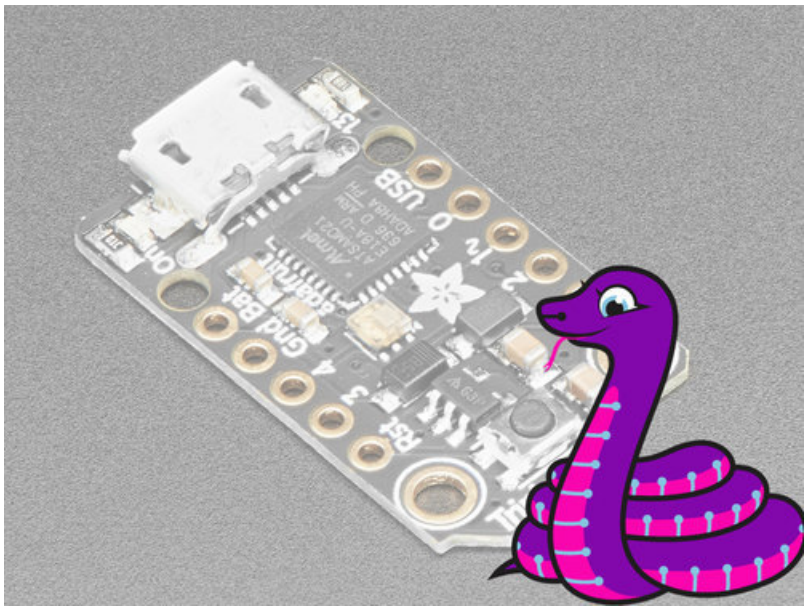
adafruit learning system

# Adafruit Trinket M0

Created by lady ada

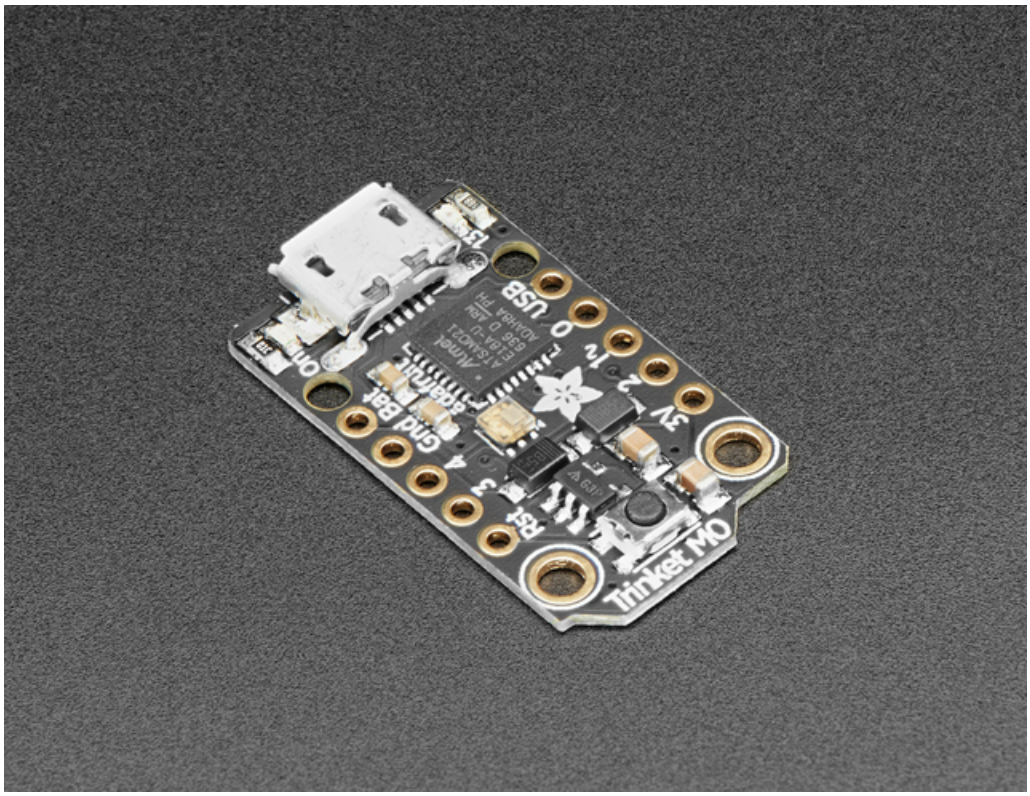


Last updated on 2017-12-25 12:11:41 AM UTC

## Guide Contents

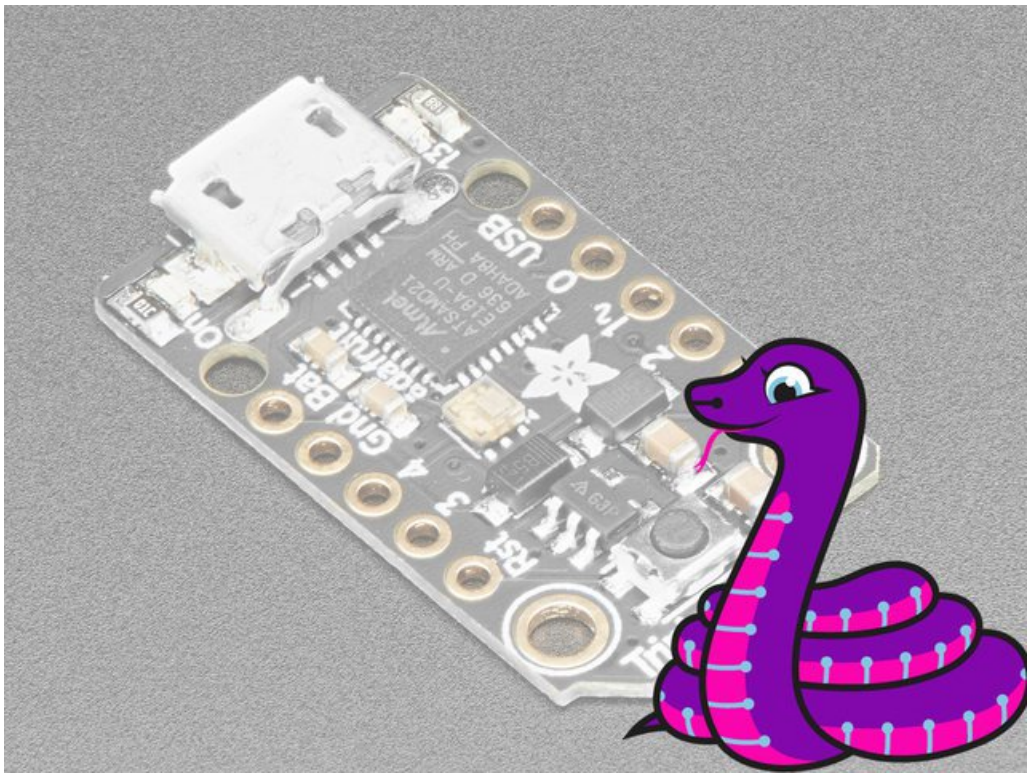
## Overview

The Adafruit Trinket M0 may be small, but do not be fooled by its size! It's a tiny microcontroller board, built around the Atmel ATSAMD21, a little chip with *a lot* of power. We wanted to design a microcontroller board that was small enough to fit into any project, and low cost enough to use without hesitation. Perfect for when you don't want to give up your expensive dev-board and you aren't willing to take apart the project you worked so hard to design. It's our lowest-cost CircuitPython programmable board!




We've taken the same form factor we used for the original ATtiny85-based Trinket and gave it an upgrade. The Trinket M0 has swapped out the lightweight ATtiny85 for a ATSAMD21E18 powerhouse. It's just as small, and it's easier to use, so you can do more.

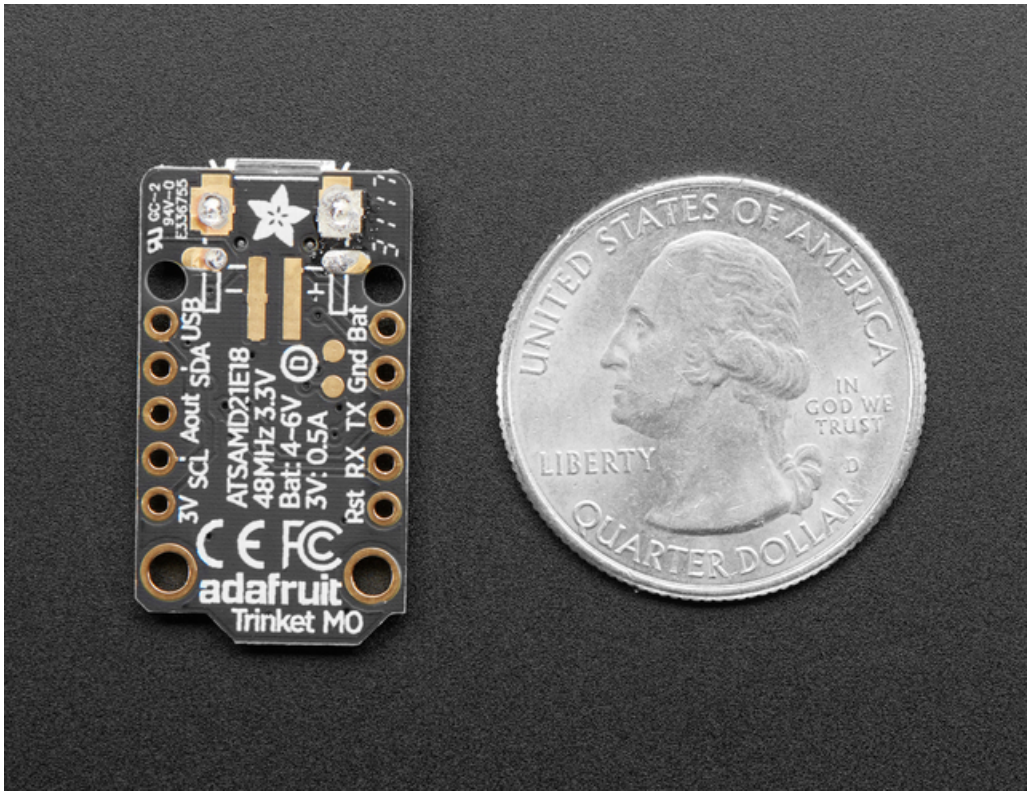
The most exciting part of the Trinket M0 is that while you can use it with the Arduino IDE, we are shipping it with CircuitPython on board. When you plug it in, it will show up as a very small disk drive with main.py on it.
Edit main.py with your favorite text editor to build your project using Python, the most popular programming language. No installs, IDE or compiler needed, so you can use it on any computer, even ChromeBooks or computers you can't install software on. When you're done, unplug the Trinket M0 and your code will go with you.




Here are some of the updates you can look forward to when using Trinket M0:

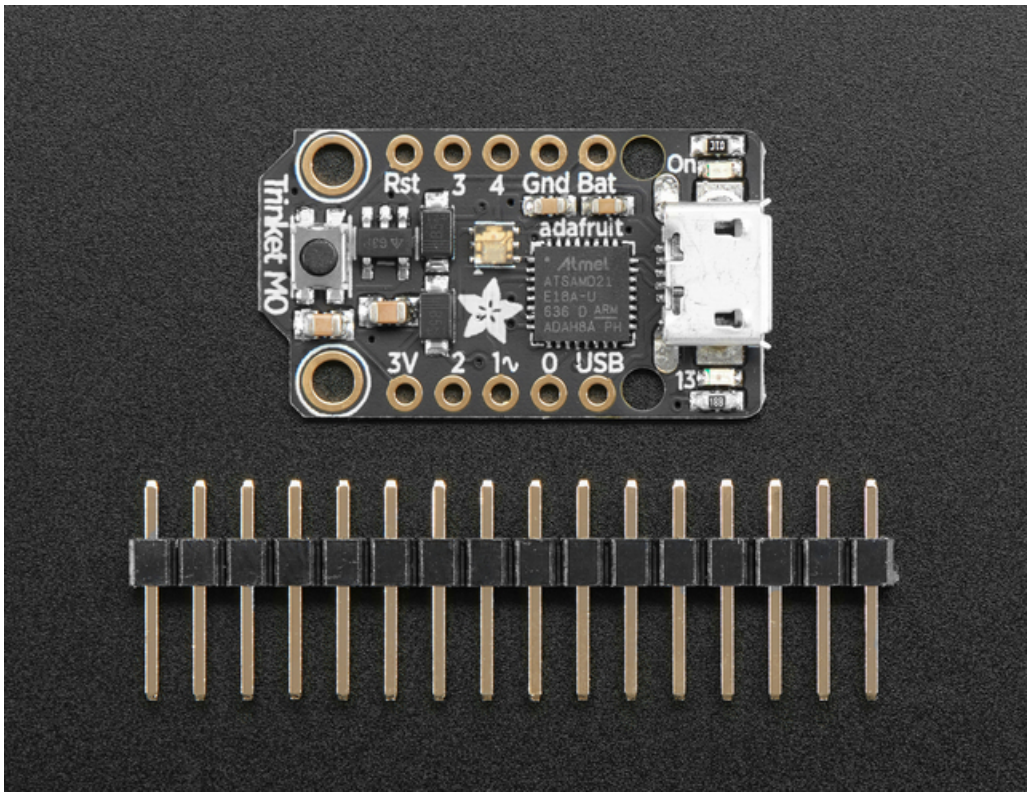
- Same size, form-factor, and pinout as classic Trinket
- Updating ATtiny85 8-bit AVR for ATSAMD21E18 32-bit Cortex M0+
- 256KB Flash - 32x as much as 8 KB on ATtiny85
- 32 KB RAM - 64x as much as 512 bytes on ATtiny85
- 48 MHz 32 bit processor - 6x as fast as ATtiny85 (not even taking into account 32-bit speedups)
- Native USB supported by every OS - can be used in Arduino or CircuitPython as USB serial console, Keyboard/Mouse HID, even a little disk drive for storing Python scripts. (ATtiny85 does not have native USB)
- Can be used with Arduino IDE or CircuitPython
- Built in green ON LED
- Built in red pin #13 LED
- Built in RGB DotStar LED
- All 5 GPIO pins are available and are not shared with USB - so you can use them for whatever you like!
    - Five GPIO pins with digital input/output with internally connected pullups or pulldowns
    - Three of the I/O pins can be used for 12-bit analog input
    - True analog output on one I/O pin - can be used to play 10-bit quality audio clips
    - We gave the M0 pads the exact same names as the original Trinket so all your existing Arduino code will work exactly the same as-is without changes
    - Two high speed PWM outputs - for servos, LEDs, etc
    - Three pins can also be used as hardware capacitive touch sensors with no additional components required
    - Can drive NeoPixels or DotStars on any pins, with enough memory to drive 8000+ pixels. DMA-NeoPixel support on one pin so you can drive pixels without having to spend any processor time on it.
    - Native hardware SPI, I2C and Serial available on two pads so you can connect to any I2C or Serial device with true hardware support (no annoying bit-banging). You can have either one SPI device or both I2C and Serial.
- Same Reset switch for starting your project code over

- Power with either USB or external output (such as a battery) - it'll automatically switch over
- Mounting holes! Yeah!
- Really really small

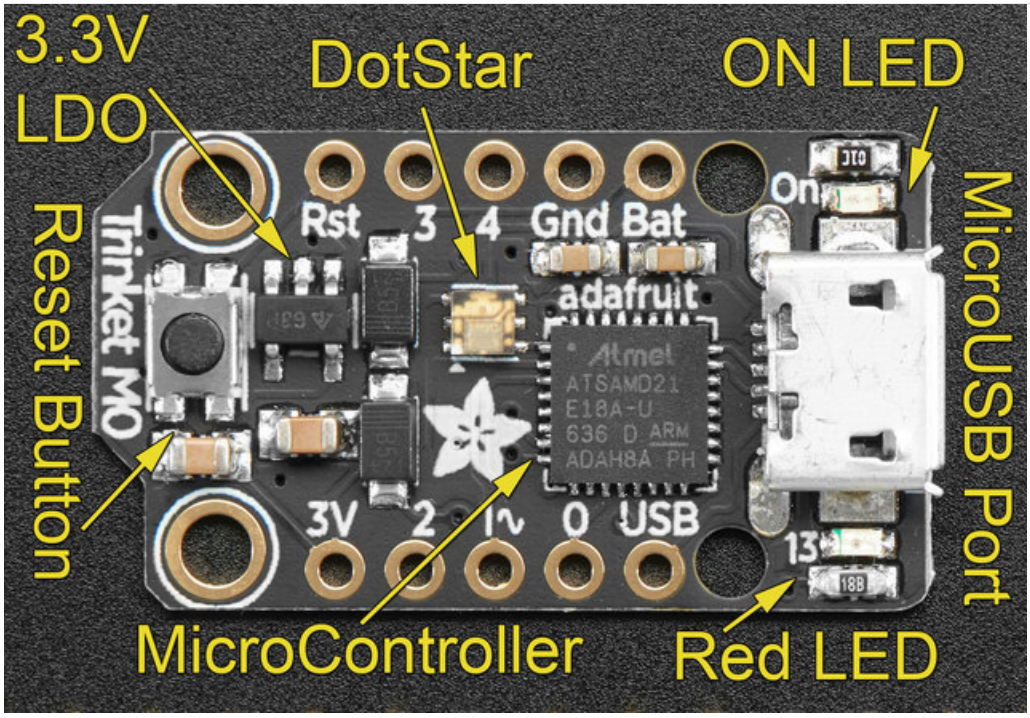



Each order comes with one fully assembled and tested Trinket M0 with CircuitPython & example code programmed in.

So what are you waiting for? Pick up a Trinket M0 today and be amazed at how easy and fast it is to get started with Trinket and CircuitPython!

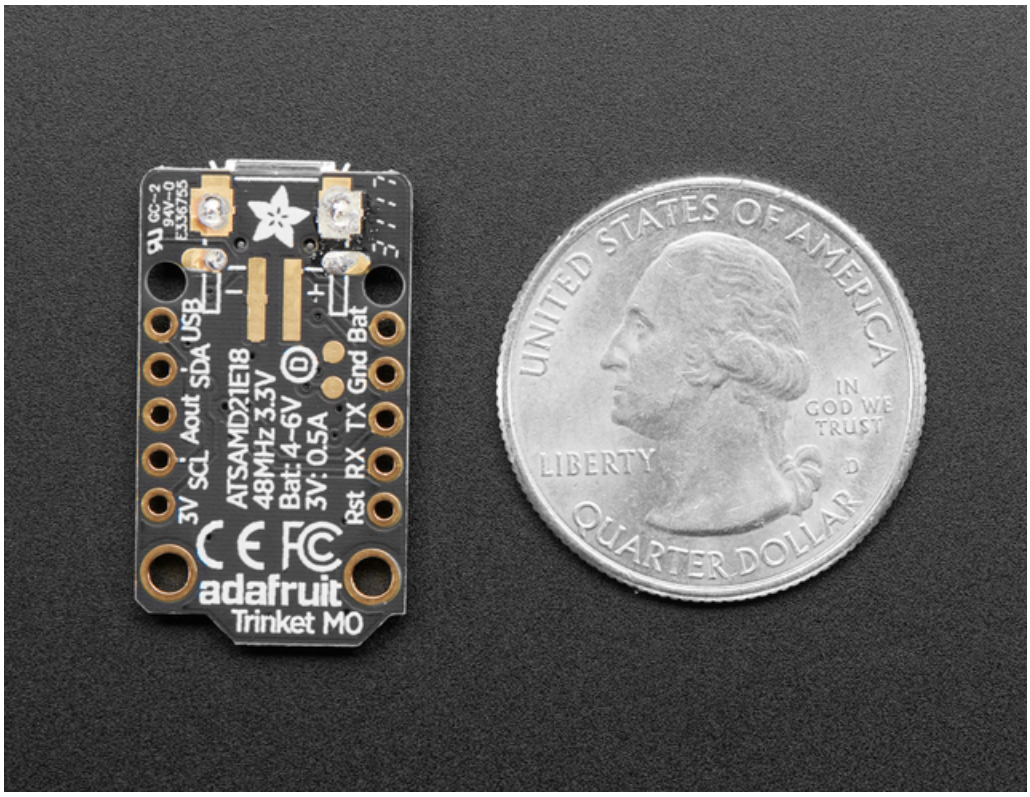

Trinket M0
Rst
3
4
Gnd
Bat
On
adafruit
3V
2
1
0
USB
13

## Guided Tour




Let me take you on a tour of your Trinket M0! Each Trinket M0 is assembled here at Adafruit and comes chock-full of good design to make it a joy to use.

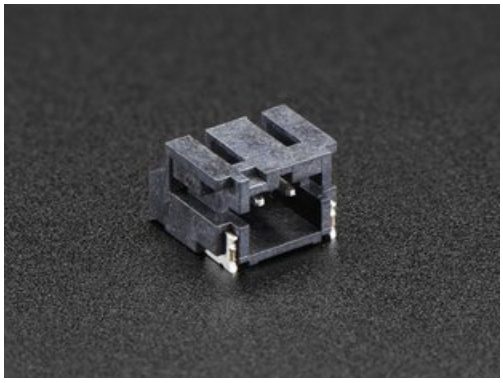
- **Micro B USB connector** - We went with the tried and true micro-B USB connector for power and/or USB communication (bootloader, serial, HID, etc). Use with any computer with a standard data/sync cable.
- **RGB DotStar LED** - Instead of an always-on green LED we provide a full RGB LED. You can set it to any color in the rainbow. It will also help you know when the bootloader is running (it will turn green) or if it failed to initialize USB when connected to a computer (it will turn red). By default after you boot up the Trinket M0 it will turn a lovely violet color.
- **Red #13 LED** - this LED does double duty. Its connected with a series resistor to the digital #13 GPIO pin. It pulses nicely when the Trinket is in bootloader mode, and its also handy for when you want an indicator LED.
- **ON LED** - this LED lets you know when the Trinket is powered up, it will shine green whenever the 3.3V regulator is working.
- **10 Header Pins** - Check the next page for the pinout details but these have all the power and analog/digital IO you need! Solder 0.1" headers or wires directly
- **Reset Button** - an onboard reset button will launch the bootloader when pressed and the Trinket is plugged into a computer. If it is not connected to a computer, it's smart enough to go straight to the program.




## On the Back

**JST Battery Input** - you can optionally solder on a a JST PH connector on the back so you can take your Trinket anywhere and power it from an external battery. The connector is also tied to the **BAT** pin on the headers but it can be nice to just plug in a cable

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 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)



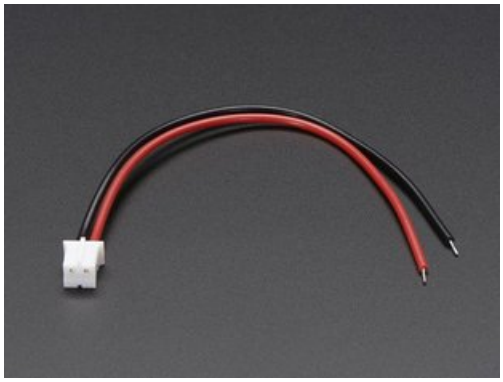
JST-PH 2-Pin SMT Right Angle Connector

PRODUCT ID: 1769

https://adafru.it/e2T

$0.75

IN STOCK



JST 2-pin cable

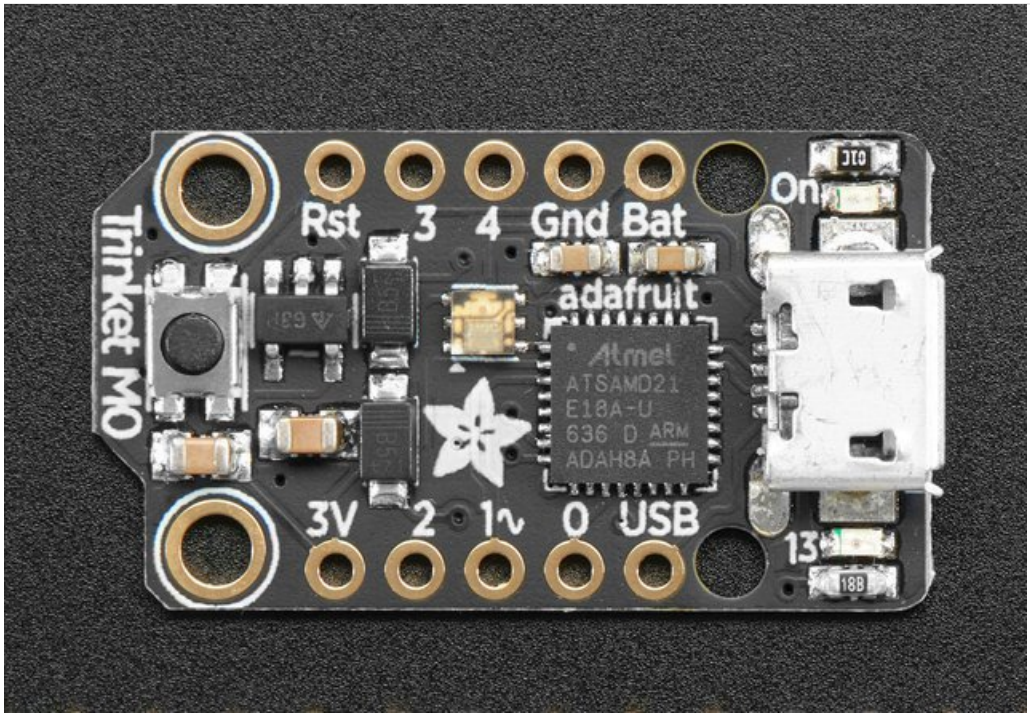
PRODUCT ID: 261

https://adafru.it/drM

$0.75

IN STOCK

## Pinouts




## Power Pins

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

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

## Input/Output Pins

Next we will cover the 5 GPIO (General Purpose Input Ouput) pins! For reference you may want to also check out the datasheet-reference in the downloads section for the core ATSAMD21E18 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.** All pads can also be used as hardware interrupts 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

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

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

### Unique pad capabilities

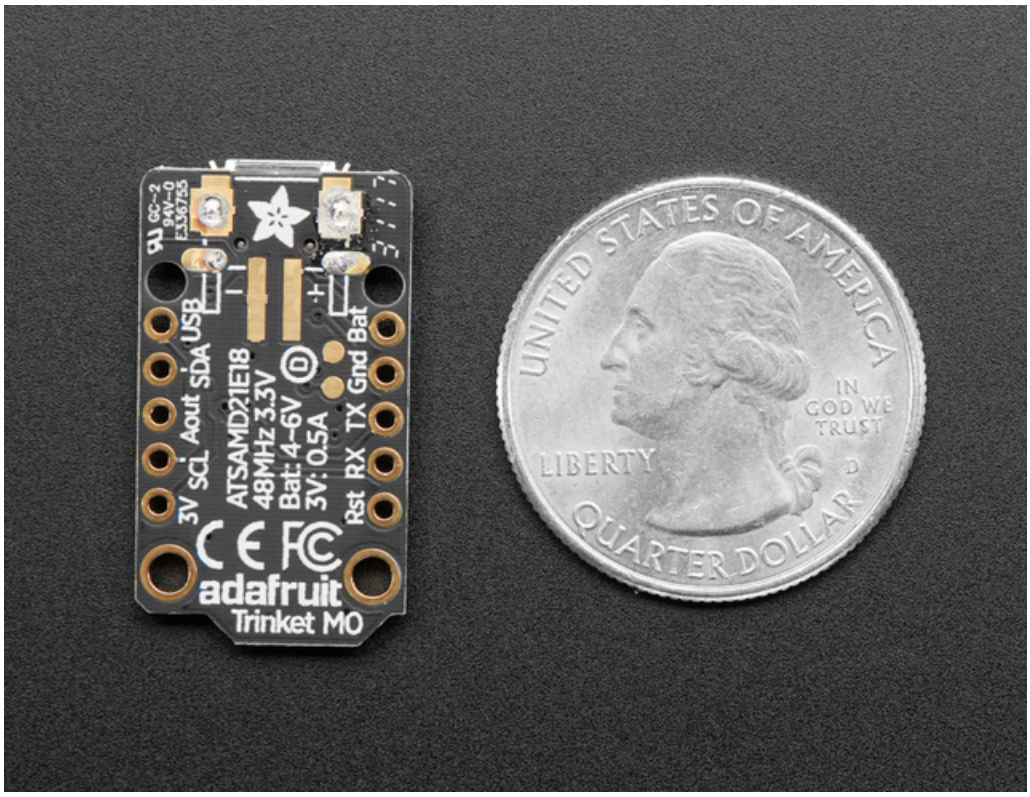
- **Digital #0 / A2** - this is connected to **PA08** on the ATSAMD21. This pin can be used as a digital I/O with selectable pullup or pulldown, analog input (use 'A2'), PWM output, and is also used for I2C data (SDA)
- **Digital #1 / A0** - this is connected to **PA02** on the ATSAMD21. This pin can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

Other Pads!

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

## Secret SWD Pads

On the bottom of the Trinket M0 you will see two small pads. These are used for our programming/test but you can use them too.




Starting from the pad closest to the microUSB connector:

- SWCLK
- SWDIO

On the off chance you want to reprogram your Trinket M0 or debug it using a Cortex M0 SWD debug/programmer, you will need to solder/connect to these pads. We use them for testing and you will likely never need it but they are there if you do!

## Windows Driver Installation

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

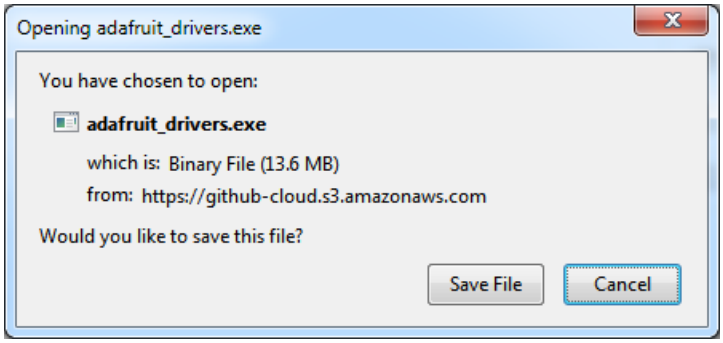
Before you plug in your board, you'll need to possibly install a driver!

Click below to download our Driver Installer.

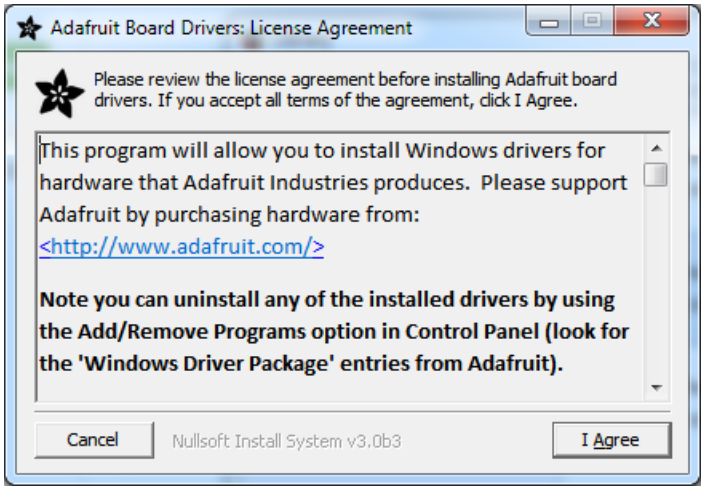
Download Latest Adafruit Windows Driver Installer

https://adafru.it/A0N

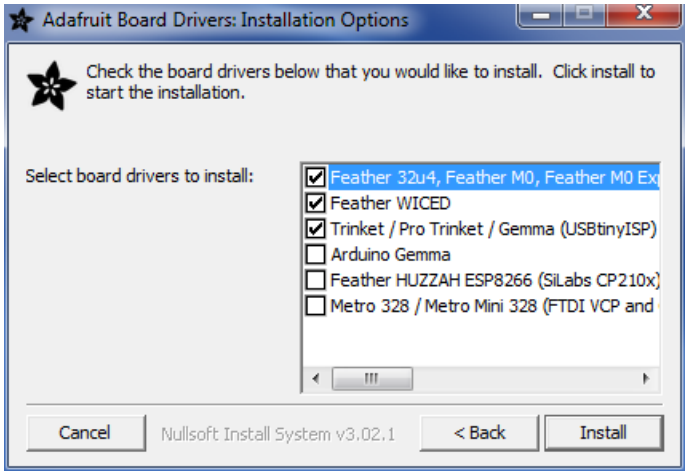
Download and run the installer.




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!



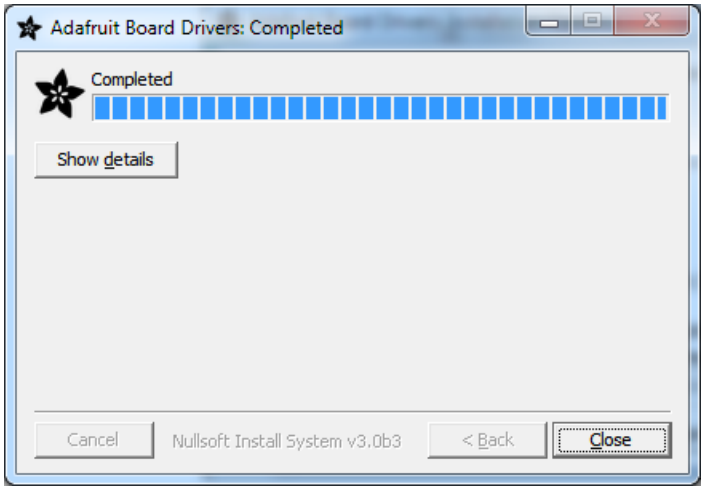

On Windows 7, by default, we install a single driver for most of Adafruit's boards, including the **Feather 32u4, the Feather M0, Feather M0, Express, Circuit Playground, Circuit Playground Express, Gemma M0, Trinket M0, Metro M0 Express**. On Windows 10 that driver is not necessary (it's built in to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

Adafruit Windows Drivers source (v2.0.0.0)

https://adafru.it/zel

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

## 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

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

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

If you've already plugged your board into your computer, you should see a drive called **CIRCUITPY**. The drive will contain a few files. If you want to make a 'backup' of the current firmware on the device, drag-off and save the **CURRENT.UF2** file. Other that that you can ignore the index.htm and info_uf2.txt files. They cannot be deleted and are only for informational purposes.

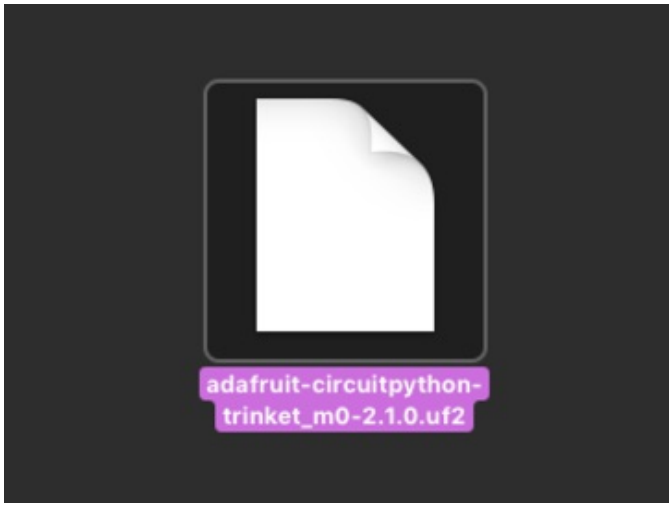
**If you have already plugged in your board, start by ejecting or "safely remove" the CIRCUITPY drive**. This is a good practice to get into. Always eject before unplugging or resetting your board!

### Set up CircuitPython Quick Start!

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

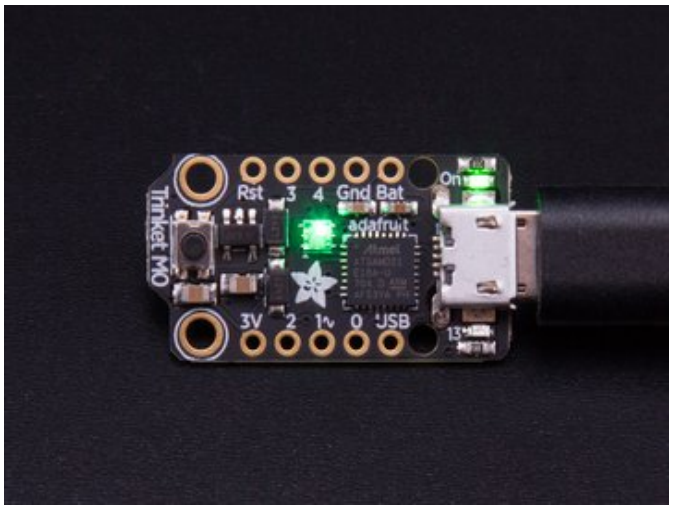
Download the latest Trinket CircuitPython UF2

https://adafru.it/Alb




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

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

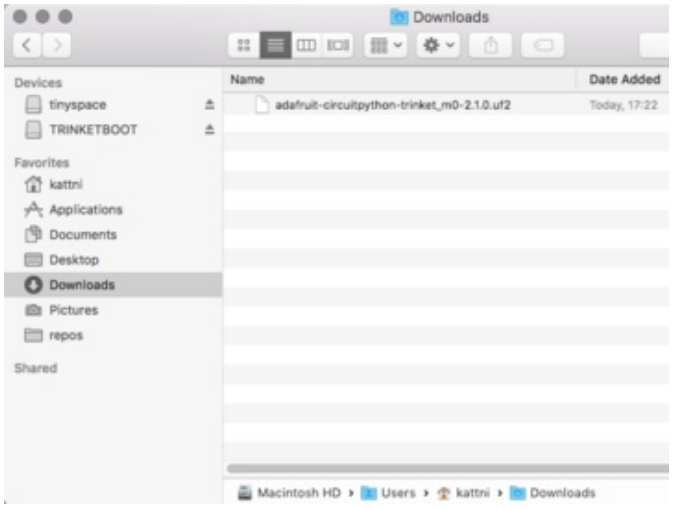



Plug your Trinket into your computer using a known-good USB cable.

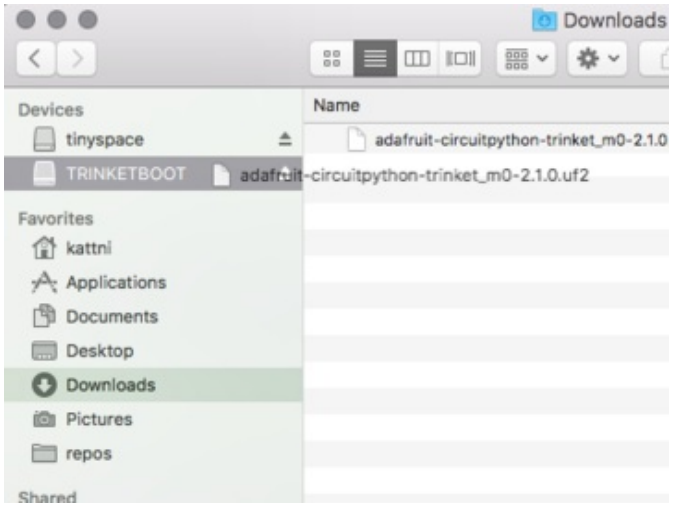
**A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.**

Double-click the small **Reset** button next to the Trinket M0 name printed on your board, and you will see the LED turn green. If it turns red, check the USB cable, try another USB port, etc.

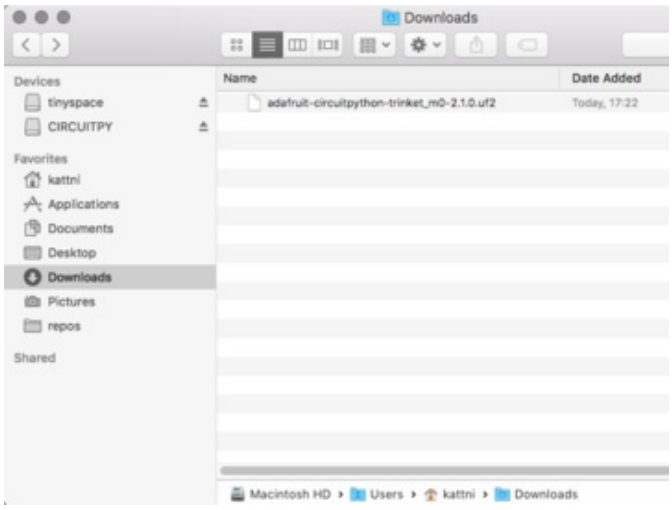
If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!




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




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




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

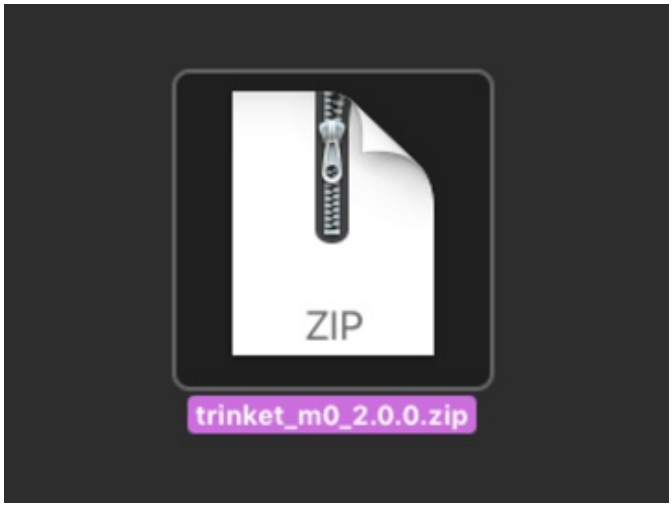
That's it, you're done! :)

## Trinket Default Zip Install

Trinket M0 is limited on space. As you begin working on projects, you may run out of space. Operating systems can create hidden files that take up space. To prevent these files from being added to your Trinket, we suggest installing the Trinket Default Zip.

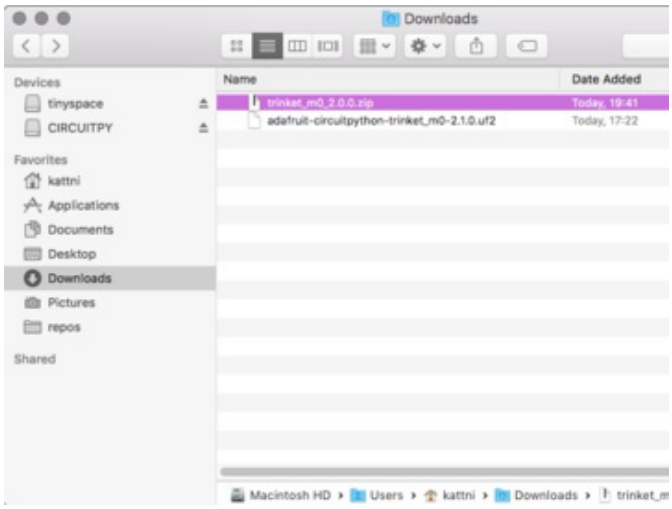
Download the Trinket default zip

https://adafru.it/zdF



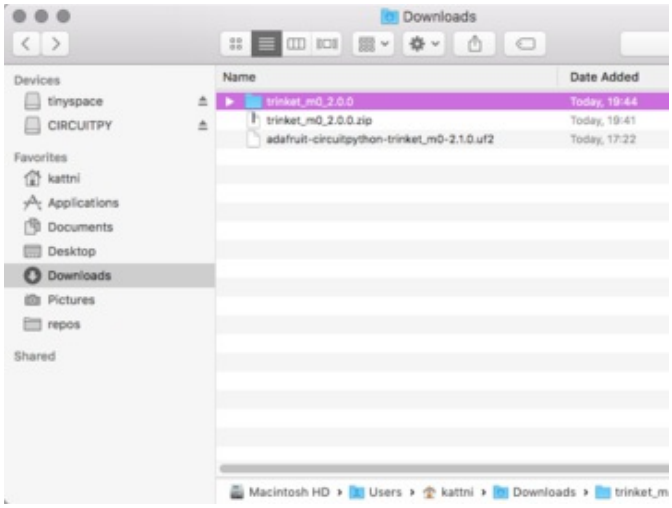

**Click the link above to download the default zip.**

Download and save it to your desktop, or wherever is handy!

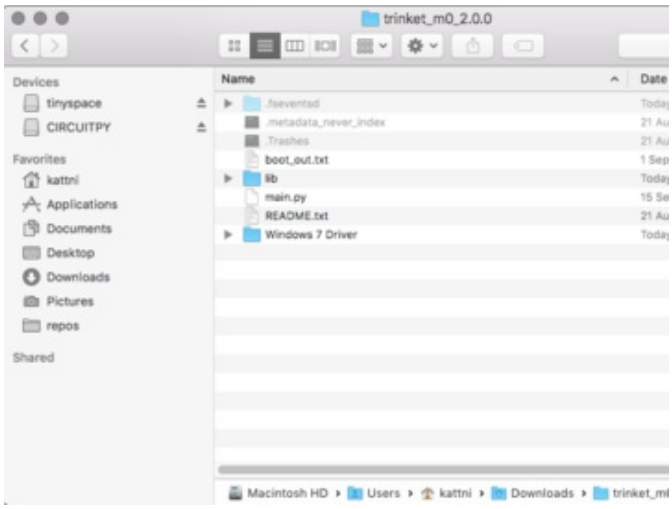



If you haven't already, **plug your Trinket into your computer using a known-good USB cable.**

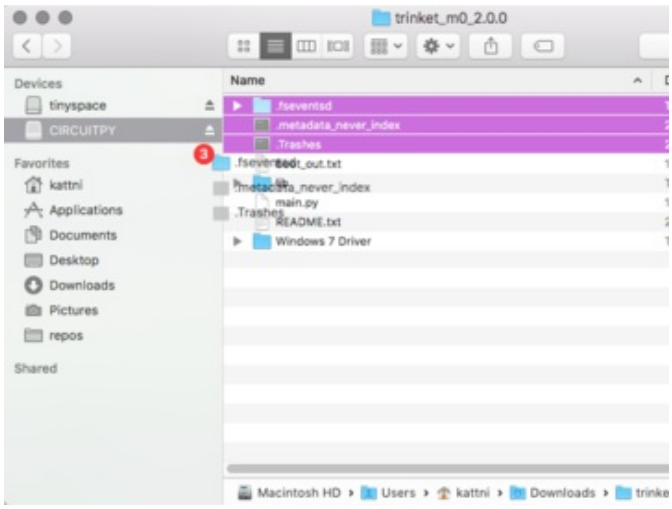
Make sure your **CIRCUITPY** drive appears.




Once downloaded, **double-click the file to extract the contents.**




Double click newly extracted folder to open it.



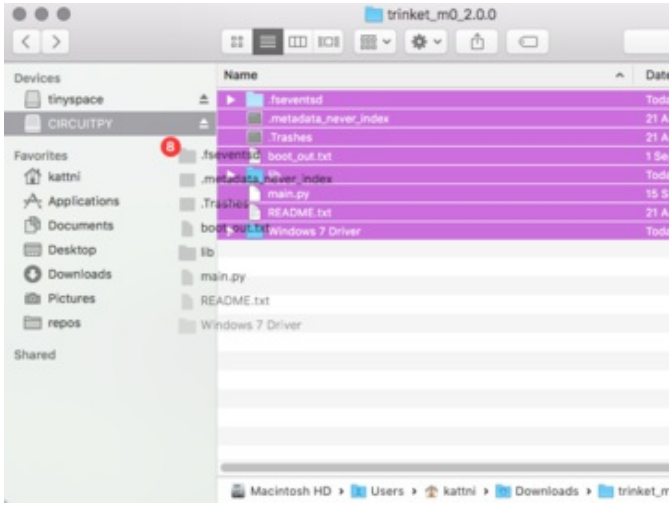

To load the files that will keep the system from adding hidden files to your drive, highlight these three files:

**.fseventsd**

**.metadata_never_index**

**.Trashes**

Drag them to your **CIRCUITPY** drive. If it asks to replace any, say yes!




If you'd like to **reset your Trinket to the same files it shipped with**, you can do that with these files. **If you changed main.py, and you want to keep your changes, back up main.py first.**

Highlight all the files in this folder. Drag them all to your CIRCUITPY drive.

If it asks to replace anything, say yes.

## 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!)

### Installing Mu for Windows or Mac OS X

To install Mu for Windows, follow these steps:

Download the latest Mu for Windows

https://adafru.it/Amb

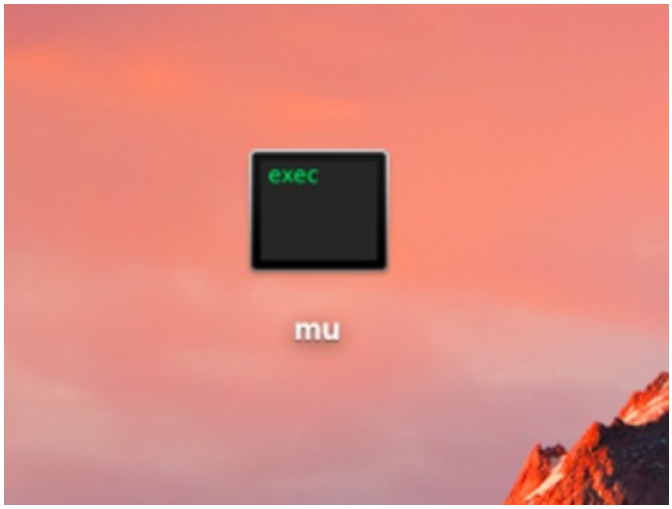
Download the latest Mu for Mac OS X

https://adafru.it/Amc


Click the link above to download the latest versionof Mu.

Download and save the file to your desktop or wherever is handy.

Double-click the file to open Mu. You're ready to go!



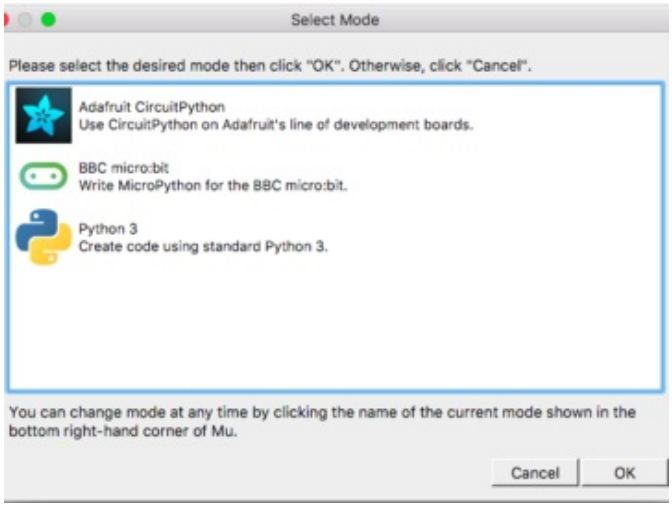

## Installing Mu for Linux

Each Linux distro is a little different, so use this as a guideline!

1. Mu require python version 3. If you haven't installed python yet, do so via your command line using something like `sudo apt-get install python3`
2. You'll also need pip3 (or pip if you only have python3 installed) - try running `pip3 --version`. If that didn't work, you ran `sudo apt-get install python3-pip`
3. Finally, run `pip3 install mu_editor`
4. You can now run `mu` directly from the command line

First you'll want to make sure you have `pip` installed. Open a terminal and type `pip3 --version`.

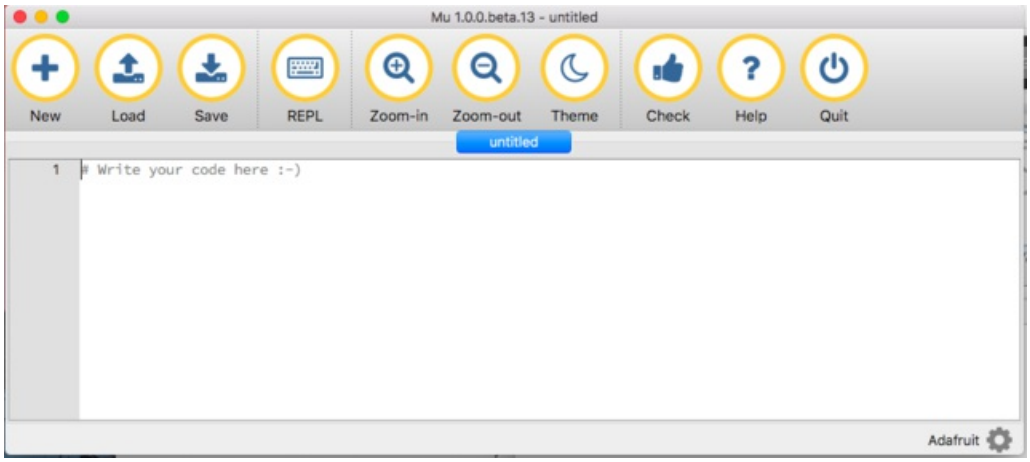
## Using Mu




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


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

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


Mu 1.0.0.beta.13 - untitled
New
Load
Save
REPL
Zoom-in
Zoom-out
Theme
Check
Help
Quit
untitled
1
# Write your code here :-)
Adafruit

## Creating and Editing Code

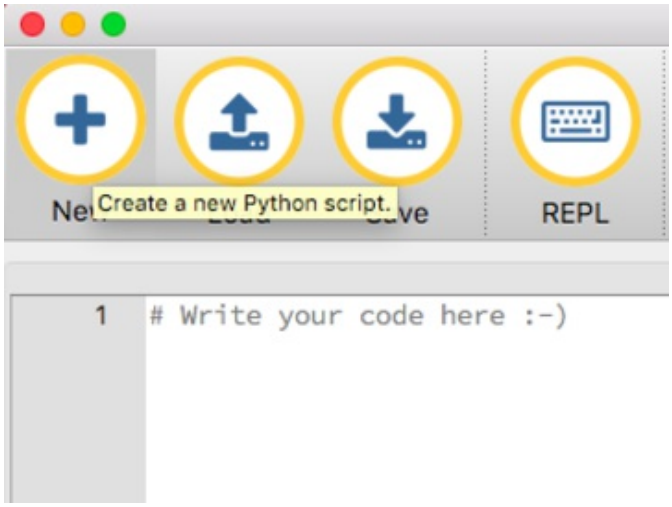
One of the best things about CircuitPython is how simple it is to get code up and running. In this section, we're going to cover how to create and edit your first CircuitPython program.

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. There are also excellent options available for download that are designed for editing code. Atom is a code editor that works on all three operating systems. There are many options for all operating systems.

**Code editors have features that are specific to editing code, but any text editor will be fine.**

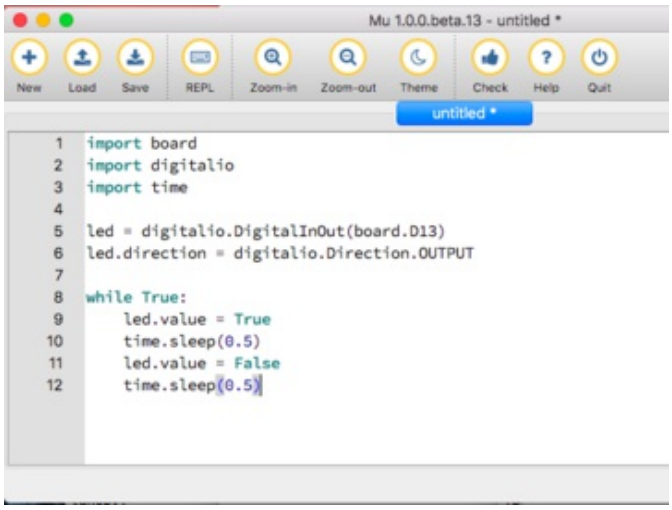
### Creating Code



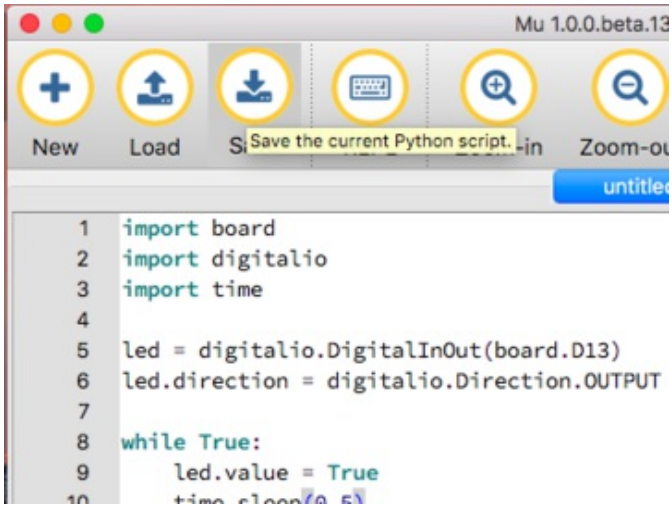

Open your editor, and create a new file. If you are using Mu, click the **New** button in the top left

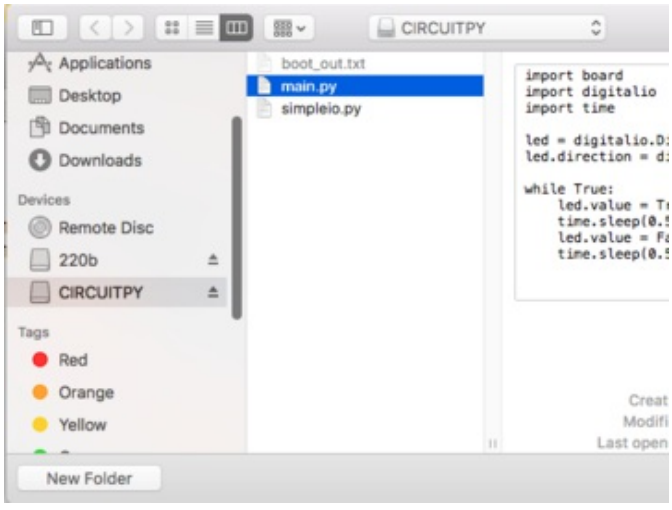
Copy and paste the following code into your editor:

```
    import board
import digitalio
import time

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

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




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



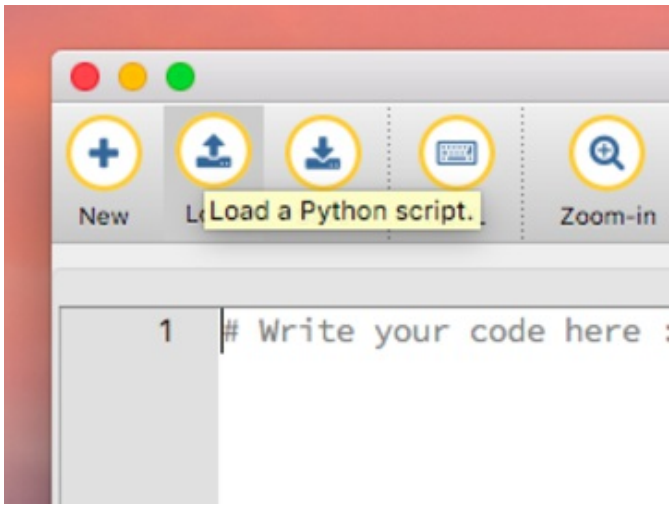




Save this file as **code.py** on your CIRCUITPY drive.

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

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

## Editing Code




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

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

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

There's just one warning we have to give you before we continue...

**Don't Click Reset or Unplug!**

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

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

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

There are a few ways to avoid this:

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

Recommended editors:

- **mu** is an editor that safely writes all changes (it's also our recommended editor!)
- **emacs** is also an editor that will fully write files on save
- **vim** / **vi** safely writes all changes
- **Sublime Text** safely writes all changes
- The **PyCharm IDE** is safe if "Safe Write" is turned on in Settings->System Settings->Synchronization (true by default).
- If you are using **Atom**, install this package so that it will always write out all changes to files on `CIRCUITPY`.

- **Visual Studio Code** appears to safely write all changes
- **gedit** on Linux appears to safely write all changes

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** does not force-write out the file
- **Anything else** - we haven't tested other editors so please use a recommended one!

### 2. Eject or Sync the Drive After Writing

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

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

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

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

## Back to Editing Code...

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

```
import board
import digitalio
import time

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

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

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

## Exploring Your First CircuitPython Program

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

Here is the original code again:

```
import board
import digitalio
import time

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

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

## Imports & Libraries

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

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

The `import` statements tells the board that you're going to use a particular library in your code. In this example, we imported three libraries: `board`, `digitalio`, and `time`. All three of these libraries are built into CircuitPython, so no separate files are needed. That's one of the things that makes this an excellent first example. You don't need 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.D13)
led.direction = digitalio.Direction.OUTPUT
```

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

## Loop-de-loops

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

Inside our loop, we have four items:

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

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

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

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

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

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

## 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.

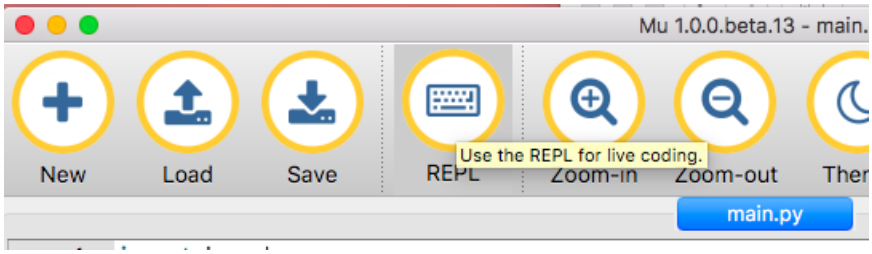
## Are you using Mu?

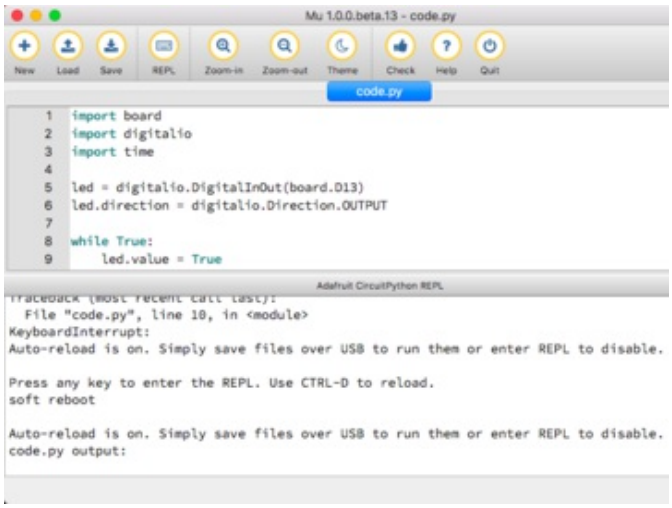
If so, good news! The serial console **is built into Mu** and will **autodetect your board** making using the REPL *really really easy*.

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 **REPL** button in the menu and click it




The editor window will split in half.

The bottom half is your serial output/input. You can see text *from* the CircuitPython board as well as send text *to* the board.

## 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

Mac and Linux both have one built in, though other options are available for download, check this page for more details

## Interacting with the Serial Console

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

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

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

```
import board
import digitalio
import time

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

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

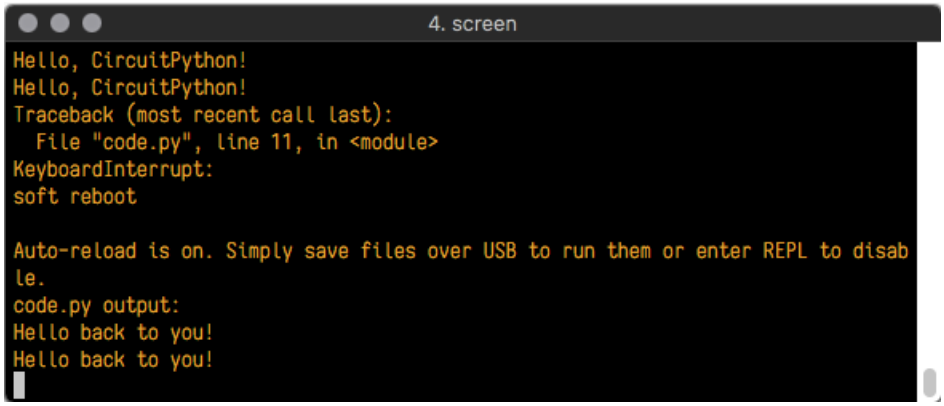
Save your file.

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


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


Keep your serial console window where you can see it. Save your file. You'll see what the serial console displays when

the board reboots. Then you'll see your new change!

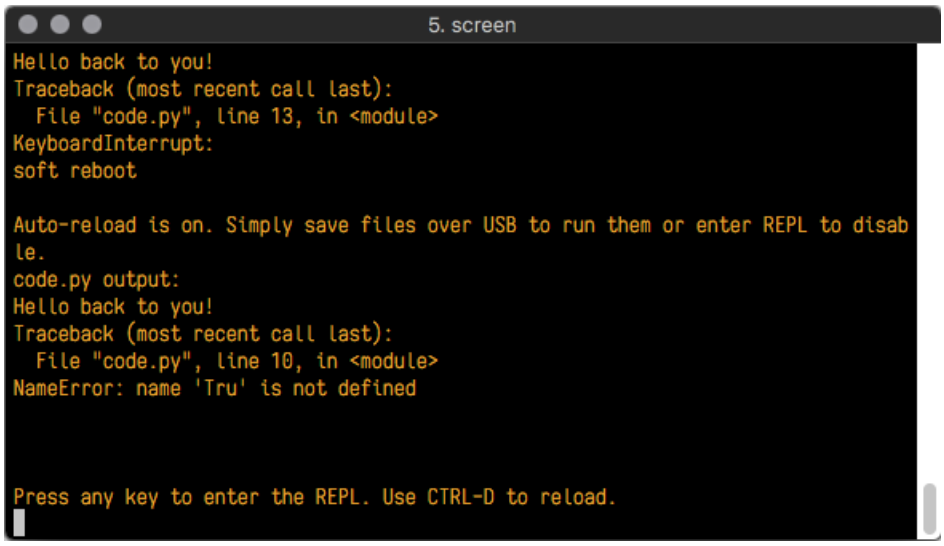



The `Traceback (most recent call last):` is telling you the last thing your board was doing before you saved your file. This is normal behavior and will happen every time the board resets. This is really handy for troubleshooting. Let's introduce an error so we can see how it is used.

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

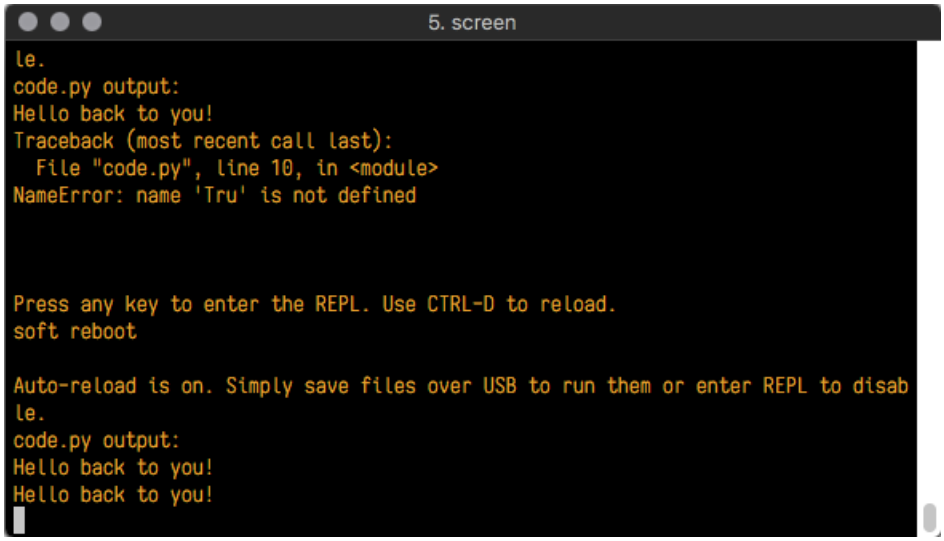

Save your file. You will notice that your red LED will stop blinking, and you may have a colored status LED blinking at you. This is because the code is no longer correct and can no longer run properly. We need to fix it!

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!

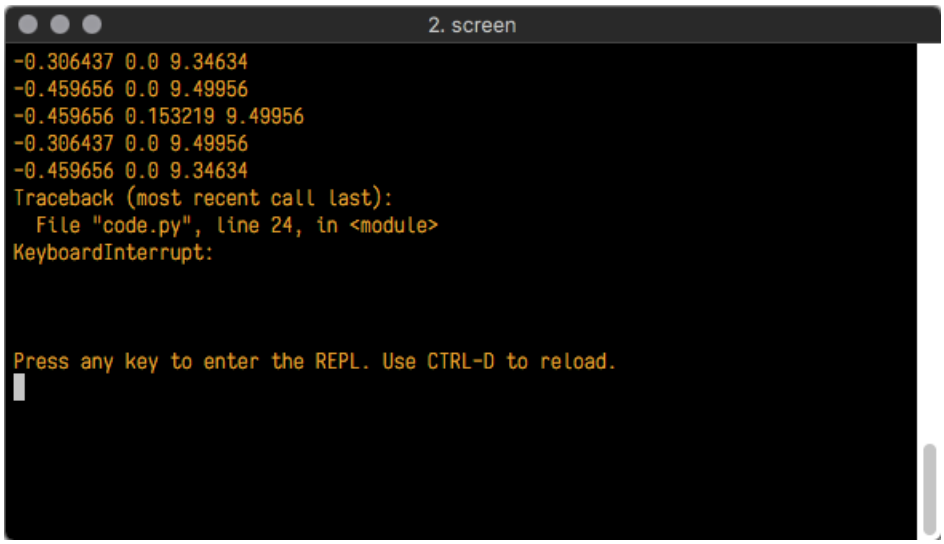
## The REPL

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.

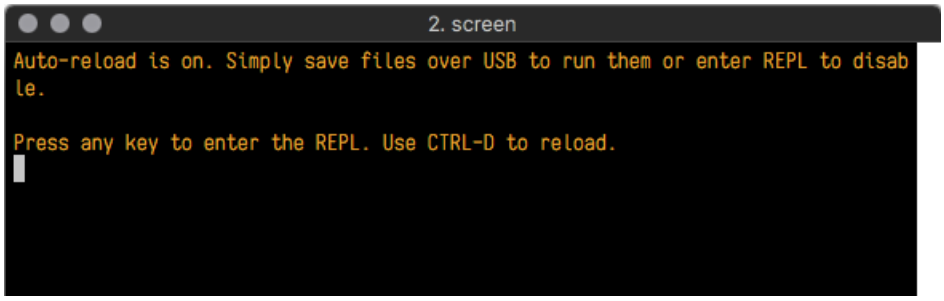
To use the REPL, you first need to be connected to the serial console. Once that connection has been established, you'll want to press **Ctrl** + **C**.

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!


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
```

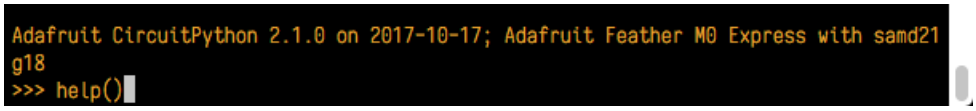
This line tells you the version of CircuitPython you're using and when it was released. Next, it gives you the type of board you're using and the type of microcontroller the board uses. Each part of this may be different for your board depending on the versions you're working with.

This is followed by the CircuitPython prompt.

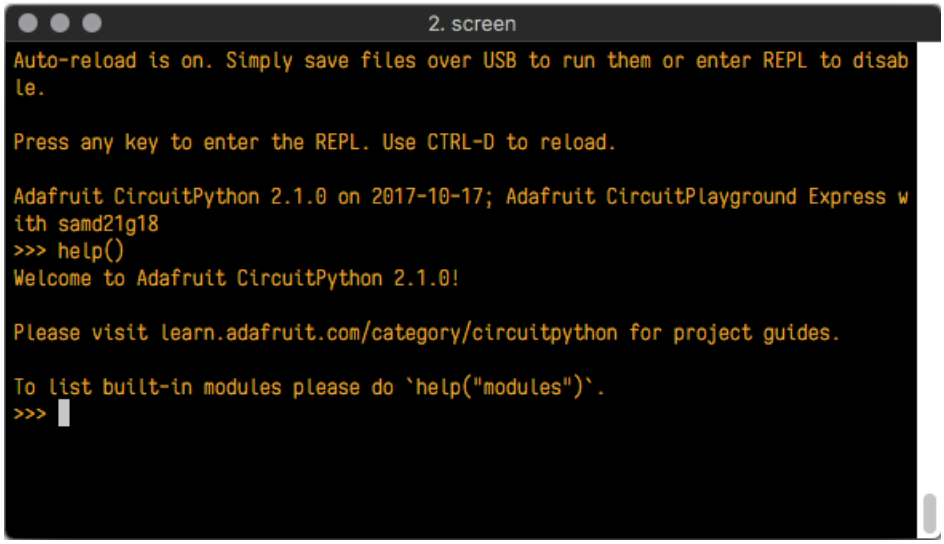
```
>>>
```

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.




Then press enter. You should then see a message.



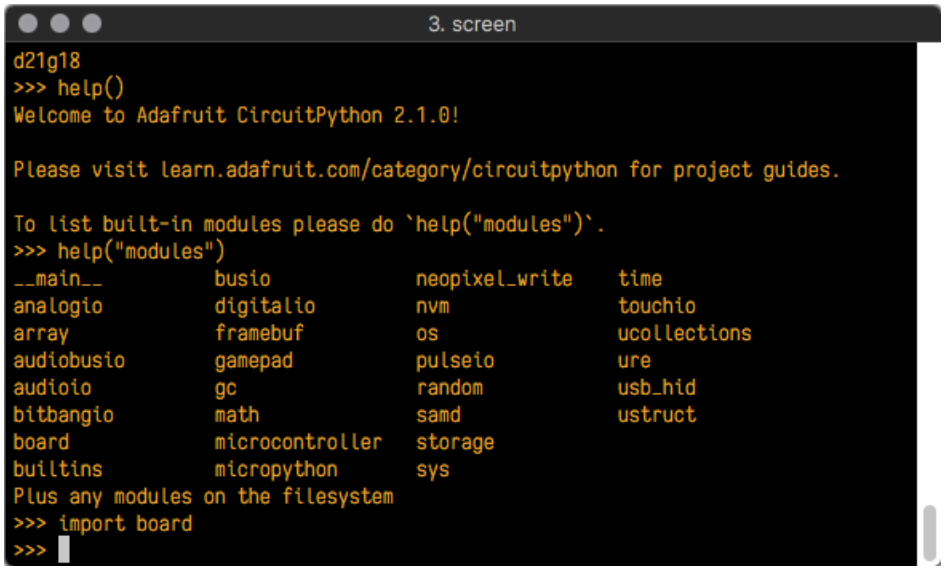

First part of the message is another reference to the version of CircuitPython you're using. Second, a URL for the CircuitPython related project guides. Then... wait. What's this? `To list built-in modules, please do `help("modules")`.` Remember the libraries you learned about while going through creating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

This is a perfect place to start. Let's take a look!

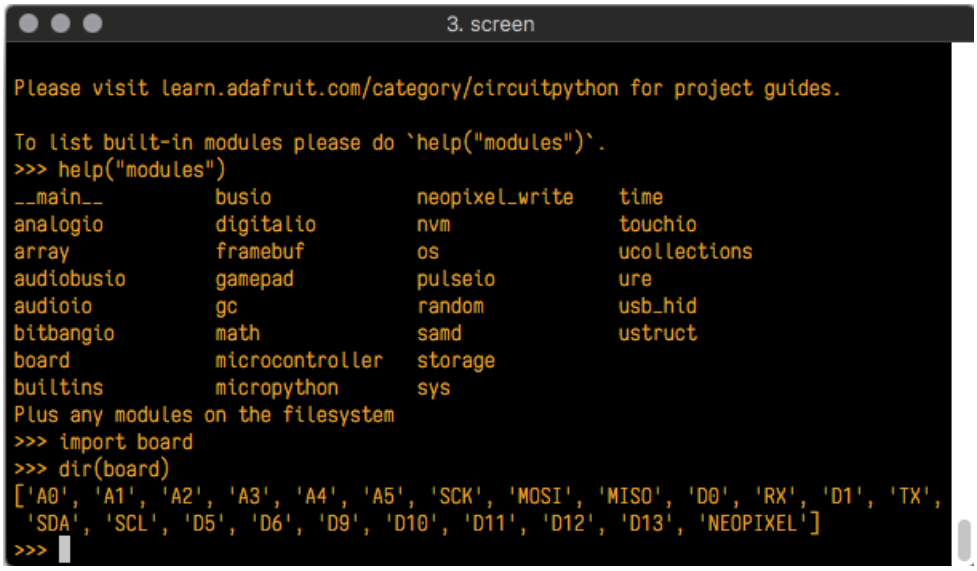
Type `help("modules")` into the REPL next to the prompt, and press enter.


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.




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




This is a list of all of the pins on your board that are available for you to use in your code. Each board's list will differ slightly depending on the number of pins available. Do you see `D13`? That's the pin you used to blink the red LED!

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

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

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

Then press enter.


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

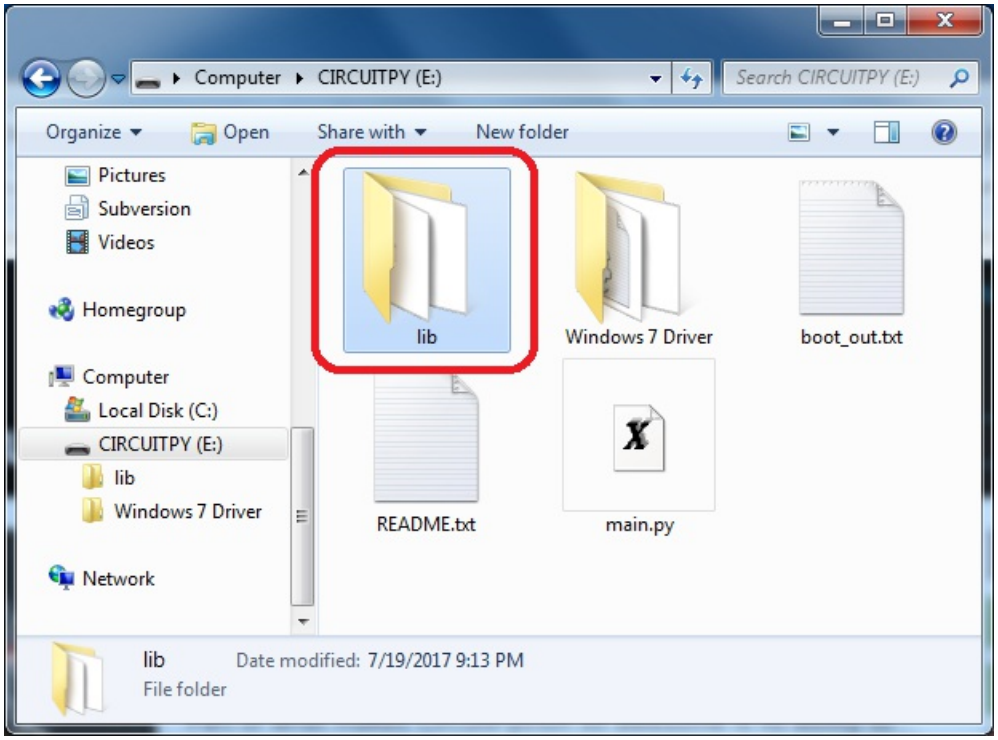
When you're ready to leave the REPL and return to the serial console, simply press **Ctrl + D**. This will reload your board and reenter the serial console. You will restart the program you had running before entering the REPL. In the console window, you'll see any output from the program you had running. 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

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, its in the base directory of the drive. If not, simply create the folder yourself.




CircuitPython libraries work in the same was as regular Python modules so the Python docs 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 2.x Bundle release by clicking this button:

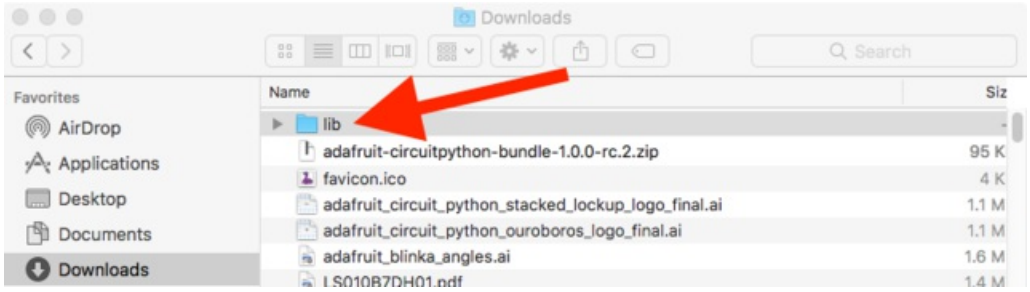
Click for the Latest Adafruit CircuitPython Library Bundle Release

https://adafru.it/AgR

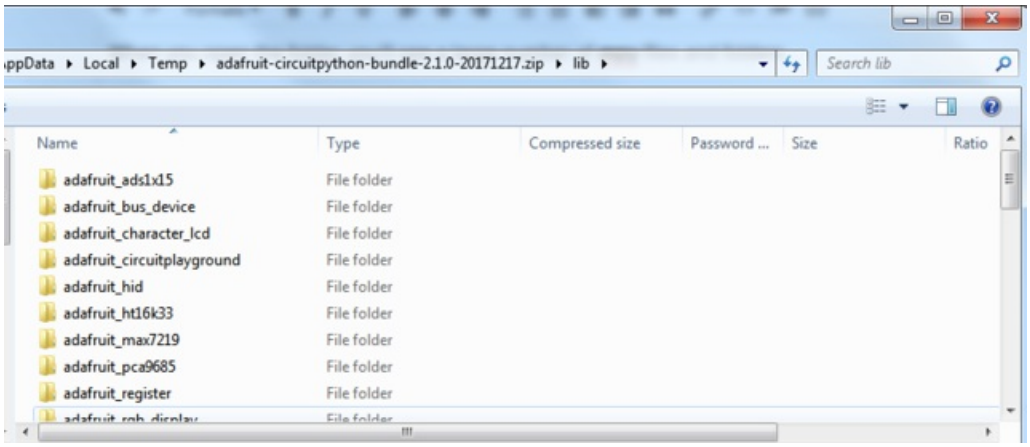
If you need another version, you can also visit the bundle release page which will let you select exactly what version you're looking for, as well as information about changes.

**Either way, download the version that matches your CircuitPython run-time.** For example, if you're running v2.2 download the v2 bundle. If you're running 3.0, download the v3 bundle. There's also a `py` bundle which contains the uncompressed python files, you probably *don't* want that!

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.




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




## Express Boards

If you are using a Feather M0 Express, Metro M0 Express or Circuit Playground Express (or any other "Express" board) your CircuitPython board comes with at least 2 MB of Flash storage. This is *plenty* of space for all of our library files so we recommend you just install them all! (If you have a Gemma M0 or Trinket M0 or other non-Express board, skip down to the next section)

On Express boards, the `lib` directory can be copied directly to the CIRCUITPY drive.

Just drag the entire `lib` folder into the CIRCUITPY drive, and 'replace' any old files if your operating system prompts you

## Non-Express Boards

**If you are using Trinket M0 or Gemma M0, you will need to load the libraries individually, due to file space restrictions.** If you are using a non-express board, or you would rather load libraries as you use them, you'll first 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.

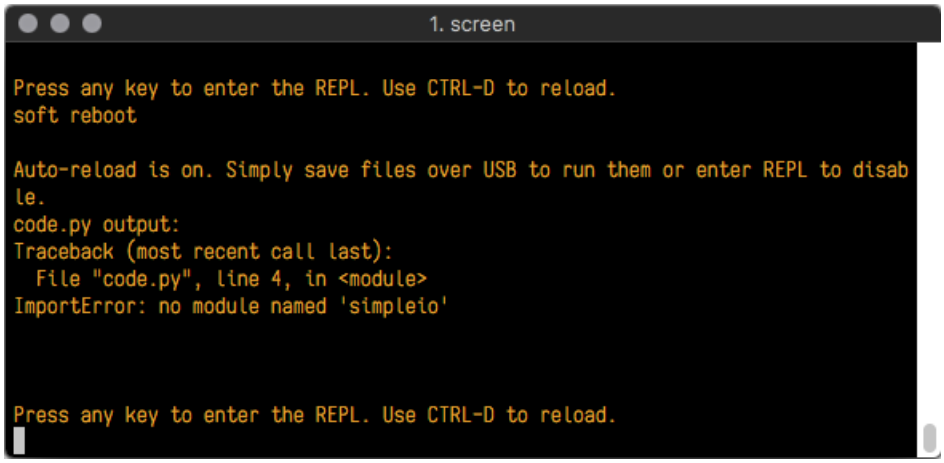
## Example: `ImportError` Due to Missing Library

If you choose to load libraries as you need them, you may write up code that tries to use a library you haven't yet loaded. We're going to demonstrate what happens when you try to utilise a library that you don't have loaded on your board, and cover the steps required to resolve the issue. **This demonstration will only return an error if you do not have the required library loaded into the `lib` folder on your `CIRCUITPY` drive.**

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

```
import board
import time
import simpleio

led = simpleio.DigitalOut(board.D13)

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

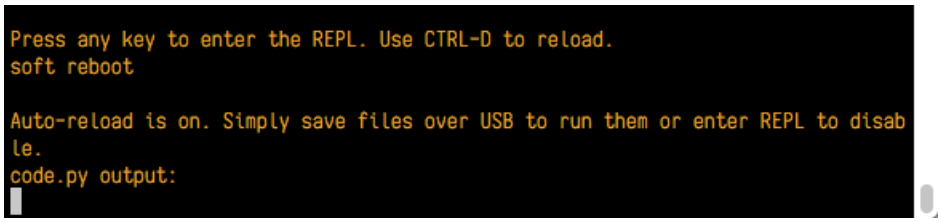
Save this file. Nothing happens to your board. Let's check the serial console to see what's going on.




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.




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.

For these boards, your internal storage is from the chip itself. So, these boards don't have enough space for all of the libraries. If you try to copy over the entire `lib` folder you won't have enough space 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

Libraries 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, 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!

If you'd like to update the entire bundle at once, drag the lib folder to your CIRUCITPY drive. Different operating systems will have a different dialog pop up. You want to tell it to replace the current folder. Then you're updated and ready to go!

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.

# 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.

## 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 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 program on Circuit Playground Express, press the reset button just once to get the CPLAYBOOT drive to show up. Pressing it twice will not work.

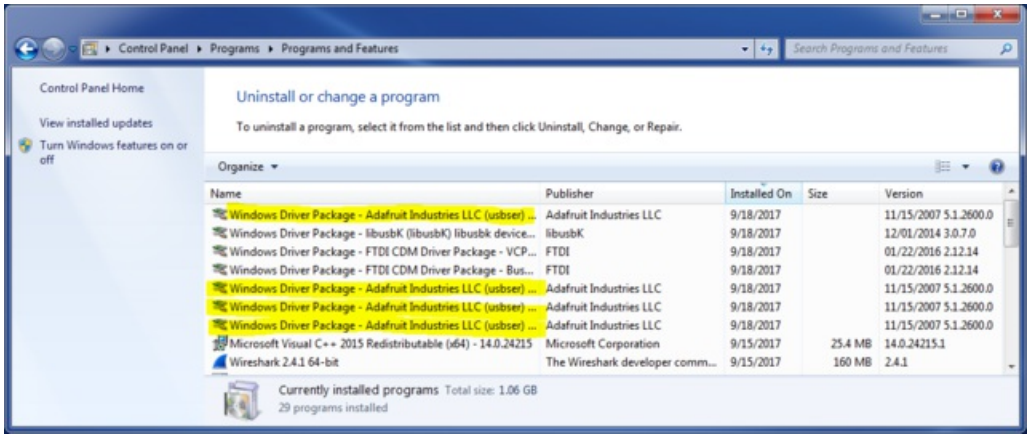
### Windows 10

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

### Windows 7

The latest version of the Adafruit Windows Drivers (version 2.0.0.0 or later) will fix the missing *boardname*BOOT drive problem on Windows 7. To resolve this, first uninstall the old versions of the drivers:

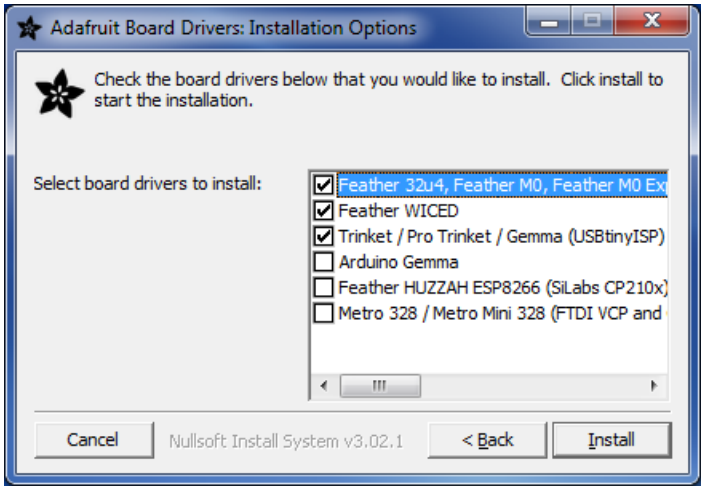
- Unplug any boards. In **Uninstall or Change a Program (Control Panel->Programs->Uninstall a program)**, uninstall everything named "Windows Driver Package - Adafruit Industries LLC ...".




- Now install the new 2.0.0.0 (or higher) Adafruit Windows Drivers Package:

Download Latest Drivers

- 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 or on the Adafruit Discord if this does not work for you!

## CircuitPython RGB Status Light

The Feather M0 Express, Metro M0 Express, Gemma M0, and Trinket M0 all have a single NeoPixel or DotStar RGB LED on the board that indicates the status of 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
- **YELLOW**: Circuit Python is in safe mode: it crashed and restarted
- **WHITE**: REPL is running
- **BLUE**: Circuit Python is starting up

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.

## 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 become corrupted.

This happens most often when the `CIRCUITPY` disk is not safely ejected before being reset by the button or being disconnected from USB. 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.

### For the Circuit Playground Express, Feather M0 Express, and Metro M0 Express:

1. Download the correct erase file:

Circuit Playground Express

https://adafru.it/AdI

Feather M0 Express

https://adafru.it/AdJ

Metro M0 Express

https://adafru.it/AdK

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 blue, indicating the erase has started.
5. After approximately 15 seconds, the NeoPixel will start flashing green.
6. Double-click the reset button on the board to bring up the *boardname*`BOOT` drive.
7. Drag the appropriate latest release of CircuitPython `.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, you can find it here.

### For the Gemma M0, Trinket M0, Feather M0: Basic (Proto) and Feather Adalogger:

1. Download the erase file:

Gemma M0, Trinket M0, Feather M0 Basic, Feather Adalogger

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 .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 version of CircuitPython for your board, you can find it here.

## 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.

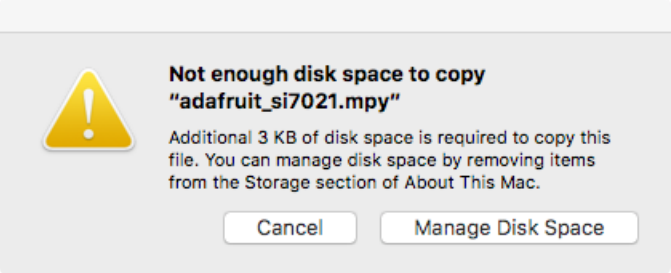
### Delete something!

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.

### 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.

### Mac OSX loves to add extra files.




Luckily you can disable some of the extra hidden files that Mac OSX 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 OSX:

### Prevent & Remove Mac OSX Hidden Files

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

```
ls -l /Volumes
```

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

Now follow the steps from this question to run these terminal commands that stop hidden files from being created on the board:

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

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

However there are still some cases where hidden files will be created by Mac OSX. In particular if you copy a file that was downloaded from the internet it will have special metadata that Mac OSX 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 Mac OSX Without Creating Hidden Files

Once you've disabled and removed hidden files with the above commands on Mac OSX 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:

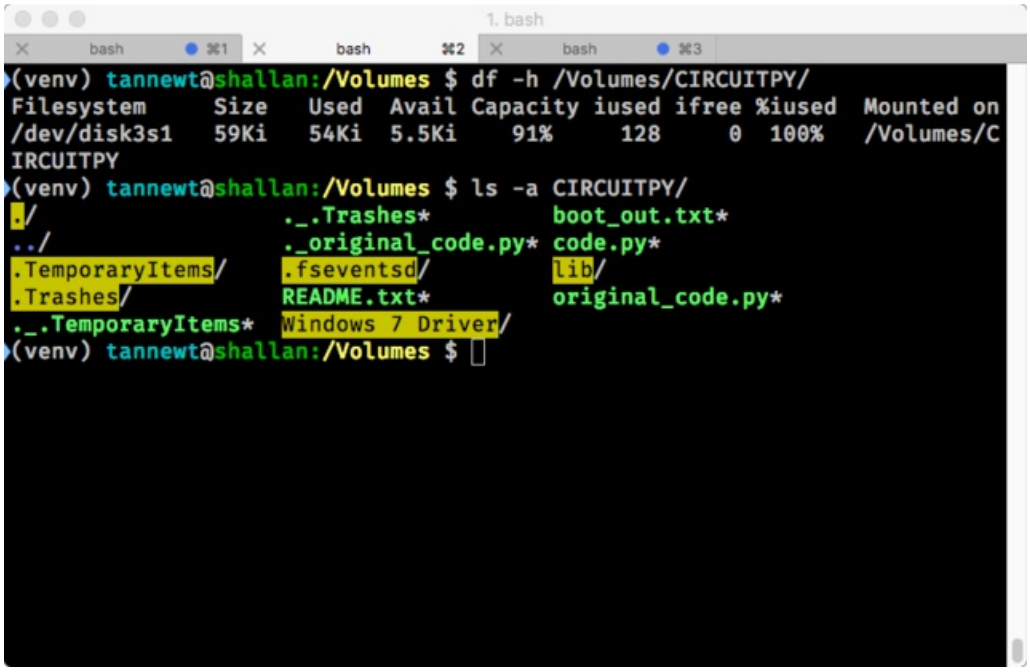
```
cp -X foo.mpy /Volumes/CIRCUITPY
```

Or to copy a folder and all of its child files/folders use a command like:

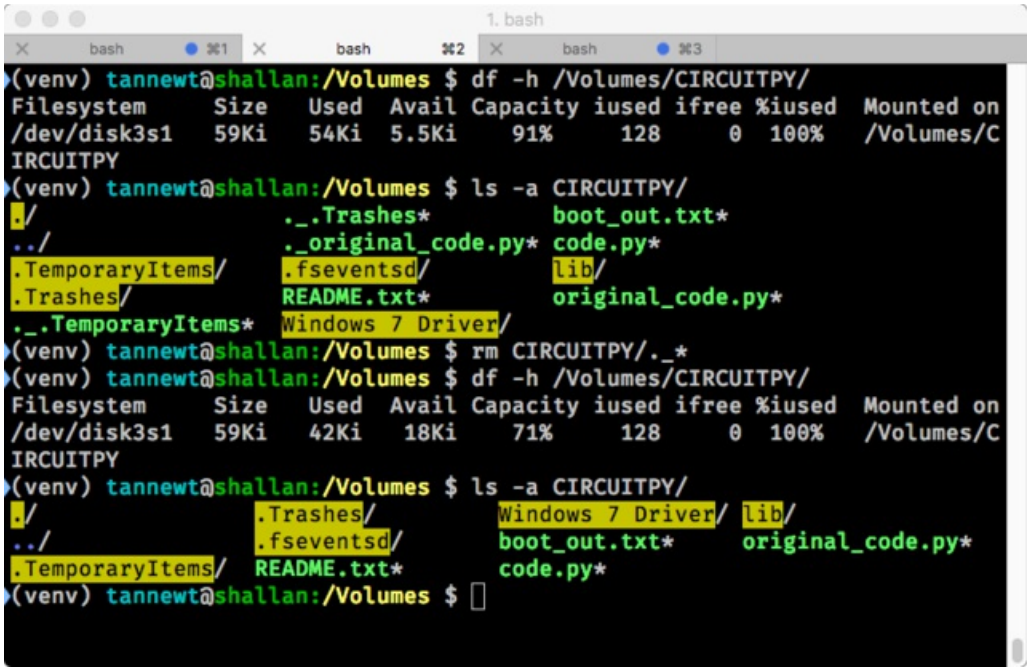
```
cp -rX folder_to_copy /Volumes/CIRCUITPY
```

### Other Mac OSX 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:

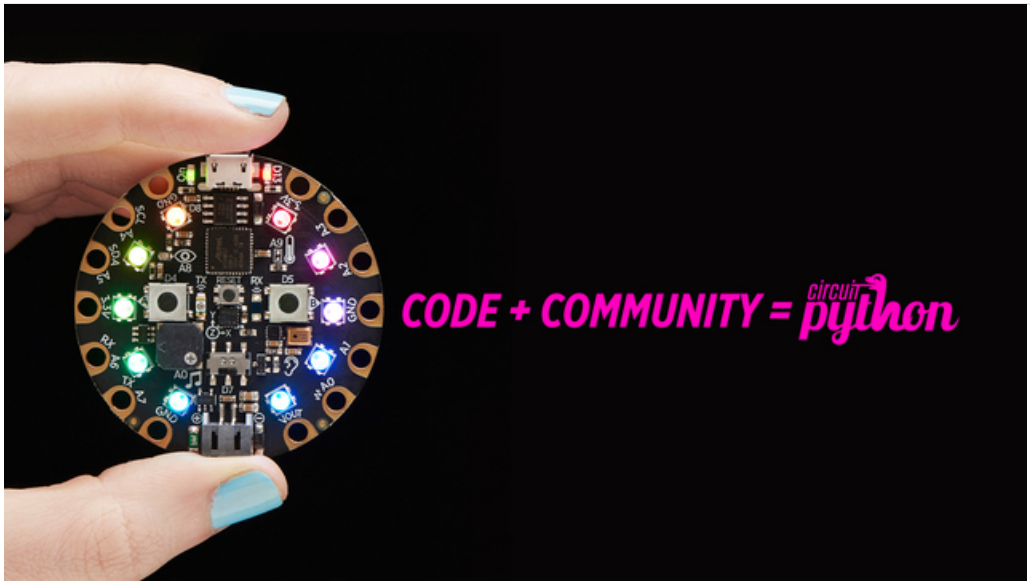



Lets remove the `._` files first.




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

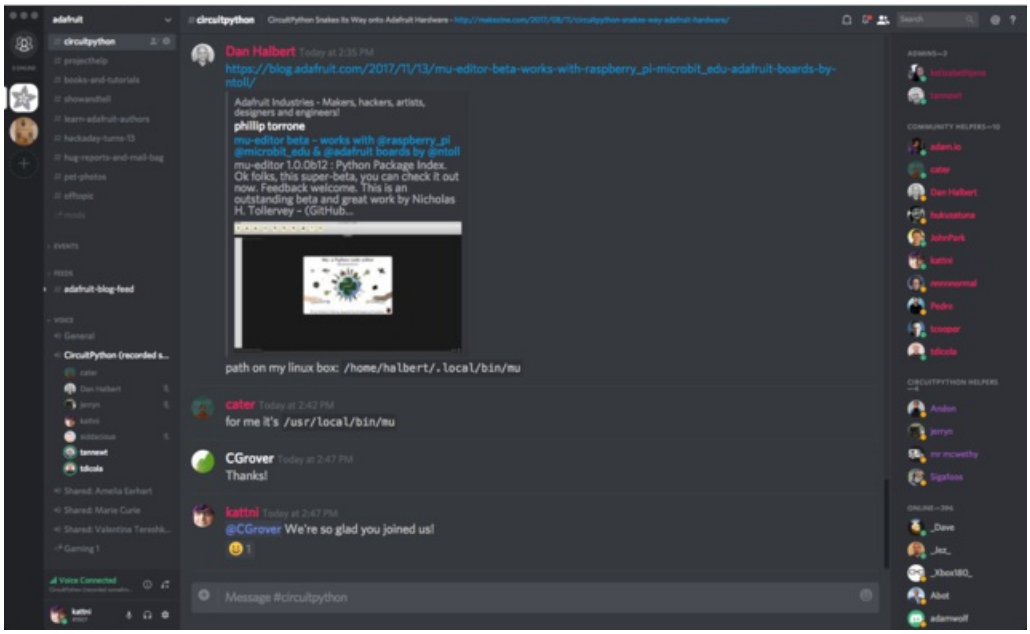
## Welcome to the Community!




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

Everyone is welcome! CircuitPython is Open Source. This means it's available for anyone to use, edit, copy and improve upon. This also means CircuitPython becomes better because of you being a part of it. It doesn't matter whether this is your first microcontroller board or you're a computer engineer, you have something important to offer the Adafruit CircuitPython community. We're going to highlight some of the many ways you can be a part of it!

## Adafruit Discord




The Adafruit Discord server is the best place to start. Discord is where the community comes together to volunteer and provide live support of all kinds. From general discussion to detailed problem solving, and everything in between,

Discord is a digital maker space with makers from around the world.

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

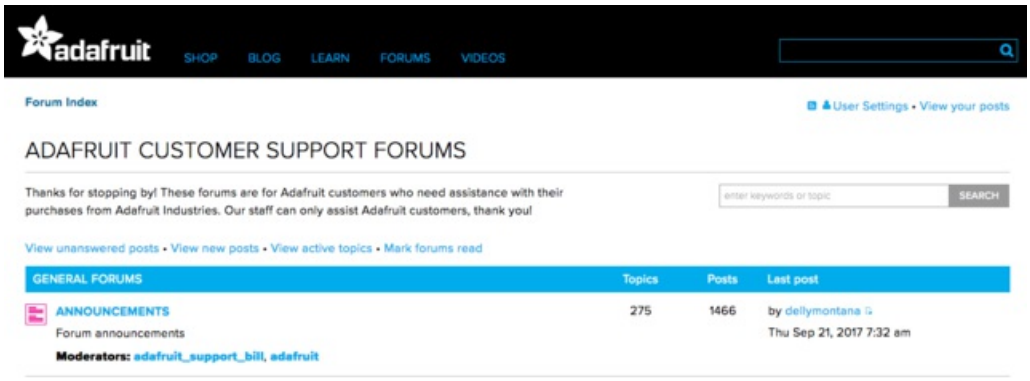
The CircuitPython channel is where to go with your CircuitPython questions. #circuitpython is there for new users and developers alike so feel free to ask a question or post a comment! Everyone of any experience level is welcome to join in on the conversation. We'd love to hear what you have to say!

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

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

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

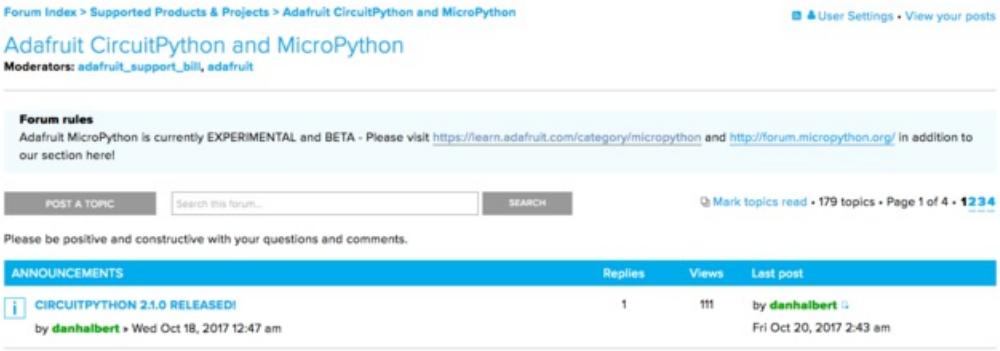
## Adafruit Forums




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

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

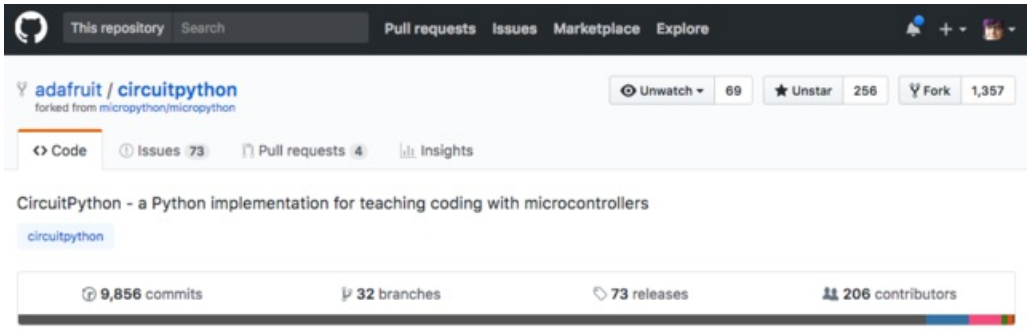
There are forum categories that cover all kinds of topics, including everything Adafruit. The Adafruit CircuitPython and MicroPython category under "Supported Products & Projects" is the best place to post your CircuitPython questions.




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.

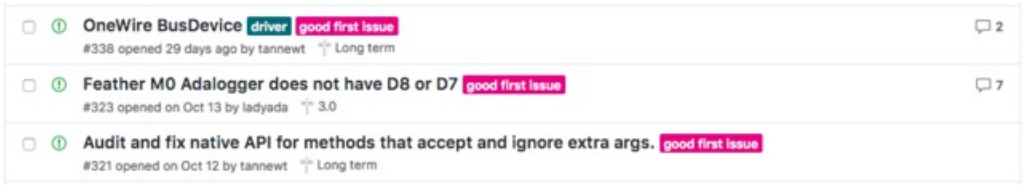
You might think you're just getting started, but you definitely know something that someone else doesn't. The great thing about the forums is that you can help others too! Everyone is welcome and encouraged to provide constructive feedback to any of the posted questions. This is an excellent way to contribute 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 itself. If you need an account, visit https://github.com/ and sign up.

If you're new to GitHub or programming in general, there are great opportunities for you. Head over to adafruit/circuitpython on GitHub, click on "Issues", and you'll find a list that includes issues labeled "good first issue". These are things we've identified as something that someone with any level of experience can help with. These issues include options like updating documentation, providing feedback, and fixing simple bugs.



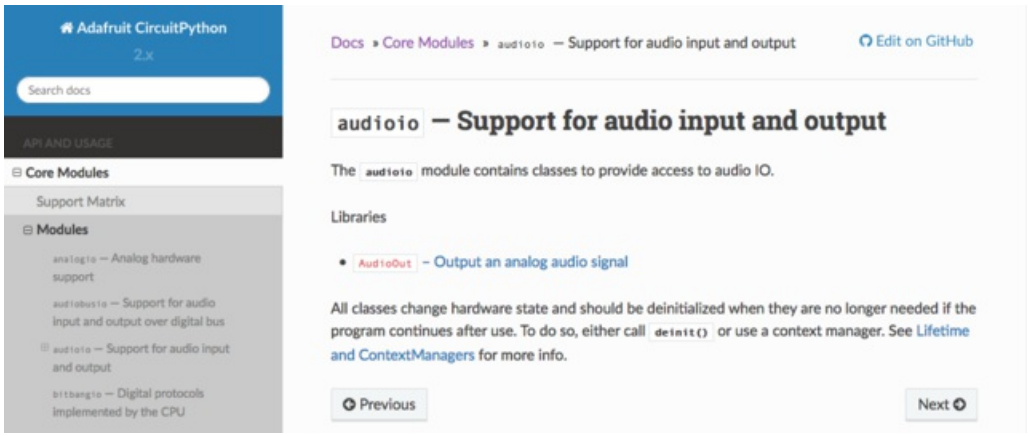

Already experienced and looking for a challenge? Checkout the rest of the issues list and you'll find plenty of ways to contribute. You'll find everything from new driver requests to core module updates. There's plenty of opportunities for everyone at any level!

When working with CircuitPython, you may find problems. If you find a bug, that's great! We love bugs! Posting a detailed issue to GitHub is an invaluable way to contribute to improving CircuitPython. Be sure to include the steps to replicate the issue as well as any other information you think is relevant. The more detail, the better!

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

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

## ReadTheDocs




ReadTheDocs 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. 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!

## CircuitPython Playground

Here's a bunch of examples you can get started with your Trinket M0 + CircuitPython

## CircuitPython Expectations

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

## Small Disk Space

Since we use the internal flash for disk, and that's shared with runtime code, its limited! Only about 50KB of space. Our Express line of boards have a whopping 2 MB of external Flash, if you need more space

## No PWM & PulseIO

As of **CircuitPython 2.1** we have added PulseIO support to Trinket & Gemma M0. That means PWM, piezo, servo, DHT22 and Infrared support!

## 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'. For that support, check out the Circuit Playground Express or other Express boards

However, I2C, UART, capacitive touch, NeoPixel, PWM, analog in and out, digital IO, logging storage, and HID do work! Check below for quick starts on all these.

## 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 just some of the many featuers you can use

## Things that are Built In and Work

### flow control

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

### math

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

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

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

### tuples, lists, arrays, and dictionaries

You can organize data in ()', []'s , and {}'s 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 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
```

## Things to watch out for!

- The wide body of python libraries have not been ported over, so while we wish you could `import numpy` , numpy isn't available. So you may have to port some code over yourself!
- For the ATSAMD21 based boards (Feather M0, Metro M0, Trinket M0, Gemma M0, Circuit PlayGround Express) there's a limited amount of RAM, we've found you can have about 250-ish lines of python (that's with various libraries) before you hit MemoryErrors. The upcoming SAMD51 chipset will help with that a ton but its not yet available)
- Non-Express boards like Trinket M0 and Gemma M0 and non-Express Feathers do not include all of the hardware support. For example, `audioio` and `bitbangio` are not included.
- Integers can only be up to 31 bits. Integers of unlimited size are not supported.

- We keep up with MicroPython stable releases, so check out the core 'differences' they document here.

## CircuitPython Digital In & Out

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

This quick-start example shows how you can turn one of the Gemma pads into a button *input* with pullup resistor (built in) and then use that to control another digital *output* - the built in LED

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

```
# CircuitPython IO demo #1 - General Purpose I/O

from digitalio import DigitalInOut, Direction, Pull
import board
import time

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

button = DigitalInOut(board.D2)
button.direction = Direction.INPUT
button.pull = Pull.UP

while True:
    # we could also just do "led.value = not button.value" !
    if button.value:
  led.value = False
    else:
  led.value = True

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

Note that we made the code a little less 'pythonic' than necessary, the if/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.

Find the pin or pad labeled **D2** (sometimes just **2**) and use a wire to touch it to **GND**, the onboard red LED will turn on!

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

## CircuitPython Analog In

This quick-start example shows how you can read the analog voltages on all five Trinket M0 pins.

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

```
# Trinket IO demo - analog inputs

from analogio import AnalogIn
import board
import time

analog0in = AnalogIn(board.D0)
analog1in = AnalogIn(board.D1)
analog2in = AnalogIn(board.D2)
analog3in = AnalogIn(board.D3)
analog4in = AnalogIn(board.D4)

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

while True:
    print("D0: %0.2f \t D1: %0.2f \t D2: %0.2f \t D3: %0.2f \t D4: %0.2f" %
          (getVoltage(analog0in),
           getVoltage(analog1in),
           getVoltage(analog2in),
           getVoltage(analog3in),
           getVoltage(analog4in) ))
    time.sleep(0.1)
```

### Creating analog inputs

`analog0in = AnalogIn(D0)`
`analog1in = AnalogIn(D1)`
`analog2in = AnalogIn(D2)`
`analog3in = AnalogIn(D3)`
`analog4in = AnalogIn(D4)`

Creates five objects, one for each pad, and connects the objects to D0 through D4 inclusive, as analog inputs.

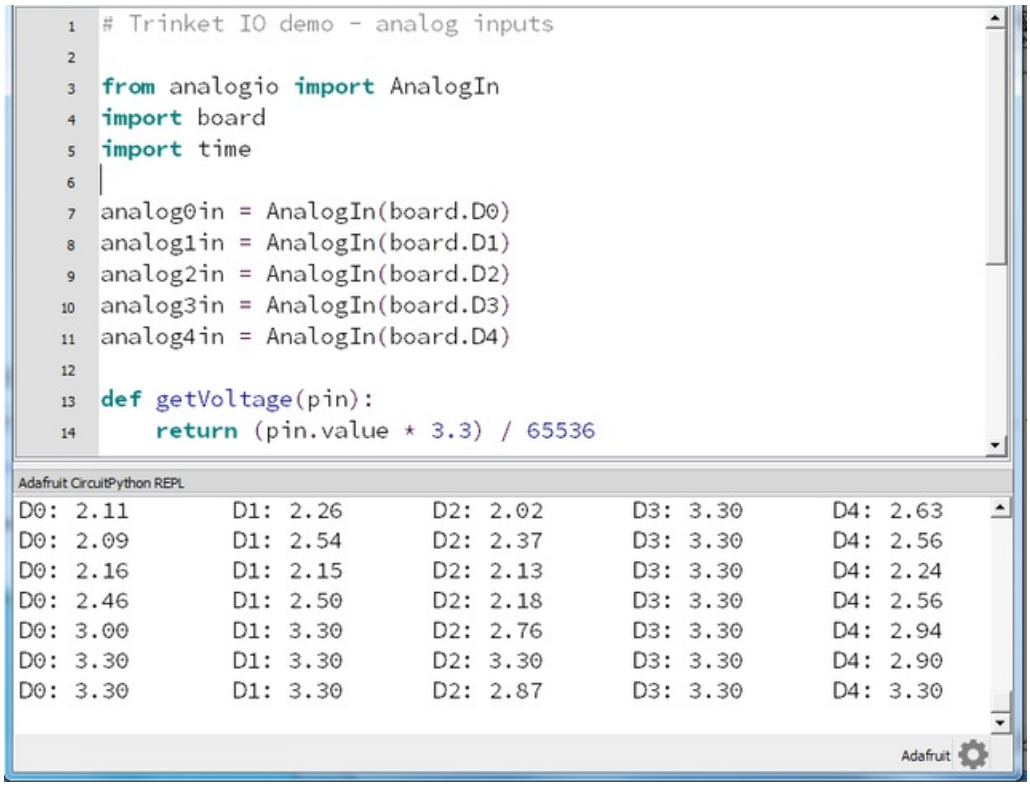
### GetVoltage Helper

`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 three voltages as floating point values (the %f indicates to print as **f**loating point) by calling getVoltage on each of our analog objects.

If you connect to the serial port REPL, you'll see the voltages printed out. By default the pins are *floating* so the voltages will vary. Try touching a wire from **D0** to the **GND** or **3Vo** pad to see the voltage change!


1 # Trinket IO demo - analog inputs
2
3 from analogio import AnalogIn
4 import board
5 import time
6
7 analog0in = AnalogIn(board.D0)
8 analog1in = AnalogIn(board.D1)
9 analog2in = AnalogIn(board.D2)
10 analog3in = AnalogIn(board.D3)
11 analog4in = AnalogIn(board.D4)
12
13 def getVoltage(pin):
14 return (pin.value * 3.3) / 65536
Adafruit CircuitPython REPL
D0: 2.11 D1: 2.26 D2: 2.02 D3: 3.30 D4: 2.63
D0: 2.09 D1: 2.54 D2: 2.37 D3: 3.30 D4: 2.56
D0: 2.16 D1: 2.15 D2: 2.13 D3: 3.30 D4: 2.24
D0: 2.46 D1: 2.50 D2: 2.18 D3: 3.30 D4: 2.56
D0: 3.00 D1: 3.30 D2: 2.76 D3: 3.30 D4: 2.94
D0: 3.30 D1: 3.30 D2: 3.30 D3: 3.30 D4: 2.90
D0: 3.30 D1: 3.30 D2: 2.87 D3: 3.30 D4: 3.30
Adafruit

## CircuitPython Analog Out

This quick-start example shows how you can set the DAC (true analog voltage output) on Trinket M0 pad **D1** (no other pins do analog out). There's a little squiggle on the pin so you know its analog output.

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

```
# Trinket IO demo - analog output

from analogio import AnalogOut
import board
import time

aout = AnalogOut(board.D1)

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):
    aout.value = i
```

### Creating an analog output

`aout = AnalogOut(D1)`

Creates an object **aout** that is connected to the only DAC pin available - D1.

### Setting the analog output

The DAC on the Trinket M0 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.

E.g. 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

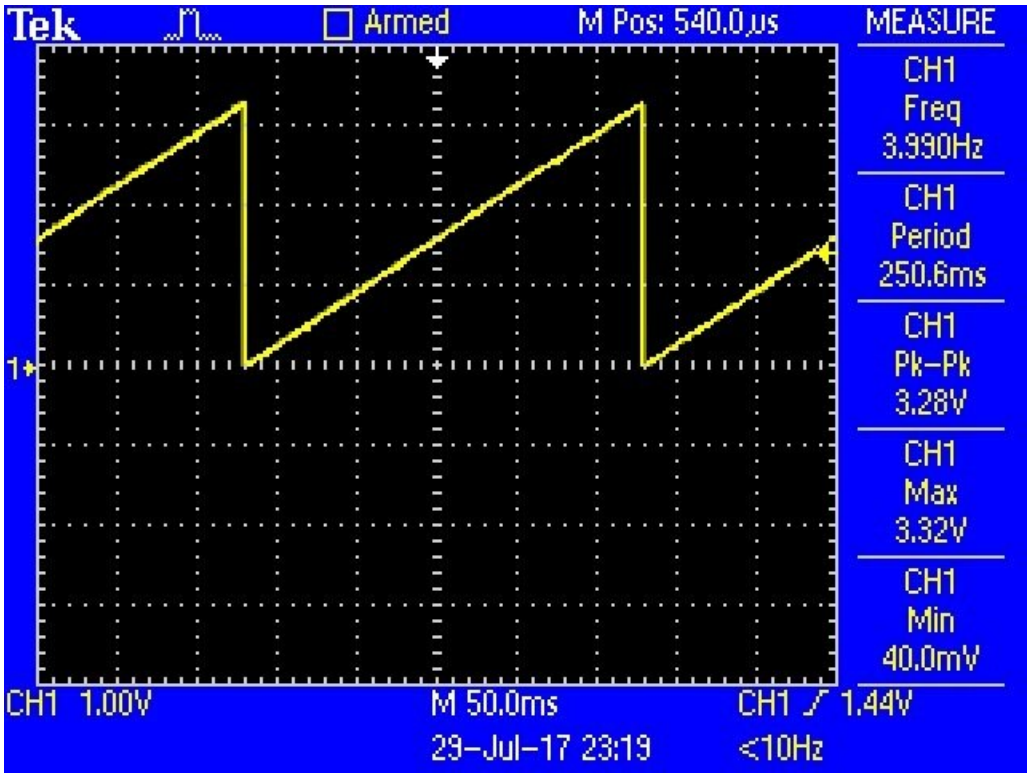
Writing 65535 is the same as 1023 which is the top range and you'll get 3.3V output

### Main Loop

The main loop is fairly simple, it just 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.

Bigger boards like the Metro or Feather M0 have more code space and can perform audio playback capabilities via the DAC.


Tek
Armed
M Pos: 540.0µs
MEASURE
CH1
Freq
3.990Hz
CH1
Period
250.6ms
CH1
Pk-Pk
3.28V
CH1
Max
3.32V
CH1
Min
40.0mV
CH1 1.00V
M 50.0ms
CH1 ⁄ 1.44V
29-Jul-17 23:19
<10Hz

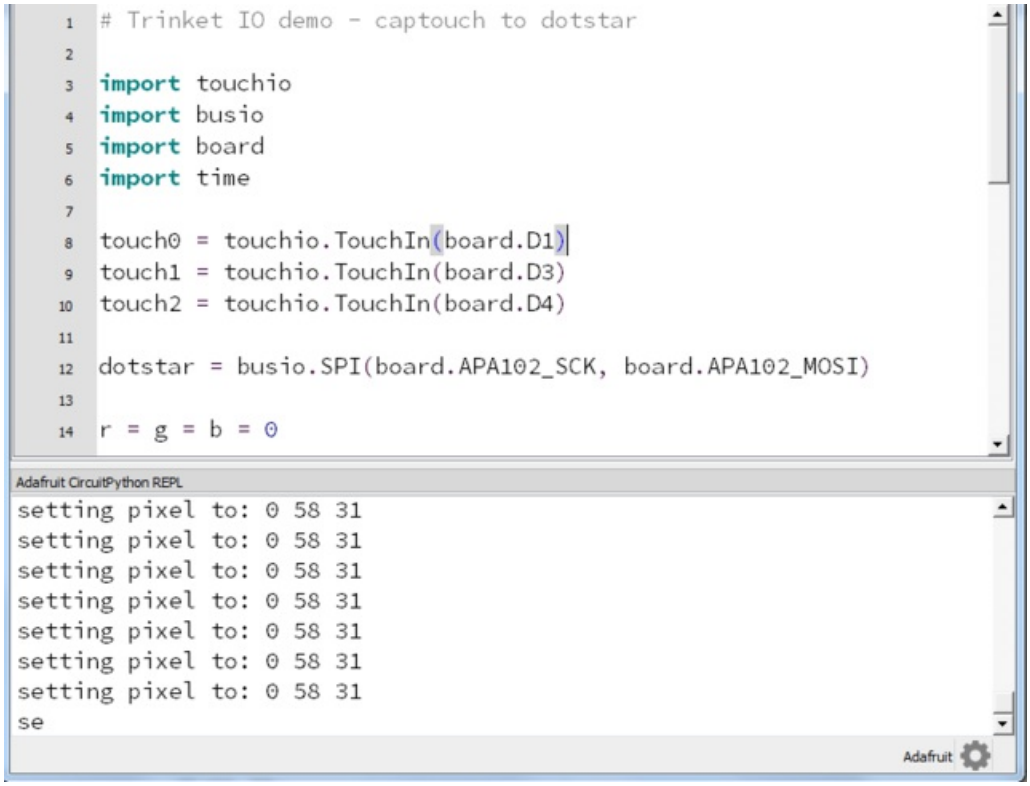
## CircuitPython Internal DotStar

This quick-start example builds upon the previous example, but shows how you can create interactivity using capacitive touch. It also demonstrates the built in DotStar LED and how you can change the color on your own.

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

```
# Trinket IO demo - captouch to dotstar

import touchio
import busio
import board
import time

touch0 = touchio.TouchIn(board.D1)
touch1 = touchio.TouchIn(board.D3)
touch2 = touchio.TouchIn(board.D4)

dotstar = busio.SPI(board.APA102_SCK, board.APA102_MOSI)

r = g = b = 0

def setPixel(red, green, blue):
    if not dotstar.try_lock():
        return
    print("setting pixel to: %d %d %d" % (red, green, blue))

    data = bytearray([0x00, 0x00, 0x00, 0x00,
                      0xff, blue, green, red,
                      0xff, 0xff, 0xff, 0xff])
    dotstar.write(data)
    dotstar.unlock()
    time.sleep(0.01)

while True:
    if touch0.value:
        r = (r+1) % 256
    if touch1.value:
        g = (g+1) % 256
    if touch2.value:
        b = (b+1) % 256

    setPixel(r, g, b)
```

Each of the three pads will change the color of the built in mini DotStar LED. You can touch each pad in order to see the LED change colors, or you can open up the serial console to see the touches detected and the pixel color printed out.


1 # Trinket IO demo - captouch to dotstar
2
3 import touchio
4 import busio
5 import board
6 import time
7
8 touch0 = touchio.TouchIn(board.D1)
9 touch1 = touchio.TouchIn(board.D3)
10 touch2 = touchio.TouchIn(board.D4)
11
12 dotstar = busio.SPI(board.APA102_SCK, board.APA102_MOSI)
13
14 r = g = b = 0
Adafruit CircuitPython REPL
setting pixel to: 0 58 31
setting pixel to: 0 58 31
setting pixel to: 0 58 31
setting pixel to: 0 58 31
setting pixel to: 0 58 31
setting pixel to: 0 58 31
setting pixel to: 0 58 31
se
Adafruit

## CircuitPython Cap Touch

This quick-start example shows how you can read the capacitive touch sensors built into three of the Trinket M0 pins.

Note that because we are using the built in hardware capacitive touch support, you can only use pins **D1**, **D3** and **D4**

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

```
# Trinket IO demo - captouch

import touchio
import board
import time

touch0 = touchio.TouchIn(board.D1)
touch1 = touchio.TouchIn(board.D3)
touch2 = touchio.TouchIn(board.D4)

while True:
    if touch0.value:
        print("D1 touched!")
    if touch1.value:
        print("D3 touched!")
    if touch2.value:
        print("D4 touched!")
    time.sleep(0.01)
```

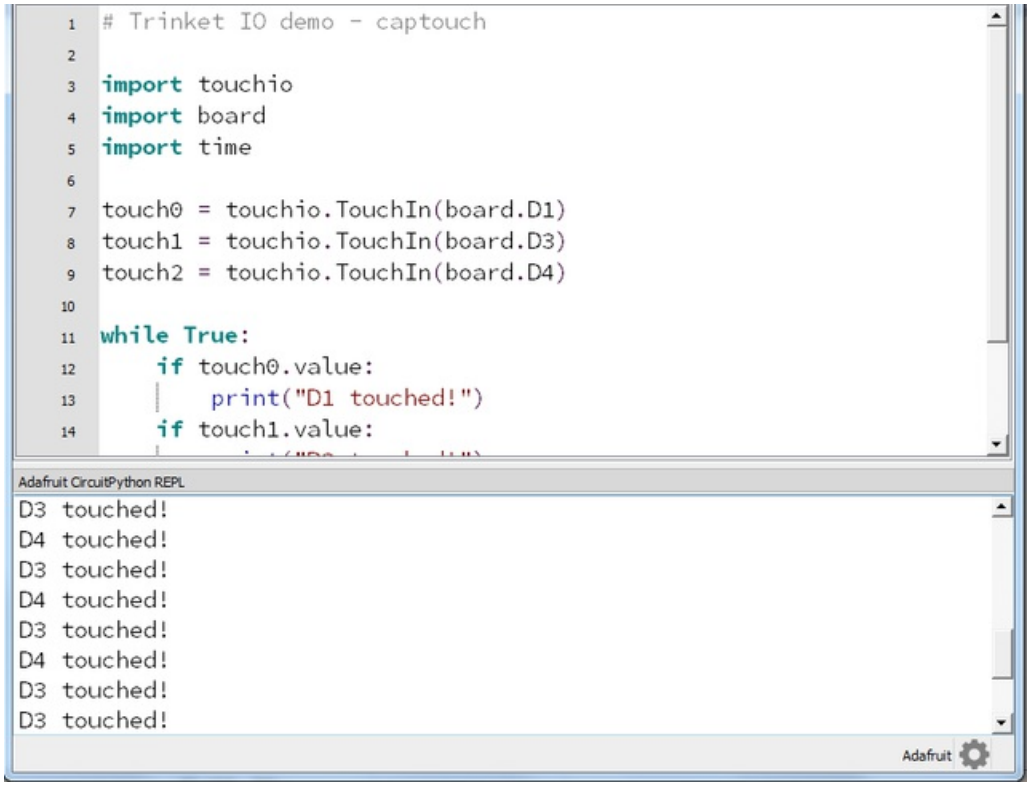
You can open up the serial console to see the touches detected and printed out.

### Creating an capacitive touch input

All three pads can be used as capacitive TouchIn devices:

```
touch0 = touchio.TouchIn(D1)
touch1 = touchio.TouchIn(D3)
touch2 = touchio.TouchIn(D4)
```

Creates three objects, one connected to each pin that has hardware capacitive touch support

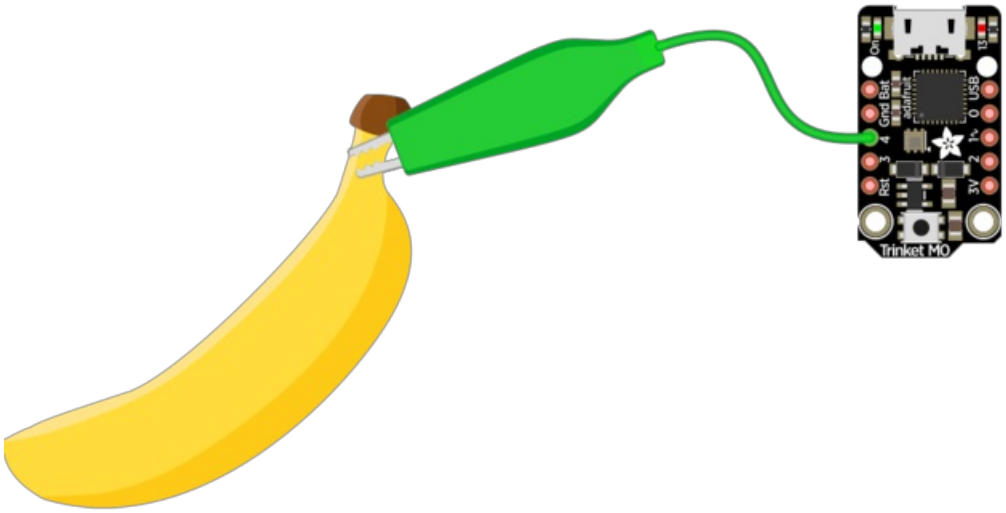



## Main Loop

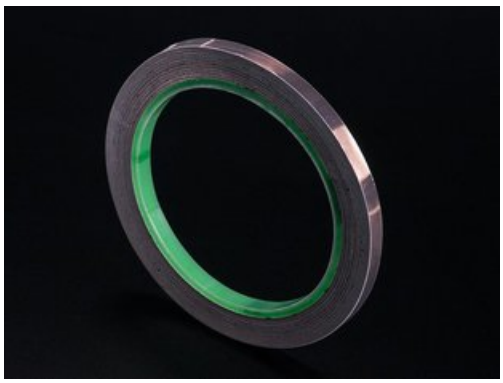
The main loop checks each sensor one after the other, to determine if it has been touched. If `touch0.value` returns True, that means that that pin `D1`, detected a touch. For each pin, if it has been touched, a message will print.

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 speed it up.

Note that no extra hardware is required, you can touch the pins directly, but you may want to attach wires to foil tape, 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

PRODUCT ID: 1128

https://adafru.it/eNZ

$5.95
IN STOCK

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

PRODUCT ID: 1127

https://adafru.it/y8F

$19.95
IN STOCK

Small Alligator Clip to Male Jumper Wire Bundle - 12 Pieces

PRODUCT ID: 3255

https://adafru.it/xAV

$7.95

IN STOCK

## CircuitPython I2C Scan

This quick-start example shows how you can use CircuitPython to scan the I2C bus for all connected devices

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

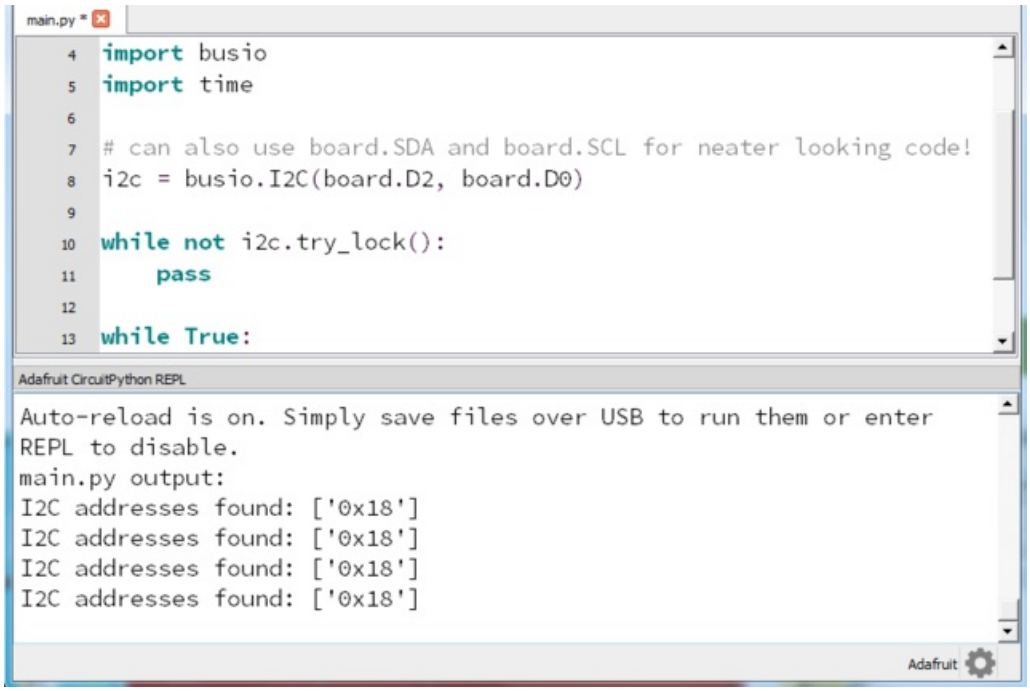
```
# Gemma/Trinket IO demo - I2C scan

import board
import busio
import time

# can also use board.SDA and board.SCL for neater looking code!
i2c = busio.I2C(board.D2, board.D0)

while not i2c.try_lock():
    pass

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

You can also use the Trinket to chat with I2C sensors and devices. Before you start, we recommend connecting it up and doing an I2C scan so you can tell if it was detected.

You can create the I2C devices on the Trinket M0's **D2** (default SCL) and **D0** (default SDA) pins. You can use **board.D2** or **board.SDA** (which is more flexible if you're going to run the code on another device).

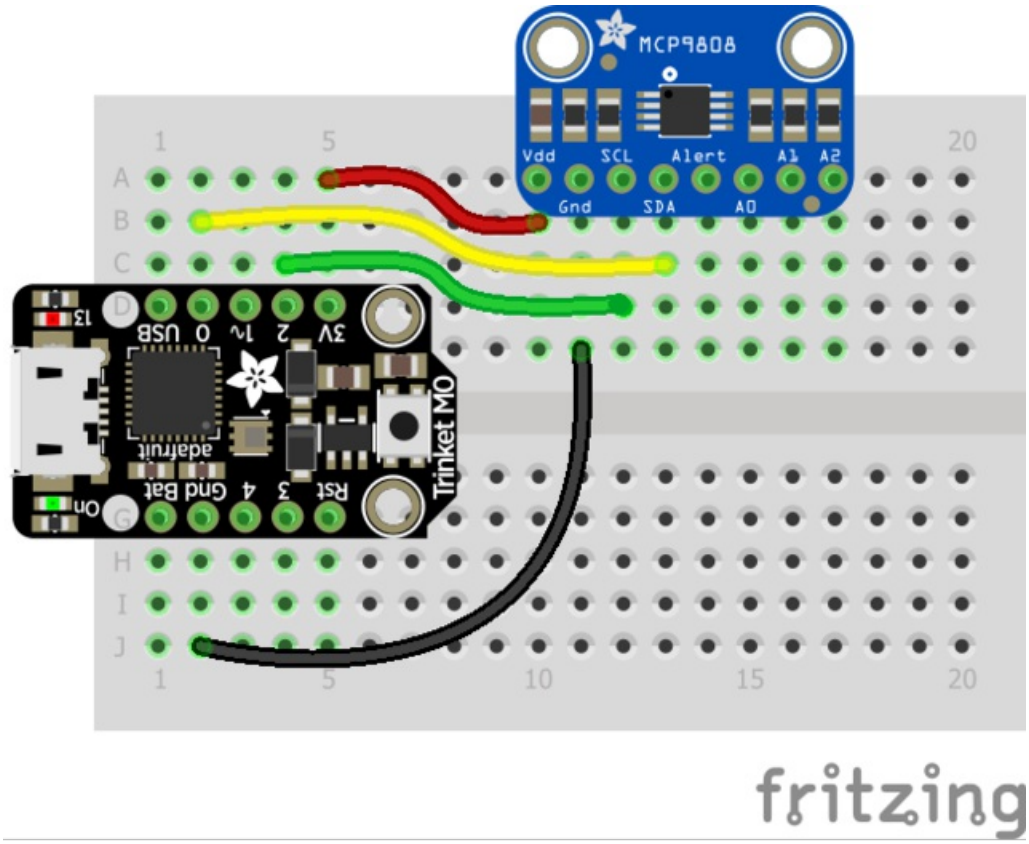
Then the I2C device must be locked (that means you are reserving access to it)

Then run a scan with `i2c.scan()` It will return an array of addresses, but since usually they are referred to in hex format, you may want to convert the array to hexadecimals with `[hex(i) for i in i2c.scan()])`



Don't forget that the Trinket M0 does not have I2C pullup resistors built in, you must add 2.2-10K ohm pullups on both SDA and SCL to 3.3V (our breakouts come with them already)

We wired up a MCP9808 breakout with address 0x18 to test it!




trinkmcp.fzz

https://adafru.it/zyA

## CircuitPython I2C Sensor

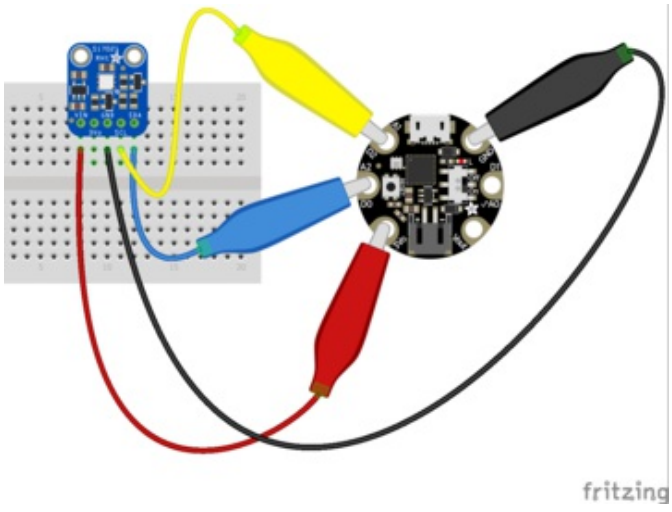
We have drivers for many popular I2C sensors in our driver bundle (and more being written all the time!)

I2C is a 2-wire protocol for communicating with simple sensors and devices and its really easy to use with CircuitPython

Remember that the Gemma & Trinket M0 does not have the required i2c pull-up resistors on SDA or SCL! You must have those on the sensor board (all of ours do) or add them yourself. 10K ohm pullups to 3.3V work well, you cannot use the 'internal' pullups.

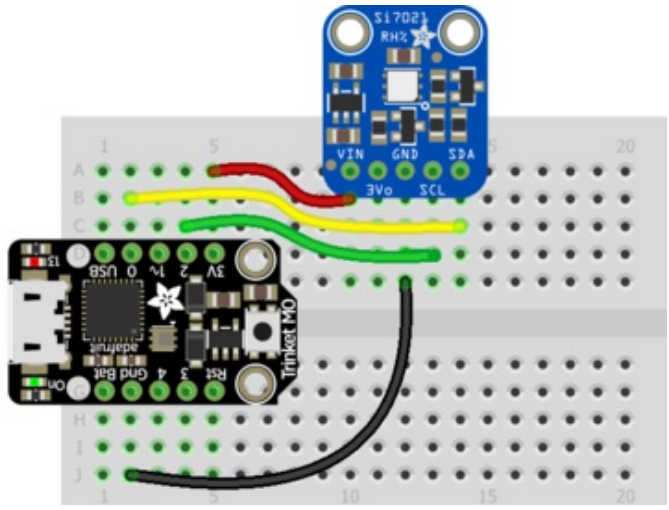
Lets try wiring up to a nice Si7021 temperature & humidity sensor:

```
# I2C sensor demo

import board
import busio
import adafruit_si7021
import time

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

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

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

# Create library object on our I2C port
si7021 = adafruit_si7021.SI7021(i2c)


# Use library to read the data!
while True:
    print("Temp: %0.2F *C   Humidity: %0.1F %%" % (si7021.temperature, si7021.relative_humidity))
    time.sleep(1)
```



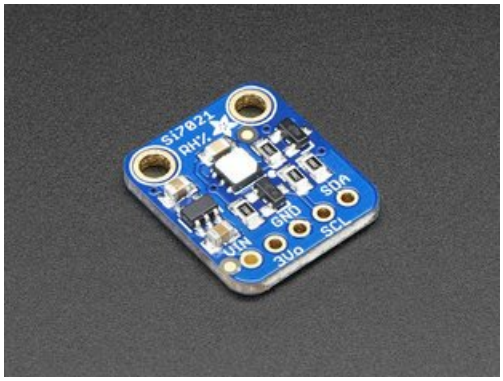

We used our Alligator-to-breadboard wires to connect up the Gemma to a Si7021 breakout

- Vin connects to 3.3V
- GND connects to GND
- SDA connects to D0
- SCL connects to D2




With a Trinket M0, a small breadboard fits both pieces, just wire it up so

- Vin connects to 3.3V
- GND connects to GND
- SDA connects to D0
- SCL connects to D2



Adafruit Si7021 Temperature & Humidity Sensor Breakout Board

PRODUCT ID: 3251

https://adafru.it/y6F

$6.95
IN STOCK

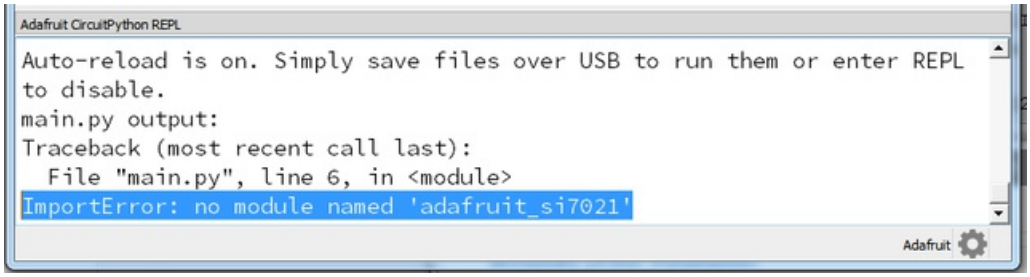
Small Alligator Clip to Male Jumper Wire Bundle - 12 Pieces

PRODUCT ID: 3255

https://adafru.it/xAV

$7.95
IN STOCK

Then check the REPL. If you have not yet used this chip you may get an **ImportError: no module named 'adafruit_si7021'**

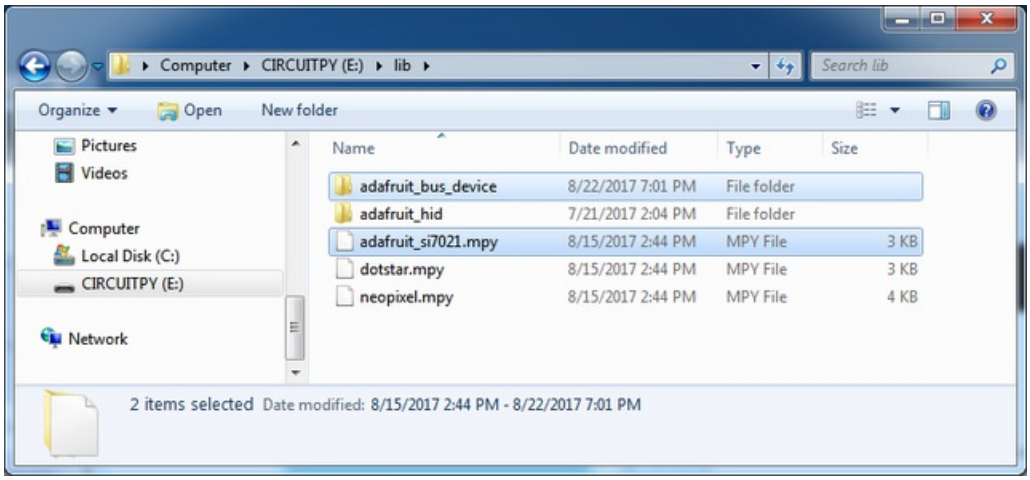


That means you need to install the **Adafruit_Si7021** library that gives you the friendly interface we use above.

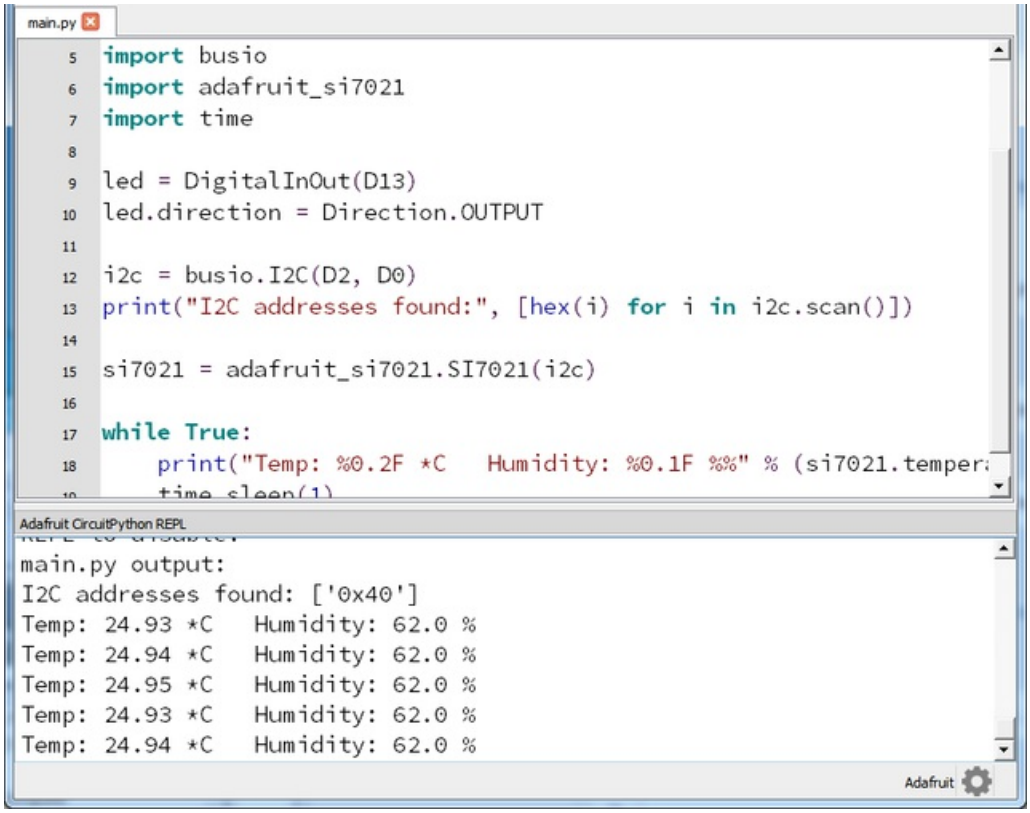
Check out our page on **Installing Libraries** to learn how to download the driver bundle and drag the driver you need to the **lib** folder

You will also need the **adafruit_bus_device** library folder - that will give you I2C access in a nice manner!

Once you're done you'll see you have the libraries installed:




Finally if you re-run you will be able to see the temperature and humidity data from the sensor:



## CircuitPython UART Serial

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

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

```
# Trinket IO demo - USB/Serial echo

from digitalio import DigitalInOut, Direction
import board
import busio
import time

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

# You can also use board.TX and board.RX for prettier code!
uart = busio.UART(board.D4, board.D3, baudrate=9600)

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

    if data != None:
        led.value = True

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

        led.value = False
```

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 GPS's, some sensors, or other microcontrollers!

You can create a new UART object with `uart = busio.UART(board.D4, board.D3, baudrate=9600)` You can use either **D4** and **D3** *or* D0 and D2 as the transmitting and receiving pins (respectively) on the Trinket M0 - you can even have *two* UART objects!

We've marked **D4** and **D3** as RX and RX on the bottom of the PCB so our example will use those. Set the baudrate to whatever you like.

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 bytearray 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.

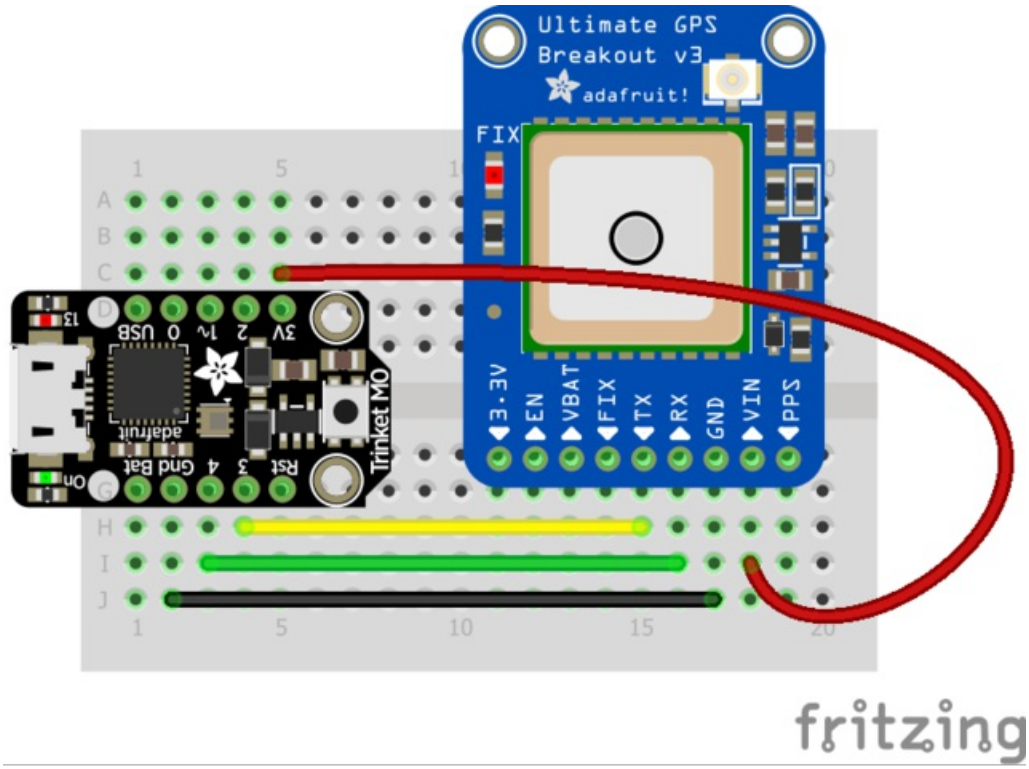
The data that is returned is in a byte array, if you want to convert it to a string, you can use this handy line of code which will run chr() on each byte:

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

For more UART details, check out the module documentation!

https://adafru.it/yCH

To run this demo, you'll need something to generate UART data. We connected up a GPS!




trinkgps.fzz

https://adafru.it/zyf

```
main.py
1  # Trinket IO demo - USB/Serial echo
2
3  from digitalio import DigitalInOut, Direction
4  import board
5  import busio
6  import time
7
8  led = DigitalInOut(board.D13)
9  led.direction = Direction.OUTPUT
10
```

```
Adafruit CircuitPython REPL
$GPGGA,001145.799,,,,,0,00,,,M,,M,,*7E
$GPGSA,A,1,,,,,,,,,,,,,,*1E
$GPRMC,001145.799,V,,,,,0.00,0.00,060180,,,N*44
$GPVTG,0.00,T,,M,0.00,N,0.00,K,N*32
$GPGGA,001146.799,,,,,0,00,,,M,,M,,*7D
$GPGSA,A,1,,,,,,,,,,,,,,*1E
$GPRMC,001146.799,V,,,,,0.00,0.00,060180,,,N*47
$GPVTG,0.00,T,,M,0.00,N,0.00,
Adafruit
```

## CircuitPython NeoPixel

NeoPixels are a revolutionary and ultra-popular way to add lights and color to your project. These stranded RGB lights have the controller inside the LED, so you just push the RGB data and the LEDs do all the work for you! They're a perfect match for CircuitPython

You can drive 300 pixels with brightness control and 1000 pixels without (set `brightness=1.0` in object creation). That's because to adjust the brighness 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!

```
# CircuitPython demo - NeoPixel

import board
import neopixel
import time

pixpin = board.D1
numpix = 10

strip = neopixel.NeoPixel(pixpin, numpix, brightness=0.3, auto_write=False)


def wheel(pos):
    # Input a value 0 to 255 to get a color value.
    # The colours are a transition r - g - b - back to r.
    if (pos < 0) or (pos > 255):
        return (0, 0, 0)
    if (pos < 85):
        return (int(pos * 3), int(255 - (pos*3)), 0)
    elif (pos < 170):
        pos -= 85
        return (int(255 - pos*3), 0, int(pos*3))
    else:
        pos -= 170
        return (0, int(pos*3), int(255 - pos*3))

def rainbow_cycle(wait):
    for j in range(255):
        for i in range(len(strip)):
            idx = int ((i * 256 / len(strip)) + j)
            strip[i] = wheel(idx & 255)
        strip.write()
        time.sleep(wait)

while True:
    strip.fill((255, 0, 0))
    strip.write()
    time.sleep(1)

    strip.fill((0, 255, 0))
    strip.write()
    time.sleep(1)

    strip.fill((0, 0, 255))
    strip.write()
    time.sleep(1)

    rainbow_cycle(0.001)    # rainbowcycle with 1ms delay per step
```

This code will work with any NeoPixel-compatible.

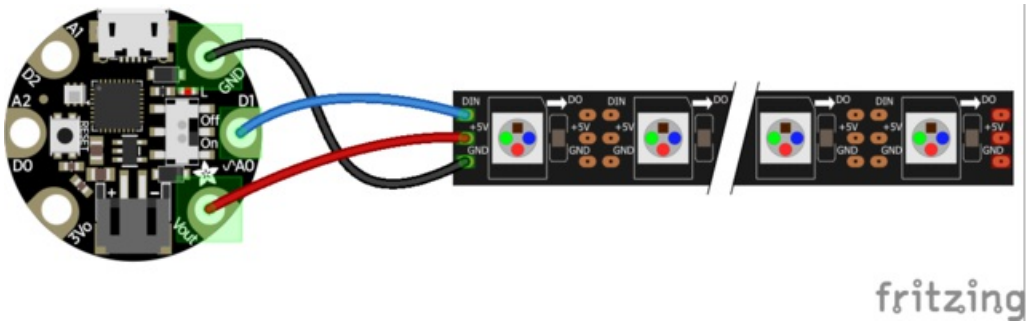
NeoPixels can be driven by any pin.

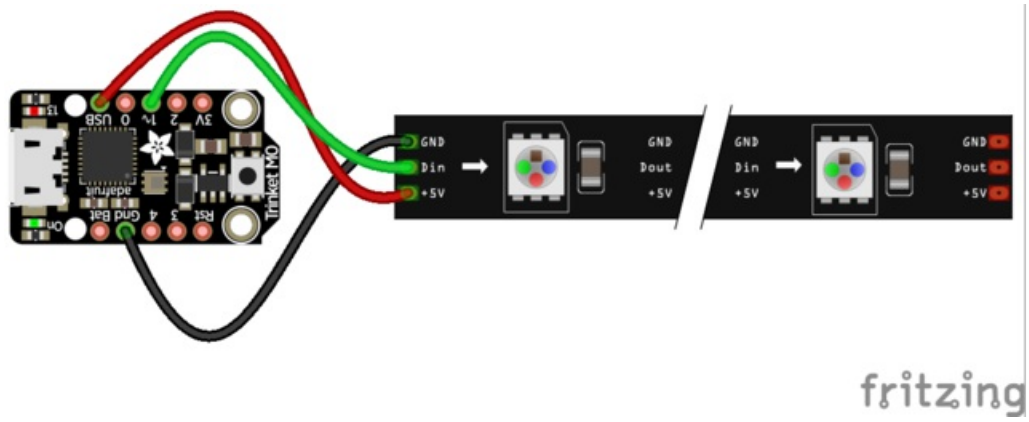
For powering the pixels from the board, the 3.3V regulator output from the Trinket/Gemma M0 can handle about 500mA peak which is about 50 pixels with 'average' use. If you want really bright lights and a lot of pixels, we recommend powering direct from the power source. On the Gemma M0 this is the **Vout** pad - that pad has direct power from USB or BAT, depending on which is higher voltage. On the Trinket M0 the **USB** or **BAT** pins will give you

direct power from the USB port or battery.

The NeoPixel object's argument list requires the pin you'll use (any pin can be used) and the number of pixels. There's two optional arguments, `brightness` (range from 0 off to 1.0 full brightness) and `auto_write`. When `auto_write` default is set to `True`, every change is immediately written to the strip of pixels, this is easier to use but *way* slower. if you set `auto_write=False` then you will have to call `strip.show()` when you want to actually write color data out.

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




Verify the wiring on your strip or device - plugging into the 'DOUT' side is a common mistake! Wire up NeoPixels only while the Trinket or Gemma 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.

You can drive 300 pixels with brightness control and 1000 pixels without (set `brightness=1.0` in object creation). That's because to adjust the brighness 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 **adafruit_dotstar.mpy** library file if you don't have it yet!

The DotStar object's argument list requires the two pins you'll use and the number of pixels. 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!

On the Gemma M0, if you use `adafruit_dotstar.DotStar(board.D2, board.D0...)` you'll get hardware SPI

On the Trinket M0, you can use **D2** & **D0**, **D2** & **D3**, **D3** & **D0**, or **D3** & **D4**

There's two optional arguments, `brightness` (range from 0 off to 1.0 full brightness) and auto_write. When `auto_write` default is set to True, where every change is immediately written to the strip of pixels, this is easier to use but *way* slower. if you set `auto_write=False` then you will have to call `strip.show()` when you want to actually write color data out.

```
# CircuitPython demo - Dotstar

import board
import adafruit_dotstar
import time

numpix = 64
strip = adafruit_dotstar.DotStar(board.D2, board.D0, numpix, brightness=0.2)

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 (int(pos * 3), int(255 - (pos*3)), 0)
    elif (pos < 170):
        pos -= 85
        return (int(255 - pos*3), 0, int(pos*3))
    else:
        pos -= 170
        return (0, int(pos*3), int(255 - pos*3))

def rainbow_cycle(wait):
    for j in range(255):
        for i in range(len(strip)):
            idx = int ((i * 256 / len(strip)) + j)
            strip[i] = wheel(idx & 255)
        strip.show()
        time.sleep(wait)

while True:
    strip.fill((255, 0, 0))
    strip.show()
    time.sleep(1)

    strip.fill((0, 255, 0))
    strip.show()
    time.sleep(1)

    strip.fill((0, 0, 255))
    strip.show()
    time.sleep(1)

    rainbow_cycle(0.001) # high speed rainbow cycle w/1ms delay per sweep
```

This code will work with any DotStar-compatible.

DotStars can be driven by any two pins (just slower if they are not hardware pins)

For powering the pixels from the board, the 3.3V regulator output from the Trinket/Gemma M0 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 Gemma M0 this is the **Vout** pad - that pad has direct power from USB or BAT, depending on which is higher voltage. On the Trinket M0 the **USB** or **BAT** pins will give you direct power from the USB port or battery.

The DotStar object's argument list requires the 2 pins you'll use and the number of pixels. There's two optional arguments, `brightness` (range from 0 off to 1.0 full brightness) and auto_write. When `auto_write` default is set to True, where every change is immediately written to the strip of pixels, this is easier to use but *way* slower. if you set `auto_write=False` then you will have to call `strip.show()` when you want to actually write color data out.

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


Verify the wiring on your strip or device - plugging into the 'DOUT' side is a common mistake! Wire up DotStars only while the Trinket/Gemma is not on, to avoid possible damage!

If the power to the pixels 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 DotStar know-how at our DotStar UberGuide https://learn.adafruit.com/adafruit-dotstar-leds

## CircuitPython PWM

As of CircuitPython 2.1 we have `pulseio` support for Trinket, so you can PWM LEDs, control servos, beep piezos, and manage 'pulse train' type devices like DHT22 and Infrared.

On Trinket, you get four PWMs, **D0, D2, D3** and **D4.** Actually, you get a fifth, but that's on the **D13** LED). D1/A0 has true analog out but does not have PWM!

## Timer mapping

There's a limited number of timers available. But timers have many outputs. You have have two PWM outputs that share a timer **but** they must have the same frequency (they can vary the duty cycle just not frequency)

When you create an `pulseio` object, the lowest # Timer that is not already being used, will be used. For example, **D13** can use timer 0 or timer 1, but will default to the lowest timer.

Also, you can **only have two PWM outputs on timer 1**. So if you want all the PWMs, you can put **D0**, **D2** and **D13** on timer 0 and **D3** and **D4** on timer 1.

| Pin name | Timers available |
|---|---|
| **D0** | Timer #0.0 and Timer #1.2 |
| **D2** | Timer #0.1 and Timer #1.3 |
| **D3** | Timer #1.1 |
| **D4** | Timer #1.0 |
| **D13 LED** | Timer #0.2 and Timer #1.0 |

Both **D3** and **D4** are on the same timer so if you want to use both at the same time, they MUST be the same frequency! You can have **D0** or **D2** on Timer #0 so they can be a different frequency form each other but only if **D3**/**D4** aren't being used.

Basically just keep track of which timers you are using if you want to have unique frequencies. If unique freqs are not important to you, then this doesn't matter. CircuitPython will give you an error "All timers in use" if you are trying to arrange the pins in a way that doesn't work

## PWM Output with Fixed Frequency

This sketch demonstrates how to create four PWM outputs, one on each pin.

The frequency for **D3** and **D4** must be the same (1000 hz) but you can vary the duty cycle between the two! You can use **D13**, **D2** and **D0** on timer #0.

Timer #0 and Timer #1 can have unique frequencies, in this case we have 5 KHz for one and 1 KHz for the other.

```
import pulseio
import time
import board

pwm3 = pulseio.PWMOut(board.D3, frequency=1000, duty_cycle=0) # timer 1
pwm4 = pulseio.PWMOut(board.D4, frequency=1000, duty_cycle=0) # timer 1

pwm2 = pulseio.PWMOut(board.D2, frequency=5000, duty_cycle=0) # timer 0
pwm0 = pulseio.PWMOut(board.D0, frequency=5000, duty_cycle=0) # timer 0
led = pulseio.PWMOut(board.D13, frequency=5000, duty_cycle=0) # timer 0

while True:
    for i in range(100):
        # PWM D0 and D3 from low to high duty cycle
        pwm3.duty_cycle = pwm0.duty_cycle = int(i * 65535 / 100)
        # PWM D2 and D4 from high to low
        pwm4.duty_cycle = pwm2.duty_cycle = 65535 - pwm0.duty_cycle
        time.sleep(0.01)
 # 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
```

Create a PWM output with pulseio.PWMOut and pass in the pin to use, then you can set the initial `frequency` and also initial `duty_cycle`!

## 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.

Remember that on the Trinket, some pins share a single timer, so if you want two piezos, for example, make sure they are on two different timers

```
import pulseio
import time
import board

piezo = pulseio.PWMOut(board.D2, 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 =65536//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 installed we have a nice little helper that makes a tone for you on a piezo with a single command:

```
import pulseio
import time
import board
import simpleio

while True:
    for f in (262, 294, 330, 349, 392, 440, 494, 523):
 simpleio.tone(board.D2, f, 0.25)  # on for 1/4 second
 time.sleep(0.05)                  # pause between notes
    time.sleep(0.5)
```

As you can tell, its a lot prettier!

## CircuitPython HID Keyboard

One of the things we baked into CircuitPython is 'HID' control - Keyboard and Mouse capabilities. This means a Trinket or Gemma 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 install the adafruit_hid bundle which comes with Keyboard, Keycode and Mouse support

Then try running this example code which will create 3 'buttons' on three Trinket or Gemma pins

```
# CircuitPlayground demo - Keyboard emu

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

# A simple neat keyboard demo in circuitpython

# The button pins we'll use, each will have an internal pullup
buttonpins = [board.D2, board.D1, board.D0]
# our array of button objects
buttons = []
# The keycode sent for each button, will be paired with a control key
buttonkeys = [Keycode.A, Keycode.B, "Hello World!\n"]
controlkey = Keycode.SHIFT

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

# make all pin objects, make them inputs w/pullups
for pin in buttonpins:
    button = DigitalInOut(pin)
    button.direction = Direction.INPUT
    button.pull = Pull.UP
    buttons.append(button)

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

print("Waiting for button presses")

while True:
    # check each button
    for button in buttons:
        if not button.value:   # pressed?
            i = buttons.index(button)
            print("Button #%d Pressed" % i)

            # turn on the LED
            led.value = True
```

```
            led.value = True

            while not button.value:
                pass  # wait for it to be released!
            # type the keycode or string
            k = buttonkeys[i]    # get the corresp. keycode/str
            if type(k) is 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)
```

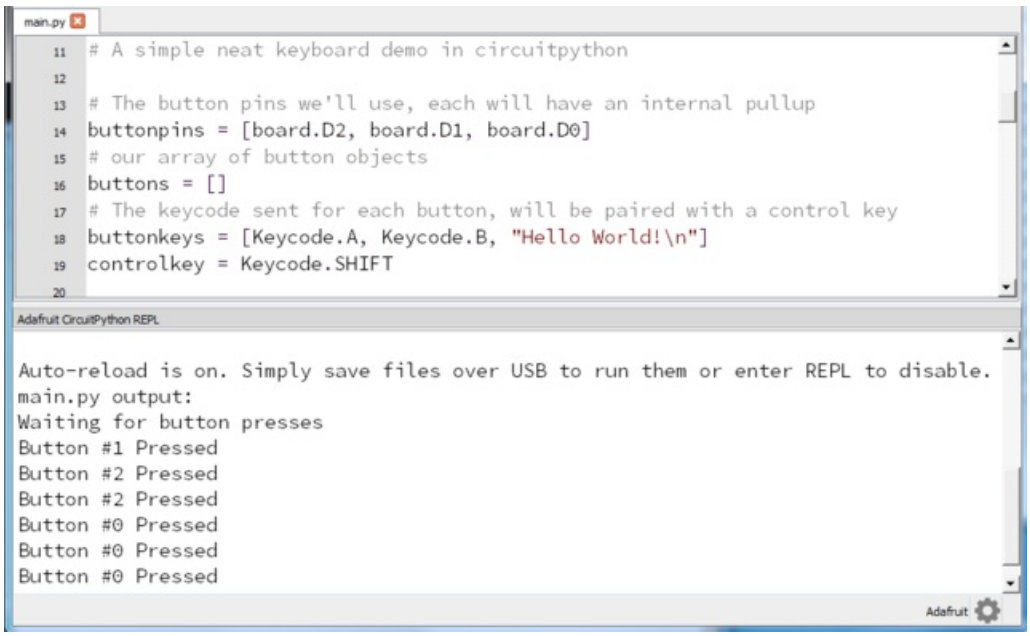
Touch any of the digital IO pads to ground using a wire to have the keypresses sent.

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()
# we're americans :)
layout = KeyboardLayoutUS(kbd)
```

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

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




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

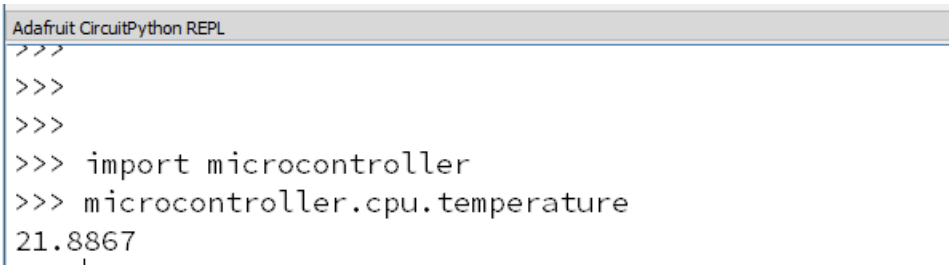
## CircuitPython CPU Temp

This little built in sensor comes with all ATSAMD21 chips, and its really nice to have a temperature sensor so we let you read it via CircuitPython, its new since 2.0.0 and only available on the ATSAMD21-based boards (e.g. not ESP8266)

It's so easy, we'll just give you the two REPL commands

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

That's it! You'll have the temperature in Centigrade printed out. Note it is not *exactly* the same as ambient temperature, and its not super precise. But it's kinda close!



## CircuitPython SPI & SD Card

CircuitPython boards with at least 4 pins can take advantage of a full SPI interface to talk to complex devices like SD cards and color TFT displays.

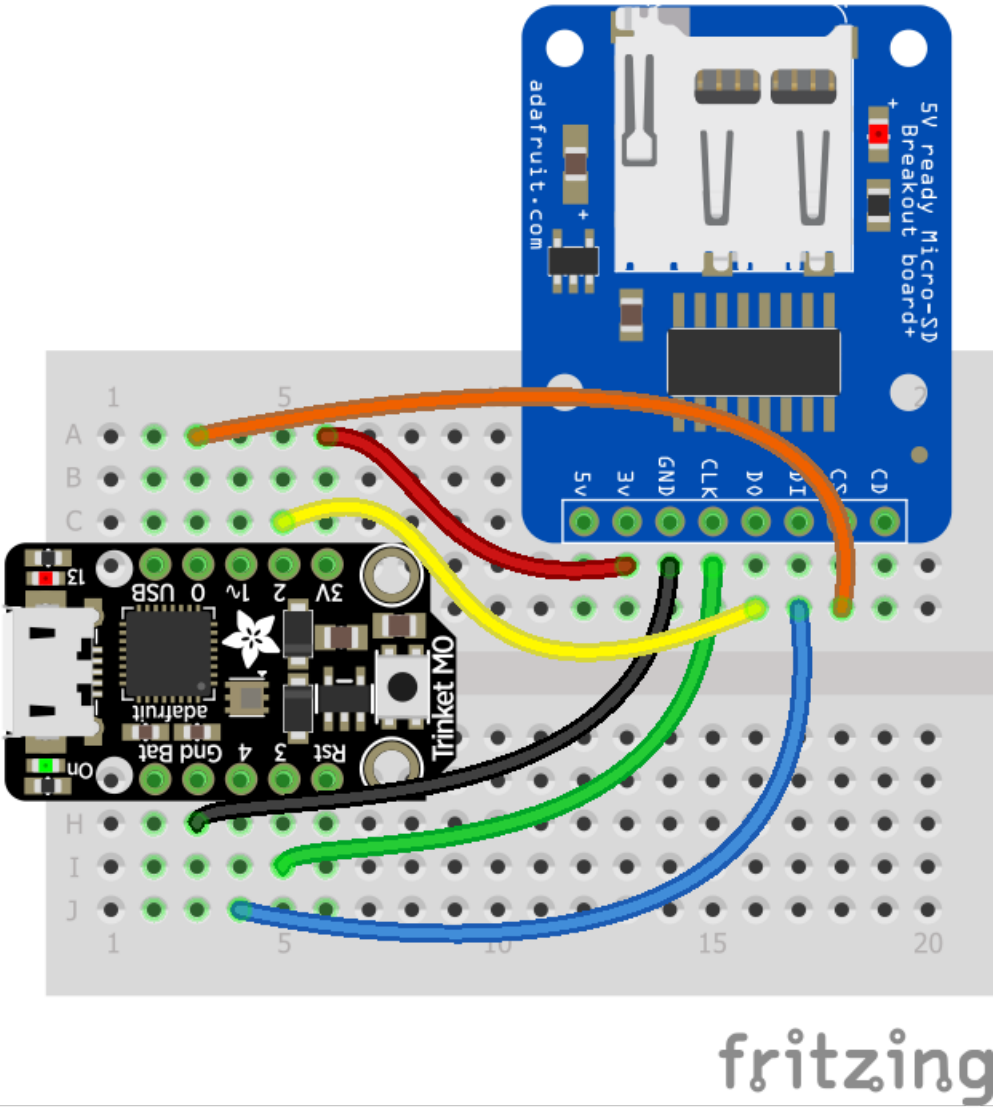
The Gemma M0 only has 3 pads available which means that you can't have a full 4-pin SPI interface. You can still do a 2-wire SPI interface (say, clock and data out for talking to DotStar LEDs) or a 3-wire SPI interface where you have a chip select, clock and then one data line, the MAX31855 for example only needs 3 pins.

But other devices like the Trinket M0 have plenty of pins! You can easily wire up an SD card that lets you log or read data without being restricted to the small internal filesystem on the Trinket.

The Trinket M0 has only one hardware SPI port:

- **SCLK** on **D3**
- **MOSI** on **D4**
- **MISO** on **D2**

SD cards also need a chip select line, but that can be any pin, we'll use **D1**




sdcard.fzz

https://adafru.it/zyB

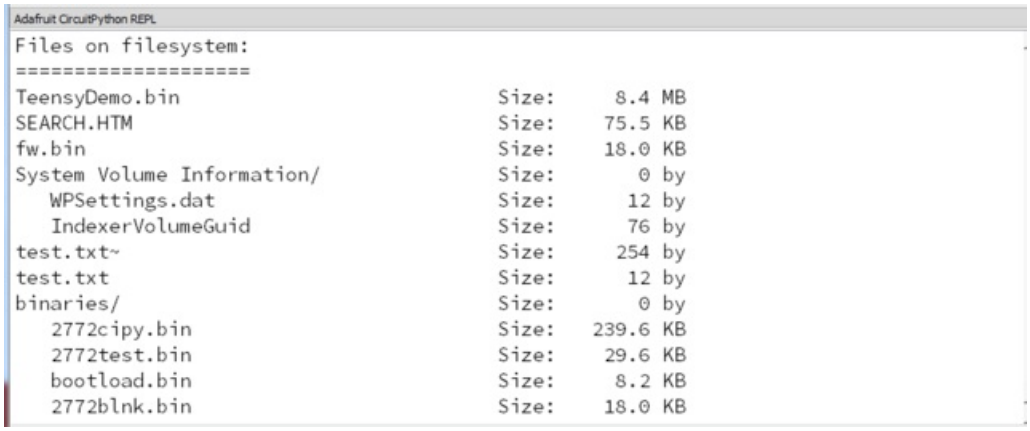
You'll need to install the CircuitPython adafruit_sdcard library file from our driver bundle. Visit the CircuitPython Libraries page for information on how to install it. You'll also need the **adafruit_bus_device** library folder.

## List Files

Once you're ready, load this into **main.py**

```
import adafruit_sdcard
import busio
import digitalio
import board
import storage
import os

# Use any pin that is not taken by SPI
SD_CS = board.D0

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

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

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

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

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

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


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

Once it's loaded up, open up the REPL (and restart it with ^D if necessary) to get a printout of all the files included. We recursively print out all files and also the filesize. This is a good demo to start with because you can at least tell if your files exist!



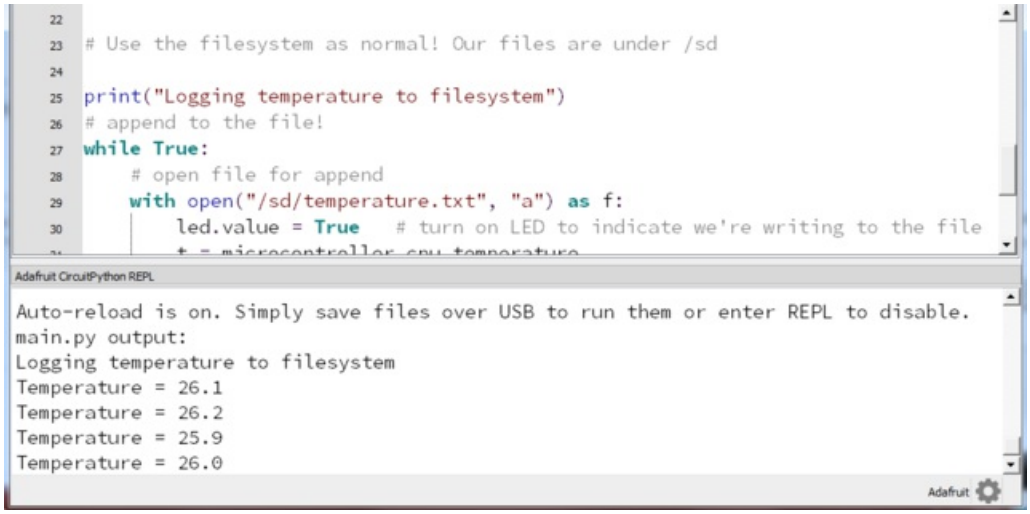

But you probably want to do a little more, lets log the temperature from the chip to a file.

Here's the new script

```
import adafruit_sdcard
import microcontroller
import busio
import digitalio
import board
import storage
import os
import time

# Use any pin that is not taken by SPI
SD_CS = board.D0

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

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

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

print("Logging temperature to filesystem")
# append to the file!
while True:
    # open file for append
    with open("/sd/temperature.txt", "a") as f:
        led.value = True  # turn on LED to indicate we're writing to the file
        t = microcontroller.cpu.temperature
        print("Temperature = %0.1f" % t)
        f.write("%0.1f\n" % t)
        led.value = False  # turn off LED to indicate we're done
    # file is saved
    time.sleep(1)
```

When saved, the Trinket will start saving the temperature once per second to the SD card under the file

temperature.txt




The key part of this demo is in these lines:

```
print("Logging temperature to filesystem")
# append to the file!
while True:
    # open file for append
    with open("/sd/temperature.txt", "a") as f:
        led.value = True   # turn on LED to indicate we're writing to the file
        t = microcontroller.cpu.temperature
        print("Temperature = %0.1f" % t)
        f.write("%0.1f\n" % t)
        led.value = False   # turn off LED to indicate we're done
    # file is saved
    time.sleep(1)
```

This is a slightly complex demo but it's for a good reason. We use **with** (a 'context') to open the file for **a**ppending, that way the file is only opened for the very short time its written to. This is safer because then if the SD card is removed or the board turned off, all the data will be safe(r).

We use the LED to let the person using this know that the temperature is being written, it turns on just before the write and then off right after.

After the LED is turned off the **with** ends and the context closes, the file is safely stored.

## CircuitPython Storage

You have been using that little USB drive to put code on, but maybe you've wondered "Hey can I *write* data from Python to the storage drive to act as a datalogger?" The answer is **yes** (as of CircuitPython 2.0.0)!

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

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

Here is your new **boot.py**:

```
import digitalio
import board
import storage

switch = digitalio.DigitalInOut(board.D0)
switch.direction = digitalio.Direction.INPUT
switch.pull = digitalio.Pull.UP

# If the D0 is connected to ground with a wire
# CircuitPython can write to the drive
storage.remount("/", switch.value)
```

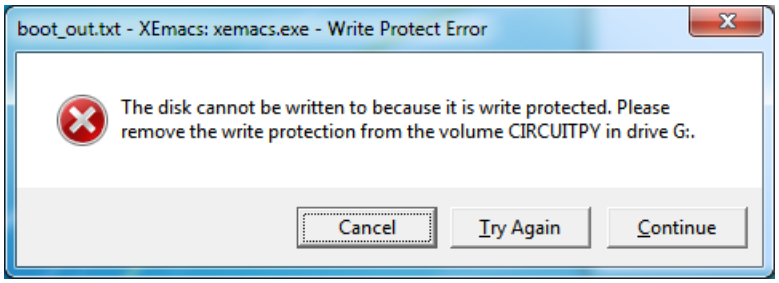
And here is the **main.py**

```
import board
import digitalio
import microcontroller
import time

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

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

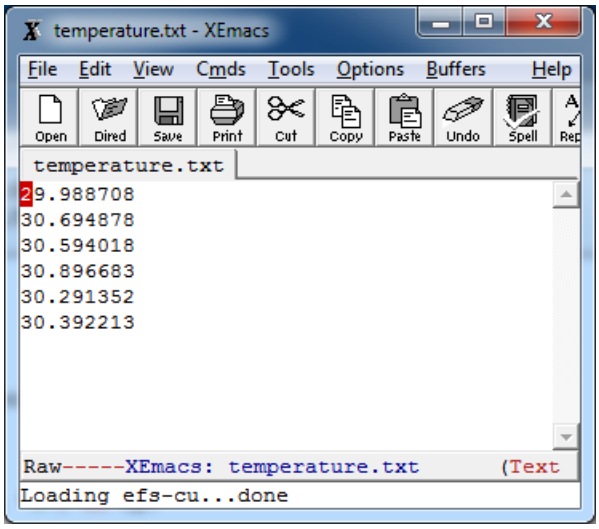
boot.py only runs on first boot of the device, not if you re-start the REPL with ^D or if you save the file, so you must EJECT the USB drive, then physically press the reset button!

Eject & unplug the Trinket or Gemma once you have written these files. Then connect a wire from **D0** to **ground**. This will enable the internal filesystem writing. Now power up the board again.

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




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




This file gets updated once a second but you wont see data come in live. Instead, when you're ready to grab the data, remove the D0 wire and re-plug-in the Trinket/Gemma (or press the reset button). 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 https://learn.adafruit.com/cpu-temperature-logging-with-circuit-python

## Handy Tips

## Check Heap Memory Usage

`import gc`

`gc.mem_free()`

Will give you the number of bytes available for use.

## 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

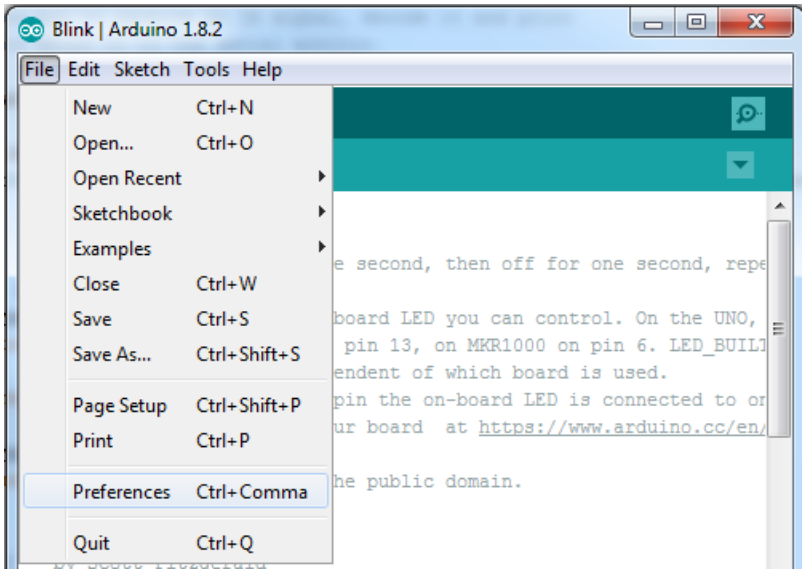
## Arduino IDE Setup

The first thing you will need to do is to download the latest release of the Arduino IDE. You will need to be using **version 1.8** or higher for this guide

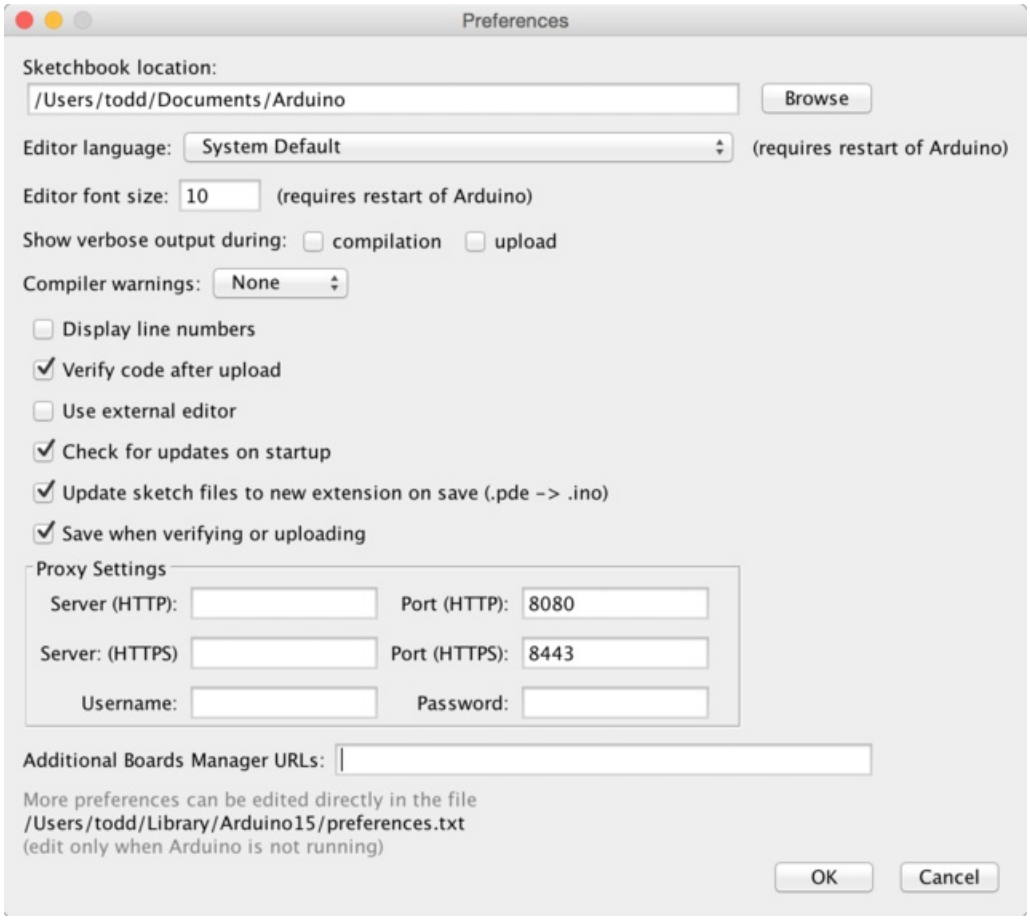
Arduino IDE Download

https://adafru.it/f1P

After you have downloaded and installed **the latest version of Arduino IDE**, you will need to start the IDE and navigate to the **Preferences** menu. You can access it from the **File** menu in *Windows* or *Linux*, or the **Arduino** menu on *OS X*.



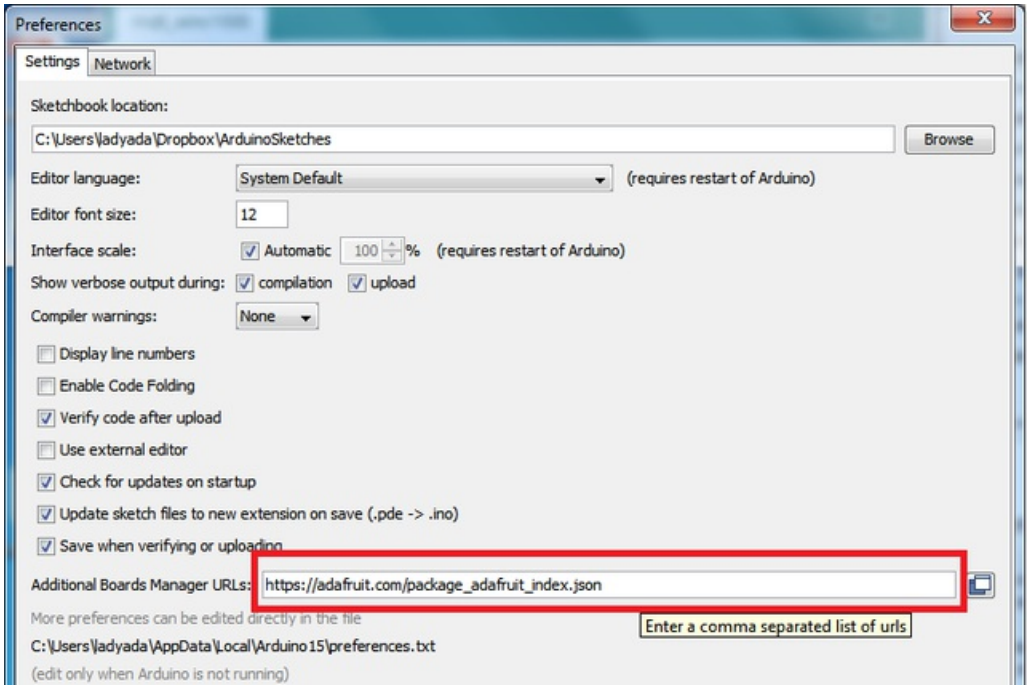

A dialog will pop up just like the one shown below.




We will be adding a URL to the new **Additional Boards Manager URLs** option. The list of URLs is comma separated, and *you will only have to add each URL once.* New Adafruit boards and updates to existing boards will automatically be picked up by the Board Manager each time it is opened. The URLs point to index files that the Board Manager uses to build the list of available & installed boards.

To find the most up to date list of URLs you can add, you can visit the list of third party board URLs on the Arduino IDE wiki. We will only need to add one URL to the IDE in this example, but ***you can add multiple URLS by separating them with commas***. Copy and paste the link below into the **Additional Boards Manager URLs** option in the Arduino IDE preferences.

```
https://adafruit.github.io/arduino-board-index/package_adafruit_index.json
```




Here's a short description of each of the Adafruit supplied packages that will be available in the Board Manager when you add the URL:

- **Adafruit AVR Boards** - Includes support for Flora, Gemma, Feather 32u4, Trinket, & Trinket Pro.
- **Adafruit SAMD Boards** - Includes support for Feather M0, Metro M0, Circuit Playground Express, Gemma M0 and Trinket M0
- **Arduino Leonardo & Micro MIDI-USB** - This adds MIDI over USB support for the Flora, Feather 32u4, Micro and Leonardo using the arcore project.

If you have multiple boards you want to support, say ESP8266 and Adafruit, have both URLs in the text box separated by a comma (,)

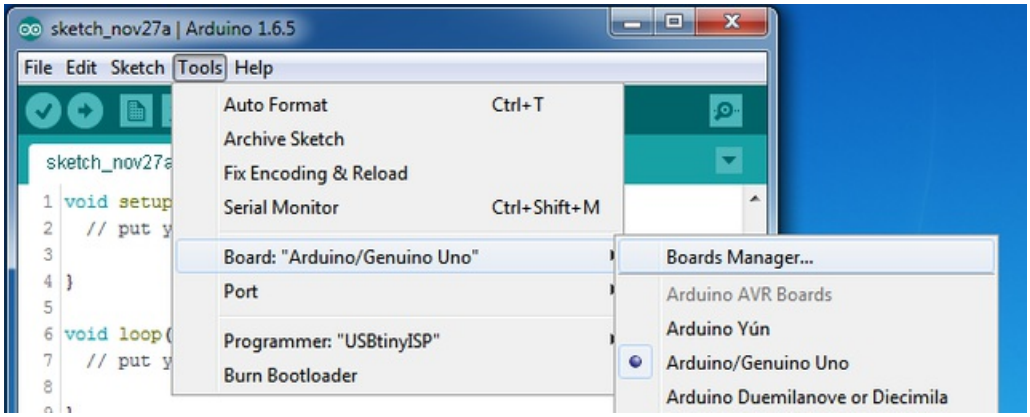
Once done click **OK** to save the new preference settings. Next we will look at installing boards with the Board Manager.

Now continue to the next step to actually install the board support package!

## Using with Arduino IDE

Since the Feather/Metro/Gemma/Trinket M0 use 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.

Now that you have added the appropriate URLs to the Arduino IDE preferences in the previous page, you can open the **Boards Manager** by navigating to the **Tools->Board** menu.

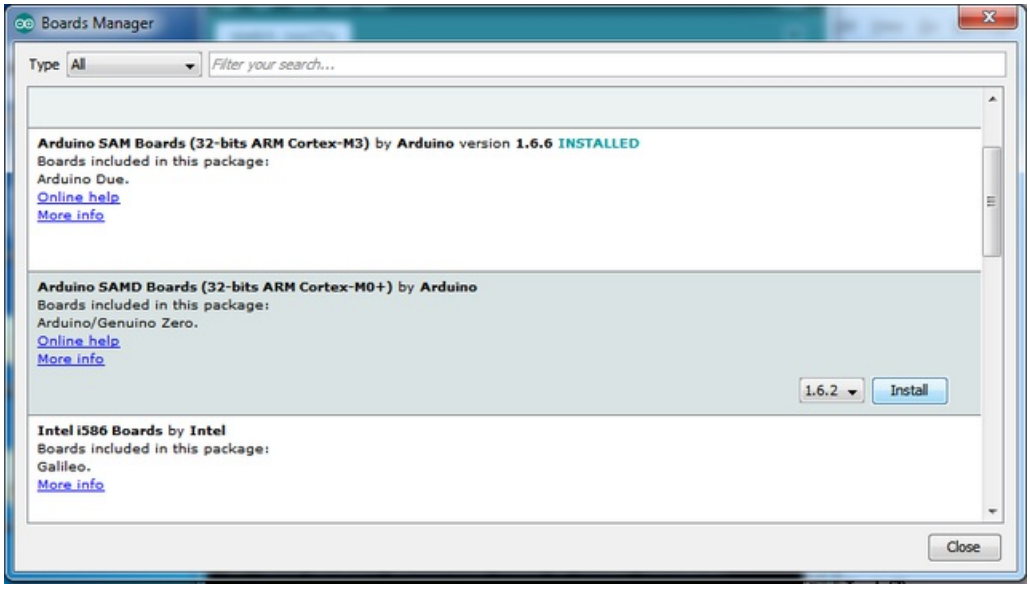



Once the Board Manager opens, click on the category drop down menu on the top left hand side of the window and select **Contributed**. You will then be able to select and install the boards supplied by the URLs added to the prefrences.

## Install SAMD Support

First up, install the **Arduino SAMD Boards** version **1.6.15** or later

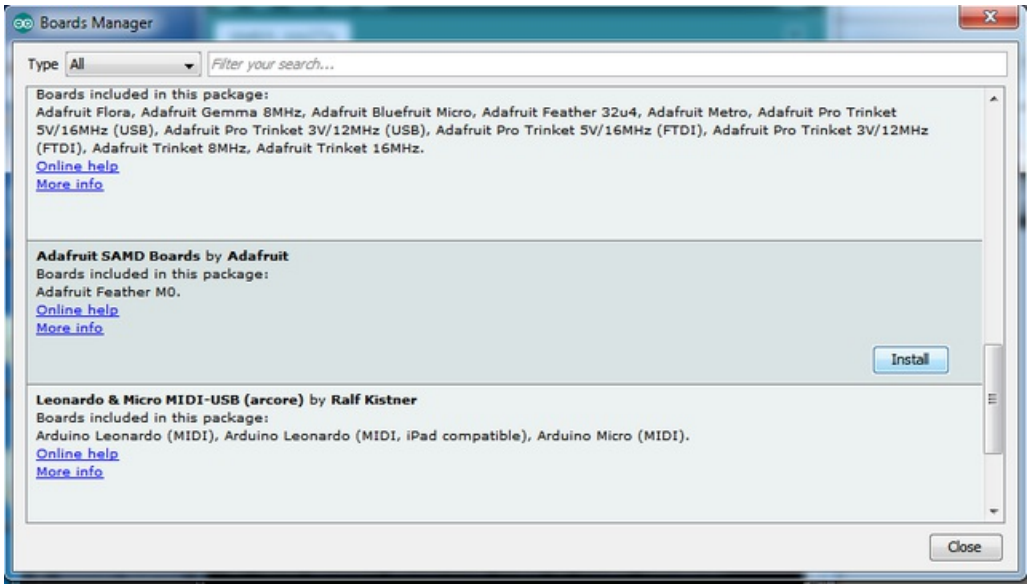
You can type **Arduino SAMD** in the top search bar, then when you see the entry, click **Install**




## Install Adafruit SAMD

Next you can install the Adafruit SAMD package to add the board file definitions

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

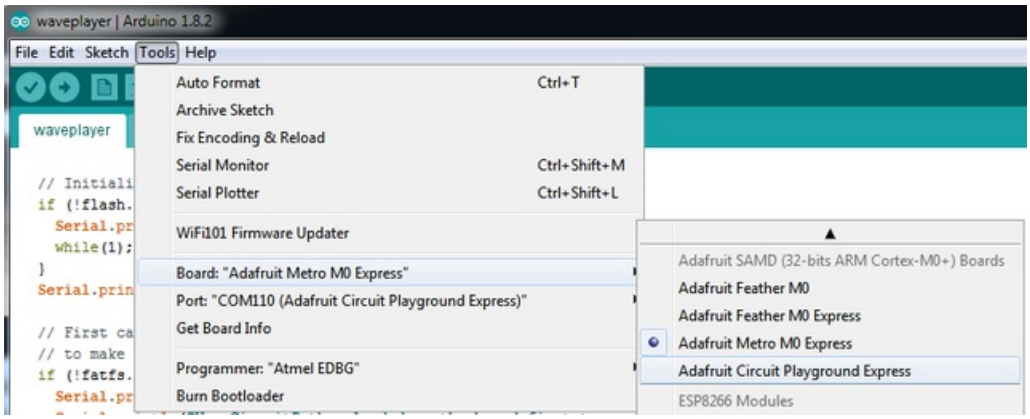



Even though in theory you don't need to - I recommend rebooting the IDE

**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 matching board, the current options are:

- **Feather M0** (for use with any Feather M0 other than the Express)
- **Feather M0 Express**
- **Metro M0 Express**
- **Circuit Playground Express**
- **Gemma M0**
- **Trinket M0**




## Install Drivers (Windows 7 Only)

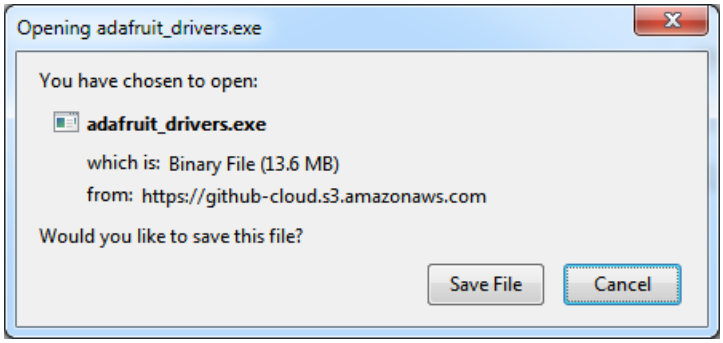
When you plug in the board, you'll need to possibly install a driver

Click below to download our Driver Installer

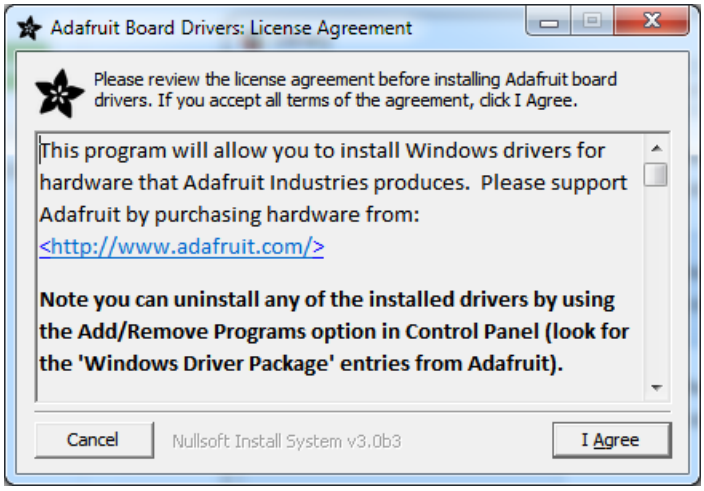
Download Adafruit Driver Installer v2.0.0.0

https://adafru.it/zek

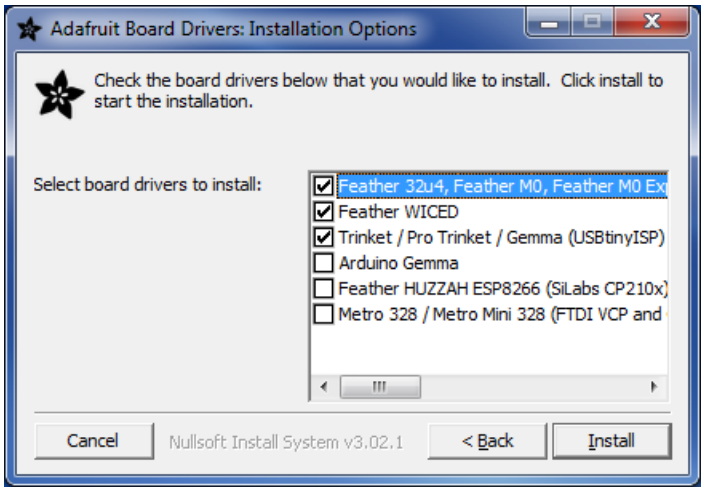
Download and run the installer



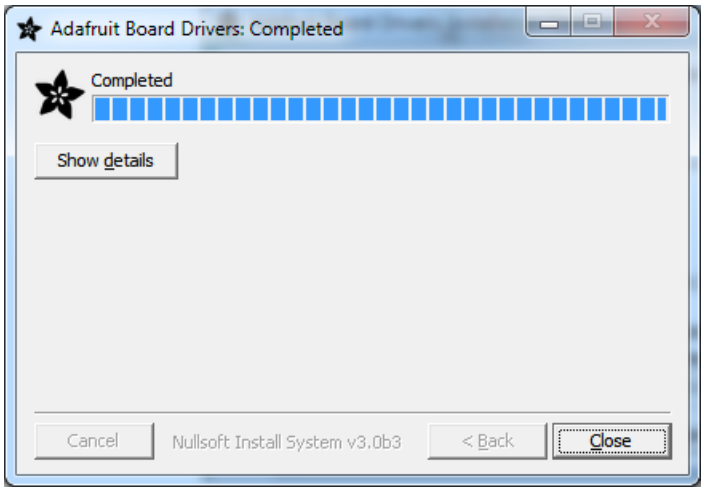

Run the installer! Since we bundle the SiLabs and FTDI drivers as well, you'll need to click through the license




Select which drivers you want to install, the defaults will set you up with just about every Adafruit board!



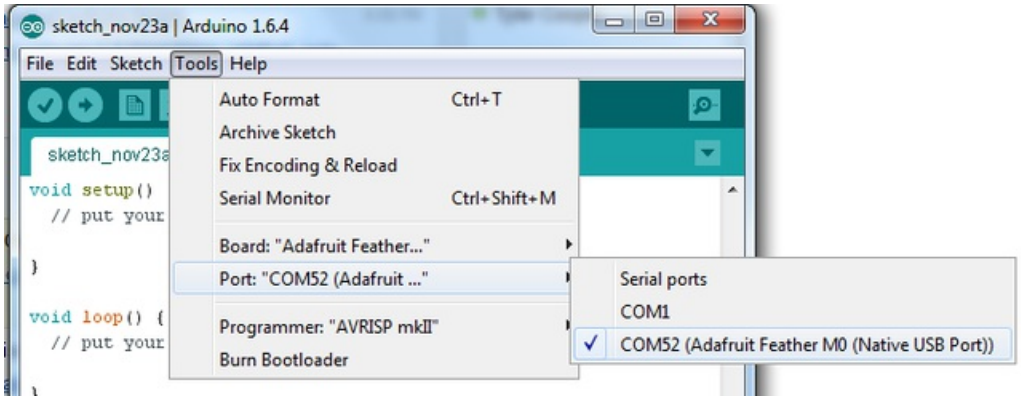

Click **Install** to do the installin'




## Blink

Now you can upload your first blink sketch!

Plug in the Gemma M0, Trinket M0, Metro M0 or Feather M0 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 Trinket/Gemma/Metro/Feather M0!




Now load up the Blink example

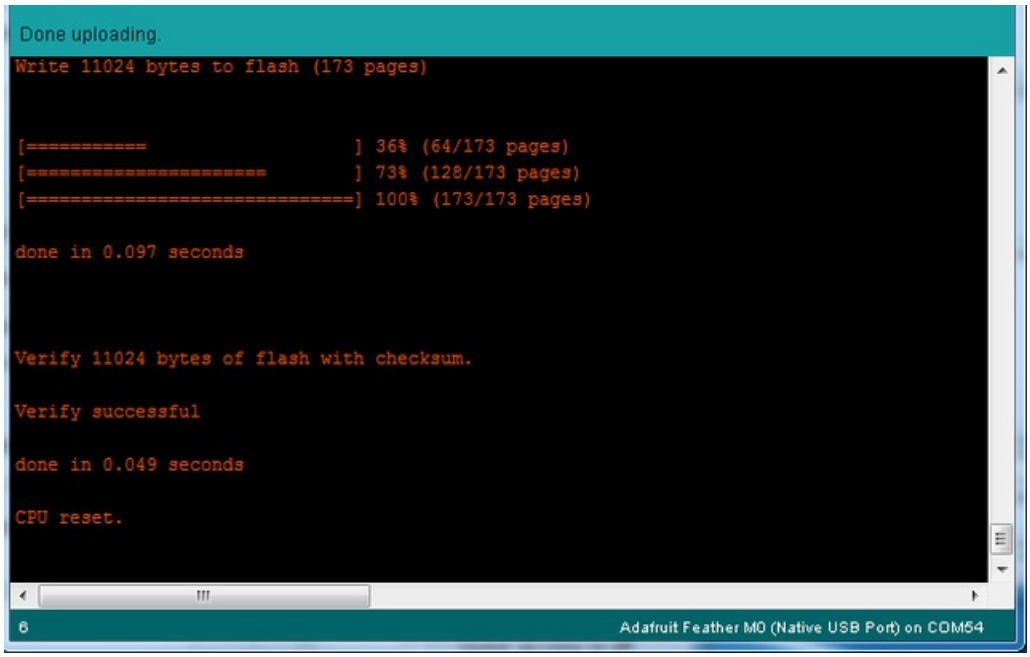
```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// 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

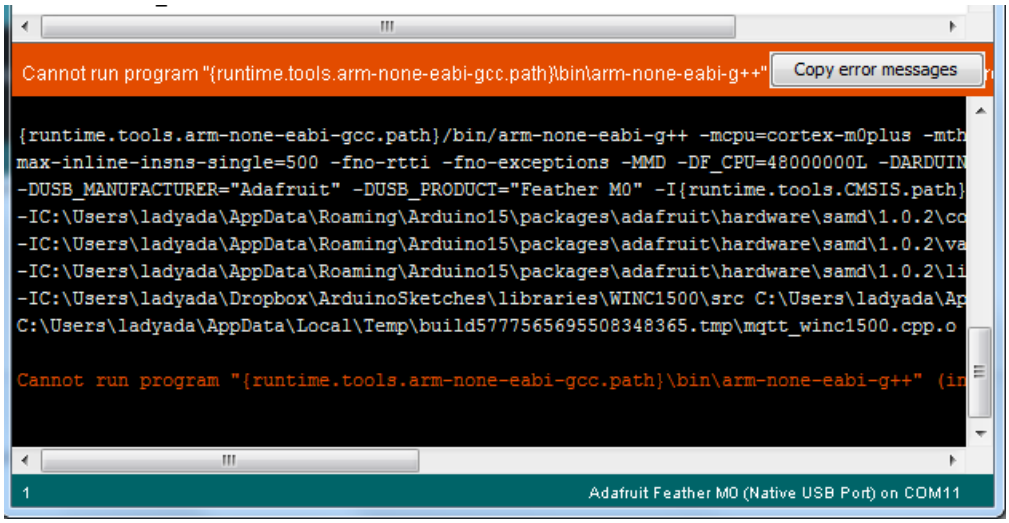



## Compilation Issues

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, you need *both* Arduino & Adafruit SAMD board packages

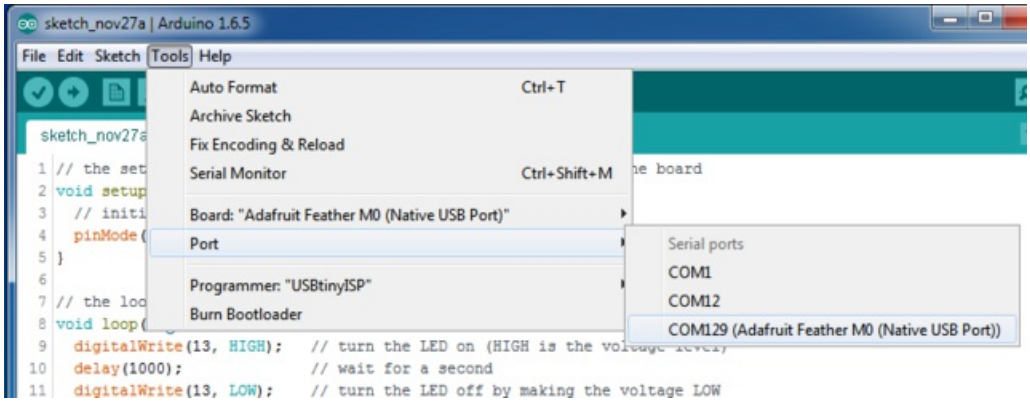


## Manually bootloading

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 **RST** button **twice** (like a double-click)to get back into the bootloader.

**The red LED will pulse, so you know that its in bootloader mode.**

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




You may need to go back and reselect 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.

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.

## Adapting Sketches to M0

The ATSAMD21 is a very nice little chip but its fairly new as Arduino-compatible cores go. **Most** sketches & libraries will work but here's a few things we noticed!

The below note are for all M0 boards, but not all may apply (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 is the same as the output-selection register.

For the M0, you can't do this anymore! Instead, use

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

which has the benefit of being backwards compatible with AVR.

## 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 Core, we fixed it so that Serial goes to USB when you use a Feather M0 so it will automatically work just fine.

**However, on the off chance you are using the official Arduino SAMD core not the Adafruit version (which really, we recommend you use our version because as you can see it can vary) & 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)`

## 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_AR
 #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, 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

## 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 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

## 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 Express and Gemma/Trinket M0 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.

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, then check out the UF2 file format specification. Visit Adafruit's fork of the Microsoft UF2-samd bootloader GitHub repository for source code and releases of pre-built bootloaders.

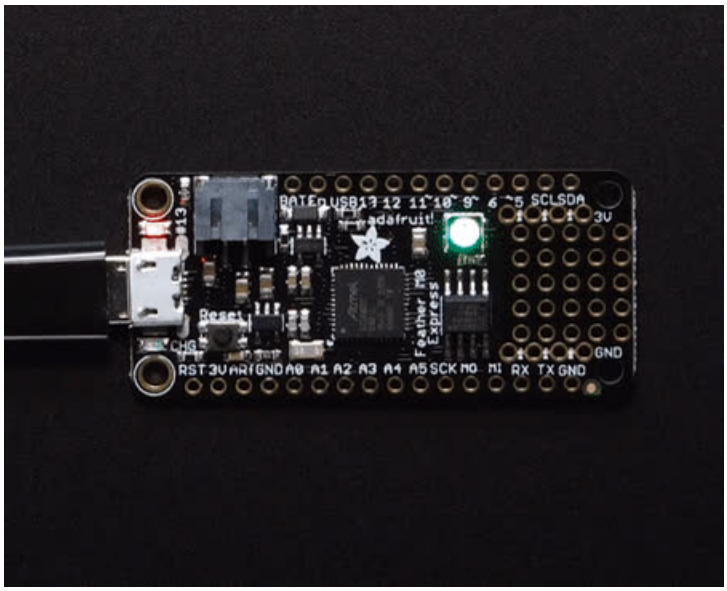
The bootloader is not needed when changing your CircuitPython code. Its only needed when upgrading the CircuitPython core or changing between CircuitPython, Arduino and Microsoft MakeCode.

## 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**

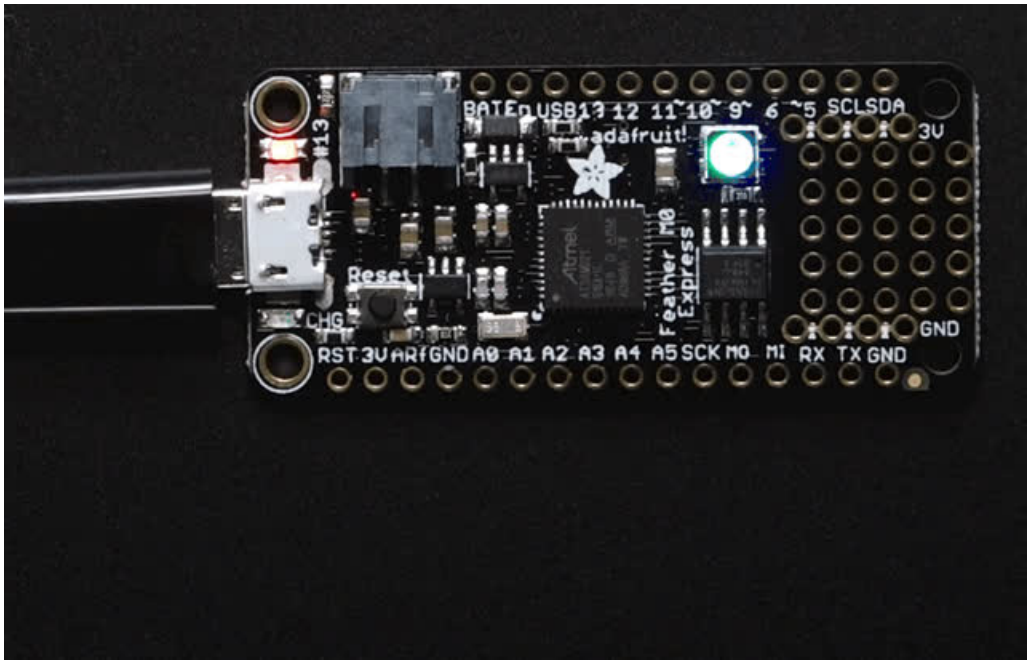
Furthermore, when the bootloader is active, it will change the color of one or more onboard neopixels to indicate the connection status, red for disconnected and green for connected. If the board is plugged in but still showing that its disconnected, try a different USB cable. Some cables only provide 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.

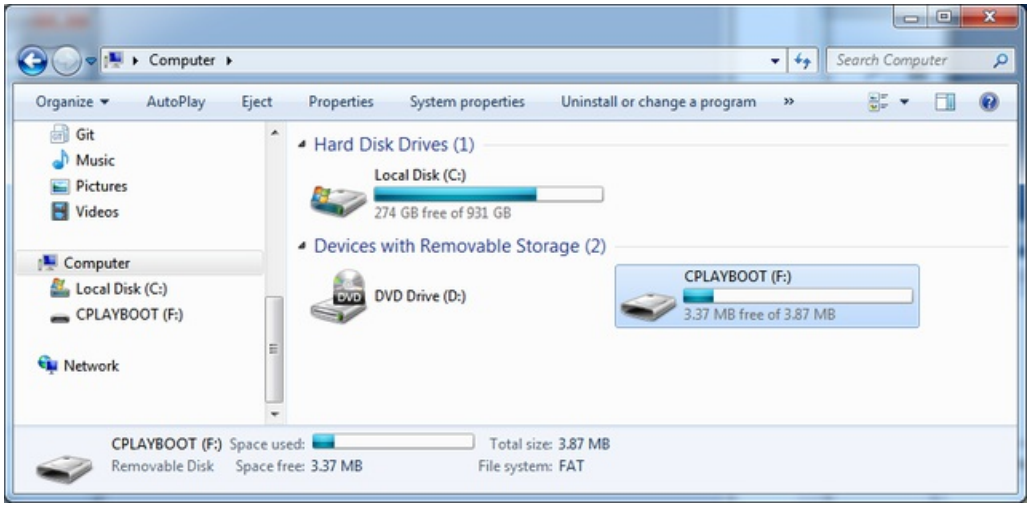



If the bootloader couldn't start, you will get a red NeoPixel LED.

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!

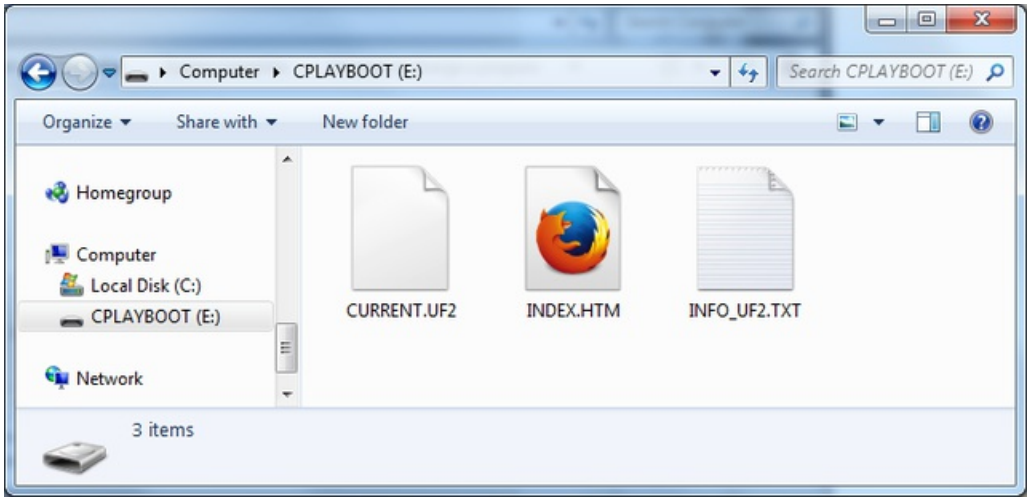



Once the bootloader is running, check your computer. You should see a USB Disk drive...



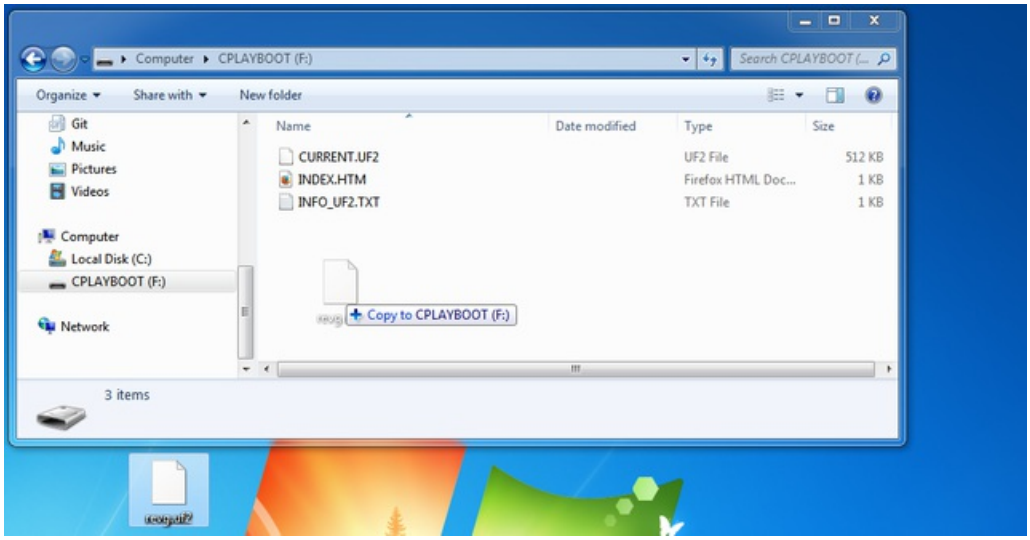

Once the bootloader is successfully connected you can open the drive and browse the virtual filesystem. This isn't the same filesystem as you use with CircuitPython or Arduino. It should have three files:

- **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.

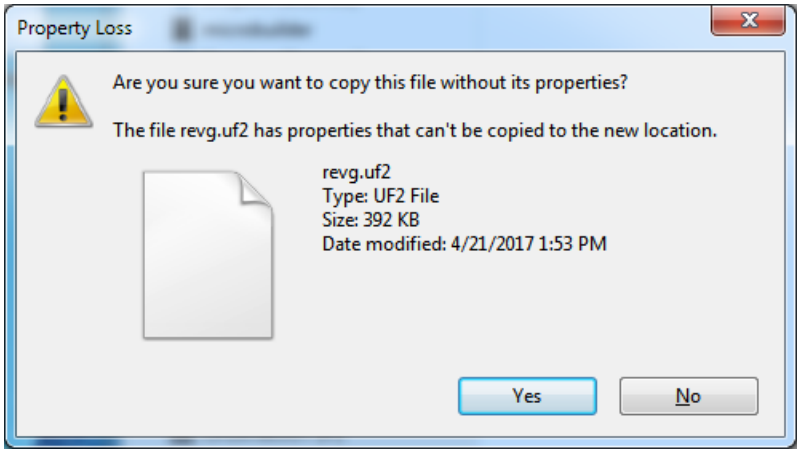



## Using the Mass Storage Bootloader

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 atShumaTech.

### Windows 7 Drivers

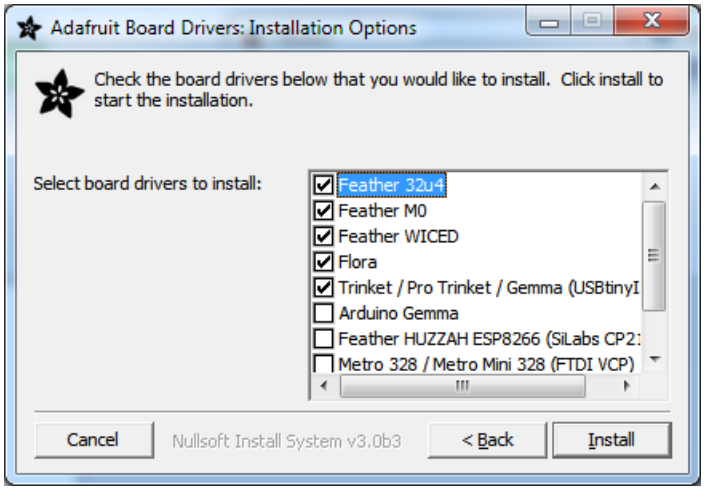
If you are running Windows 7 (or, goodness, something earlier?) You will need a Serial Port driver file. Windows 10 users do not need this so skip this step.

You can download our full driver package here:

Download Latest Adafruit Driver Installer

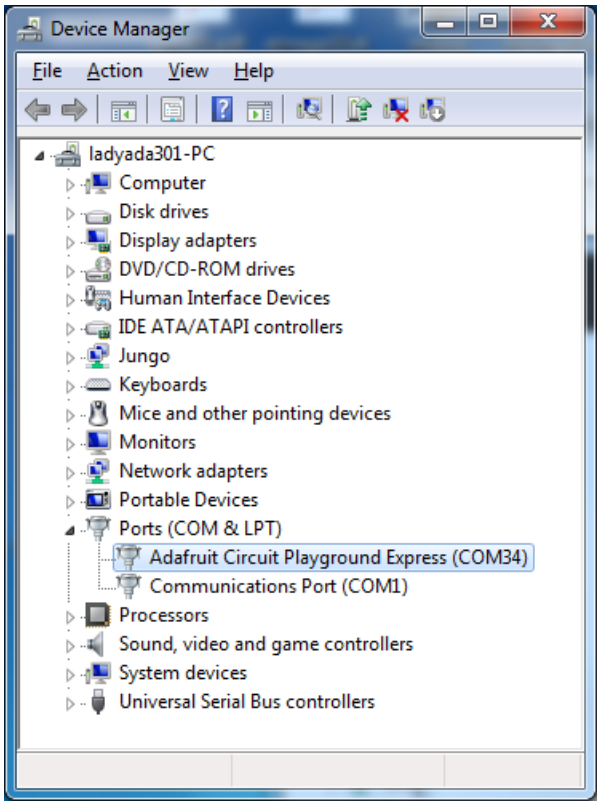
https://adafru.it/A0N

Download and run the installer. We recommend just selecting all the serial port drivers available (no harm to do so) and installing them.

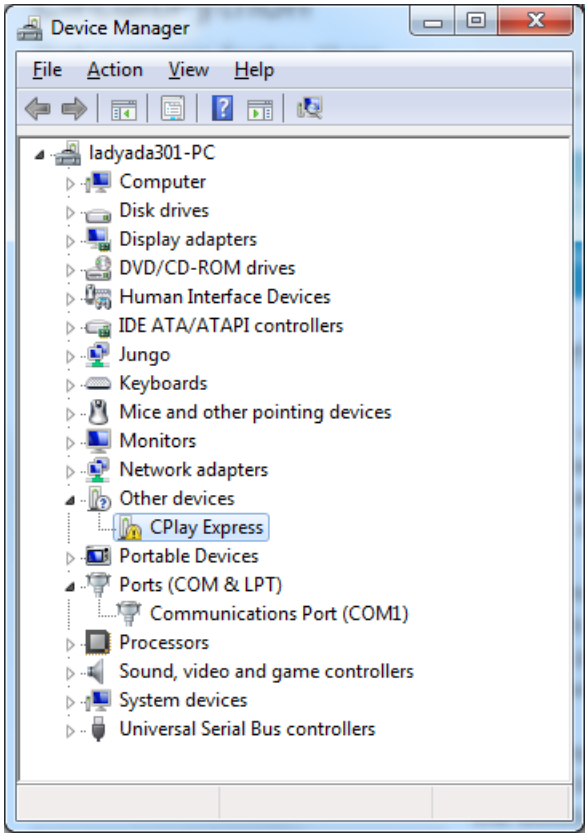



## Verifying Serial Port in Device Manager

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

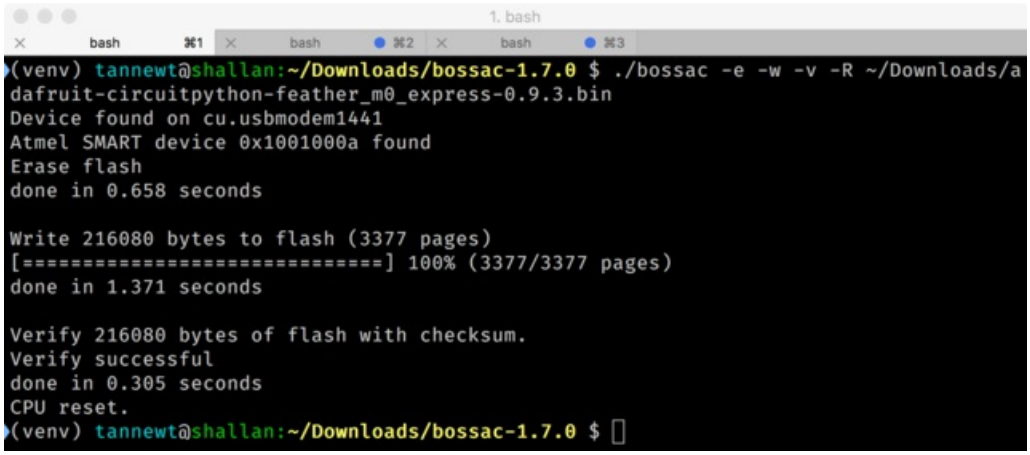
If you are using the Arduino IDE, this step is not required. But sometimes you want to read/write custom binary files, say for loading CircuitPython or your own code. We recommend using bossac v 1.7.0 (or greater), which has been tested. The Arduino branch is most recommended.

You can download the latest builds here. 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`

For example here's the command line you probably want to run:

```
bossac -e -w -v -R ~/Downloads/adafruit-circuitpython-feather_m0_express-0.9.3.bin
```

This will `-e` rase the chip, `-w` rite the given file, `-v` erify the write and `-R` eset the board. After reset, CircuitPython should be running. Express boards may cause a warning of an early eject of a USB drive but just ignore it. Nothing important was being written to the drive. A hard power-reset is also recommended after **bossac**, just in case.




## Updating the bootloader

The UF2 bootloader is a new bootloader, 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.

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 v1.20.0 SFHR
Model: Adafruit Feather M0
Board-ID: SAMD21G18A-Feather-v0
```

Lastly, reload your code from Arduino or MakeCode or flash the latest CircuitPython core.

The latest updaters for various boards:

Circuit Playground Express v1.23 update-bootloader.uf2

https://adafru.it/yDv

Feather M0 Express v1.23 update-bootloader.uf2

https://adafru.it/yDw

Metro M0 Express v1.23 update-bootloader.uf2

https://adafru.it/yDx

Gemma M0 v1.23 update-bootloader.uf2

https://adafru.it/yDy

Trinket M0 v1.23 update-bootloader.uf2

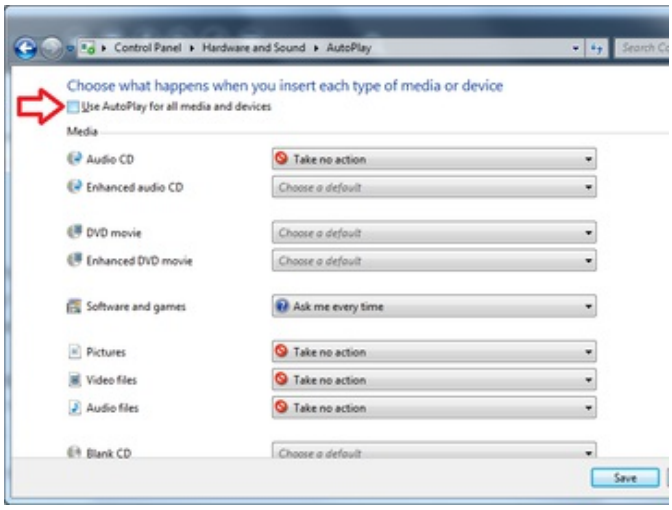
https://adafru.it/yDz

## 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 andthe Python conversion script. Make sure that your program was compiled to start at 0x2000 (8k) because the bootloader takes the first 8k. CircuitPython's linker script 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:

```
uf2conv.py -c -o build-circuitplayground_express/revg.uf2 build-circuitplayground_express/revg.bin
```

This will produce a revg.uf2 file in the same directory as the source revg.bin. The uf2 can then be flashed in the same way as above.

## Downloads

## Files:

- ATSAMD21 Datasheet
- Webpage for the ATSAMD21E18 (main chip used)
- EagleCAD files on GitHub
- Fritzing object in Adafruit Fritzing library

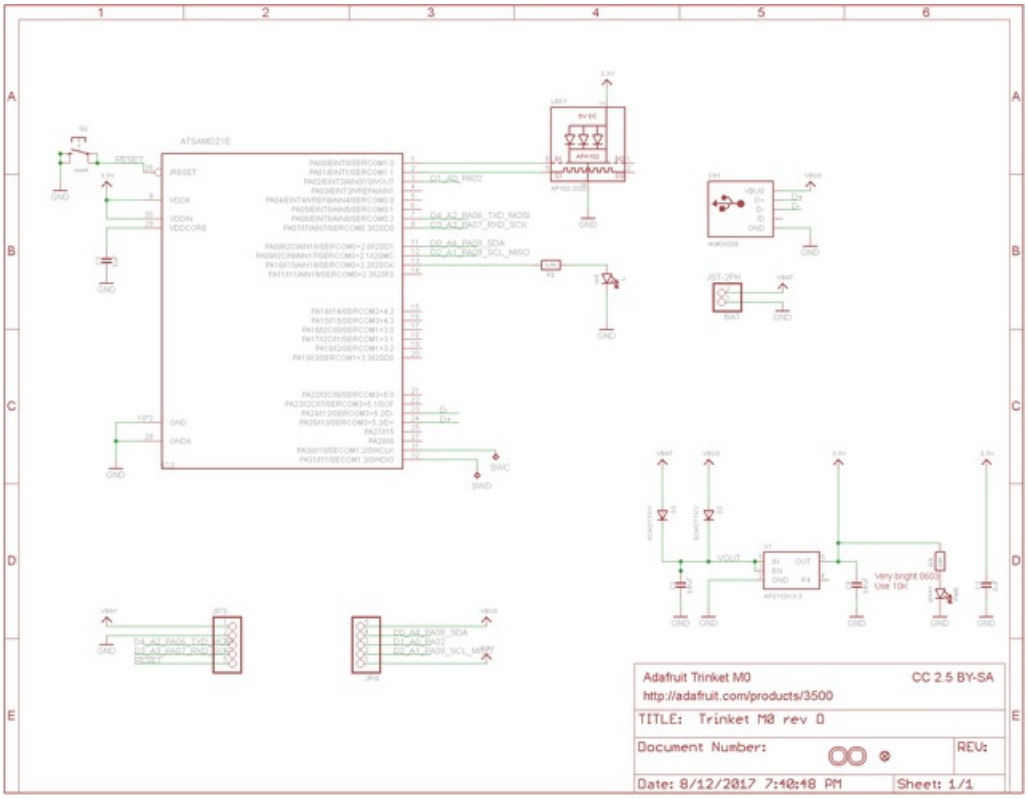
Default CircuitPython files included with v2.0.0

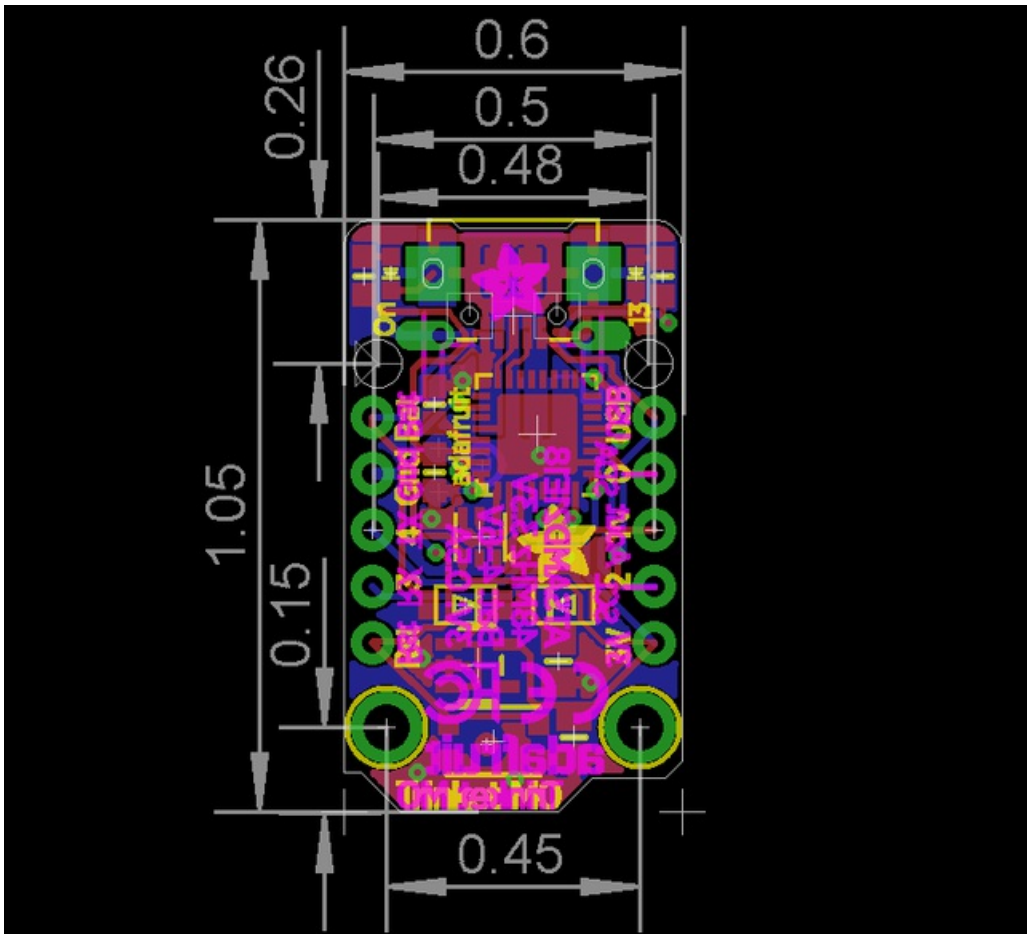
https://adafru.it/zdF

Default CircuitPython files included with v1.0.0

https://adafru.it/yDB

## Schematic & Fabrication Print




0.6
0.5
0.48
0.26
1.05
0.15
0.45