM 204

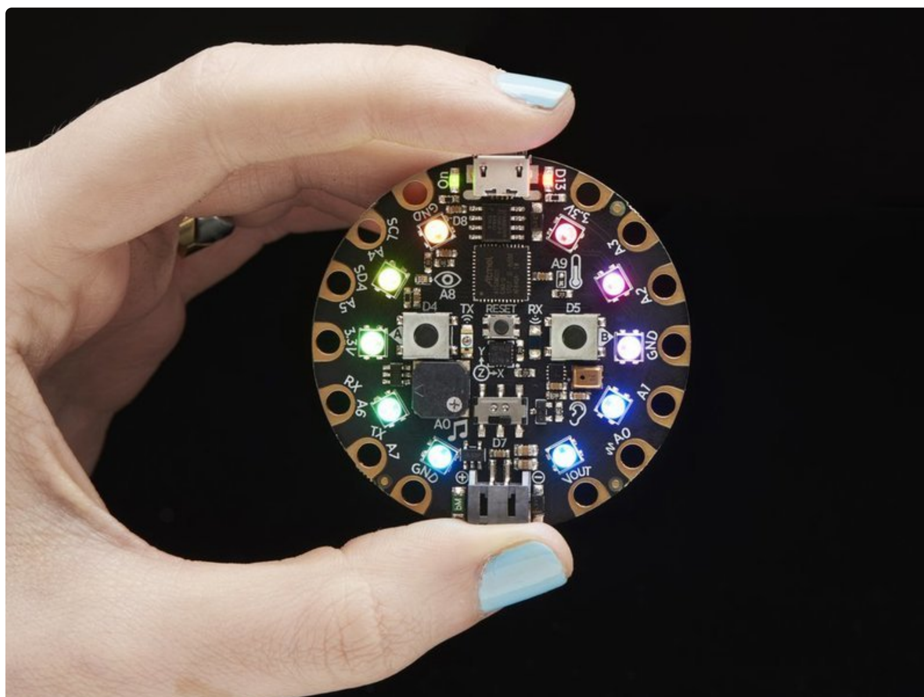
Differences Between CircuitPython and MicroPython 204
Differences Between CircuitPython and Python 205
- Python Libraries 205
- Integers in CircuitPython 205
- Floating Point Numbers and Digits of Precision for Floats in CircuitPython 205
- Differences between MicroPython and Python 205

Playground Light Sensor 206

Playground Temperature 208
Playground Drum Machine 210
Playground Sound Meter 214
Arduino 218
Set Up Arduino IDE 219
Install SAMD Support 219
Install Drivers (Windows 7 Only) 220
Blink 220
Sucessful Upload 221
Compilation Issues 222
Manually bootloading 222
Ubuntu & Linux Issue Fix 223
Arduino Switches 224
- Without Library Assist 224
Adapting Sketches to M0 & M4 226
Analog References 226
Pin Outputs & Pullups 226
Serial vs SerialUSB 226
AnalogWrite / PWM on Feather/Metro M0 227
analogWrite() PWM range 228
analogWrite() DAC on A0 228
Missing header files 228
Bootloader Launching 229
Aligned Memory Access 229
Floating Point Conversion 229
How Much RAM Available? 230
Storing data in FLASH 230
Pretty-Printing out registers 230
M4 Performance Options 231
- CPU Speed (overclocking) 231
- Optimize 232
- Cache 232
- Max SPI and Max QSPI 232
Enabling the Buck Converter on some M4 Boards 233
Troubleshooting 234
- Using external NeoPixel strips/dots with Circuit Playground in Arduino 234
- CPLAYBOOT Does not Appear on Windows 10; "ARM7TDMI" Error in Arduino 234
Accessories 236
Packs 236
Connections 236
Capacitive Touch 237
Robotics 238
Sewable 239
Power 240
UF2 Bootloader Details 243
Entering Bootloader Mode 244

Using the Mass Storage Bootloader 246
Using the BOSSA Bootloader 247
- Windows 7 Drivers 247
- Verifying Serial Port in Device Manager 248
- Running bossac on the command line 250
  - Using bossac Versions 1.7.0, 1.8 250
  - Using bossac Versions 1.9 or Later 250
Updating the bootloader 251
Getting Rid of Windows Pop-ups 252
Making your own UF2 253
Installing the bootloader on a fresh/bricked board 254
Downloads 255
- Files: 255
Circuit Playground Express Schematic 255

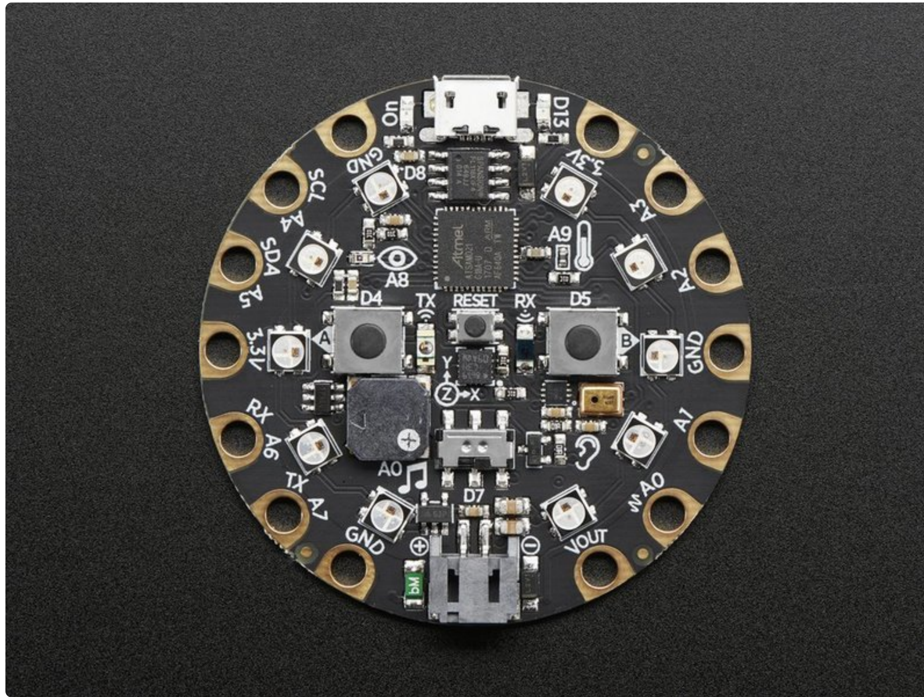
Overview



Circuit Playground Express is the next step towards a perfect introduction to electronics and programming. We've taken the original Circuit Playground Classic and made it even better! Not only did we pack even more sensors in, we also made it even easier to program.

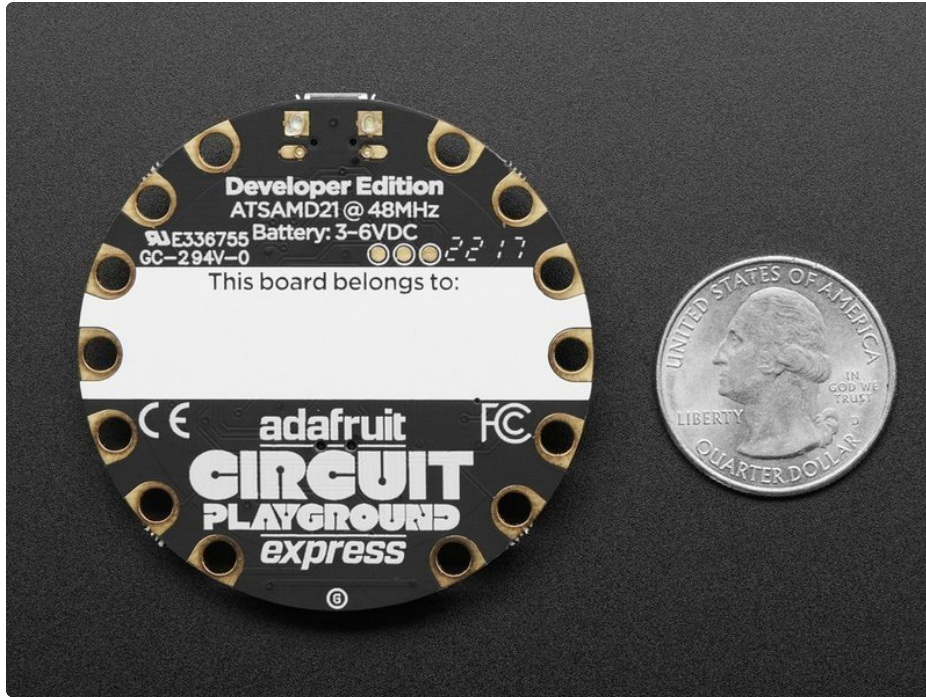
Start your journey with **Microsoft MakeCode** block-based or Javascript programming (https://adafru.it/wWd). Or, you can follow along with code.org CS Discoveries (https://adafru.it/BVX). Then, you can use the same board to try CircuitPython (https://adafru.it/BeW), with the Python interpreter running right on the Express. As you progress, you can advance to using **Arduino IDE**, which has full support of all the hardware down to the low level, so you can make powerful projects.

Because you can program the same board in 3 different ways - the Express has great value and re-usability. From beginners to experts, Circuit Playground Express has something for everyone.



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

- 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
- Infrared receiver and transmitter - can receive and transmit any remote control codes, as well as send messages between Circuit Playground Expresses. Can also act as a proximity sensor.
- 8 x alligator-clip friendly input/output pins
- Includes I2C, UART, 8 pins that can do analog inputs, multiple PWM output
- 7 pads can act as capacitive touch inputs and the 1 remaining is a true analog output
- Green "ON" LED so you know its powered
- Red "#13" LED for basic blinking
- Reset button
- ATSAMD21 ARM Cortex M0 Processor, running at 3.3V and 48MHz
- 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!


Developer Edition
ATSAMD21 @ 48MHz
E336755
Battery: 3-6VDC
GC-294V-0
This board belongs to:
CE
FC
adafruit
CIRCUIT
PLAYGROUND
express
UNITED STATES OF AMERICA
LIBERTY
IN GOD WE TRUST
QUARTER DOLLAR

Classic vs. Express

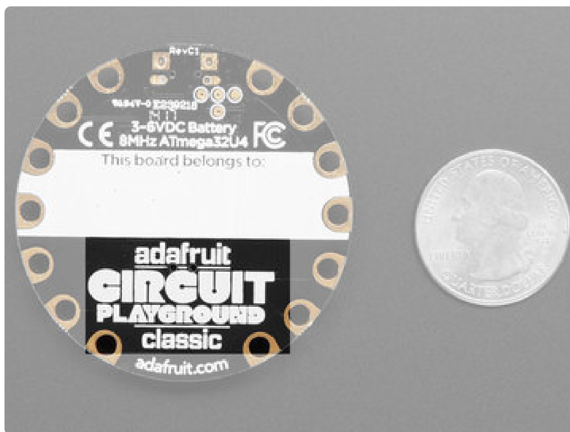
Circuit Playground started its life as a board with simple requirements - just work with Arduino IDE and Code.org. But since it's initial launch in 2015 we've learned a lot and improved the board greatly!

There are **TWO** Circuit Playgrounds - one **Classic** and one **Express**.

The Classic version can run Arduino and Code.org

The Express version can run MakeCode, CircuitPython, Arduino, and Code.org CS Discoveries.

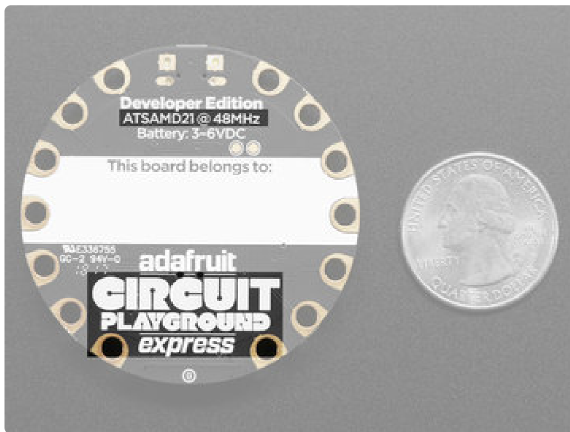
How to tell if you have a Classic



Current Circuit Playground Classic boards have **Classic** written on the back lower half

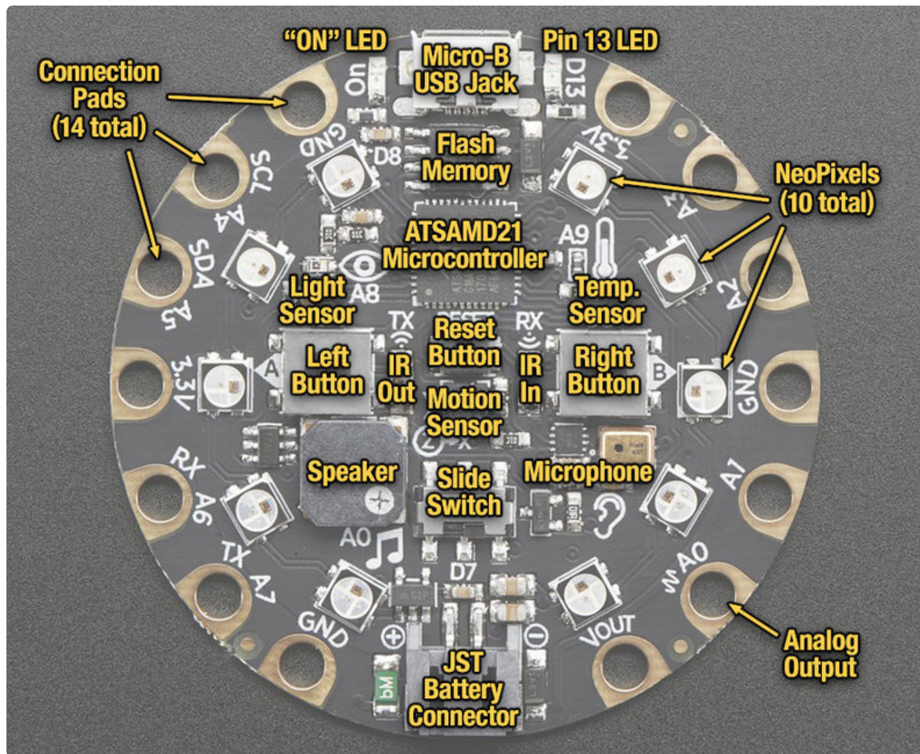
Older Circuit Playground Classics didn't have the word **Classic** on the back, but they do have text that says the chip type, **ATmega32U4**

How to tell if you have an Express



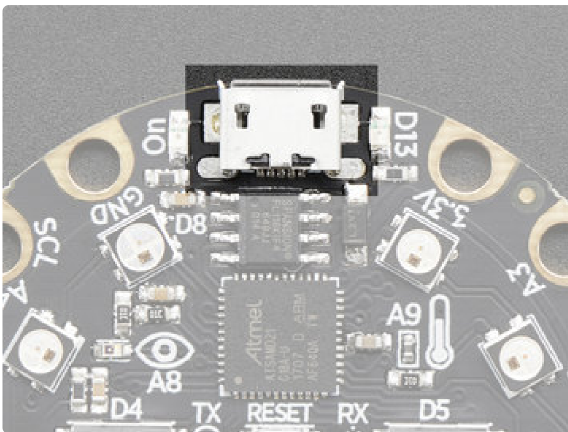
Express boards have **CIRCUIT PLAYGROUND EXPRESS** on the back lower half. They'll also note the chip is an **ATSAMD21**

Guided Tour



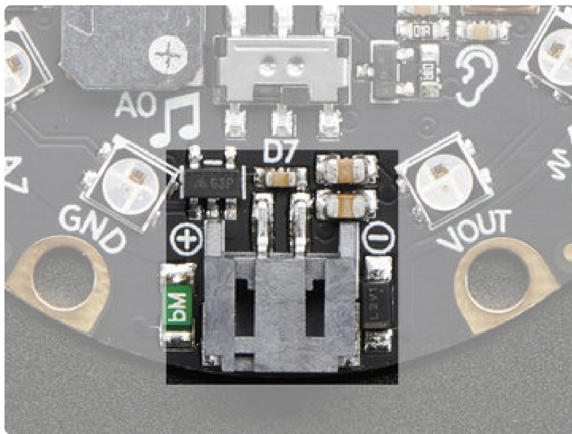
Let me take you on a tour of your Circuit Playground Express (we'll shorten that to **CPX**). Each CPX is assembled here at Adafruit and comes chock-full of good design to make it a joy to use.

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.

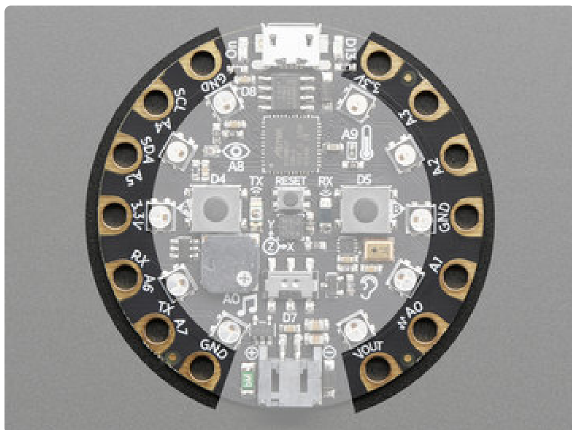


JST Battery Input

This is at the bottom of the board. You can take your CPX 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!

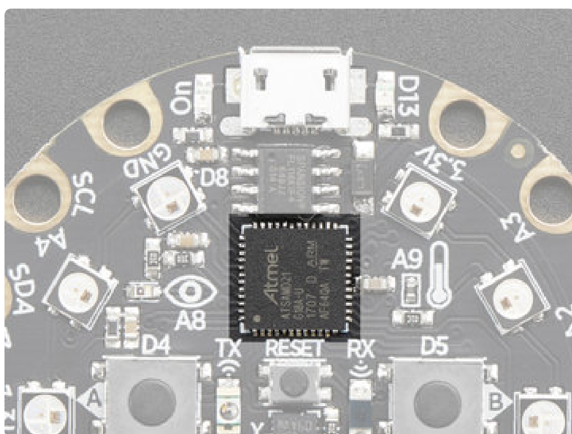


Of the 14 pads, you get a wide range of power pins, I2C, UART, Analog In, Digital In/Out, PWM, and Analog Out.

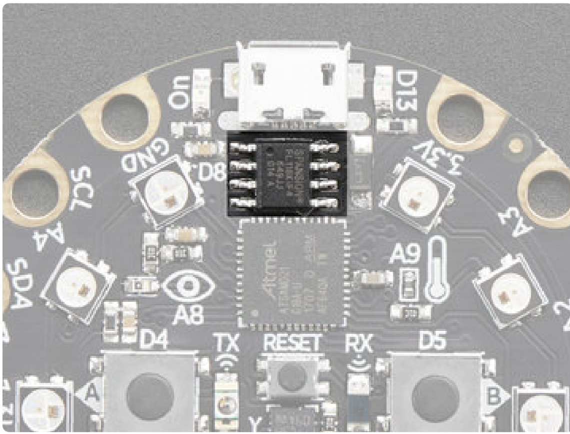
Some of them can even sense the touch of your finger!

See the next pinouts page for more details!

Microchips

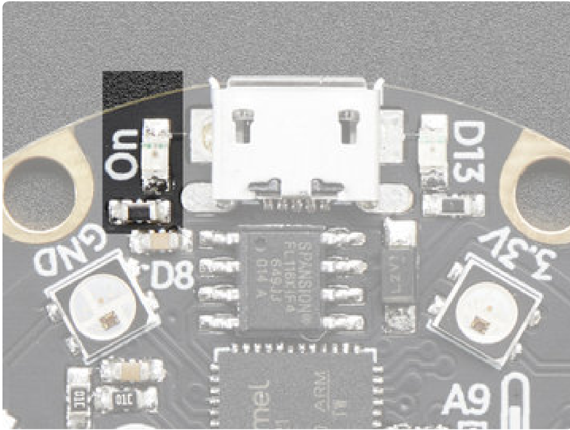


The **brains** of the operation here is the **ATSAMD21G18** a Cortex M0 microcontroller. This chip is much more powerful than the original Circuit Playground. It sits in the top center, and is what you use to run MakeCode, CircuitPython or Arduino!



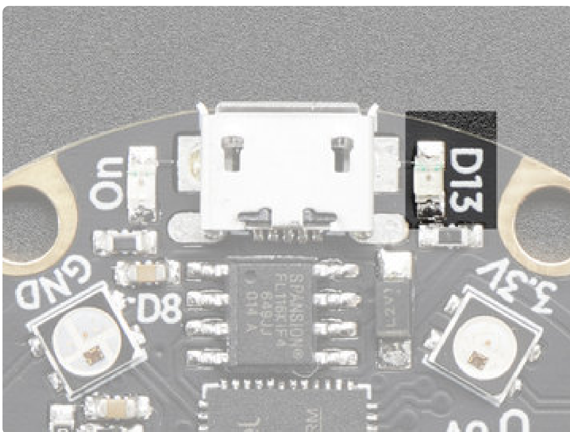
New in Circuit Playground Express, we have added a new storage chip, called SPI Flash. This is a very, very small disk drive, only 2 MB large. You can use this in Arduino or CircuitPython to store files. In CircuitPython this is where all your code lives, and what you see when you use the **CIRCUITPY**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

LEDs



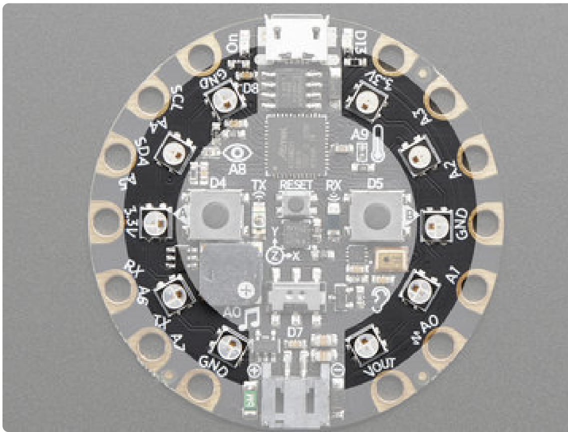
Green ON LED

To the left of the USB connector. This LED lets you know that the CPX 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 CPX 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.



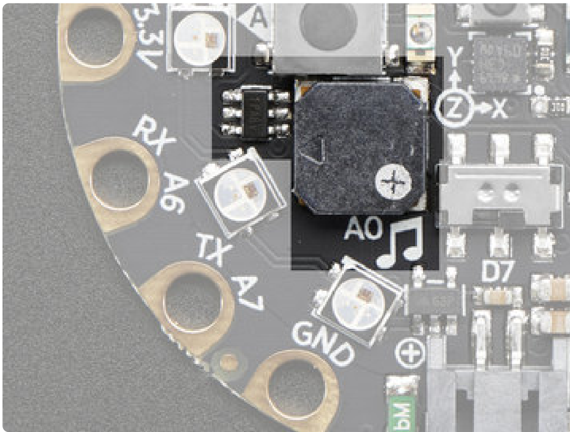
10 x Color NeoPixel LED

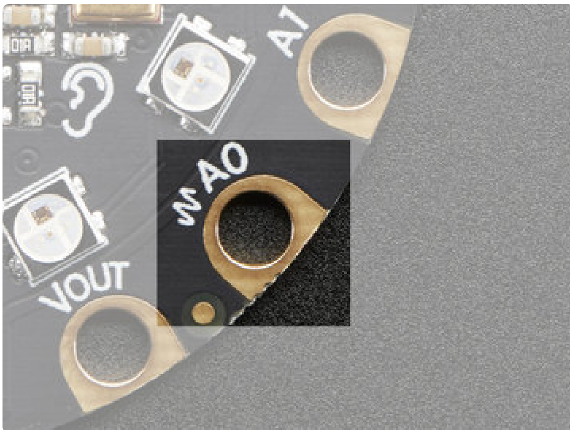
The ten LEDs surrounding the outer edge of the boards are all full color, RGB LEDs, each one can be set to any color in the rainbow. Great for beautiful lighting effects! The NeoPixels will also help you know when the bootloader is running (they will turn green) or if it failed to initialize USB when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

Speaker

We have upgraded the buzzer from the original Circuit Playground to be a mini speaker with a metal disc, it's not going to compete with your HiFi stereo, but it can play simple songs and tones

> The speaker is small and designed for small beeps and alerts, it isn't going to be very loud and you may hear a metallic ticking when driving the speaker (there's a metal disk inside). It's best used for beeps and tones - if you need better audio, connect a headphone or external amplified speaker to A0





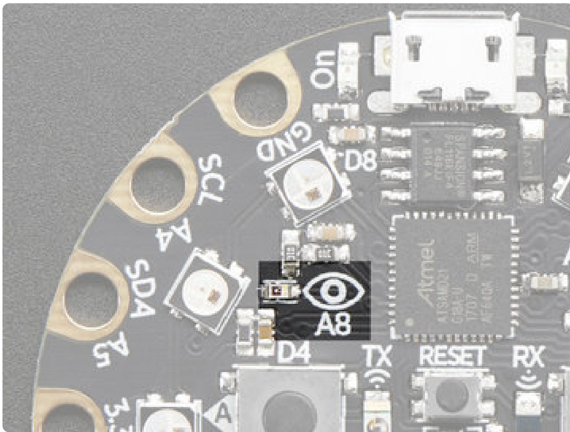
The speaker is the squarish gray chunk on the bottom left of the board. There is a small class D amplifier connected to the speaker that can be enabled or disabled. Note: it won't sound good if too loud, so some experimentation may be necessary

The amplifier is connected to the true analog output **A0** pin -- this pin is also available on one of the connection pads in the lower right. You can tell it's the analog output pin because of the squiggly symbol.

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 Express has a large number of sensor **inputs** that let you add all sorts of interactivity to your project.

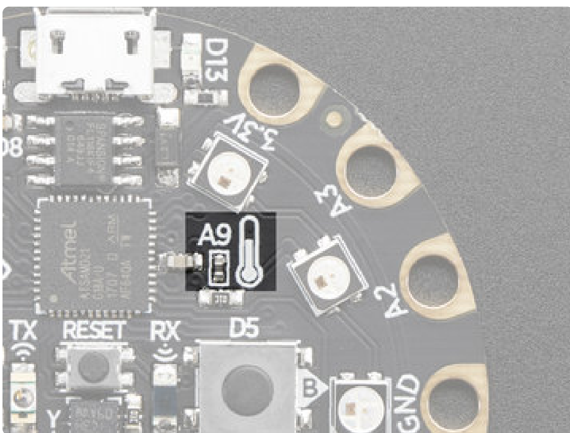


Light Sensor

There is an analog light sensor, part number ALS-PT19 (https://adafru.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.

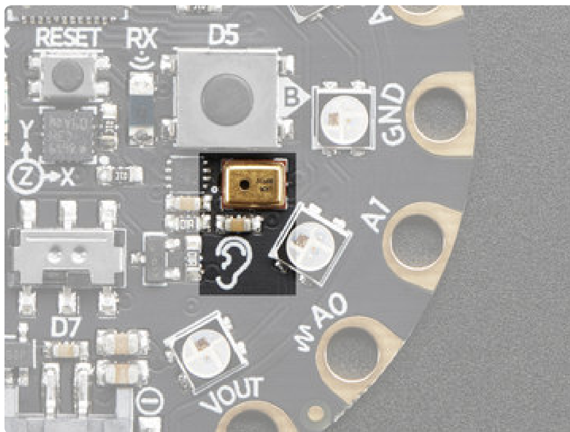
This sensor is connect to analog pin **A8** and will read between 0 and 1023 with higher values corresponding to higher light levels. A reading of about 300 is common for most indoor light levels.

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



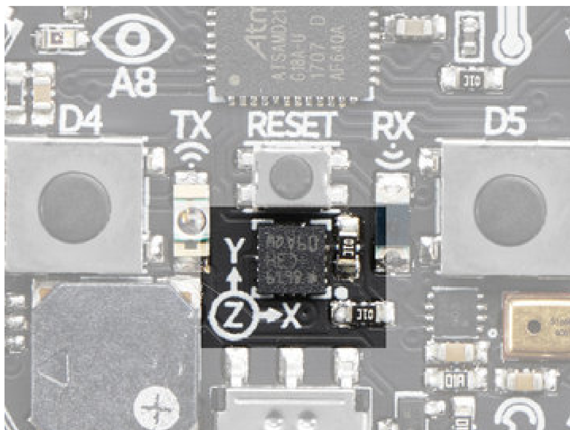
Temperature Sensor

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

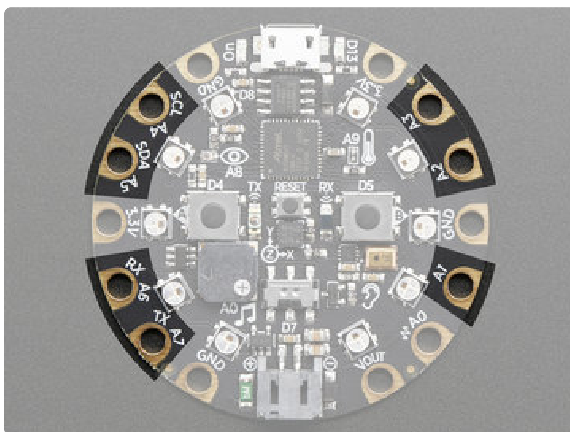
A MEMS microphone can be used to detect audio levels and even perform basic FFT functions. Instead of an analog microphone, that requires an external op-amp and level management, we've decided to go with a PDM microphone. This is a digital mic, and is a lot smaller and less expensive! You will have to use the MakeCode/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 **#36**

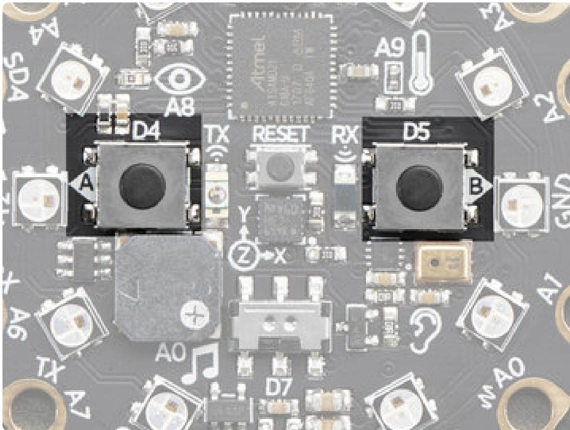


Capacitive Touch

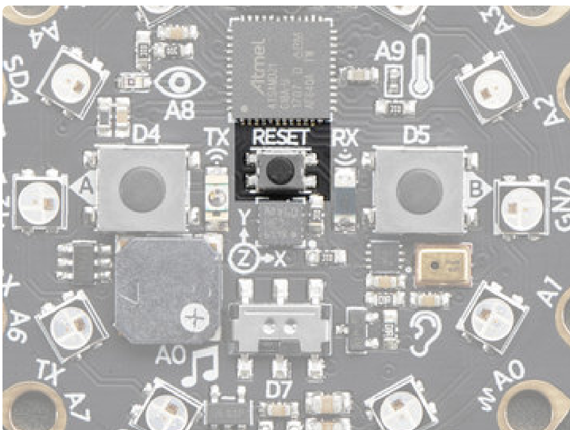
The chip in the CPX has a built in ability to perform capacitive touch readings. This is a great way to sense human touch without additional components. Even animals will work if its directly touching their skin!

On the Express you get **seven** capacitive touch pads. All pads but **A0** can do it. (**A0** is used for the audio output pad so cannot do captouch). Since we use the chip's built-in hardware support, you will use the provided MakeCode/Arduino/CircuitPython library code to perform the readings

Switches & Buttons



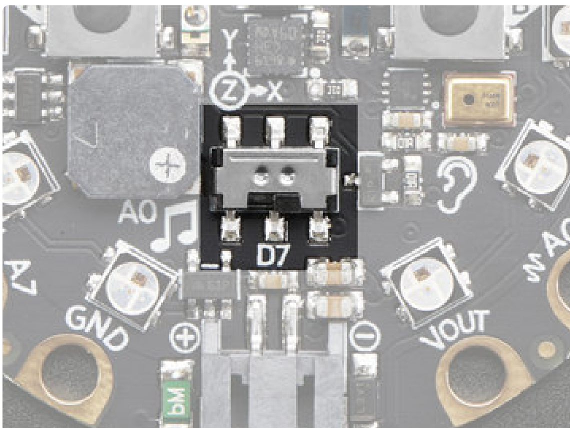
There are two large **A** and **B** buttons, connected to digital **#4** (Left) and **#5** (Right) each. These are unconnected when not pressed, and connected to 3.3V when pressed, so they read HIGH. Set the pins **#4** and **#5** to use an internal pull-down resistor when reading these pins so they 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 CPX.

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

If using MakeCode, press this button *twice* to reset, once to enter the bootloader manually.



There is a single slide switch near the center bottom of the Circuit Playground Express. It is connected to digital **#7**. The switch is unconnected when slid to the left and connected to ground when slid to the right. Set pin #7 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

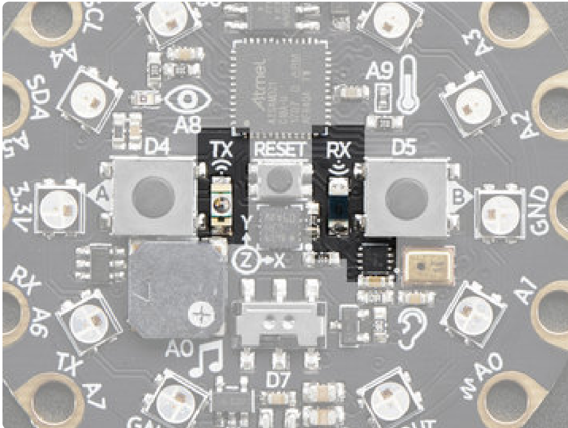
Note that you need to use an internal *pull-up* for the slide switch, but an internal *pull-down* for the pushbuttons.

Infrared Receive/Transmit & Proximity

New in Circuit Playground Express, we've added IR receive and transmit. This means you can communicate with TVs & other household devices to control them. You can also send commands to the CPX with a remote control. And, finally, you can send simple messages **between** multiple Circuit

Playgrounds!

By bouncing IR light off of items and reading the reflected light, you can also do very simple proximity sensing!

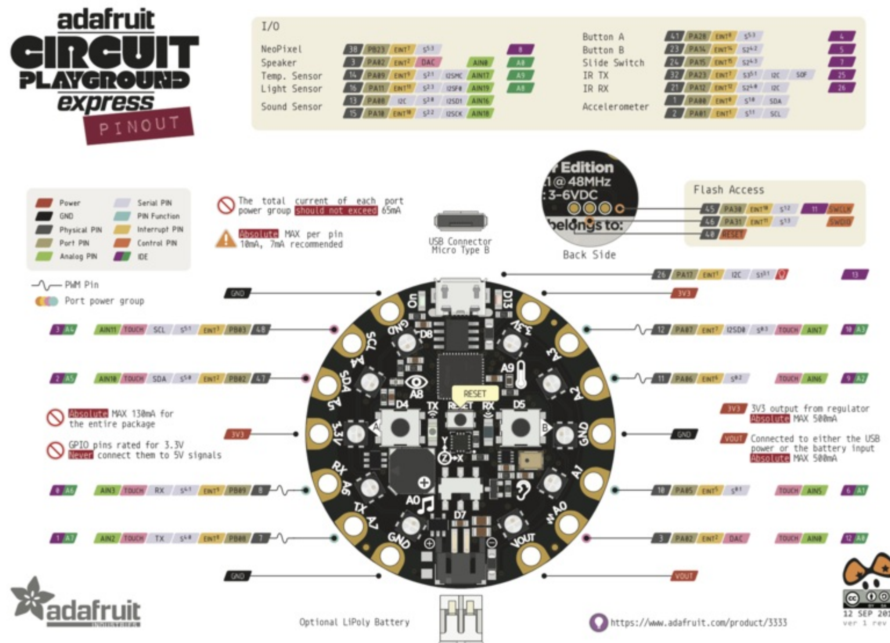


There's two elements for infrared. The **Transmitter** is on the left and is a big clear LED, this is connected via a transistor to pin **#29**

On the right is the darker **receiver** LED, which also has a decoder chip. This chip will receive the 38KHz signals and demodulate them for you. The demodulated output is available on pin **#39**

There's also a *secret* capability, you can read the 'raw' analog value from the receiver LED diode to do basic IR proximity sensing. The direct analog value is available from pin **A10**

Pinouts

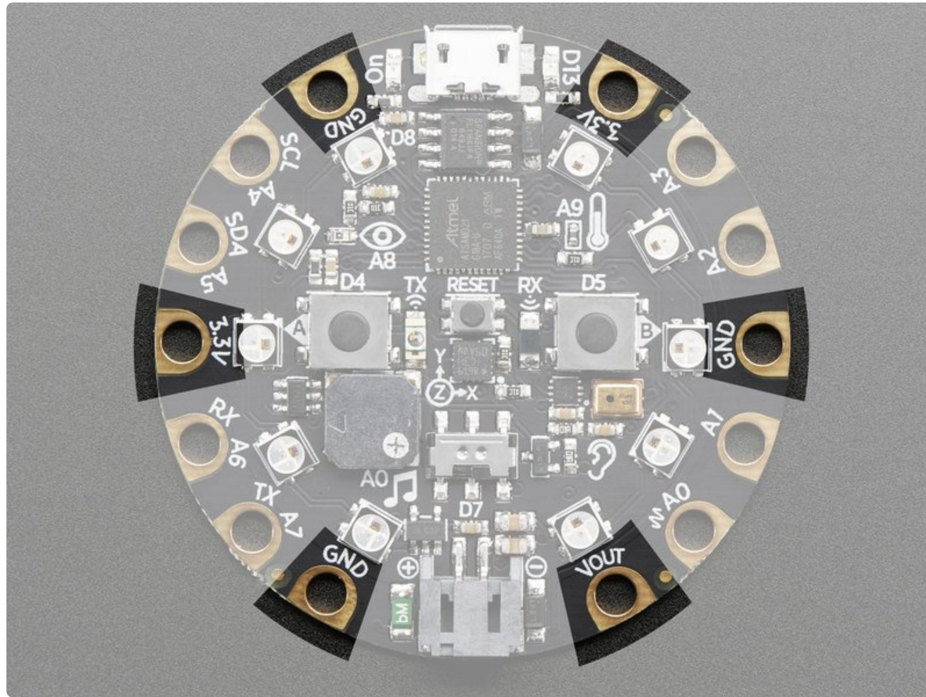


The diagram above is not definitive! For example there is a PWM ~ missing from the A1 line.

Despite having only 14 pads with 8 general purpose I/O pins available, there are a *lot* of possibilities with Circuit Playground Express. 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 A0, 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.

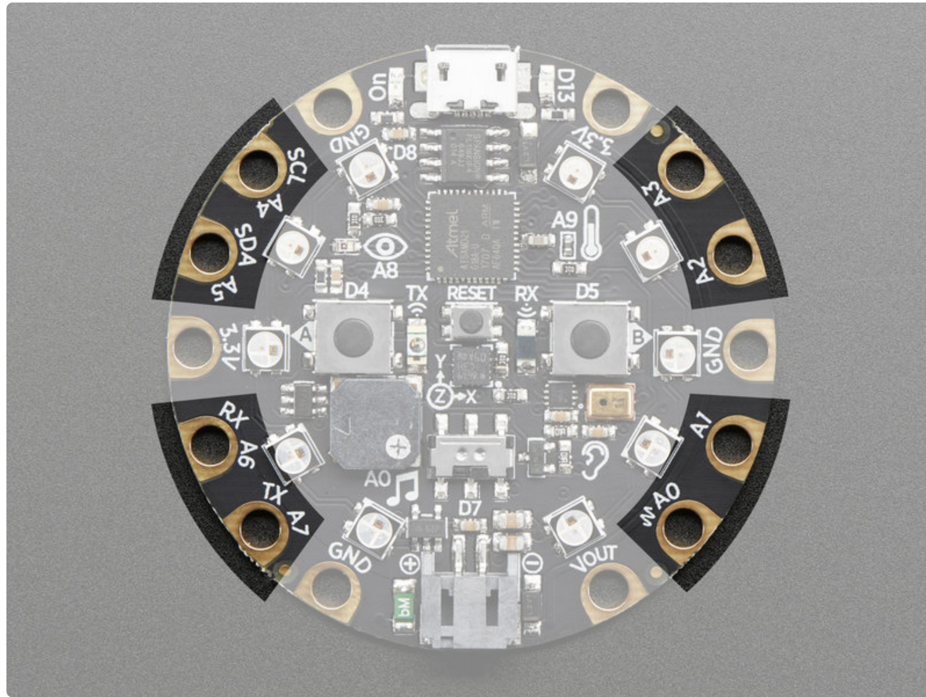
- **GND** - there are 3 x **Ground** pads. They are all connected together, and are all the signal/power ground connections
- **3.3V** - there are two **3.3 Volt output** pads. They are connected to the output of the onboard regulator. The regulator can provide about 500mA max, but that includes all 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 Ouput) pins! For reference you may want to also check out the datasheet-reference in the downloads section for the core ATSAMD21G18 pin. 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, all can be used as analog inputs (12-bit ADC). All but A0 can be used for hardware 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://adafru.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 A0, 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 **A0** which is in the bottom right corner, and work our way counter-clockwise

- **A0** (a.k.a **D12**) - This is a special pin that can do true analog output so it's great for playing audio clips. In can be digital I/O, or analog 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. This pin has PWM output and can be capacitive touch sensor

- **A2 / D9** - This pin can be digital I/O, or analog Input. This pin has PWM output and can be capacitive touch sensor
- **A3 / D10** - This pin can be digital I/O, or analog Input. This pin has PWM output and can be capacitive touch sensor
- **A4 / D3** - This pin can be digital I/O, or analog Input. This pin is also the **I2C SCL** pin, and can be capacitive touch sensor
- **A5 / D2** - This pin can be digital I/O, or analog Input. This pin is also the **I2C SDA** pin, and can be capacitive touch sensor
- **A6 / D0** - This pin can be digital I/O, or analog Input. This pin has PWM output, **Serial Receive,** and can be capacitive touch sensor
- **A7 / D1** - This pin can be digital I/O, or analog Input. 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!

- **D4** - Left Button A
- **D5** - Right Button B
- **D7** - Slide Switch
- **D8** - Built-in 10 NeoPixels
- **D13** - Red LED
- **D27** - Accelerometer interrupt
- **D25** - IR Transmitter
- **D26** - IR Receiver
- **A0** - Speaker analog output
- **A8** - Light Sensor
- **A9** - Temperature Sensor
- **A10** - IR Proximity Sensor

- **D28** - Internal I2C SDA (access with **Wire1**)
- **D29** - Internal I2C SCL (access with **Wire1**)
- **D30 (PIN_SPI_MISO)** - SPI FLASH MISO
- **D31 (PIN_SPI_SCK)** - SPI FLASH SCK
- **D32 (PIN_SPI_MOSI)** - SPI FLASH MOSI
- **D33** - SPI FLASH Chip Select

Update the Bootloader (Mac Users Especially!)

If you are using a Mac and are programming with MakeCode, you probably need to update your bootloader!

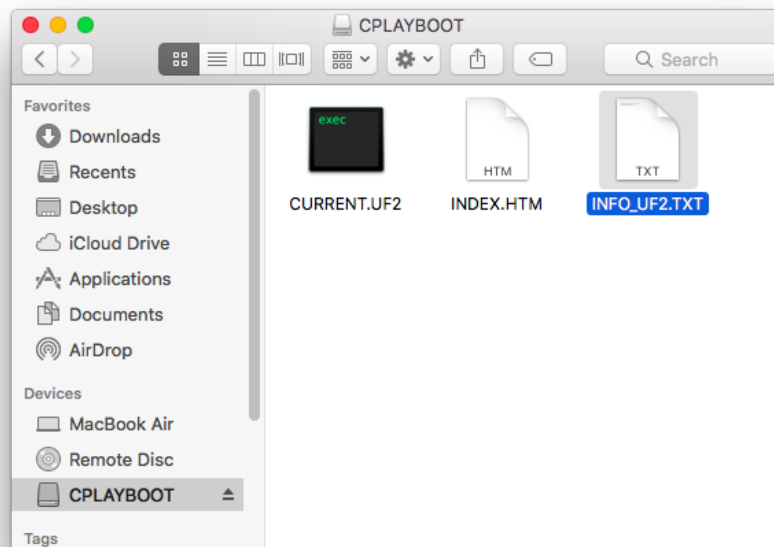
If you are using MakeCode (https://makecode.adafruit.com (https://adafru.it/wpC) or https://maker.makecode.com (https://adafru.it/EEB)) on a Mac, you need to update your board's bootloader to avoid a serious problem.

Starting with MacOS 10.14.4, Apple changed how USB devices are recognized on certain Macs. This caused a timing problem with boards that were loaded with a MakeCode program, preventing the `CPLAYBOOT` drive (or other `...BOOT` drive) from appearing.

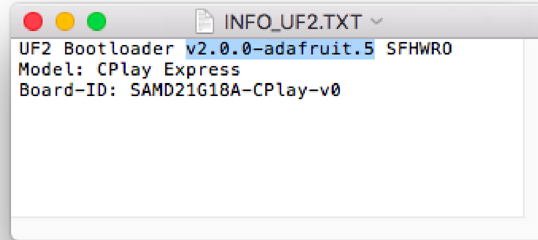
Updating Your Circuit Playground Express Bootloader (see below for other boards)

To see if you need to update your bootloader, get `CPLAYBOOT` to appear on your board. If you're running MakeCode, click the reset button once. If you're running CircuitPython or an Arduino program, double-click the reset button. All the NeoPixels should turn green.

Click the `CPLAYBOOT` drive in the Finder and then double-click the `INFO_UF2.TXT` file to see what's inside.



The bootloader version is listed in `INFO_UF2.TXT`. In this example, the version is " `V2.0.0-adafruit.5` ".



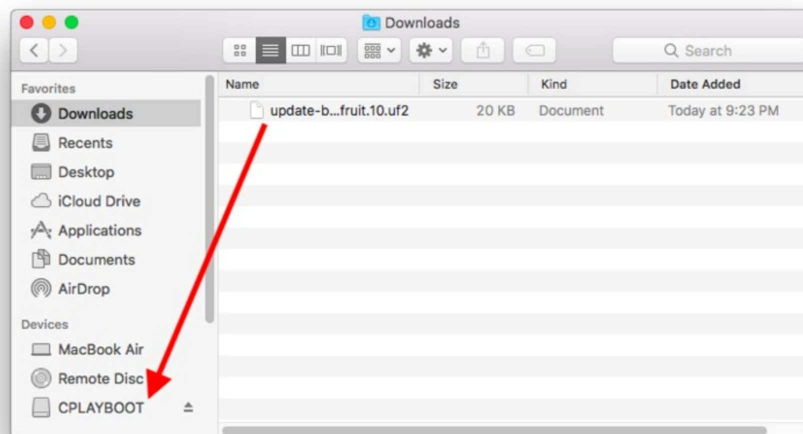
If the bootloader version you see is older than "**v3.3.0-adafruit.10**", you need to update. For instance, the bootloader above needs to be upgraded.

Download the latest version of the Circuit Playground Express bootloader updater here:

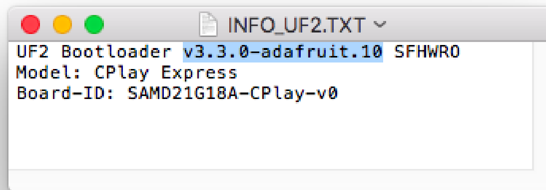
https://adafru.it/Bmg

https://adafru.it/Bmg

The bootloader updater will be named `update-bootloader-circuitplay_m0-v3.7.0.uf2` or some later version. Drag that file from your `Downloads` folder onto the `CPLAYBOOT` drive.



After you drag the updater onto `CPLAYBOOT`, the red LED on the board will flicker as the bootloader is updated. The NeoPixels will flash and turn green again. A few seconds later, `CPLAYBOOT` will reappear in the Finder. After that, you can click on `CPLAYBOOT` and double-click `INFO_UF2.TXT` again to confirm you've updated the bootloader.



Oh no, I updated MacOS already and I can't see CPLAYBOOT!

If your Mac has already been updated to MacOS 10.14.4 and now you can't see `CPLAYBOOT` in the Finder you need to find another computer that will work. Not all upgraded Macs will fail to show `CPLAYBOOT`: older ones can work. Or find a Mac that hasn't been upgraded yet. Any Windows 10 or Linux computer should work for upgrading your bootloader. Windows 7 computers will need drivers installed (https://adafru.it/Bf7), but then can work.

Upgrading Other Boards

If you don't have a Circuit Playground Express, but are using a different Adafruit board with a UF2 bootloader, the procedure is similar to what you see above. Single-click (if using MakeCode) or double-click the reset button. You'll see a drive whose name ends in `...BOOT`. Check the version as above. If you need to update, download a bootloader updater for your board. Links to the latest updaters are available here in the Updating the bootloader (https://adafru.it/D3D) section on the UF2 Bootloader Details (https://adafru.it/D3D) page. Then drag or copy the updater .uf2 file onto your `BOOT` drive.

Windows Driver Installation

Mac and Linux do not require drivers, only Windows folks need to do this step

Windows 10 users probably can skip this step, because Windows 10 already has many drivers built in. Try skipping the installation first to see if it's unnecessary.

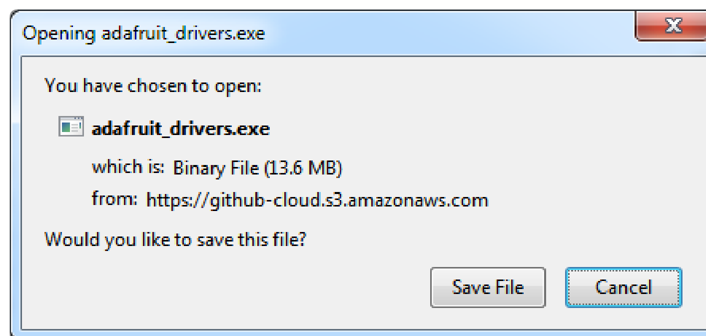
Before you plug in your board, you'll need to possibly install a driver!

Click below to download our Driver Installer.

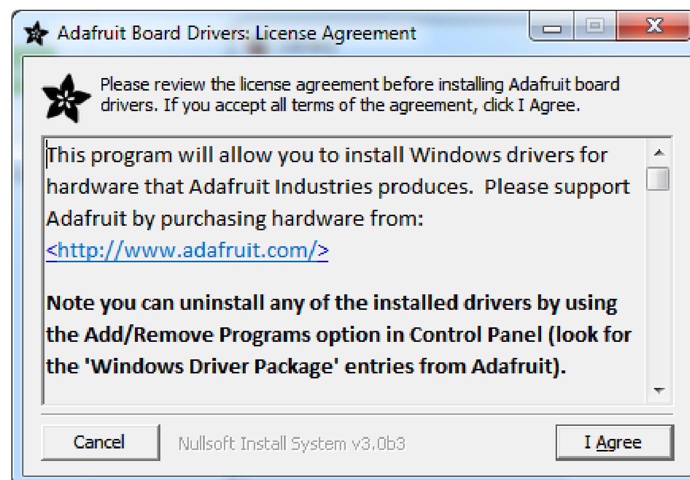
https://adafru.it/AB0

https://adafru.it/AB0

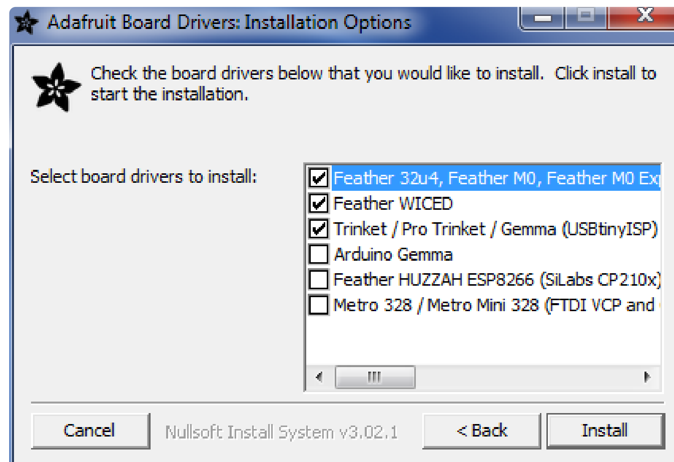
Download and run the installer.



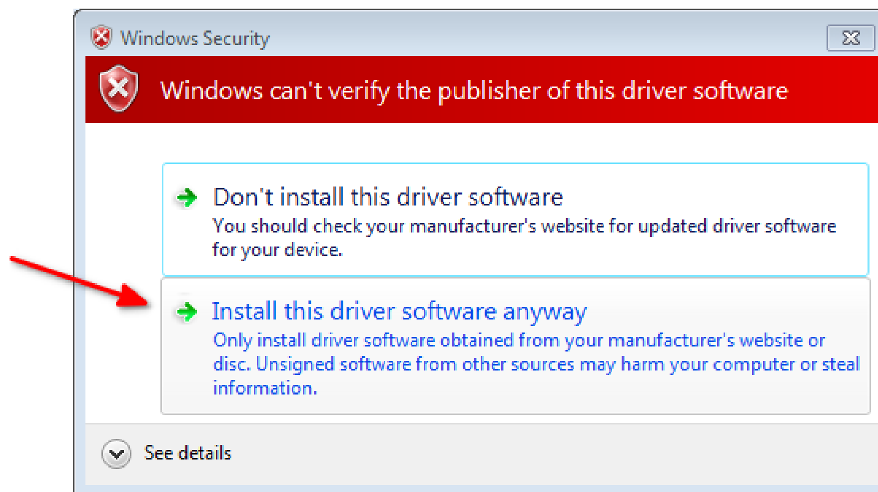
Run the installer! Since we bundle the SiLabs and FTDI drivers as well, you'll need to click through the license



Select which drivers you want to install, we suggest selecting all of them so you don't have to do this again!



As of version 2.5.0.0, the Adafruit drivers package is no longer signed, and some of the drivers it contains are also no longer signed. You'll need to click the second item in this dialog box when it appears:



On Windows 7, by default, we install a single driver for most of Adafruit's boards, including the **Feather 32u4, the Feather M0, Feather M0, Express, Circuit Playground, Circuit Playground Express, Gemma M0, Trinket M0, Metro M0 Express**. On Windows 10 that driver is not necessary (it's built in to Windows) and it will not be listed.

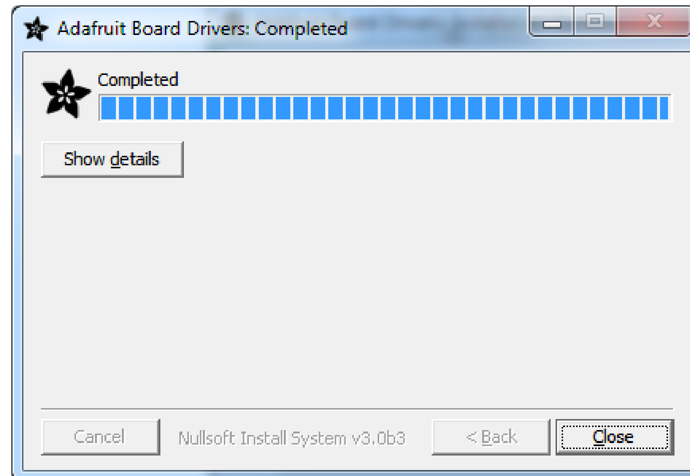
The **Trinket / Pro Trinket / Gemma / USBtinyISP** drivers are also installed by default.

You can also, optionally, install the **Arduino Gemma** (different than the Adafruit Gemma!), **Huzzah and Metro 328** drivers.

Click **Install** to do the installin'.

Note that on Windows 10, support for many boards is built in. If you end up not checking any

boxes, you don't need to run the installer at all!



Manual Driver Installation

If windows needs the driver files (inf/cat) for some reason you can get all the drivers by downloading the source code zip file from this link:

https://adafru.it/AB0

https://adafru.it/AB0

And point windows to the **Drivers** folder when it asks for the driver location

Code.org CSD

Did you know you can use the Circuit Playground Express (CPX) with Code.org CS Discoveries course? Well you can now!

All you have to do is load up the "Firmata" firmware onto your CPX. Luckily, its very easy and **you only have to do this once per device**

Step 1. Connect to USB

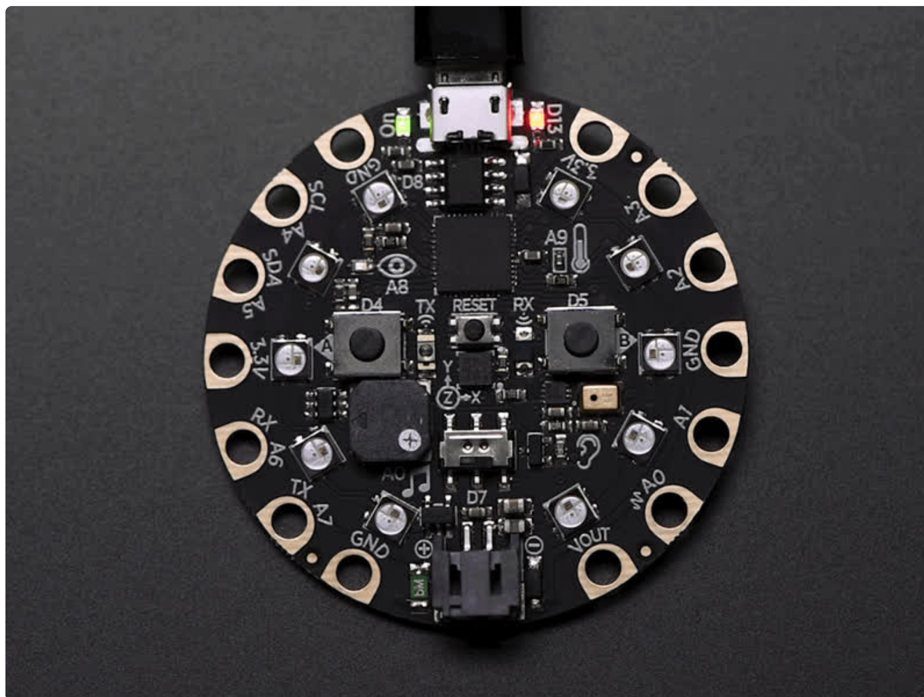
Plug your Circuit Playground Express into your computer using a known-good USB cable

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

Step 2. Press RESET to get into bootloader mode

There's a small button in the middle of the board. Try clicking it once, to put it into bootloader mode.

You'll know you are in bootloader mode successfull when all the LEDs turn a green color



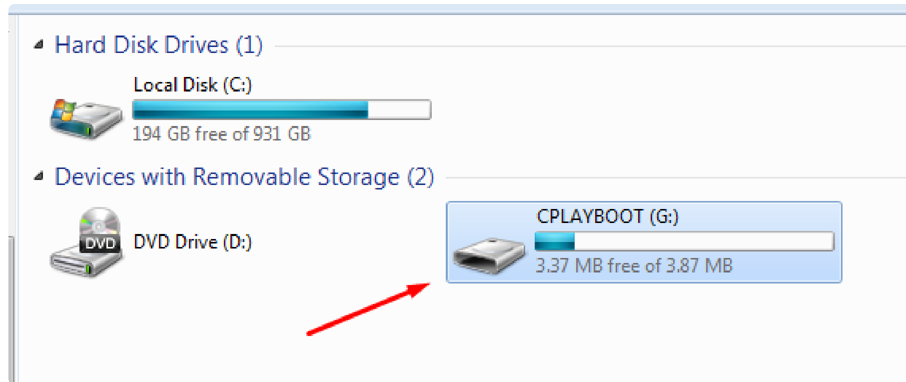
If clicking RESET once doesn't do it, try double-clicking (MakeCode requires single-click, Arduino requires

double-click)

If the color LEDs turn all red, check your USB cable, try another cable or another USB port

Step 3. Copy over Firmata firmware

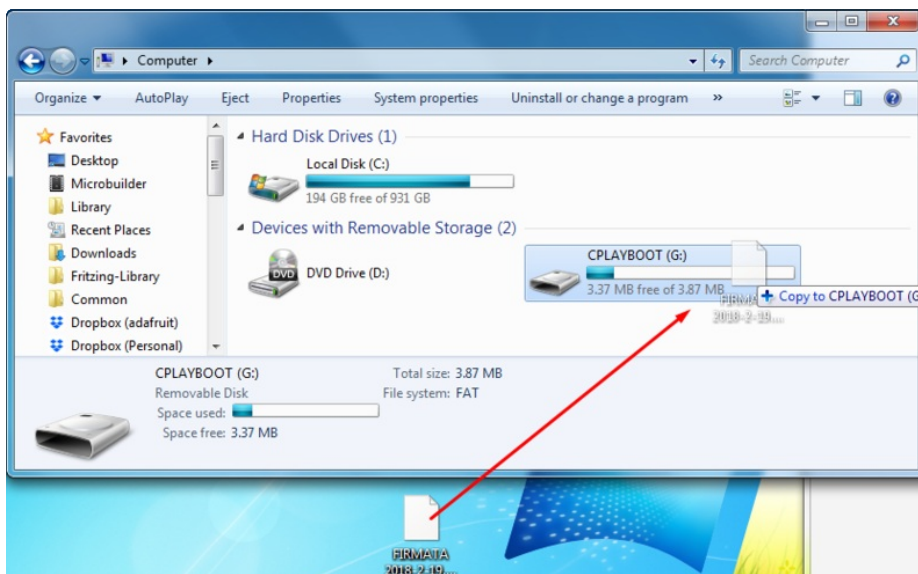
Your computer will now have a **CPLAYBOOT** disk drive appear.



Click this link to download **FIRMATA yyyy-mm-dd.UF2** (the year, month, date may vary) and drag that onto the drive

https://adafru.it/AGk

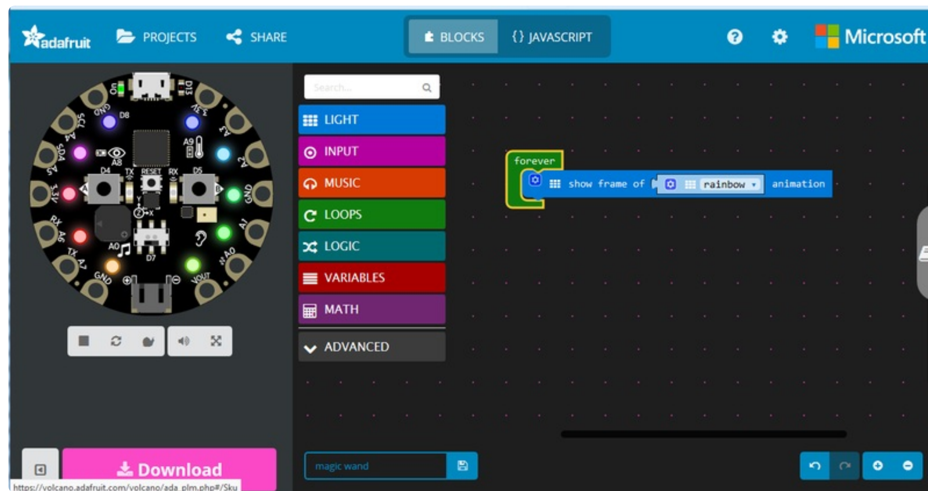
https://adafru.it/AGk



The CPX will reboot automatically after a few seconds. The CPLAYBOOT disk drive will go away (you may get a warning from the computer that it didn't expect the ejection, ignore it!)

Your CPX is now ready to run **Code.org CS Discoveries!**

MakeCode



One of the beautiful things about Circuit Playground Express is that you can make **three ways**:

1. MakeCode (https://adafru.it/wmd)
2. CircuitPython (https://adafru.it/BeW)
3. Arduino IDE (or direct ARM GCC programming)

If this is your very first time programming or coding, we suggest starting with MakeCode (https://adafru.it/wmd) - it is the super fast to get started not just with Circuit Playground Express, but coding in general! (Once you've got the hang of MakeCode (https://adafru.it/wmd) you can then take a look at CircuitPython or Arduino.)

MakeCode does not require any software installation, it works on any computer with a web-browser. You can use it with Mac, Windows, Linux, Chromebooks.

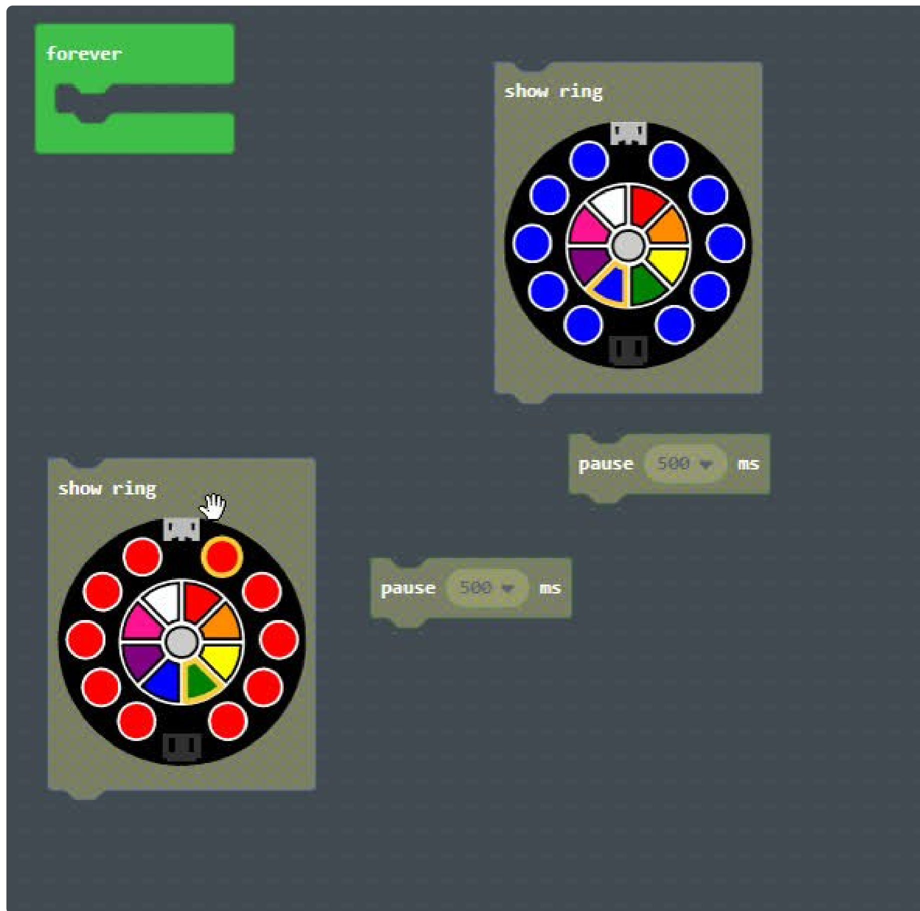
MakeCode uses drag-and-drop blocks just like Scratch, there is no syntax or semicolons. The blocks snap together to create larger and more complex projects

MakeCode lets you get started in 5 minutes or less, there are built in guides and projects, but its also just fun to play around.

MakeCode has surprisingly advanced runtime, despite looking very simple it can do very complex tasks that normally would require advanced programming. You can get creative very very quickly.

Read on to learn how to use MakeCode (https://adafru.it/wmd) and build your first project!

What is MakeCode?



Microsoft MakeCode for Adafruit is a web-based code editor for physical computing. It provides a block editor, similar to Scratch or Code.org, and also a JavaScript editor for more advanced users. **Try it now at https://makecode.adafruit.com/ (https://adafru.it/wmd) !**

Some of the key features of MakeCode are:

- **web based editor:** nothing to install
- **cross platform:** works in most modern browsers (https://adafru.it/wqe) from tiny phone to giant touch screens
- **compilation in the browser:** the compiler runs in your browser, it's fast and works offline
- **blocks + JavaScript:** drag and drop blocks or type JavaScript, MakeCode let's you go back and forth between the two.
- **works offline:** once you've loaded the editor, it stays cached in your browser.
- **event based runtime:** easily respond to button clicks, shake gestures and more

MakeCode currently supports the **Adafruit Circuit Playground Express.** For other boards, like the Adafruit Metro M0, try https://maker.makecode.com (https://adafru.it/BeY) .

Circuit Playground Express

Circuit Playground Express is the next step towards a perfect introduction to electronics and programming. We've taken the original Circuit Playground Classic and...

$24.95

In Stock

Add to Cart

Circuit Playground Express - Base Kit

It's the Circuit Playground Express Base Kit! It provides the few things you'll need to get started with the new

Out of Stock

Out of Stock

MakeCode works for the Express edition of the Circuit Playground, not the Classic.

Editing Blocks

The block editor is the easiest way to get started with MakeCode. You can drag and drop blocks from the category list. Each time you make a change to the blocks, the simulator will automatically restart and run the code. You can test your program in the browser!

- try our getting started tutorial (https://adafru.it/wmd) that helps you build a siren program
- try one of our projects (https://adafru.it/wpD) to learn more about the features of the Circuit Playground
- our favorite way to learn is by watching MakeCode Live with John Park, either the playlist of recorded videos (https://adafru.it/McY), or by watching it live on Tuesdays at 3pm ET on the Adafruit Youtube Channel (https://adafru.it/McZ)
- take a deep dive in the documentation (https://adafru.it/wpE)

Blinky!

Let's show how MakeCode works by building a simple program that blinks the 10 awesome NeoPixels.

Creating a blink effect is done by setting all the ring LEDs to red, **pause** for a little, then turn them off, pause for a little, then repeat **forever**.

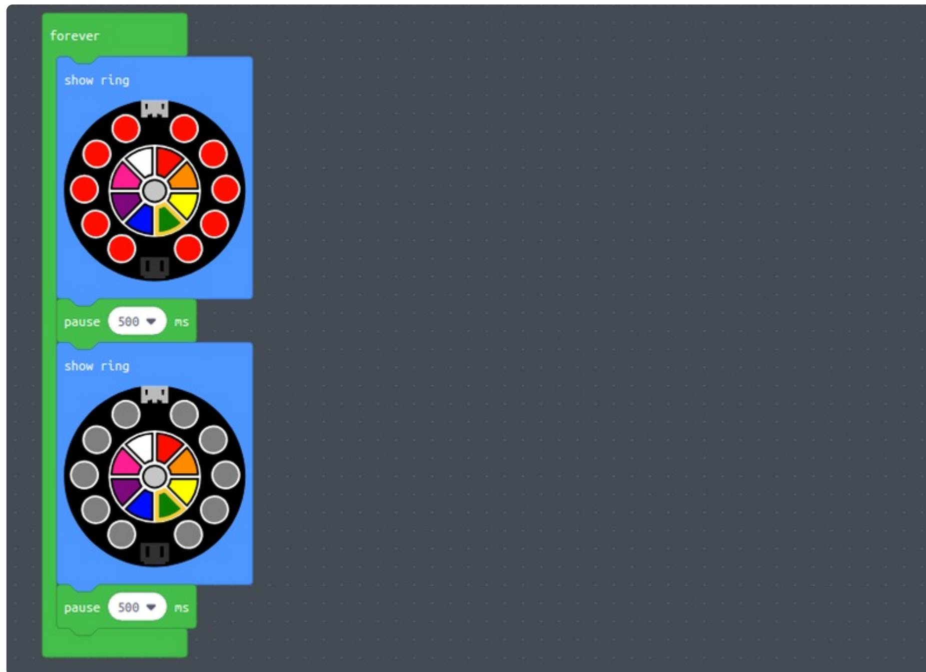
Let's gather the blocks we need to convert the description above into Blocks that the Circuit Playground can understand and run:

- **forever** runs blocks in a loop with a 20ms pause in between (it is similar to Arduino **loop**).
- **show ring** sets the color on the 10 neopixels at once
- **pause** blocks the current thread for 100ms. If other events or forever loops are running, they have the opporunity to run at this time.

Do you want to select or change colors? The **show ring** block has a built-in color picker. Select the color from the color wheel to select a color, then click one of the ten Neopixel rings to modify its color.

How do I disable a Neopixel? The grey dot in the middle of the color wheel indicates that the pixel is off. Select the grey from the color wheel and then click the Neopixel ring.

You can see the Blinky block program in action in the MakeCode editor below. You can also see how the blocks are "slotted together". Clicking the question marks on the box will pop-up a comment to explain what the block does.



https://adafru.it/BAI

https://adafru.it/BAI

In the next section, we'll load the Blinky code onto the Circuit Playground Express!

Downloading and Flashing

Getting your code into your device is very easy with MakeCode. You do not need to install any software on your machine and the process takes two steps:

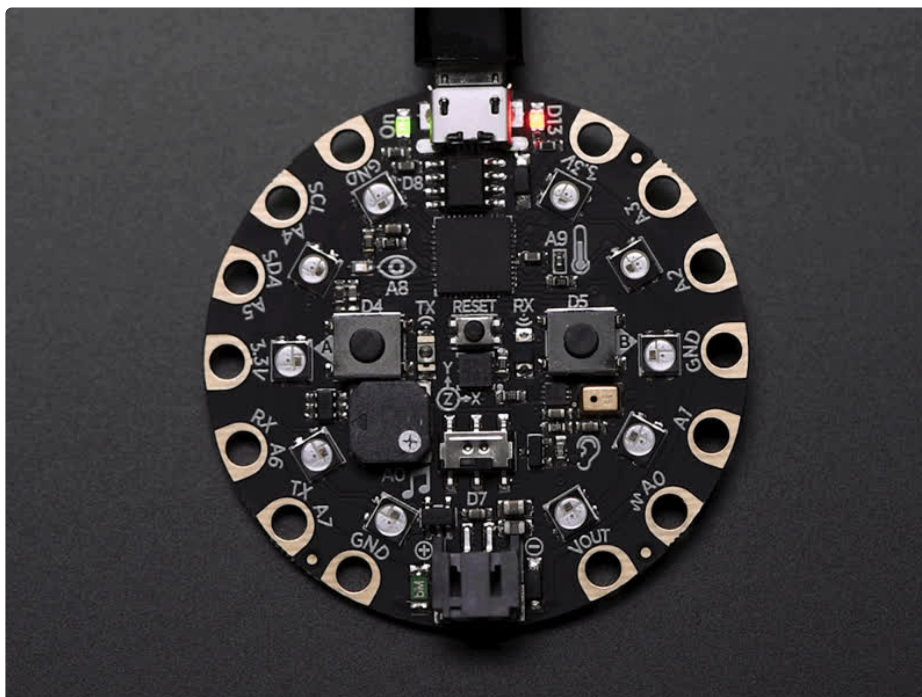
- Step 1: Connect your board via USB and enter **bootloader mode**
- Step 2: **Compile and Download** the .uf2 file into your board drive

We are going to go through these two steps in detail.

Step 1: Bootloader mode

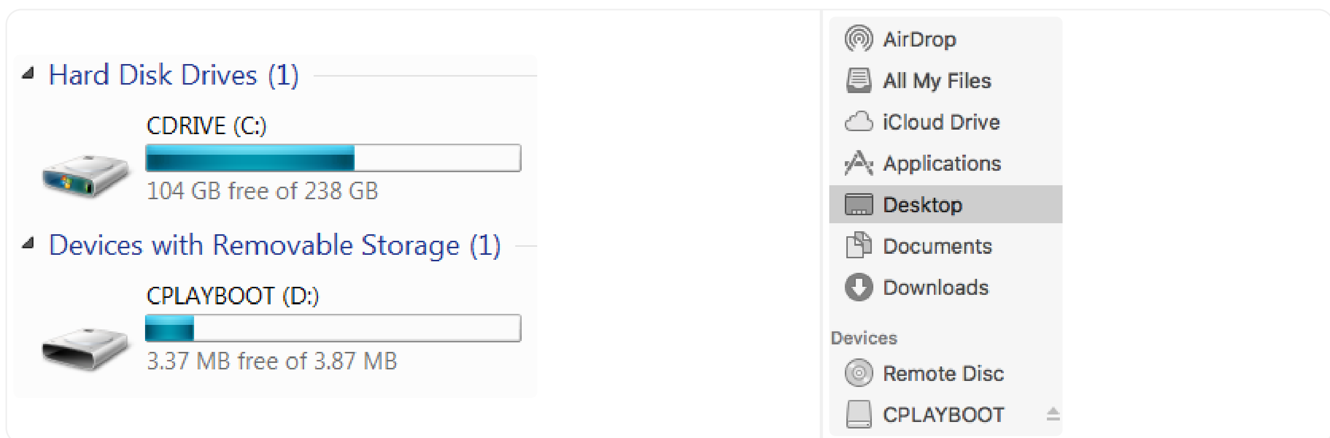
Connect your board to your computer via a USB cable. Press the reset button once to put the board in bootloader mode.

> If it is your first time running MakeCode or if you have previously installed Arduino or CircuitPython, you may need to double press the reset button to get your board into bootloader mode.



When the Circuit Playground Express is in Bootloader mode, all the LEDs will turn **red briefly, then green. Verify your status LED is also pulsing red.** Your computer should show a new removable drive called "CPLAYBOOT"

If the LEDs are all red: Either the computer is still installing drivers (Please wait a minute, Windows takes some time to install updates.) or you have a bad USB connection. If you keep getting red LEDs - try a new USB cable (you may want to ensure your USB cable is not charge only, it needs to transfer data) or a different USB port.



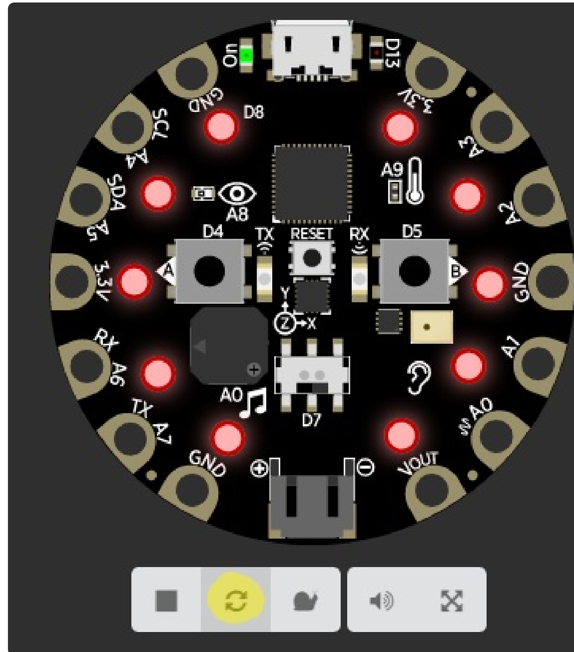
Once your LEDs are all green, you should see a CPLAYBOOT drive appear in your drive list in your file explorer.

We are now ready to compile our blinky code and download it to our board!

Step 2: Compile and Download

Let's first verify that our code compiles properly in MakeCode .

MakeCode has a built-in simulator that re-loads and re-runs code when restarted. This is an easy way to both ensure that our code compiles and simulate it before moving it onto the board. The refresh button re-loads the simulator with your latest version of block code.



If you receive a "we could not run this project" error, please check over your code for errors.

If your board is working in the simulator, it's time to download it to your actual board! Click the "Download" button. It will generate a .uf2 file and download it to your computer (UF2 (https://adafru.it/vPE) is a file format designed to flash microcontrollers over USB).

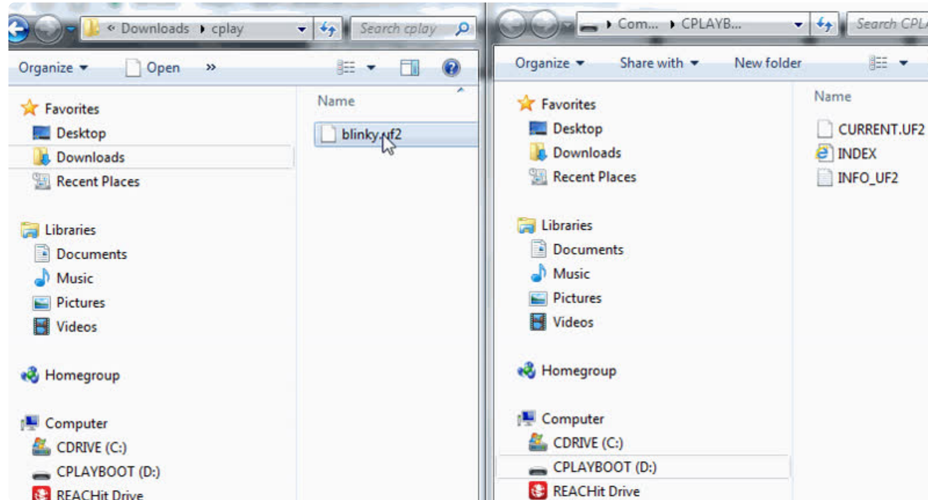
General Steps to copy over your program (not specific to any Operating system)

1. Ensure your board is in bootloader mode.
2. Find the .uf2 file generated by MakeCode in your file explorer. Copy it to the "CPLAYBOOT" volume.
3. The status LED on the board will blink while the file is transferring. Once it's done transferring your file, the board will automatically reset and start running your code (just like in the simulator!)

On a Mac, you can safely ignore the "Disk Not Ejected Properly" notification that may appear after copying your .uf2 file.

Windows: Open Windows Explorer (Windows key + E key) and locate the "blinky.uf2" file you generated. It's probably in your **Downloads** folder!

You can copy/paste the file to your **CPLAYBOOT** volume or you can drag/drop it like in the GIF below.



macOS: Open Finder and locate the "blinky.uf2" file. You can copy/paste this file to the "cplayboot" volume or drag/drop it from the same finder window.

If you want to avoid the copying process : You can download your programs directly to the board. To do this: change the download location in Chrome (https://adafru.it/wHA), Firefox (https://adafru.it/wHB), Safari (https://adafru.it/wHC), or Opera (https://adafru.it/wHD)to the main directory of your "CPLAYBOOT" volume.

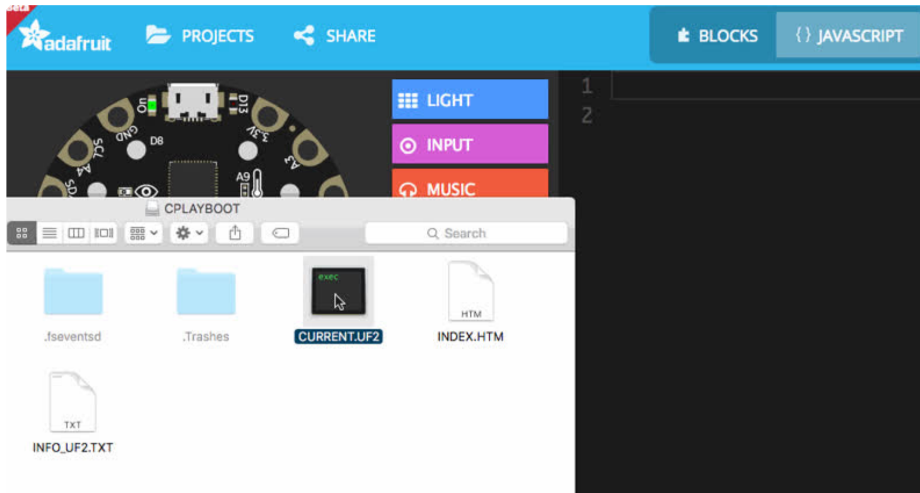
Running MakeCode that's Already Loaded

If you unplug your board and then plug it back in again to your computer, it may not run the program again automatically. If it instead shows all green NeoPixels, just press the reset button and your program will start.

Saving and Sharing

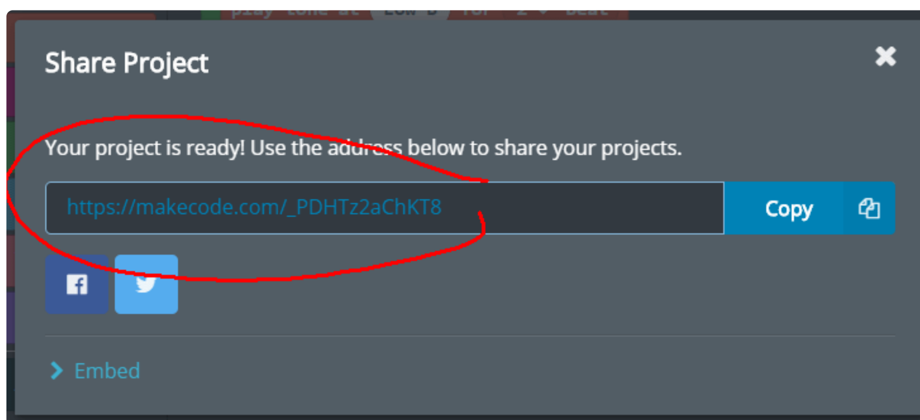
Want to change the Blinky program to display different colors, or make it play a sound? Editing the program on your Circuit Playground is super easy!

Extracting your code from the Circuit Playground



The .uf2 file **(CURRENT.UF2)** you created by clicking on the Compile button in MakeCode also contains the source code of your program!

You can open this file in MakeCode by dragging and dropping it into the browser to edit it.



Sharing

You can share your code by clicking on the **share** button. After confirmation, MakeCode will create a short unique URL for your code. Anyone with that URL will be able to reload the code.

These URLs can also be used to embed the editor your blog or web pages! Just copy paste the URL in

your text editor and (if it supports oEmbed) it will automatically load it in your page.

Editing JavaScript

If you already have some experience coding or you feel ready to take the next step, MakeCode features a fully-feature JavaScript editor in the browser!

- read the JavaScript docs

Give it a try!

You can also switch between blocks and JavaScript by clicking the button on the top.

https://adafru.it/BAn

https://adafru.it/BAn

Apps

MakeCode also provides various apps to provide additional features not available in browsers.

Windows Store

The MakeCode for Adafruit Windows Store app.

- **Super fast HID-based flashing**: no more drag and drop.
- Reading debug message and surfacing them in the editor (debug messages are sent over HID not CDC Uart/Serial)

Node.JS

The (open source) GitHub repo contains instructions to run a local node.js web server with HID flashing and serial monitoring.

Other Good Stuff

This guide is meant as a starter block but it's worth mentioning briefly about other feature of MakeCode...

GitHub packages

Additional blocks or drivers can be packaged in github repo and loaded in the editor via the **Add Package** dialog. Packages can contain JavaScript, C++ and yes! ASM! https://makecode.adafruit.com/packages

We are Open Source on GitHub

Checkout https://makecode.com/about for more info about the various repos.

We have crowd-sourced translations

Whether you want to code in Klingon or your native non-English language, MakeCode supports crowd sourced translations at https://makecode.adafruit.com/translate.

MakeCode and Windows 10

Nearly all tutorials on the Adafruit Learning System are written such that the user can use the web version of MakeCode. The web version runs on multiple computer types providing broad compatibility.

There is another version of MakeCode available from Microsoft in the form of an app that runs on Windows 10 machines.

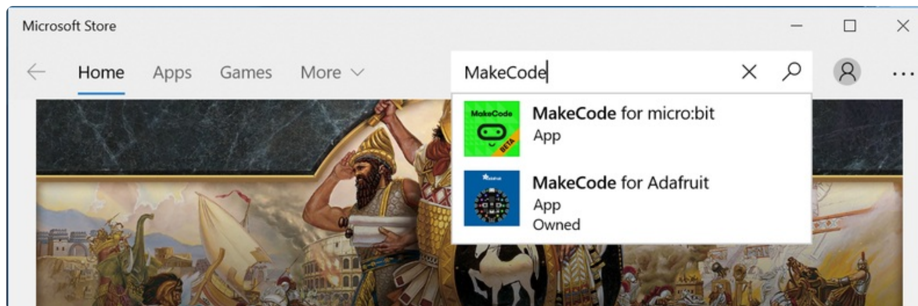
The big advantage the Windows 10 Microsoft Store version of MakeCode is the ability to read data back and easily plot it, something the web version lacks at present.

The MakeCode App for Windows 10

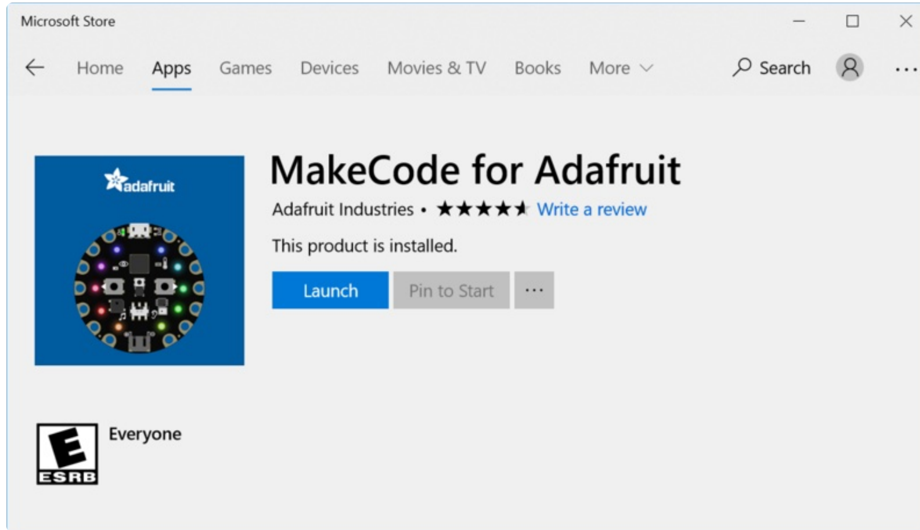
The Microsoft MakeCode App is located in the Microsoft App Store.

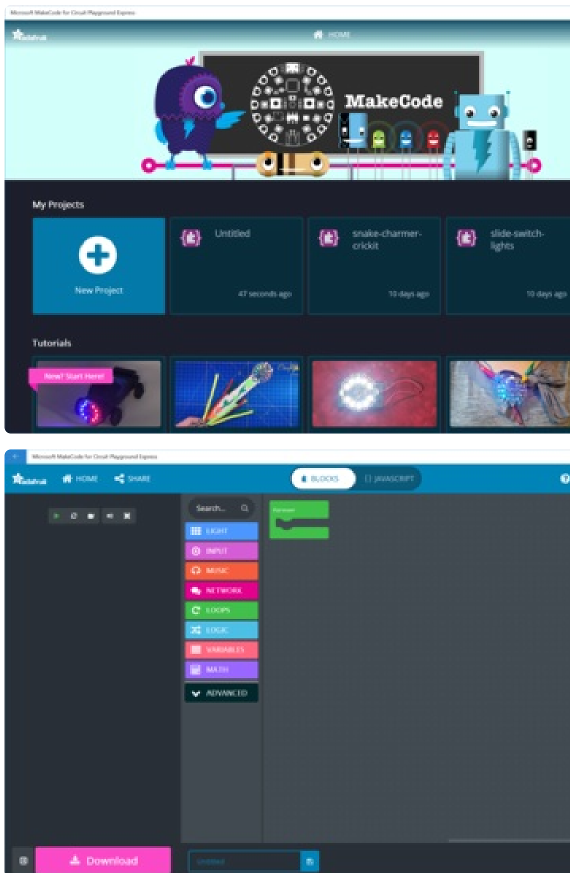
Installation

In the Windows 10 Search bar (usually next to your Start Button), type "Microsoft Store". In the search bar for the store app, type MakeCode. Select MakeCode for Adafruit.



Click the Install button for MakeCode for Adafruit. The app will install and you can then press the Launch button.





The introductory screens are identical to the web application.

The interface looks nearly identical to the web version but has a couple of differences noted on the next page.

See More MakeCode Projects

See More MakeCode Projects (https://adafru.it/Bwv)

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!

CircuitPython

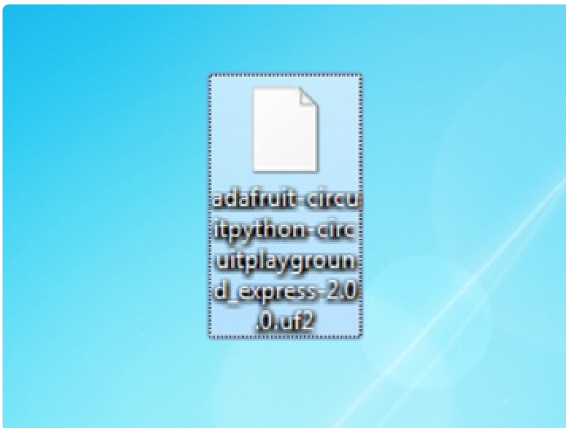
As we continue to develop CircuitPython and create new releases, we will stop supporting older releases. If you are running an older version of CircuitPython, you need to update. Click the button below to download the latest!

Install or update CircuitPython!

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

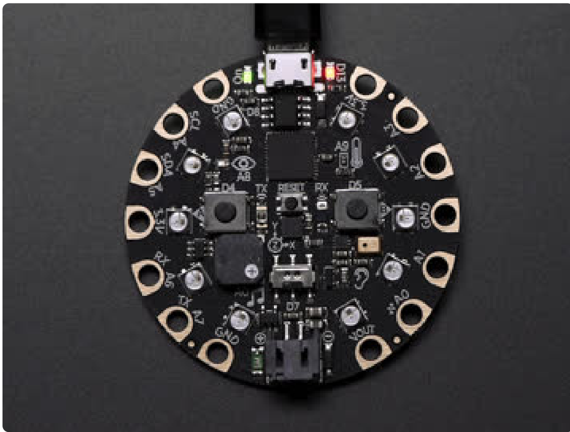
https://adafru.it/cp-cpx

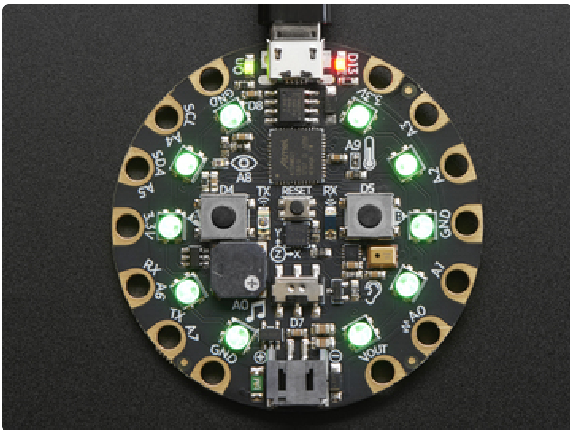
https://adafru.it/cp-cpx



Click the link above and download the latest UF2 file

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



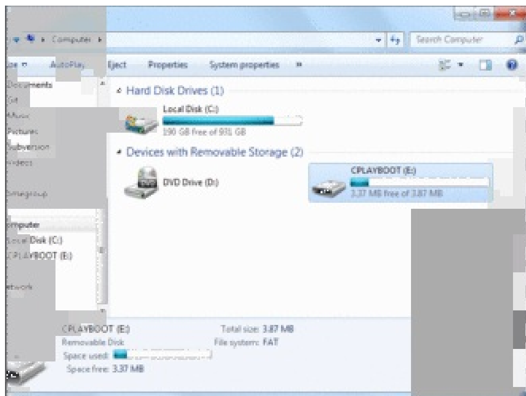


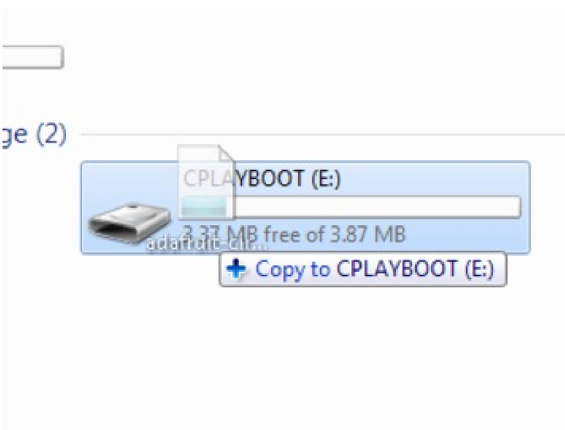
Plug your Circuit Playground Express into your computer using a known-good USB cable

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

Double-click the small **Reset** button in the middle of the CPX, you will see all of the LEDs turn green. If they turn all red, check the USB cable, try another USB port, etc.

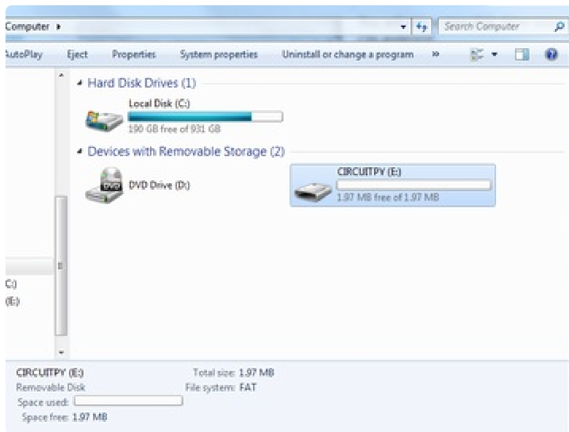
(If double-clicking doesn't do it, try a single-click!)





You will see a new disk drive appear called **CPLAYBOOT**

Drag the **adafruit-circuitpython-etc...uf2** file onto it



The **CPLAYBOOT** drive will disappear and a new disk drive will appear called **CIRCUITPY**

That's it! You're done :)

Further Information

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

Installing Mu Editor

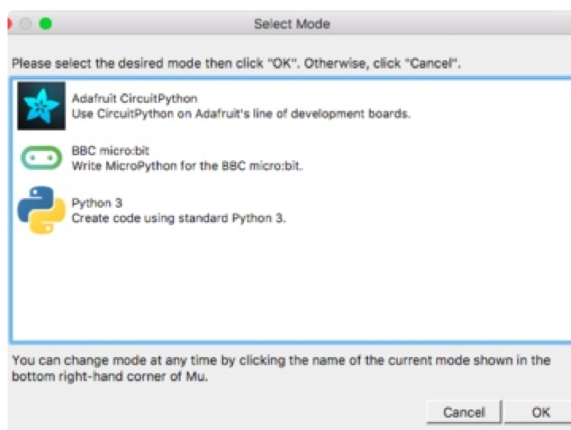
Mu is a simple code editor that works with the Adafruit CircuitPython boards. It's written in Python and works on Windows, MacOS, Linux and Raspberry Pi. The serial console is built right in so you get immediate feedback from your board's serial output!

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

Download and Install Mu

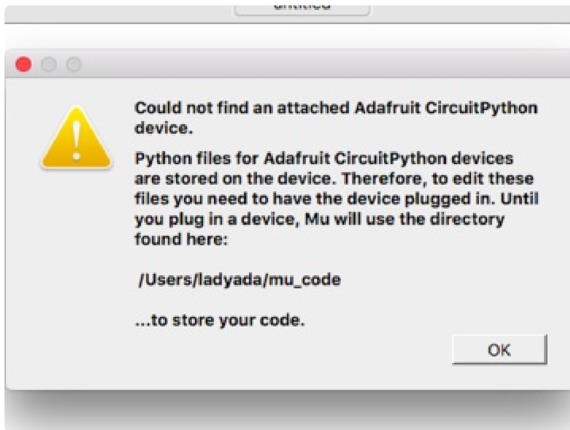
Download Mu from https://codewith.mu (https://adafru.it/Be6). Click the **Download** or **Start Here** links there for downloads and installation instructions. The website has a wealth of other information, including extensive tutorials and and how-to's.

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 **CircuitPython**!

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



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

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

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

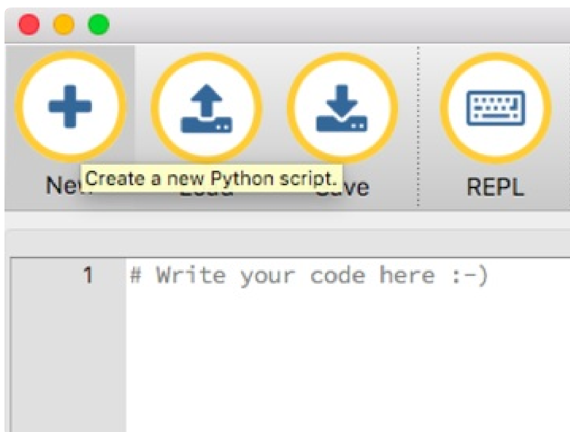
Creating and Editing Code

One of the best things about CircuitPython is how simple it is to get code up and running. 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) 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.LED)
led.direction = digitalio.Direction.OUTPUT

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

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

If you're using QT Py or a Trinkey, please download the NeoPixel blink example (https://adafru.it/SB2).

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

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

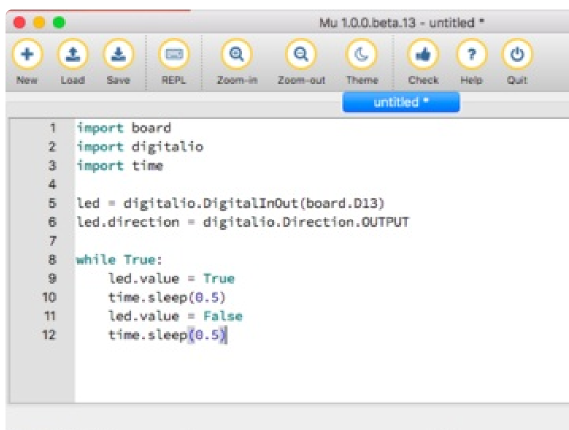
For Adafruit CLUE, you'll need to use `board.D17` instead of `board.LED`. The rest of the code remains the same. Make the following change to the `led =` line:

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

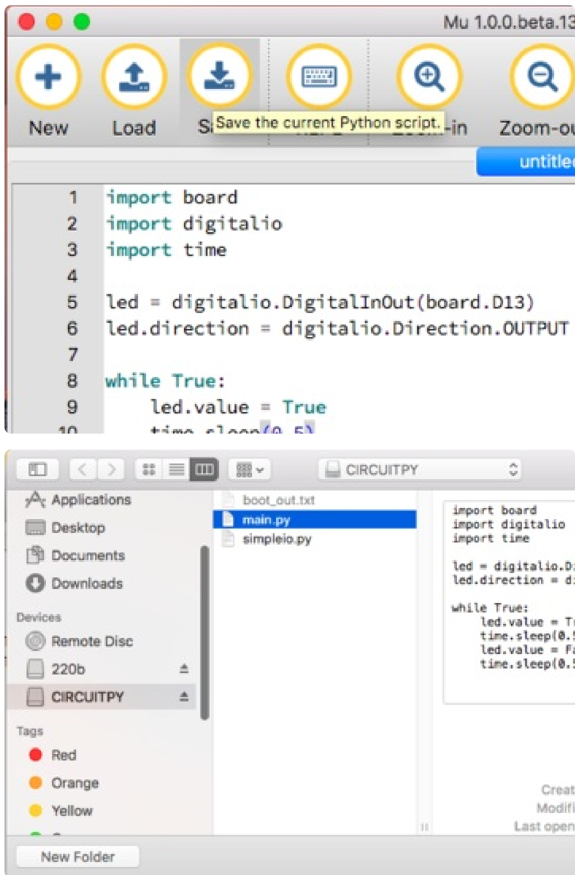
If you are using Adafruit ItsyBitsy nRF52840, you will need to edit the code to use board.BLUE_LED as shown below!

For Adafruit ItsyBitsy nRF52840, you'll need to use `board.BLUE_LED` instead of `board.LED`. The rest of the code remains the same. Make the following change to the `led =` line:

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



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.



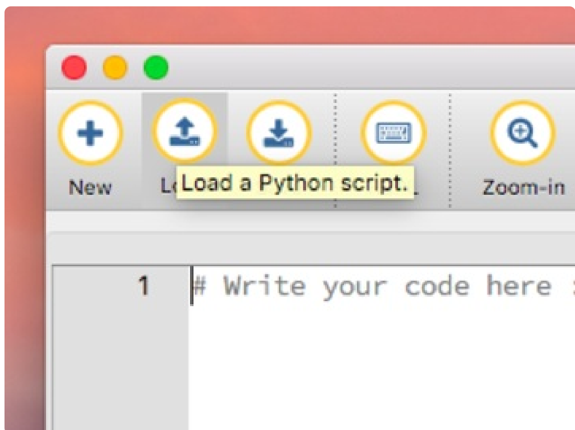
Save this file as **code.py** on your CIRCUITPY drive.

On each board (except the ItsyBitsy nRF52840) you'll find a tiny red LED. On the ItsyBitsy nRF52840, you'll find a tiny blue LED.

The little LED 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
- thonny (https://adafru.it/Qb6) fully writes files on save

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!

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

2. Eject or Sync the Drive After Writing

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

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

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

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

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

Back to Editing Code...

Now! Let's try editing the program you added to your board. Open your **code.py** file into your editor. 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.LED)
led.direction = digitalio.Direction.OUTPUT

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

Leave the rest of the code as-is. Save your file. See what happens to the LED on your board? Something changed! Do you know why? 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.LED)
led.direction = digitalio.Direction.OUTPUT

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

Imports & Libraries

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

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

The `import` statements tells the board that you're going to use a particular library in your code. In this example, we imported three modules: `board`, `digitalio`, and `time`. All three of these modules are built into CircuitPython, so no separate library files are needed. That's one of the things that makes this an excellent first example. You don't need any thing 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` let's you pass time by 'sleeping'

Setting Up The LED

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

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

Your board knows the red LED as `LED`. So, 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 Happens When My Code Finishes Running?

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

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

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

What if I don't have the loop?

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

The simplest loop would be:

```
while True:
    pass
```

And remember - you can press to exit the loop.

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

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
```

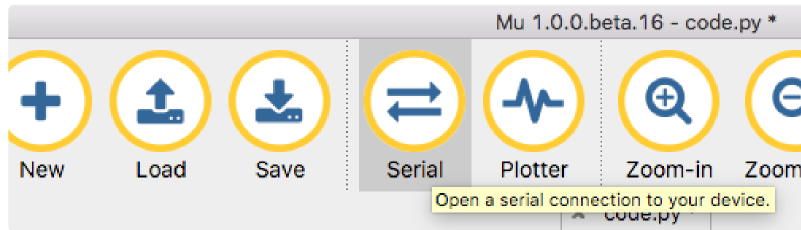
Are you using Mu?

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://adafru.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://adafru.it/AAH)

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

Interacting with the Serial Console

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

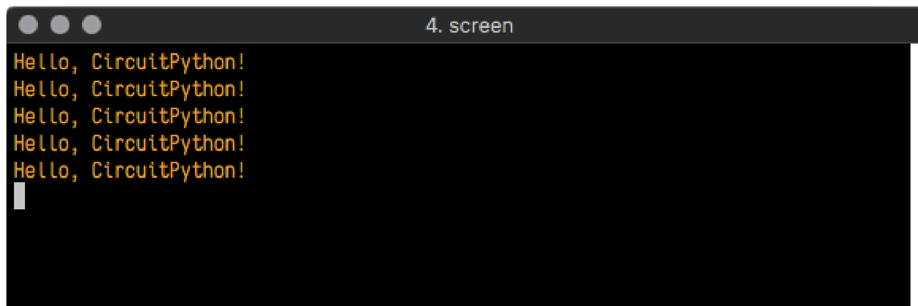
The code you wrote earlier has no output to the serial console. So, we're going to edit it to create some output.

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.LED)
led.direction = digitalio.Direction.OUTPUT

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

Save your file.

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



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

```
import board
import digitalio
import time

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

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

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

```
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 disab
le.
code.py output:
Hello back to you!
Hello back to you!
```

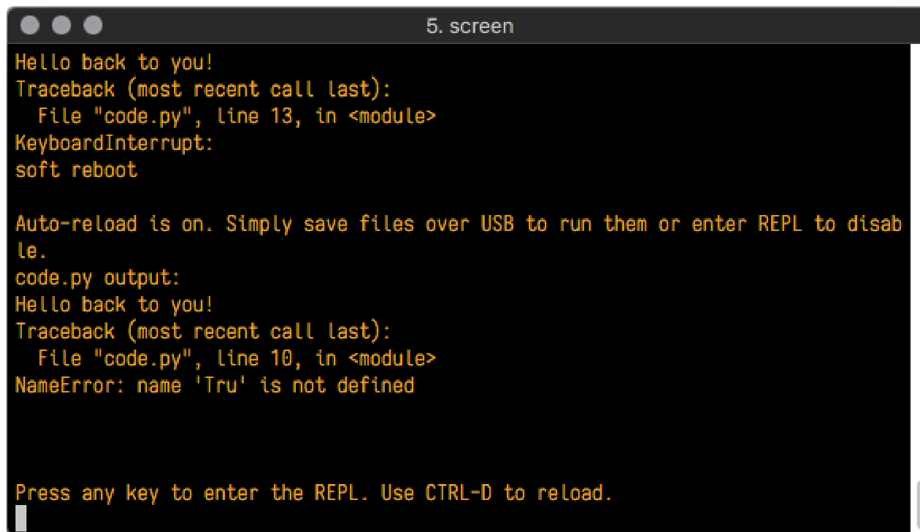
The `Traceback (most recent call last):` is telling you the last thing your board was doing before you saved your file. This is normal behavior and will happen every time the board resets. This is really handy for troubleshooting. Let's introduce an error so we can see how it is used.

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

```
import board
import digitalio
import time

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

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

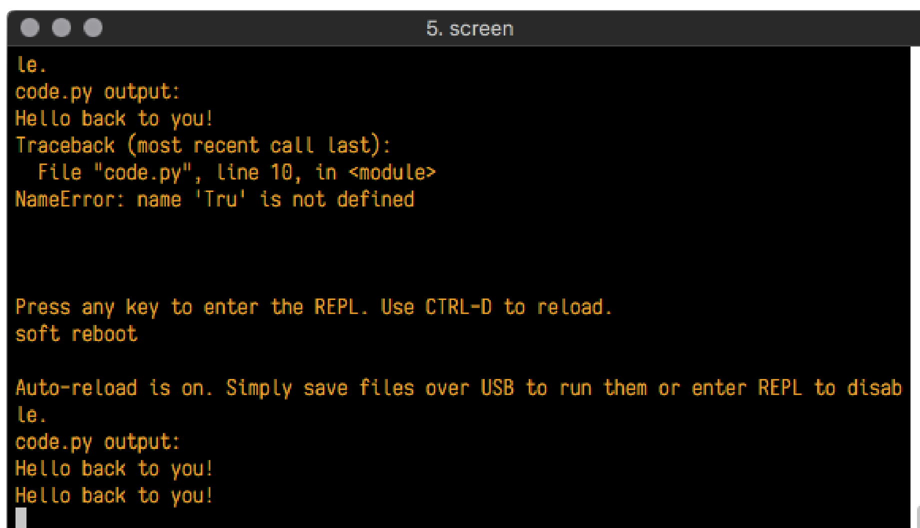
Save your file. You will notice that your red LED will stop blinking, and you may have a colored status LED blinking at you. This is because the code is no longer correct and can no longer run properly. We need to fix it!

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!



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

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



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

The serial console will display any output generated by your code. Some sensors, such as a humidity

sensor or a thermistor, receive data and you can use print statements to display that information. You can also use print statements for troubleshooting. If your code isn't working, and you want to know where it's failing, you can put print statements in various places to see where it stops printing.

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

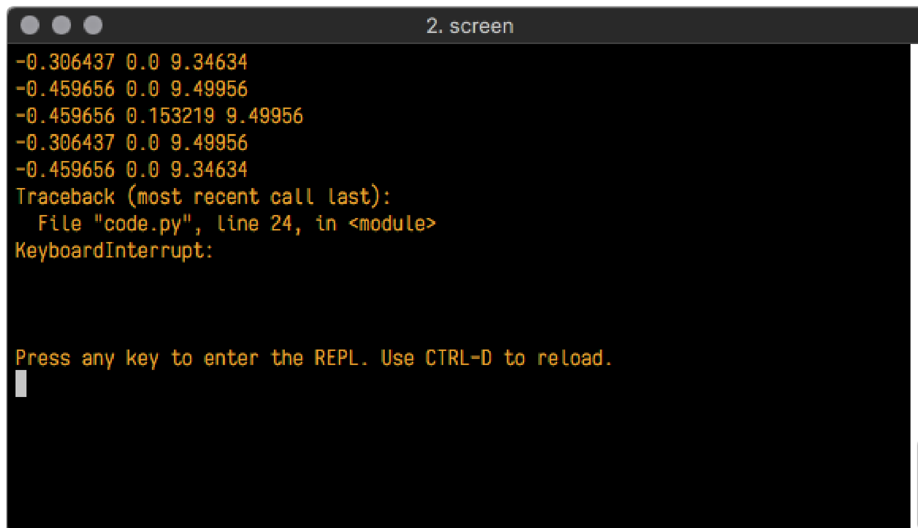
The REPL

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

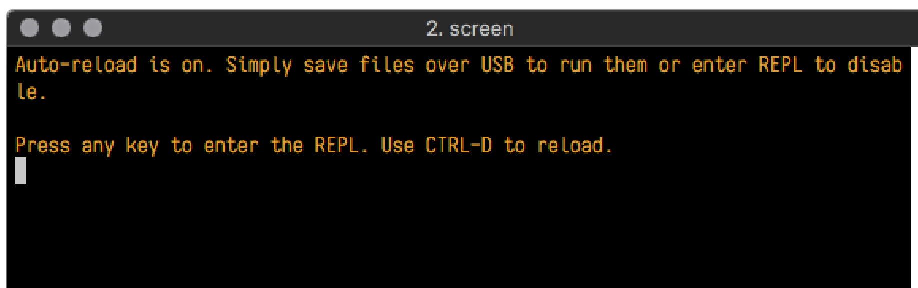
To use the REPL, you first need to be connected to the serial console. Once that connection has been established, you'll want to press **Ctrl** + **C**.

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.



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.



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

```
2. screen

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.

```
2. screen
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             framebuf          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.

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

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

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

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

The REPL can also be used to run code. Be aware that **any code you enter into the REPL isn't saved**

anywhere. If you're testing something new that you'd like to keep, make sure you have it saved somewhere on your computer as well!

Every programmer in every programming language starts with a piece of code that says, "Hello, World." 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.

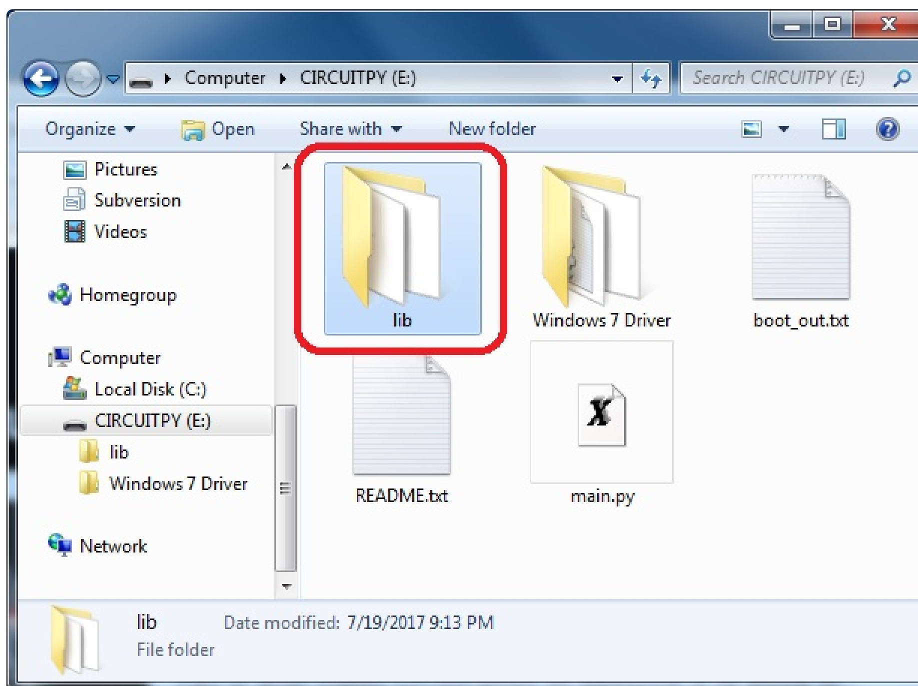
You can return to the REPL at any time!

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) are a great reference for how it all should work. In Python terms, we can place our library files in the **lib** directory because its 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.

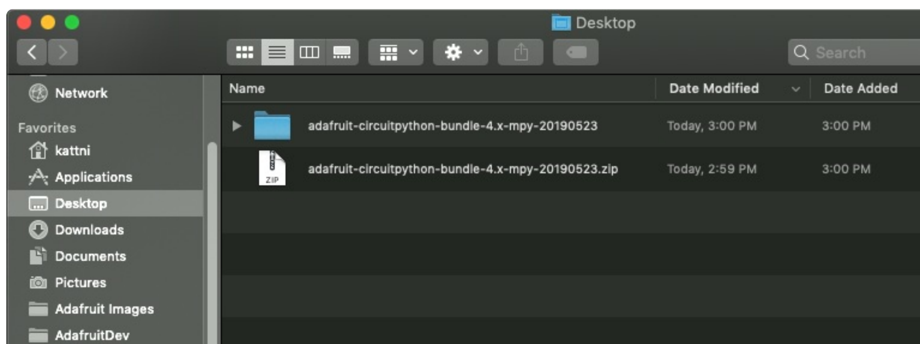
https://adafru.it/ENC

https://adafru.it/ENC

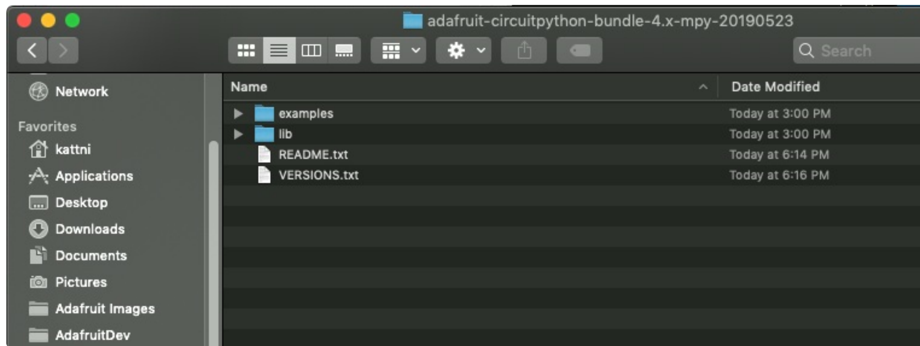
If you need another version, you can also visit the bundle release page (https://adafru.it/Ayy) which will let you select exactly what version you're looking for, as well as information about changes.

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.

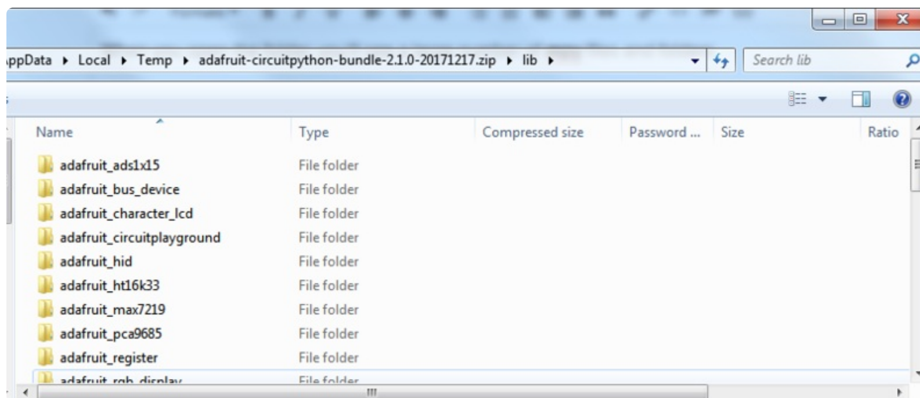
After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.



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.



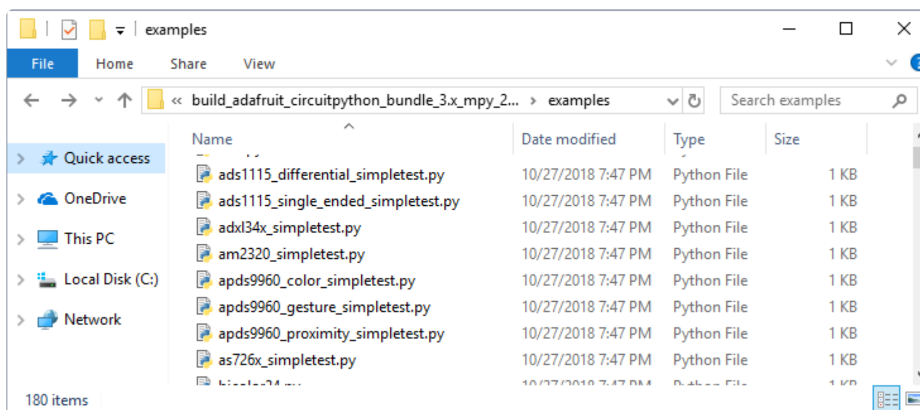
Now open the lib folder. When you open the folder, you'll see a large number of **mpy** files and folders



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.

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

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** folde 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.

```
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 disab
le.
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.

```
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 disab
le.
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 an older version of CircuitPython; where can I find compatible libraries?

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

- 2.x (https://adafru.it/FJA)
- 3.x (https://adafru.it/FJB)
- 4.x (https://adafru.it/QDL)
- 5.x (https://adafru.it/QDJ)

Is ESP8266 or ESP32 supported in CircuitPython? Why not?

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

https://learn.adafruit.com/welcome-to-circuitpython/circuitpython-for-esp8266 (https://adafru.it/CiG)

We do not support ESP32 because it does not have native USB. We do support ESP32-S2, which does.

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. (https://adafru.it/Dwa)

Is there asyncio support in CircuitPython?

We do not have asyncio support in CircuitPython at this time. However, `async` and `await` are turned on in many builds, and we are looking at how to use event loops and other constructs effectively and easily.

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

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

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://adafru.it/uap) (https://adafru.it/uap) for your version of CircuitPython.

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

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

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

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

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

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

You can download `mpy-cross` for your operating system from [https://adafruit-circuit-python.s3.amazonaws.com/index.html?prefix=bin/mpy-cross/](https://adafru.it/QDK) (https://adafru.it/QDK). Builds are available for Windows, macOS, x64 Linux, and Raspberry Pi Linux. Choose the latest `mpy-cross` whose version matches the version of CircuitPython you are using.

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

How do I check how much memory I have free?

```
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 AVRs such as ATmega328 or ATmega2560 run CircuitPython?

No.

Commonly Used Acronyms

CP or CPy = CircuitPython (https://adafru.it/cpy-welcome)
CPC = Circuit Playground Classic (https://adafru.it/ncE)
CPX = Circuit Playground Express (https://adafru.it/wpF)

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 5.x, 4.x, 3.x or 2.x, where can I find compatible libraries?

We are no longer building or supporting the CircuitPython 2.x, 3.x, 4.x or 5.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), the last available 3.x build here (https://adafru.it/FJB), the last available 4.x build here (https://adafru.it/QDL), and the last available 5.x build here (https://adafru.it/QDJ).

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) (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 `*boardname*BOOT` drive.

MakeCode

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

MacOS

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

Windows 10

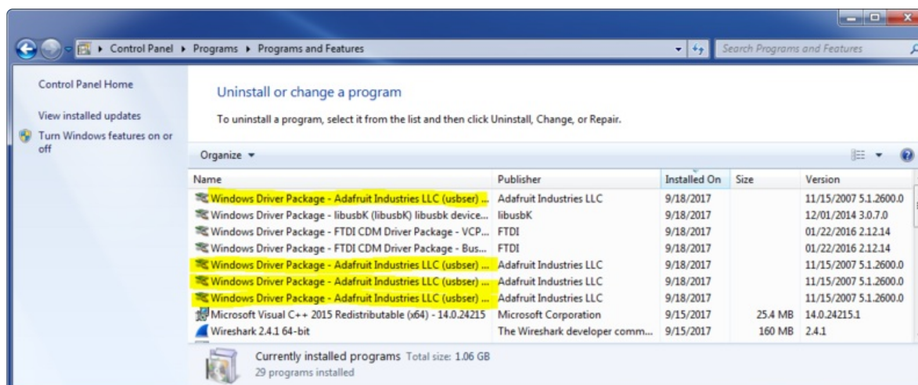
Did you install the Adafruit Windows Drivers package by mistake, or did you upgrade to Windows 10 with the driver package installed? You don't need to install this package on Windows 10 for most Adafruit boards. The old version (v1.5) can interfere with recognizing your device. Go to **Settings** ->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

Windows 7 or 8.1

Version 2.5.0.0 or later of the Adafruit Windows Drivers will fix the missing `*boardname*BOOT` drive problem on Windows 7 and 8.1. 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 ...".

We [recommend](https://adafru.it/Amd) (https://adafru.it/Amd) that you upgrade to Windows 10 if possible; an upgrade is probably still free for you: see the link.

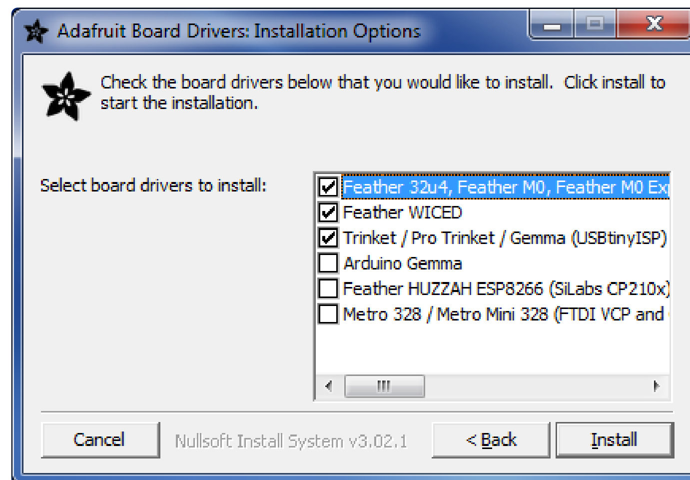


- Now install the new 2.5.0.0 (or higher) Adafruit Windows Drivers Package:

https://adafru.it/AB0

https://adafru.it/AB0

- 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 *boardname*BOOT drive.

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

Windows Explorer Locks Up When Accessing boardnameBOOT Drive

On Windows, several third-party programs 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.

Windows 7 and 8.1 Problems

Windows 7 and 8.1 can become confused about USB device installations. We recommend (https://adafru.it/Amd) that you upgrade to Windows 10 if possible; an upgrade is probably still free for you: see the link. If not, try cleaning up your USB devices with your board unplugged. Use Uwe Sieber's Device Cleanup Tool (https://adafru.it/RWd), which you must run as Administrator.

Serial Console in Mu Not Displaying Anything

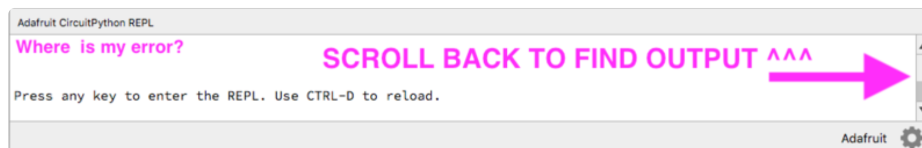
There are times when the serial console will accurately not display anything, such as, when no code is currently running, or when code with no serial output is already running before you open the console. However, if you find yourself in a situation where you feel it should be displaying something like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

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) to do this.

1. Connect to the CircuitPython REPL (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

https://adafru.it/Q5B

https://adafru.it/Q5B

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

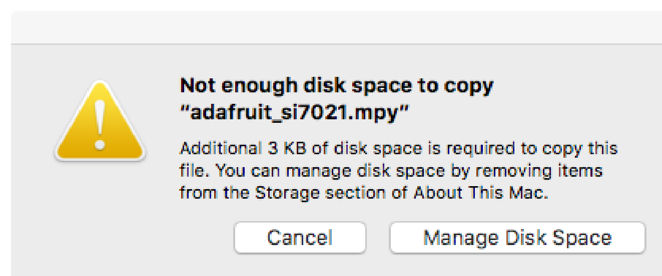
Delete something!

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) 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:

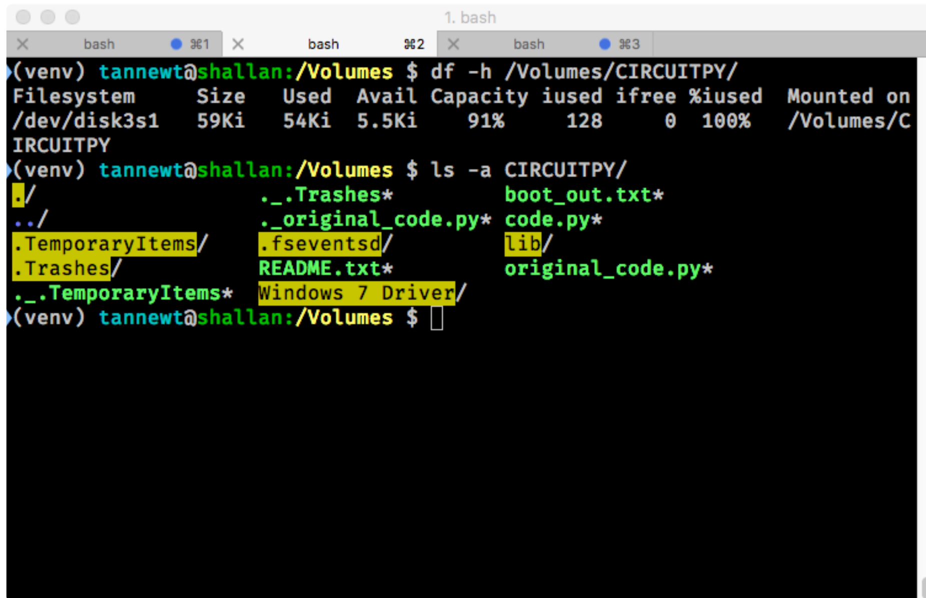
```
cp -rX folder_to_copy /Volumes/CIRCUITPY
```

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

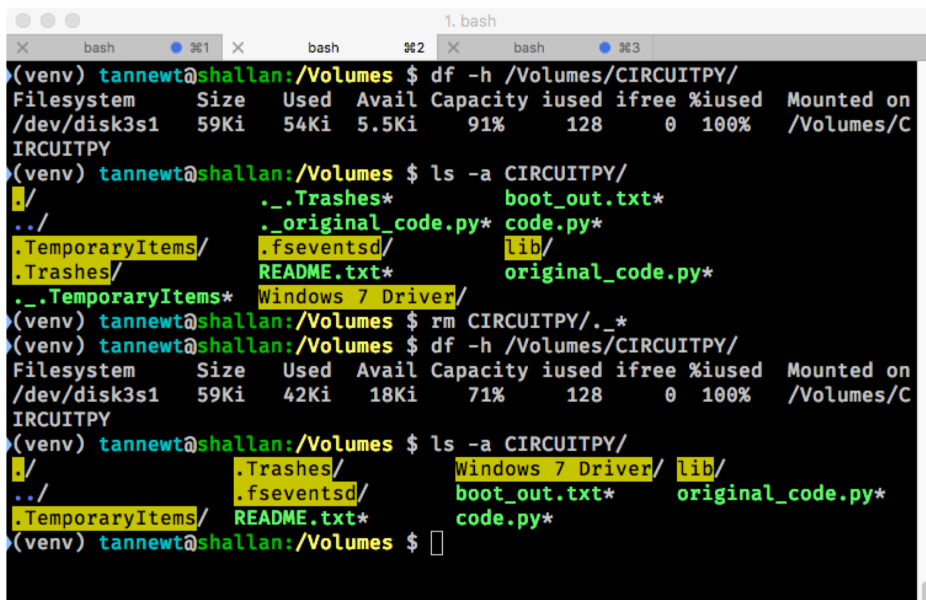
```
# if lib does not exist, you'll create a file named lib !
cp -X foo.mpy /Volumes/CIRCUITPY/lib
# This is safer, and will complain if a lib folder does not exist.
cp -X foo.mpy /Volumes/CIRCUITPY/lib/
```

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:



Lets remove the `._` files first.



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

Device locked up or boot looping

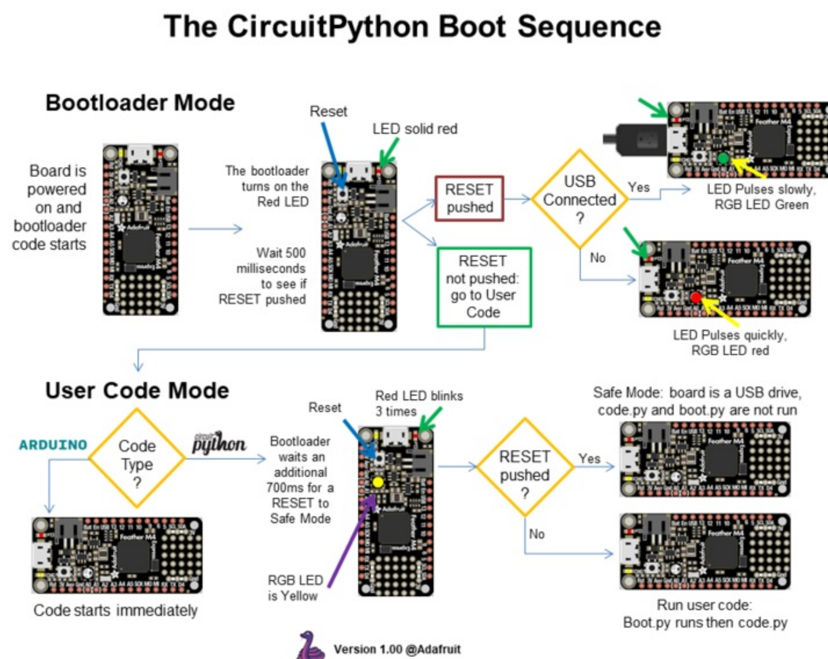
In rare cases, it may happen that something in your **code.py** or **boot.py** files causes the device to get locked up, or even go into a boot loop. These are not your everyday Python exceptions, typically it's the result of a deeper problem within CircuitPython. In this situation, it can be difficult to recover your device if **CIRCUITPY** is not allowing you to modify the **code.py** or **boot.py** files. Safe mode is one recovery option. When the device boots up in safe mode it will not run the **code.py** or **boot.py** scripts, but will still connect the **CIRCUITPY** drive so that you can remove or modify those files as needed.

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

For most devices:
Press the reset button, and then when the RGB status LED is yellow, press the reset button again.

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

Refer to the following diagram for boot sequence details:



Uninstalling CircuitPython

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

Maybe you tried CircuitPython and want to go back to MakeCode or Arduino? Not a problem

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

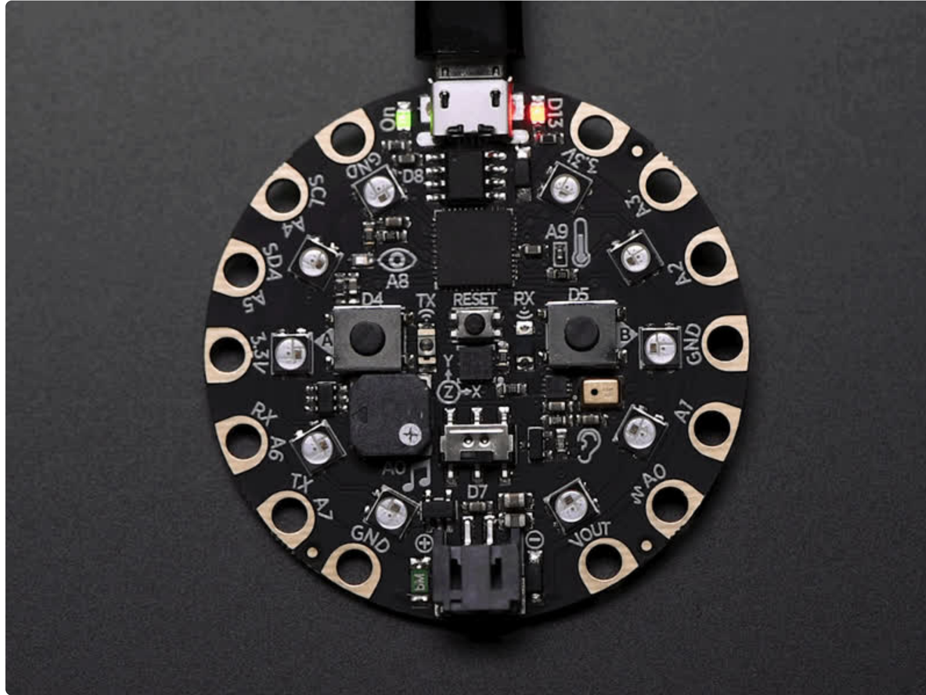
Backup Your Code

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.

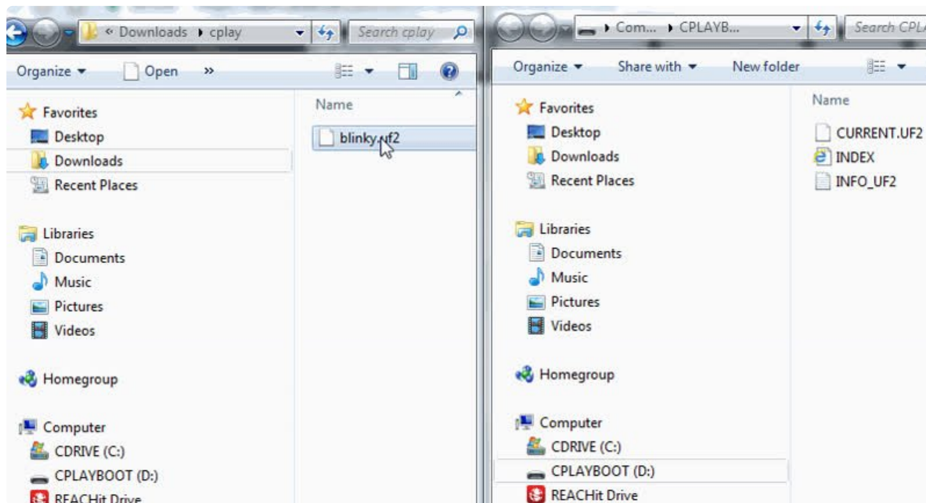
Moving Circuit Playground Express to MakeCode

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.



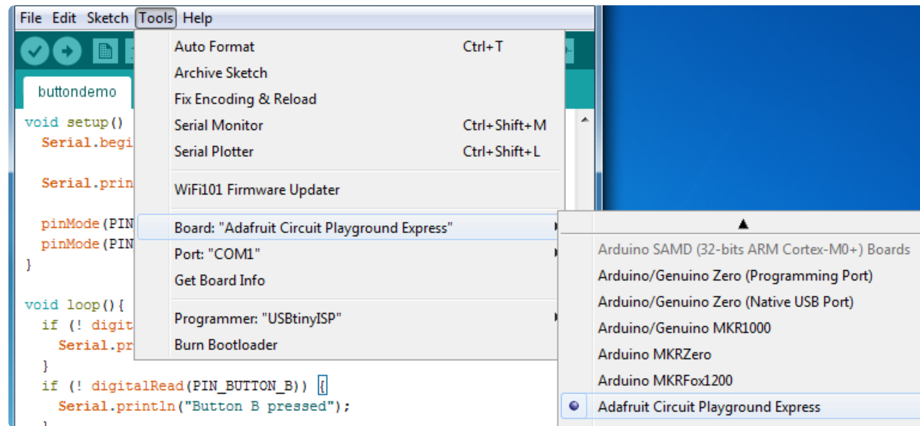
Your MakeCode is now running and CircuitPython has been removed. Going forward you only have to **single click** the reset button

Moving to Arduino

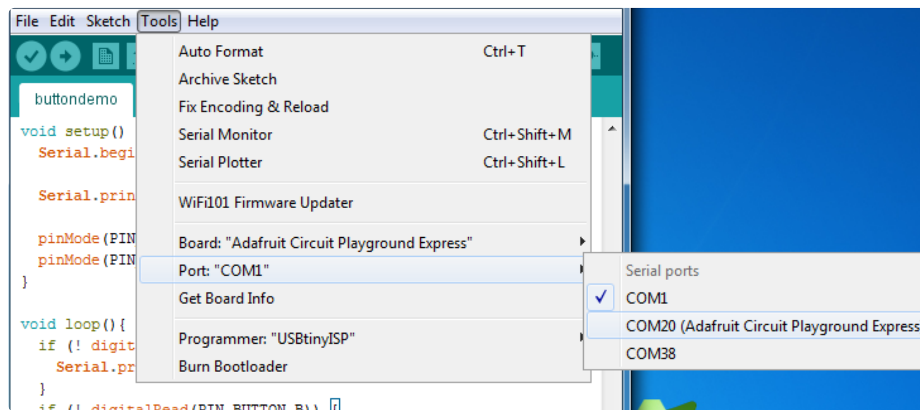
If you want to change your firmware to Arduino, it's also pretty easy.

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:

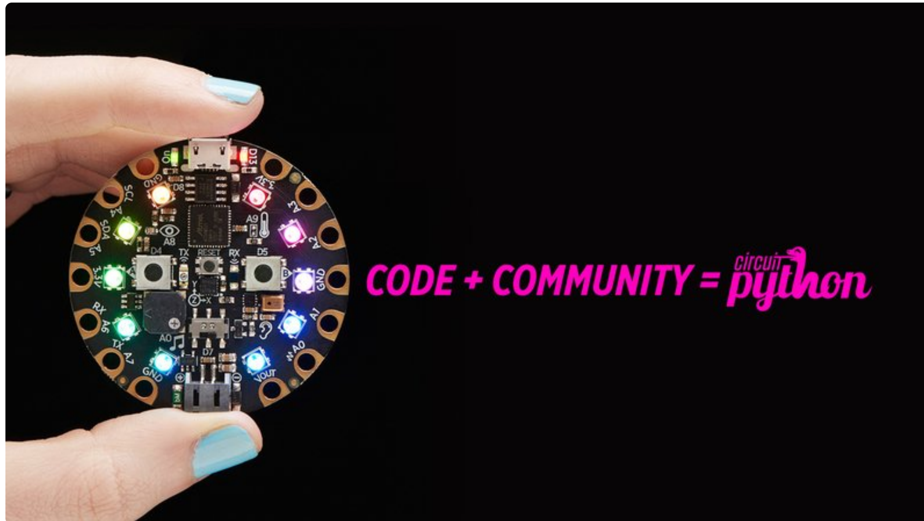
```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// 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

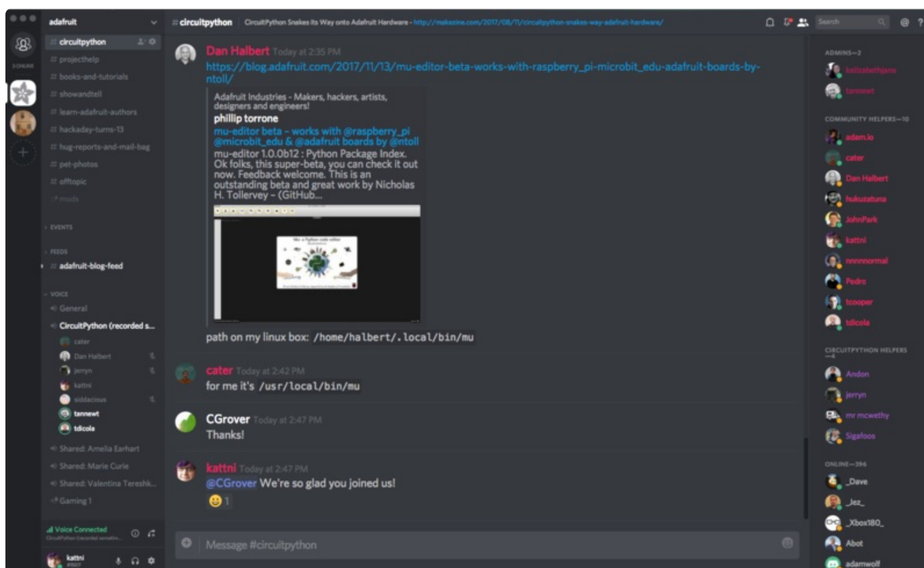
Welcome to the Community!



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 #help-with-projects 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.

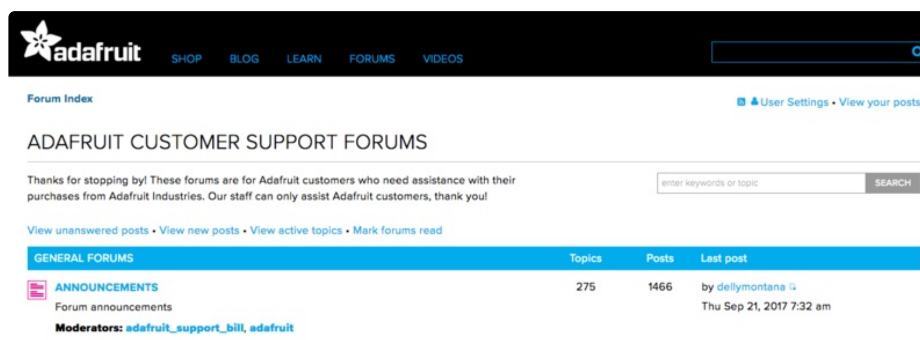
The help with CircuitPython channel is where to go with your CircuitPython questions. #help-with-circuitpython is there for new users and developers alike so feel free to ask a question or post a comment! Everyone of any experience level is welcome to join in on the conversation. We'd love to hear what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

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

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

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

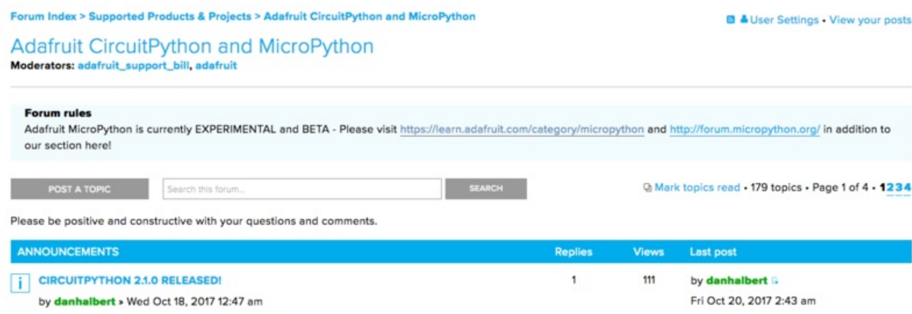
Adafruit Forums



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

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

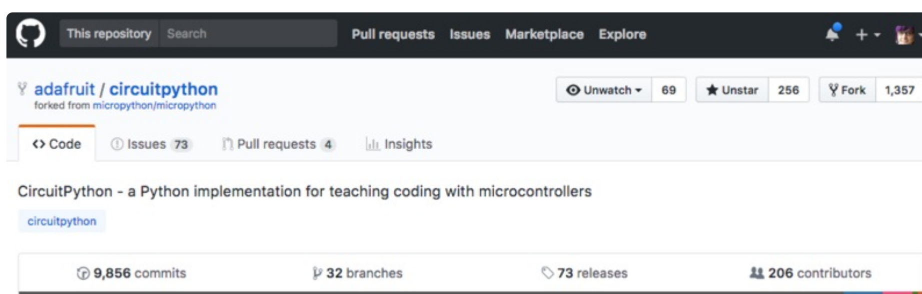
There are forum categories that cover all kinds of topics, including everything Adafruit. The [Adafruit CircuitPython and MicroPython](https://adafru.it/xXA) (https://adafru.it/xXA) category under "Supported Products & Projects" is the best place to post your CircuitPython questions.



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) (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) (https://adafru.it/Bee)", and you'll find a list that includes issues labeled "[good first issue](https://adafru.it/Bef) (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.

OneWire BusDevice driver good first issue
#338 opened 29 days ago by tannewt Long term
2

Feather M0 Adalogger does not have D8 or D7 good first issue
#323 opened on Oct 13 by ladyada 3.0
7

Audit and fix native API for methods that accept and ignore extra args. good first issue
#321 opened on Oct 12 by tannewt Long term

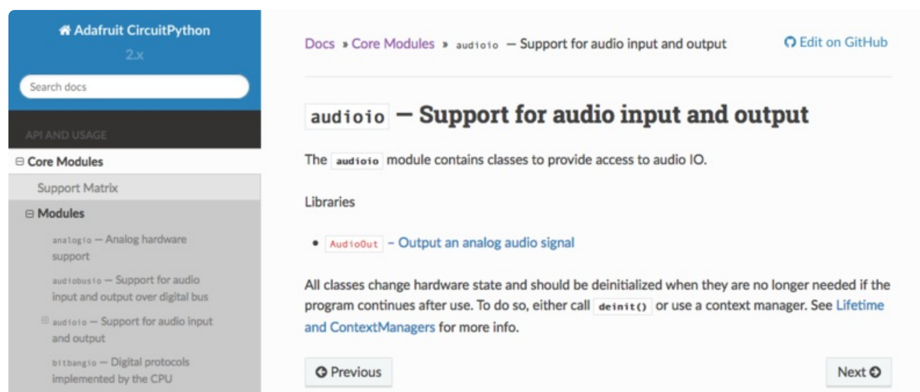
Already experienced and looking for a challenge? Checkout the rest of the issues list and you'll find plenty of ways to contribute. You'll find everything from new driver requests to core module updates. There's plenty of opportunities for everyone at any level!

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) is a 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). 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)

CircuitPython Playground

We have a few demos showing how to use the Circuit Playground Express with CircuitPython!

Click on any of the page links on the left side of the web page to try them out (e.g., CircuitPython NeoPixel, CircuitPython PWM, etc.)

CircuitPython Pins and Modules

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

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

CircuitPython Pins

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

`import board`

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

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

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

Here is the output for the QT Py.

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

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

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

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

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

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

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

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

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

I2C, SPI, and UART

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

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

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

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

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

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

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

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

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

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

What Are All the Available Names?

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

On a microcontroller board running CircuitPython, connect to the serial console. Then, save the following as **code.py** on your **CIRCUITPY** drive.

```
"""CircuitPython Essentials Pin Map Script"""
import microcontroller
import board

board_pins = []
for pin in dir(microcontroller.pin):
    if isinstance(getattr(microcontroller.pin, pin), microcontroller.Pin):
        pins = []
        for alias in dir(board):
            if getattr(board, alias) is getattr(microcontroller.pin, pin):
                pins.append("board.{}".format(alias))
        if len(pins) > 0:
            board_pins.append(" ".join(pins))
for pins in sorted(board_pins):
    print(pins)
```

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

```
board.A0 board.D0
board.A1 board.D1
board.A10 board.D10 board.MOSI
board.A2 board.D2
board.A3 board.D3
board.A6 board.D6 board.TX
board.A7 board.D7 board.RX
board.A8 board.D8 board.SCK
board.A9 board.D9 board.MISO
board.D4 board.SDA
board.D5 board.SCL
board.NEOPIXEL
board.NEOPIXEL_POWER
```

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

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

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

Microcontroller Pin Names

The pin names available to you in the CircuitPython `board` module are not the same as the names of the pins on the microcontroller itself. The board pin names are aliases to the microcontroller pin names. If you

look at the datasheet for your microcontroller, you'll likely find a pinout with a series of pin names, such as "PA18" or "GPIO5". If you want to get to the actual microcontroller pin name in CircuitPython, you'll need the `microcontroller.pin` module. As with `board`, you can run `dir(microcontroller.pin)` in the REPL to receive a list of the microcontroller pin names.

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

CircuitPython Built-In Modules

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

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

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

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

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

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

CircuitPython Built-Ins

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

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

Thing That Are Built In and Work

Flow Control

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

Math

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

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

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

Tuples, Lists, Arrays, and Dictionaries

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

Classes, Objects and Functions

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

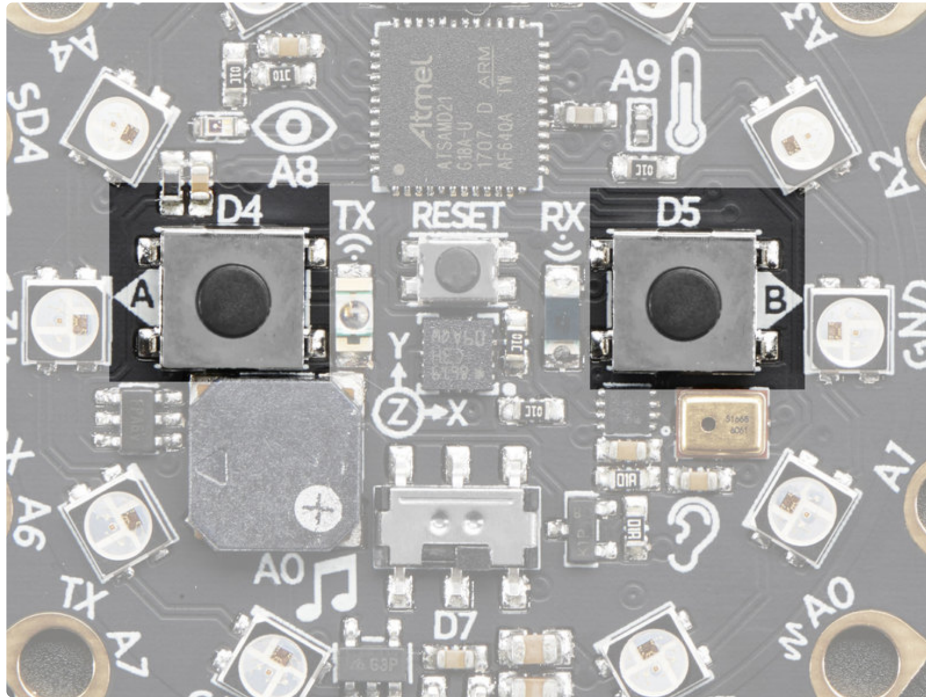
Though the following example uses the Circuit Playground Express to demonstrate, the code works exactly the same way with the Circuit Playground Bluefruit. Simply copy the code and follow along with your Circuit Playground Bluefruit!

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 =
digitalio.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:

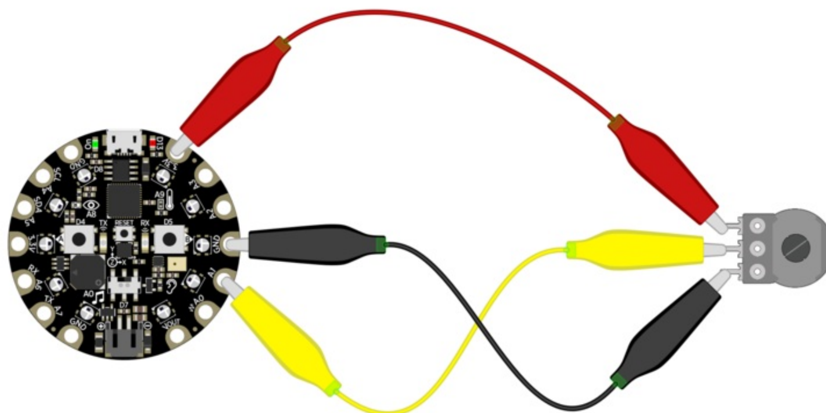
```
switch =
digitalio.DigitalInOut(board.SLIDE_SWITCH)
switch.direction = digitalio.Direction.INPUT
switch.pull = digitalio.Pull.UP
```

```
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!!

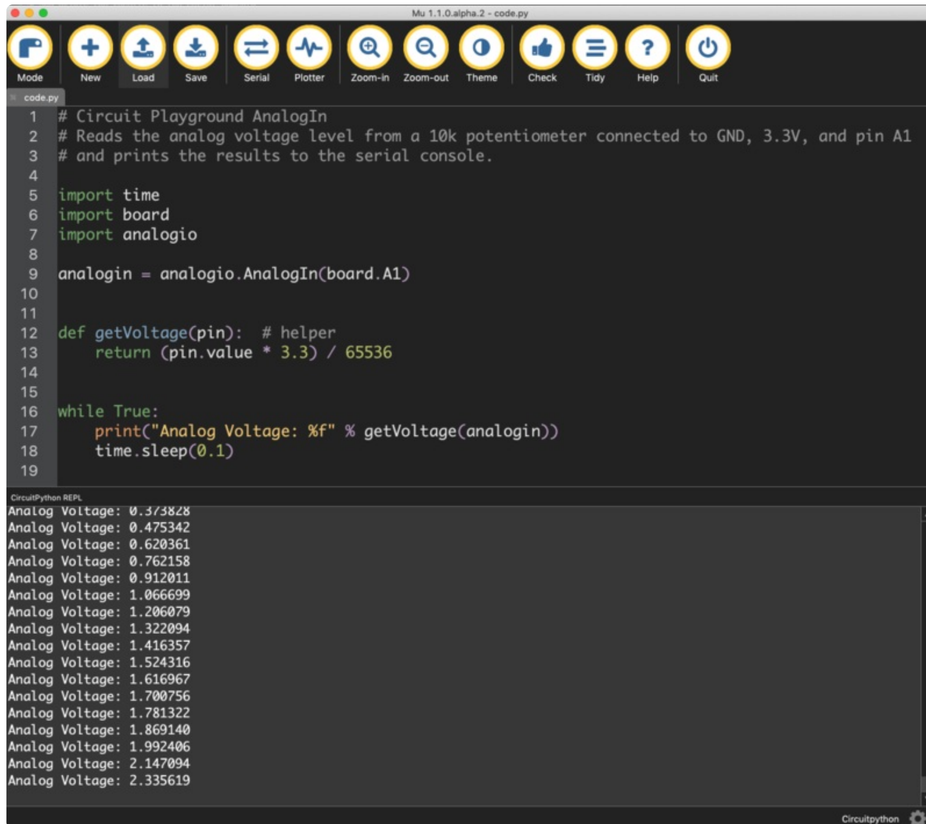
GetVoltage Helper

`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!


Mu 1.1.0.alpha.2 - code.py
Mode
New
Load
Save
Serial
Plotter
Zoom-in
Zoom-out
Theme
Check
Tidy
Help
Quit
code.py
# 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)
CircuitPython REPL
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 Analog Out

This example shows you how you can set the DAC (true analog output) on pin A0.

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

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

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

analog_out = AnalogOut(board.A0)

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

Creating an analog output

`analog_out = AnalogOut(A0)`

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

Setting the analog output

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

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

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

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

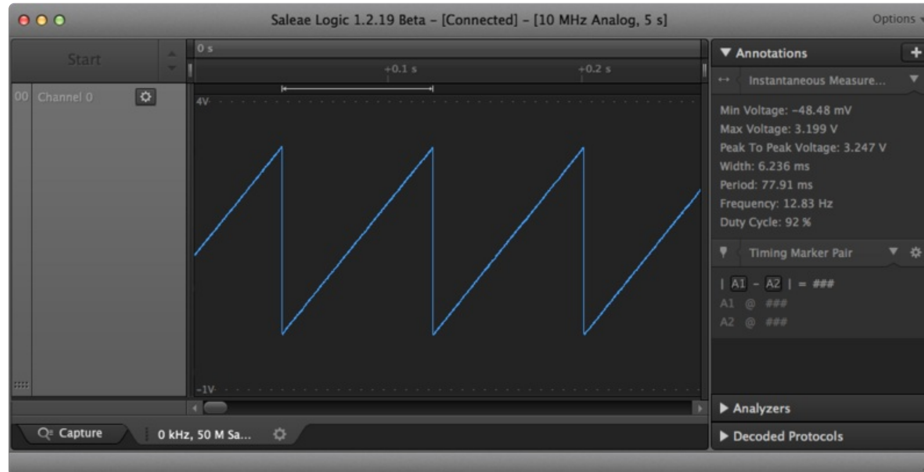
Main Loop

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

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

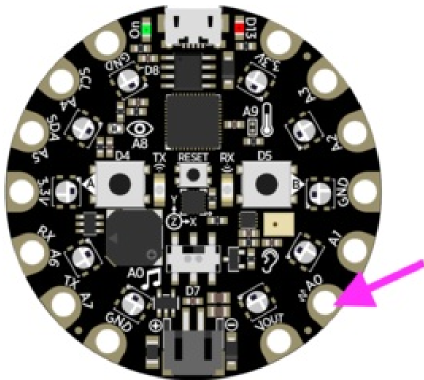
Express boards like the Circuit Playground Express, Metro M0 Express, ItsyBitsy M0 Express, ItsyBitsy M4 Express, Metro M4 Express, Feather M4 Express, or Feather M0 Express have more code space and can perform audio playback capabilities via the DAC. QT Py M0, Gemma M0 and Trinket M0 cannot!

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



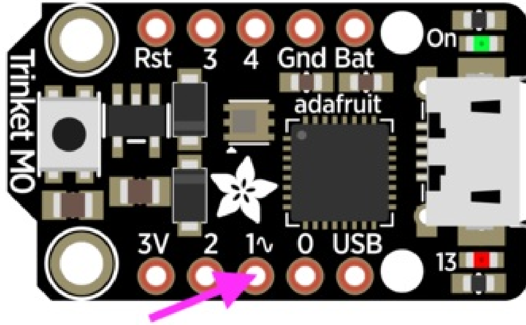
Find the pin

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



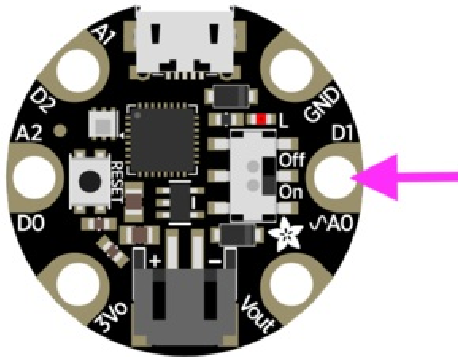
Circuit Playground Express

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



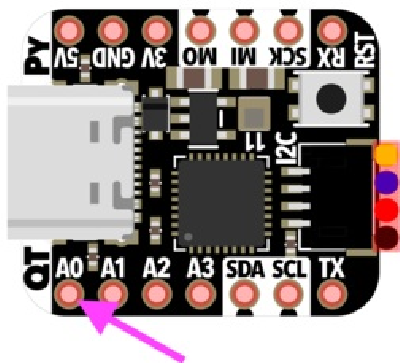
Trinket M0

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



Gemma M0

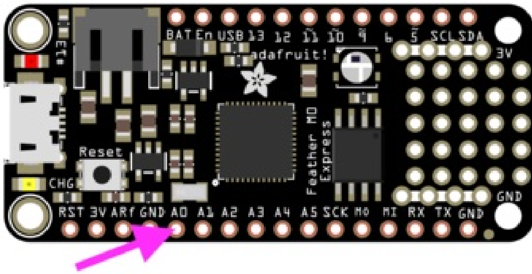
A0 is located in the middle of the right side of the board next to the On/Off switch.



QT Py M0

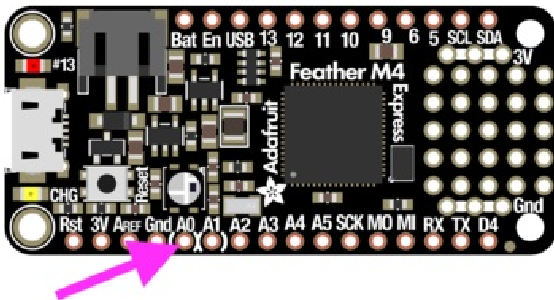
A0 is located next to the USB port, by the "QT" label on the board silk.

Feather M0 Express



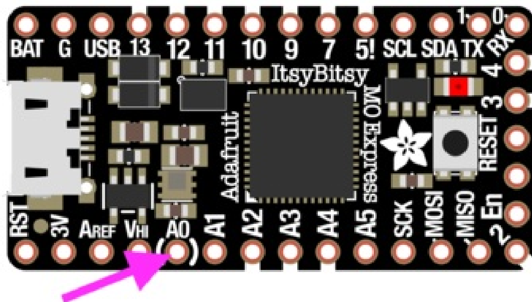
A0 is located between GND and A1 on the opposite side of the board from the battery connector, towards the end with the Reset button.

Feather M4 Express



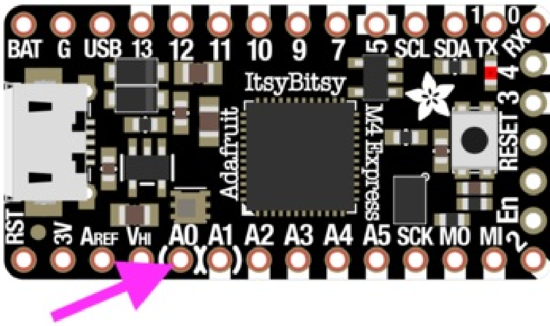
A0 is located between GND and A1 on the opposite side of the board from the battery connector, towards the end with the Reset button, and the pin pad has left and right white parenthesis markings around it

ItsyBitsy M0 Express



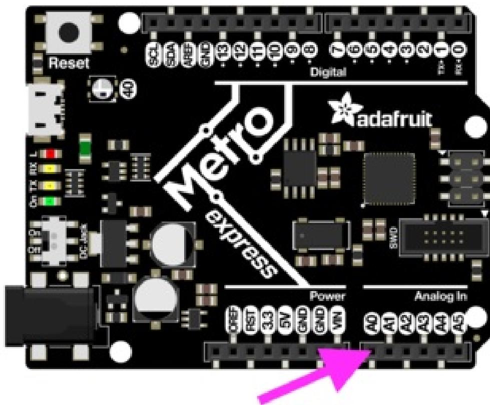
A0 is located between VHI and A1, near the "A" in "Adafruit", and the pin pad has left and right white parenthesis markings around it.

ItsyBitsy M4 Express



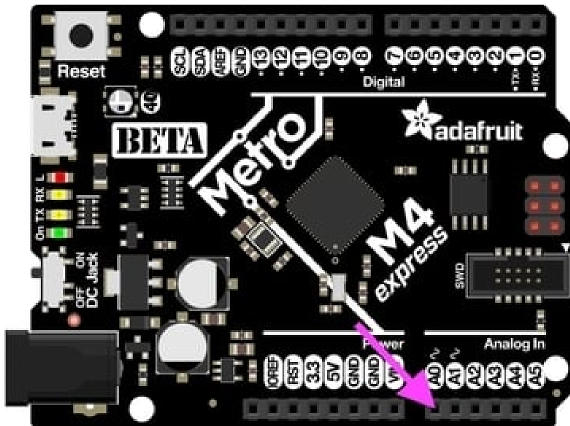
A0 is located between VHI and A1, and the pin pad has left and right white parenthesis markings around it.

Metro M0 Express



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

Metro M4 Express



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

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

CircuitPython PWM

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

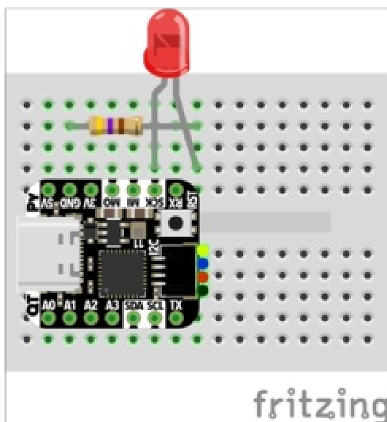
Nearly every pin has PWM support! For example, all ATSAMD21 board have an **A0** pin which is 'true' analog out and *does not* have PWM support.

PWM with Fixed Frequency

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

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

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



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

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

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

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

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

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

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

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

Create a PWM Output

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

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

Main Loop

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

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

PWM Output with Variable Frequency

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

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

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

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

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

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

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

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

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

If you have `simpleio` library loaded into your /lib folder on your board, we have a nice little helper that makes a tone for you on a piezo with a single command.

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

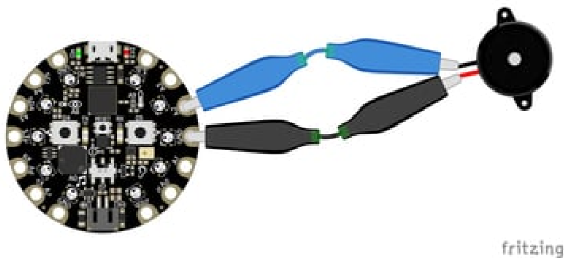
To use with the Metro M4 Express, ItsyBitsy M4 Express or the Feather M4 Express, you must comment out the `simpleio.tone(board.A2, f, 0.25)` line and uncomment the `simpleio.tone(board.A1, f, 0.25)` line. A2 is not a supported PWM pin on the M4 boards!

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

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

As you can see, it's much simpler!

Wire it up

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

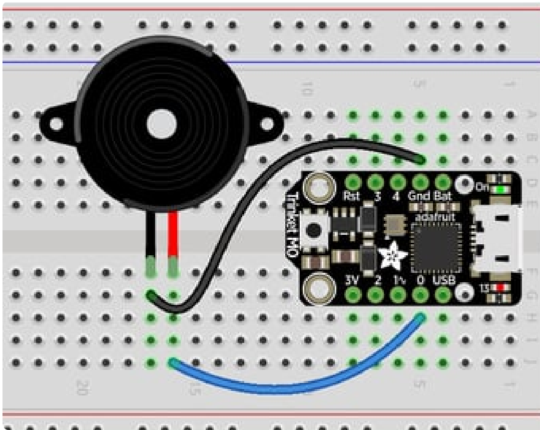


Circuit Playground Express

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

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

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



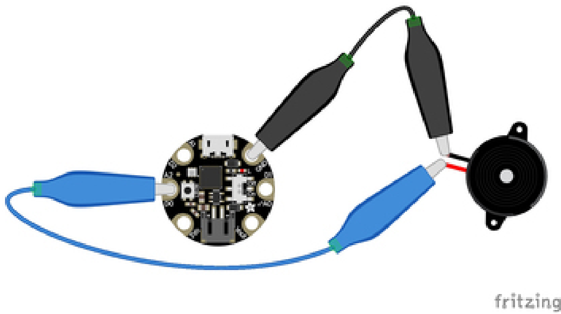
Trinket M0

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

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

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

There is NO PWM on: A0, D1.

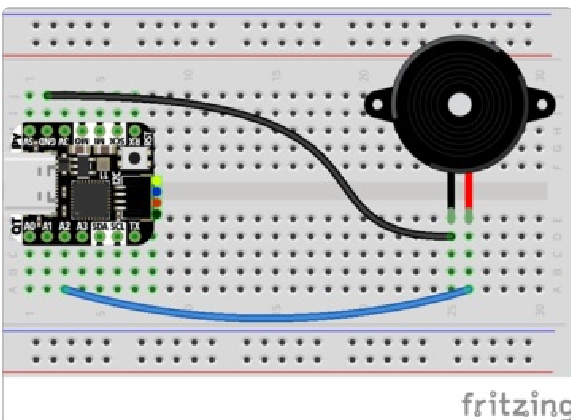


Gemma M0

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

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

There is NO PWM on: A0, D1.

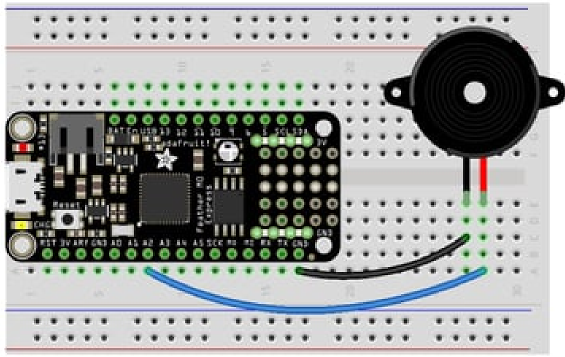


QT Py M0

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

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

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

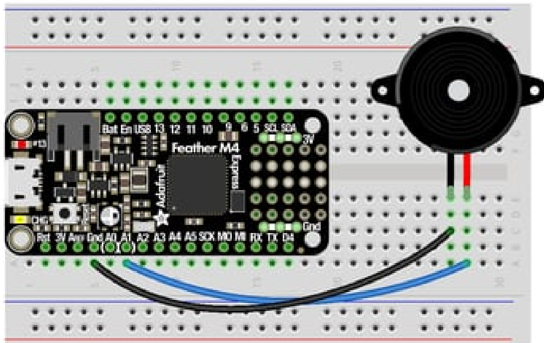


Feather M0 Express

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

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

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



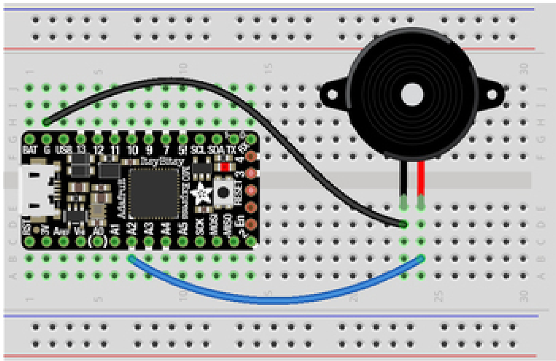
Feather M4 Express

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

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

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

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

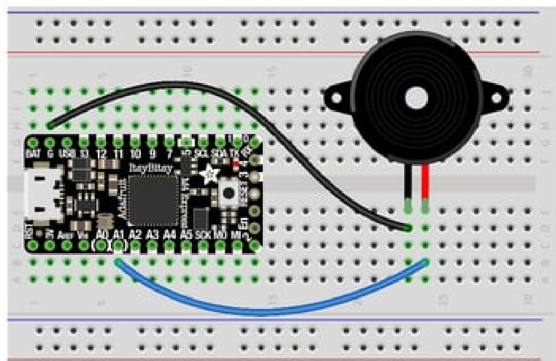


ItsyBitsy M0 Express

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

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

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



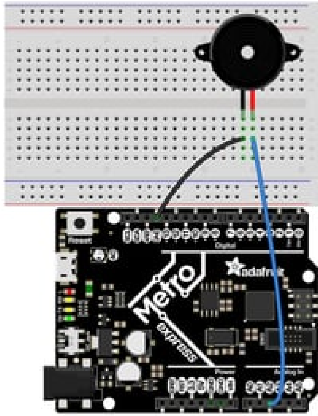
ItsyBitsy M4 Express

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

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

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

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

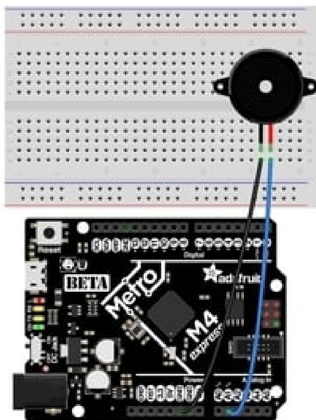


Metro M0 Express

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

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

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



Metro M4 Express

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

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

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

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

Where's My PWM?

Want to check to see which pins have PWM yourself? We've written this handy script! It attempts to setup PWM on every pin available, and lets you know which ones work and which ones don't. Check it out!

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

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

CircuitPython Servo

In order to use servos, we take advantage of `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

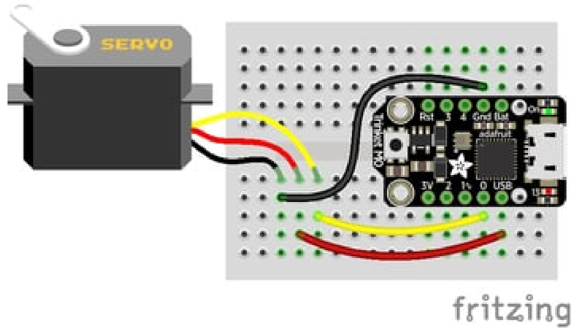
Servos will only work on PWM-capable pins! Check your board details to verify which pins have PWM outputs.

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.

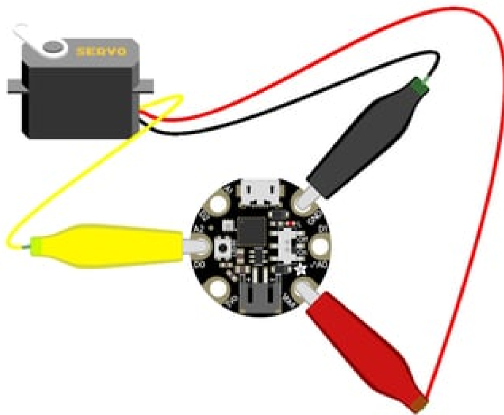
Connect the servo's **red** power wire to 5V power, USB power is good for a servo or two. For more than that, you'll need an external battery pack. Do not use 3.3V for powering a servo!

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.

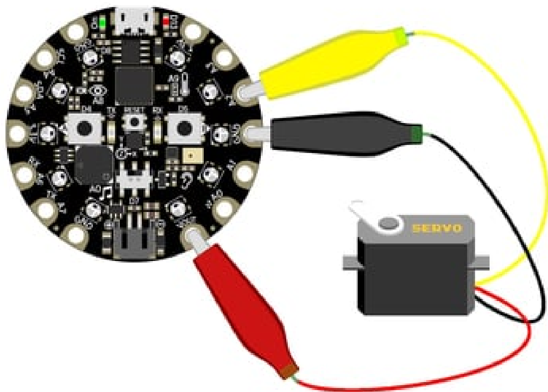


For example, to wire a servo to **Trinket**, connect the ground wire to **GND**, the power wire to **USB**, and the signal wire to **0**.

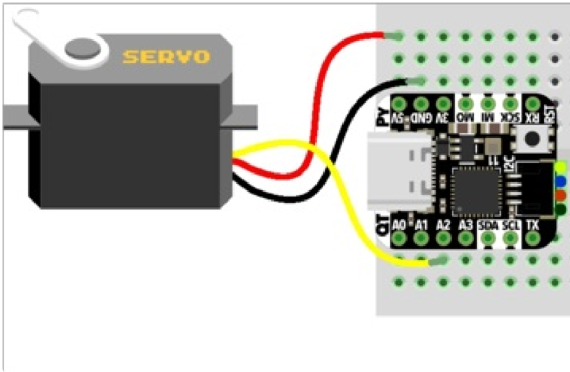
Remember, **A2 on Trinket is labeled "0"**.



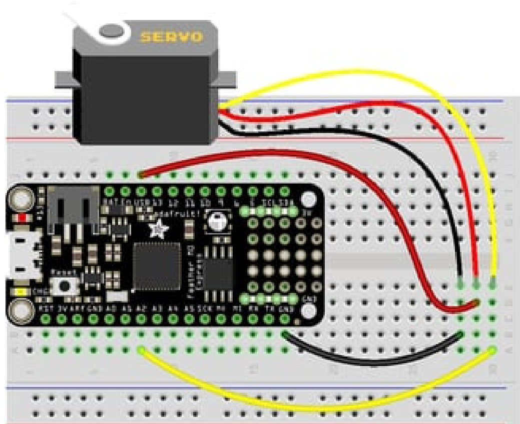
For **Gemma**, use jumper wire alligator clips to connect the ground wire to **GND**, the power wire to **VOUT**, and the signal wire to **A2**.



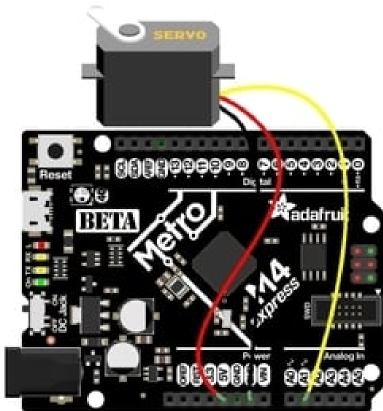
For **Circuit Playground Express and Circuit Playground Bluefruit**, use jumper wire alligator clips to connect the ground wire to **GND**, the power wire to **VOUT**, and the signal wire to **A2**.



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



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



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

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

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

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

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

Continuous Servo Code

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

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

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

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

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

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

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

Pretty simple!

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

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

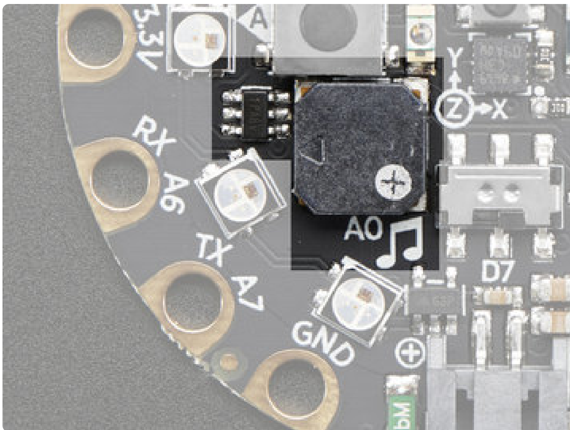
For more detailed information on using servos with CircuitPython, check out the [CircuitPython section of the servo guide](https://adafru.it/Bei) (https://adafru.it/Bei)!

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.

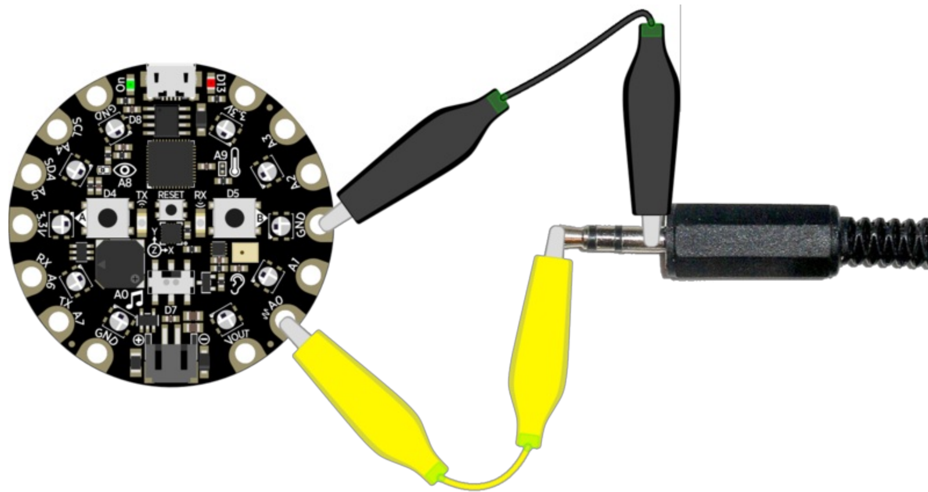
There are **two** ways to get audio output, one is via the small built in speaker. The other is by using alligator clips to connect a headphone or powered speaker to the **A0/AUDIO** pin.



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 / length) * (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:

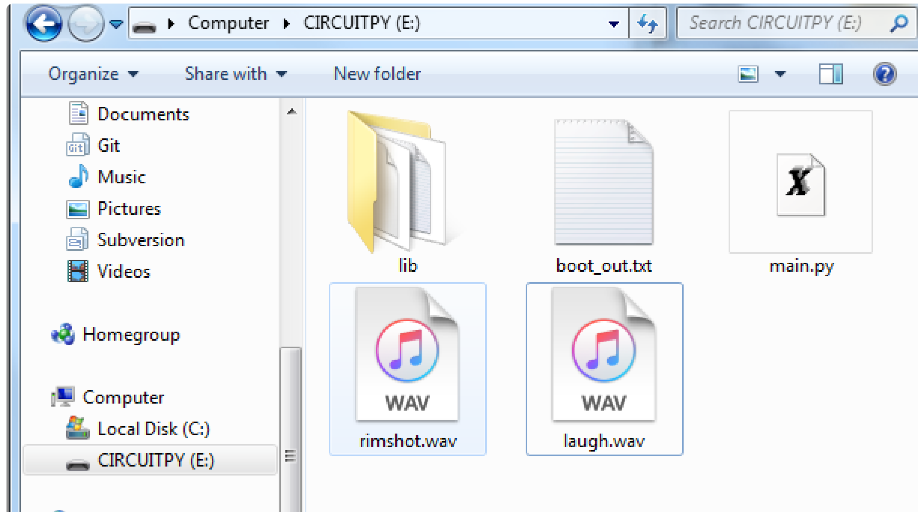
https://adafru.it/zFK

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:

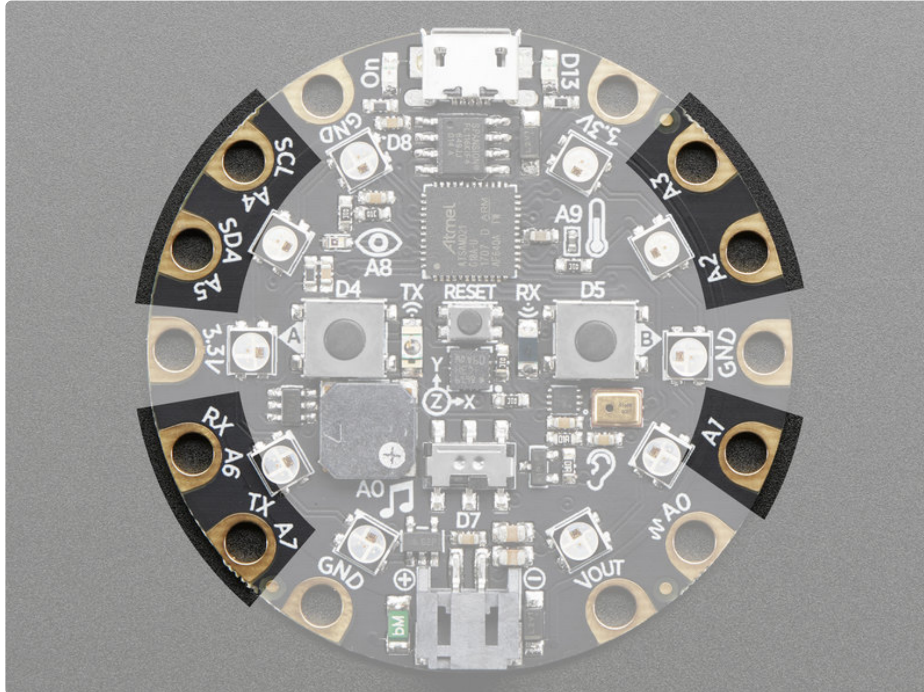
```
audio.play(wave)
while audio.playing:
    pass
```

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 site, 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:

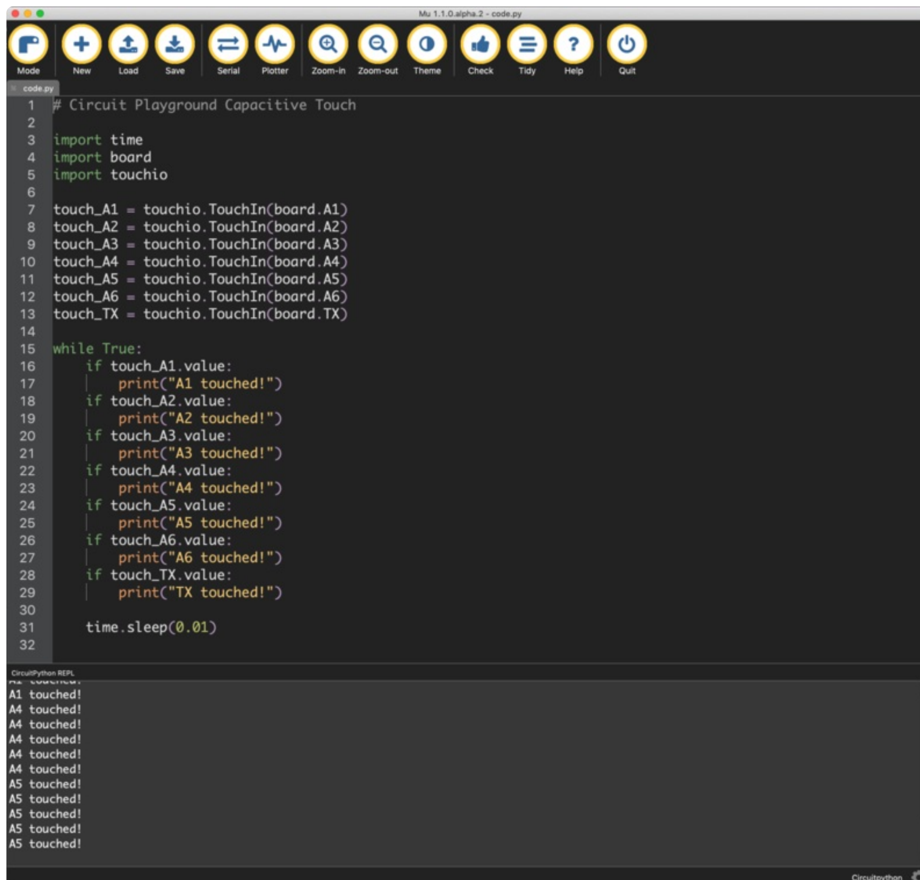
```
touch_A1 = touchio.TouchIn(board.A1)
touch_A2 = touchio.TouchIn(board.A2)
touch_A3 = touchio.TouchIn(board.A3)
touch_A4 = touchio.TouchIn(board.A4)
touch_A5 = touchio.TouchIn(board.A5)
touch_A6 = touchio.TouchIn(board.A6)
touch_TX = touchio.TouchIn(board.TX)
```

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

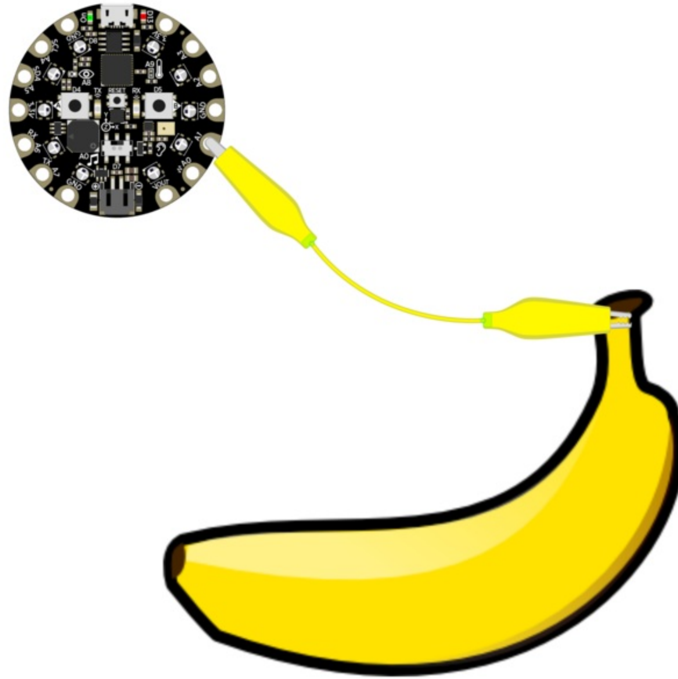
Main Loop

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.



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

Copper tape can be an interesting addition to your toolbox. The tape itself is made of thin pure copper so its extremely flexible and can take on nearly any shape. You can easily...

$5.95

In Stock

Add to Cart

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

Copper tape can be an interesting addition to your toolbox. The tape itself is made of thin pure copper so its extremely flexible and can take on nearly any shape. You can easily...

$19.95

In Stock

Add to Cart

Small Alligator Clip Test Lead (set of 12)

Connect this to that without soldering using these handy mini alligator clip test leads. 15" cables with alligator clip on each end, color coded. You get 12 pieces in 6 colors....

$3.95

In Stock

Add to Cart

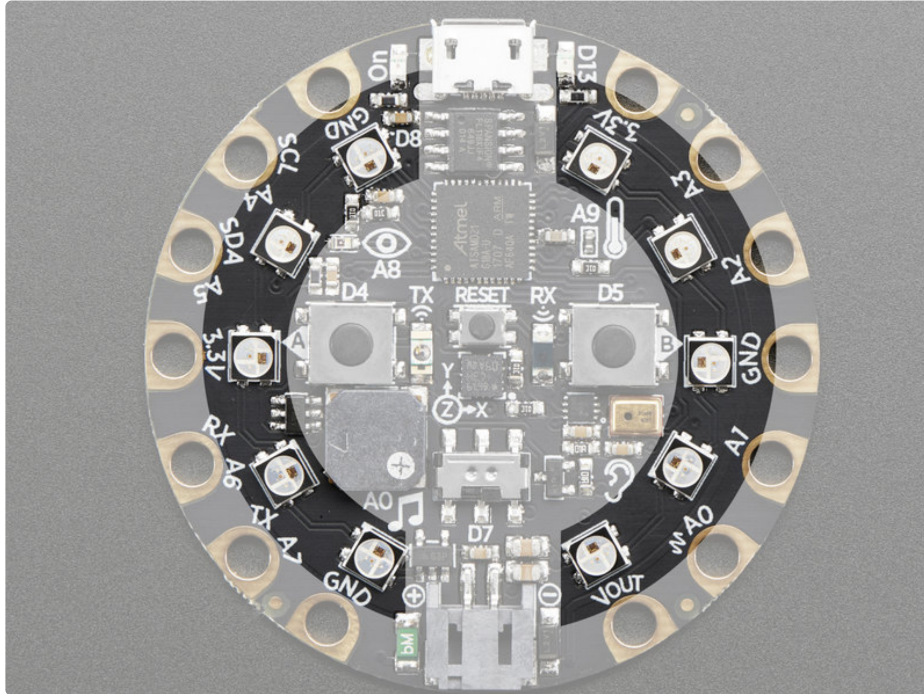
Capacitive Touch and the Audio Pin on Circuit Playground Bluefruit

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)
```

CircuitPython NeoPixel

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) (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:
```

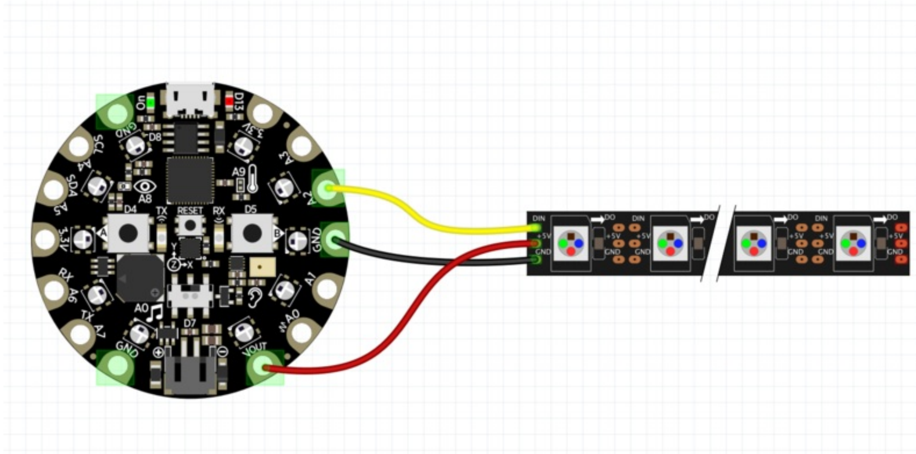
```
        color_chase(RED, 0.1)  # Increase the number to slow down the color chase
        color_chase(YELLOW, 0.1)
        color_chase(GREEN, 0.1)
        color_chase(CYAN, 0.1)
        color_chase(BLUE, 0.1)
        color_chase(PURPLE, 0.1)
        color_chase(OFF, 0.1)

    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 > 5.5V 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

> 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). If you need help installing the library, check out the CircuitPython Libraries page (https://adafru.it/ABU).

Wire It Up

You'll need to solder up your DotStars first. Verify your connection is on the **DATA INPUT** or **DI** and **CLOCK INPUT** or **CI** side. Plugging into the DATA OUT/DO or CLOCK OUT/CO side is a common mistake! The connections are labeled and some formats have arrows to indicate the direction the data must flow. Always verify your wiring with a visual inspection, as the order of the connections can differ from strip to strip!

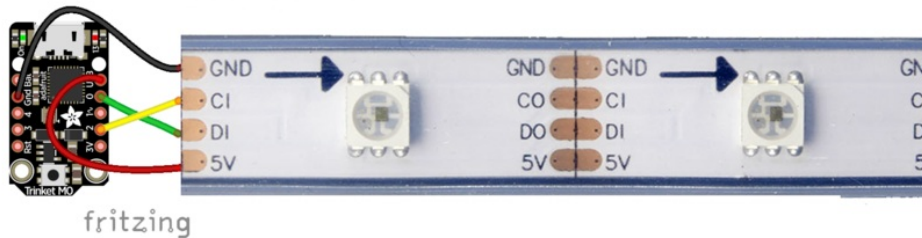
For powering the pixels from the board, the 3.3V regulator output can handle about 500mA peak which is about 50 pixels with 'average' use. If you want really bright lights and a lot of pixels, we recommend powering direct from an external power source.

- On Gemma M0 and Circuit Playground Express this is the **Vout** pad - that pad has direct power from USB or the battery, depending on which is higher voltage.
- On Trinket M0, Feather M0 Express, Feather M4 Express, ItsyBitsy M0 Express and ItsyBitsy M4 Express the **USB** or **BAT** pins will give you direct power from the USB port or battery.
- On Metro M0 Express and Metro M4 Express, use the **5V** pin regardless of whether it's powered via

USB or the DC jack.
- On QT Py M0, use the **5V** pin.

If the power to the DotStars is greater than 5.5V you may have some difficulty driving some strips, in which case you may need to lower the voltage to 4.5-5V or use a level shifter.

Do not use the VIN pin directly on Metro M0 Express or Metro M4 Express! The voltage can reach 9V and this can destroy your DotStars!



Note that the wire ordering on your DotStar strip or shape may not exactly match the diagram above. Check the markings to verify which pin is DIN, CIN, 5V and GND

The Code

This example includes multiple visual effects. Copy and paste the code into **code.py** using your favorite editor, and save the file.

```
"""CircuitPython Essentials DotStar example"""
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 colorwheel(pos):
    # Input a value 0 to 255 to get a color value.
    # The colours are a transition r - g - b - back to r.
    if pos < 0 or pos > 255:
        return (0, 0, 0)
    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()
    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[4::6] = [BLUE] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)
    pixels[5::6] = [PURPLE] * (num_pixels // 6)
    pixels.show()
    time.sleep(wait)


def rainbow_cycle(wait):
    for j in range(255):
        for i in range(num_pixels):
            rc_index = (i * 256 // num_pixels) + j
            pixels[i] = colorwheel(rc_index & 255)
        pixels.show()
        time.sleep(wait)


RED = (255, 0, 0)
YELLOW = (255, 150, 0)
ORANGE = (255, 40, 0)
GREEN = (0, 255, 0)
TEAL = (0, 255, 120)
CYAN = (0, 255, 255)
BLUE = (0, 0, 255)
PURPLE = (180, 0, 255)
MAGENTA = (255, 0, 20)
WHITE = (255, 255, 255)

while True:
    # Change this number to change how long it stays on each solid color.
    color_fill(RED, 0.5)
    color_fill(YELLOW, 0.5)
    color_fill(ORANGE, 0.5)
    color_fill(GREEN, 0.5)
    color_fill(TEAL, 0.5)
    color_fill(CYAN, 0.5)
    color_fill(BLUE, 0.5)
    color_fill(PURPLE, 0.5)
    color_fill(MAGENTA, 0.5)
    color_fill(WHITE, 0.5)

    # Increase or decrease this to speed up or slow down the animation.
    slice_alternating(0.1)

    color_fill(WHITE, 0.5)

    # Increase or decrease this to speed up or slow down the animation.
    slice_rainbow(0.1)

    time.sleep(0.5)

    # Increase this number to slow down the rainbow animation.
    rainbow_cycle(0)
```

We've chosen pins A1 and A2, but these are not SPI pins on all boards. DotStars respond faster

when using hardware SPI!

Create the LED

The first thing we'll do is create the LED object. The DotStar object has three required arguments and two optional arguments. You are required to set the pin you're using for data, set the pin you'll be using for clock, and provide the number of pixels you intend to use. You can optionally set `brightness` and `auto_write`.

DotStars can be driven by any two pins. We've chosen **A1** for clock and **A2** for data. To set the pins, include the pin names at the beginning of the object creation, in this case `board.A1` and `board.A2`.

To provide the number of pixels, assign the variable `num_pixels` to the number of pixels you'd like to use. In this example, we're using a strip of `72`.

We've chosen to set `brightness=0.1`, or 10%.

By default, `auto_write=True`, meaning any changes you make to your pixels will be sent automatically. Since `True` is the default, if you use that setting, you don't need to include it in your LED object at all. We've chosen to set `auto_write=False`. If you set `auto_write=False`, you must include `pixels.show()` each time you'd like to send data to your pixels. This makes your code more complicated, but it can make your LED animations faster!

DotStar Helpers

We've included a few helper functions to create the super fun visual effects found in this code.

First is `wheel()` which we just learned with the [Internal RGB LED](https://adafru.it/Bel) (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!

```
"""CircuitPython Essentials Hardware SPI pin verification script"""
import board
import busio


def is_hardware_spi(clock_pin, data_pin):
    try:
        p = busio.SPI(clock_pin, data_pin)
        p.deinit()
        return True
    except ValueError:
        return False


# Provide the two pins you intend to use.
if is_hardware_spi(board.A1, board.A2):
    print("This pin combination is hardware SPI!")
else:
    print("This pin combination isn't hardware SPI.")
```

Read the Docs

For a more in depth look at what `dotstar` can do, check out DotStar on Read the Docs (https://adafru.it/C4d).

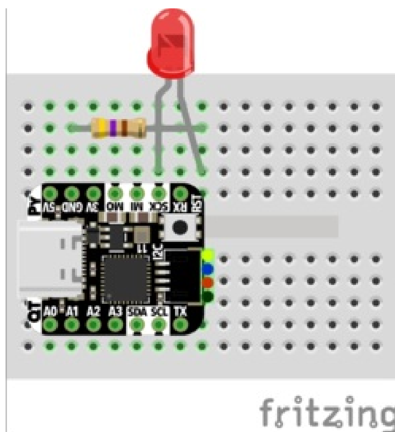
CircuitPython UART Serial

In addition to the USB-serial connection you use for the REPL, there is also a *hardware* UART you can use. This is handy to talk to UART devices like GPSs, some sensors, or other microcontrollers!

This quick-start example shows how you can create a UART device for communicating with hardware serial devices.

To use this example, you'll need something to generate the UART data. We've used a GPS! Note that the GPS will give you UART data without getting a fix on your location. You can use this example right from your desk! You'll have data to read, it simply won't include your actual location.

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



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

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

```
"""CircuitPython Essentials UART Serial example"""
import board
import busio
import digitalio

# For most CircuitPython boards:
led = digitalio.DigitalInOut(board.LED)
# For QT Py M0:
# led = digitalio.DigitalInOut(board.SCK)
led.direction = digitalio.Direction.OUTPUT

uart = busio.UART(board.TX, board.RX, baudrate=9600)

while True:
    data = uart.read(32)  # read up to 32 bytes
    # print(data)  # this is a bytearray type

    if data is not None:
        led.value = True

        # convert bytearray to string
        data_string = ''.join([chr(b) for b in data])
        print(data_string, end="")

        led.value = False
```

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

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

The Code

First we create the UART object. We provide the pins we'd like to use, `board.TX` and `board.RX`, and we set the `baudrate=9600`. While these pins are labeled on most of the boards, be aware that RX and TX are not labeled on Gemma, and are labeled on the bottom of Trinket. See the diagrams below for help with finding the correct pins on your board.

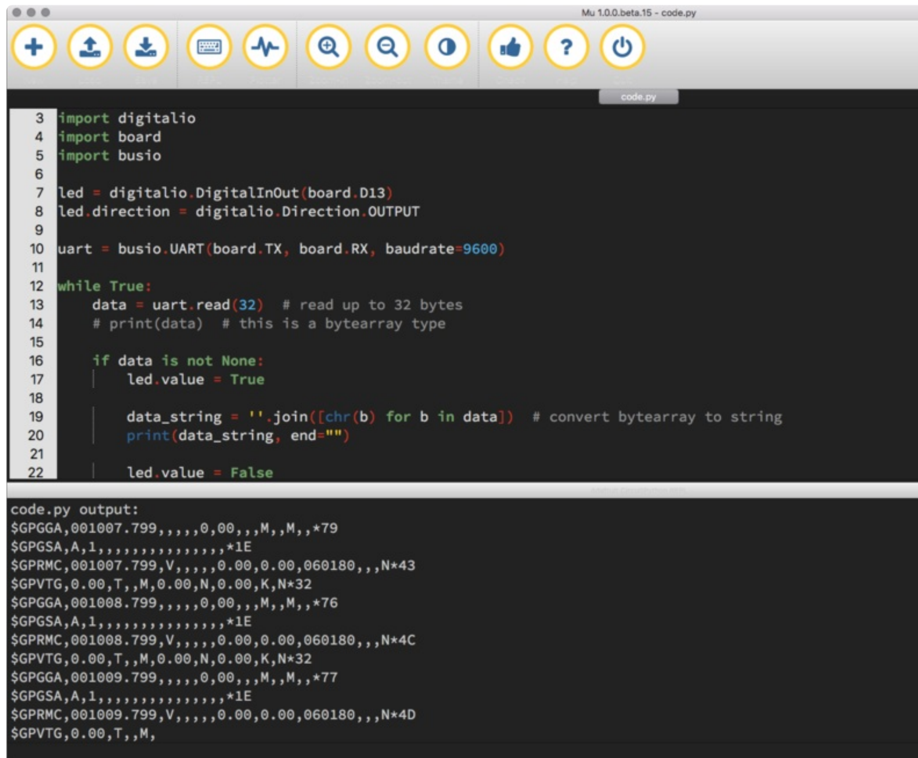
Once the object is created you read data in with `read(numbytes)` where you can specify the max number of bytes. It will return a byte array type object if anything was received already. Note it will always return immediately because there is an internal buffer! So read as much data as you can 'digest'.

If there is no data available, `read()` will return `None`, so check for that before continuing.

The data that is returned is in a byte array, if you want to convert it to a string, you can use this handy line of code which will run `chr()` on each byte:

`datastr = ''.join([chr(b) for b in data]) # convert bytearray to string`

Your results will look something like this:



For more information about the data you're reading and the Ultimate GPS, check out the Ultimate GPS guide: https://learn.adafruit.com/adafruit-ultimate-gps

Wire It Up

You'll need a couple of things to connect the GPS to your board.

For Gemma M0 and Circuit Playground Express, you can use use alligator clips to connect to the Flora Ultimate GPS Module.

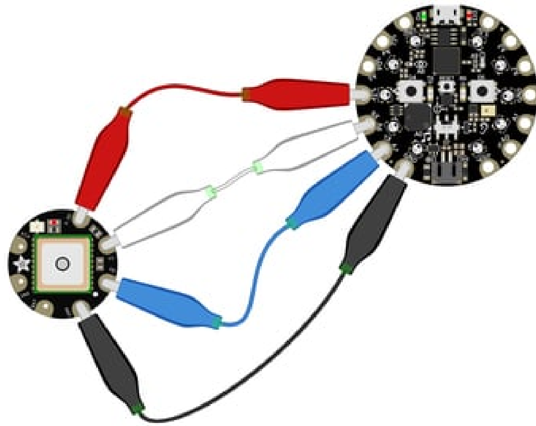
For Trinket M0, Feather M0 Express, Metro M0 Express and ItsyBitsy M0 Express, you'll need a breadboard and jumper wires to connect to the Ultimate GPS Breakout.

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

- The **black** wire connects between the **ground** pins.
- The **red** wire connects between the **power** pins on the GPS and your board.
- The **blue** wire connects from **TX** on the GPS to **RX** on your board.
- The **white** wire connects from **RX** on the GPS to **TX** on your board.

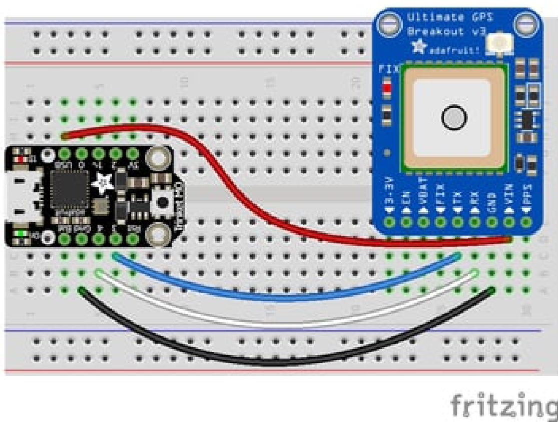
Check out the list below for a diagram of your specific board!

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



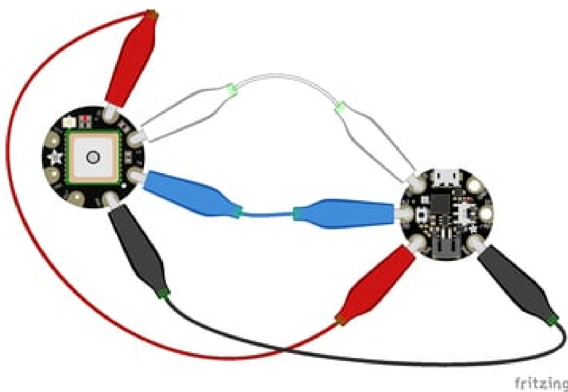
Circuit Playground Express and Circuit Playground Bluefruit

- Connect **3.3v** on your CPX to **3.3v** on your GPS.
- Connect **GND** on your CPX to **GND** on your GPS.
- Connect **RX/A6** on your CPX to **TX** on your GPS.
- Connect **TX/A7** on your CPX to **RX** on your GPS.



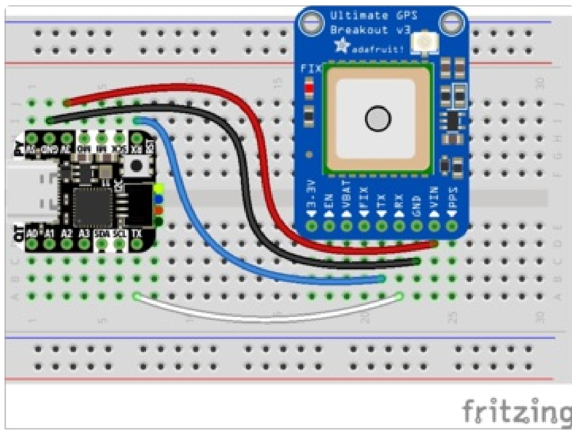
Trinket M0

- Connect **USB** on the Trinket to **VIN** on the GPS.
- Connect **Gnd** on the Trinket to **GND** on the GPS.
- Connect **D3** on the Trinket to **TX** on the GPS.
- Connect **D4** on the Trinket to **RX** on the GPS.



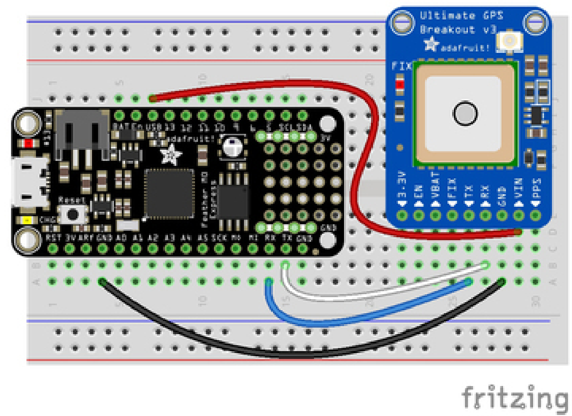
Gemma M0

- Connect **3vo** on the Gemma to **3.3v** on the GPS.
- Connect **GND** on the Gemma to **GND** on the GPS.
- Connect **A1/D2** on the Gemma to **TX** on the GPS.
- Connect **A2/D0** on the Gemma to **RX** on the GPS.



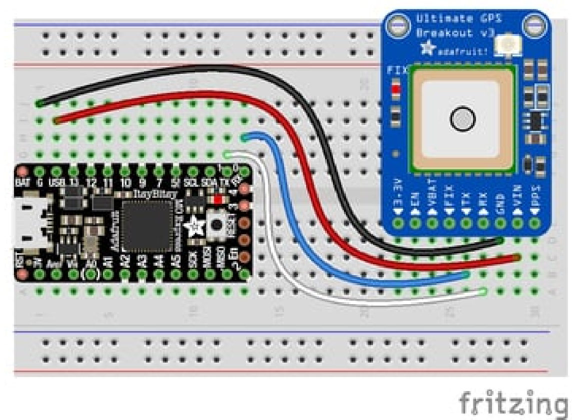
QT Py M0

- Connect **3V** on the QT Py to **VIN** on the GPS.
- Connect **GND** on the QT Py to **GND** on the GPS.
- Connect **RX** on the QT Py to **TX** on the GPS.
- Connect **TX** on the QT Py to **RX** on the GPS.



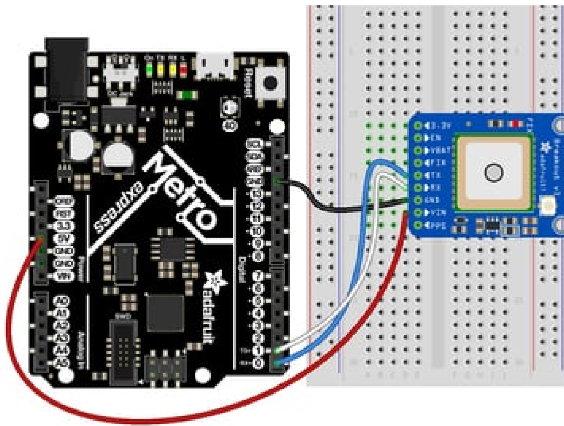
Feather M0 Express and Feather M4 Express

- Connect **USB** on the Feather to **VIN** on the GPS.
- Connect **GND** on the Feather to **GND** on the GPS.
- Connect **RX** on the Feather to **TX** on the GPS.
- Connect **TX** on the Feather to **RX** on the GPS.



ItsyBitsy M0 Express and ItsyBitsy M4 Express

- Connect **USB** on the ItsyBitsy to **VIN** on the GPS
- Connect **G** on the ItsyBitsy to **GND** on the GPS.
- Connect **RX/0** on the ItsyBitsy to **TX** on the GPS.
- Connect **TX/1** on the ItsyBitsy to **RX** on the GPS.



Metro M0 Express and Metro M4 Express

- Connect **5V** on the Metro to **VIN** on the GPS.
- Connect **GND** on the Metro to **GND** on the GPS.
- Connect **RX/D0** on the Metro to **TX** on the GPS.
- Connect **TX/D1** on the Metro to **RX** on the GPS.

Where's my UART?

On the SAMD21, we have the flexibility of using a wide range of pins for UART. Compare this to some chips like the ESP8266 with *fixed* UART pins. The good news is you can use many but not *all* pins. Given the large number of SAMD boards we have, its impossible to guarantee anything other than the labeled 'TX' and 'RX'. So, if you want some other setup, or multiple UARTs, how will you find those pins? Easy! We've written a handy script.

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)

```
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
```

```
"""CircuitPython Essentials UART possible pin-pair identifying script"""
import board
import busio
from microcontroller import Pin


def is_hardware_uart(tx, rx):
    try:
        p = busio.UART(tx, rx)
        p.deinit()
        return True
    except ValueError:
        return False


def get_unique_pins():
    exclude = ['NEOPIXEL', 'APA102_MOSI', 'APA102_SCK']
    pins = [pin for pin in [
        getattr(board, p) for p in dir(board) if p not in exclude]
            if isinstance(pin, Pin)]
    unique = []
    for p in pins:
        if p not in unique:
            unique.append(p)
    return unique


for tx_pin in get_unique_pins():
    for rx_pin in get_unique_pins():
        if rx_pin is tx_pin:
            continue
        if is_hardware_uart(tx_pin, rx_pin):
            print("RX pin:", rx_pin, "\t TX pin:", tx_pin)
```

Trinket M0: Create UART before I2C

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

```
>>> import board
>>> uart = board.UART()   # Uses pins 4 and 3 for TX and TX, baudrate 9600.
>>> i2c = board.I2C()     # Uses pins 2 and 0 for SCL and SDA.

# or alternatively,
```

Creating the I2C object first does not work:

```
>>> import board
>>> i2c = board.I2C()     # Uses pins 2 and 0 for SCL and SDA.
>>> uart = board.UART()   # Uses pins 4 and 3 for TX and TX, baudrate 9600.
Traceback (most recent call last):
File "", line 1, in
ValueError: Invalid pins
```

CircuitPython I2C

I2C is a 2-wire protocol for communicating with simple sensors and devices, meaning it uses two connections for transmitting and receiving data. There are many I2C devices available and they're really easy to use with CircuitPython. We have libraries available for many I2C devices in the library bundle (https://adafru.it/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 Adafruit TSL2591 (https://adafru.it/dGE), a common, low-cost light sensor. While the exact code we're running is specific to the TSL2591 the overall process is the same for just about any I2C sensor or device.

You'll need the **adafruit_tsl2591.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). If you need help installing the library, check out the CircuitPython Libraries page (https://adafru.it/ABU).

These examples will use the TSL2591 lux sensor breakout. The first thing you'll want to do is get the sensor connected so your board has I2C to talk to.

Wire It Up

You'll need a couple of things to connect the TSL2591 to your board. The TSL2591 comes with STEMMA QT / QWIIC connectors on it, which makes it super simple to wire it up. No further soldering required!

For Gemma M0, Circuit Playground Express and Circuit Playground Bluefruit, you can use use the STEMMA QT to alligator clips cable (https://adafru.it/KKa) to connect to the TSL2591.

For Trinket M0, Feather M0 and M4 Express, Metro M0 and M4 Express and ItsyBitsy M0 and M4 Express, you'll need a breadboard and STEMMA QT to male jumper wires cable (https://adafru.it/FA-) to connect to the TSL2591.

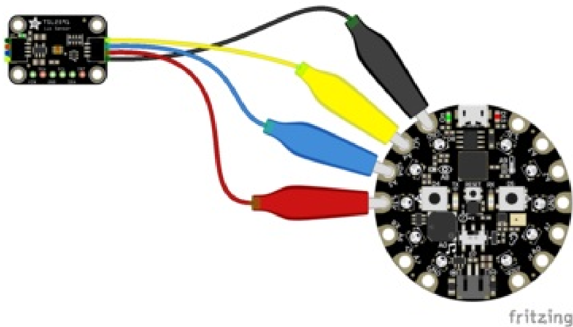
For QT Py M0, you'll need a STEMMA QT cable (https://adafru.it/FNS) to connect to the TSL2591.

We've included diagrams show you how to connect the TSL2591 to your board. In these diagrams, the wire colors match the STEMMA QT cables and connect to the same pins on each board.

- The **black** wire connects from **GND** on the TSL2591 to **ground** on your board.
- The **red** wire connects from **VIN** on the TSL2591 to **power** on your board.
- The **yellow** wire connects from **SCL** on the TSL2591 to **SCL** on your board.
- The **blue** wire connects from **SDA** on the TSL2591 to **SDA** on your board.

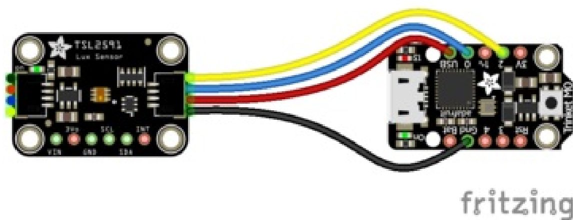
Check out the list below for a diagram of your specific board!

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



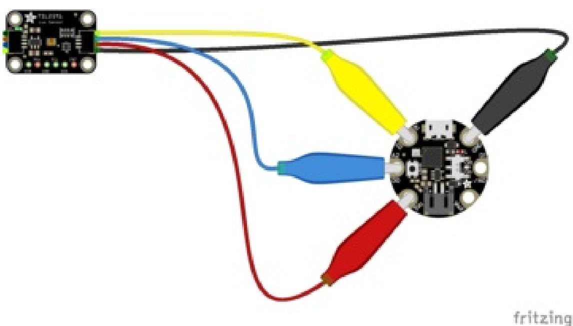
Circuit Playground Express and Circuit Playground Bluefruit

- Connect **3.3v** on your CPX to **3.3v** on your TSL2591.
- Connect **GND** on your CPX to **GND** on your TSL2591.
- Connect **SCL/A4** on your CPX to **SCL** on your TSL2591.
- Connect **SDL/A5** on your CPX to **SDA** on your TSL2591.



Trinket M0

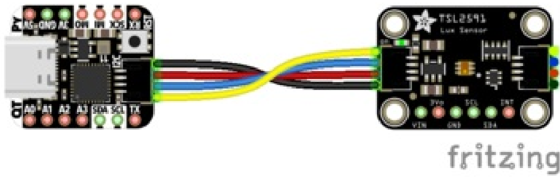
- Connect **USB** on the Trinket to **VIN** on the TSL2591.
- Connect **Gnd** on the Trinket to **GND** on the TSL2591.
- Connect **D2** on the Trinket to **SCL** on the TSL2591.
- Connect **D0** on the Trinket to **SDA** on the TSL2591.



Gemma M0

- Connect **3vo** on the Gemma to **3V** on the TSL2591.
- Connect **GND** on the Gemma to **GND** on the TSL2591.
- Connect **A1/D2** on the Gemma to **SCL** on the TSL2591.
- Connect **A2/D0** on the Gemma to **SDA** on the TSL2591.

QT Py M0



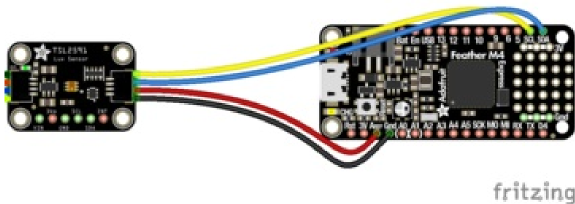
If using the STEMMA QT cable:

- Connect the **STEMMA QT cable** from **the connector on the QT Py** to **the connector on the TSL2591.**

Alternatively, if using a breadboard:

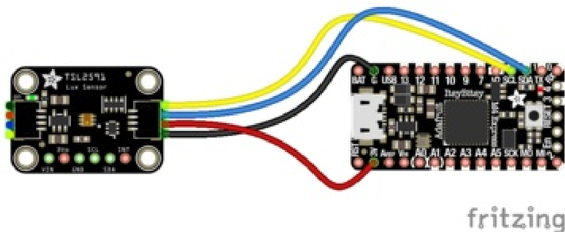
- Connect **3V** on the QT Py to **VIN** on the TSL2591.
- Connect **GND** on the QT Py to **GND** on the TSL2591.
- Connect **SCL** on the QT Py to **SCL** on the TSL2591.
- Connect **SDA** on the QT Py to **SDA** on the TSL2591.

Feather M0 Express and Feather M4 Express

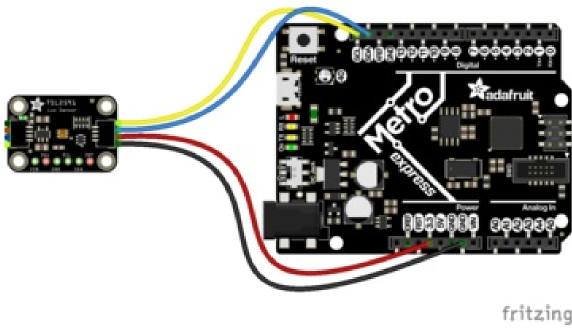


- Connect **USB** on the Feather to **VIN** on the TSL2591.
- Connect **GND** on the Feather to **GND** on the TSL2591.
- Connect **SCL** on the Feather to **SCL** on the TSL2591.
- Connect **SDA** on the Feather to **SDA** on the TSL2591.

ItsyBitsy M0 Express and ItsyBitsy M4 Express



- Connect **USB** on the ItsyBitsy to **VIN** on the TSL2591
- Connect **G** on the ItsyBitsy to **GND** on the TSL2591.
- Connect **SCL** on the ItsyBitsy to **SCL** on the TSL2591.
- Connect **SDA** on the ItsyBitsy to **SDA** on the TSL2591.



Metro M0 Express and Metro M4 Express

- Connect **5V** on the Metro to **VIN** on the TSL2591.
- Connect **GND** on the Metro to **GND** on the TSL2591.
- Connect **SCL** on the Metro to **SCL** on the TSL2591.
- Connect **SDA** on the Metro to **SDA** on the TSL2591.

Find Your Sensor

The first thing you'll want to do after getting the sensor wired up, is make sure it's wired correctly. We're going to do an I2C scan to see if the board is detected, and if it is, print out its I2C address.

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

```
"""CircuitPython Essentials I2C Scan example"""
# If you run this and it seems to hang, try manually unlocking
# your I2C bus from the REPL with
#  >>> import board
#  >>> board.I2C().unlock()

import time
import board

i2c = board.I2C()

while not i2c.try_lock():
    pass

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

finally:  # unlock the i2c bus when ctrl-c'ing out of the loop
    i2c.unlock()
```

First we create the `i2c` object, using `board.I2C()`. This convenience routine creates and saves a `busio.I2C` object using the default pins `board.SCL` and `board.SDA`. If the object has already been created, then the existing object is returned. No matter how many times you call `board.I2C()`, it will return the same object. This is called a **singleton**.

To be able to scan it, we need to lock the I2C down so the only thing accessing it is the code. So next we

include a loop that waits until I2C is locked and then continues on to the scan function.

Last, we have the loop that runs the actual scan, `i2c_scan()`. Because I2C typically refers to addresses in hex form, we've included this bit of code that formats the results into hex format: `[hex(device_address) for device_address in i2c.scan()]`.

Open the serial console to see the results! The code prints out an array of addresses. We've connected the TSL2591 which has a 7-bit I2C address of 0x29. The result for this sensor is `I2C addresses found: ['0x29']`. If no addresses are returned, refer back to the wiring diagrams to make sure you've wired up your sensor correctly.

I2C Sensor Data

Now we know for certain that our sensor is connected and ready to go. Let's find out how to get the data from our sensor!

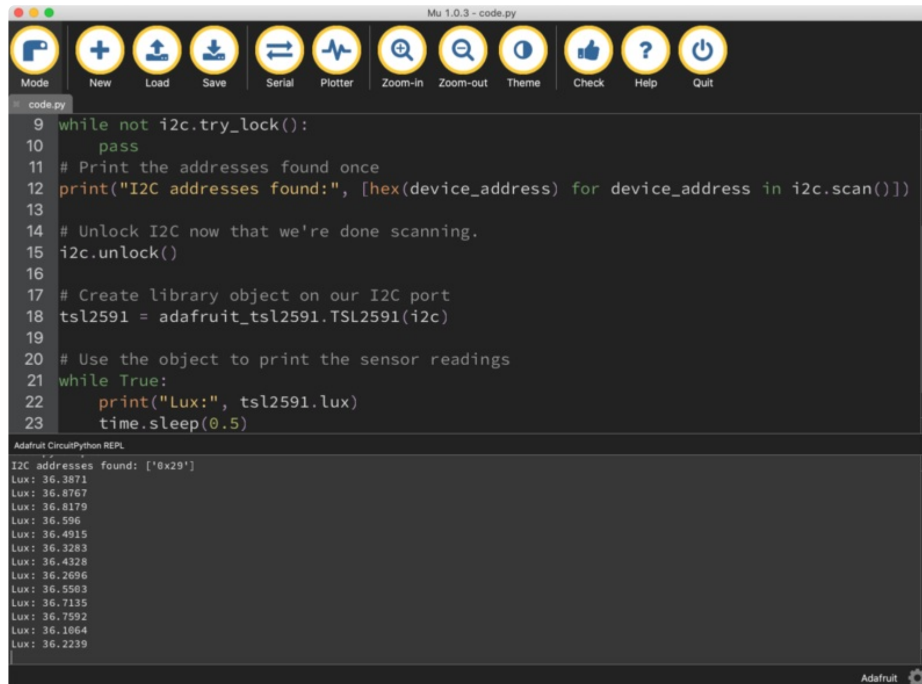
Copy and paste the code into **code.py** using your favorite editor, and save the file.

```
"""CircuitPython Essentials I2C sensor example using TSL2591"""
import time
import board
import adafruit_tsl2591

i2c = board.I2C()

# Lock the I2C device before we try to scan
while not i2c.try_lock():
    pass
# Print the addresses found once
print("I2C addresses found:", [hex(device_address) for device_address in i2c.scan()])

# Unlock I2C now that we're done scanning.
i2c.unlock()

# Create library object on our I2C port
tsl2591 = adafruit_tsl2591.TSL2591(i2c)

# Use the object to print the sensor readings
while True:
    print("Lux:", tsl2591.lux)
    time.sleep(0.5)
```

This code begins the same way as the scan code. We've included the scan code so you have verification that your sensor is wired up correctly and is detected. It prints the address once. After the scan, we unlock I2C with `i2c_unlock()` so we can use the sensor for data.

We create our sensor object using the sensor library. We call it `tsl2591` and provide it the `i2c` object.

Then we have a simple loop that prints out the lux reading using the sensor object we created. We add a `time.sleep(1.0)`, so it only prints once per second. Connect to the serial console to see the results. Try shining a light on it to see the results change!



Where's my I2C?

On the SAMD21, SAMD51 and nRF52840, we have the flexibility of using a wide range of pins for I2C. On the nRF52840, any pin can be used for I2C! Some chips, like the ESP8266, require using bitbangio, but can also use any pins for I2C. There's some other chips that may have fixed I2C pin.

The good news is you can use many but not *all* pins. Given the large number of SAMD boards we have, its impossible to guarantee anything other than the labeled 'SDA' and 'SCL'. So, if you want some other setup, or multiple I2C interfaces, how will you find those pins? Easy! We've written a handy script.

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://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:
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.
```

```python
"""CircuitPython Essentials I2C possible pin-pair identifying script"""
import board
import busio
from microcontroller import Pin


def is_hardware_I2C(scl, sda):
    try:
        p = busio.I2C(scl, sda)
        p.deinit()
        return True
    except ValueError:
        return False
    except RuntimeError:
        return True


def get_unique_pins():
    exclude = ['NEOPIXEL', 'APA102_MOSI', 'APA102_SCK']
    pins = [pin for pin in [
        getattr(board, p) for p in dir(board) if p not in exclude]
            if isinstance(pin, Pin)]
    unique = []
    for p in pins:
        if p not in unique:
            unique.append(p)
    return unique


for scl_pin in get_unique_pins():
    for sda_pin in get_unique_pins():
        if scl_pin is sda_pin:
            continue
        if is_hardware_I2C(scl_pin, sda_pin):
            print("SCL pin:", scl_pin, "\t SDA pin:", sda_pin)
```

CircuitPython HID Keyboard

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 CircuitPlayground 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 HID Keyboard and Mouse

These examples have been updated for version 4+ of the CircuitPython HID library. On some boards, such as the CircuitPlayground Express, this library is built into CircuitPython. So, please use the latest version of CircuitPython with these examples. (At least 5.3.1)

One of the things we baked into CircuitPython is 'HID' (**H**uman **I**nterface **D**evice) control - that means keyboard and mouse capabilities. This means your CircuitPython board can act like a keyboard device and press key commands, or a mouse and have it move the mouse pointer around and press buttons. This is really handy because even if you cannot adapt your software to work with hardware, there's almost always a keyboard interface - so if you want to have a capacitive touch interface for a game, say, then keyboard emulation can often get you going really fast!

This section walks you through the code to create a keyboard or mouse emulator. First we'll go through an example that uses pins on your board to emulate keyboard input. Then, we will show you how to wire up a joystick to act as a mouse, and cover the code needed to make that happen.

You'll need the **adafruit_hid** 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).

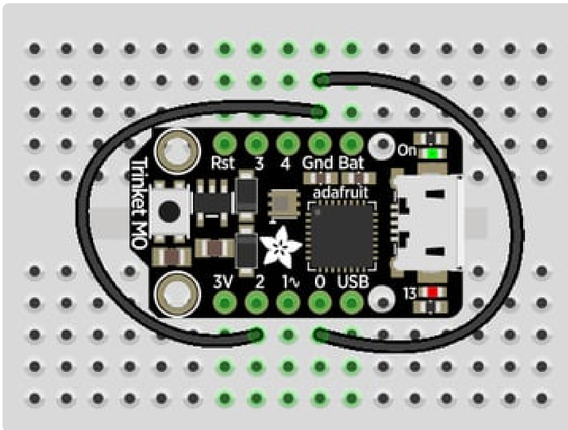
CircuitPython Keyboard Emulator

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

```
"""CircuitPython Essentials HID Keyboard example"""
import time

import board
import digitalio
import usb_hid
from adafruit_hid.keyboard import Keyboard
from adafruit_hid.keyboard_layout_us import KeyboardLayoutUS
from adafruit_hid.keycode import Keycode

# A simple neat keyboard demo in CircuitPython

# The pins we'll use, each will have an internal pullup
keypress_pins = [board.A1, board.A2]
# Our array of key objects
key_pin_array = []
# The Keycode sent for each button, will be paired with a control key
keys_pressed = [Keycode.A, "Hello World!\n"]
control_key = Keycode.SHIFT

# The keyboard object!
time.sleep(1)  # Sleep for a bit to avoid a race condition on some systems
```

```
keyboard = Keyboard(usb_hid.devices)
keyboard_layout = KeyboardLayoutUS(keyboard)  # We're in the US :)

# Make all pin objects inputs with pullups
for pin in keypress_pins:
    key_pin = digitalio.DigitalInOut(pin)
    key_pin.direction = digitalio.Direction.INPUT
    key_pin.pull = digitalio.Pull.UP
    key_pin_array.append(key_pin)

# For most CircuitPython boards:
led = digitalio.DigitalInOut(board.LED)
# For QT Py M0:
# led = digitalio.DigitalInOut(board.SCK)
led.direction = digitalio.Direction.OUTPUT

print("Waiting for key pin...")

while True:
    # Check each pin
    for key_pin in key_pin_array:
        if not key_pin.value:  # Is it grounded?
            i = key_pin_array.index(key_pin)
            print("Pin #%d is grounded." % i)

            # Turn on the red LED
            led.value = True

            while not key_pin.value:
                pass  # Wait for it to be ungrounded!
            # "Type" the Keycode or string
            key = keys_pressed[i]  # Get the corresponding Keycode or string
            if isinstance(key, str):  # If it's a string...
                keyboard_layout.write(key)  # ...Print the string
            else:  # If it's not a string...
                keyboard.press(control_key, key)  # "Press"...
                keyboard.release_all()  # ..."Release"!

            # Turn off the red LED
            led.value = False

    time.sleep(0.01)
```

Connect pin **A1** *or* **A2** to ground, using a wire or alligator clip, then disconnect it to send the key press "A" or the string "Hello world!"



This wiring example shows A1 and A2 connected to ground.

Remember, on Trinket, A1 and A2 are labeled 2 and 0! On other boards, you will have A1 and A2 labeled as expected.

Create the Objects and Variables

First, we assign some variables for later use. We create three arrays assigned to variables: `keypress_pins`, `key_pin_array`, and `keys_pressed`. The first is the pins we're going to use. The second is empty because we're going to fill it later. The third is what we would like our "keyboard" to output - in this case the letter "A" and the phrase, "Hello world!". We create our last variable assigned to `control_key` which allows us to later apply the shift key to our keypress. We'll be using two keypresses, but you can have up to six keypresses at once.

Next `keyboard` and `keyboard_layout` objects are created. We only have US right now (if you make other layouts please submit a GitHub pull request!). The `time.sleep(1)` avoids an error that can happen if the program gets run as soon as the board gets plugged in, before the host computer finishes connecting to the board.

Then we take the pins we chose above, and create the pin objects, set the direction and give them each a pullup. Then we apply the pin objects to `key_pin_array` so we can use them later.

Next we set up the little red LED to so we can use it as a status light.

The last thing we do before we start our loop is `print`, "Waiting for key pin..." so you know the code is ready and waiting!

The Main Loop

Inside the loop, we check each pin to see if the state has changed, i.e. you connected the pin to ground. Once it changes, it prints, "Pin # grounded." to let you know the ground state has been detected. Then we turn on the red LED. The code waits for the state to change again, i.e. it waits for you to unground the pin by disconnecting the wire attached to the pin from ground.

Then the code gets the corresponding keys pressed from our array. If you grounded and ungrounded A1, the code retrieves the keypress `a`, if you grounded and ungrounded A2, the code retrieves the string, `"Hello world!"`

If the code finds that it's retrieved a string, it prints the string, using the `keyboard_layout` to determine the keypresses. Otherwise, the code prints the keypress from the `control_key` and the keypress "a", which result in "A". Then it calls `keyboard.release_all()`. You always want to call this soon after a keypress or you'll end up with a stuck key which is really annoying!

Instead of using a wire to ground the pins, you can try wiring up buttons like we did in CircuitPython Digital In & Out (https://adafru.it/Beo). Try altering the code to add more pins for more keypress options!

CircuitPython Mouse Emulator

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

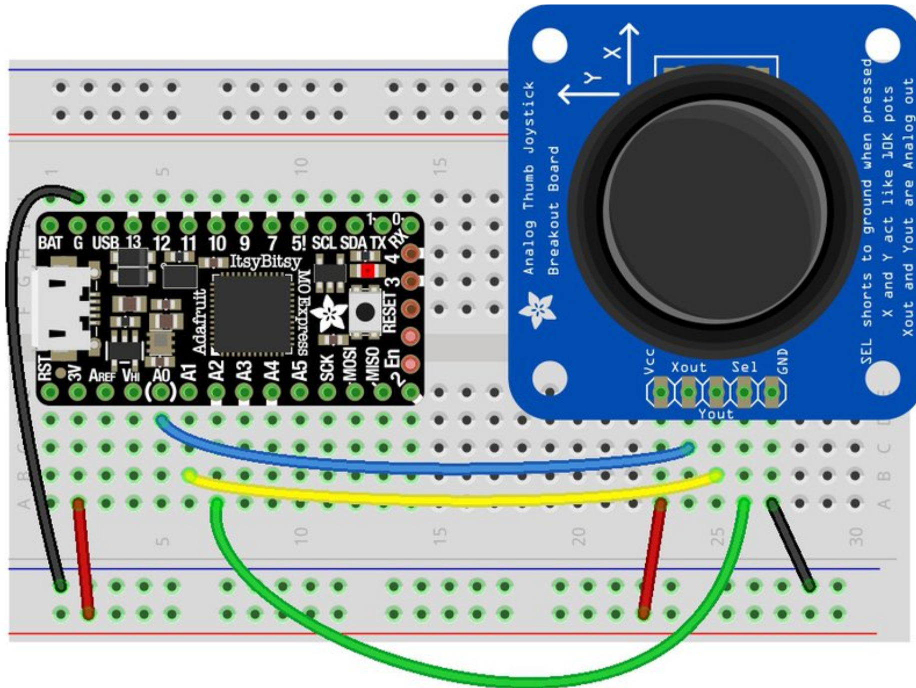
```
"""CircuitPython Essentials HID Mouse example"""
import time
import analogio
import board
import digitalio
import usb_hid
from adafruit_hid.mouse import Mouse

mouse = Mouse(usb_hid.devices)

x_axis = analogio.AnalogIn(board.A0)
y_axis = analogio.AnalogIn(board.A1)
select = digitalio.DigitalInOut(board.A2)
select.direction = digitalio.Direction.INPUT
select.pull = digitalio.Pull.UP

pot_min = 0.00
pot_max = 3.29
step = (pot_max - pot_min) / 20.0


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


def steps(axis):
    """ Maps the potentiometer voltage range to 0-20 """
    return round((axis - pot_min) / step)
```

```
while True:
    x = get_voltage(x_axis)
    y = get_voltage(y_axis)

    if select.value is False:
        mouse.click(Mouse.LEFT_BUTTON)
        time.sleep(0.2)  # Debounce delay

    if steps(x) > 11.0:
        # print(steps(x))
        mouse.move(x=1)
    if steps(x) < 9.0:
        # print(steps(x))
        mouse.move(x=-1)

    if steps(x) > 19.0:
        # print(steps(x))
        mouse.move(x=8)
    if steps(x) < 1.0:
        # print(steps(x))
        mouse.move(x=-8)

    if steps(y) > 11.0:
        # print(steps(y))
        mouse.move(y=-1)
    if steps(y) < 9.0:
        # print(steps(y))
        mouse.move(y=1)

    if steps(y) > 19.0:
        # print(steps(y))
        mouse.move(y=-8)
    if steps(y) < 1.0:
        # print(steps(y))
        mouse.move(y=8)
```

For this example, we've wired up a 2-axis thumb joystick with a select button. We use this to emulate the mouse movement and the mouse left-button click. To wire up this joytick:

- Connect **VCC** on the joystick to the **3V** on your board. Connect **ground** to **ground**.
- Connect **Xout** on the joystick to pin **A0** on your board.
- Connect **Yout** on the joystick to pin **A1** on your board.
- Connect **Sel** on the joystick to pin **A2** on your board.

Remember, Trinket's pins are labeled differently. Check the [Trinket Pinouts page](https://adafru.it/AMd) (https://adafru.it/AMd) to verify your wiring.



To use this demo, simply move the joystick around. The mouse will move slowly if you move the joystick a little off center, and more quickly if you move it as far as it goes. Press down on the joystick to click the mouse. Awesome! Now let's take a look at the code.

Create the Objects and Variables

First we create the mouse object.

Next, we set `x_axis` and `y_axis` to pins `A0` and `A1`. Then we set `select` to `A2`, set it as input and give it a pullup.

The x and y axis on the joystick act like 2 potentiometers. We'll be using them just like we did in CircuitPython Analog In (https://adafru.it/Bep). We set `pot_min` and `pot_max` to be the minimum and maximum voltage read from the potentiometers. We assign `step = (pot_max - pot_min) / 20.0` to use in a helper function.

CircuitPython HID Mouse Helpers

First we have the `get_voltage()` helper so we can get the correct readings from the potentiometers. Look familiar? We learned about it in Analog In (https://adafru.it/Bep).

Second, we have `steps(axis)`. To use it, you provide it with the axis you're reading. This is where we're going to use the `step` variable we assigned earlier. The potentiometer range is 0-3.29. This is a small range. It's even smaller with the joystick because the joystick sits at the center of this range, 1.66, and the

+ and - of each axis is above and below this number. Since we need to have thresholds in our code, we're going to map that range of 0-3.29 to while numbers between 0-20.0 using this helper function. That way we can simplify our code and use larger ranges for our thresholds instead of trying to figure out tiny decimal number changes.

Main Loop

First we assign `x` and `y` to read the voltages from `x_axis` and `y_axis`.

Next, we check to see when the state of the select button is `False`. It defaults to `True` when it is not pressed, so if the state is `False`, the button has been pressed. When it's pressed, it sends the command to click the left mouse button. The `time.sleep(0.2)` prevents it from reading multiple clicks when you've only clicked once.

Then we use the `steps()` function to set our mouse movement. There are two sets of two `if` statements for each axis. Remember that `10` is the center step, as we've mapped the range `0-20`. The first set for each axis says if the joystick moves 1 step off center (left or right for the x axis and up or down for the y axis), to move the mouse the appropriate direction by 1 unit. The second set for each axis says if the joystick is moved to the lowest or highest step for each axis, to move the mouse the appropriate direction by 8 units. That way you have the option to move the mouse slowly or quickly!

To see what `step` the joystick is at when you're moving it, uncomment the `print` statements by removing the `#` from the lines that look like `# print(steps(x))`, and connecting to the serial console to see the output. Consider only uncommenting one set at a time, or you end up with a huge amount of information scrolling very quickly, which can be difficult to read!

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

CircuitPython CPU Temp

There is a CPU temperature sensor built into every ATSAMD21, ATSAMD51 and nRF52840 chips. CircuitPython makes it really simple to read the data from this sensor. This works on the Adafruit CircuitPython boards it's built into the microcontroller used for these boards.

The data is read using two simple commands. We're going to enter them in the REPL. Plug in your board, connect to the serial console (https://adafru.it/Bec), and enter the REPL (https://adafru.it/Awz). Then, enter the following commands into the REPL:

```
import microcontroller
microcontroller.cpu.temperature
```

That's it! You've printed the temperature in Celsius to the REPL. Note that it's not exactly the ambient temperature and it's not super precise. But it's close!

```
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-compatible microcontrollers show up as a **CIRCUITPY** drive when plugged into your computer, allowing you to edit code directly on the board. Perhaps you've wondered whether or not you can write data *from CircuitPython* directly to the board to act as a data logger. The answer is **yes**!

The `storage` module in CircuitPython enables you to write code that allows CircuitPython to write data to the **CIRCUITPY** drive. This process requires you to include a **boot.py** file on your **CIRCUITPY** drive, along side your **code.py** file.

The **boot.py** file is special - the code within it is executed when CircuitPython starts up, either from a hard reset or powering up the board. It is not run on soft reset, for example, if you reload the board from the serial console or the REPL. This is in contrast to the code within **code.py**, which is executed after CircuitPython is already running.

The CIRCUITPY drive is typically writable by your computer; this is what allows you to edit your code directly on the board. The reason you need a **boot.py** file is that you have to set the filesystem to be read-only by your computer to allow it to be writable by CircuitPython. This is because CircuitPython cannot write to the filesystem at the same time as your computer. Doing so can lead to filesystem corruption and loss of all content on the drive, so CircuitPython is designed to only allow one at at time.

You can only have either your computer edit the CIRCUITPY drive files, or CircuitPython. You cannot have both write to the drive at the same time. (Bad Things Will Happen so we do not allow you to do it!)

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

Click the **Download Project Bundle** button, open the resulting zip file, and copy the **boot.py** file to your **CIRCUITPY** drive.

The filesystem will NOT automatically be set to read-only on creation of this file! You'll still be able to edit files on CIRCUITPY after saving this boot.py.

```
"""CircuitPython Essentials Storage logging boot.py file"""
import board
import digitalio
import storage

# For Gemma M0, Trinket M0, Metro M0/M4 Express, ItsyBitsy M0/M4 Express
switch = digitalio.DigitalInOut(board.D2)

# For Feather M0/M4 Express
# switch = digitalio.DigitalInOut(board.D5)

# For Circuit Playground Express, Circuit Playground Bluefruit
# switch = digitalio.DigitalInOut(board.D7)

switch.direction = digitalio.Direction.INPUT
switch.pull = digitalio.Pull.UP

# If the switch pin is connected to ground CircuitPython can write to the drive
storage.remount("/", switch.value)
```

The `storage.remount()` command has a `readonly` keyword argument. **This argument refers to the read/write state of CircuitPython.** It does NOT refer to the read/write state of your computer.

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

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

For **Feather M0 Express and Feather M4 Express**, comment out `switch = digitalio.DigitalInOut(board.D2)`, and uncomment `switch = digitalio.DigitalInOut(board.D5)`.

For **Circuit Playground Express and Circuit Playground Bluefruit**, comment out `switch = digitalio.DigitalInOut(board.D2)`, and uncomment `switch = digitalio.DigitalInOut(board.D7)`. Remember, D7 is the onboard slide switch, so there's no extra wires or alligator clips needed.

On the Circuit Playground Express or Circuit Playground Bluefruit, the switch is in the right position (closer to the ear icon on the silkscreen) it returns `False`, and the CIRCUITPY drive will be writable by CircuitPython. If the switch is in the left position (closer to the music icon on the silkscreen), it returns `True`, and the CIRCUITPY drive will be writable by your computer.

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

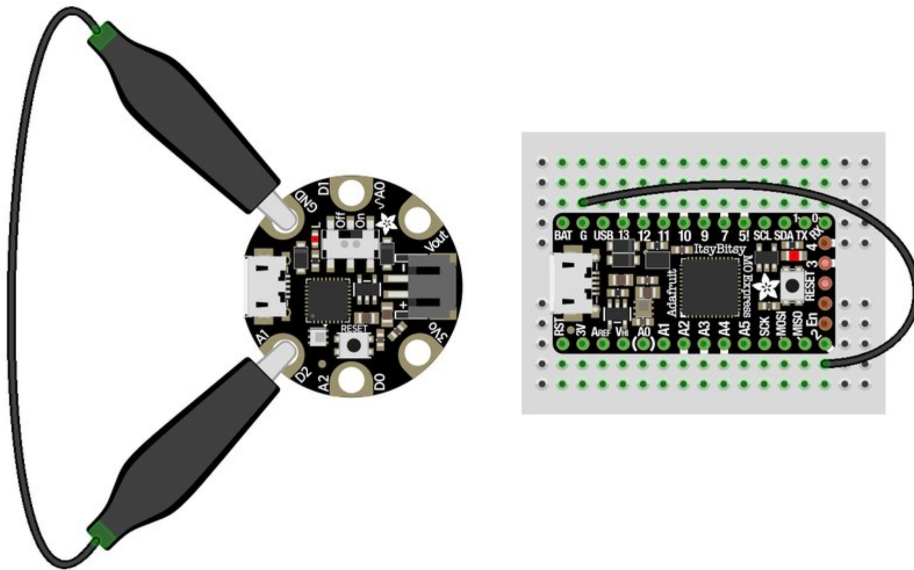
The following is your new **code.py**. Copy and paste the code into **code.py** using your favorite editor.

```
"""CircuitPython Essentials Storage logging example"""
import time
import board
import digitalio
import microcontroller

# For most CircuitPython boards:
led = digitalio.DigitalInOut(board.LED)
# For QT Py M0:
# led = digitalio.DigitalInOut(board.SCK)
led.switch_to_output()

try:
    with open("/temperature.txt", "a") as fp:
        while True:
            temp = microcontroller.cpu.temperature
            # do the C-to-F conversion here if you would like
            fp.write('{0:f}\n'.format(temp))
            fp.flush()
            led.value = not led.value
            time.sleep(1)
except OSError as e:  # Typically when the filesystem isn't writeable...
    delay = 0.5  # ...blink the LED every half second.
    if e.args[0] == 28:  # If the file system is full...
        delay = 0.25  # ...blink the LED faster!
    while True:
        led.value = not led.value
        time.sleep(delay)
```

Logging the Temperature

The way **boot.py** works is by checking to see if the pin you specified in the switch setup in your code is connected to a ground pin. If it is, it changes the read-write state of the file system, so the CircuitPython core can begin logging the temperature to the board.

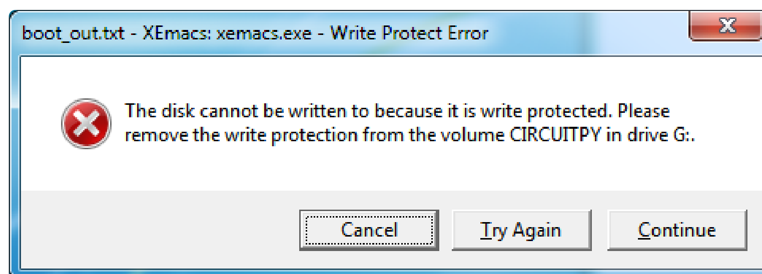
For help finding the correct pins, see the wiring diagrams and information in the [Find the Pins section of the CircuitPython Digital In & Out guide](https://adafru.it/Bes) (https://adafru.it/Bes). Instead of wiring up a switch, however, you'll be connecting the pin directly to ground with alligator clips or jumper wires.



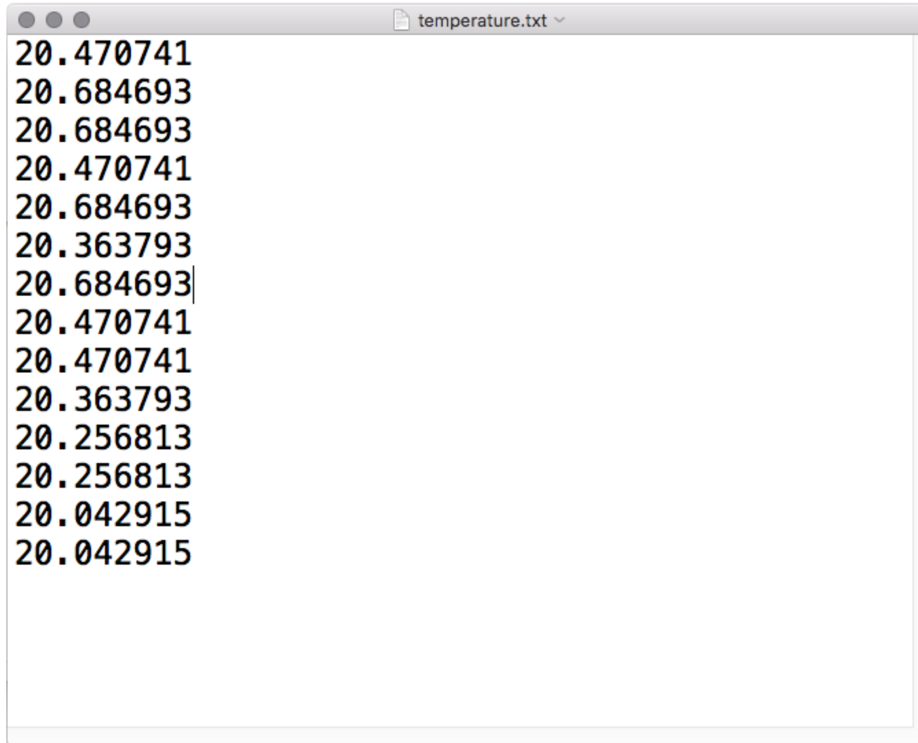
boot.py only runs on first boot of the device, not if you re-load the serial console with ctrl+D or if you save a file. You must EJECT the USB drive, then physically press the reset button!

Once you copied the files to your board, eject it and unplug it from your computer. If you're using your Circuit Playground Express, all you have to do is make sure the switch is to the right. Otherwise, use alligator clips or jumper wires to connect the chosen pin to ground. Then, plug your board back into your computer.

You will not be able to edit code on your **CIRCUITPY** drive anymore!



The red LED should blink once a second and you will see a new **temperature.txt** file on **CIRCUITPY**.



This file gets updated once per second, but you won't see data come in live. Instead, when you're ready to grab the data, eject and unplug your board. For CPX, move the switch to the left, otherwise remove the wire connecting the pin to ground. Now it will be possible for you to write to the filesystem from your computer again, but it will not be logging data.

We have a more detailed guide on this project available here: CPU Temperature Logging with CircuitPython. (https://adafru.it/zuF) If you'd like more details, check it out!

CircuitPython Expectations

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

Always Run the Latest Version of CircuitPython and Libraries

As we continue to develop CircuitPython and create new releases, we will stop supporting older releases. **You need to update to the latest CircuitPython (https://adafru.it/Em8).**

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

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

I have to continue using CircuitPython 3.x or 2.x, where can I find compatible libraries?

We are no longer building or supporting the CircuitPython 2.x and 3.x library bundles. We highly encourage you to update CircuitPython to the latest version (https://adafru.it/Em8) and use the current version of the libraries (https://adafru.it/ENC). However, if for some reason you cannot update, you can find the last available 2.x build here (https://adafru.it/FJA) and the last available 3.x build here (https://adafru.it/FJB).

Switching Between CircuitPython and Arduino

Many of the CircuitPython boards also run Arduino. But how do you switch between the two? Switching between CircuitPython and Arduino is easy.

If you're currently running Arduino and would like to start using CircuitPython, follow the steps found in Welcome to CircuitPython: Installing CircuitPython (https://adafru.it/Amd).

If you're currently running CircuitPython and would like to start using Arduino, plug in your board, and then load your Arduino sketch. If there are any issues, you can double tap the reset button to get into the bootloader and then try loading your sketch. Always backup any files you're using with CircuitPython that you want to save as they could be deleted.

That's it! It's super simple to switch between the two.

The Difference Between Express And Non-Express Boards

We often reference "Express" and "Non-Express" boards when discussing CircuitPython. What does this mean?

Express refers to the inclusion of an extra 2MB flash chip on the board that provides you with extra space for CircuitPython and your code. This means that we're able to include more functionality in CircuitPython and you're able to do more with your code on an Express board than you would on a non-Express board.

Express boards include Circuit Playground Express, ItsyBitsy M0 Express, Feather M0 Express, Metro M0 Express and Metro M4 Express.

Non-Express boards include Trinket M0, Gemma M0, QT Py, Feather M0 Basic, and other non-Express Feather M0 variants.

Non-Express Boards: Gemma, Trinket, and QT Py

CircuitPython runs nicely on the Gemma M0, Trinket M0, or QT Py M0 but there are some constraints

Small Disk Space

Since we use the internal flash for disk, and that's shared with runtime code, its limited! Only about 50KB of space.

No Audio or NVM

Part of giving up that FLASH for disk means we couldn't fit everything in. There is, at this time, no support for hardware audio playpack or NVM 'eeprom'. Modules `audioio` and `bitbangio` are not included. For that support, check out the Circuit Playground Express or other Express boards.

However, I2C, UART, capacitive touch, NeoPixel, DotStar, PWM, analog in and out, digital IO, logging storage, and HID do work! Check the CircuitPython Essentials for examples of all of these.

Differences Between CircuitPython and MicroPython

For the differences between CircuitPython and MicroPython, check out the [CircuitPython documentation](https://adafru.it/Bvz) (https://adafru.it/Bvz).

Differences Between CircuitPython and Python

Python (also known as CPython) is the language that MicroPython and CircuitPython are based on. There are many similarities, but there are also many differences. This is a list of a few of the differences.

Python Libraries

Python is advertised as having "batteries included", meaning that many standard libraries are included. Unfortunately, for space reasons, many Python libraries are not available. So for instance while we wish you could `import numpy`, `numpy` isn't available (look for the `ulab` library for similar functions to `numpy` which works on many microcontroller boards). So you may have to port some code over yourself!

Integers in CircuitPython

On the non-Express boards, integers can only be up to 31 bits long. Integers of unlimited size are not supported. The largest positive integer that can be represented is 2^{30}-1, 1073741823, and the most negative integer possible is -2^{30}, -1073741824.

As of CircuitPython 3.0, Express boards have arbitrarily long integers as in Python.

Floating Point Numbers and Digits of Precision for Floats in CircuitPython

Floating point numbers are single precision in CircuitPython (not double precision as in Python). The largest floating point magnitude that can be represented is about +/-3.4e38. The smallest magnitude that can be represented with full accuracy is about +/-1.7e-38, though numbers as small as +/-5.6e-45 can be represented with reduced accuracy.

CircuitPython's floats have 8 bits of exponent and 22 bits of mantissa (not 24 like regular single precision floating point), which is about five or six decimal digits of precision.

Differences between MicroPython and Python

For a more detailed list of the differences between CircuitPython and Python, you can look at the MicroPython documentation. [We keep up with MicroPython stable releases, so check out the core 'differences' they document here.](https://adafru.it/zwA) (https://adafru.it/zwA)

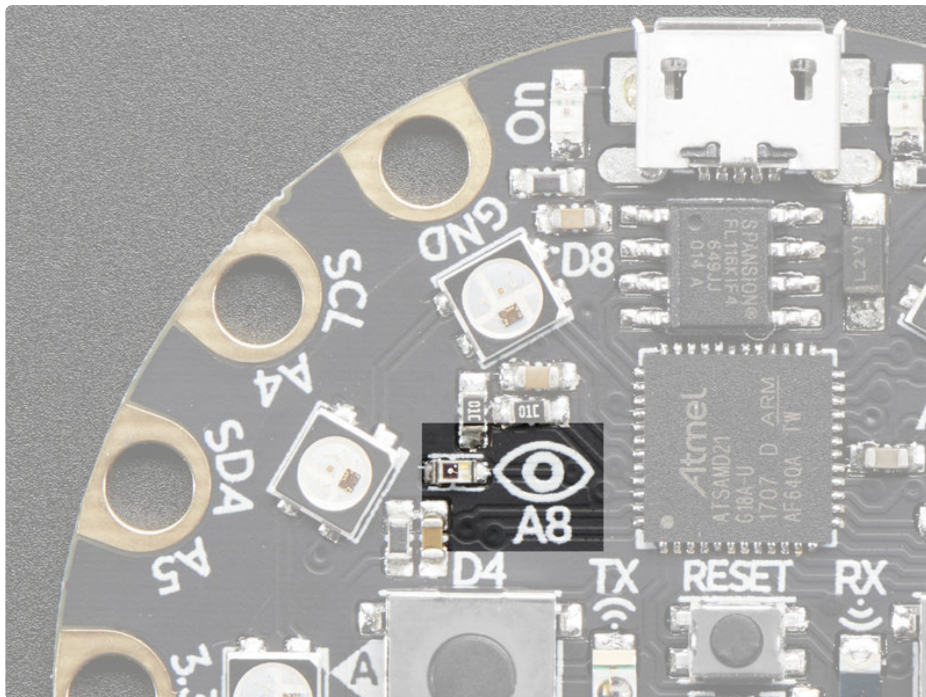
Playground Light Sensor

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!

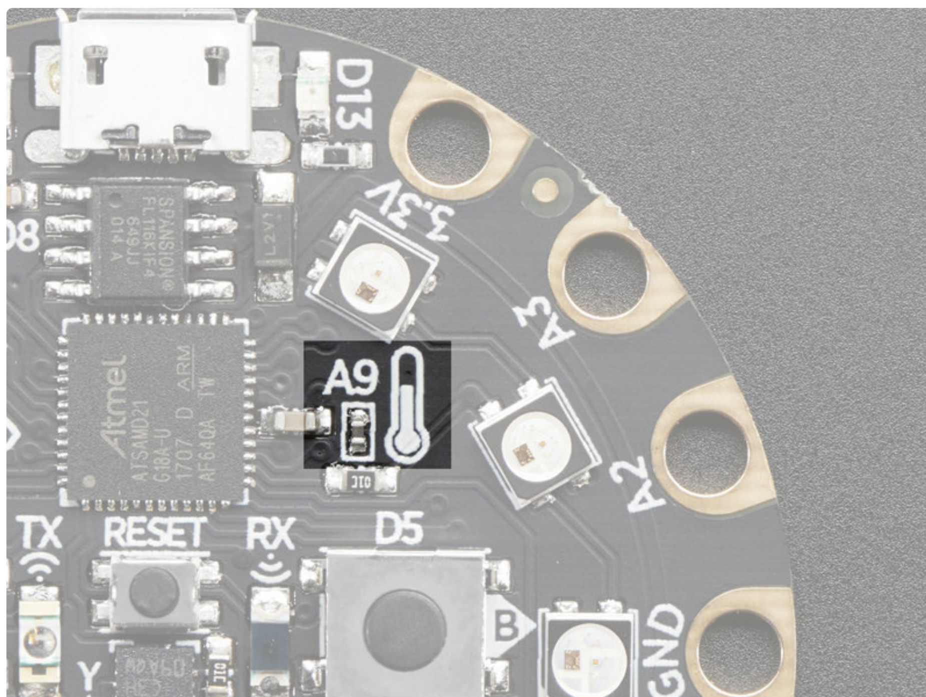
Playground Temperature

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 build in thermistor. This little sensor is a thermally sensitive resistor, meaning it's 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.

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.

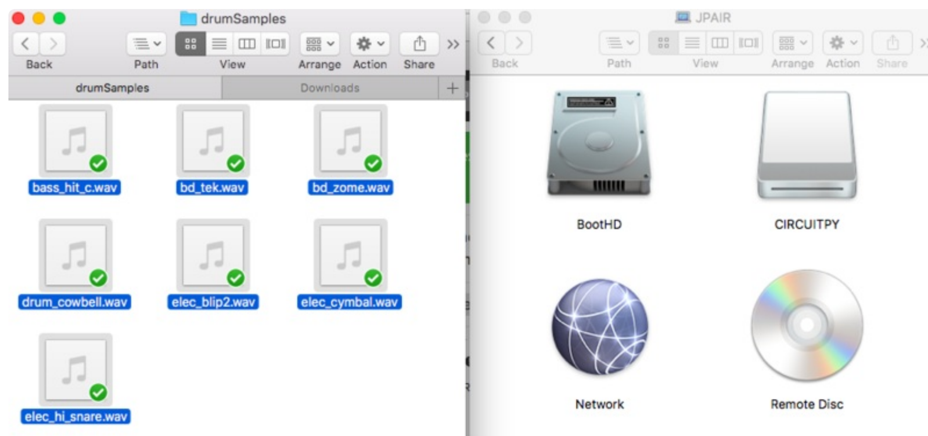
We can use the capacitive touch pads on the Circuit Playground Express as triggers, and small .wav files for our drum sounds!

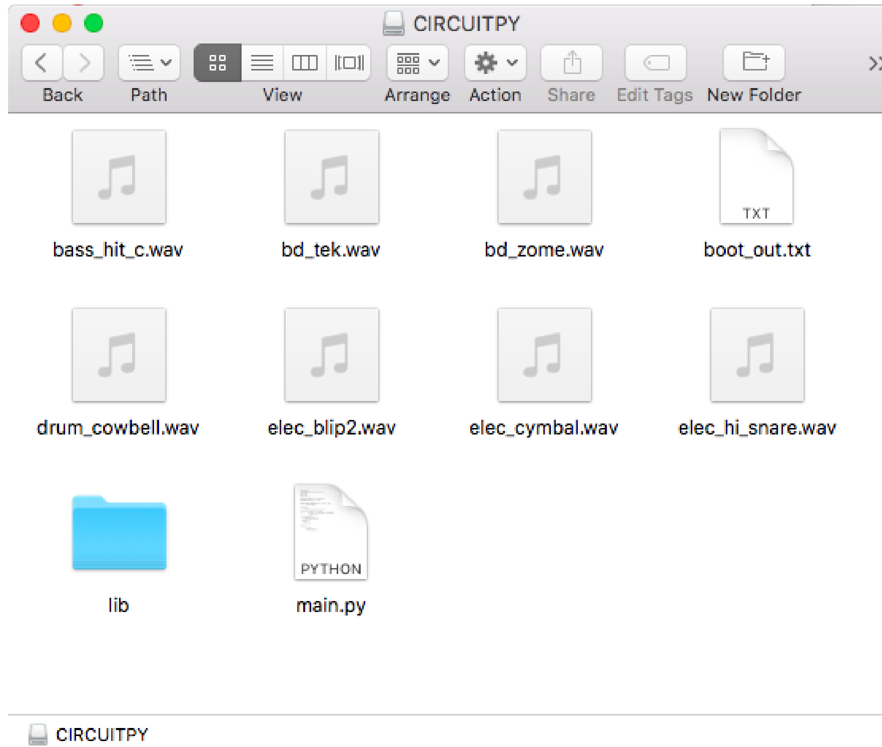
First, download the .zip file below, which contains all of the drum samples we'll be using. Save the file to your desktop or somewhere else easy to find, and then unzip it.

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.



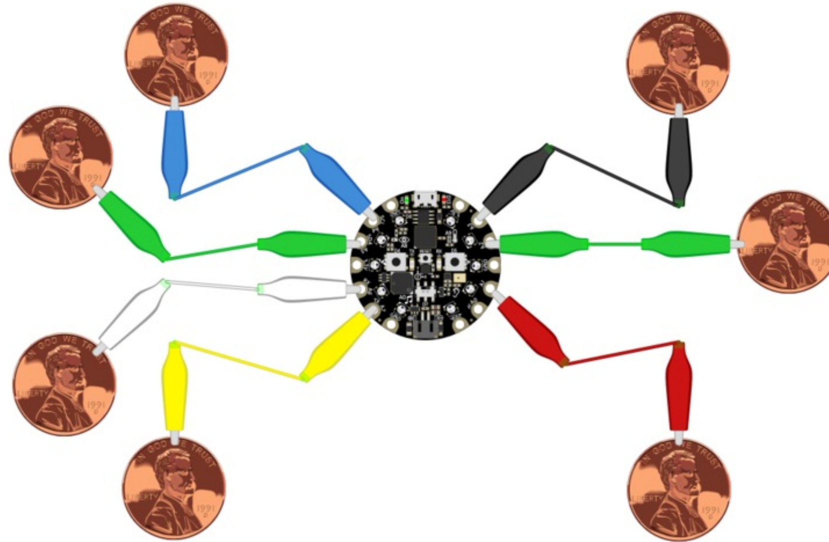


Now, it's time to code the Circuit Playground Express! Copy the code shown below, and then paste it into your code editor of choice. Save the file as **code.py** on your CIRCUITPY drive.

```
# 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 crammed, 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!

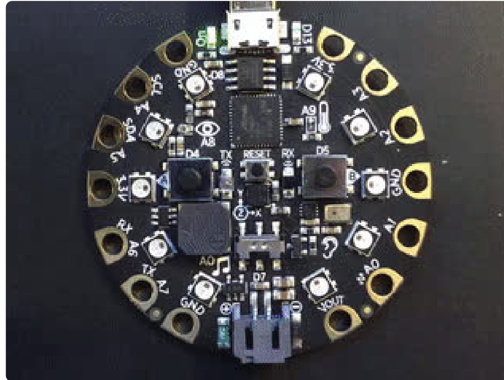
If you want to use your own sound files, you can! Record, sample, remix, or simply download files from a sound file sight, such as freesample.org. Then, to make sure you have the files converted to the proper specifications, check out this guide here (https://adafru.it/s8f) that'll show you how! Spoiler 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 1/8" (https://adafru.it/Bf3) or 1/4" phono output (https://adafru.it/Bf4) 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 the computes the energy in the sample (corresponding to volume) using a Root-Mean-Square (https://adafru.it/Bf5) computation (RMS). 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
```

```
# 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)
```

```
        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:
```

```
    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()
```

Arduino

> Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino boards (https://adafru.it/oVa) are able to read inputs - light on a sensor, a finger on a button, or a Twitter message - and turn it into an output - activating a motor, turning on an LED, publishing something online. You can tell your board what to do by sending a set of instructions to the microcontroller on the board. To do so you use the Arduino programming language (https://adafru.it/oVb) (based on Wiring (https://adafru.it/oVc)), and the Arduino Software (IDE) (https://adafru.it/fvm), based on Processing (https://adafru.it/ddm).
>
> -- https://www.arduino.cc/en/Guide/Introduction (https://adafru.it/Bf6)

Arduino is the third and oldest of the progamming languages supported by Circuit Playground Express. Arduino has over a decade of projects and history, so you'll find a lot of existing code that you can use with your CPX.

Arduino is essentially C/C++ with a built in library of hardware interfaces. It is the most low-level of the three languages - you can embed assembly, write ultra-fast code, and you get the full use of the entire memory and filesystem. But... its harder for beginners to use! If you're an Arduino expert you may want to just use CPX with the Arduino IDE.

If you're starting out on your coding journey, check out MakeCode first.

If you want fast development without uploading/compiling or memory management, check out CircuitPython. It's not as clock-cycle-fast as Arduino but is faster to code in, simpler and more fun.

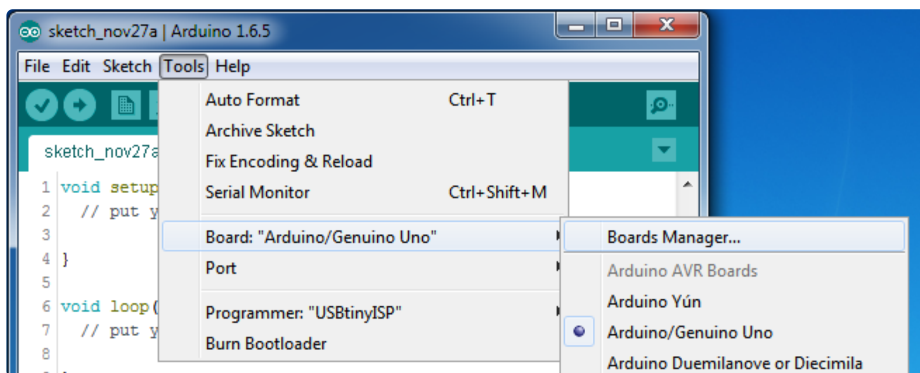
Set Up Arduino IDE

Since the Circuit Playground Express uses an ATSAMD21 chip running at 48 MHz, you can pretty easily get it working with the Arduino IDE. Most libraries (including the popular ones like NeoPixels and display) will work with the M0, especially devices & sensors that use i2c or SPI.

The Circuit Playground Express is 'natively' supported in the Arduino IDE so its **really easy** to set up!

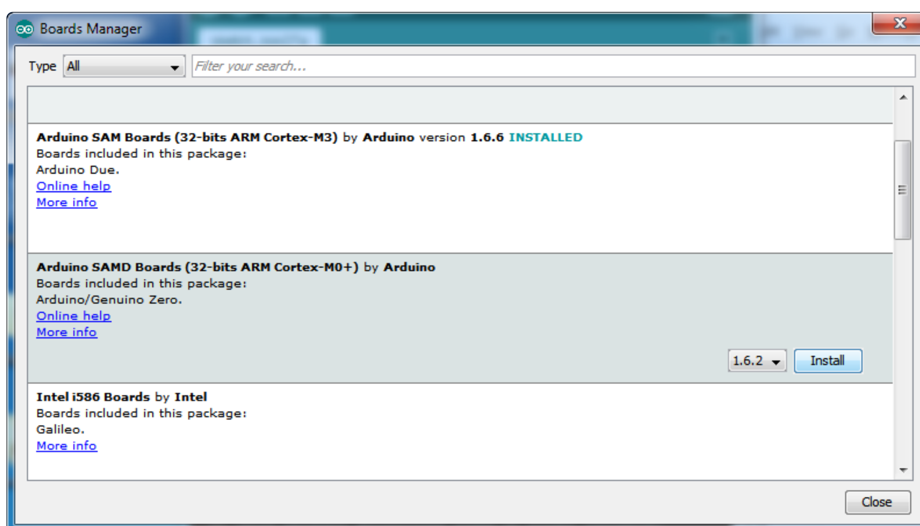
Install SAMD Support

Open the **Boards Manager** by navigating to the **Tools->Board** menu



First up, install the **Arduino SAMD Boards** version **1.6.16** or later

You can type **Arduino SAMD** in the top search bar, then when you see the entry, click **Install**

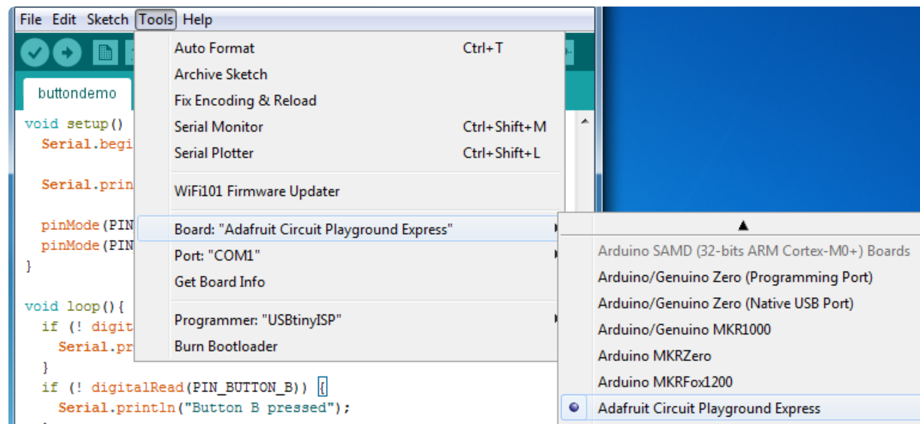


Even though in theory you don't need to - I recommend rebooting the IDE

Quit and reopen the Arduino IDE to ensure that all of the boards are properly installed. You should now be

able to select and upload to the new boards listed in the **Tools->Board** menu.

Select the Circuit Playground Express board:



There's also a definition for the Adafruit Circuit Playground Express in the 'Adafruit' board support package. If you'd like to try that definition for some reason

1. First follow this page to activate the Adafruit URL (https://adafru.it/npA)
2. Then follow this page to install Adafruit SAMD package (https://adafru.it/s1C)

Install Drivers (Windows 7 Only)

When you plug in the board, you'll need to possibly install a driver

Click below to download our Driver Installer

https://adafru.it/zek

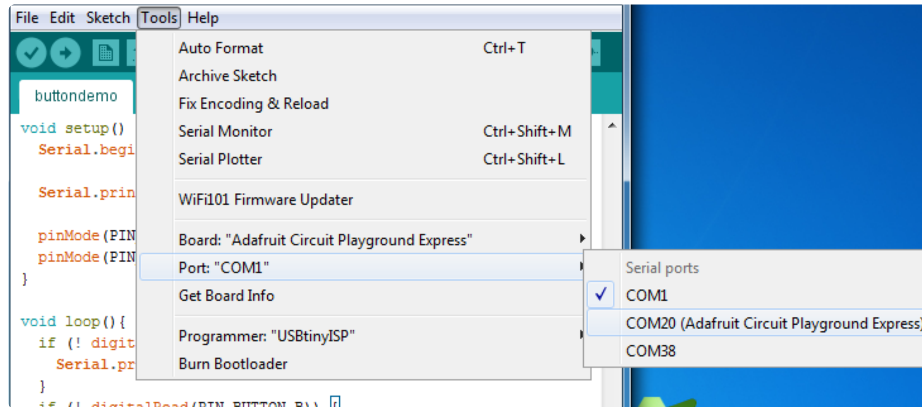
https://adafru.it/zek

Download and run the installer, more details are here (https://adafru.it/Bf7)

Blink

Now you can upload your first blink sketch!

Plug in the Circuit Playground Express and wait for it to be recognized by the OS (just takes a few seconds). It will create a serial/COM port, you can now select it from the dropdown, it'll even be 'indicated' as a Circuit Playground Express board!



Now load up the Blink 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
}
```

And click upload! That's it, you will be able to see the LED blink rate change as you adapt the **delay()** calls.

If you are having issues, make sure you selected the matching Board in the menu that matches the hardware you have in your hand.

Sucessful Upload

If you have a successful upload, you'll get a bunch of red text that tells you that the device was found and it was programmed, verified & reset

```
Done uploading.
Write 11024 bytes to flash (173 pages)

[===========                   ] 36% (64/173 pages)
[======================        ] 73% (128/173 pages)
[==============================] 100% (173/173 pages)

done in 0.097 seconds


Verify 11024 bytes of flash with checksum.

Verify successful

done in 0.049 seconds

CPU reset.

6                                   Adafruit Feather M0 (Native USB Port) on COM54
```

Compilation Issues

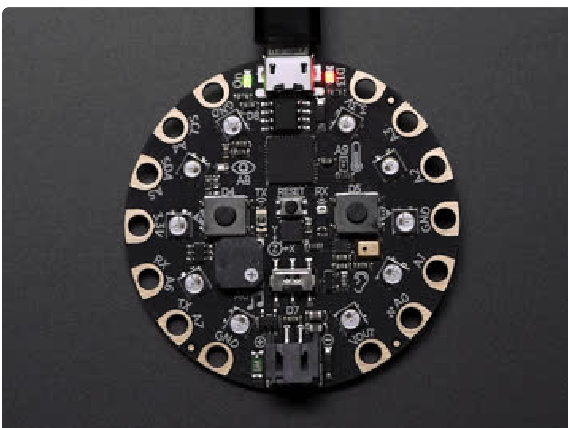
If you get an alert that looks like

Cannot run program "{runtime.tools.arm-none-eabi-gcc.path}\bin\arm-non-eabi-g++"

Make sure you have installed the **Arduino SAMD** boards package, try re-installing it or getting the latest version! Also try updating the IDE

Manually bootloading

Once it is in bootloader mode, you can select the newly created COM/Serial port and re-try uploading.



If you ever get in a 'weird' spot with the bootloader, or you have uploaded code that crashes and doesn't auto-reboot into the bootloader, click the **RESET** button **twice** (like a double-click) to get back into the bootloader.

The red LED will pulse, and all the NeoPixels will turn green so you know that its in bootloader mode.

In the Arduino IDE, **re-select the Serial Port** to the new port that has been created for the bootloader.

Then upload **Blink** - make sure that works!

Once that works, go back and re-select the 'normal' USB serial port next time you want to use the normal upload.

Ubuntu & Linux Issue Fix

Note if you're using Ubuntu 15.04 (or perhaps other more recent Linux distributions) there is an issue with the modem manager service which causes the Bluefruit LE micro to be difficult to program. If you run into errors like "device or resource busy", "bad file descriptor", or "port is busy" when attempting to program then you are hitting this issue. (https://adafru.it/sHE)

The fix for this issue is to make sure Adafruit's custom udev rules are applied to your system. One of these rules is made to configure modem manager not to touch the Feather board and will fix the programming difficulty issue. Follow the steps for installing Adafruit's udev rules on this page. (https://adafru.it/iOE)

Arduino Switches

The first part of interfacing with hardware is being able to manage digital inputs and outputs. With the built in Circuit Playground hardware & the Arduino library it's super easy!

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

```
#include <Adafruit_CircuitPlayground.h>

void setup() {
  // Initialize the circuit playground
  CircuitPlayground.begin();
}

void loop() {
  // If the left button is pressed....
  if (CircuitPlayground.leftButton()) {
      CircuitPlayground.redLED(HIGH);  // LED on
  } else {
      CircuitPlayground.redLED(LOW);   // LED off
  }
}
```

Note that we made the code a little less elegant than necessary, the if/then could be replaced with a simple `CircuitPlayground.redLED(CircuitPlayground.leftButton())` but I wanted to make it super clear how to test the inputs.

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

Without Library Assist

If you are interested in 'lower level' interfacing to the hardware, you don't *have* to use the CircuitPlayground helper library. This code shows manually setting the button/LED pins to inputs & outputs as well as setting and reading the state of the pins:

```
#include <Adafruit_CircuitPlayground.h>

void setup() {
  pinMode(CPLAY_LEFTBUTTON, INPUT_PULLDOWN);
  pinMode(CPLAY_REDLED, OUTPUT);
}

void loop() {
  // If the left button is pressed....
  if (digitalRead(CPLAY_LEFTBUTTON)) {
      digitalWrite(CPLAY_REDLED, HIGH);  // LED on
  } else {
      digitalWrite(CPLAY_REDLED, LOW);  // LED off
  }
}
```

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

Maybe you're setting up your own external button with pullup or pulldown resistor. If you want to turn off the internal pullup/pulldown just set the `pinMode()` to plain `INPUT`

Adapting Sketches to M0 & M4

The ATSAMD21 and 51 are very nice little chips, but fairly new as Arduino-compatible cores go. **Most** sketches & libraries will work but here's a collection of things we noticed.

The notes below cover a range of Adafruit M0 and M4 boards, but not every rule will apply to every board (e.g. Trinket and Gemma M0 do not have ARef, so you can skip the Analog References note!).

Analog References

If you'd like to use the **ARef** pin for a non-3.3V analog reference, the code to use is `analogReference(AR_EXTERNAL)` (it's AR_EXTERNAL not EXTERNAL)

Pin Outputs & Pullups

The old-style way of turning on a pin as an input with a pullup is to use

```
pinMode(pin, INPUT)
digitalWrite(pin, HIGH)
```

This is because the pullup-selection register on 8-bit AVR chips is the same as the output-selection register.

For M0 & M4 boards, you can't do this anymore! Instead, use:

```
pinMode(pin, INPUT_PULLUP)
```

Code written this way still has the benefit of being *backwards compatible with AVR.* You don't need separate versions for the different board types.

Serial vs SerialUSB

99.9% of your existing Arduino sketches use **Serial.print** to debug and give output. For the Official Arduino SAMD/M0 core, this goes to the Serial5 port, which isn't exposed on the Feather. The USB port for the Official Arduino M0 core is called ***SerialUSB*** instead.

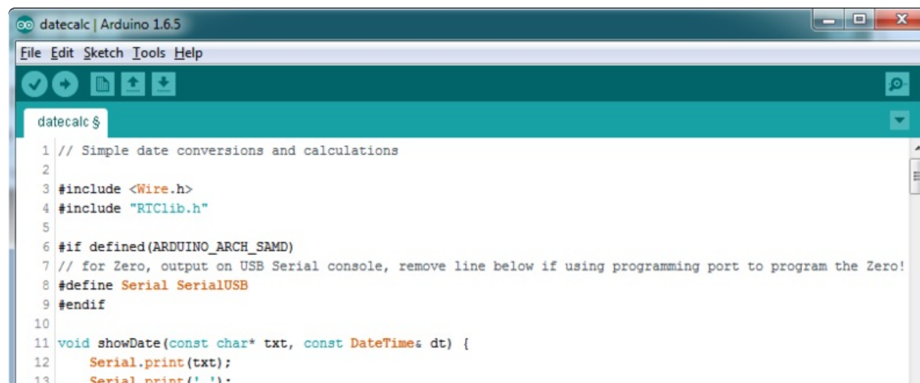
In the Adafruit M0/M4 Core, we fixed it so that **Serial goes to USB so it will automatically work just fine** .

However, on the off chance you are using the official Arduino SAMD core and *not* the Adafruit version (which really, we recommend you use our version because it's been tuned to our boards), and you want your Serial prints and reads to use the USB port, use *SerialUSB* instead of *Serial* in your sketch.

If you have existing sketches and code and you want them to work with the M0 without a huge find-replace, put

```
#if defined(ARDUINO_SAMD_ZERO) && defined(SERIAL_PORT_USBVIRTUAL)
 // Required for Serial on Zero based boards
 #define Serial SERIAL_PORT_USBVIRTUAL
#endif
```

right above the first function definition in your code. For example:



AnalogWrite / PWM on Feather/Metro M0

After looking through the SAMD21 datasheet, we've found that some of the options listed in the multiplexer table don't exist on the specific chip used in the Feather M0.

For all SAMD21 chips, there are two peripherals that can generate PWM signals: The Timer/Counter (TC) and Timer/Counter for Control Applications (TCC). Each SAMD21 has multiple copies of each, called 'instances'.

Each TC instance has one count register, one control register, and two output channels. Either channel can be enabled and disabled, and either channel can be inverted. The pins connected to a TC instance can output identical versions of the same PWM waveform, or complementary waveforms.

Each TCC instance has a single count register, but multiple compare registers and output channels. There are options for different kinds of waveform, interleaved switching, programmable dead time, and so on.

The biggest members of the SAMD21 family have five TC instances with two 'waveform output' (WO) channels, and three TCC instances with eight WO channels:

- TC[0-4],WO[0-1]
- TCC[0-2],WO[0-7]

And those are the ones shown in the datasheet's multiplexer tables.

The SAMD21G used in the Feather M0 only has three TC instances with two output channels, and three

TCC instances with eight output channels:

- **TC**[**3-5**],**WO**[**0-1**]
- **TCC**[**0-2**],**WO**[**0-7**]

Tracing the signals to the pins broken out on the Feather M0, the following pins can't do PWM at all:

- **Analog pin A5**

The following pins can be configured for PWM without any signal conflicts as long as the SPI, I2C, and UART pins keep their protocol functions:

- **Digital pins 5, 6, 9, 10, 11, 12, and 13**
- **Analog pins A3 and A4**

If only the SPI pins keep their protocol functions, you can also do PWM on the following pins:

- **TX and SDA (Digital pins 1 and 20)**

analogWrite() PWM range

On AVR, if you set a pin's PWM with `analogWrite(pin, 255)` it will turn the pin fully HIGH. On the ARM cortex, it will set it to be 255/256 so there will be very slim but still-existing pulses-to-0V. If you need the pin to be fully on, add test code that checks if you are trying to `analogWrite(pin, 255)` and, instead, does a `digitalWrite(pin, HIGH)`

analogWrite() DAC on A0

If you are trying to use `analogWrite()` to control the DAC output on **A0**, make sure you do **not** have a line that sets the pin to output. ***Remove***: `pinMode(A0, OUTPUT)`.

Missing header files

There might be code that uses libraries that are not supported by the M0 core. For example if you have a line with

```
#include <util/delay.h>
```

you'll get an error that says

```
fatal error: util/delay.h: No such file or directory
 #include <util/delay.h>
```

```
             ^
compilation terminated.
Error compiling.
```

In which case you can simply locate where the line is (the error will give you the file name and line number) and 'wrap it' with #ifdef's so it looks like:

```
#if !defined(ARDUINO_ARCH_SAM) && !defined(ARDUINO_ARCH_SAMD) && !defined(ESP8266) &&
!defined(ARDUINO_ARCH_STM32F2)
 #include <util/delay.h>
#endif
```

The above will also make sure that header file isn't included for other architectures

If the #include is in the arduino sketch itself, you can try just removing the line.

Bootloader Launching

For most other AVRs, clicking **reset** while plugged into USB will launch the bootloader manually, the bootloader will time out after a few seconds. For the M0/M4, you'll need to ***double click*** the button. You will see a pulsing red LED to let you know you're in bootloader mode. Once in that mode, it wont time out! Click reset again if you want to go back to launching code.

Aligned Memory Access

This is a little less likely to happen to you but it happened to me! If you're used to 8-bit platforms, you can do this nice thing where you can typecast variables around. e.g.

```
uint8_t mybuffer[4];
float f = (float)mybuffer;
```

You can't be guaranteed that this will work on a 32-bit platform because **mybuffer** might not be aligned to a 2 or 4-byte boundary. The ARM Cortex-M0 can only directly access data on 16-bit boundaries (every 2 or 4 bytes). Trying to access an odd-boundary byte (on a 1 or 3 byte location) will cause a Hard Fault and stop the MCU. Thankfully, there's an easy work around ... just use memcpy!

```
uint8_t mybuffer[4];
float f;
memcpy(&f, mybuffer, 4)
```

Floating Point Conversion

Like the AVR Arduinos, the M0 library does not have full support for converting floating point numbers to ASCII strings. Functions like sprintf will not convert floating point. Fortunately, the standard AVR-LIBC

library includes the dtostrf function which can handle the conversion for you.

Unfortunately, the M0 run-time library does not have dtostrf. You may see some references to using **#include <avr/dtostrf.h>** to get dtostrf in your code. And while it will compile, it does **not** work.

Instead, check out this thread to find a working dtostrf function you can include in your code:

http://forum.arduino.cc/index.php?topic=368720.0 (https://adafru.it/IFS)

How Much RAM Available?

The ATSAMD21G18 has 32K of RAM, but you still might need to track it for some reason. You can do so with this handy function:

```
extern "C" char *sbrk(int i);

int FreeRam () {
  char stack_dummy = 0;
  return &stack_dummy - sbrk(0);
}
```

Thx to http://forum.arduino.cc/index.php?topic=365830.msg2542879#msg2542879 (https://adafru.it/m6D) for the tip!

Storing data in FLASH

If you're used to AVR, you've probably used **PROGMEM** to let the compiler know you'd like to put a variable or string in flash memory to save on RAM. On the ARM, its a little easier, simply add **const** before the variable name:

const char str[] = "My very long string";

That string is now in FLASH. You can manipulate the string just like RAM data, the compiler will automatically read from FLASH so you dont need special progmem-knowledgeable functions.

You can verify where data is stored by printing out the address:
Serial.print("Address of str $"); Serial.println((int)&str, HEX);

If the address is $2000000 or larger, its in SRAM. If the address is between $0000 and $3FFFF Then it is in FLASH

Pretty-Printing out registers

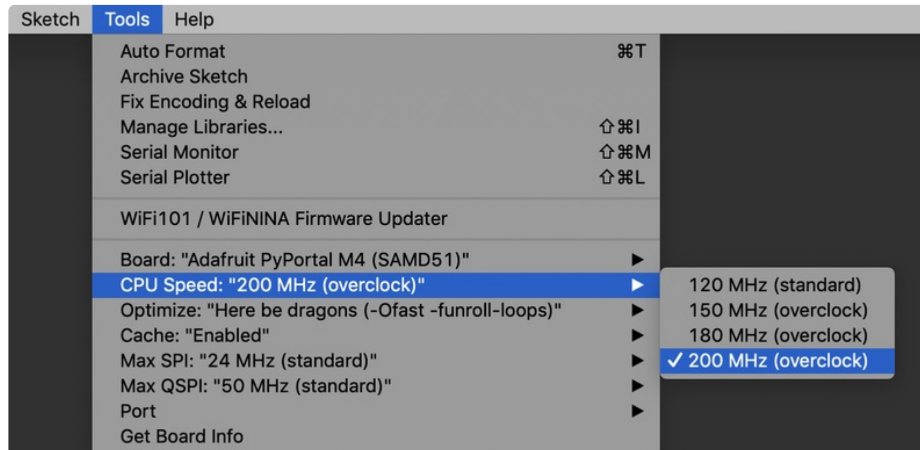
There's *a lot* of registers on the SAMD21, and you often are going through ASF or another framework to

get to them. So having a way to see exactly what's going on is handy. This library from drewfish will help a ton!

https://github.com/drewfish/arduino-ZeroRegs (https://adafru.it/Bet)

M4 Performance Options

As of version 1.4.0 of the *Adafruit SAMD Boards* package in the Arduino Boards Manager, some options are available to wring extra performance out of M4-based devices. These are in the *Tools* menu.



All of these performance tweaks involve a degree of uncertainty. There's *no guarantee* of improved performance in any given project, and *some may even be detrimental,* failing to work in part or in whole. If you encounter trouble, **select the default performance settings** and re-upload.

Here's what you get and some issues you might encounter...

CPU Speed (overclocking)

This option lets you adjust the microcontroller core clock...the speed at which it processes instructions... beyond the official datasheet specifications.

Manufacturers often rate speeds conservatively because such devices are marketed for harsh industrial environments...if a system crashes, someone could lose a limb or worse. But most creative tasks are less critical and operate in more comfortable settings, and we can push things a bit if we want more speed.

There is a small but nonzero chance of code **locking up** or **failing to run** entirely. If this happens, try **dialing back the speed by one notch and re-upload**, see if it's more stable.

Much more likely, **some code or libraries may not play well** with the nonstandard CPU speed. For example, currently the NeoPixel library assumes a 120 MHz CPU speed and won't issue the correct data at other settings (this will be worked on). Other libraries may exhibit similar problems, usually anything that strictly depends on CPU timing...you might encounter problems with audio- or servo-related code

depending how it's written. **If you encounter such code or libraries, set the CPU speed to the default 120 MHz and re-upload.**

Optimize

There's usually more than one way to solve a problem, some more resource-intensive than others. Since Arduino got its start on resource-limited AVR microcontrollers, the C++ compiler has always aimed for the **smallest compiled program size**. The "Optimize" menu gives some choices for the compiler to take different and often faster approaches, at the expense of slightly larger program size...with the huge flash memory capacity of M4 devices, that's rarely a problem now.

The "**Small**" setting will compile your code like it always has in the past, aiming for the smallest compiled program size.

The "**Fast**" setting invokes various speed optimizations. The resulting program should produce the same results, is slightly larger, and usually (but not always) noticably faster. It's worth a shot!

"**Here be dragons**" invokes some more intensive optimizations...code will be larger still, faster still, but there's a possibility these optimizations could cause unexpected behaviors. *Some code may not work the same as before.* Hence the name. Maybe you'll discover treasure here, or maybe you'll sail right off the edge of the world.

Most code and libraries will continue to function regardless of the optimizer settings. If you do encounter problems, **dial it back one notch and re-upload**.

Cache

This option allows a small collection of instructions and data to be accessed more quickly than from flash memory, boosting performance. It's enabled by default and should work fine with all code and libraries. But if you encounter some esoteric situation, the cache can be disabled, then recompile and upload.

Max SPI and Max QSPI

These should probably be left at their defaults. They're present mostly for our own experiments and can cause **serious headaches**.

Max SPI determines the clock source for the M4's SPI peripherals. Under normal circumstances this allows transfers up to 24 MHz, and should usually be left at that setting. But...if you're using write-only SPI devices (such as TFT or OLED displays), this option lets you drive them faster (we've successfully used 60 MHz with some TFT screens). The caveat is, if using *any* read/write devices (such as an SD card), *this will not work at all...*SPI reads *absolutely* max out at the default 24 MHz setting, and anything else will fail. **Write** = **OK. Read** = **FAIL.** This is true *even if your code is using a lower bitrate setting...* just having the different clock source prevents SPI reads.

Max QSPI does similarly for the extra flash storage on M4 “Express” boards. *Very few* Arduino sketches access this storage at all, let alone in a bandwidth-constrained context, so this will benefit next to nobody. Additionally, due to the way clock dividers are selected, this will only provide some benefit when certain “CPU Speed” settings are active. Our PyPortal Animated GIF Display (https://adafru.it/EkO) runs marginally better with it, if using the QSPI flash.

Enabling the Buck Converter on some M4 Boards

If you want to reduce power draw, some of our boards have an inductor so you can use the 1.8V buck converter instead of the built in linear regulator. If the board does have an inductor (see the schematic) you can add the line `SUPC->VREG.bit.SEL = 1;` to your code to switch to it. Note it will make ADC/DAC reads a bit noisier so we don't use it by default. You'll save ~4mA (https://adafru.it/F0H).

Troubleshooting

Using external NeoPixel strips/dots with Circuit Playground in Arduino

We have an 'internal' version of the NeoPixel library in the Arduino CircuitPlayground library. If you want to use external LEDs, make sure the top of your sketch includes the libraries so that NeoPixel comes first. For example:

```
#include <Adafruit_NeoPixel.h>
#include <Adafruit_CircuitPlayground.h>
```

in that order!

CPLAYBOOT Does not Appear on Windows 10; "ARM7TDMI" Error in Arduino

Some people have had difficulty with the Circuit Playground Express on Windows 10 after installing the Arduino IDE and the Arduino Board Support Packages **Arduino SAMD** and/or **Adafruit SAMD**. The problem is that these Board Support Packages install unnecessary drivers on Windows 10. This causes two problems:

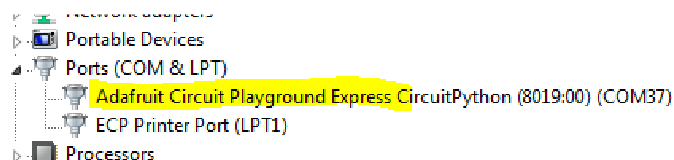
1. **CPLAYBOOT** does not appear when double-clicking the reset button on the board.

2. Attempting to upload an Arduino program generates an error like:

```
An error occurred while uploading the sketch
chipId=0x30455553
Unsupported ARM7TDMI architecture
```

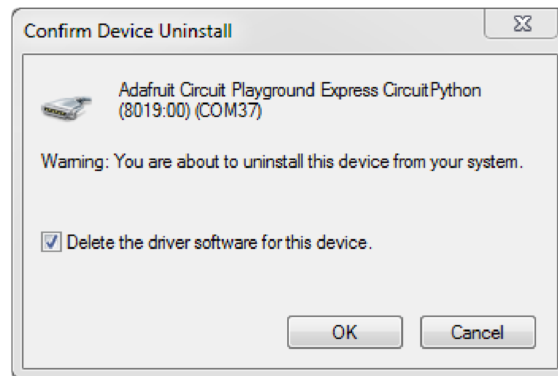
We are working with Arduino to update these packages to fix the problem. In the meantime, the fix is to uninstall the bad drivers. Three unnecessary drivers are installed, one for **CPLAYBOOT**, one for **CIRCUITPY**, and one for Arduino programs running on the Circuit Playground Express. You should uninstall all of them.

In all three cases, you open the **Device Manager**, by right-clicking on the Start menu and choosing Device Manager. Then open the **Ports** section in the new window. You'll see something like the the listing below. The wording will vary, but it will include "Adafruit Circuit Playground Express".



You want to uninstall the driver for that device. Right-click on the device, choose **Uninstall Device,** and

when the dialog box comes up, **check the box that says "Delete the driver software for this device".** and click **OK**. It will look something like this:



You do this three times, following the procedure above each time.

1. First, double-click the reset button, so all the NeoPixels turn green. Normally **CPLAYBOOT** would appear but if the incorrect driver is installed, it will not appear. Then unplug and replug in the board.

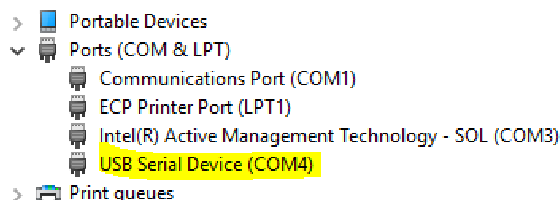
2. Double-click the reset button again. This time **CPLAYBOOT** should appear. Upload an Arduino program to the board, such as Blink, following the directions in the [Arduino](https://adafru.it/C3t) (https://adafru.it/C3t) section of this guide.

3. Uninstall when **CIRCUITPY** is visible as a drive, if you have CircuitPython installed on the board.

Each time you'll see an entry in the Device Manager similar to the above, with slightly different wording, and a different COM port. Uninstall the device as shown above.

If you are pressed for time, number 1. is the most important driver to uninstall.

After you uninstall each device, you can unplug and plug the board in again, and it will reinstall, using a built-in Windows driver, instead of the incorrect one. Double-click the reset button to get **CPLAYBOOT** and reinstall that as well. The new devices will show up in **Device Manager** as "USB Serial Device (COMxx)" instead of "Adafruit Circuit Playground", similar to the image below:



If this does not solve the problem of CPLAYBOOT not appearing and the ARM7TDMI error, please post in [https://learn.adafruit.com](https://adafru.it/id3) (https://adafru.it/id3)and let us know.

Accessories

Circuit Playground Express is a great tool for teaching. It has a ton built in, but there's so much more you can do with it. Check out the lists below for some compatible products that will add more to your projects!

Packs

These packs will help get you started with some accessories already included.

Circuit Playground Express - Base Kit

It's the Circuit Playground Express Base Kit! It provides the few things you'll need to get started with the new

Out of Stock

Out of Stock

Circuit Playground Express Advanced Pack

Circuit Playground Express is the next step towards a perfect introduction to electronics and programming. We've taken the original Circuit Playground Classic...

$99.95

In Stock

Add to Cart

Connections

These products can be used to connect all kinds of interesting things to the Circuit Playground Express such as sensors, servos, and bananas.

Small Alligator Clip Test Lead (Set of 18)

Connect this to that without soldering using these mini alligator clip test leads. 15" cables with alligator clip on each end in six colors. Strong and grippy, these always come...

$5.95

In Stock

Add to Cart

Small Alligator Clip to Male Jumper Wire Bundle - 12 Pieces

For bread-boarding with unusual non-header-friendly surfaces, these cables will be your best friends! No longer will you have long strands of alligator clips that are grabbing little...

$7.95

In Stock

Add to Cart

Small Alligator Clip to Male Jumper Wire Bundle - 6 Pieces

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

Out of Stock

Out of Stock

Conductive Hook & Loop Tape - 3" long

Conductive hook & loop tape is just like the stuff you've seen on jackets, clothes, shoes, and bags, but is coated with silver to make it fully conductive. Originally this was...

$6.95

In Stock

Add to Cart

Capacitive Touch

These products can be used with the capacitive touch pads to create interactive art that reacts to human touch.

Bare Conductive Paint - 50mL

Bare Conductive Paint is a multipurpose electrically conductive material perfect for all of your DIY projects! Bare Paint is water based, nontoxic and dries at room temperature.

Out of Stock

Out of Stock

Bare Conductive Paint Pen - 10mL

Bare Conductive Paint is a multipurpose electrically conductive material perfect for all of your DIY projects! Bare Paint is water based, nontoxic and dries at room temperature.

Out of Stock

Out of Stock

Conductive Silver Ink Pen - Standard Tip

There are two 'tips' available - standard and micro. The microtip allows for finer control but is a little harder to use (because it doesn't deposit as...

Out of Stock

Out of Stock

Conductive Silver Ink Pen - Micro Tip

Experiment with paper electronics with this silver conductive ink pen that will let you draw traces! We like the make of this pen, it has a nice liquid silver ink that flows easily,...

$49.95

In Stock

Add to Cart

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

Copper tape can be an interesting addition to your toolbox. The tape itself is made of thin pure copper so its extremely flexible and can take on nearly any shape. You can easily...

$19.95

In Stock

Add to Cart

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

Copper tape can be an interesting addition to your toolbox. The tape itself is made of thin pure copper so its extremely flexible and can take on nearly any shape. You can easily...

$5.95

In Stock

Add to Cart

Proto-Pasta - 1.75mm 500g Conductive PLA Filament

It'd be Fusilli not to buy this filament! Pici up this filament today! Did we...

$58.00

In Stock

Add to Cart

Proto-pasta - 2.85mm Diameter - Conductive Graphite Filament

Pici up this filament today! It'd be Fusilli not to buy...

Out of Stock

Out of Stock

Robotics

These products can add some robot friend flavor to your project.

Micro servo

Tiny little servo can rotate approximately 180 degrees (90 in each direction), and works just like the standard kinds you're used to but smaller. You can use any servo...

$5.95

In Stock

Add to Cart

Your browser does not support the video tag.

Continuous Rotation Micro Servo

Need to make a tiny robot? This little micro servo rotates 360 degrees fully forward or backwards, instead of moving to a single position. You can use any servo code, hardware...

$7.50

In Stock

Add to Cart

Wheel for Micro Continuous Rotation FS90R Servo

We're keepin' it wheel with this one!Need a great drive solution for your little robotic friends? This black plastic Micro Continuos...

$2.50

In Stock

Add to Cart

Sewable

You can use the following products to sew additions onto your project.

Flora RGB Smart NeoPixel version 2 - Pack of 4

What's a wearable project without LEDs? Our favorite part of the Flora platform is these tiny smart pixels. Designed specifically for wearables, these updated Flora NeoPixels have...

$7.95

In Stock

Add to Cart

Stainless Thin Conductive Thread - 2 ply - 23 meter/76 ft

After months of searching, we finally have what we consider to be the ultimate conductive thread. It's thin, strong, smooth, and made completely of 316L stainless steel. Once you...

$6.95

In Stock

Add to Cart

Sewable Snaps - 5mm Diameter - Card of 24

The small 5mm size of these tin-plated brass snaps means they fit perfectly on Flora's pads! Snaps make a great connector for wearables-- solder one side to the board and sew the...

$3.95

In Stock

Add to Cart

Needle set - 3/9 sizes - 20 needles

Mighty needles, sew like the wind! This needle set is the only one you'll need for any sort of hand sewing, especially using our conductive thread and wearable electronics...

$1.95

In Stock

Add to Cart

Power

These products can be used to power your project without using a USB cable.

JST-PH Battery Extension Cable - 500mm

By popular demand, we now have a handy extension cord for all of our JST-terminated battery packs (such as our LiIon/LiPoly and 3xAAA holders). One end has a JST-PH socket, and the...

$1.95

In Stock

Add to Cart

JST 2-pin Extension Cable with On/Off Switch - JST PH2

By popular request - we now have a way you can turn on-and-off Lithium Polymer batteries without unplugging them.This PH2 Female/Male JST 2-pin Extension...

$2.95

In Stock

Add to Cart

3 x AAA Battery Holder with On/Off Switch, JST, and Belt Clip

This battery holder connects 3 AAA batteries together in series for powering all kinds of projects. We spec'd these out because the box is slim, and 3 AAA's add up to about...

$2.95

In Stock

Add to Cart

3 x AA Battery Holder with On/Off Switch, JST, and Belt Clip

This battery holder connects 3 AA batteries together in series for powering all kinds of projects. We spec'd these out because the box is compact, and 3 AA's add up to about...

$2.95

In Stock

Add to Cart

3 x AAA Battery Holder with On/Off Switch and 2-Pin JST

This battery holder connects 3 AAA batteries together in series for powering all kinds of projects. We spec'd these out because the box is slim, and 3 AAA's add up to about...

Out of Stock

Out of Stock

Lithium Ion Polymer Battery - 3.7v 500mAh

Lithium ion polymer (also known as 'lipo' or 'lipoly') batteries are thin, light and powerful. The output ranges from 4.2V when completely charged to 3.7V. This battery...

$7.95

In Stock

Add to Cart

Lithium Ion Polymer Battery - 3.7v 1200mAh

Lithium ion polymer (also known as 'lipo' or 'lipoly') batteries are thin, light and powerful. The output ranges from 4.2V when completely charged to 3.7V. This battery...

$9.95

In Stock

Add to Cart

Lithium Ion Polymer Battery - 3.7v 350mAh

Lithium ion polymer (also known as 'lipo' or 'lipoly') batteries are thin, light and powerful. The output ranges from 4.2V when completely charged to 3.7V. This battery...

$6.95

In Stock

Add to Cart

---

2 x 2032 Coin Cell Battery Holder - 6V output with On/Off switch

This tiny coin cell battery holder is ideal for small portable or wearable projects. It holds two 20mm coin cells (2032 are the most popular size) in series to generate 6V nominal. (If...

$1.95

In Stock

Add to Cart

These products can be used to charge the lithium ion polymer batteries listed above.

Adafruit Micro-Lipo Charger for LiPo/LiIon Batt w/MicroUSB Jack

Oh so handy, this little lipo charger is so small and easy to use you can keep it on your desk or mount it easily into any project! Simply plug it via any MicroUSB cable into a USB...

$6.95

In Stock

Add to Cart

PowerBoost 500 Charger - Rechargeable 5V Lipo USB Boost @ 500mA+

PowerBoost 500C is the perfect power supply for your portable project! With a built-in battery charger circuit, you'll be able to keep your project running even while recharging...

$14.95

In Stock

Add to Cart

UF2 Bootloader Details

> This is an information page for advanced users who are curious how we get code from your computer into your Express board!

Adafruit SAMD21 (M0) and SAMD51 (M4) boards feature an improved bootloader that makes it easier than ever to flash different code onto the microcontroller. This bootloader makes it easy to switch between Microsoft MakeCode, CircuitPython and Arduino.

Instead of needing drivers or a separate program for flashing (say, `bossac`, `jlink` or `avrdude`), one can simply ***drag a file onto a removable drive***.

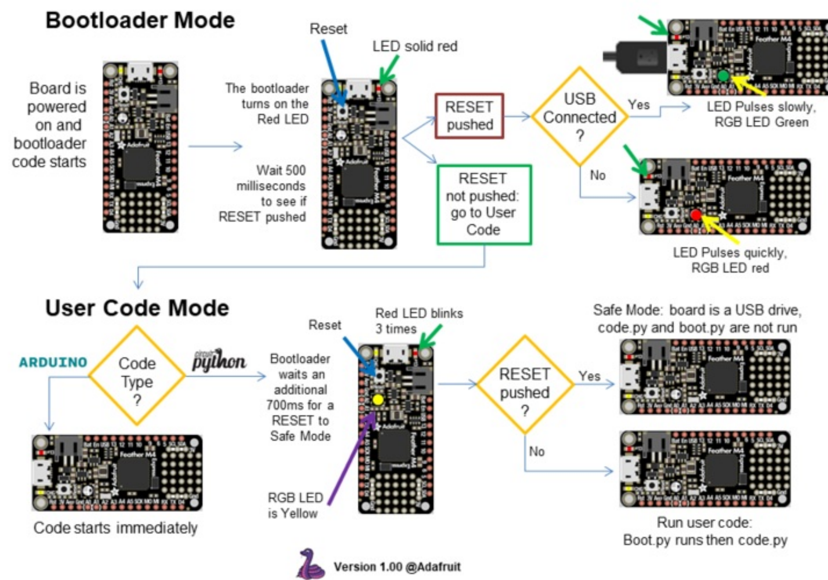
The format of the file is a little special. Due to 'operating system woes' you cannot just drag a binary or hex file (trust us, we tried it, it isn't cross-platform compatible). Instead, the format of the file has extra information to help the bootloader know where the data goes. The format is called UF2 (USB Flashing Format). Microsoft MakeCode generates UF2s for flashing and CircuitPython releases are also available as UF2. You can also create your own UF2s from binary files using uf2tool, available here. (https://adafru.it/vPE)

The bootloader is ***also BOSSA compatible***, so it can be used with the Arduino IDE which expects a BOSSA bootloader on ATSAMD-based boards

For more information about UF2, you can read a bunch more at the MakeCode blog (https://adafru.it/w5A), then check out the UF2 file format specification. (https://adafru.it/vPE)

Visit Adafruit's fork of the Microsoft UF2-samd bootloader GitHub repository (https://adafru.it/Beu) for source code and releases of pre-built bootloaders on CircuitPython.org (https://adafru.it/Em8).

> The bootloader is not needed when changing your CircuitPython code. Its only needed when upgrading the CircuitPython core or changing between CircuitPython, Arduino and Microsoft MakeCode.

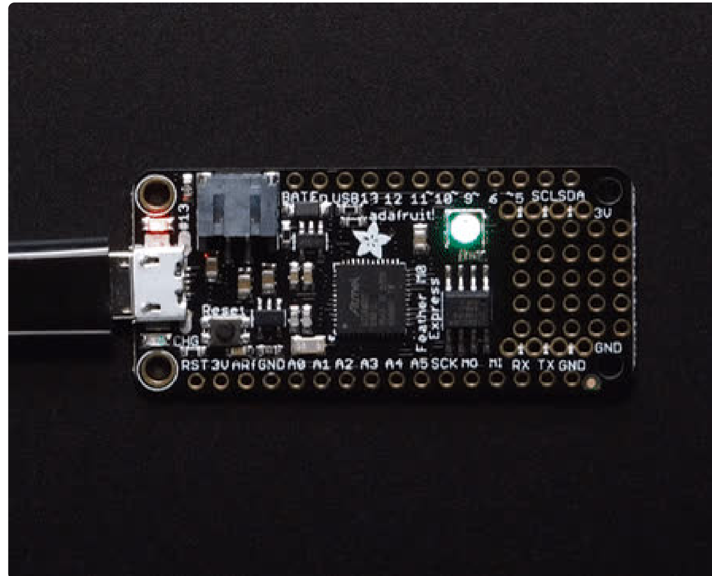


Entering Bootloader Mode

The first step to loading new code onto your board is triggering the bootloader. It is easily done by double tapping the reset button. Once the bootloader is active you will see the small red LED fade in and out and a new drive will appear on your computer with a name ending in **BOOT**. For example, feathers show up as **FEATHERBOOT**, while the new CircuitPlayground shows up as **CPLAYBOOT**, Trinket M0 will show up as **TRINKETBOOT**, and Gemma M0 will show up as **GEMMABOOT**

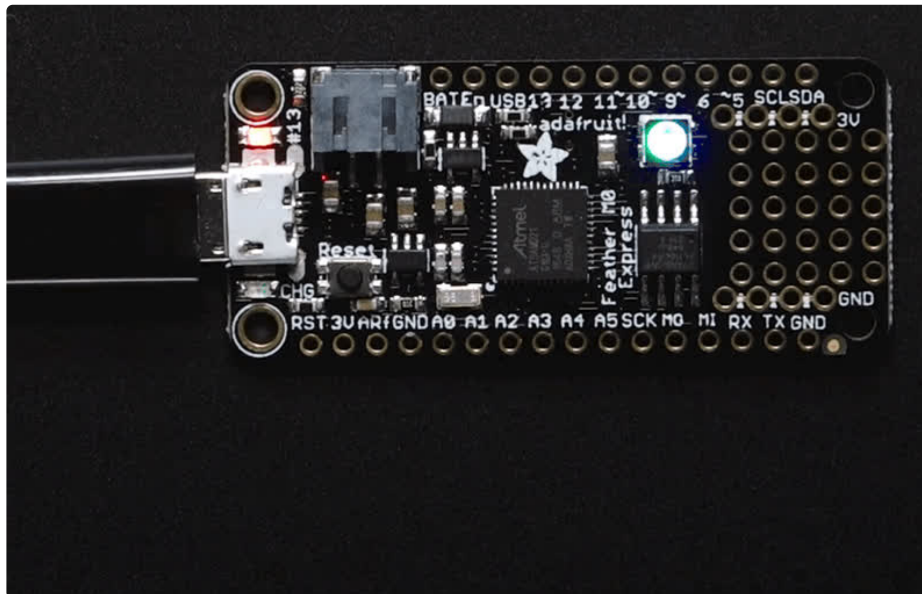
Furthermore, when the bootloader is active, it will change the color of one or more onboard neopixels to indicate the connection status, red for disconnected and green for connected. If the board is plugged in but still showing that its disconnected, try a different USB cable. Some cables only provide power with no communication.

For example, here is a Feather M0 Express running a colorful Neopixel swirl. When the reset button is double clicked (about half second between each click) the NeoPixel will stay green to let you know the bootloader is active. When the reset button is clicked once, the 'user program' (NeoPixel color swirl) restarts.

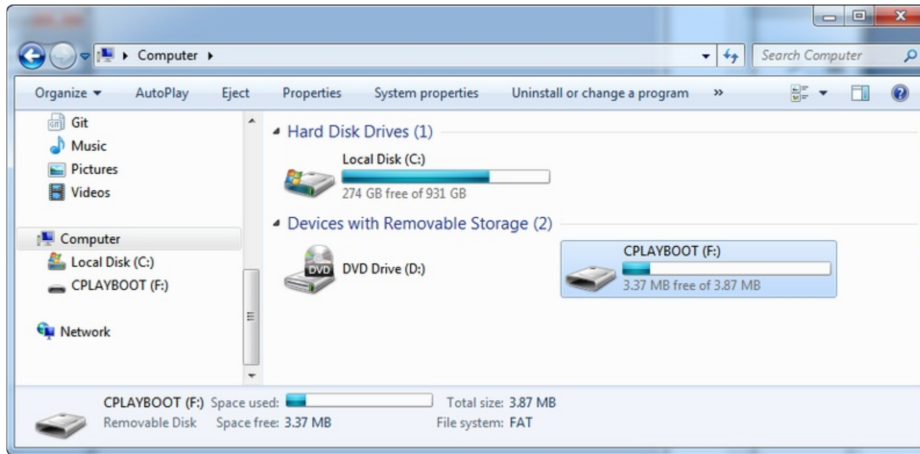


If the bootloader couldn't start, you will get a red NeoPixel LED.

That could mean that your USB cable is no good, it isn't connected to a computer, or maybe the drivers could not enumerate. Try a new USB cable first. Then try another port on your computer!

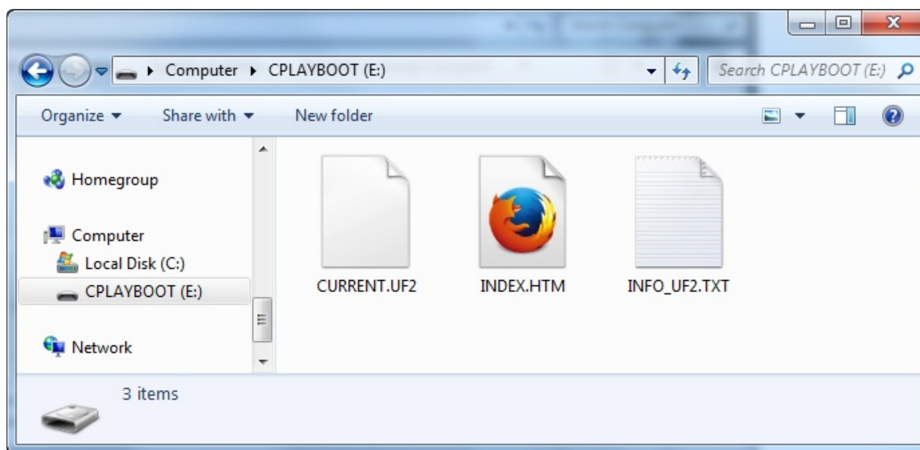


Once the bootloader is running, check your computer. You should see a USB Disk drive...



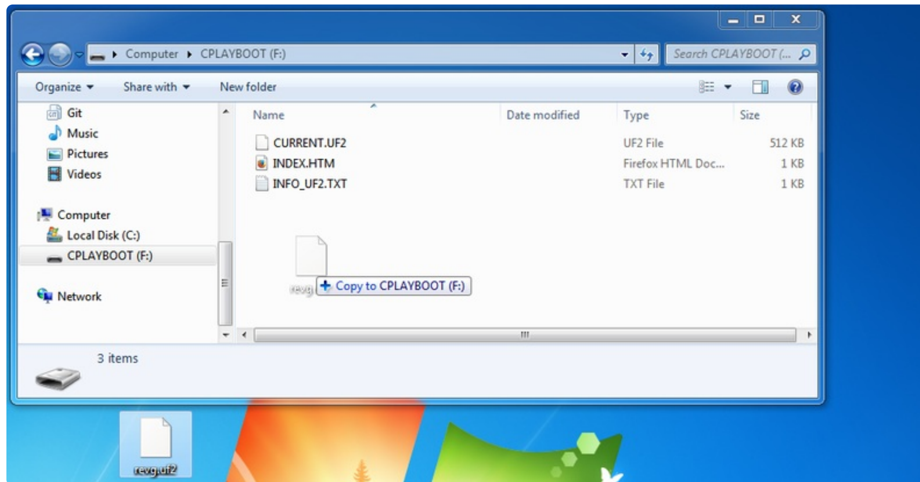
Once the bootloader is successfully connected you can open the drive and browse the virtual filesystem. This isn't the same filesystem as you use with CircuitPython or Arduino. It should have three files:

- **CURRENT.UF2** - The current contents of the microcontroller flash.
- **INDEX.HTM** - Links to Microsoft MakeCode.
- **INFO_UF2.TXT** - Includes bootloader version info. Please include it on bug reports.

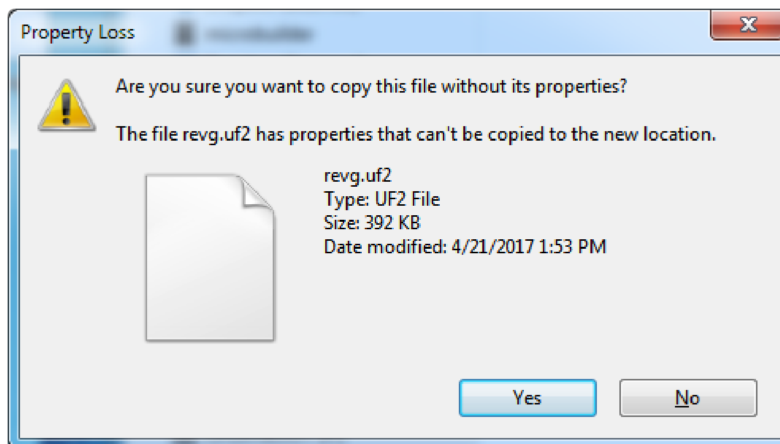


Using the Mass Storage Bootloader

To flash something new, simply drag any UF2 onto the drive. After the file is finished copying, the bootloader will automatically restart. This usually causes a warning about an unsafe eject of the drive. However, its not a problem. The bootloader knows when everything is copied successfully.



You may get an alert from the OS that the file is being copied without it's properties. You can just click **Yes**



You may also get get a complaint that the drive was ejected without warning. Don't worry about this. The drive only ejects once the bootloader has verified and completed the process of writing the new code

Using the BOSSA Bootloader

As mentioned before, the bootloader is also compatible with BOSSA, which is the standard method of updating boards when in the Arduino IDE. It is a command-line tool that can be used in any operating system. We won't cover the full use of the **bossac** tool, suffice to say it can do quite a bit! More information is available at ShumaTech (https://adafru.it/vQa).

Windows 7 Drivers

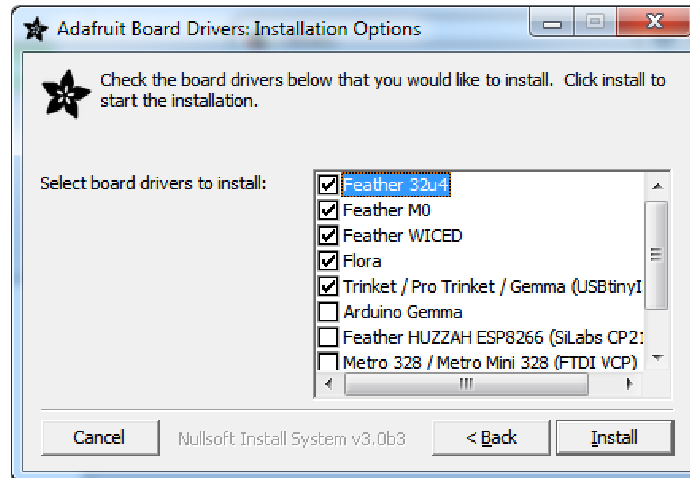
If you are running Windows 7 (or, goodness, something earlier?) You will need a Serial Port driver file. Windows 10 users do not need this so skip this step.

You can download our full driver package here:

https://adafru.it/AB0

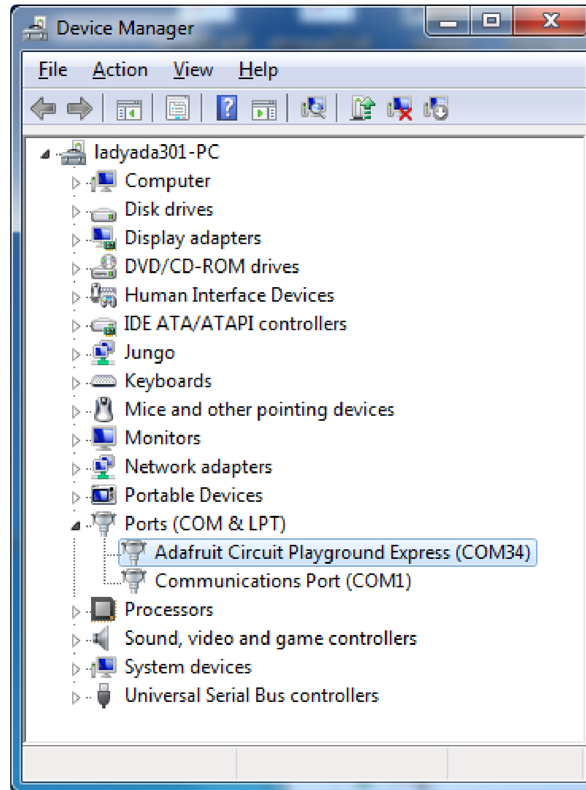
https://adafru.it/AB0

Download and run the installer. We recommend just selecting all the serial port drivers available (no harm to do so) and installing them.

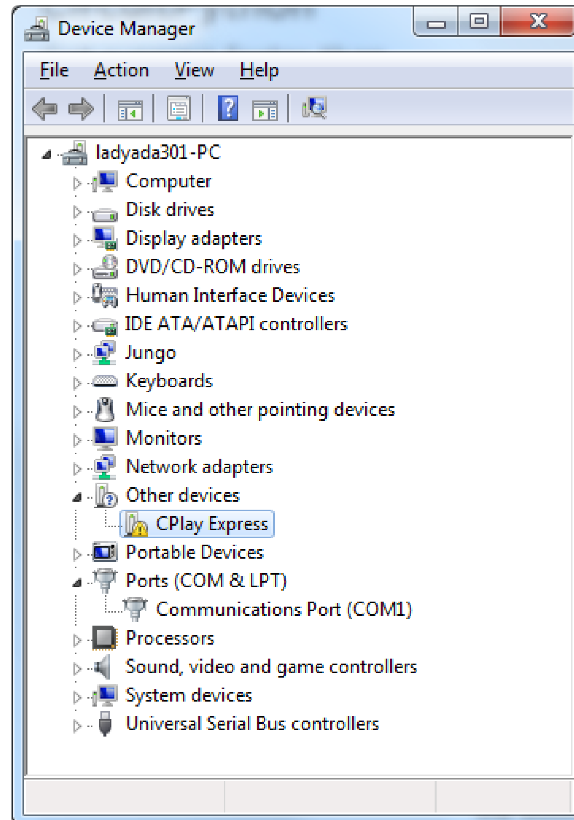


Verifying Serial Port in Device Manager

If you're running Windows, its a good idea to verify the device showed up. Open your Device Manager from the control panel and look under **Ports (COM & LPT)** for a device called **Feather M0** or **Circuit Playground** or whatever!



If you see something like this, it means you did not install the drivers. Go back and try again, then remove and re-plug the USB cable for your board



Running bossac on the command line

If you are using the Arduino IDE, this step is not required. But sometimes you want to read/write custom binary files, say for loading CircuitPython or your own code. We recommend using bossac v 1.7.0 (or greater), which has been tested. The Arduino branch is most recommended (https://adafru.it/vQb).

You can download the latest builds here. (https://adafru.it/s1B) The `mingw32` version is for Windows, `apple-darwin` for Mac OSX and various `linux` options for Linux. Once downloaded, extract the files from the zip and open the command line to the directory with `bossac`.

> With bossac versions 1.9 or later, you must use the --offset parameter on the command line, and it must have the correct value for your board.

With bossac version 1.9 or later, you must give an `--offset` parameter on the command line to specify where to start writing the firmware in flash memory. This parameter was added in bossac 1.8.0 with a default of `0x2000`, but starting in 1.9, the default offset was changed to `0x0000`, which is not what you want in most cases. If you omit the argument for bossac 1.9 or later, you will probably see a "**Verify Failed**" error from bossac. Remember to change the option for `-p` or `--port` to match the port on your Mac.

Replace the filename below with the name of your downloaded `.bin`: it will vary based on your board!

Using bossac Versions 1.7.0, 1.8

There is no `--offset` parameter available. Use a command line like this:

```
bossac -p=/dev/cu.usbmodem14301 -e -w -v -R adafruit-circuitpython-boardname-version.bin
```

For example,

```
bossac -p=/dev/cu.usbmodem14301 -e -w -v -R adafruit-circuitpython-feather_m0_express-3.0.0.bin
```

Using bossac Versions 1.9 or Later

For M0 boards, which have an 8kB bootloader, you must specify `-offset=0x2000`, for example:

```
bossac -p=/dev/cu.usbmodem14301 -e -w -v -R --offset=0x2000 adafruit-circuitpython-feather_m0_express-3.0.0.bin
```

For M4 boards, which have a 16kB bootloader, you must specify `-offset=0x4000`, for example:

```
bossac -p=/dev/cu.usbmodem14301 -e -w -v -R --offset=0x4000 adafruit-circuitpython-feather_m4_express-3.0.0.bin
```

This will `e`rase the chip, `w`rite the given file, `v`erify the write and `R`eset the board. On Linux or MacOS you may need to run this command with `sudo ./bossac ...`, or add yourself to the **dialout** group first.

```
1. bash
bash ⌘1    bash ⌘2    bash ⌘3
(venv) tannewt@shallan:~/Downloads/bossac-1.7.0 $ ./bossac -e -w -v -R ~/Downloads/a
dafruit-circuitpython-feather_m0_express-0.9.3.bin
Device found on cu.usbmodem1441
Atmel SMART device 0x1001000a found
Erase flash
done in 0.658 seconds

Write 216080 bytes to flash (3377 pages)
[==============================] 100% (3377/3377 pages)
done in 1.371 seconds

Verify 216080 bytes of flash with checksum.
Verify successful
done in 0.305 seconds
CPU reset.
(venv) tannewt@shallan:~/Downloads/bossac-1.7.0 $
```

Updating the bootloader

The UF2 bootloader is relatively new and while we've done a ton of testing, it may contain bugs. Usually these bugs effect reliability rather than fully preventing the bootloader from working. If the bootloader is flaky then you can try updating the bootloader itself to potentially improve reliability.

If you're using MakeCode on a Mac, you need to make sure to upload the bootloader to avoid a serious problem with newer versions of MacOS. See instructions and more details here (https://adafru.it/ECU).

In general, you shouldn't have to update the bootloader! If you do think you're having bootloader related issues, please post in the forums or discord.

Updating the bootloader is as easy as flashing CircuitPython, Arduino or MakeCode. Simply enter the bootloader as above and then drag the *update bootloader uf2* file below. This uf2 contains a program which will unlock the bootloader section, update the bootloader, and re-lock it. It will overwrite your existing code such as CircuitPython or Arduino so make sure everything is backed up!

After the file is copied over, the bootloader will be updated and appear again. The **INFO_UF2.TXT** file should show the newer version number inside.

For example:

```
UF2 Bootloader v2.0.0-adafruit.5 SFHWRO
Model: Metro M0
Board-ID: SAMD21G18A-Metro-v0
```

Lastly, reload your code from Arduino or MakeCode or flash the latest CircuitPython core (https://adafru.it/Em8).

Below are the latest updaters for various boards. The latest versions can always be found here (https://adafru.it/Bmg). Look for the `update-bootloader...` files, not the `bootloader...` files.

https://adafru.it/JcN

https://adafru.it/JcN

https://adafru.it/JcO

https://adafru.it/JcO

https://adafru.it/JcR

https://adafru.it/JcR

https://adafru.it/JcU

https://adafru.it/JcU

https://adafru.it/JcX

https://adafru.it/JcX

https://adafru.it/Jc-

https://adafru.it/Jc-

https://adafru.it/Jd2

https://adafru.it/Jd2

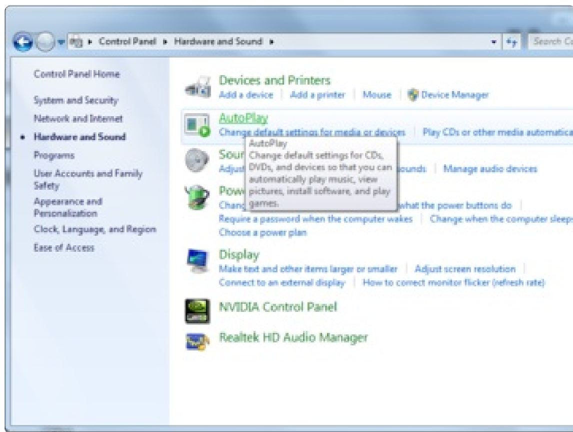
https://adafru.it/Bmg

https://adafru.it/Bmg

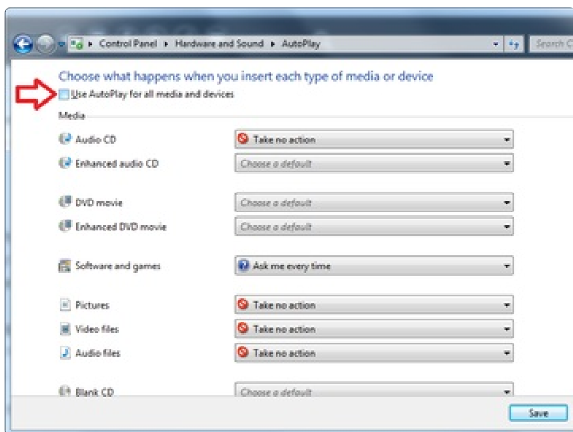
Getting Rid of Windows Pop-ups

If you do a *lot* of development on Windows with the UF2 bootloader, you may get annoyed by the constant "Hey you inserted a drive what do you want to do" pop-ups.

Go to the Control Panel. Click on the **Hardware and Sound** header



Click on the **Autoplay** header



Uncheck the box at the top, labeled **Use Autoplay for all devices**

Making your own UF2

Making your own UF2 is easy! All you need is a .bin file of a program you wish to flash and [the Python conversion script](https://adafru.it/vZb) (https://adafru.it/vZb). Make sure that your program was compiled to start at 0x2000 (8k) for M0 boards or 0x4000 (16kB) for M4 boards. The bootloader takes up the first 8kB (M0) or 16kB (M4). CircuitPython's [linker script](https://adafru.it/CXh) (https://adafru.it/CXh) is an example on how to do that.

Once you have a .bin file, you simply need to run the Python conversion script over it. Here is an example from the directory with **uf2conv.py**. This command will produce a **firmware.uf2** file in the same directory as the source **firmware.bin**. The uf2 can then be flashed in the same way as above.

```
# For programs with 0x2000 offset (default)
uf2conv.py -c -o build-circuitplayground_express/firmware.uf2 build-
circuitplayground_express/firmware.bin

# For programs needing 0x4000 offset (M4 boards)
uf2conv.py -c -b 0x4000 -o build-metro_m4_express/firmware.uf2 build-
metro_M4_express/firmware.bin
```

Installing the bootloader on a fresh/bricked board

If you somehow damaged your bootloader or maybe you have a new board, you can use a JLink to re-install it.

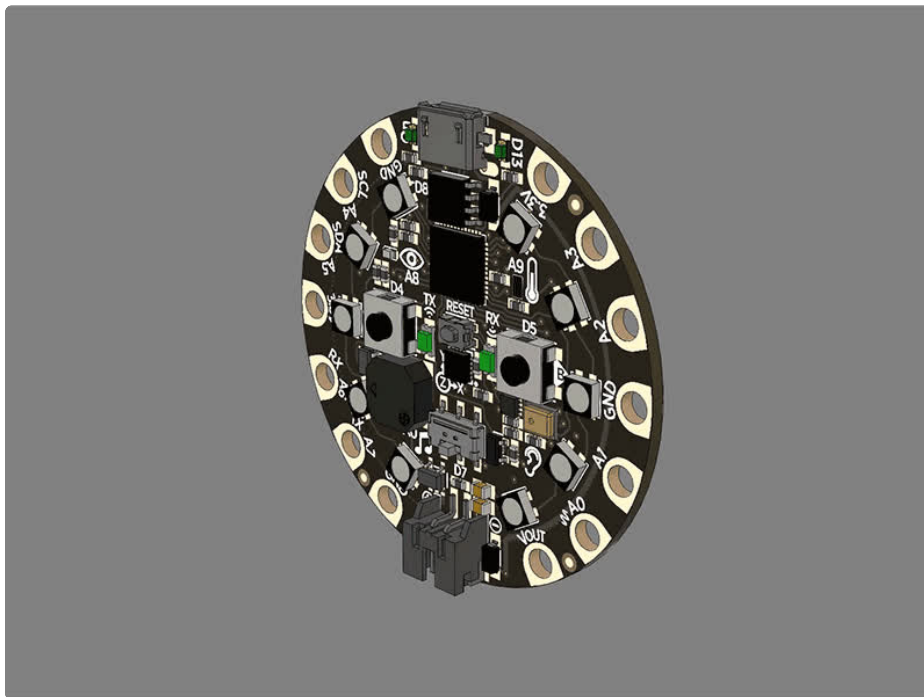
Here's a Learn Guide explaining how to fix the bootloader on a variety of boards using Atmel Studio (https://adafru.it/F5f)

Here's a short writeup by turbinenreiter on how to do it for the Feather M4 (but adaptable to other boards) (https://adafru.it/ven)

Downloads

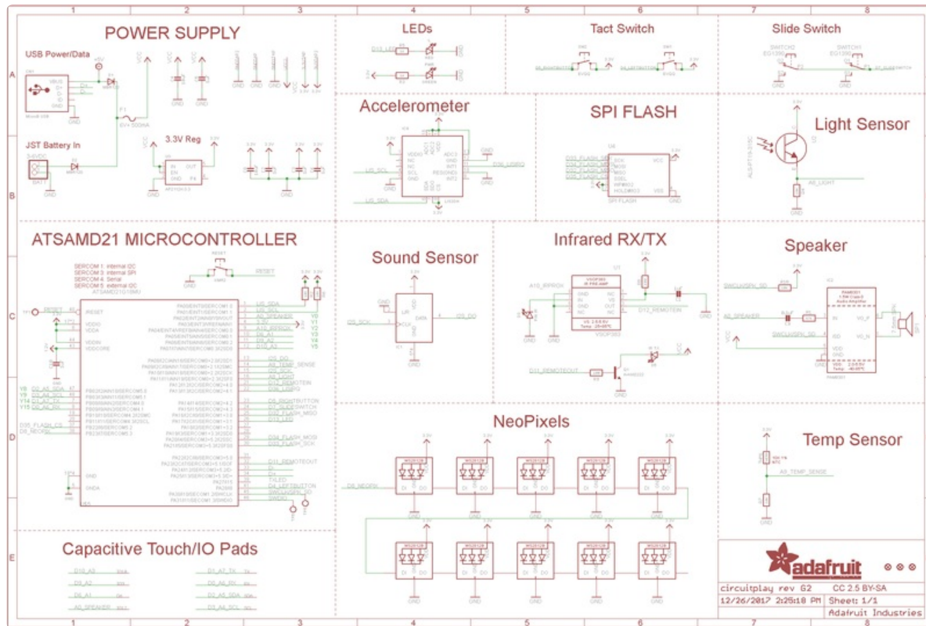
Files:

- EagleCAD PCB files for Circuit Playground Express (https://adafru.it/Bf9)
- Fritzing files in Adafruit Fritzing Library (https://adafru.it/aP3)
- PigHixxx Circuit Playground Express pinout diagram (PDF) (https://adafru.it/Bfa)
- Circuit Playground Express Arduino Demo .uf2 (https://adafru.it/Bfb) (copy this to CPLAYBOOT)
- Circuit Playground Express CircuitPython Demo zip (https://adafru.it/Bfc) (updated for compatibility with CircuitPython 3.x libraries; unzip, and copy contents to CIRCUITPY)
- 3D Models on GitHub (https://adafru.it/GAC)



Circuit Playground Express Schematic

click to embiggen



https://adafru.it/AmO

https://adafru.it/AmO