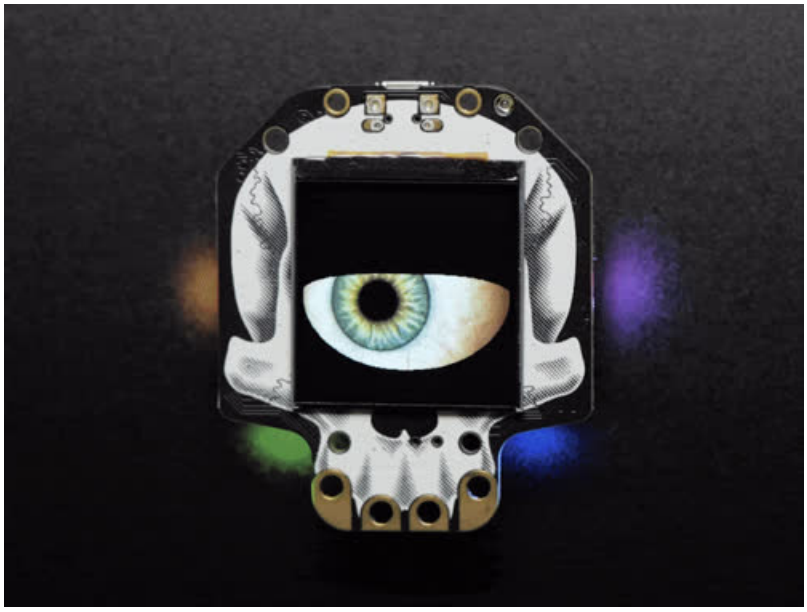
adafruit learning system

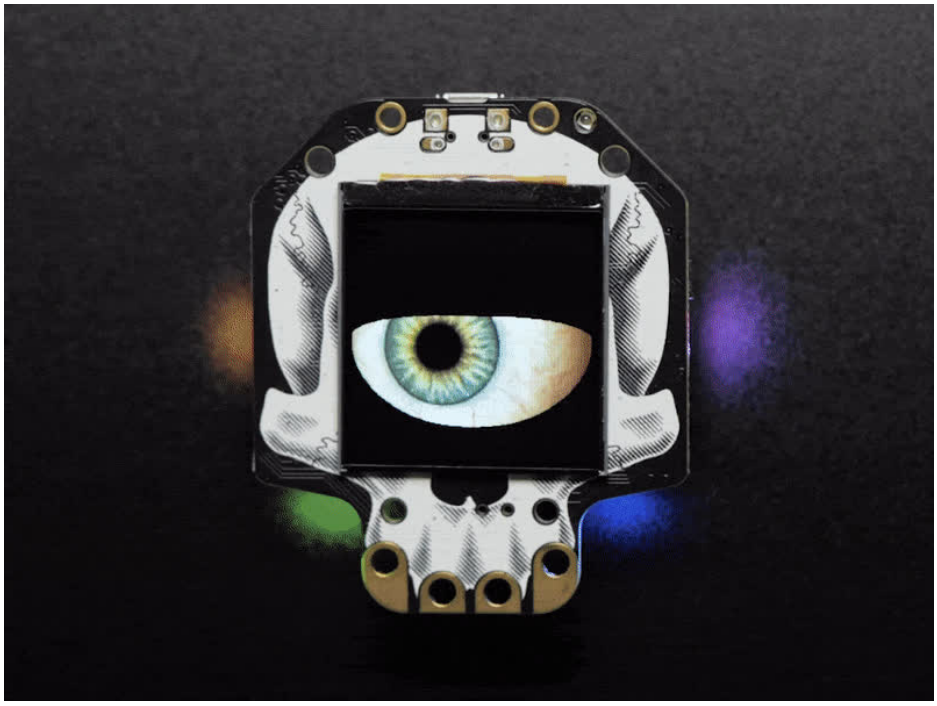
Adafruit Hallowing M4

Created by Kattni Rembor



Last updated on 2019-10-12 12:55:57 PM UTC

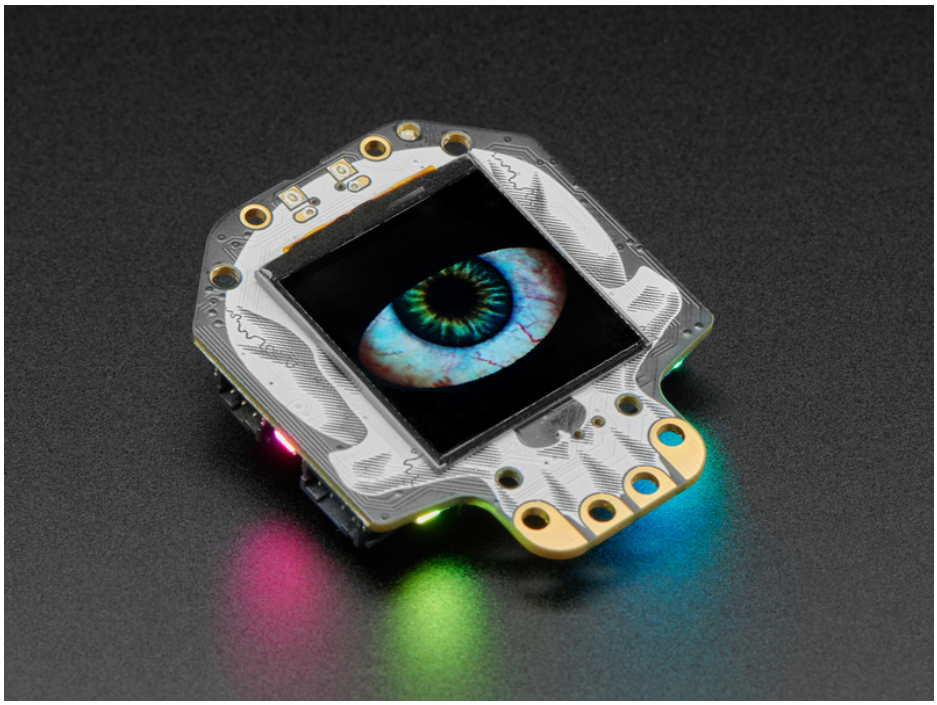
Overview



This is Hallowing..this is Hallowing... Hallowing! Hallowing! (https://adafru.it/C8m)

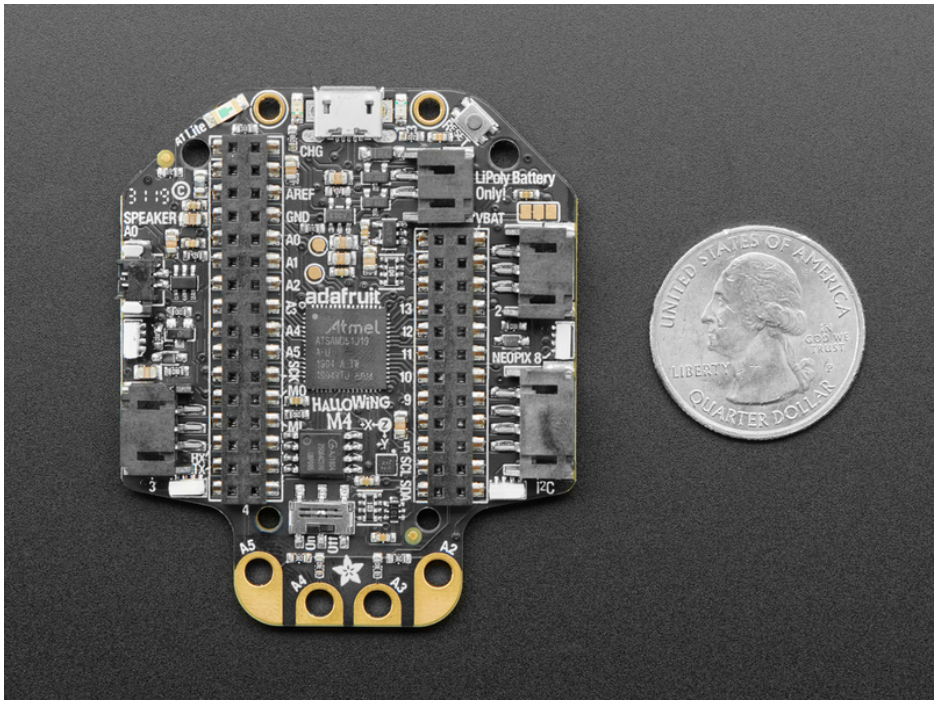
Following up on 2018's most-successful-skull-shaped development board (https://adafru.it/CmY), we UPPED our -skull-shaped development board game, and re-spinned (re-spun?) the HalloWing M0 (https://adafru.it/CmY) into the HalloWing M4 with MORE of everything that makes this the spoooookiest dev board.

Are you the kind of person who doesn't like taking down the skeletons and spiders until after January? Well, we've got the development board for you. This is electronics at its *most spooky!* The **Adafruit HalloWing M4** is a skull-shaped ATSAM521 board with a ton of extras built in to make for an adorable wearable, badge, development kit, or the engine for your next cosplay or prop.



On the front is a cute **1.54" sized 240x240 full color IPS TFT.** Compared to the HalloWing M0's 1.44" 128x128, this has 4x as many pixels and is IPS for great color and brightness. Our default example code has our new fully-customizable spooky eye demo (https://adafru.it/FPD) running but you can use it for anything you like to display in glorious color.

There's also 4 fang-teeth below the display, these are analog/capacitive touch inputs with big alligator-clip holes.

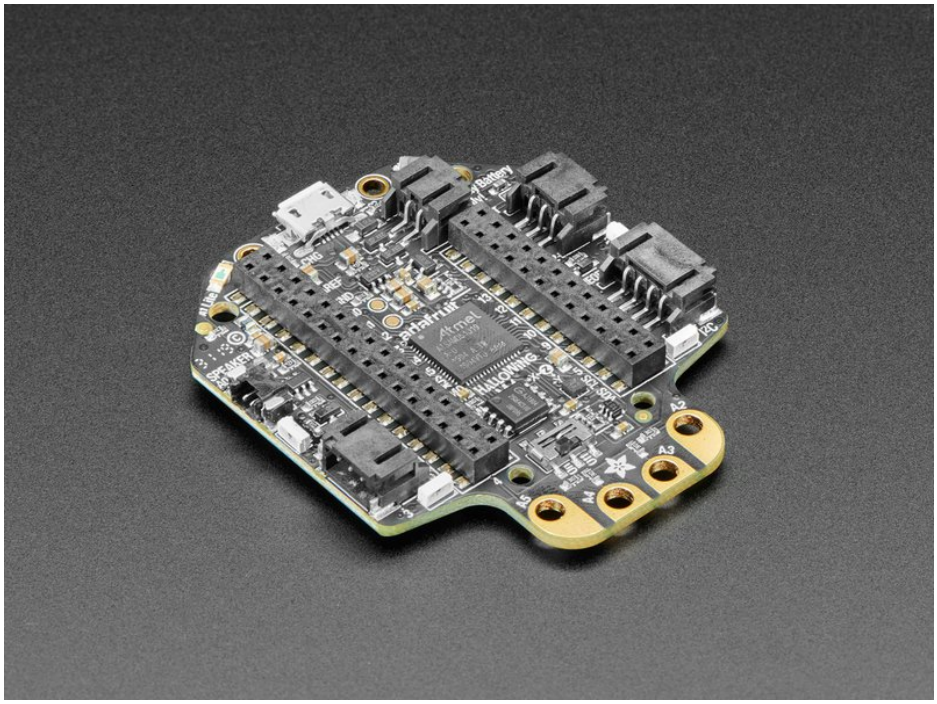


On the reverse is a smorgasbord of electronic goodies:

- **ATSAMD51G18** @ 120MHz with 3.3V logic/power - 512KB of FLASH + 192KB of RAM, can run Arduino or CircuitPython super fast

- **8 MB of SPI Flash** for storing images, sounds, animations, whatever!
- **3-axis accelerometer** (motion sensor)
- **Light sensor**, reverse-mount so that it points out the front
- **Mono Class-D speaker driver** for 4-8 ohm speakers, up to 1 Watt, connected to a 12-bit DAC on the SAMD51
- **Four side-light NeoPixel LEDs** for cool underlighting effects
- LiPoly battery port with built in recharging capability
- USB port for battery charging, programming and debugging
- Two female header strips with Feather-compatible pinout so you can plug any FeatherWings in
- JST ports for Neopixels, sensor input, and I2C (you can fit I2C Grove connectors in here)
- 3.3V regulator with 500mA peak current output
- Reset button
- On-Off switch

OK so technically it's more like a really tricked-out Feather M4 Express than a *Wing* but we simply could not resist the HalloWing pun.

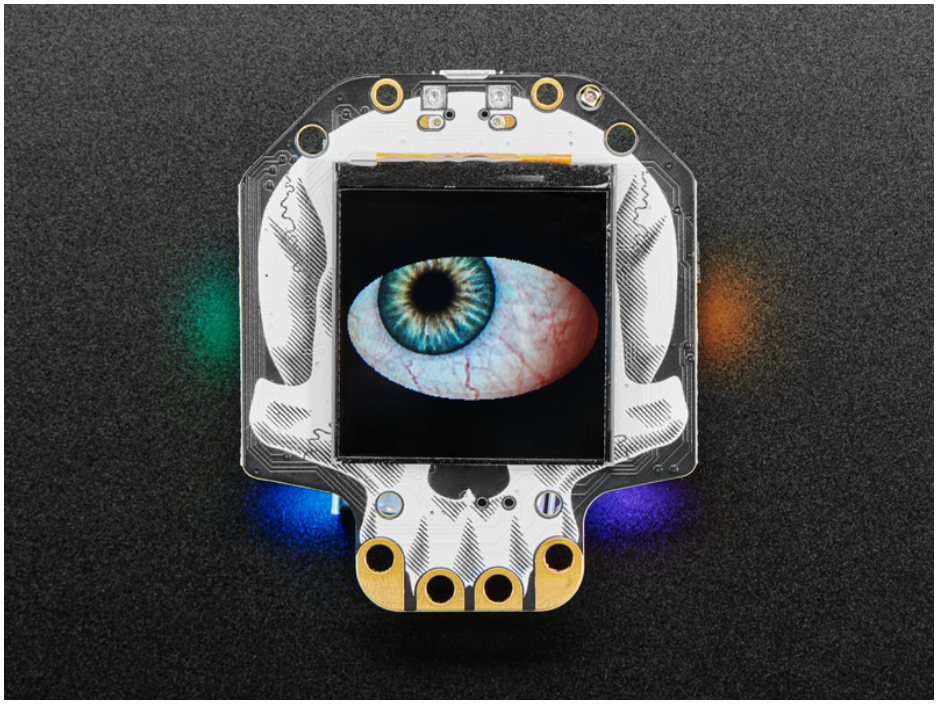


You can use the Hallowing similarly Feather M4 Express, it's got the same chip although the pins have been rearranged. We've got both Arduino and CircuitPython build support for it so you can pick your favorite development language! The extra 8 MB of SPI Flash is great for sound effects projects where you want to play up to 3 minutes of WAV files.

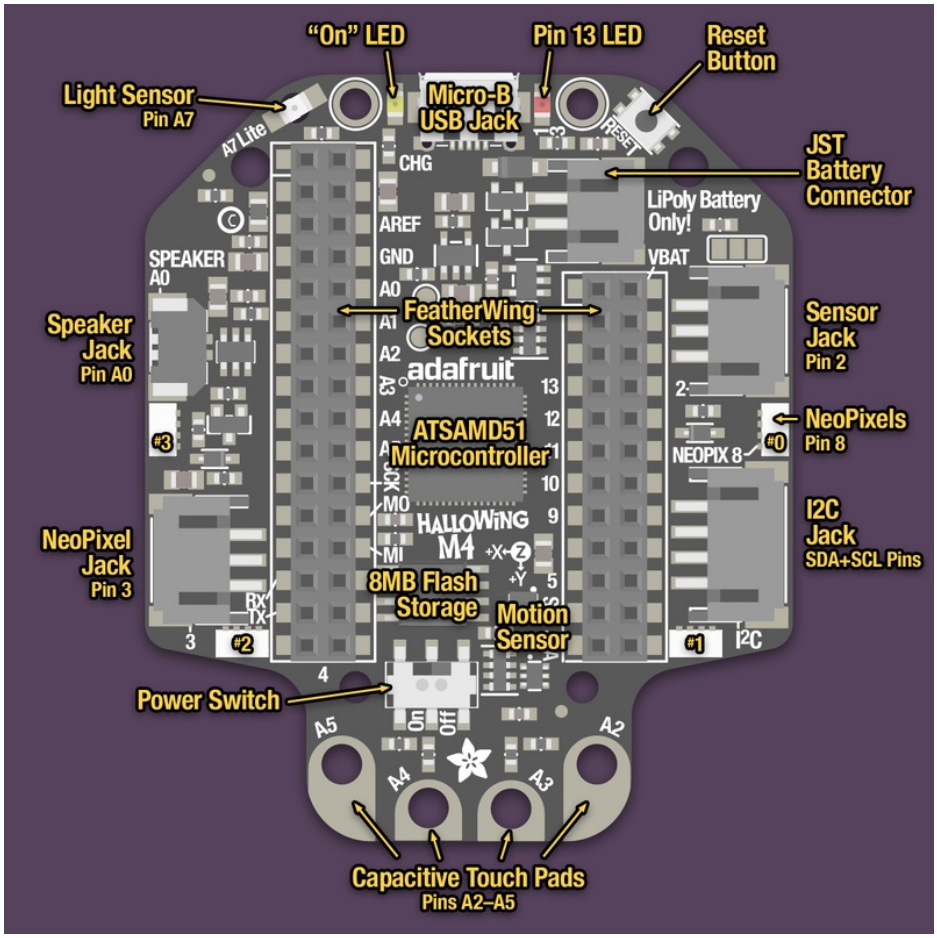
On each side of the Hallowing are JST-PH plugs for connecting external devices. The 3-pin JSTs connect to analog pins on the SAMD21, so you can use them for analog inputs. We label one for NeoPixel and one for Sensors since we think most people will have one of each. The 4-pin JST connector connects to the I2C port and you can fit Grove connectors in it for additional hardware support.

Does not come with a Lipoly battery! We recommend our 350mAh (https://adafru.it/kBj) or 500mAh batteries (https://adafru.it/drL) but any 3.7/4.2V Adafruit Lipoly will do the trick.

Comes fully assembled and ready to be your spooky skull friend. We install the UF2 bootloader on it so updating code and converting it to CircuitPython is easy

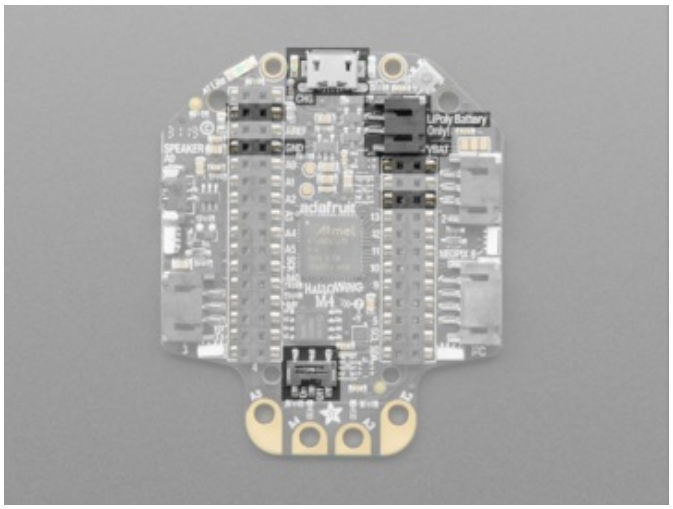


Pinouts



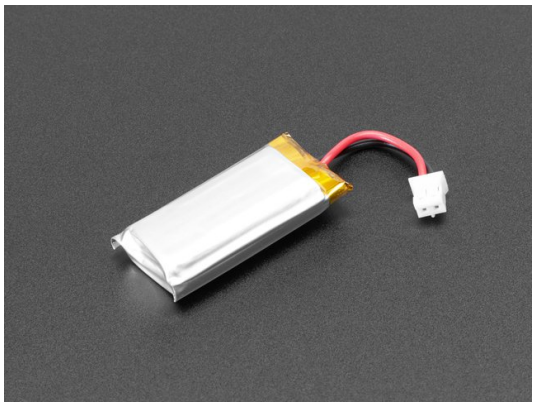
We put a ton of stuff on this HalloWing M4, above you can see a 'guided tour' of whats available!

Power Pins and Ports



There's two ways to power your Hallowing. The best way is to plug in a 3.7/4.2V Lipoly battery into the **JST 2-PH port**. You can then recharge the battery over the Micro USB jack. You can also just run the board directly from **Micro USB**, it will automatically 'switch over' to USB power when that's plugged in

The Hallowing JST battery port is expecting a LiPo with the 'standard' Adafruit polarity wiring. Using other battery packs with opposite wiring or voltages may destroy your Hallowing!



Lithium Ion Polymer Battery Ideal For Feathers - 3.7V 400mAh

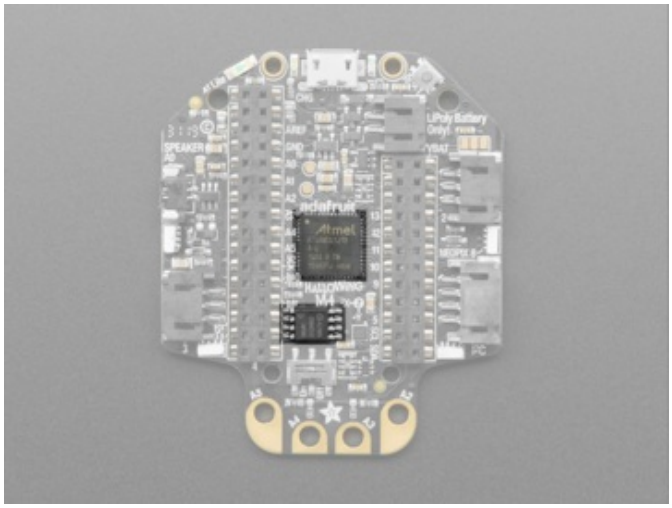
$6.95
IN STOCK

Add To Cart

You can turn off power completely with the on/off switch at the bottom of the board.

If you need access to the power pins, the 'Feather Headers' have **3.3V** regulated out, **GND** on the left. On the right there's the **BAT** pin (connects directly to lipoly) and two pins below that is the **USB** pin. You can measure the voltage on the battery by reading analog pin **A6** - this is divided by two with resistors so don't forget to x2 once you do the reading. The voltage, after doubling, will range from about 3.5 (empty) to 4.2V (charged)

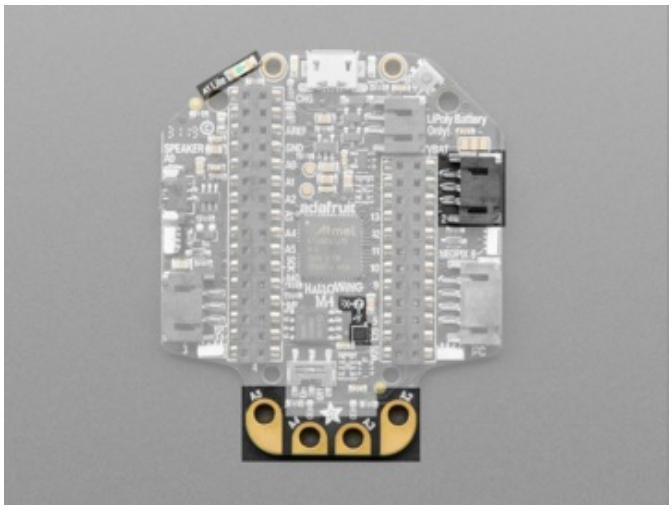
Microcontroller and Flash



The main processor is the **ATSAMD51G18** running at 120MHz with 3.3V logic/power. It has 512KB of flash and 192KB of RAM. It can run Arduino or CircuitPython.

We also include 8 MB of SPI Flash for storing images, sounds, animations, whatever!

Sensors



There's a few built in sensors.

On the top there's a light sensor, connected to pin **A7** - it's reverse mounted so you can read light levels from the front.

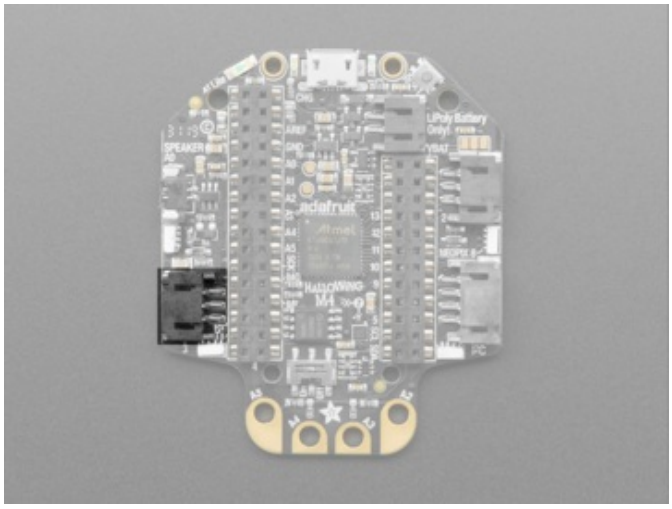
There's also a MSA301 3-axis accelerometer connected to the I2C pins for detection motion, tilt or taps

On the bottom of the board are four pads designed for capacitive touch. They are connected to **A2, A3, A4** and **A5**.

On the right is a **SENSE** port, this is a **JST 3-PH port** for connecting an external sensor. From the top to bottom the pads are **GND**, **V+**, **D2** (in Arduino this is also A8). **V+** is either LiPoly or USB power, whchever is plugged in and higher. There's a 1 Kohm+3.6V zener diode connection to protect against voltages higher than 3.3V coming in.

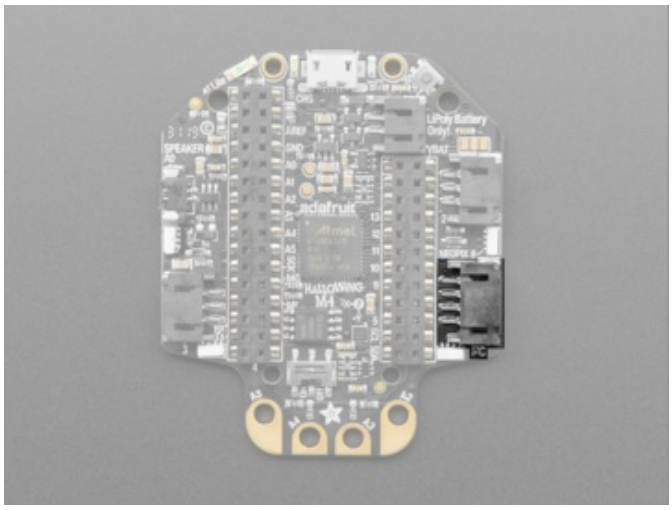
The pin for the light sensor is A7 even though your board may have "A1 Lite" printed next to the light sensor.

External NeoPixel Connector



On the left is the **EXTERNAL NEOPIXEL** port, this is a **JST 3-PH port** for connecting external NeoPixel strips. From top to bottom, the pads are **D3**, **V+**, **GND** (in Arduino this is also A9).

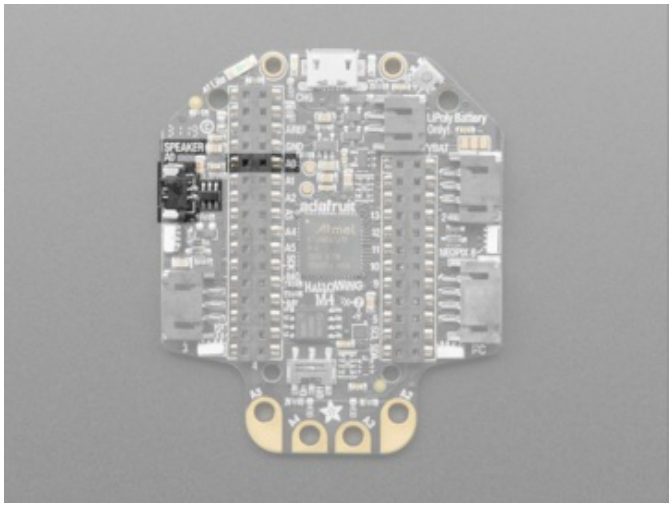
I2C Connector



There is a 4-pin JST **I2C connector** on the right, that is STEMMA and Grove compatible. The I2C has pullups to 3.3V power and is connected to the MSA301 already.

The I2C connector defaults to 5V. There is a jumper you can cut or solder to change it between 5V and 3V. It also changes the STEMMA 3-pin connector power pin voltage!

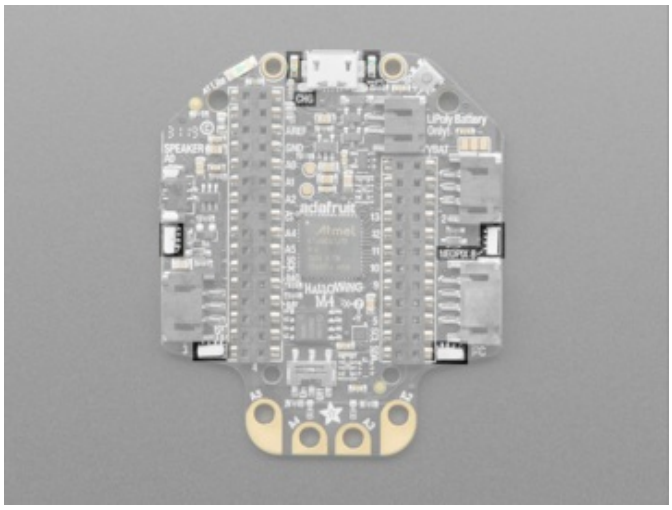
Speaker



There is a **speaker connector** with a mono 2 Watt class D audio amp connected to **A0** - that's the DAC output on the SAMD51,. It is good for many simple sound effects or musical output.

The speaker connector is a Molex PicoBlade (https://adafru.it/C8p).

LEDs



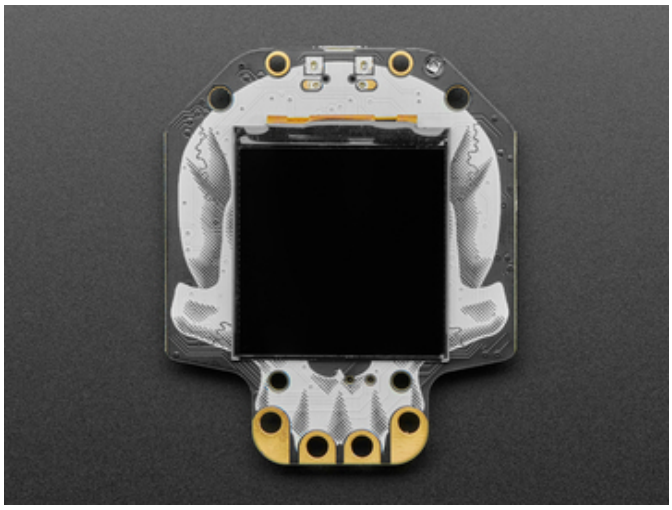
There are six LEDs. There is the red LED on pin **D13** (top to the right of the USB connector), and the **CHG** LED that will let you know when the battery is charging (top to the left of the USB connector)

Located on the sides and towards the bottom are four side light NeoPixels on **D8** (in Arduino) or `board.NEOPIXEL` (in CircuitPython). Check out the image at the top of this page to see what order they're numbered!

It's normal for the yellow **CHG** LED to flicker when no battery is in place, that's the charge circuitry trying to detect whether a battery is there or not. If you are powering only over USB, you can cover it with tape

The charge LED is automatically driven by the Lipoly charger circuit. It will try to detect a battery and is expecting one to be attached. If there isn't one it may flicker once in a while when you use power because it's trying to charge a (non-existant) battery. It's not harmful, and its totally normal!

TFT



On the front is a **1.54" sized 240x240 full color IPS TFT.** Compared to the HalloWing M0's 1.44" 128x128, this has 4x as many pixels and is IPS for great color and brightness.

Quickstart

The HalloWing M4 board should arrive **ready-to-scare**. Connect a battery or USB cable, move the power switch to the "on" position, and after a few seconds you should get a **blinking eye**.

The next couple of pages cover...

- How to **install or update the eye firmware** for the **latest features** and **bug fixes**.
- Loading **different eye designs**.

These pages reference the MONSTER M4SK board repeatedly...but the principle is exactly the same, just on a smaller board with a single eye.

M4 EYES Firmware

To **update firmware** on your MONSTER M4SK board with the latest **eye animation** software, start by downloading this .UF2 file:

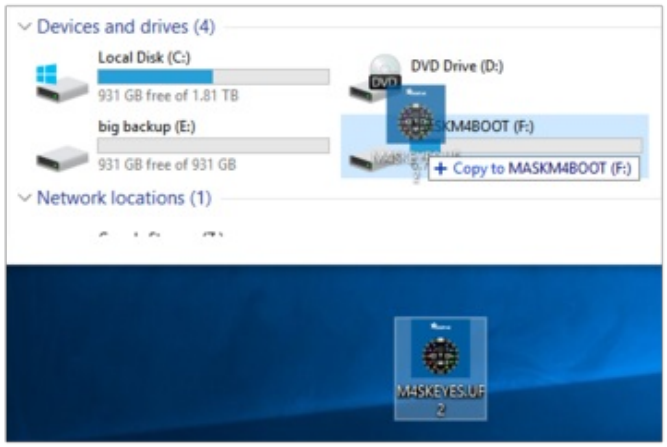
https://adafru.it/FSd

https://adafru.it/FSd

For a HALLOWING M4 board, use this .UF2 instead:

https://adafru.it/FV6

https://adafru.it/FV6



The firmware update sequence is:

1. Connect a USB cable and put the power switch in the “on” position
2. Double-tap the reset button
3. **Wait for the MASKM4BOOT drive to appear!**
4. Drag the M4SKEYES.UF2 file to the MASKM4BOOT drive and wait for it to copy over
5. The board will automatically reboot

After installing the firmware (and a brief pause while the software initializes) you should get some **animated eyeballs**.

If you **don't** see any eyes, make sure you dragged the **M4SKEYES.UF2** file to the **M4SKBOOT** bootloader drive not the **CIRCUITPY** drive

If you get simple **flat-colored** eyes like shown here...that just means **no graphics files** are installed yet. We cover that on the next page. But at least we know the code’s installed!

If you get **textured** eyes that **blink**...code and graphics are all in good shape! The next page shows how to install different looks...and later we get into total customization.

Ready-Made Graphics

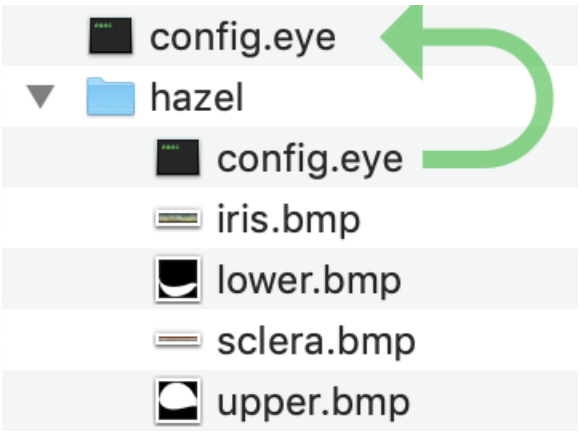
To install different “looks,” download and unzip this collection of eyeball graphics:

https://adafru.it/FAH

https://adafru.it/FAH

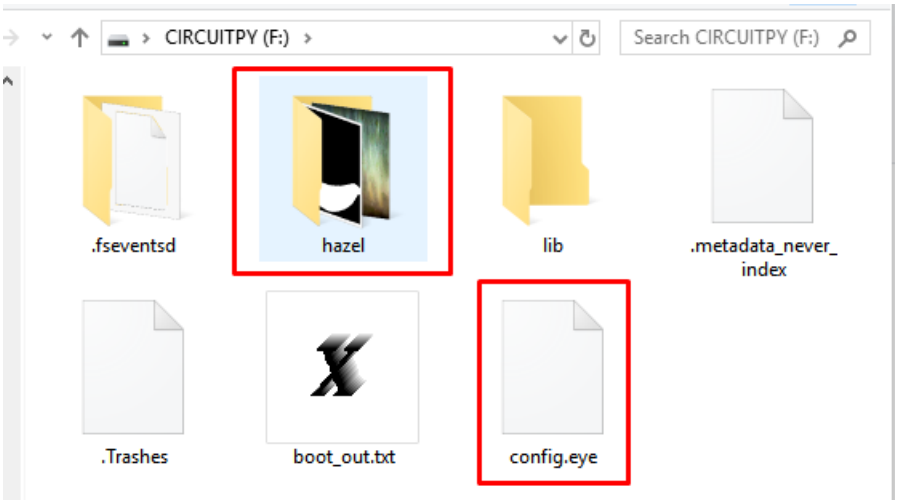
Inside the “eyes” folder are several sub-folders, one for each of our ready-made eye designs. “hazel” is our standard human eye from prior projects...then we’re adding more as Halloween approaches.

Each folder contains a set of .bmp images for that eye, plus a file called “config.eye.” Copy one of these folders to the CIRCUITPY drive — *not* the individual files, but the *whole folder.* Then copy or move the file called **config.eye** out of the folder and into the drive’s root directory.



Copy these files to the CIRCUITPY drive.

Remember that **config.eye** goes in the root directory, along with the **hazel** (or other eye name) folder.



Press the board’s **reset** button...

There will be a **delay** of **several seconds** as the eye code initializes. Then...animation!

If you get a flat-colored white eyeball with blue irises and no eyelids, something's wrong! Verify that you've copied over a whole folder of graphics (not the individual files) and moved or copied that folder's **config.eye** to the root directory.

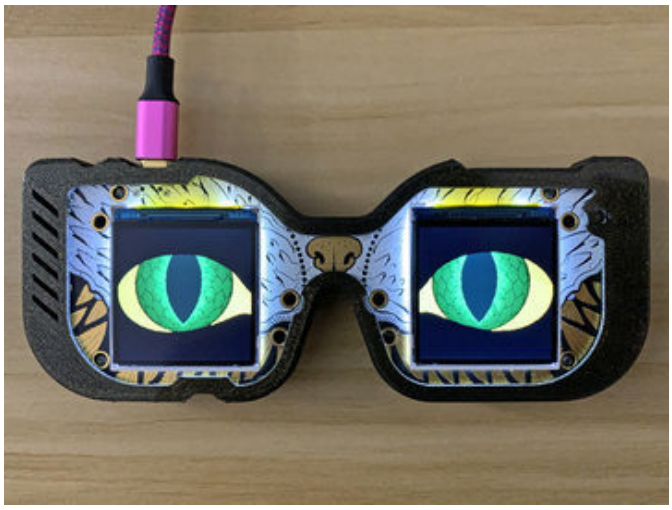
Aside from the stock **hazel** eyes, some of the alternate designs include:

big_blue is a pair of large and friendly blue-gray eyes. The sclera doesn't have all the veins of the hazel eyes, making this less creepy.

fish_eyes is the same unblinking eyes used in our Fish Head MONSTER M4SK project (https://adafru.it/FQj).

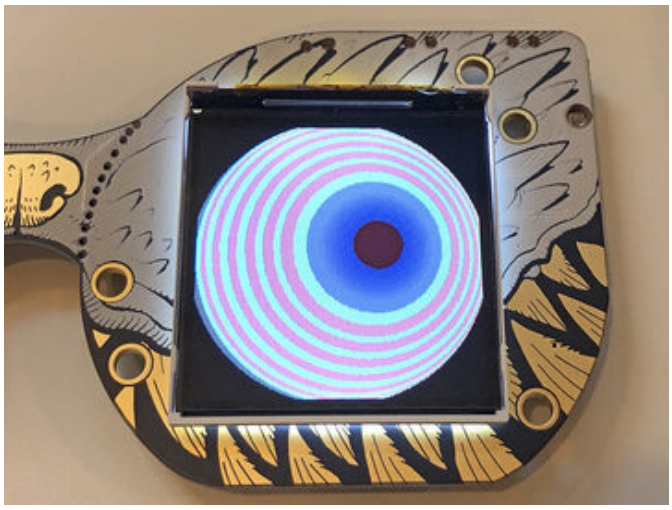
hypno_eyes have that cartoon mesmerizing look. "You are in my power!"

reflection looks like a pair of shiny spheres. No eyelids or pupils, just spheres looking extremely reflective. **ting!**



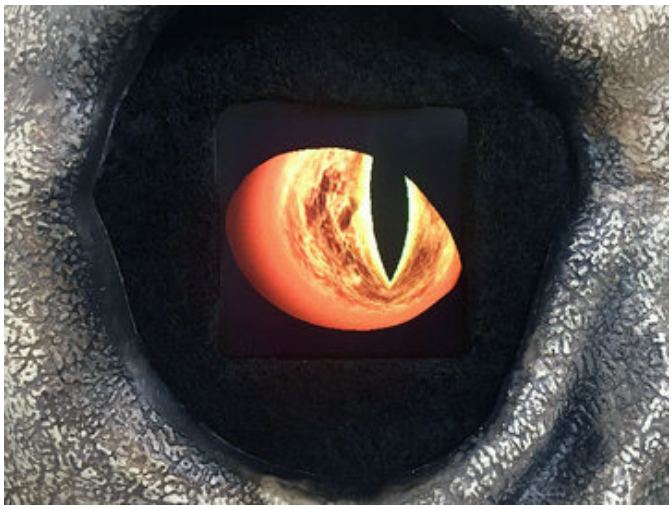
snake_green is perfect for dragons and other reptilian characters. *Rar!* This shows off the **slit pupil** option...also useful for cats and the like.

spikes is adapted from the guide CustomEyesation: DIY Monster M4SK Graphics (https://adafru.it/FFU) and demonstrates a bit how the distortion of texture mapping works.



toonstripe is a different “mesmerizing eyes” look...candy colors, no eyelids.

doom-red and **doom-spiral** were designed for the MONSTER M4SK Toon Hat (https://adafru.it/FQk) guide but might have other uses or tricks to learn from.



The **demon** eyes have swirling fire ... a bit of an “Eye of Sauron” resemblance.

anime does its best approximation of The Quaking Moist Anime Eyes Effect™.

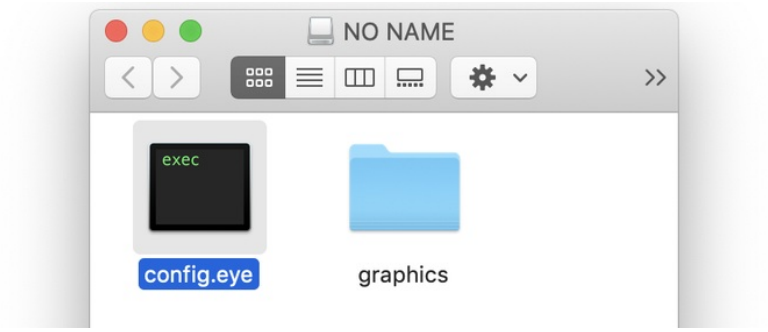
fizzgig resembles the fuzzy *Dark Crystal* character. These react to ambient light and were designed with the lenses in mind.

If one of the ready-made designs does what you need...fantastic, you're all done! If you want to make **changes**, or create your own **custom eyes**, read on...

If our ready-made eye designs don't meet your needs, you can personalize quite a bit by editing a text file and providing some graphics.

On startup, our eyeball code loads a file called **config.eye** from the device's root directory. This is a plain-text file that any simple editor can handle...*Notepad, TextEdit, vim* and so forth.

You wont be able to double-click this file, so open it from WITHIN your text editor



The *syntax* of this file is a format called ***JSON***. There's good and bad news...

The good: JSON has a standardized syntax and we can leverage existing code to read it...we're not starting from scratch with a new file format.

The bad: JSON files are really meant to be read and written by machines, as a way to preserve program state...*not* edited by humans. It's *phenomenally* picky about getting syntax *just right* and does not fall back gracefully.

The saving grace: if you're just changing colors and textures, you might not need to edit this file at all! Quite often you can just substitute different graphics files with the same names.

Troubleshooting JSON

If even a *single character* is out of place in **config.eye**, the *whole thing* will fail to load and there's no helpful indication of where the problem lies.

If this happens, the eyes will run in a **default state**: blue eyes with no eyelids, and everything is "flat" colors, no textures. This isn't helpful in *isolating* the problem but at least tells you there *is* a problem.

One way to troubleshoot this is to start with a known valid eye configuration, then change one setting at a time and restart the code. If the eyes revert to the default state, the last change contains the syntax error. Quite often it's just a missing double quote or extra comma.

Another option is to use a **JSON validator** such as this one at hjson.org (https://adafru.it/FzV). Copy-and-paste your eyeball configuration into the left pane...if there's a problem, this will be reported on the right.

Here's an simple **config.eye** file to start with:

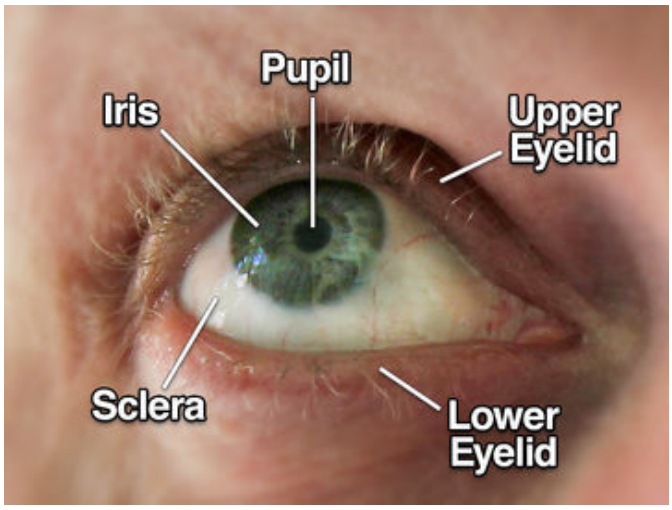
```
{
  "pupilColor"    : [ 0, 0, 0 ],
  "backColor"     : [ 140, 40, 20 ],
  "irisTexture"   : "graphics/iris.bmp",
  "scleraTexture" : "graphics/sclera.bmp",
  "upperEyelid"   : "graphics/upper.bmp",
  "lowerEyelid"   : "graphics/lower.bmp"
}
```

Some things to notice:

- The whole thing is contained within *curly braces* — { and }
- Each item has a **name** (always in quotes), a colon **separator** (**:**) and a **value**. Values might be numbers, strings (in quotes) or arrays (in square brackets) depending on what's being configured.
- This is a *list* and each line ends with a **comma**...*except* for the last item in the list. **Watch out for missing or surplus commas!**
- JSON is **case-sensitive**. "Foo", "foo" and "FOO" are all different things.
- Using extra spaces to line up columns really isn't necessary, just something I do to assist with legibility. Some folks just find it annoying.
- Comments (starting with //) are *not* standard JSON syntax, but the library we use allows them and I find them very helpful.

Each option will be explained in detail on the "Configurable Settings" page. But first let's talk about **graphics**...

So we're singing from the same page, let's lay out some eye terminology...



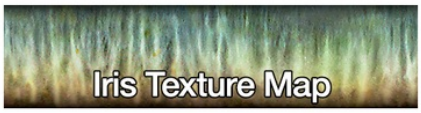
The technical term for the "white" of the eye is the *sclera.*

The *iris* is the muscle that contracts to adjust the size of the *pupil* in response to light.

The *upper* and *lower eyelids* are involved in blinking.

This program's eye graphics are stored flat and unrolled, like a map projection. The horizontal (X) axis works like the longitude, or angle around the eye, while the vertical (Y) axis is the latitude. The images are wrapped around the pupil in a **clockwise** direction.

There are **two** images (or *texture maps*) associated with the eyes...one for the iris, another for the sclera.



The **iris** is what we think of as the "color" of the eye and is most often what you'll want to edit. Sometimes you just need to edit the hue & saturation in a program like Photoshop, or you can make something totally custom if you're after a particular look.

The **sclera** is the "white" of the eye...which really isn't that white at all. There's veins and blotches and gross stuff!

Image Storage

The texture maps are stored as **24-bit BMP** images…nobody's favorite, but easy for microcontrollers to handle. (This is also sometimes called "Windows Bitmap" format, though plenty of non-Windows software can read and write these images…and *not* to be confused with "X BitMap" or "Portable Bitmap Format," different animals.)

Textures are loaded from drive into RAM, downsampled to 16-bit color (what the displays natively use) and then written to the chip's internal flash memory (different from the flash filesystem). Although the code places no strict limit on image dimensions, **RAM and flash are both finite resources**, and this limits to how large these textures can be…both individually and in total.

No *single* image can exceed available RAM, or about **160 kilobytes**. The total of *all images* must fit within flash, or about **360 kilobytes**. These figures might change a bit in the future, so try to leave yourself some overhead.

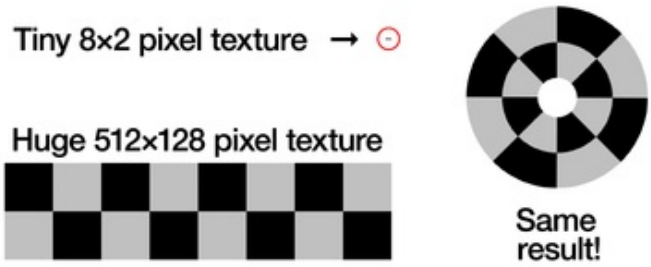
Use the following formula to determine the space needed for an image:

width in pixels × height in pixels × 2 bytes

For example, a 500 × 150 pixel texture would consume 500 × 150 × 2 = 150,000 bytes. This fits in RAM just fine, and takes up a bit less than half of the flash space.

You *don't* have to texture-map both the iris and sclera if you don't want to…on the *Configurable Settings* page we explain how to use **solid colors** for either or both. Also shown there…it's possible to assign independent textures to the left and right eyes. When both eyes are sharing the same texture, the code will place only one instance in flash, saving space.

To make the re-use of textures less obvious, the "seam" where a texture map wraps around is normally at the 12 o'clock position for the right eye and 6 o'clock for the left eye. The *Configurable Settings* page shows how to change this.



Eye textures can be any size (RAM permitting), on either axis…even down to a single pixel. When wrapping small images around the whole eye, **nearest-neighbor** sampling (no interpolation) is used. This can be exploited to create stylized blocky or grid designs…no need to waste space on a big image when just a few pixels will do.

Too-large images may exceed available space, while too-small images may exhibit visible jaggies (unless you're aiming for that effect as described above). The **ideal size** can be calculated based on the circumference of the iris or the whole eye…

The iris and overall eye size are configurable (shown on following page)…but for example, let's assume you've got an

iris with a 60 pixel radius (120 pixel diameter) and the eyeball has a 125 pixel radius (250 pixel diameter)...both of these are the defaults.

Multiply the iris and/or eye **diameters** by **Pi (3.14)** to get the **ideal width in pixels** for the iris and sclera images.

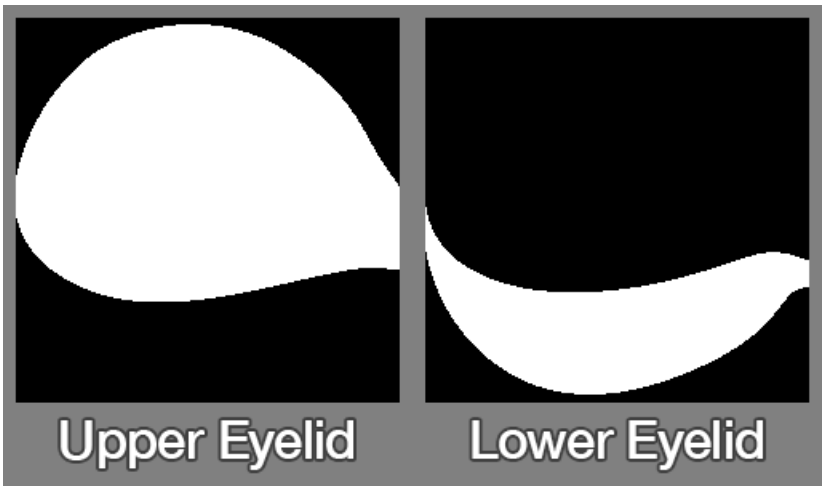
For example: iris with 120 pixel diameter. 120 px × 3.14 = 377 pixels wide. You can use that, or round up or down a smidge to a round number if you like (e.g. 360 or 380 pixels).

Sclera image width for 250 pixel diameter eye: 250 × 3.14 = 785 pixels wide...but again, OK to round up or down a little...use 800 pixels wide if you like, unless really pressed for space.

The ideal image *heights* are a bit different. First, although the code can *load* any size image, it won't actually *benefit* above 128 pixels on the vertical axis, it's just wasted space. For the iris, use its radius (60 pixels in the case described above) or even a little less, since the pupil is always open a bit. For the sclera...try 200 minus the iris radius, keeping in mind the 128 pixel recommended maximum (e.g. with a 60 pixel iris, 140 is our target, then cap it at 128). But...with an 800 pixel wide sclera as described above...800 × 128 × 2 = 204 kilobytes...quite a bit over the 160K RAM limit! Whittle down one or both axes, whatever you think can best handle less resolution, until you find a size that fits. Once in motion, and at a reasonable viewing distance, minor "jaggies" aren't that noticeable.

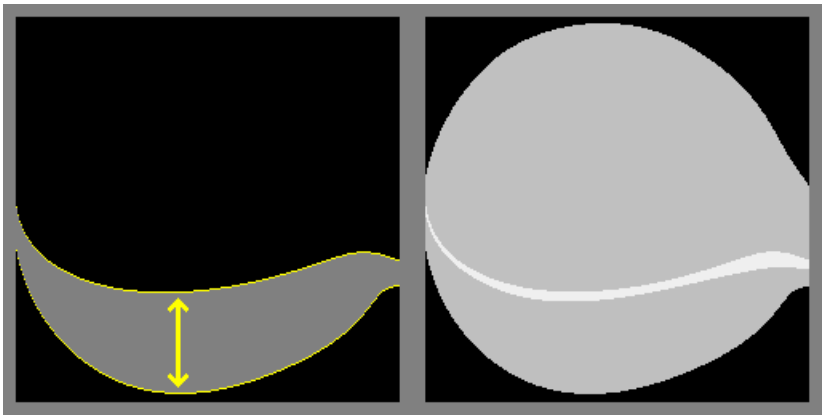
Eyelids

The eyelids have stricter requirements. There are always **two files** (one each for the **upper** and **lower** eyelids) both **240 × 240 pixels** exactly, both **1-bit BMP images** *(NOT 24-bit like the texture maps!)*.



The white area — which should span the **entire 240 pixel width** — represents the extent of vertical motion, the "open-most" and "closed-most" range. The eyes won't hold the open-most position all the time when idle...the upper eyelid "tracks" the moving pupil, because that's how eyes work.

A good set of eyelid images will **overlap by a few pixels**, so the eye makes a good solid blink.



This particular set of eyelids is slightly **asymmetrical** to approximate the eye's *caruncle** — that triangular bit by the tear duct. The shape is mirrored between the two eyes. But it is super 100% okay to make **symmetrical** eyelids if you prefer...a simple "football shape"...that usually looks better on a single-eye board like the HalloWing M4. For some of the eye designs we'll offer both.

* Which itself is a vestigial remnant of the *nictitating membrane,* the "third eyelid" that reptiles have. *How cool is that!?*

The eyelid images as shown above are for the right eye. That is...**the *monster's* right eye**, meaning the eye on the **left** when looking at the M4SK.

Configurable Settings

Let's look at the configuration for our stock "hazel" human eye. It doesn't reference every configurable setting, but shows the general format of these files...

```
{
  "eyeRadius"     : 125,
  "eyelidIndex"   : "0x00", // From table: learn.adafruit.com/assets/61921
  "pupilColor"    : [ 0, 0, 0 ],
  "backColor"     : [ 140, 40, 20 ],
  "irisTexture"   : "hazel/iris.bmp",
  "scleraTexture" : "hazel/sclera.bmp",
  "upperEyelid"   : "hazel/upper.bmp",
  "lowerEyelid"   : "hazel/lower.bmp",
  "left" : {
  },
  "right" : {
  }
}
```

This is a plain text file that any simple editor should be able to handle...Notepad or TextEdit or whatever comes bundled on your computer.

It's worth reiterating these points from the "Customization Basics" page:

- The whole thing is contained within *curly braces* — { and }
- Each item has a **name** (always in quotes), a colon **separator** (**:**) and a **value**. Values might be numbers, strings (in quotes) or arrays (in square brackets) depending on what's being configured.
- This is a *list* and each line ends with a **comma**...*except* for the last item in the list. **Watch out for missing or surplus commas!**
- JSON is **case-sensitive**. "Foo", "foo" and "FOO" are all different things.
- Using extra spaces to line up columns really isn't necessary, just something I do to assist with legibility. Some folks just find it annoying.
- Comments (starting with //) are *not* standard JSON syntax, but the library we use allows them and I find them very helpful.

The "Customization Basics" page also has some **JSON troubleshooting tips**. If you encounter trouble, review that page!

Sizes and Shapes

Two settings define the **basic geometry** of the eye...

`eyeRadius` establishes the size of the overall eyeball, in **pixels**. This is a ***radius*** — center to edge — so the overall eye size is *twice this* across. For example `eyeRadius:125` configures the eye to be 250 pixels wide. This is the default if left unspecified.

The screens are only **240 pixels** wide. Reason the eye is made a little bigger is because the code uses tricks to *fake* a rotating sphere...and that faking is more apparent as the pupil approaches an edge. So we push the edge out a few extra pixels, then *cover it up with eyelids.*

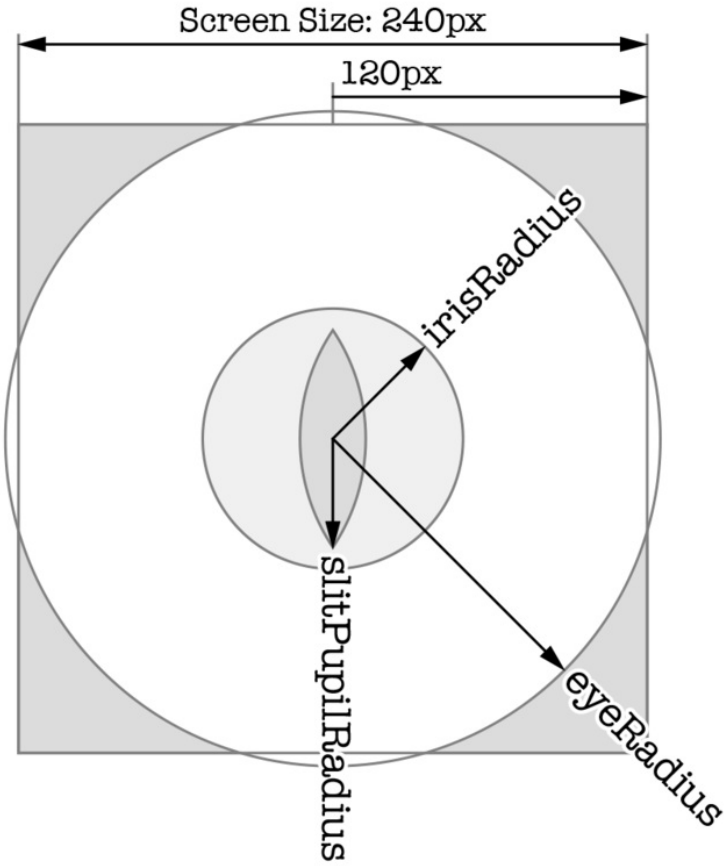
If designing an eye with **no eyelids**, you might want `eyeRadius:120` instead, which provides a nice **perfect circle** on the screen.

*Remember that JSON is **case-sensitive**. This **must** be spelled `eyeRadius`. Different capitalization will cause it to be ignored!*

`irisRadius` establishes the size of the **iris**...again a *radius,* in pixels.

`irisRadius:60` will make the iris 120 pixels across, or half the width of the screen. If you plan to use **lenses** over the displays, consider scaling down this number a bit to compensate.

Some creatures...cats and so forth...have *very large irises* and almost *no visible sclera*. In that case you can set `irisRadius` much larger, up to (but not exceeding) `eyeRadius`.



A third setting, `slitPupilRadius`, lets you make cat or dragon type eyes with a vertical slit pupil (only a *vertical slit* is available, no goat pupils, sorry). If set to **0** (the default), a normal **round** pupil is used. Larger numbers (up to `irisRadius`) make a taller/thinner pupil. This number sets the **height**. You'll probably want an in-between value...maybe `irisRadius:80` (160 pixels round) and `slitPupilRadius:60` (120 pixels tall) to start.

*Note that using `slitPupilRadius` makes the program **a bit slower to initialize**...you'll just see blank screens for several seconds while it works. This is normal and just an unfortunate math thing.*

The **texture map images** for **iris** and **sclera** are specified with `irisTexture` and `scleraTexture`:

```
"irisTexture"   : "hazel/iris.bmp",
"scleraTexture" : "hazel/sclera.bmp",
```

If you prefer a **solid color** to an image, omit those lines and instead use `irisColor` and/or `scleraColor`, with color specified as three values [red, green,blue] in brackets:

```
"scleraColor"   : [ 255, 255, 255 ],
```

Colors can either be **three integers** in the range **0 to 255** (like most folks are used to) or three **floating-point** values from **0.0 to 1.0** (same idea, just different scale).

Solid colors save a TON of space compared to texture maps, if your design can get away with it.

A few more items have configurable colors:

```
"pupilColor"    : [ 0, 0, 0 ],
"backColor"     : [ 140, 40, 20 ],
```

`pupilColor` is self-explanatory...like if you want glowing red or white pupils or something.

`backColor` covers the outermost/backmost part of the eye where the sclera texture map (or color) doesn't reach. With a little planning, this and your sclera texture map can be designed to blend together...otherwise you'll see a conspicuous crescent of `backColor` when the eye is looking off to a side.

The eyelid (and background) color is also configurable, but it's **not a normal RGB color**. Instead, use `eyelidIndex` and an 8-bit value:

```
"eyelidIndex"   : "0x00",
```

Notice the hexadecimal value must be **quoted**. The value corresponds to one of the colors in this palette:

`eyelidIndex` doesn't use a normal RGB value because the code is using an optimization trick here, which limits the available colors in this one area. The default `eyelidIndex` if unspecified is "**0x00**" — black.

Eyelids

The eyelid graphics format is explained on the “Preparing Graphics” page. These are 1-bit BMPs, 240 pixels square. Use the `upperEyelid` and `lowerEyelid` settings to specify the filenames of these images.

Eyelids are one of those global (not configurable per-eye) things. The design of the eyelid graphics might be symmetrical or asymmetrical…some eye designs might provide two sets of eyelid images, one for a single-eye device (like HalloWing M4) and another for the Monster M4SK, where the left and right eyelid shapes are mirrored.

If you want no eyelids at all, just leave out any `upperEyelid` or `lowerEyelid` filenames. You’ll then have a circular unblinking eye…looks good for skulls!

With eyelids enabled, normally the upper lid “tracks” the movement of the pupil (when the eye looks down, the eyelid follows with it). This is something that eyes do in real life…but some folks think it looks sleepy, or just want a particular caffeinated look. Use the `tracking` keyword with a value of `false` to disable the eyelid tracking and maintain wide-open eyes.

Light Sensor

The pupils normally do some dilation movement on their own...but you can have them **respond to light** if you like! Use the `lightSensor` keyword along with a pin number where the light sensor is connected. If it's on a Seesaw interface chip (e.g. on MONSTER M4SK), add 100 to the pin number. For example, on **MONSTER M4SK** the built-in light sensor is:

```
"lightSensor" : 102
```

And for **HalloWing M4**, use:

```
"lightSensor" : 21
```

(Add a trailing comma if this appears mid-file.)

Building Eyes from Source Code

The next couple of pages explain how to set up the Arduino software to work with the SAMD microcontroller boards like the HalloWing M4 or MONSTER M4SK. **If you've previously worked with Adafruit SAMD boards with Arduino, that part's already set up and you can skip ahead to the "Libraries and Settings" page.**

In summary: use the latest Arduino IDE, install the latest Adafruit SAMD boards package, install Windows drivers if necessary, and install all the library dependencies for this project...

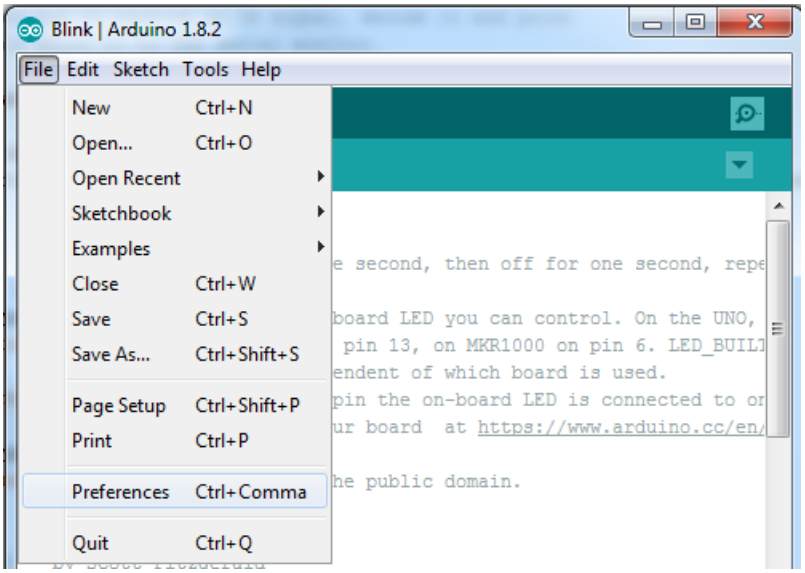
Arduino IDE Setup

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

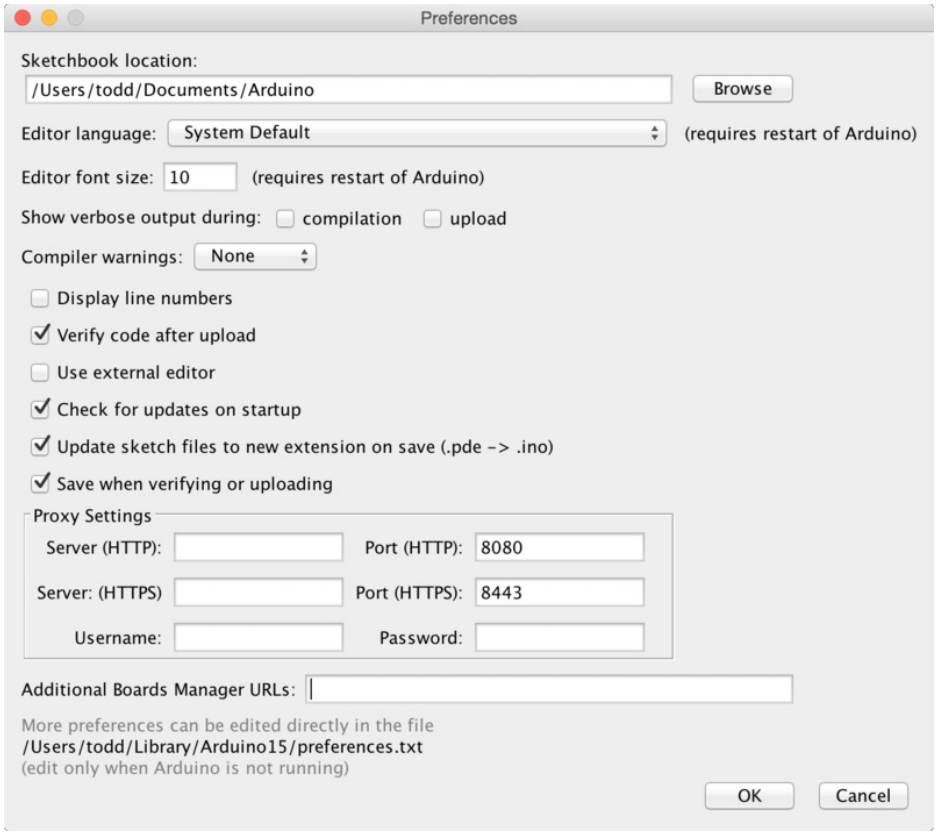
https://adafru.it/f1P

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*.



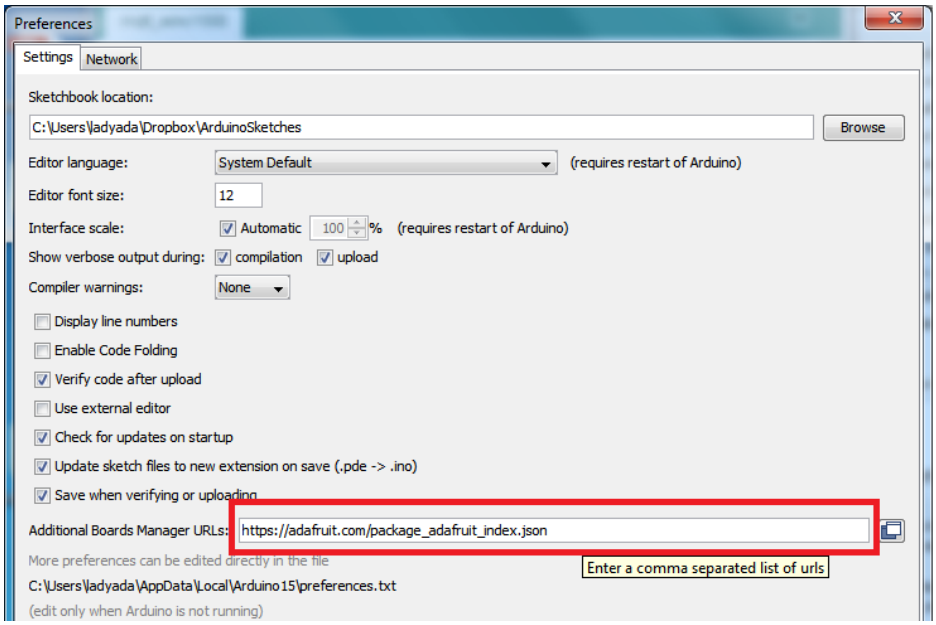
A dialog will pop up just like the one shown below.



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 (https://adafru.it/f7U). 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 and M4, Metro M0 and M4, ItsyBitsy M0 and M4, 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 (https://adafru.it/eSI).

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

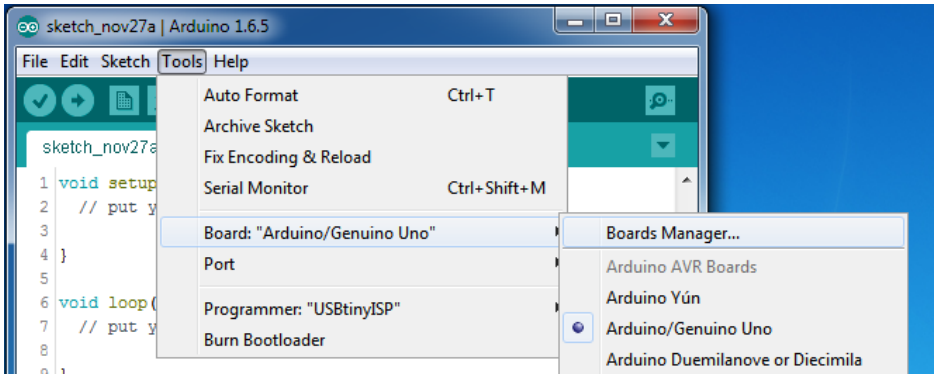
Once done click **OK** to save the new preference settings. Next we will look at installing boards with the Board Manager.

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

Using with Arduino IDE

The Feather/Metro/Gemma/Trinket M0 and M4 use an ATSAMD21 or ATSAMD51 chip, and 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 and M4, 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.



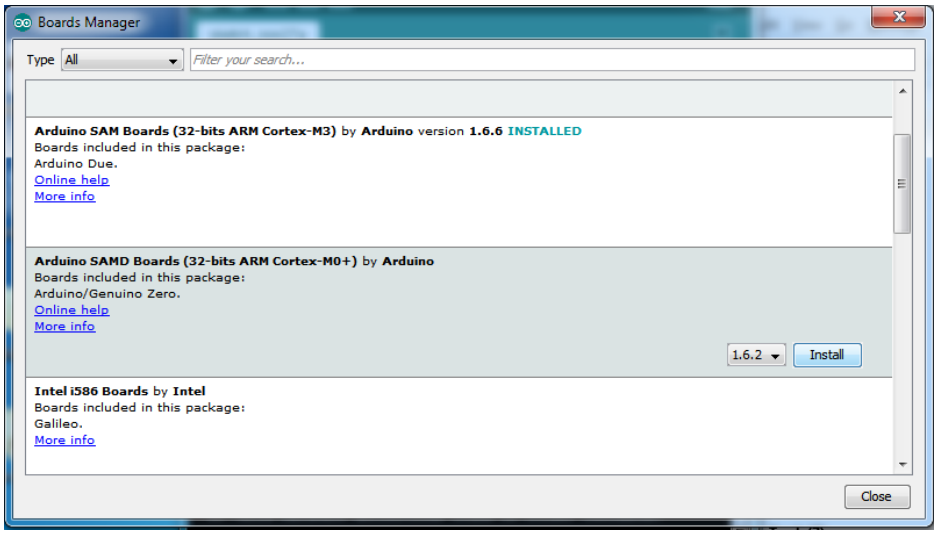
Once the Board Manager opens, click on the category drop down menu on the top left hand side of the window and select **All**. You will then be able to select and install the boards supplied by the URLs added to the preferences.

Remember you need SETUP the Arduino IDE to support our board packages - see the previous page on how to add adafruit's URL to the preferences

Install SAMD Support

First up, install the latest **Arduino SAMD Boards (**version **1.6.11** or later)

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

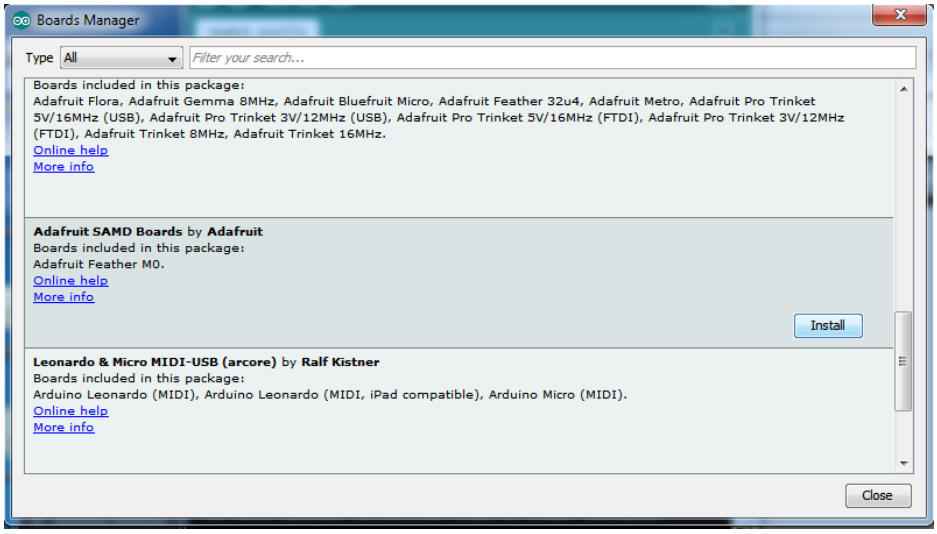


Install Adafruit SAMD

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

Make sure you have **Type All** selected to the left of the *Filter your search...* box

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

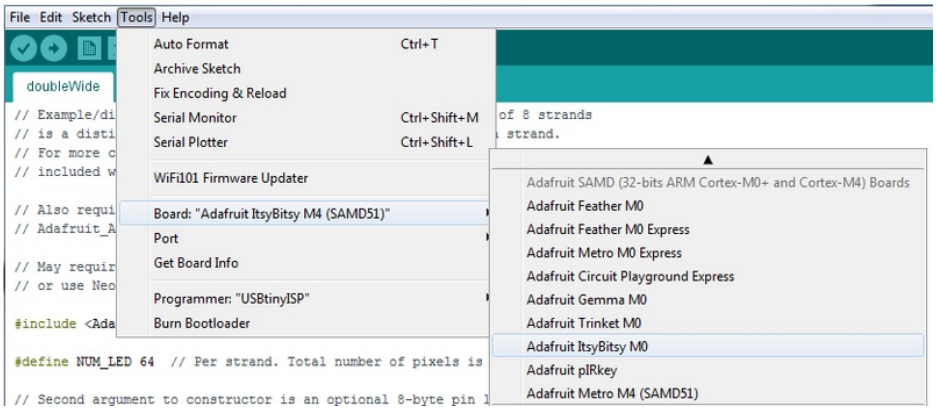


Even though in theory you don't need to - I recommend rebooting the IDE

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**
- **ItsyBitsy M0**
- **Hallowing M0**
- **Crickit M0** (this is for direct programming of the Crickit, which is probably not what you want! For advanced hacking only)
- **Metro M4 Express**
- **ItsyBitsy M4 Express**
- **Feather M4 Express**
- **Trellis M4 Express**
- **Grand Central M4 Express**



Install Drivers (Windows 7 & 8 Only)

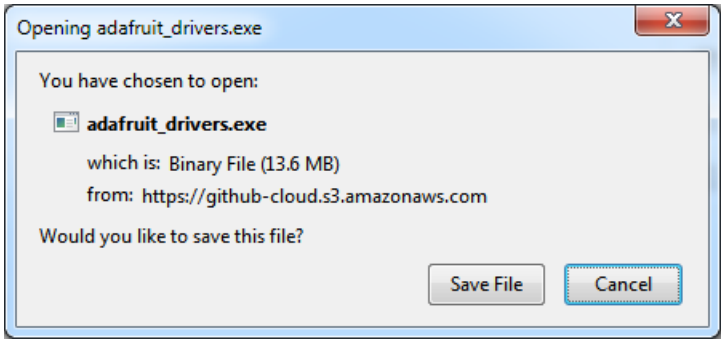
When you plug in the board, you'll need to possibly install a driver

Click below to download our Driver Installer

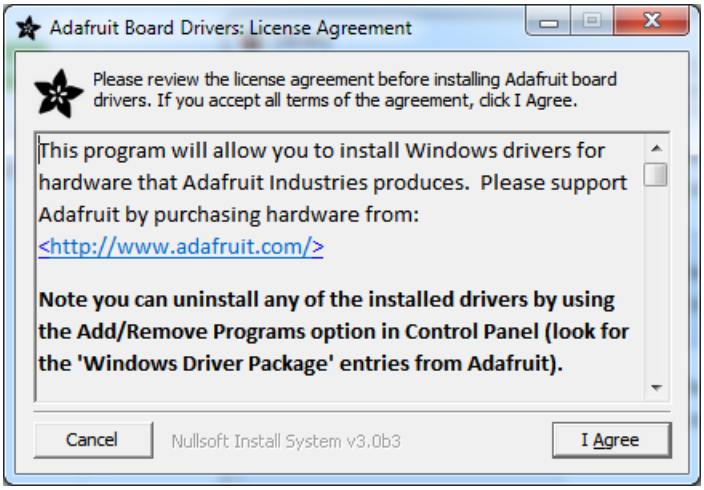
https://adafru.it/EC0

https://adafru.it/EC0

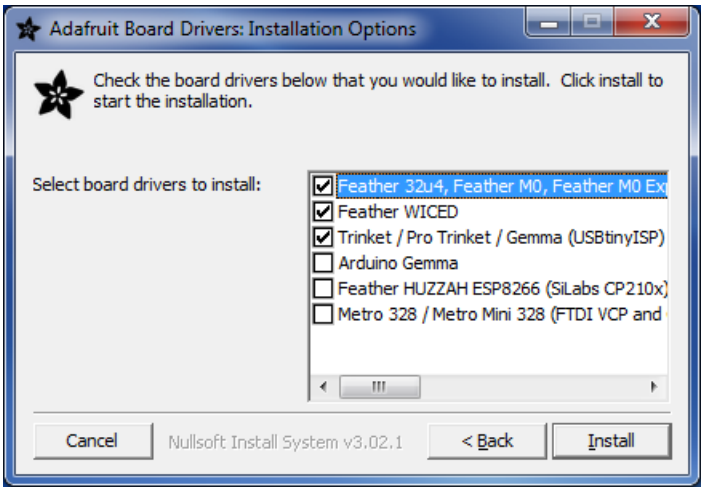
Download and run the installer



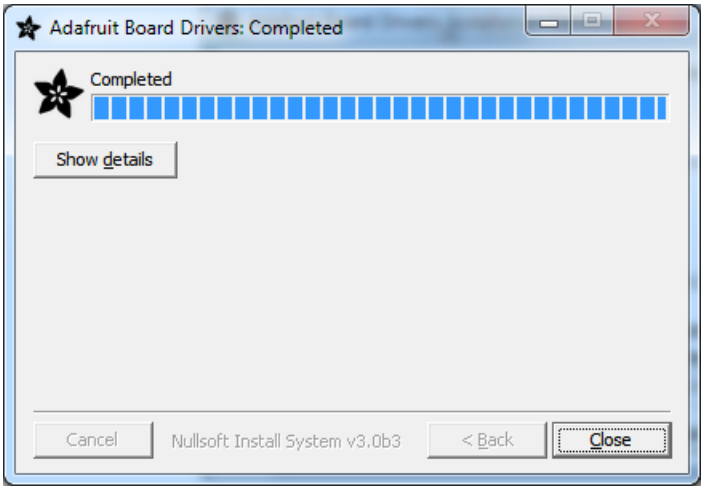
Run the installer! Since we bundle the SiLabs and FTDI drivers as well, you'll need to click through the license



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



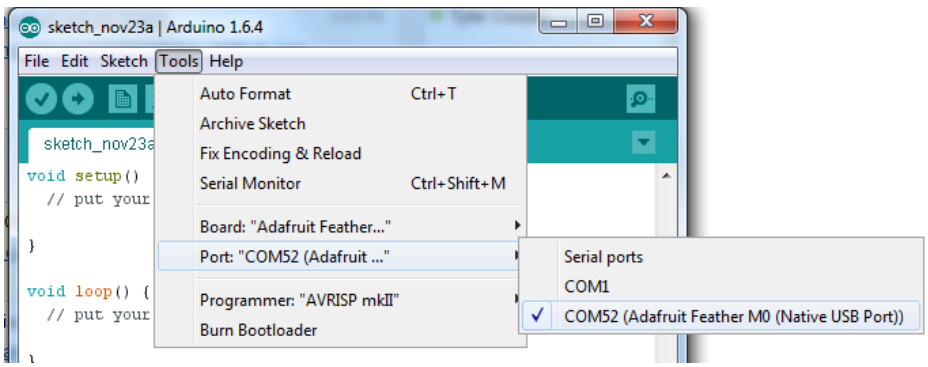
Click **Install** to do the installin'



Blink

Now you can upload your first blink sketch!

Plug in the M0 or M4 board, 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 drop-down, it'll even be 'indicated' as Trinket/Gemma/Metro/Feather/ItsyBitsy/Trellis!



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
}
```

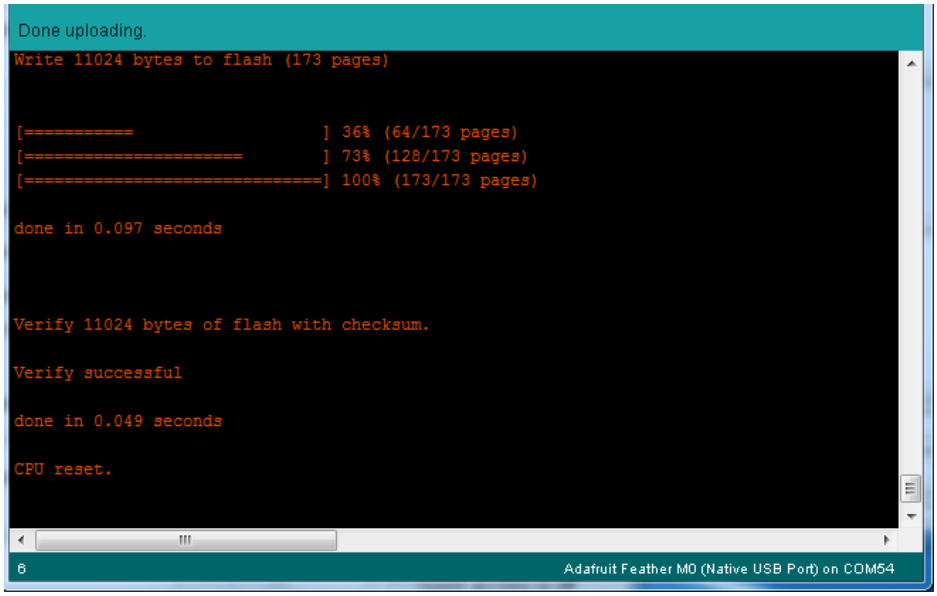
And click upload! That's it, you will be able to see the LED blink rate change as you adapt the **delay()** calls.

If you're using **Trellis M4 Express**, you can go to the next page cause there's no pin 13 LED - so you won't see it blink. Still this is a good thing to test compile and upload!

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

Successful 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



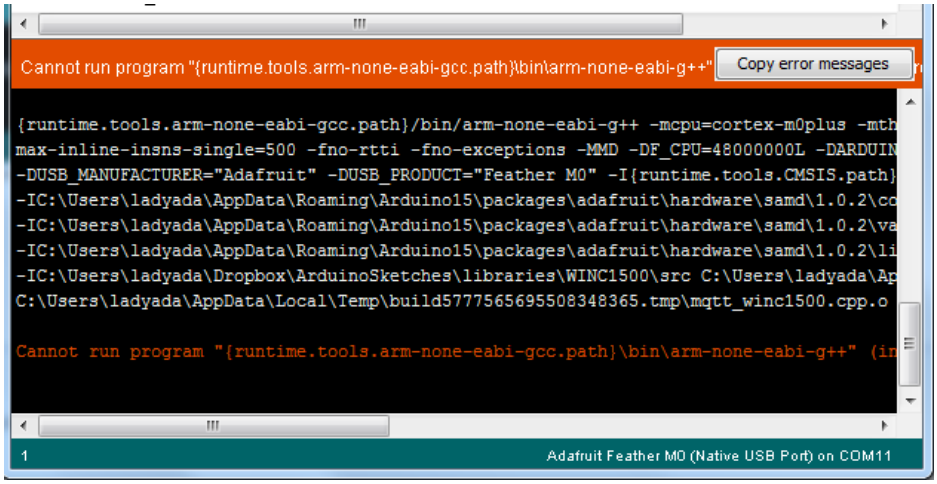
After uploading, you may see a message saying "Disk Not Ejected Properly" about the ...BOOT drive. You can ignore that message: it's an artifact of how the bootloader and uploading work.

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

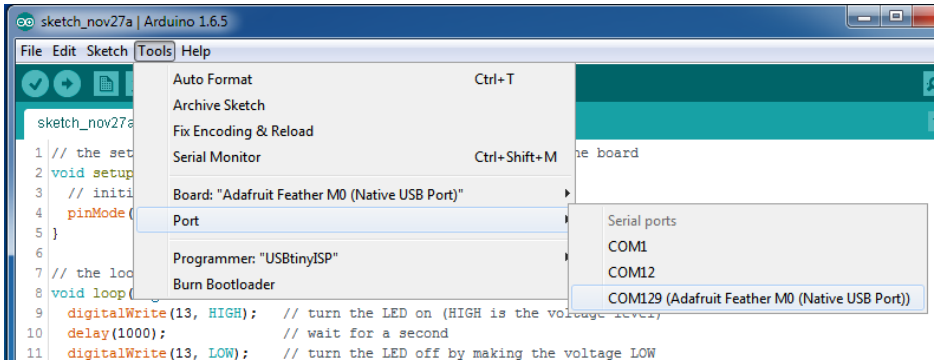


Manually bootloading

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

Follow the steps for installing Adafruit's udev rules on this page. (https://adafru.it/iOE)

Adapting Sketches to M0 & M4

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

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

Analog References

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

Pin Outputs & Pullups

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

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

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

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

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

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

Serial vs SerialUSB

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

In the Adafruit M0/M4 Core, we fixed it so that **Serial goes to USB so it will automatically work just fine**.

However, on the off chance you are using the official Arduino SAMD core and *not* the Adafruit version (which really, we recommend you use our version because it's been tuned to our boards), and you want your Serial prints and reads to use the USB port, use *SerialUSB* instead of *Serial* in your sketch.

If you have existing sketches and code and you want them to work with the M0 without a huge find-replace, put

```
#if defined(ARDUINO_SAMD_ZERO) && defined(SERIAL_PORT_USBVIRTUAL)
 // Required for Serial on Zero based boards
 #define Serial SERIAL_PORT_USBVIRTUAL
#endif
```

right above the first function definition in your code. For example:

AnalogWrite / PWM on Feather/Metro M0

After looking through the SAMD21 datasheet, we've found that some of the options listed in the multiplexer table don't exist on the specific chip used in the Feather M0.

For all SAMD21 chips, there are two peripherals that can generate PWM signals: The Timer/Counter (TC) and Timer/Counter for Control Applications (TCC). Each SAMD21 has multiple copies of each, called 'instances'.

Each TC instance has one count register, one control register, and two output channels. Either channel can be enabled and disabled, and either channel can be inverted. The pins connected to a TC instance can output identical versions of the same PWM waveform, or complementary waveforms.

Each TCC instance has a single count register, but multiple compare registers and output channels. There are options for different kinds of waveform, interleaved switching, programmable dead time, and so on.

The biggest members of the SAMD21 family have five TC instances with two 'waveform output' (WO) channels, and three TCC instances with eight WO channels:

- TC[0-4],WO[0-1]
- TCC[0-2],WO[0-7]

And those are the ones shown in the datasheet's multiplexer tables.

The SAMD21G used in the Feather M0 only has three TC instances with two output channels, and three TCC instances with eight output channels:

- **TC[3-5],WO[0-1]**
- **TCC[0-2],WO[0-7]**

Tracing the signals to the pins broken out on the Feather M0, the following pins can't do PWM at all:

- **Analog pin A5**

The following pins can be configured for PWM without any signal conflicts as long as the SPI, I2C, and UART pins keep their protocol functions:

- **Digital pins 5, 6, 9, 10, 11, 12, and 13**
- **Analog pins A3 and A4**

If only the SPI pins keep their protocol functions, you can also do PWM on the following pins:

- **TX and SDA (Digital pins 1 and 20)**

analogWrite() PWM range

On AVR, if you set a pin's PWM with `analogWrite(pin, 255)` it will turn the pin fully HIGH. On the ARM cortex, it will set it to be 255/256 so there will be very slim but still-existing pulses-to-0V. If you need the pin to be fully on, add test code that checks if you are trying to `analogWrite(pin, 255)` and, instead, does a `digitalWrite(pin, HIGH)`

analogWrite() DAC on A0

If you are trying to use `analogWrite()` to control the DAC output on **A0**, make sure you do **not** have a line that sets the pin to output. ***Remove***: `pinMode(A0, OUTPUT)`.

Missing header files

There might be code that uses libraries that are not supported by the M0 core. For example if you have a line with

```
#include <util/delay.h>
```

you'll get an error that says

```
fatal error: util/delay.h: No such file or directory
 #include <util/delay.h>
                ^
compilation terminated.
Error compiling.
```

In which case you can simply locate where the line is (the error will give you the file name and line number) and 'wrap it' with #ifdef's so it looks like:

```
#if !defined(ARDUINO_ARCH_SAM) && !defined(ARDUINO_ARCH_SAMD) && !defined(ESP8266) &&
!defined(ARDUINO_ARCH_STM32F2)
 #include <util/delay.h>
#endif
```

The above will also make sure that header file isn't included for other architectures

If the #include is in the arduino sketch itself, you can try just removing the line.

Bootloader Launching

For most other AVRs, clicking **reset** while plugged into USB will launch the bootloader manually, the bootloader will time out after a few seconds. For the M0/M4, you'll need to ***double click*** the button. You will see a pulsing red LED to let you know you're in bootloader mode. Once in that mode, it wont time out! Click reset again if you want to go back to launching code.

Aligned Memory Access

This is a little less likely to happen to you but it happened to me! If you're used to 8-bit platforms, you can do this nice thing where you can typecast variables around. e.g.

```
uint8_t mybuffer[4];
```

`float f = (float)mybuffer;`

You can't be guaranteed that this will work on a 32-bit platform because **mybuffer** might not be aligned to a 2 or 4-byte boundary. The ARM Cortex-M0 can only directly access data on 16-bit boundaries (every 2 or 4 bytes). Trying to access an odd-boundary byte (on a 1 or 3 byte location) will cause a Hard Fault and stop the MCU. Thankfully, there's an easy work around ... just use memcpy!

```
uint8_t mybuffer[4];
float f;
memcpy(&f, mybuffer, 4)
```

Floating Point Conversion

Like the AVR Arduinos, the M0 library does not have full support for converting floating point numbers to ASCII strings. Functions like sprintf will not convert floating point. Fortunately, the standard AVR-LIBC library includes the dtostrf function which can handle the conversion for you.

Unfortunately, the M0 run-time library does not have dtostrf. You may see some references to using **#include <avr/dtostrf.h>** to get dtostrf in your code. And while it will compile, it does **not** work.

Instead, check out this thread to find a working dtostrf function you can include in your code:

http://forum.arduino.cc/index.php?topic=368720.0 (https://adafru.it/IFS)

How Much RAM Available?

The ATSAMD21G18 has 32K of RAM, but you still might need to track it for some reason. You can do so with this handy function:

```
extern "C" char *sbrk(int i);

int FreeRam () {
  char stack_dummy = 0;
  return &stack_dummy - sbrk(0);
}
```

Thx to http://forum.arduino.cc/index.php?topic=365830.msg2542879#msg2542879 (https://adafru.it/m6D) for the tip!

Storing data in FLASH

If you're used to AVR, you've probably used **PROGMEM** to let the compiler know you'd like to put a variable or string in flash memory to save on RAM. On the ARM, its a little easier, simply add **const** before the variable name:

const char str[] = "My very long string";

That string is now in FLASH. You can manipulate the string just like RAM data, the compiler will automatically read from FLASH so you dont need special progmem-knowledgeable functions.

You can verify where data is stored by printing out the address:
Serial.print("Address of str $"); Serial.println((int)&str, HEX);

If the address is $2000000 or larger, its in SRAM. If the address is between $0000 and $3FFFF Then it is in FLASH

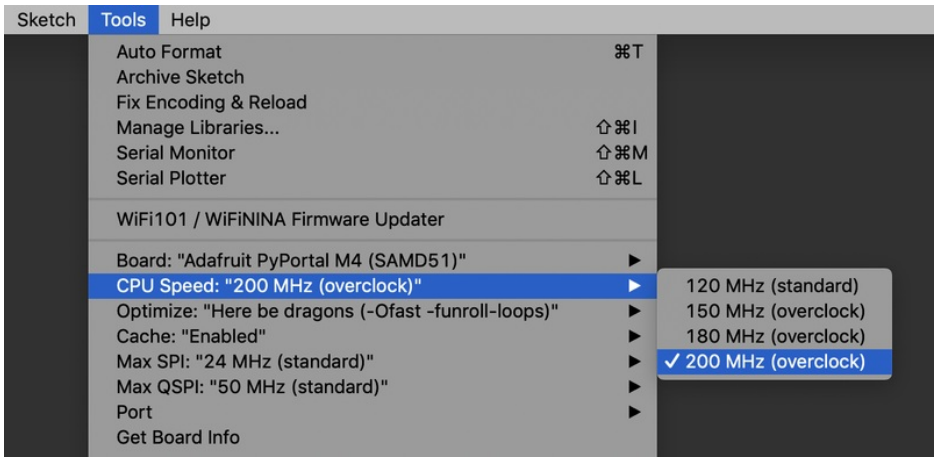
Pretty-Printing out registers

There's *a lot* of registers on the SAMD21, and you often are going through ASF or another framework to get to them. So having a way to see exactly what's going on is handy. This library from drewfish will help a ton!

https://github.com/drewfish/arduino-ZeroRegs (https://adafru.it/Bet)

M4 Performance Options

As of version 1.4.0 of the *Adafruit SAMD Boards* package in the Arduino Boards Manager, some options are available to wring extra performance out of M4-based devices. These are in the *Tools* menu.



All of these performance tweaks involve a degree of uncertainty. There's *no guarantee* of improved performance in any given project, and *some may even be detrimental,* failing to work in part or in whole. If you encounter trouble, **select the default performance settings** and re-upload.

Here's what you get and some issues you might encounter...

CPU Speed (overclocking)

This option lets you adjust the microcontroller core clock...the speed at which it processes instructions...beyond the official datasheet specifications.

Manufacturers often rate speeds conservatively because such devices are marketed for harsh industrial environments...if a system crashes, someone could lose a limb or worse. But most creative tasks are less critical and operate in more comfortable settings, and we can push things a bit if we want more speed.

There is a small but nonzero chance of code **locking up** or **failing to run** entirely. If this happens, try **dialing back the speed by one notch and re-upload**, see if it's more stable.

Much more likely, **some code or libraries may not play well** with the nonstandard CPU speed. For example, currently the NeoPixel library assumes a 120 MHz CPU speed and won't issue the correct data at other settings (this will be worked on). Other libraries may exhibit similar problems, usually anything that strictly depends on CPU timing...you might encounter problems with audio- or servo-related code depending how it's written. **If you encounter such code or libraries, set the CPU speed to the default 120 MHz and re-upload.**

Optimize

There's usually more than one way to solve a problem, some more resource-intensive than others. Since Arduino got its start on resource-limited AVR microcontrollers, the C++ compiler has always aimed for the **smallest compiled program size**. The "Optimize" menu gives some choices for the compiler to take different and often faster approaches, at the expense of slightly larger program size...with the huge flash memory capacity of M4 devices, that's rarely a problem now.

The "**Small**" setting will compile your code like it always has in the past, aiming for the smallest compiled program size.

The "**Fast**" setting invokes various speed optimizations. The resulting program should produce the same results, is slightly larger, and usually (but not always) noticably faster. It's worth a shot!

"**Here be dragons**" invokes some more intensive optimizations...code will be larger still, faster still, but there's a possibility these optimizations could cause unexpected behaviors. *Some code may not work the same as before.* Hence the name. Maybe you'll discover treasure here, or maybe you'll sail right off the edge of the world.

Most code and libraries will continue to function regardless of the optimizer settings. If you do encounter problems, **dial it back one notch and re-upload**.

Cache

This option allows a small collection of instructions and data to be accessed more quickly than from flash memory, boosting performance. It's enabled by default and should work fine with all code and libraries. But if you encounter some esoteric situation, the cache can be disabled, then recompile and upload.

Max SPI and Max QSPI

These should probably be left at their defaults. They're present mostly for our own experiments and can cause **serious headaches**.

Max SPI determines the clock source for the M4's SPI peripherals. Under normal circumstances this allows transfers up to 24 MHz, and should usually be left at that setting. But...if you're using write-only SPI devices (such as TFT or OLED displays), this option lets you drive them faster (we've successfully used 60 MHz with some TFT screens). The caveat is, if using *any* read/write devices (such as an SD card), *this will not work at all...*SPI reads *absolutely* max out at the default 24 MHz setting, and anything else will fail. **Write = OK. Read = FAIL.** This is true *even if your code is using a lower bitrate setting...*just having the different clock source prevents SPI reads.

Max QSPI does similarly for the extra flash storage on M4 "Express" boards. *Very few* Arduino sketches access this storage at all, let alone in a bandwidth-constrained context, so this will benefit next to nobody. Additionally, due to the way clock dividers are selected, this will only provide some benefit when certain "CPU Speed" settings are active. Our PyPortal Animated GIF Display (https://adafru.it/EkO) runs marginally better with it, if using the QSPI flash.

Enabling the Buck Converter on some M4 Boards

If you want to reduce power draw, some of our boards have an inductor so you can use the 1.8V buck converter instead of the built in linear regulator. If the board does have an inductor (see the schematic) you can add the line `SUPC->VREG.bit.SEL = 1;` to your code to switch to it. Note it will make ADC/DAC reads a bit noisier so we don't use it by default. You'll save ~4mA (https://adafru.it/F0H).

Source, Libraries and Settings

At this point you should be able to upload code (such as the “Blink” sketch) to the board, the basics are confirmed working. If not, work though the prior two pages.

Source Code

Source code for this project is found in the Adafruit_Learning_System_Guides (https://adafru.it/Clx) repository on Github, specifically in the M4_Eyes (https://adafru.it/FAD) subdirectory. Here’s a direct download link:

https://adafru.it/FAD

https://adafru.it/FAD

Libraries

The following libraries are used by the eye code. These can be installed through the Arduino Library Manager (**Sketch→Include Library→Manage Libraries**...)

- Adafruit_GFX
- Adafruit_ST7789
- Adafruit_ZeroDMA
- Adafruit_ImageReader
- Adafruit_SPIFlash
- Adafruit_TinyUSB
- SdFat - Adafruit fork (*not* the standard SdFat fork)
- ArduinoJson (*not* Arduino_JSON)

Project Settings

- Tools→CPU Speed→180 MHz (overclock) *(200 MHz is a bit too much for some boards and may lock up, but you can give it a try. We use 180 MHz for our prepackaged .UF2 files since it’s likely to work on more boards in the wild.)*
- Tools→Optimize→Fastest (-Ofast) *(don’t use the “dragon” setting for this, it can cause problems)*
- Tools→USB Stack→TinyUSB *(the code* will not compile *without this selected!)*

Using SPI Flash

One of the best features of the M0 express board is a small SPI flash memory chip built into the board. This memory can be used for almost any purpose like storing data files, Python code, and more. Think of it like a little SD card that is always connected to the board, and in fact with Arduino you can access the memory using a library that is very similar to the Arduino SD card library (https://adafru.it/ucu). You can even read and write files that CircuitPython stores on the flash chip!

To use the flash memory with Arduino you'll need to install the Adafruit SPI Flash Memory library (https://adafru.it/wbt) in the Arduino IDE. Click the button below to download the source for this library, open the zip file, and then copy it into an **Adafruit_SPIFlash** folder (remove the -master GitHub adds to the downloaded zip and folder) in the Arduino library folder on your computer (https://adafru.it/dNR):

https://adafru.it/wbu

https://adafru.it/wbu

Once the library is installed open the Arduino IDE and look for the following examples in the library:

- **fatfs_circuitpython**
- **fatfs_datalogging**
- **fatfs_format**
- **fatfs_full_usage**
- **fatfs_print_file**
- **flash_erase**

These examples allow you to format the flash memory with a FAT filesystem (the same kind of filesystem used on SD cards) and read and write files to it just like a SD card.

Read & Write CircuitPython Files

The **fatfs_circuitpython** example shows how to read and write files on the flash chip so that they're accessible from CircuitPython. This means you can run a CircuitPython program on your board and have it store data, then run an Arduino sketch that uses this library to interact with the same data.

Note that before you use the **fatfs_circuitpython** example you **must** have loaded CircuitPython on your board. Load the latest version of CircuitPython as explained in this guide (https://adafru.it/BeN) first to ensure a CircuitPython filesystem is initialized and written to the flash chip. Once you've loaded CircuitPython then you can run the **fatfs_circuitpython** example sketch.

To run the sketch load it in the Arduino IDE and upload it to the Feather/Metro/ItsyBitsy M0 board. Then open the serial monitor at 115200 baud. You should see the serial monitor display messages as it attempts to read files and write to a file on the flash chip. Specifically the example will look for a **boot.py** and **main.py** file (like what CircuitPython runs when it starts) and print out their contents. Then it will add a line to the end of a **data.txt** file on the board (creating it if it doesn't exist already). After running the sketch you can reload CircuitPython on the board and open the **data.txt** file to read it from CircuitPython!

To understand how to read & write files that are compatible with CircuitPython let's examine the sketch code. First notice an instance of the **Adafruit_M0_Express_CircuitPython** class is created and passed an instance of the flash chip class in the last line below:

```
#define FLASH_SS       SS1                   // Flash chip SS pin.
#define FLASH_SPI_PORT SPI1                  // What SPI port is Flash on?

Adafruit_SPIFlash flash(FLASH_SS, &FLASH_SPI_PORT);     // Use hardware SPI

// Alternatively you can define and use non-SPI pins!
//Adafruit_SPIFlash flash(SCK1, MISO1, MOSI1, FLASH_SS);

// Finally create an Adafruit_M0_Express_CircuitPython object which gives
// an SD card-like interface to interacting with files stored in CircuitPython's
// flash filesystem.
Adafruit_M0_Express_CircuitPython pythonfs(flash);
```

By using this **Adafruit_M0_Express_CircuitPython** class you'll get a filesystem object that is compatible with reading and writing files on a CircuitPython-formatted flash chip. This is very important for interoperability between CircuitPython and Arduino as CircuitPython has specialized partitioning and flash memory layout that isn't compatible with simpler uses of the library (shown in the other examples).

Once an instance of the **Adafruit_M0_Express_CircuitPython** class is created (called **pythonfs** in this sketch) you can go on to interact with it just like if it were the SD card library in Arduino (https://adafru.it/wbw). You can open files for reading & writing, create directories, delete files and directories and more. Here's how the sketch checks if a **boot.py** file exists and prints it out a character at a time:

```
// Check if a boot.py exists and print it out.
if (pythonfs.exists("boot.py")) {
  File bootPy = pythonfs.open("boot.py", FILE_READ);
  Serial.println("Printing boot.py...");
  while (bootPy.available()) {
    char c = bootPy.read();
    Serial.print(c);
  }
  Serial.println();
}
else {
  Serial.println("No boot.py found...");
}
```

Notice the **exists** function is called to check if the boot.py file is found, and then the **open** function is used to open it in read mode. Once a file is opened you'll get a reference to a File class object which you can read and write from as if it were a Serial device (again just like the SD card library, all of the same File class functions are available (https://adafru.it/wbw)). In this case the **available** function will return the number of bytes left to read in the file, and the **read** function will read a character at a time to print it to the serial monitor.

Writing a file is just as easy, here's how the sketch writes to **data.txt**:

```
// Create or append to a data.txt file and add a new line
// to the end of it.  CircuitPython code can later open and
// see this file too!
File data = pythonfs.open("data.txt", FILE_WRITE);
if (data) {
  // Write a new line to the file:
  data.println("Hello CircuitPython from Arduino!");
  data.close();
  // See the other fatfs examples like fatfs_full_usage and fatfs_datalogging
  // for more examples of interacting with files.
  Serial.println("Wrote a new line to the end of data.txt!");
}
else {
  Serial.println("Error, failed to open data file for writing!");
}
```

Again the **open** function is used but this time it's told to open the file for writing. In this mode the file will be opened for appending (i.e. data added to the end of it) if it exists, or it will be created if it doesn't exist. Once the file is open print functions like **print** and **println** can be used to write data to the file (just like writing to the serial monitor). Be sure to **close** the file when finished writing!

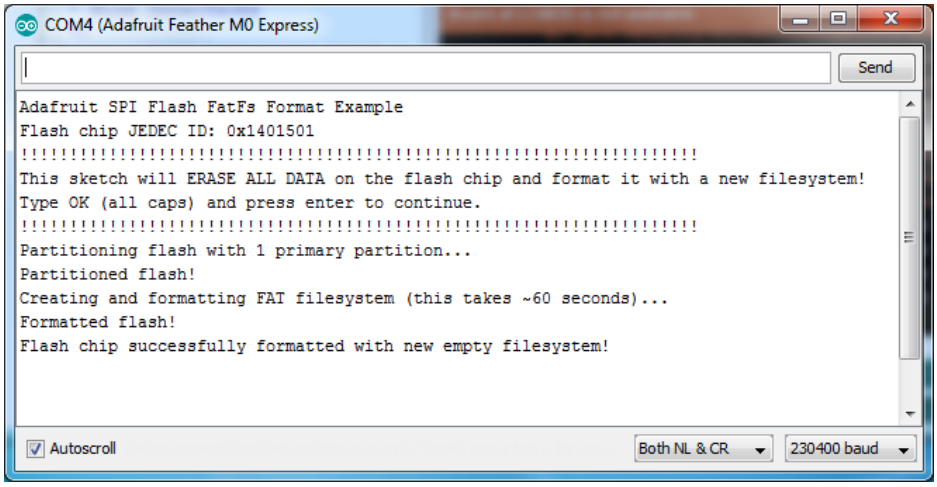
That's all there is to basic file reading and writing. Check out the **fatfs_full_usage** example for examples of even more functions like creating directories, deleting files & directories, checking the size of files, and more! Remember though to interact with CircuitPython files you need to use the **Adafruit_Feather_M0_CircuitPython** class as shown in the **fatfs_circuitpython** example above!

Format Flash Memory

The **fatfs_format** example will format the SPI flash with a new blank filesystem. **Be warned this sketch will delete all data on the flash memory, including any Python code or other data you might have stored!** The format sketch is useful if you'd like to wipe everything away and start fresh, or to help get back in a good state if the memory should get corrupted for some reason.

Be aware too the fatfs_format and examples below are not compatible with a CircuitPython-formatted flash chip! If you need to share data between Arduino & CircuitPython check out the **fatfs_circuitpython** example above.

To run the format sketch load it in the Arduino IDE and upload it to the M0 board. Then open the serial monitor at 115200 baud. You should see the serial monitor display a message asking you to confirm formatting the flash. If you don't see this message then close the serial monitor, press the board's reset button, and open the serial monitor again.



Type **OK** and press enter in the serial monitor input to confirm that you'd like to format the flash memory. **You need to enter OK in all capital letters!**

Once confirmed the sketch will format the flash memory. The format process takes about a minute so be patient as the data is erased and formatted. You should see a message printed once the format process is complete. At this point the flash chip will be ready to use with a brand new empty filesystem.

Datalogging Example

One handy use of the SPI flash is to store data, like datalogging sensor readings. The **fatfs_datalogging** example shows basic file writing/datalogging. Open the example in the Arduino IDE and upload it to your Feather M0 board. Then open the serial monitor at 115200 baud. You should see a message printed every minute as the sketch writes a new line of data to a file on the flash filesystem.

To understand how to write to a file look in the loop function of the sketch:

```
// Open the datalogging file for writing.  The FILE_WRITE mode will open
// the file for appending, i.e. it will add new data to the end of the file.
File dataFile = fatfs.open(FILE_NAME, FILE_WRITE);
// Check that the file opened successfully and write a line to it.
if (dataFile) {
  // Take a new data reading from a sensor, etc.  For this example just
  // make up a random number.
  int reading = random(0,100);
  // Write a line to the file.  You can use all the same print functions
  // as if you're writing to the serial monitor.  For example to write
  // two CSV (commas separated) values:
  dataFile.print("Sensor #1");
  dataFile.print(",");
  dataFile.print(reading, DEC);
  dataFile.println();
  // Finally close the file when done writing.  This is smart to do to make
  // sure all the data is written to the file.
  dataFile.close();
  Serial.println("Wrote new measurement to data file!");
}
```

Just like using the Arduino SD card library you create a **File** object by calling an **open** function and pointing it at the name of the file and how you'd like to open it (**FILE_WRITE** mode, i.e. writing new data to the end of the file). Notice

however instead of calling open on a global SD card object you're calling it on a **fatfs** object created earlier in the sketch (look at the top after the #define configuration values).

Once the file is opened it's simply a matter of calling **print** and **println** functions on the file object to write data inside of it. This is just like writing data to the serial monitor and you can print out text, numeric, and other types of data. Be sure to close the file when you're done writing to ensure the data is stored correctly!

Reading and Printing Files

The **fatfs_print_file** example will open a file (by default the data.csv file created by running the **fatfs_datalogging** example above) and print all of its contents to the serial monitor. Open the **fatfs_print_file** example and load it on your Feather M0 board, then open the serial monitor at 115200 baud. You should see the sketch print out the contents of data.csv (if you don't have a file called data.csv on the flash look at running the datalogging example above first).

To understand how to read data from a file look in the **setup** function of the sketch:

```
// Open the file for reading and check that it was successfully opened.
// The FILE_READ mode will open the file for reading.
File dataFile = fatfs.open(FILE_NAME, FILE_READ);
if (dataFile) {
  // File was opened, now print out data character by character until at the
  // end of the file.
  Serial.println("Opened file, printing contents below:");
  while (dataFile.available()) {
    // Use the read function to read the next character.
    // You can alternatively use other functions like readUntil, readString, etc.
    // See the fatfs_full_usage example for more details.
    char c = dataFile.read();
    Serial.print(c);
  }
}
```

Just like when writing data with the datalogging example you create a **File** object by calling the **open** function on a fatfs object. This time however you pass a file mode of **FILE_READ** which tells the filesystem you want to read data.

After you open a file for reading you can easily check if data is available by calling the **available** function on the file, and then read a single character with the **read** function. This makes it easy to loop through all of the data in a file by checking if it's available and reading a character at a time. However there are more advanced read functions you can use too--see the **fatfs_full_usage** example or even the Arduino SD library documentation (https://adafru.it/ucu) (the SPI flash library implements the same functions).

Full Usage Example

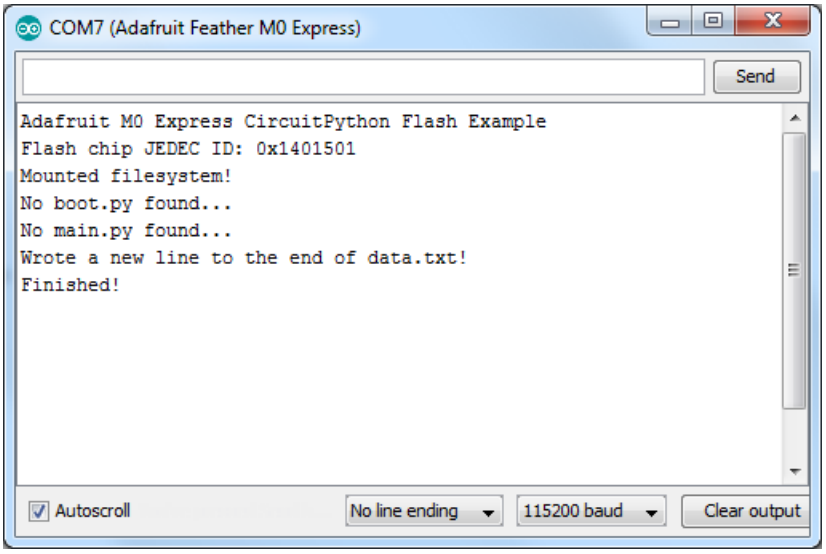
For a more complete demonstration of reading and writing files look at the **fatfs_full_usage** example. This examples uses every function in the library and demonstrates things like checking for the existence of a file, creating directories, deleting files, deleting directories, and more.

Remember the SPI flash library is built to have the same functions and interface as the Arduino SD library (https://adafru.it/ucu) so if you have code or examples that store data on a SD card they should be easy to adapt to use the SPI flash library, just create a fatfs object like in the examples above and use its open function instead of the global SD object's open function. Once you have a reference to a file all of the functions and usage should be the same between the SPI flash and SD libraries!

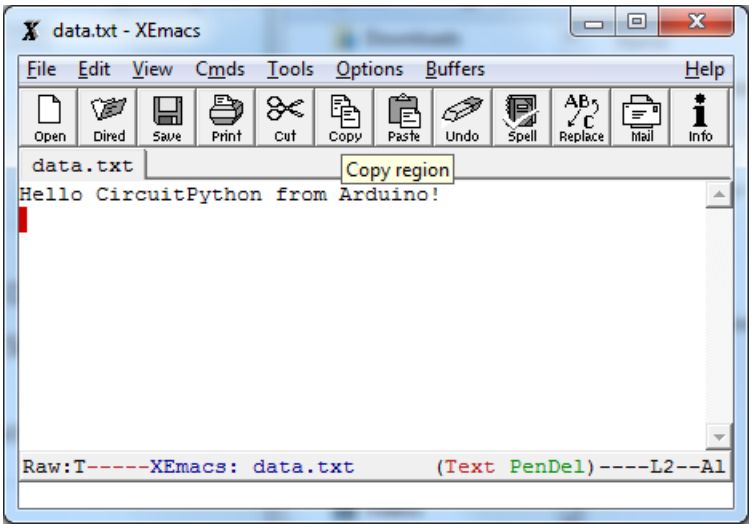
Accessing SPI Flash

Arduino doesn't have the ability to show up as a 'mass storage' disk drive. So instead we must use CircuitPython to do that part for us. Here's the full technique:

- Start the bootloader on the Express board. Drag over the latest **circuitpython** uf2 file
- After a moment, you should see a **CIRCUITPY** drive appear on your hard drive with **boot_out.txt** on it
- Now go to Arduino and upload the **fatfs_circuitpython** example sketch from the Adafruit SPI library. Open the serial console. It will successfully mount the filesystem and write a new line to **data.txt**



- Back on your computer, re-start the Express board bootloader, and re-drag **circuitpython.uf2** onto the **BOOT** drive to reinstall circuitpython
- Check the **CIRCUITPY** drive, you should now see **data.txt** which you can open to read!



Once you have your Arduino sketch working well, for datalogging, you can simplify this procedure by dragging **CURRENT.UF2** off of the **BOOT** drive to make a backup of the current program before loading circuitpython on. Then once you've accessed the file you want, re-drag **CURRENT.UF2** back onto the **BOOT** drive to re-install the Arduino sketch!

Feather HELP!

Even though this FAQ is labeled for Feather, the questions apply to ItsyBitsy's as well!

My ItsyBitsy/Feather stopped working when I unplugged the USB!

A lot of our example sketches have a

```
while (!Serial);
```

line in setup(), to keep the board waiting until the USB is opened. This makes it a lot easier to debug a program because you get to see all the USB data output. If you want to run your Feather without USB connectivity, delete or comment out that line

My Feather never shows up as a COM or Serial port in the Arduino IDE

A vast number of Itsy/Feather 'failures' are due to charge-only USB cables

We get upwards of 5 complaints a day that turn out to be due to charge-only cables!

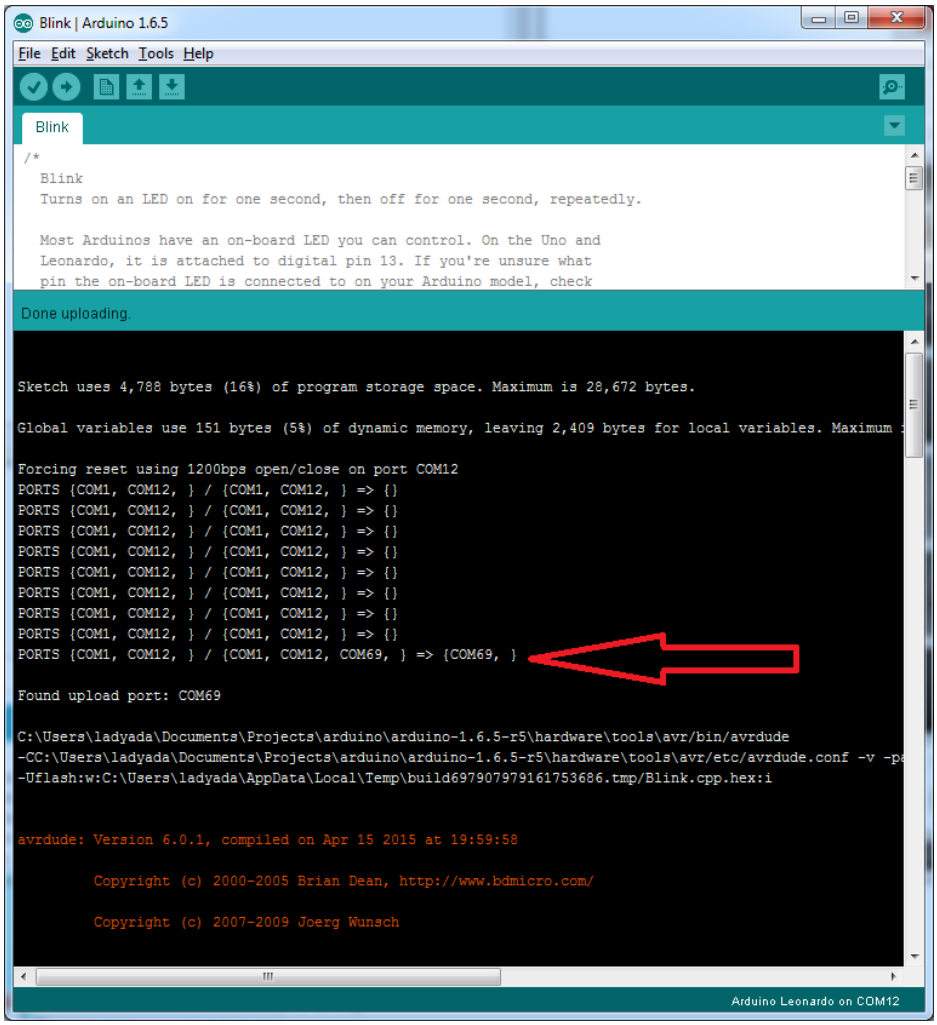
Use only a cable that you **know** is for data syncing

If you have any charge-only cables, cut them in half throw them out. We are serious! They tend to be low quality in general, and will only confuse you and others later, just get a good data+charge USB cable

Ack! I "did something" and now when I plug in the Itsy/Feather, it doesn't show up as a device anymore so I cant upload to it or fix it...

No problem! You can 'repair' a bad code upload easily. Note that this can happen if you set a watchdog timer or sleep mode that stops USB, or any sketch that 'crashes' your board

1. Turn on **verbose upload** in the Arduino IDE preferences
2. Plug in Itsy or Feather 32u4/M0, it won't show up as a COM/serial port that's ok
3. Open up the Blink example (Examples->Basics->Blink)
4. Select the correct board in the Tools menu, e.g. Feather 32u4, Feather M0, Itsy 32u4 or M0 *(physically check* your board to make sure you have the right one selected!)
5. Compile it (make sure that works)
6. Click Upload to attempt to upload the code
7. The IDE will print out a bunch of COM Ports as it tries to upload. **During this time, double-click the reset button, you'll see the red pulsing LED that tells you its now in bootloading mode**
8. The board will show up as the Bootloader COM/Serial port
9. The IDE should see the bootloader COM/Serial port and upload properly



I can't get the Itsy/Feather USB device to show up - I get "USB Device Malfunctioning" errors!

This seems to happen when people select the wrong board from the Arduino Boards menu.

If you have a Feather 32u4 (look on the board to read what it is you have) Make sure you select **Feather 32u4** for ATMega32u4 based boards! Do not use anything else, do not use the 32u4 breakout board line.

If you have a Feather M0 (look on the board to read what it is you have) Make sure you select **Feather M0** - do not

use 32u4 or Arduino Zero

If you have a ItsyBitsy M0 (look on the board to read what it is you have) Make sure you select **ItsyBitsy M0** - do not use 32u4 or Arduino Zero

I'm having problems with COM ports and my Itsy/Feather 32u4/M0

Theres **two** COM ports you can have with the 32u4/M0, one is the **user port** and one is the **bootloader port**. They are not the same COM port number!

When you upload a new user program it will come up with a user com port, particularly if you use Serial in your user

program.

If you crash your user program, or have a program that halts or otherwise fails, the user COM port can disappear.

When the user COM port disappears, Arduino will not be able to automatically start the bootloader and upload new software.

So you will need to help it by performing the click-during upload procedure to re-start the bootloader, and upload something that is known working like "Blink"

I don't understand why the COM port disappears, this does not happen on my Arduino UNO!

UNO-type Arduinos have a *seperate* serial port chip (aka "FTDI chip" or "Prolific PL2303" etc etc) which handles all serial port capability seperately than the main chip. This way if the main chip fails, you can always use the COM port.

M0 and 32u4-based Arduinos do not have a seperate chip, instead the main processor performs this task for you. It allows for a lower cost, higher power setup...but requires a little more effort since you will need to 'kick' into the bootloader manually once in a while

I'm trying to upload to my 32u4, getting "avrdude: butterfly_recv(): programmer is not responding" errors

This is likely because the bootloader is not kicking in and you are accidentally **trying to upload to the wrong COM port**

The best solution is what is detailed above: manually upload Blink or a similar working sketch by hand by manually launching the bootloader

I'm trying to upload to my Feather M0, and I get this error "Connecting to programmer: .avrdude: butterfly_recv(): programmer is not responding"

You probably don't have Feather M0 selected in the boards drop-down. Make sure you selected Feather M0.

I'm trying to upload to my Feather and i get this error "avrdude: ser_recv(): programmer is not responding"

You probably don't have Feather M0 / Feather 32u4 selected in the boards drop-down. Make sure you selected Feather M0 (or Feather 32u4).

I attached some wings to my Feather and now I can't read the battery voltage!

Make sure your Wing doesn't use pin #9 which is the analog sense for the lipo battery!

The yellow LED Is flickering on my Feather, but no battery is plugged in, why is that?

The charge LED is automatically driven by the Lipoly charger circuit. It will try to detect a battery and is expecting one to be attached. If there isn't one it may flicker once in a while when you use power because it's trying to charge a (non-existant) battery.

It's not harmful, and its totally normal!

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

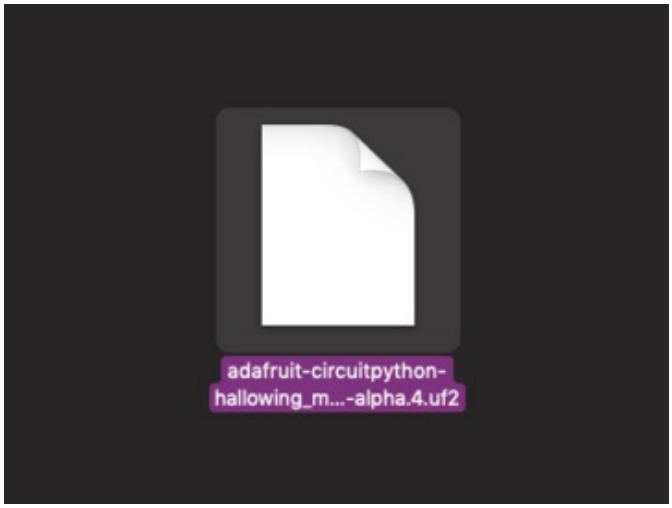
The following instructions will show you how to install CircuitPython. If you've already installed CircuitPython but are looking to update it or reinstall it, the same steps work for that as well!

Set up CircuitPython Quick Start!

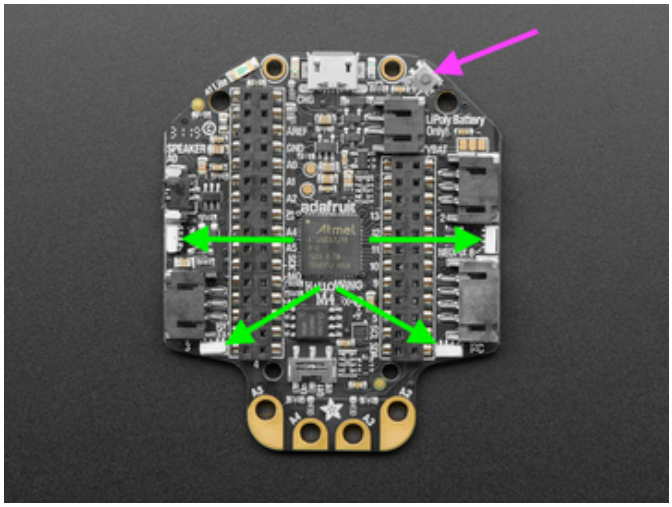
Follow this quick step-by-step for super-fast Python power :)

https://adafru.it/FQg

https://adafru.it/FQg



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

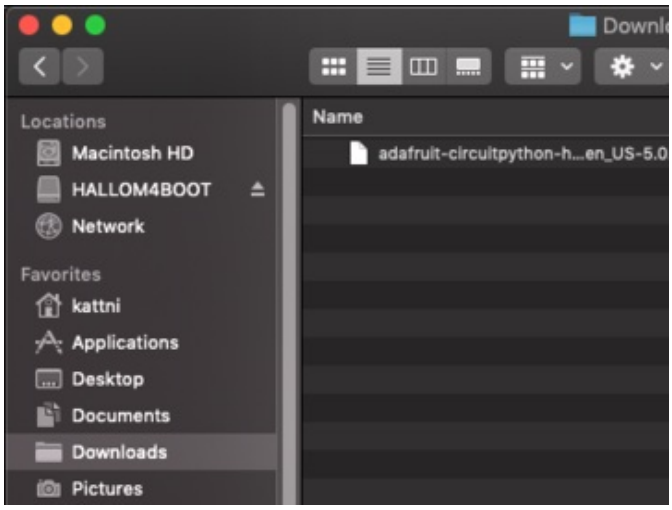


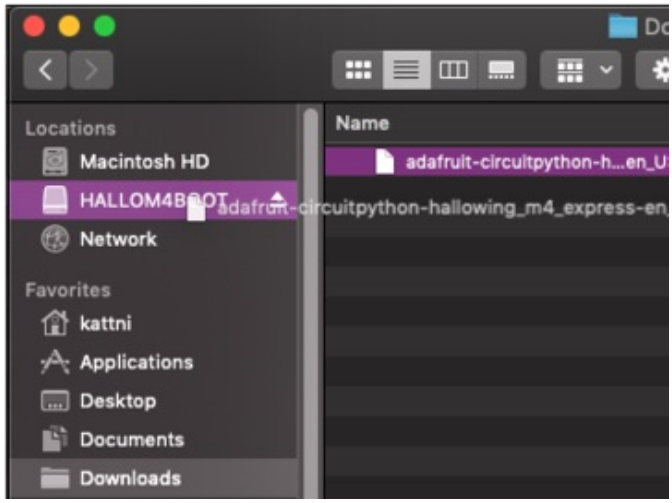
Plug your Hallowing M4 into your computer using a known-good USB cable.

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 **Reset** button next to the USB connector (magenta arrow) on your board, and you will see the four NeoPixel RGB LEDs (green arrows) turn green. If they turn red, check the USB cable, try another USB port, etc. **Note:** The little red LED next to the USB connector will be dim red, and the little yellow LED on the opposite side will flash yellow. That's ok!

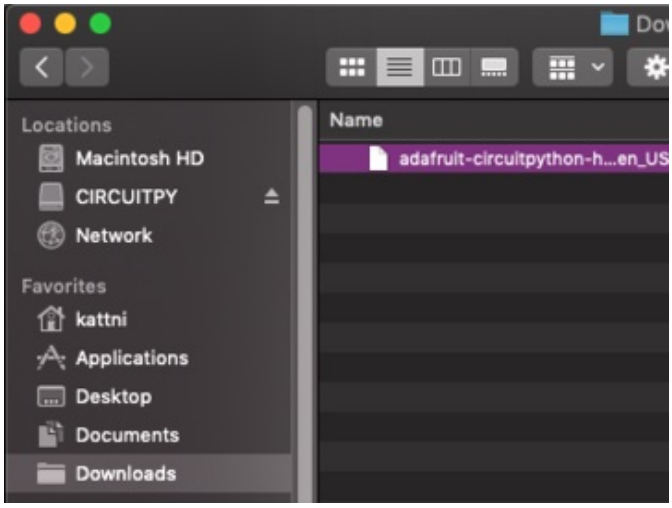
If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!





You will see a new disk drive appear called **HALLOM4BOOT**.

Drag the **adafruit_circuitpython_hallowing_m4_etc.uf2** file to **HALLOM4BOOT.**



The LED will flash. Then, the **HALLOM4BOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

If you haven't added any code to your board, the only file that will be present is **boot_out.txt**. This is absolutely normal! It's time for you to add your **code.py** and get started!

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

Installing Mu Editor

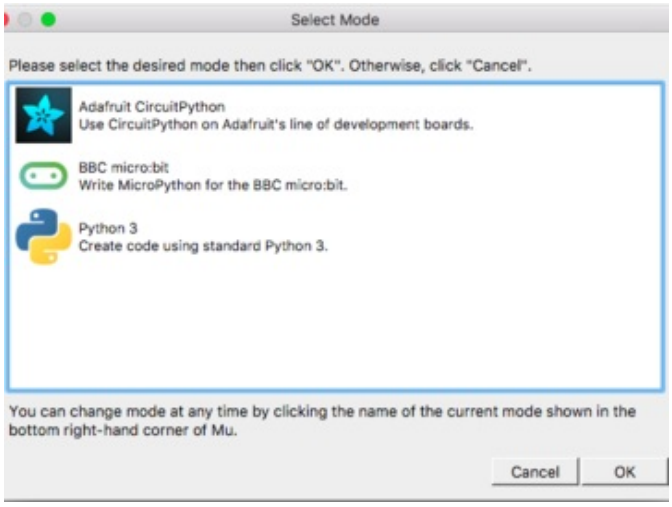
Mu is a simple code editor that works with the Adafruit CircuitPython boards. It's written in Python and works on Windows, MacOS, Linux and Raspberry Pi. The serial console is built right in so you get immediate feedback from your board's serial output!

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

Download and Install Mu

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

Using Mu

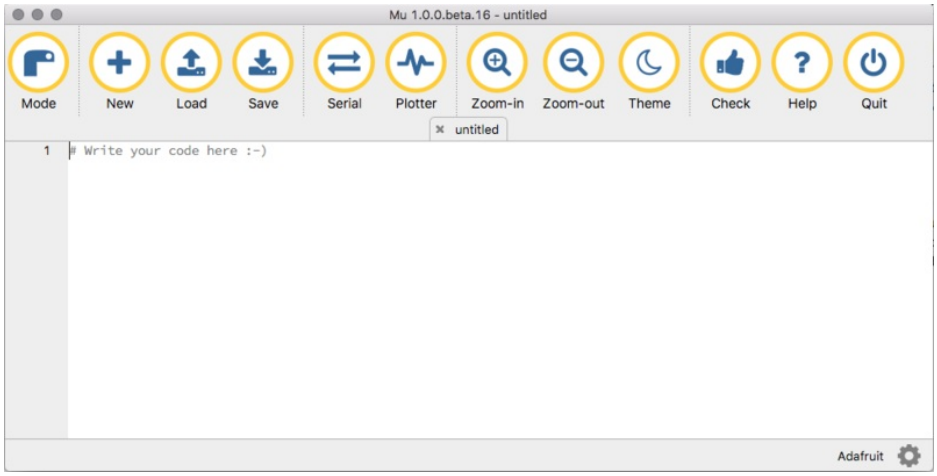


The first time you start Mu, you will be prompted to select your 'mode' - you can always change your mind later. For now please select **Adafruit**!

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

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

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



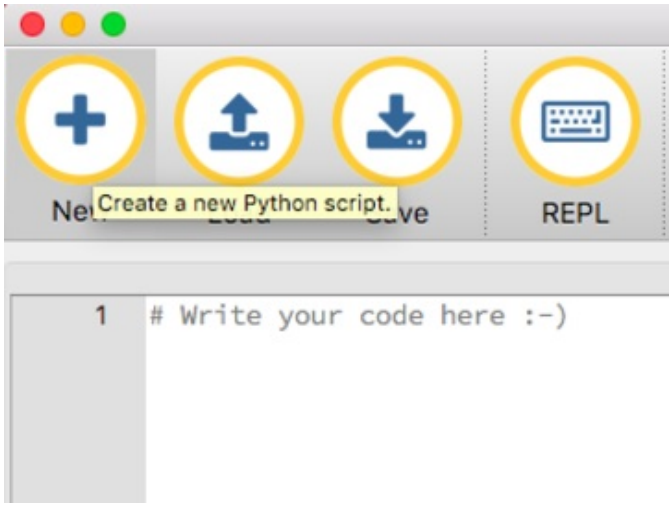
Creating and Editing Code

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

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

If you don't or can't use Mu, there are basic text editors built into every operating system such as Notepad on Windows, TextEdit on Mac, and gedit on Linux. However, many of these editors don't write back changes immediately to files that you edit. That can cause problems when using CircuitPython. See the Editing Code (https://adafru.it/id3) section below. If you want to skip that section for now, make sure you do "Eject" or "Safe Remove" on Windows or "sync" on Linux after writing a file if you aren't using Mu. (This is not a problem on MacOS.)

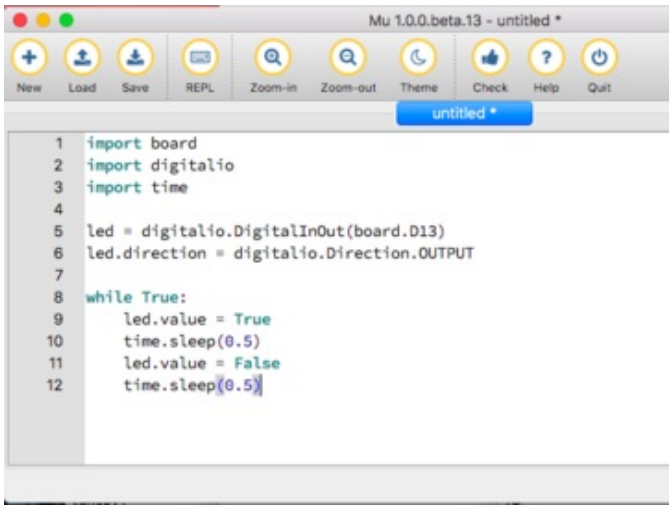
Creating Code



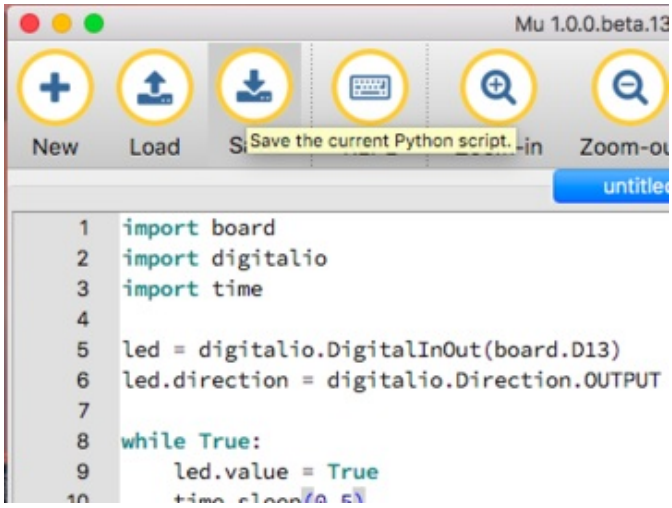
Open your editor, and create a new file. If you are using Mu, click the **New** button in the top left

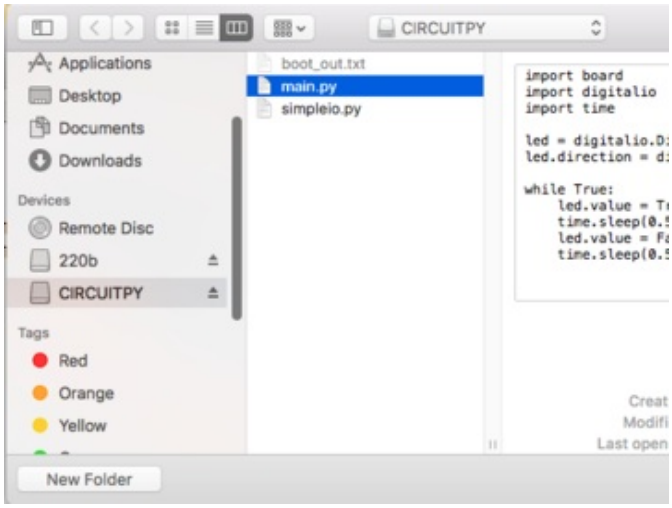
Copy and paste the following code into your editor:

```
import board
import digitalio
import time

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

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



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



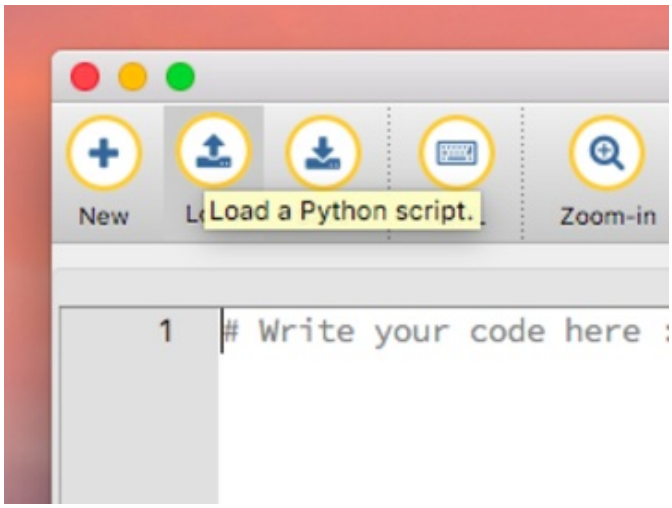


Save this file as **code.py** on your CIRCUITPY drive.

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

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

Editing Code



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

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

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

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

Don't Click Reset or Unplug!

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

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

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

There are a few ways to avoid this:

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

Recommended editors:

- **mu** (https://adafru.it/Be6) is an editor that safely writes all changes (it's also our recommended editor!)
- **emacs** (https://adafru.it/xNA) is also an editor that will fully write files on save (https://adafru.it/Be7)
- **Sublime Text (https://adafru.it/xNB)** safely writes all changes
- **Visual Studio Code (https://adafru.it/Be9)** appears to safely write all changes
- **gedit** on Linux appears to safely write all changes

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

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

We *don't* recommend these editors:

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

2. Eject or Sync the Drive After Writing

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

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

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

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

Back to Editing Code...

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

```
import board
import digitalio
import time

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

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

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

Exploring Your First CircuitPython Program

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

Here is the original code again:

```
import board
import digitalio
import time

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

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

Imports & Libraries

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

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

The `import` statements 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!

What if I don't have the loop?

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

The simplest loop would be:

```
while True:
    pass
```

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

See also the Behavior section in the docs.

More Changes

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

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

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

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

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

Naming Your Program File

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

Connecting to the Serial Console

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

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

This line would result in:

```
Hello, world!
```

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

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

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

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

```
sudo apt purge modemmanager
```

Are you using Mu?

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

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

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

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

Setting Permissions on Linux

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

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

```
sudo adduser $USER dialout
```

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

Using Something Else?

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

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

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

Interacting with the Serial Console

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

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

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

```
import board
import digitalio
import time

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

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

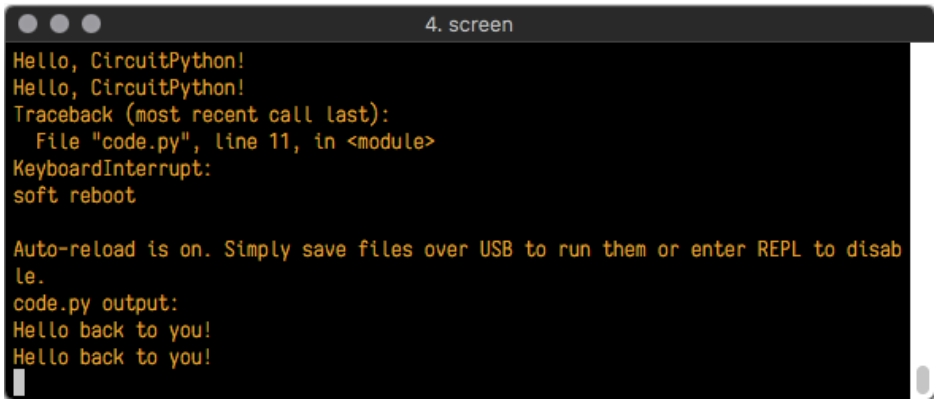
Save your file.

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!

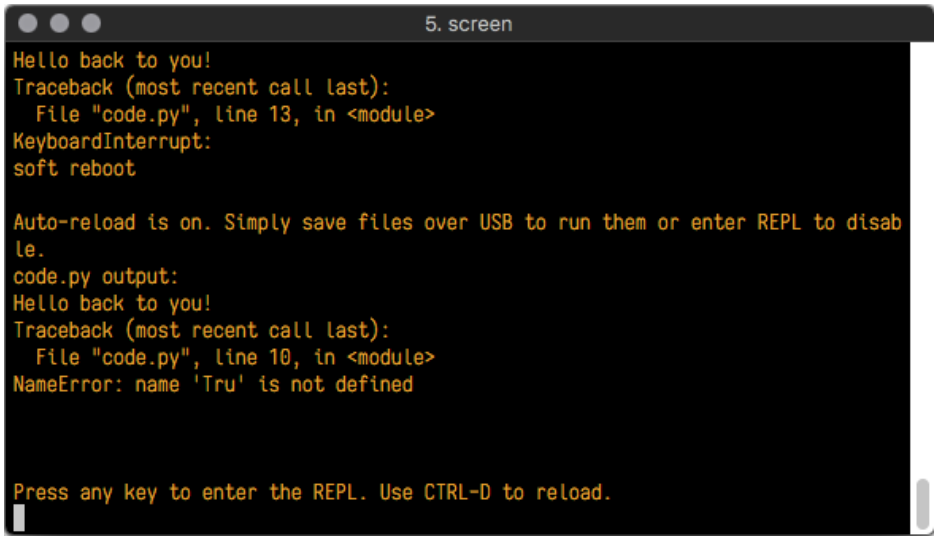


The `Traceback (most recent call last):` is telling you the last thing your board was doing before you saved your file. This is normal behavior and will happen every time the board resets. This is really handy for troubleshooting. Let's introduce an error so we can see how it is used.

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

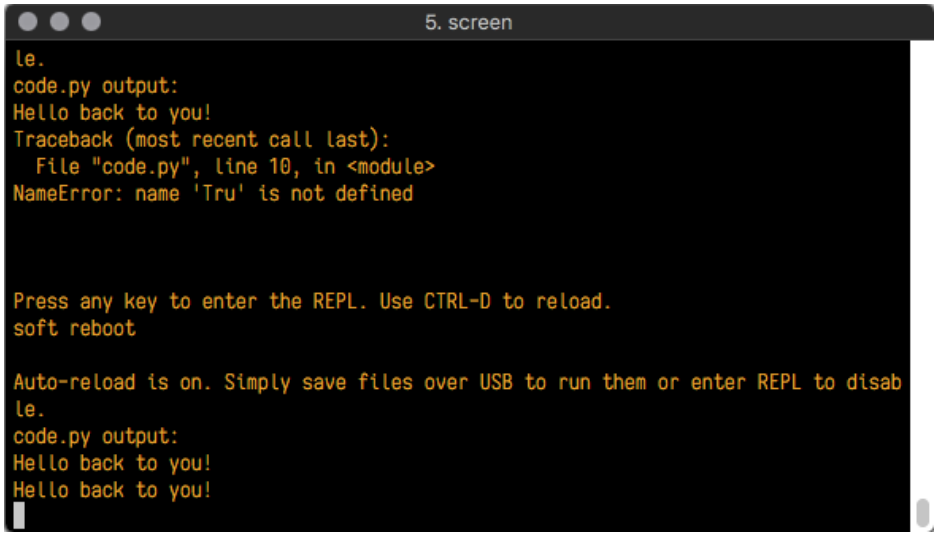
Save your file. You will notice that your red LED will stop blinking, and you may have a colored status LED blinking at you. This is because the code is no longer correct and can no longer run properly. We need to fix it!

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!

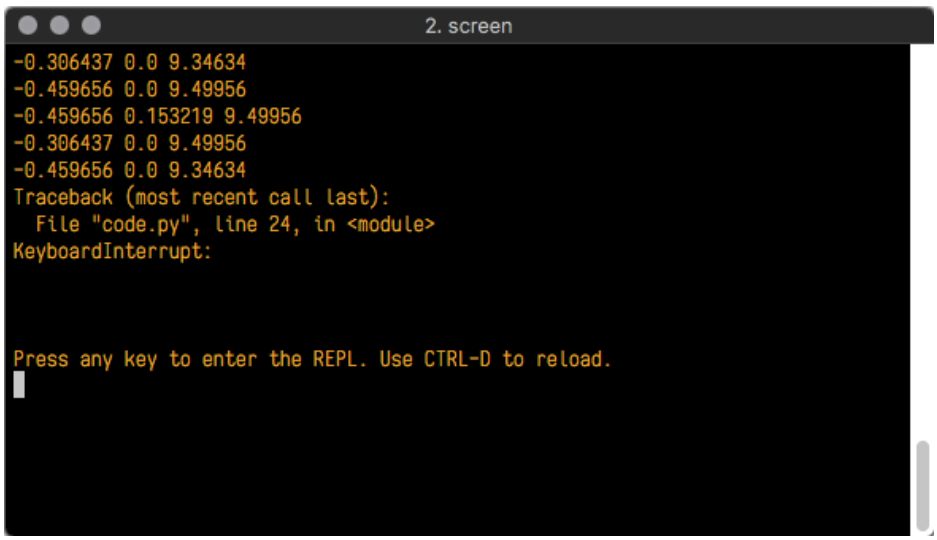
The REPL

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.

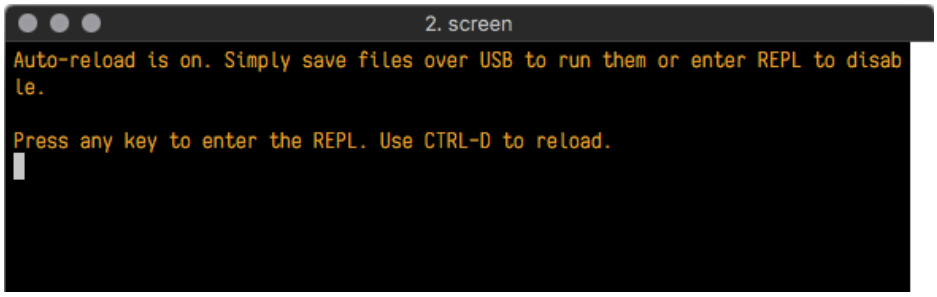
To use the REPL, you first need to be connected to the serial console. Once that connection has been established, you'll want to press **Ctrl** + **C**.

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
```

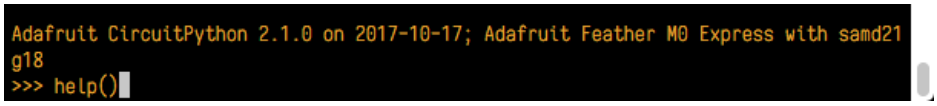
This line tells you the version of CircuitPython you're using and when it was released. Next, it gives you the type of board you're using and the type of microcontroller the board uses. Each part of this may be different for your board depending on the versions you're working with.

This is followed by the CircuitPython prompt.

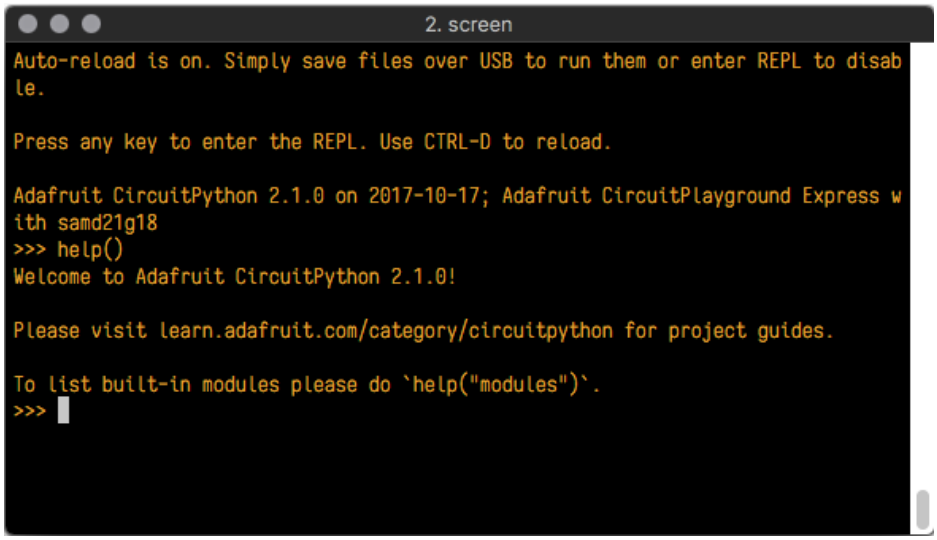
```
>>>
```

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.



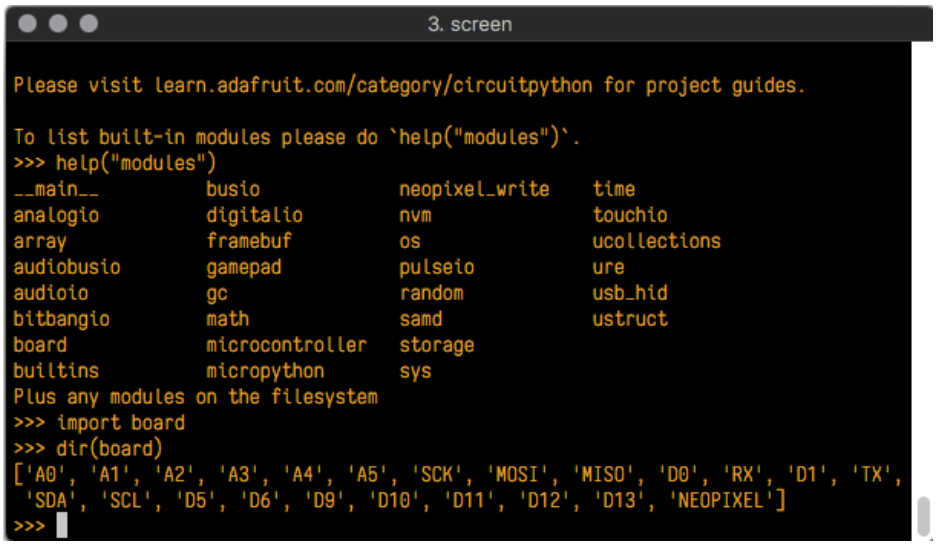
First part of the message is another reference to the version of CircuitPython you're using. Second, a URL for the CircuitPython related project guides. Then... wait. What's this? `To list built-in modules, please do `help("modules")`.` Remember the libraries you learned about while going through creating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

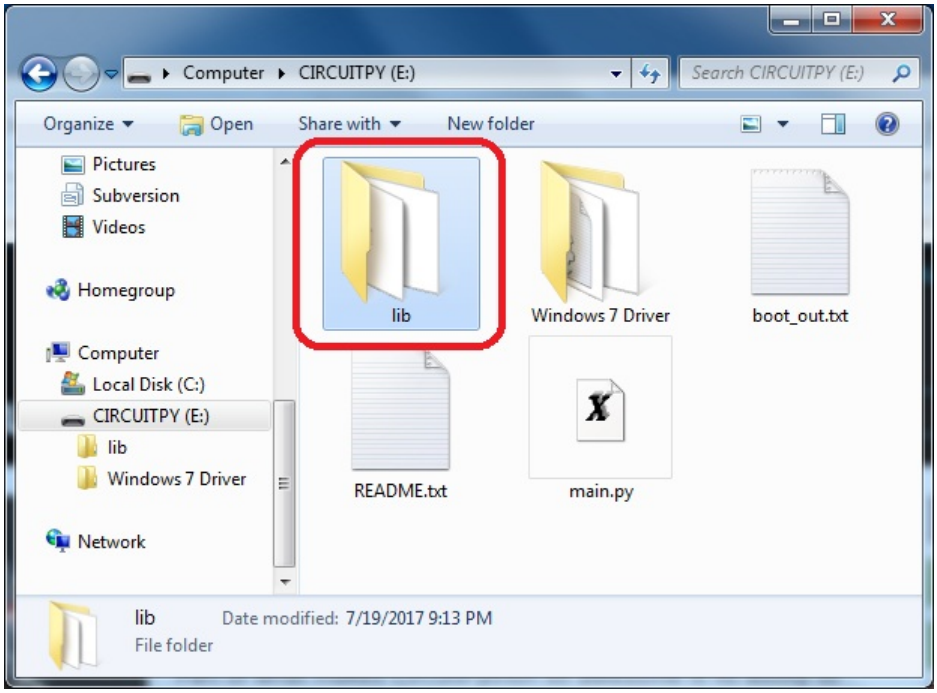
You can return to the REPL at any time!

CircuitPython Libraries

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

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

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



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

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

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

Installing the CircuitPython Library Bundle

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

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

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

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

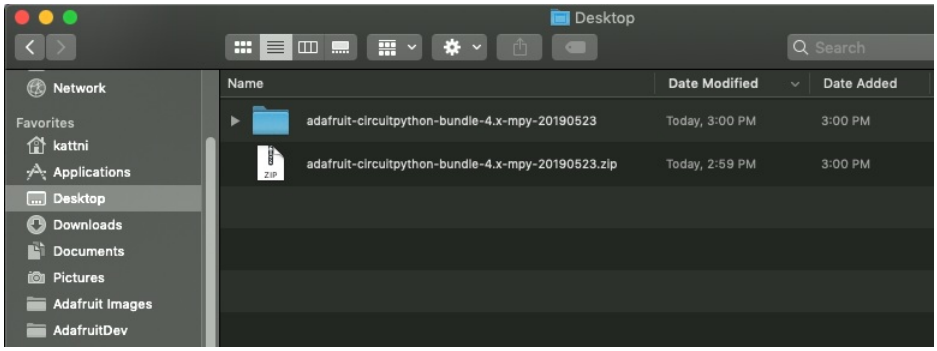
https://adafru.it/ENC

https://adafru.it/ENC

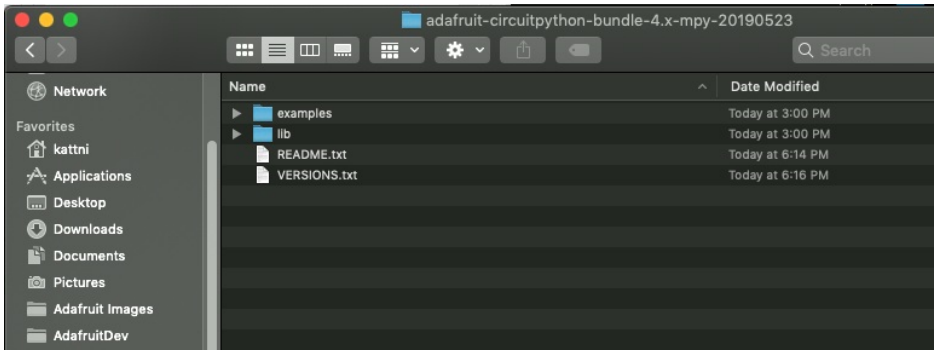
If you need another version, you can also visit the bundle release page (https://adafru.it/Ayy) which will let you select exactly what version you're looking for, as well as information about changes.

Either way, download the version that matches your CircuitPython firmware version. If you don't know the version, look at the initial prompt in the CircuitPython REPL, which reports the version. For example, if you're running v4.0.1, download the 4.x library bundle. There's also a **py** bundle which contains the uncompressed python files, you probably *don't* want that unless you are doing advanced work on libraries.

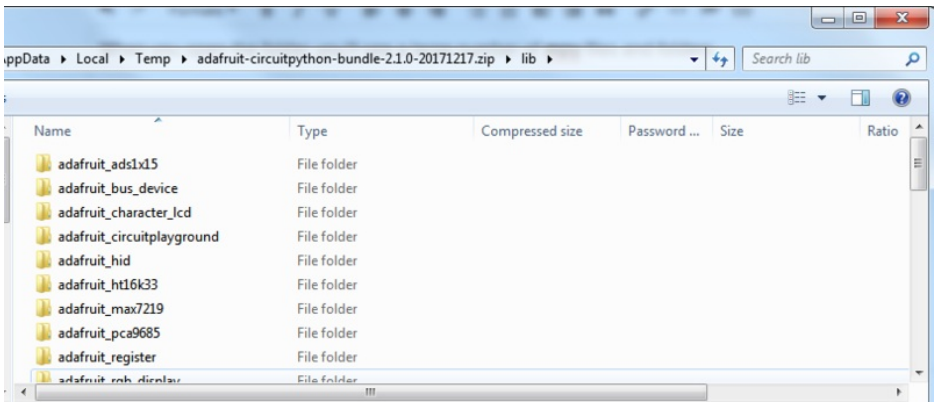
After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.



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



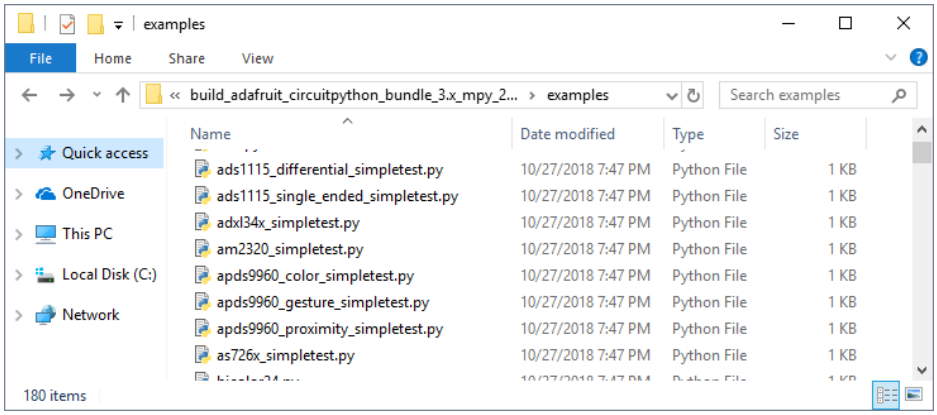
Now open the lib folder. When you open the folder, you'll see a large number of **mpy** files and folders



Example Files

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

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



Copying Libraries to Your Board

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

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

Example: `ImportError` Due to Missing Library

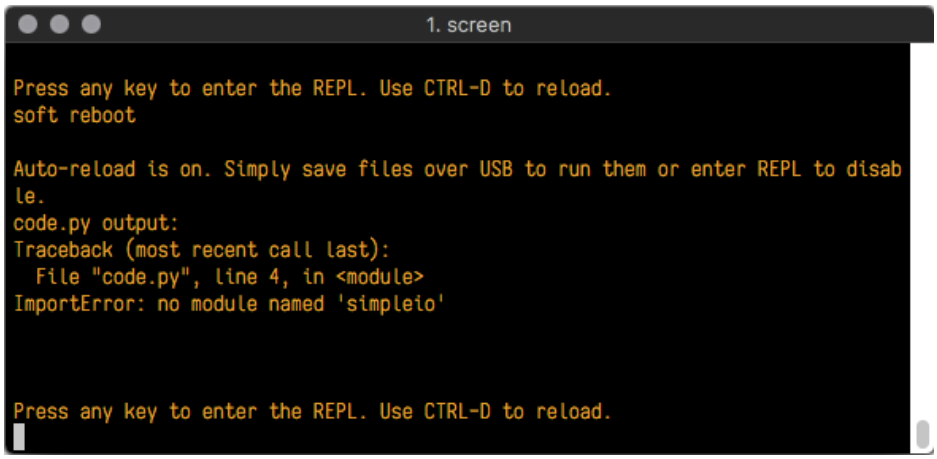
If you choose to load libraries as you need them, you may write up code that tries to use a library you haven't yet loaded. We're going to demonstrate what happens when you try to utilise a library that you don't have loaded on your board, and cover the steps required to resolve the issue.

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

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

```
import board
import time
import simpleio

led = simpleio.DigitalOut(board.D13)

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

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



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.

You may end up running out of space on your Trinket M0 or Gemma M0 even if you only load libraries as you need them. There are a number of steps you can use to try to resolve this issue. You'll find them in the Troubleshooting page in the Learn guides for your board.

Updating CircuitPython Libraries/Examples

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

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

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

Troubleshooting

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

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

Always Run the Latest Version of CircuitPython and Libraries

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

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

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

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

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

CPLAYBOOT, TRINKETBOOT, FEATHERBOOT, or GEMMABOOT Drive Not Present

You may have a different board.

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

MakeCode

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

Windows 10

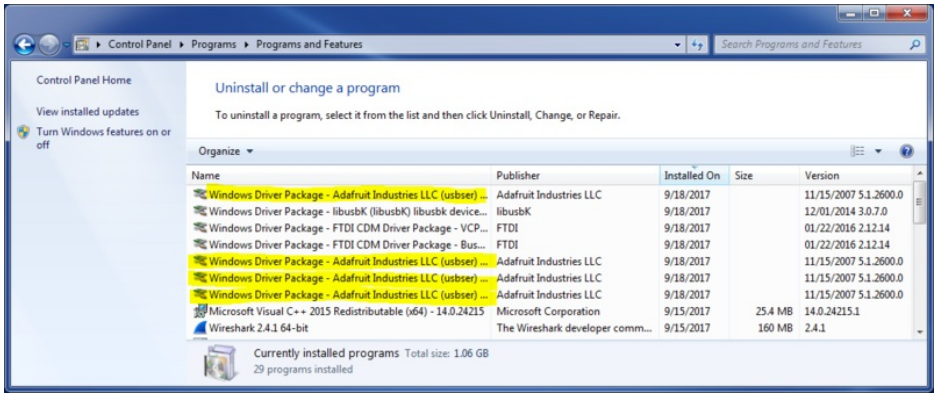
Did you install the Adafruit Windows Drivers package by mistake? You don't need to install this package on Windows 10 for most Adafruit boards. The old version (v1.5) can interfere with recognizing your device. Go to **Settings** -> **Apps**

and uninstall all the "Adafruit" driver programs.

Windows 7

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 ...".

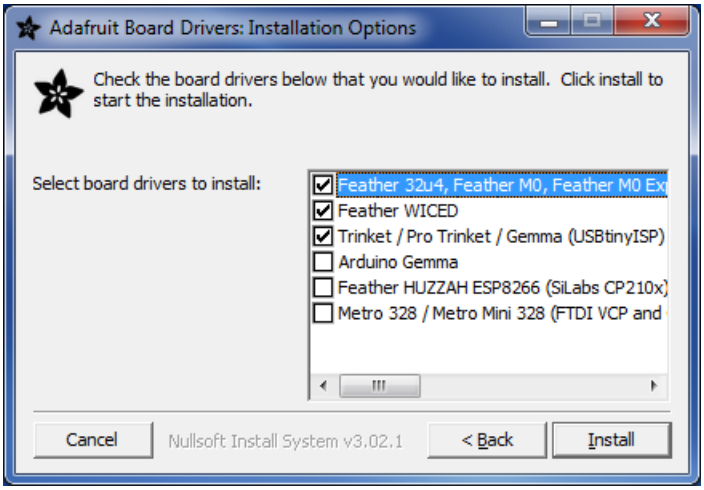


- Now install the new 2.3.0.0 (or higher) Adafruit Windows Drivers Package:

https://adafru.it/AB0

https://adafru.it/AB0

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



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

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

Windows Explorer Locks Up When Accessing `boardnameBOOT` Drive

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

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

CIRCUITPY Drive Does Not Appear

Kaspersky anti-virus can block the appearance of the `CIRCUITPY` drive. We haven't yet figured out a settings change that prevents this. Complete uninstallation of Kaspersky fixes the problem.

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

Serial Console in Mu Not Displaying Anything

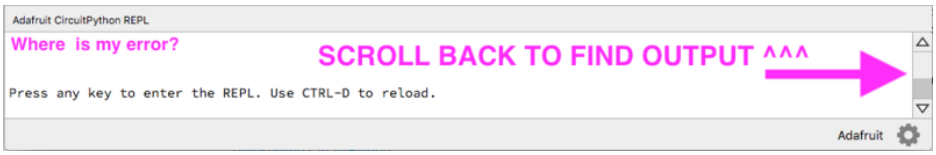
There are times when the serial console will accurately not display anything, such as, when no code is currently running, or when code with no serial output is already running before you open the console. However, if you find yourself in a situation where you feel it should be displaying something like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

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

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



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

More complex errors take even more lines!

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



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

CircuitPython RGB Status Light

The Feather M0 Express, Feather M4 Express, Metro M0 Express, Metro M4 Express, ItsyBitsy M0 Express, ItsyBitsy M4 Express, Gemma M0, and Trinket M0 all have a single NeoPixel or DotStar RGB LED on the board that indicates the status of CircuitPython.

Circuit Playground Express does NOT have a status LED. The LEDs will pulse green when in the bootloader. They do NOT indicate any status while running CircuitPython.

Here's what the colors and blinking mean:

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

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

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

These are followed by flashes indicating the line number, including place value. **WHITE** flashes are thousands' place, **BLUE** are hundreds' place, **YELLOW** are tens' place, and **CYAN** are one's place. So for example, an error on line 32 would flash **YELLOW** three times and then **CYAN** two times. Zeroes are indicated by an extra-long dark gap.

ValueError: Incompatible `.mpy` file.

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

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

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

CIRCUITPY Drive Issues

You may find that you can no longer save files to your `CIRCUITPY` drive. You may find that your `CIRCUITPY` stops showing up in your file explorer, or shows up as `NO_NAME`. These are indicators that your filesystem has issues.

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

If still broken - When the `CIRCUITPY` disk is not safely ejected before being reset by the button or being disconnected from USB, it may corrupt the flash drive. It can happen on Windows, Mac or Linux.

In this situation, the board must be completely erased and CircuitPython must be reloaded onto the board.

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

Easiest Way: Use `storage.erase_filesystem()`

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

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

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

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

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

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

1. Download the correct erase file:

https://adafru.it/Adl

https://adafru.it/Adl

https://adafru.it/AdJ

https://adafru.it/AdJ

https://adafru.it/EVK

https://adafru.it/EVK

https://adafru.it/AdK

https://adafru.it/AdK

https://adafru.it/EoM

https://adafru.it/EoM

https://adafru.it/DjD

https://adafru.it/DjD

https://adafru.it/DBA

https://adafru.it/DBA

https://adafru.it/Eca

https://adafru.it/Eca

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

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

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

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

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

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

1. Download the erase file:

https://adafru.it/AdL

https://adafru.it/AdL

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

5. Drag the appropriate latest release CircuitPython (https://adafru.it/Amd) `.uf2` file to the *boardname*`BOOT` drive.

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

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

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

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

Just follow these directions to reload CircuitPython using `bossac` (https://adafru.it/Bed), which will erase and re-create `CIRCUITPY`.

Running Out of File Space on Non-Express Boards

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

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

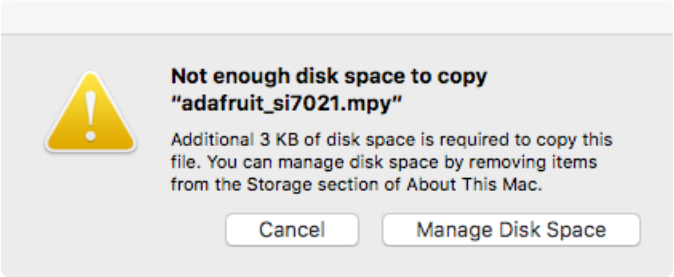
Delete something!

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

Use tabs

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

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

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

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

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
```

(Replace `foo.mpy` with the name of the file you want to copy.) Or to copy a folder and all of its child files/folders use a command like:

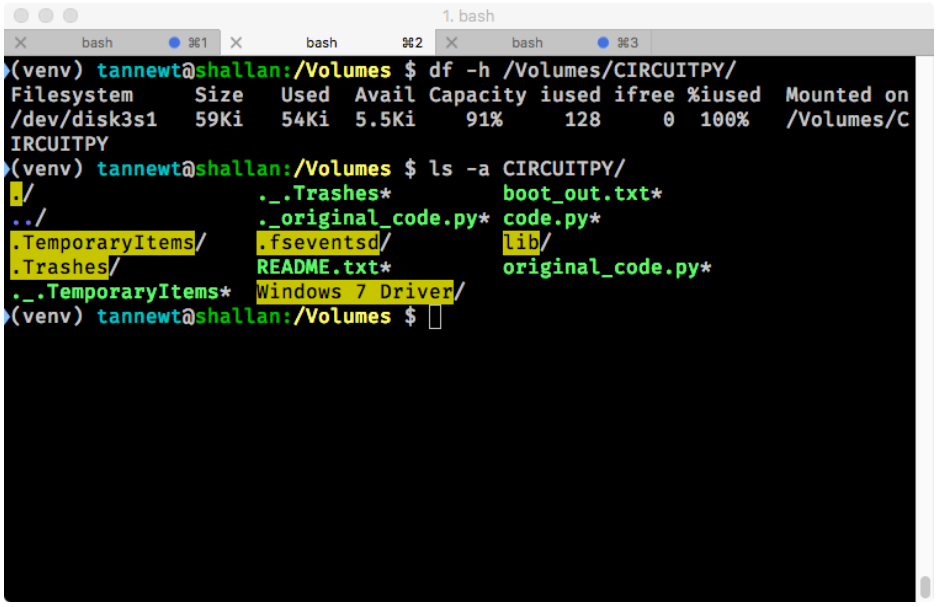
```
cp -rX folder_to_copy /Volumes/CIRCUITPY
```

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

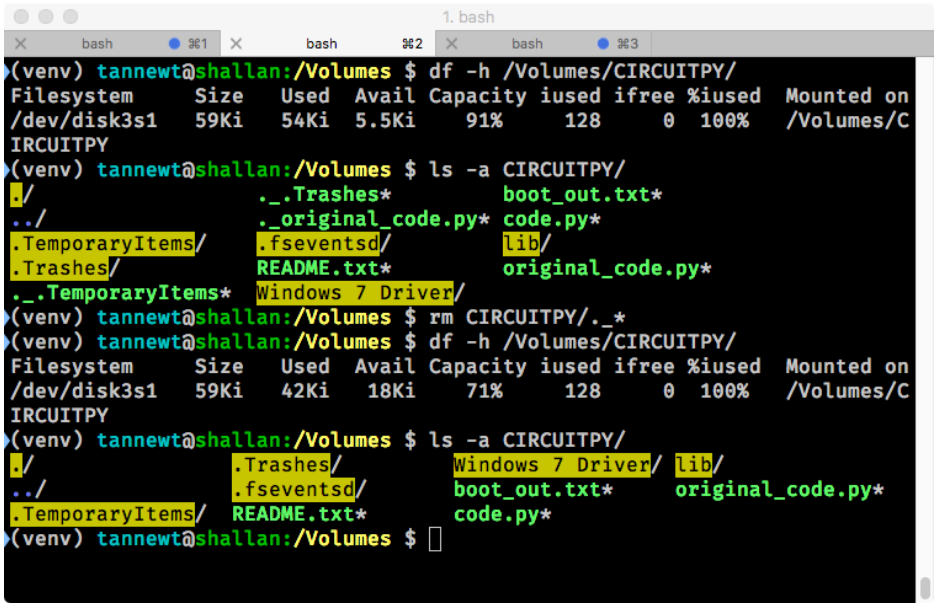
```
# if lib does not exist, you'll create a file named lib !
cp -X foo.mpy /Volumes/CIRCUITPY/lib
# This is safer, and will complain if a lib folder does not exist.
cp -X foo.mpy /Volumes/CIRCUITPY/lib/
```

Other 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!

Uninstalling CircuitPython

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

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

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

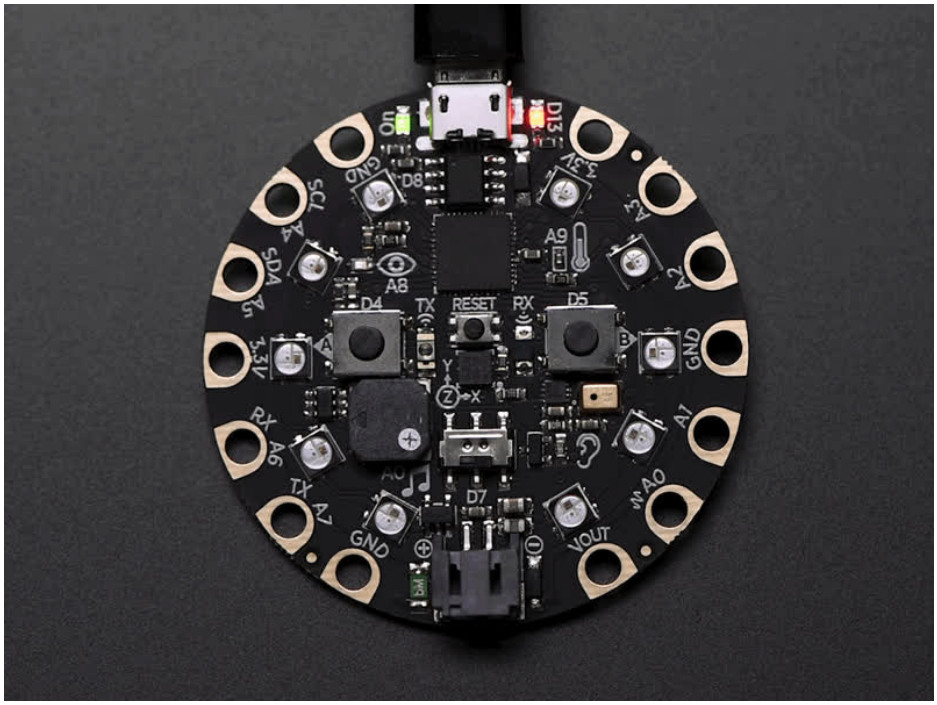
Backup Your Code

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

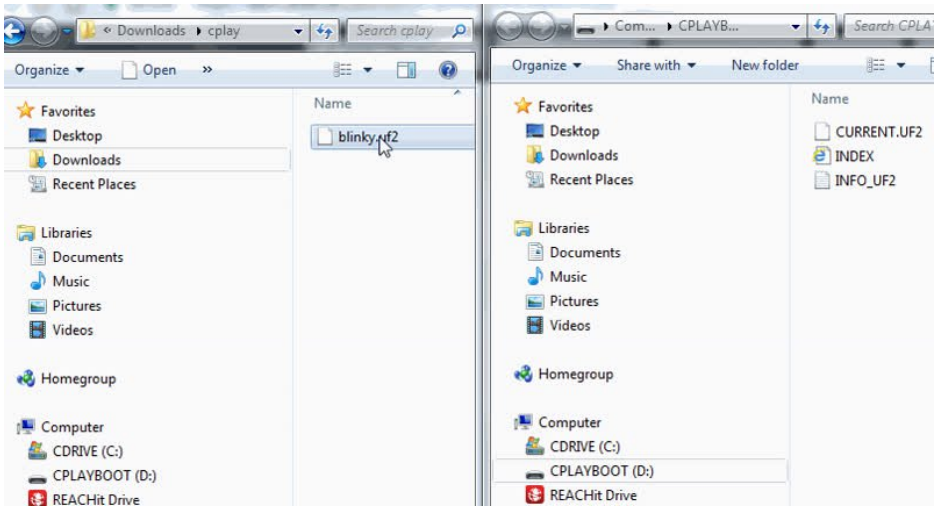
Moving to MakeCode

If you want to go back to using MakeCode, its really easy. Visit makecode.adafruit.com (https://adafru.it/wpC) and find the program you want to upload. Click Download to download the **.uf2** file that is generated by MakeCode.

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



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



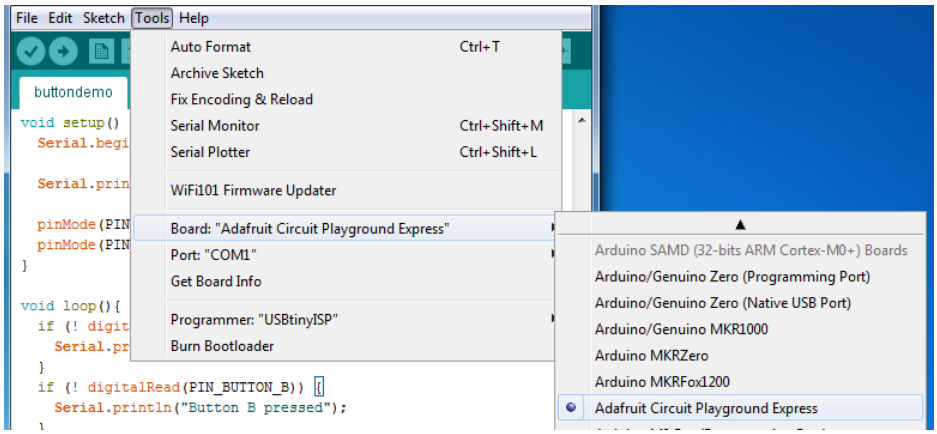
Your MakeCode is now running and CircuitPython has been removed. Going forward you only have to **single click** the reset button

Moving to Arduino

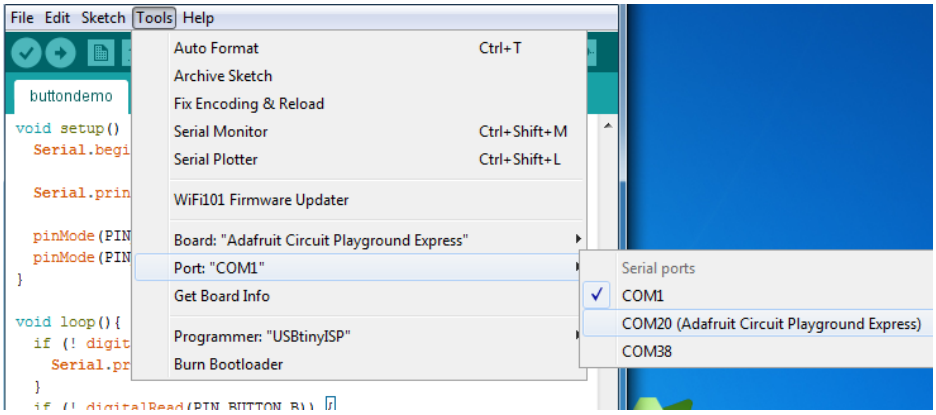
If you want to change your firmware to Arduino, it's also pretty easy.

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

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



Select the correct matching Port:



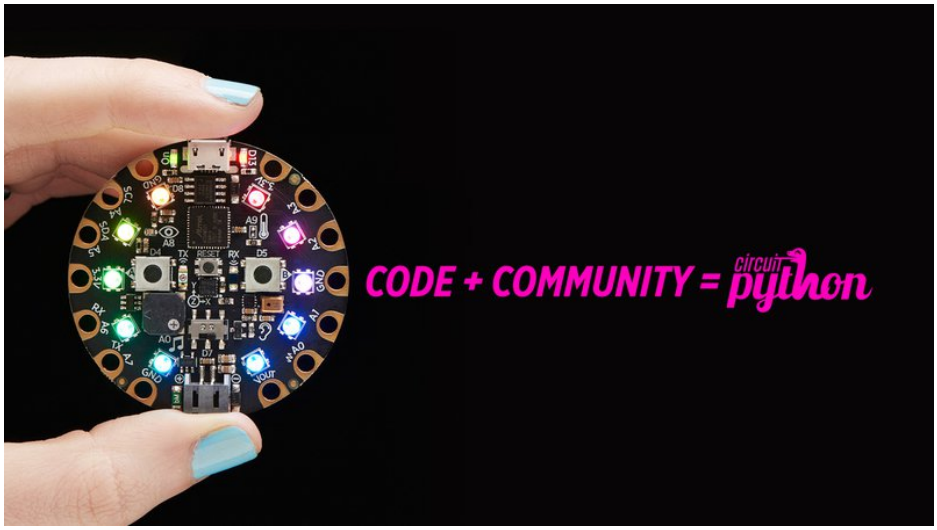
Create a new simple Blink sketch example:

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

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

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

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

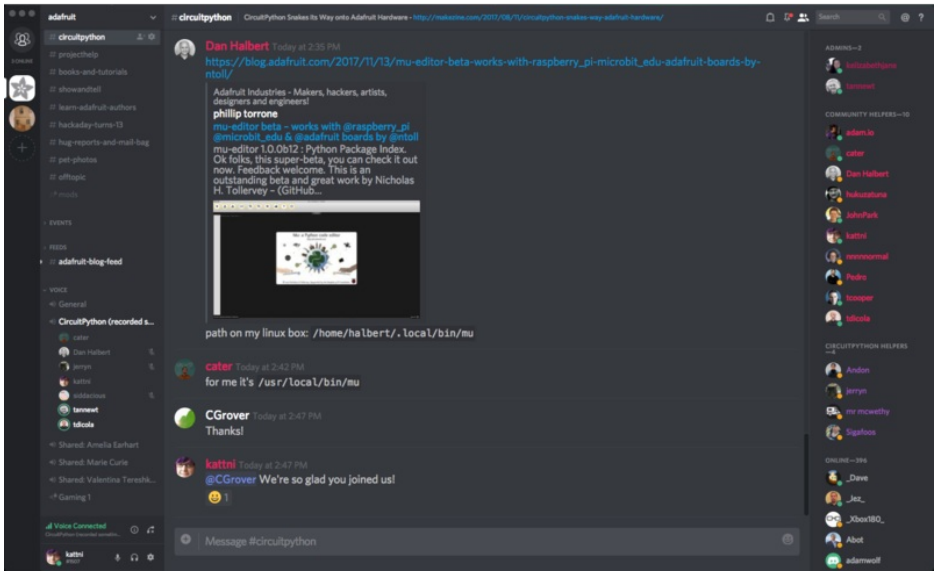
Welcome to the Community!



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

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

Adafruit Discord



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

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

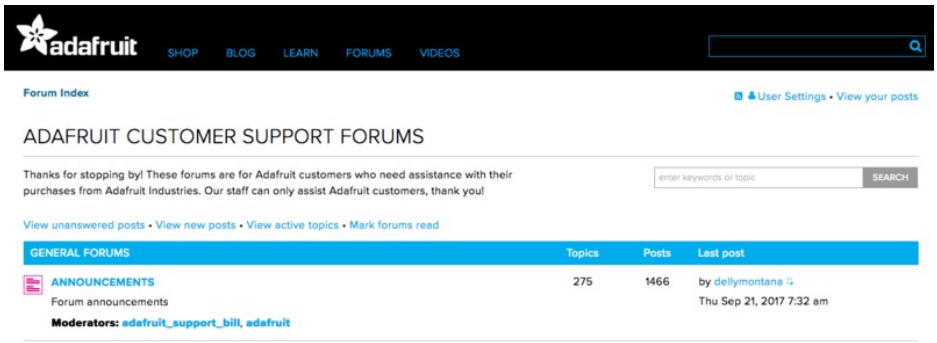
The CircuitPython channel is where to go with your CircuitPython questions. #circuitpython is there for new users and developers alike so feel free to ask a question or post a comment! Everyone of any experience level is welcome to join in on the conversation. We'd love to hear what you have to say!

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

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

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

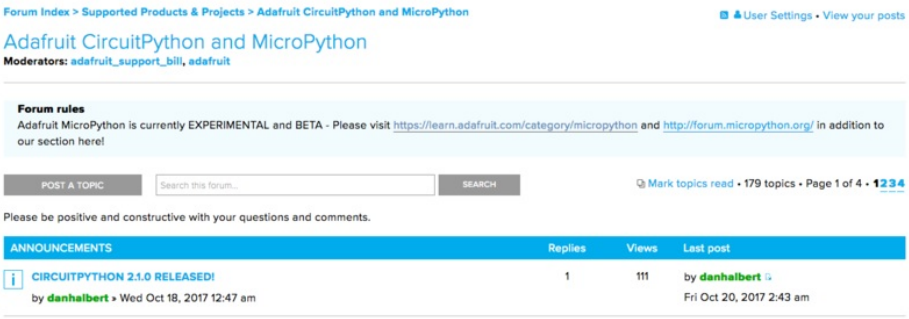
Adafruit Forums



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

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

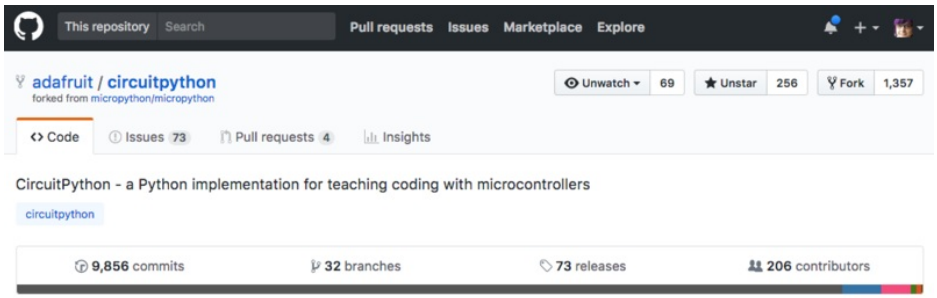
There are forum categories that cover all kinds of topics, including everything Adafruit. The Adafruit CircuitPython and MicroPython (https://adafru.it/xXA) category under "Supported Products & Projects" is the best place to post your CircuitPython questions.



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

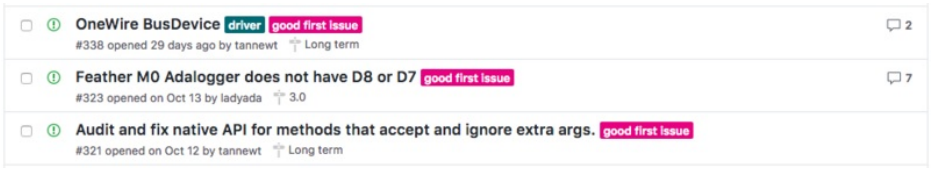
You might think you're just getting started, but you definitely know something that someone else doesn't. The great thing about the forums is that you can help others too! Everyone is welcome and encouraged to provide constructive feedback to any of the posted questions. This is an excellent way to contribute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

Adafruit Github



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

If you're new to GitHub or programming in general, there are great opportunities for you. Head over to adafruit/circuitpython (https://adafru.it/tB7) on GitHub, click on "Issues (https://adafru.it/Bee)", and you'll find a list that includes issues labeled "good first issue (https://adafru.it/Bef)". These are things we've identified as something that someone with any level of experience can help with. These issues include options like updating documentation, providing feedback, and fixing simple bugs.



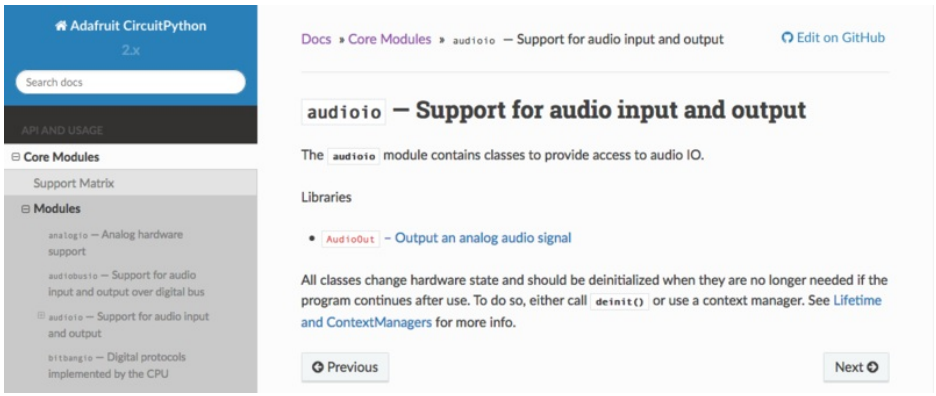
Already experienced and looking for a challenge? Checkout the rest of the issues list and you'll find plenty of ways to contribute. You'll find everything from new driver requests to core module updates. There's plenty of opportunities for everyone at any level!

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

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

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

ReadTheDocs



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

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


Here is blinky:

```
import digitalio
from board import *
import time

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

UF2 Bootloader Details

This is an information page for advanced users who are curious how we get code from your computer into your Express board!

Adafruit SAMD21 (M0) and SAMD51 (M4) boards feature an improved bootloader that makes it easier than ever to flash different code onto the microcontroller. This bootloader makes it easy to switch between Microsoft MakeCode, CircuitPython and Arduino.

Instead of needing drivers or a separate program for flashing (say, `bossac`, `jlink` or `avrdude`), one can simply ***drag a file onto a removable drive***.

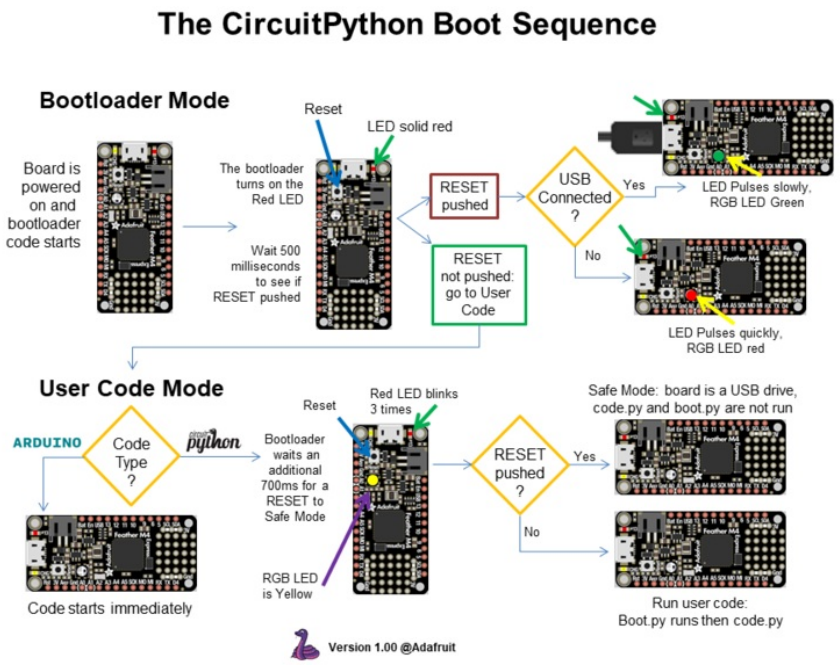
The format of the file is a little special. Due to 'operating system woes' you cannot just drag a binary or hex file (trust us, we tried it, it isn't cross-platform compatible). Instead, the format of the file has extra information to help the bootloader know where the data goes. The format is called UF2 (USB Flashing Format). Microsoft MakeCode generates UF2s for flashing and CircuitPython releases are also available as UF2. You can also create your own UF2s from binary files using uf2tool, available here. (https://adafru.it/vPE)

The bootloader is ***also BOSSA compatible***, so it can be used with the Arduino IDE which expects a BOSSA bootloader on ATSAMD-based boards

For more information about UF2, you can read a bunch more at the MakeCode blog (https://adafru.it/w5A), then check out the UF2 file format specification. (https://adafru.it/vPE)

Visit Adafruit's fork of the Microsoft UF2-samd bootloader GitHub repository (https://adafru.it/Beu) for source code and releases of pre-built bootloaders on CircuitPython.org (https://adafru.it/Em8).

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

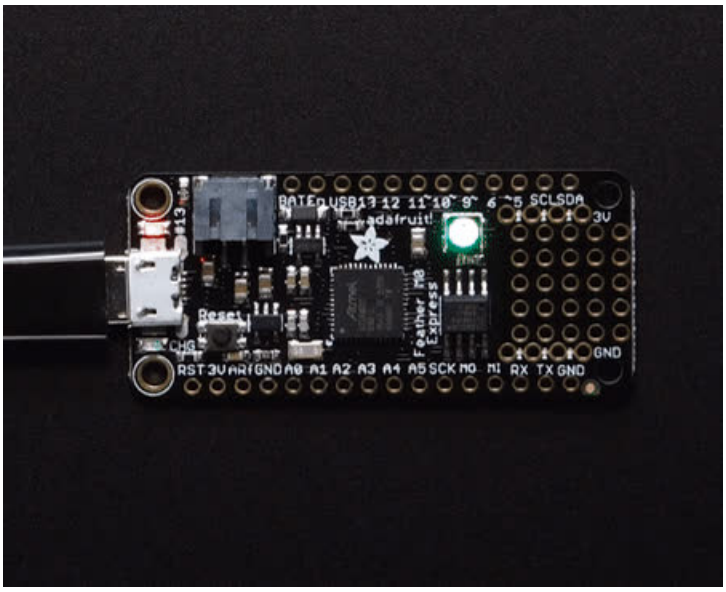


Entering Bootloader Mode

The first step to loading new code onto your board is triggering the bootloader. It is easily done by double tapping the reset button. Once the bootloader is active you will see the small red LED fade in and out and a new drive will appear on your computer with a name ending in **BOOT**. For example, feathers show up as **FEATHERBOOT**, while the new CircuitPlayground shows up as **CPLAYBOOT**, Trinket M0 will show up as **TRINKETBOOT**, and Gemma M0 will show up as **GEMMABOOT**

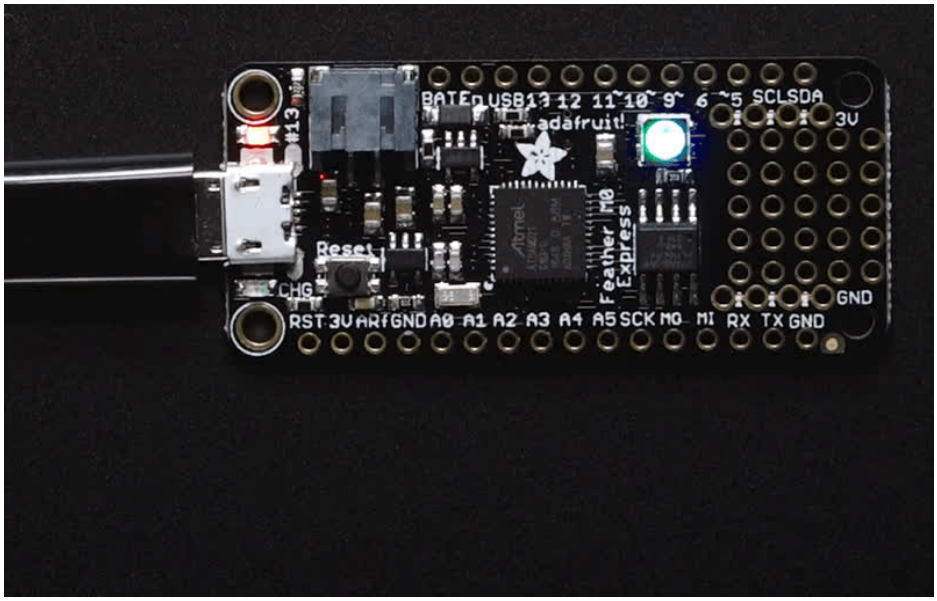
Furthermore, when the bootloader is active, it will change the color of one or more onboard neopixels to indicate the connection status, red for disconnected and green for connected. If the board is plugged in but still showing that its disconnected, try a different USB cable. Some cables only provide power with no communication.

For example, here is a Feather M0 Express running a colorful Neopixel swirl. When the reset button is double clicked (about half second between each click) the NeoPixel will stay green to let you know the bootloader is active. When the reset button is clicked once, the 'user program' (NeoPixel color swirl) restarts.

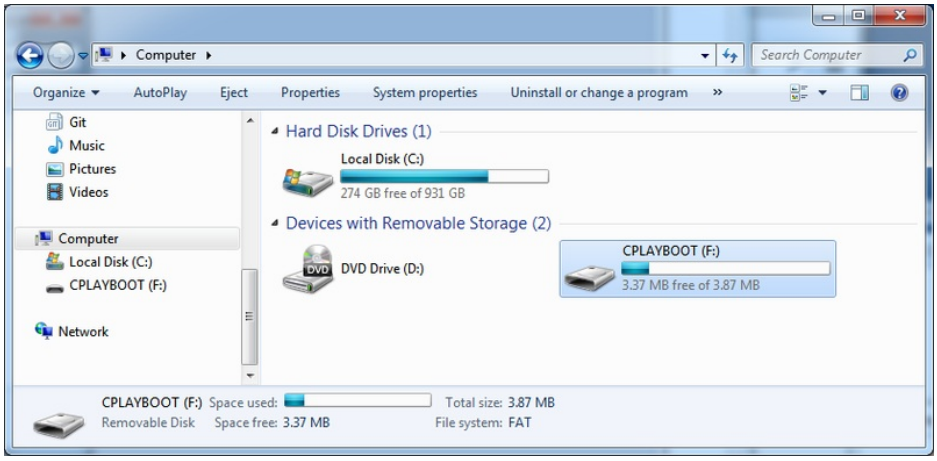


If the bootloader couldn't start, you will get a red NeoPixel LED.

That could mean that your USB cable is no good, it isn't connected to a computer, or maybe the drivers could not enumerate. Try a new USB cable first. Then try another port on your computer!

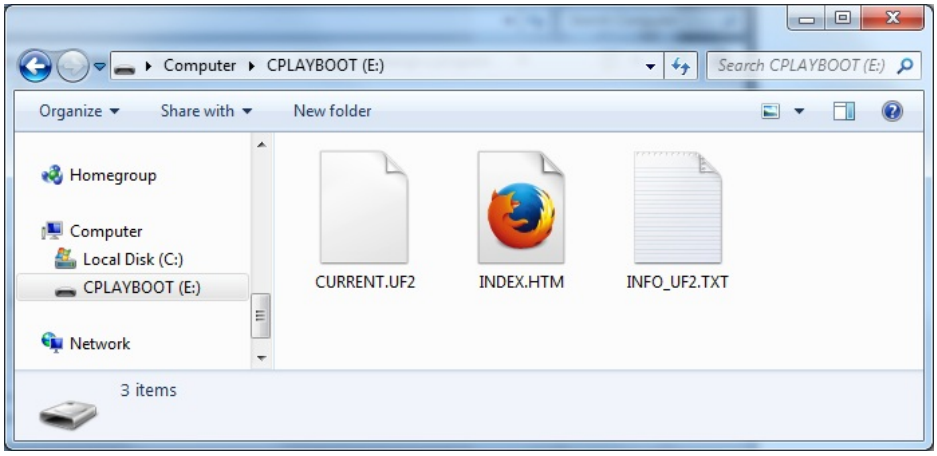


Once the bootloader is running, check your computer. You should see a USB Disk drive...



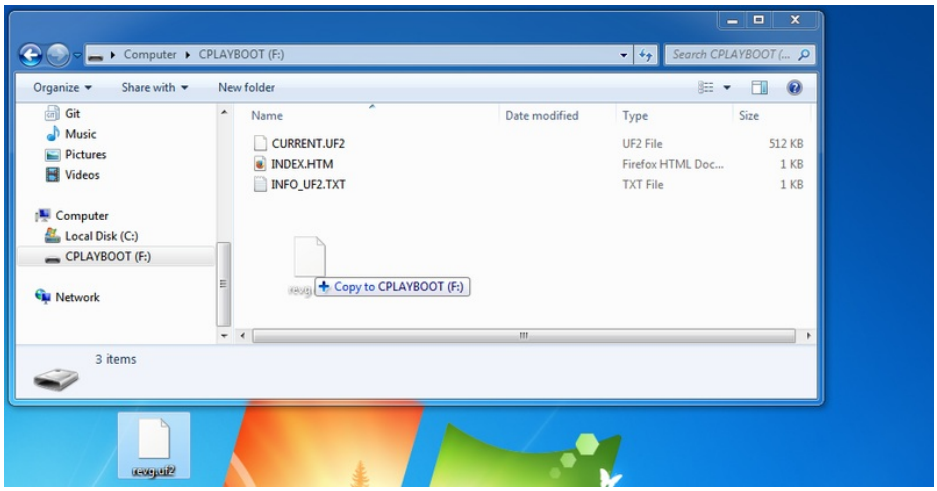
Once the bootloader is successfully connected you can open the drive and browse the virtual filesystem. This isn't the same filesystem as you use with CircuitPython or Arduino. It should have three files:

- **CURRENT.UF2** - The current contents of the microcontroller flash.
- **INDEX.HTM** - Links to Microsoft MakeCode.
- **INFO_UF2.TXT** - Includes bootloader version info. Please include it on bug reports.

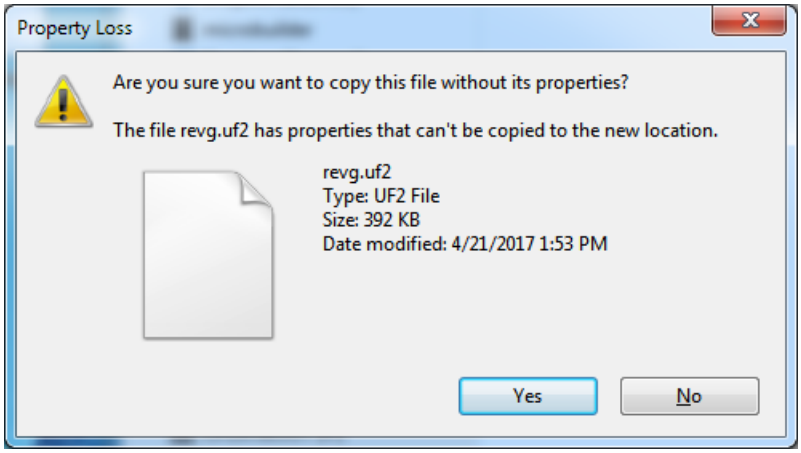


Using the Mass Storage Bootloader

To flash something new, simply drag any UF2 onto the drive. After the file is finished copying, the bootloader will automatically restart. This usually causes a warning about an unsafe eject of the drive. However, its not a problem. The bootloader knows when everything is copied successfully.



You may get an alert from the OS that the file is being copied without it's properties. You can just click **Yes**



You may also get get a complaint that the drive was ejected without warning. Don't worry about this. The drive only ejects once the bootloader has verified and completed the process of writing the new code

Using the BOSSA Bootloader

As mentioned before, the bootloader is also compatible with BOSSA, which is the standard method of updating boards when in the Arduino IDE. It is a command-line tool that can be used in any operating system. We won't cover the full use of the **bossac** tool, suffice to say it can do quite a bit! More information is available at ShumaTech (https://adafru.it/vQa).

Windows 7 Drivers

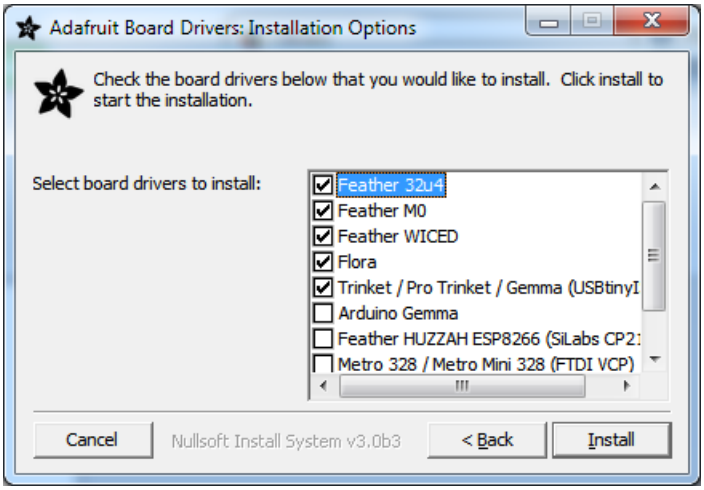
If you are running Windows 7 (or, goodness, something earlier?) You will need a Serial Port driver file. Windows 10 users do not need this so skip this step.

You can download our full driver package here:

https://adafru.it/AB0

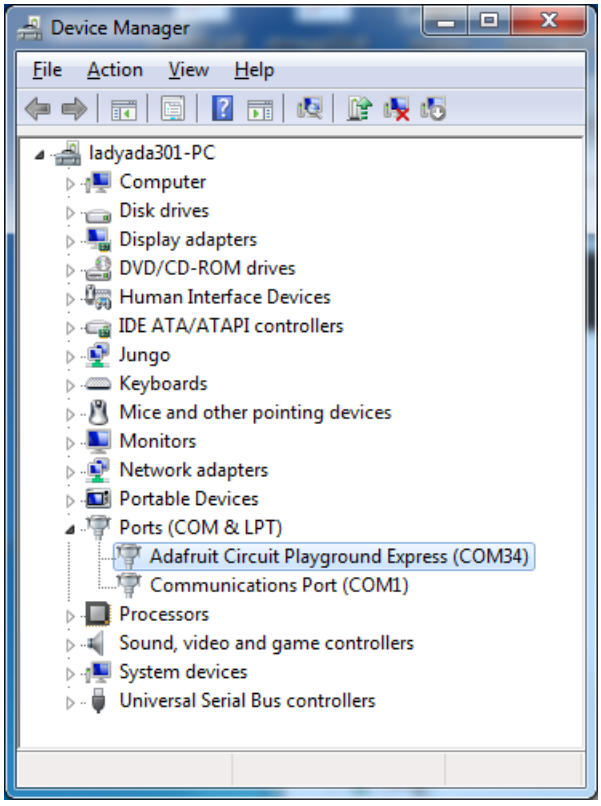
https://adafru.it/AB0

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

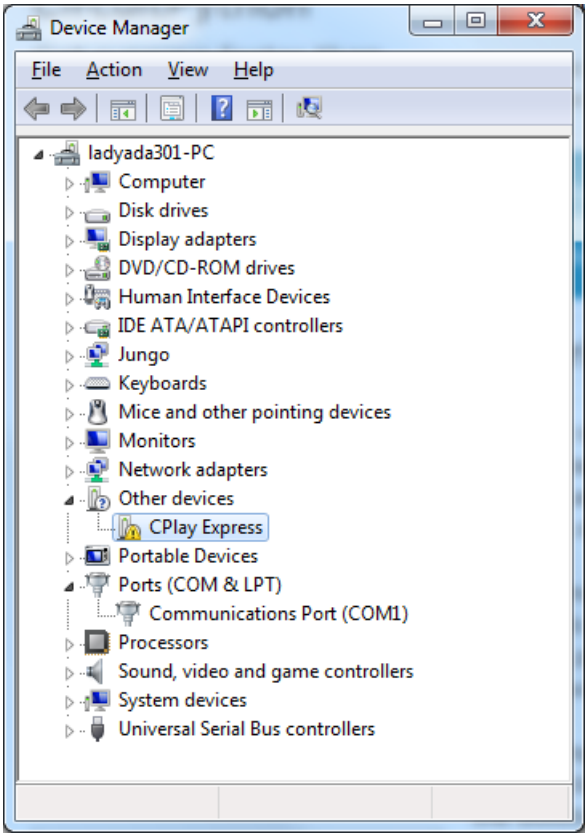


Verifying Serial Port in Device Manager

If you're running Windows, its a good idea to verify the device showed up. Open your Device Manager from the control panel and look under **Ports (COM & LPT)** for a device called **Feather M0** or **Circuit Playground** or whatever!



If you see something like this, it means you did not install the drivers. Go back and try again, then remove and re-plug the USB cable for your board



Running bossac on the command line

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

You can download the latest builds here. (https://adafru.it/s1B) The `mingw32` version is for Windows, `apple-darwin` for Mac OSX and various `linux` options for Linux. Once downloaded, extract the files from the zip and open the command line to the directory with `bossac`.

With bossac versions 1.9 or later, you must use the --offset parameter on the command line, and it must have the correct value for your board.

With bossac version 1.9 or later, you must give an `--offset` parameter on the command line to specify where to start writing the firmware in flash memory. This parameter was added in bossac 1.8.0 with a default of `0x2000`, but starting in 1.9, the default offset was changed to `0x0000`, which is not what you want in most cases. If you omit the argument for bossac 1.9 or later, you will probably see a "**Verify Failed**" error from bossac. Remember to change the option for `-p` or `--port` to match the port on your Mac.

Replace the filename below with the name of your downloaded `.bin` : it will vary based on your board!

Using bossac Versions 1.7.0, 1.8

There is no `--offset` parameter available. Use a command line like this:

`bossac -p=/dev/cu.usbmodem14301 -e -w -v -R adafruit-circuitpython-`*`boardname`*`-`*`version`*`.bin`

For example,

```
bossac -p=/dev/cu.usbmodem14301 -e -w -v -R adafruit-circuitpython-feather_m0_express-3.0.0.bin
```

Using bossac Versions 1.9 or Later

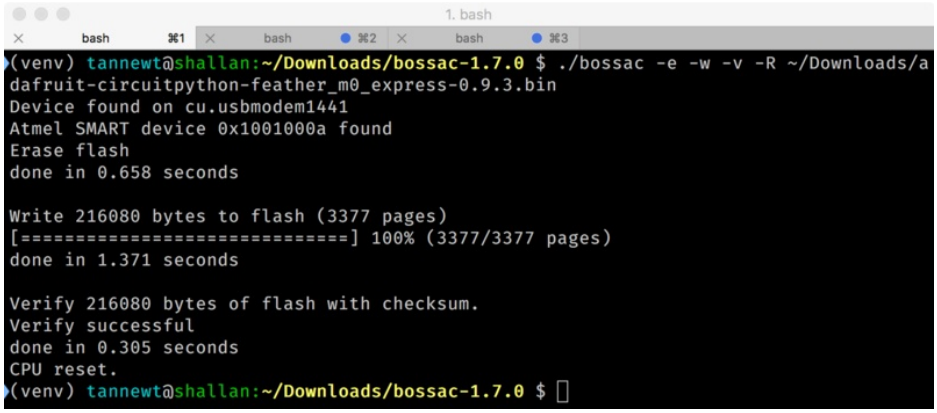
For M0 boards, which have an 8kB bootloader, you must specify `-offset=0x2000`, for example:

```
bossac -p=/dev/cu.usbmodem14301 -e -w -v -R --offset=0x2000 adafruit-circuitpython-feather_m0_express-3.0.0.bin
```

For M4 boards, which have a 16kB bootloader, you must specify `-offset=0x4000`, for example:

```
bossac -p=/dev/cu.usbmodem14301 -e -w -v -R --offset=0x4000 adafruit-circuitpython-feather_m4_express-3.0.0.bin
```

This will `e` rase the chip, `w` rite the given file, `v` erify the write and `R` eset the board. On Linux or MacOS you may need to run this command with `sudo ./bossac ...`, or add yourself to the **dialout** group first.



Updating the bootloader

The UF2 bootloader is relatively new and while we've done a ton of testing, it may contain bugs. Usually these bugs effect reliability rather than fully preventing the bootloader from working. If the bootloader is flaky then you can try updating the bootloader itself to potentially improve reliability.

If you're using MakeCode on a Mac, you need to make sure to upload the bootloader to avoid a serious problem with newer versions of MacOS. See instructions and more details here (https://adafru.it/ECU).

In general, you shouldn't have to update the bootloader! If you do think you're having bootloader related issues, please post in the forums or discord.

Updating the bootloader is as easy as flashing CircuitPython, Arduino or MakeCode. Simply enter the bootloader as above and then drag the *update bootloader uf2* file below. This uf2 contains a program which will unlock the bootloader section, update the bootloader, and re-lock it. It will overwrite your existing code such as CircuitPython or Arduino so make sure everything is backed up!

After the file is copied over, the bootloader will be updated and appear again. The **INFO_UF2.TXT** file should show the newer version number inside.

For example:

```
UF2 Bootloader v2.0.0-adafruit.5 SFHWRO
Model: Metro M0
Board-ID: SAMD21G18A-Metro-v0
```

Lastly, reload your code from Arduino or MakeCode or flash the latest CircuitPython core (https://adafru.it/Em8).

Below are the latest updaters for various boards. The latest versions can always be found here (https://adafru.it/Bmg). Look for the `update-bootloader...` files, not the `bootloader...` files.

https://adafru.it/ECV

https://adafru.it/ECV

https://adafru.it/ECW

https://adafru.it/ECW

https://adafru.it/ECY

https://adafru.it/ECY

https://adafru.it/ED0

https://adafru.it/ED0

https://adafru.it/ED3

https://adafru.it/ED3

https://adafru.it/ED6

https://adafru.it/ED6

https://adafru.it/ED8

https://adafru.it/ED8

https://adafru.it/Bmg

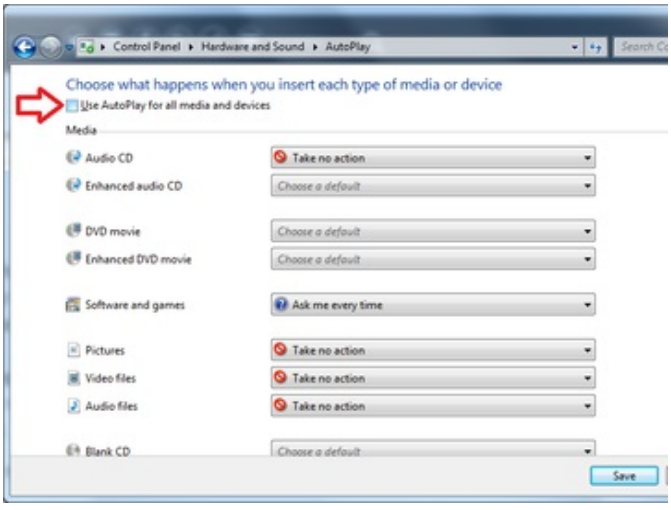
https://adafru.it/Bmg

Getting Rid of Windows Pop-ups

If you do a *lot* of development on Windows with the UF2 bootloader, you may get annoyed by the constant "Hey you inserted a drive what do you want to do" pop-ups.

Go to the Control Panel. Click on the **Hardware and Sound** header

Click on the **Autoplay** header



Uncheck the box at the top, labeled **Use Autoplay for all devices**

Making your own UF2

Making your own UF2 is easy! All you need is a .bin file of a program you wish to flash andthe Python conversion script (https://adafru.it/vZb). Make sure that your program was compiled to start at 0x2000 (8k) for M0 boards or 0x4000 (16kB) for M4 boards. The bootloader takes up the first 8kB (M0) or 16kB (M4). CircuitPython's linker script (https://adafru.it/CXh) is an example on how to do that.

Once you have a .bin file, you simply need to run the Python conversion script over it. Here is an example from the directory with **uf2conv.py**. This command will produce a **firmware.uf2** file in the same directory as the source **firmware.bin**. The uf2 can then be flashed in the same way as above.

```
# For programs with 0x2000 offset (default)
uf2conv.py -c -o build-circuitplayground_express/firmware.uf2 build-
circuitplayground_express/firmware.bin

# For programs needing 0x4000 offset (M4 boards)
uf2conv.py -c -b 0x4000 -o build-metro_m4_express/firmware.uf2 build-metro_M4_express/firmware.bin
```

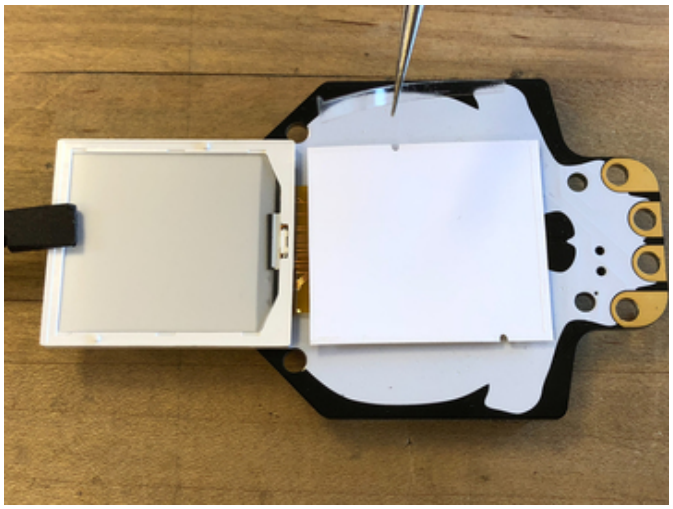
Installing the bootloader on a fresh/bricked board

If you somehow damaged your bootloader or maybe you have a new board, you can use a JLink to re-install it.Here's a short writeup by turbinenreiter on how to do it for the Feather M4 (but adaptable to other boards) (https://adafru.it/ven)

Troubleshooting

TFT Screen Adhesive

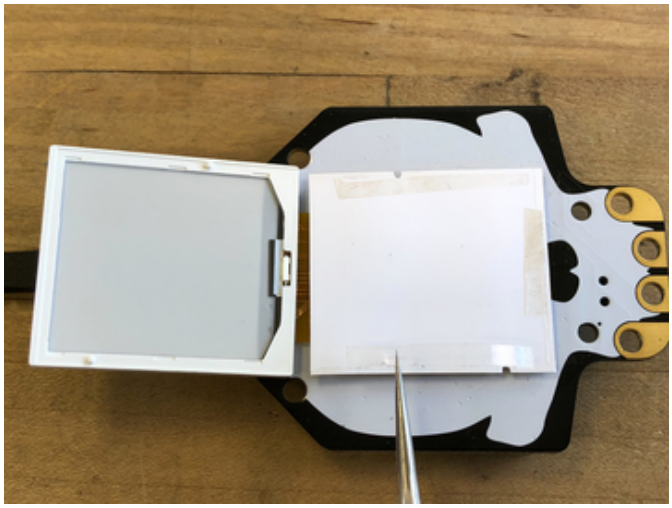
The TFT screen on the HalloWing can become un-adhered if it is bumped or wiggled too much (or left in a very hot Jeep near the windshield in the hot southern California sun, for a totally hypothetical example that didn't necessarily happen to the author). Fixing this is pretty easy -- you can use double-stick tape, E6000 glue, or Sugru to fix it back in place. Here are some action photos of these fixes.

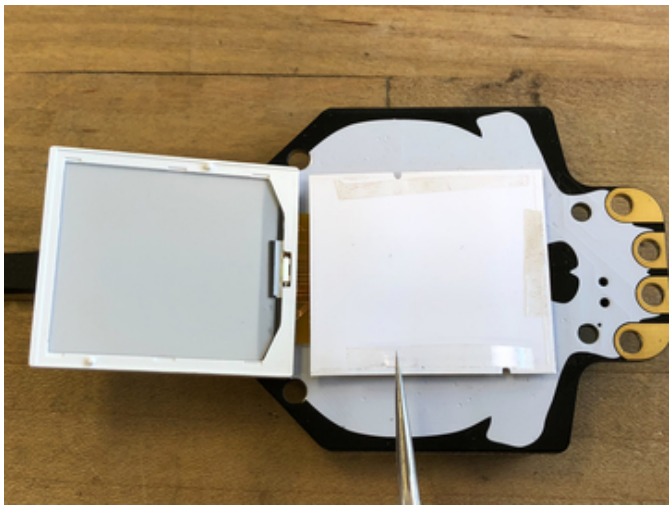


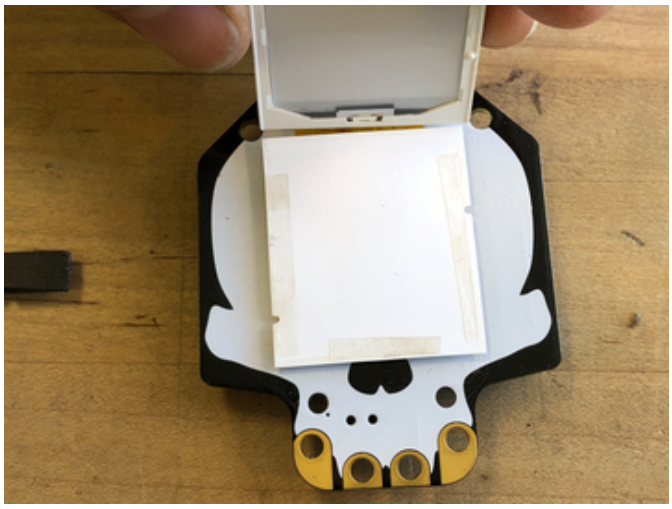
Double Stick Tape

Cut three thin slices of double stick tape, then place them around the edges of the backlight.

Press the screen down and hold for a few seconds to adhere.





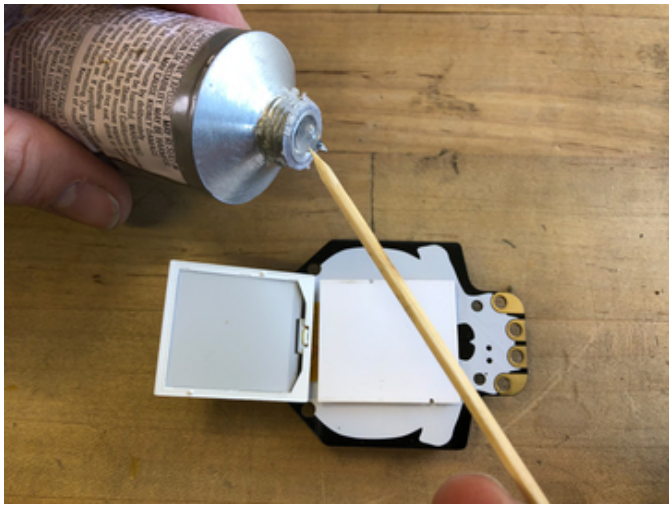


E6000 Glue

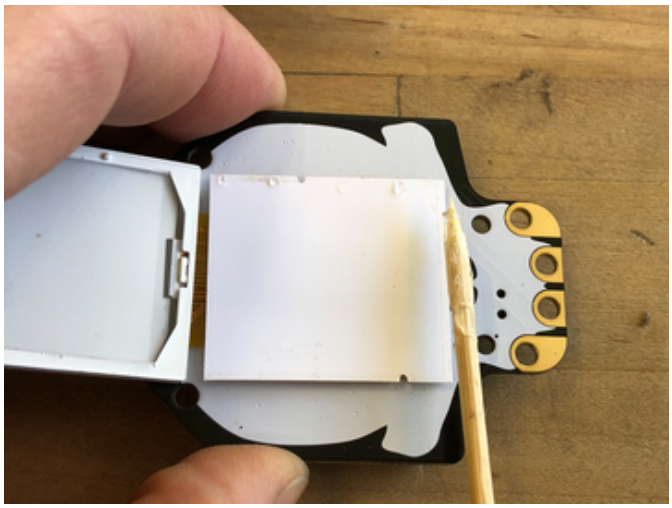
This is strong glue and is even better than tape if you plan to leave the HalloWing screen exposed on a costume without the lens and cover.

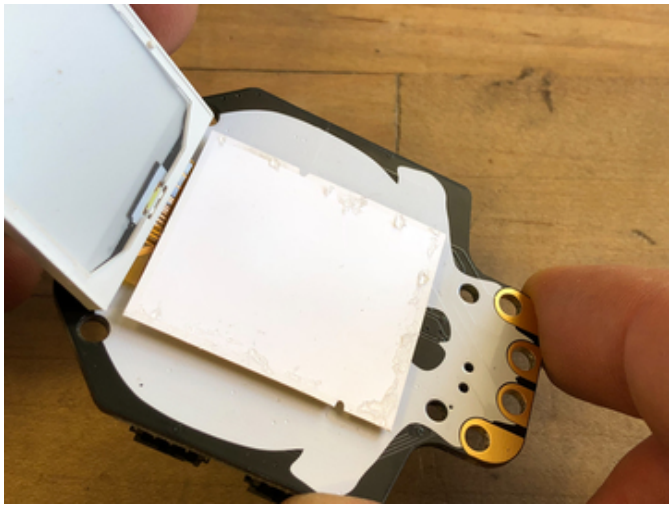
Use a toothpick or skewer to place small dabs of the

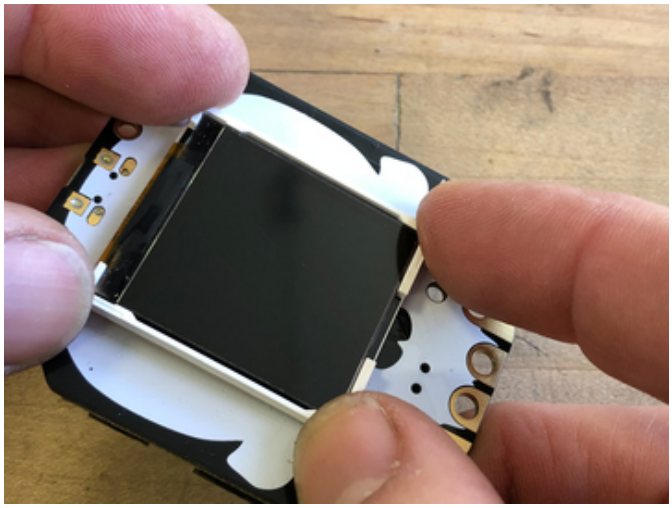
glue around the perimeter of the backlight.

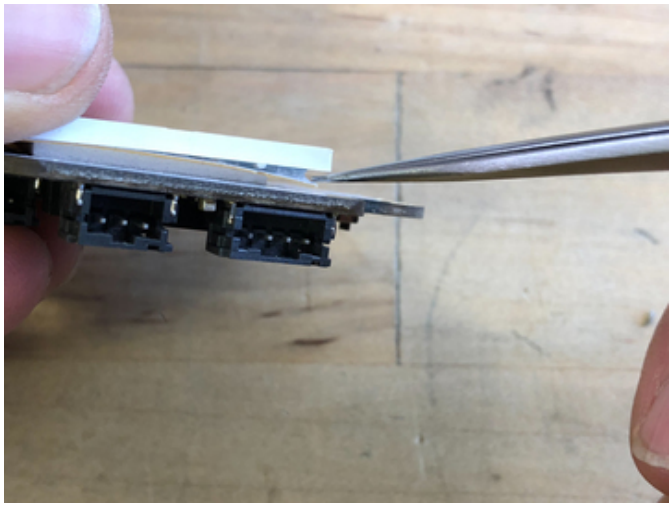


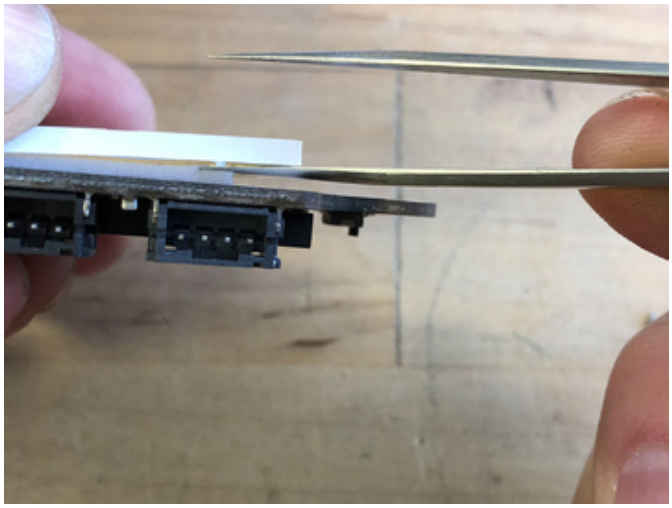
Press the screen down and hold for a few seconds, then allow to cure overnight.

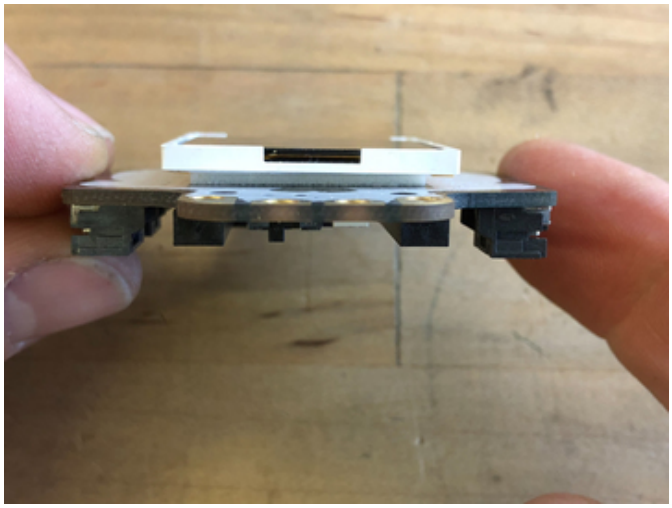






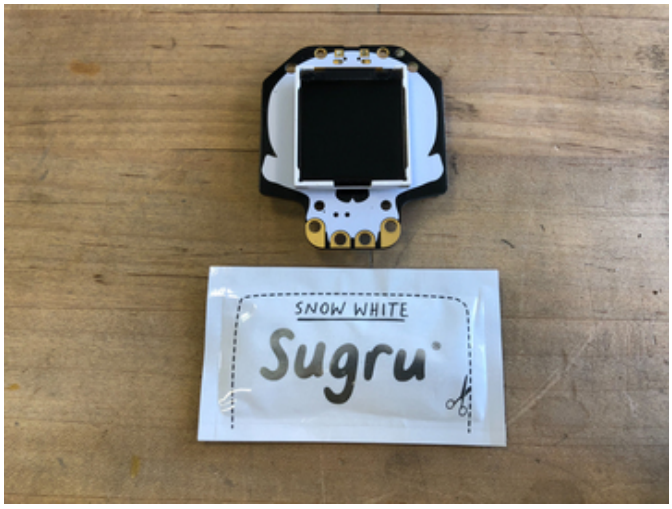






Sugru

You can go all out and create a protective frame using Sugru (https://adafru.it/ekR) moldable silicone rubber.

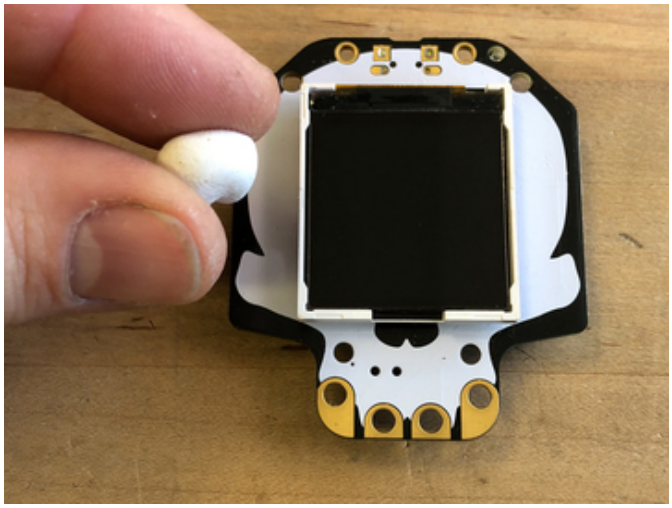


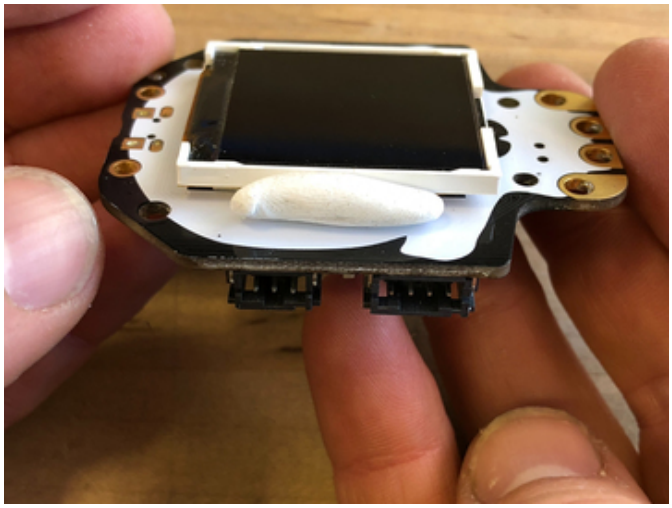
Open a sachet of Sugru and tear off a small ball of it.

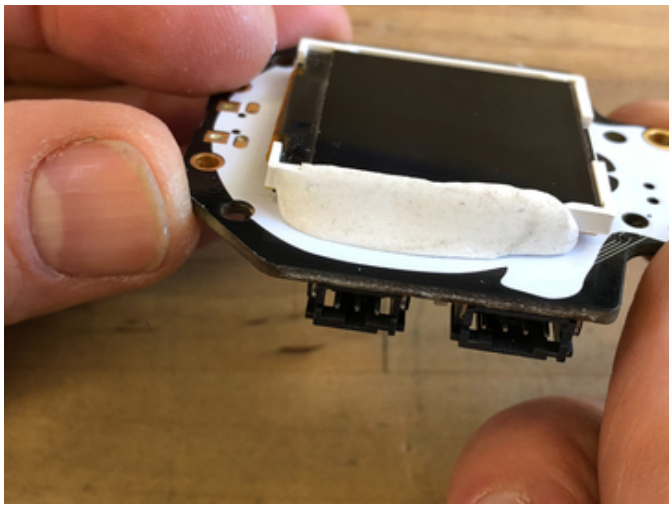
Roll the ball into a small cylinder and press up against one side of teh board and screen.

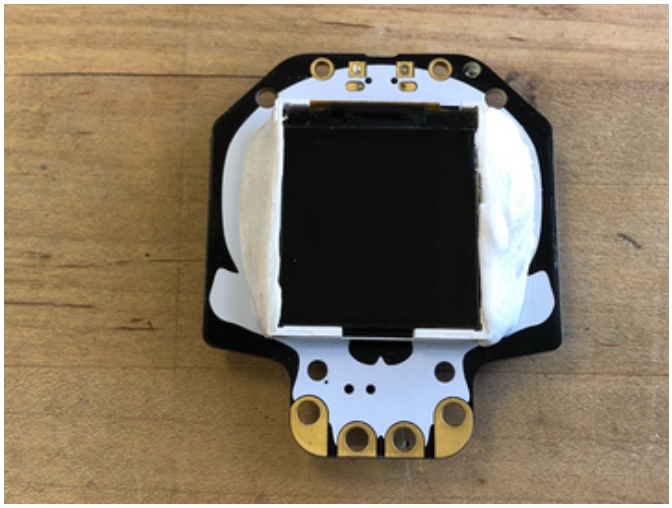
Repeat for the other side.

Allow Sugru to cure overnight.









Diagnostics

Want to see some stats on your HalloWing M4? Double-click the board's reset button to get to bootloader mode, then drag the HallowWingM4_Diagnostics.UF2 onto the HALLOWBOOT drive!

https://adafru.it/G6F

https://adafru.it/G6F

Downloads

FIles:

- Datasheet for ATSAMD51G18 (https://adafru.it/FQH)
- Datasheet for MSA301 accelerometer (https://adafru.it/FDu)
- Datasheet for ST7789 (https://adafru.it/FQI)
- EagleCAD files on GitHub (https://adafru.it/FQJ)
- 3D models on GitHub (https://adafru.it/FRS)
- Fritzing object in the Adafruit Fritzing Library (https://adafru.it/FQK)

Here us the UF2 for the factory eye animation - press the reset button one (or twice) to get the board recognized by your computer as a drive named **HALLOM4BOOT**. Copy the UF2 file below to the **HALLOM4BOOT** drive and the board should reset and be running the eye animation.

https://adafru.it/FSa

https://adafru.it/FSa

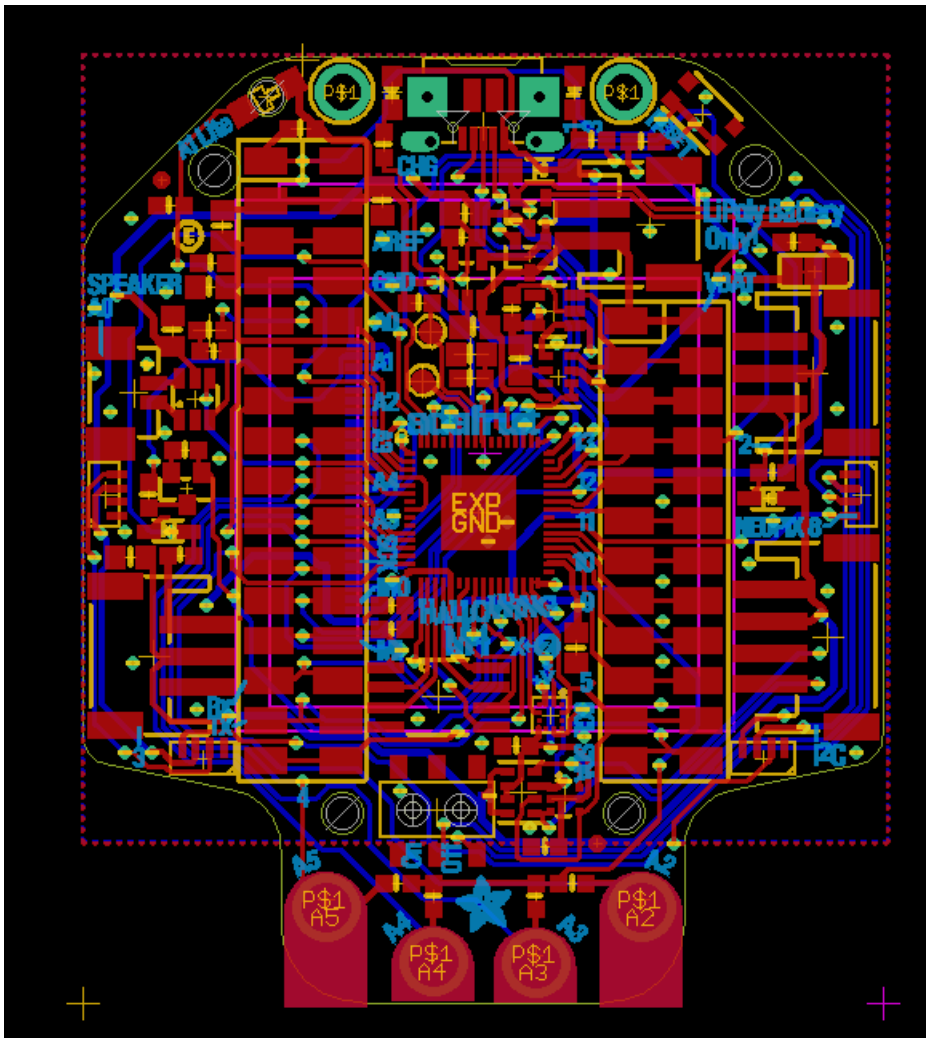
Animated eyes with NeoPixels

This UF2 features the eye animation with rainbow neopixel animation. It's similar to the demo featured in the product page hero image.

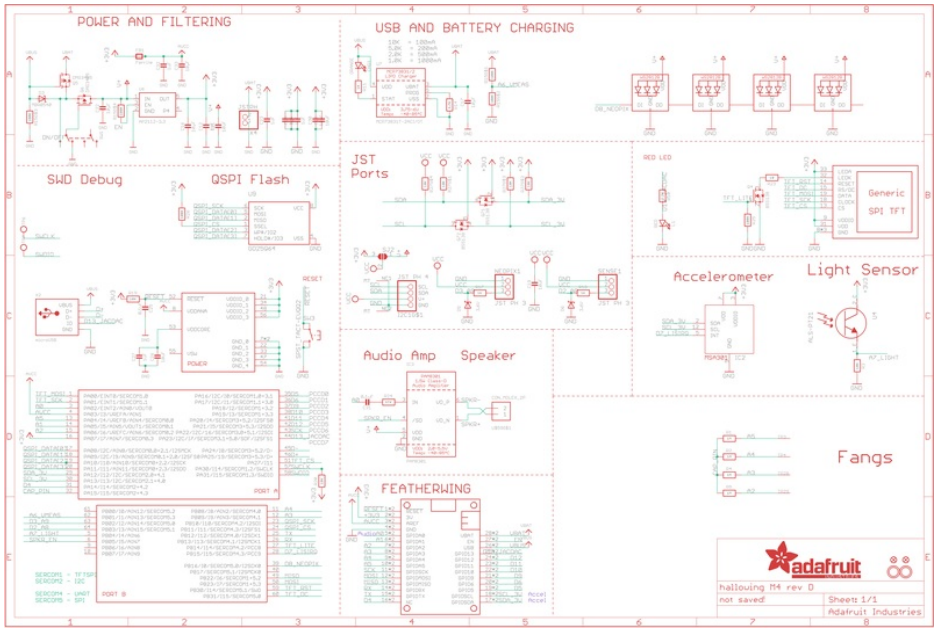
https://adafru.it/FXn

https://adafru.it/FXn

Fab Print


SPEAKER
LiPoly Battery Only
EXP GND
P$1 A5
P$1 A4
P$1 A3
P$1 A2

Schematic


POWER AND FILTERING
USB AND BATTERY CHARGING
SWD Debug
QSPI Flash
JST Ports
Generic SPI TFT
Accelerometer
Light Sensor
Audio Amp
Speaker
FEATHERWING
Fangs
adafruit
hallowing M4 rev D
not saved
Sheet 1/1
Adafruit Industries

3D Model