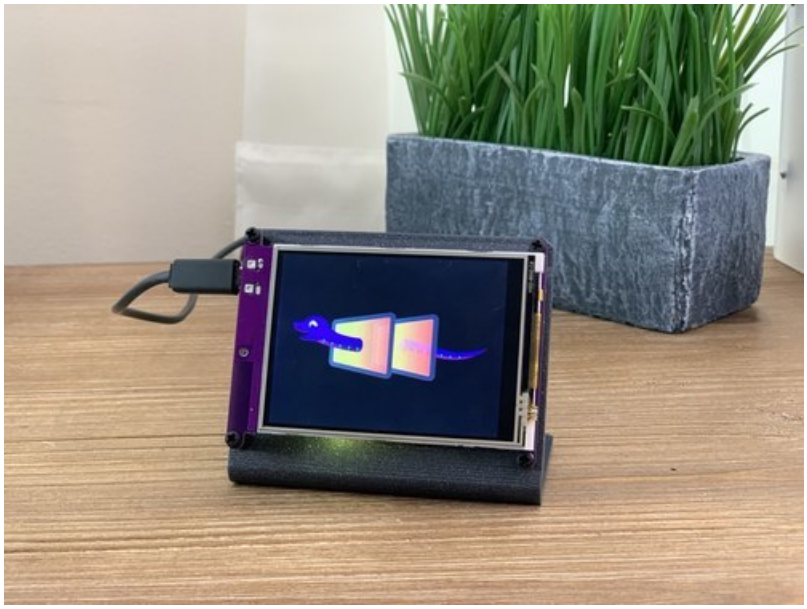
adafruit learning system

Adafruit PyPortal - IoT for CircuitPython

Created by Kattni Rembor

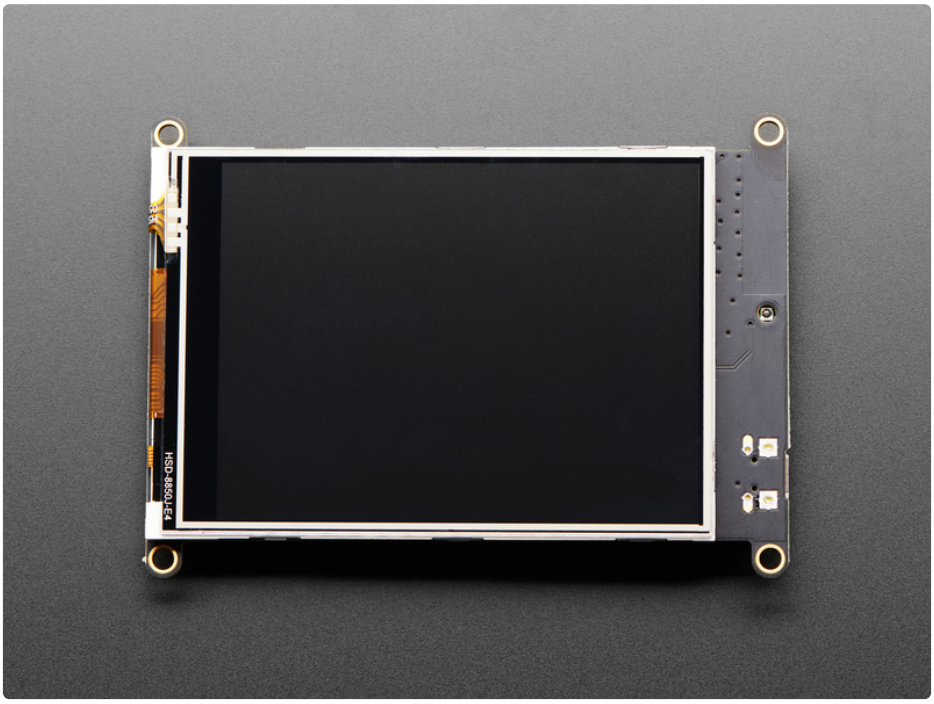


Last updated on 2021-06-07 11:31:45 AM EDT

Guide Contents

Overview

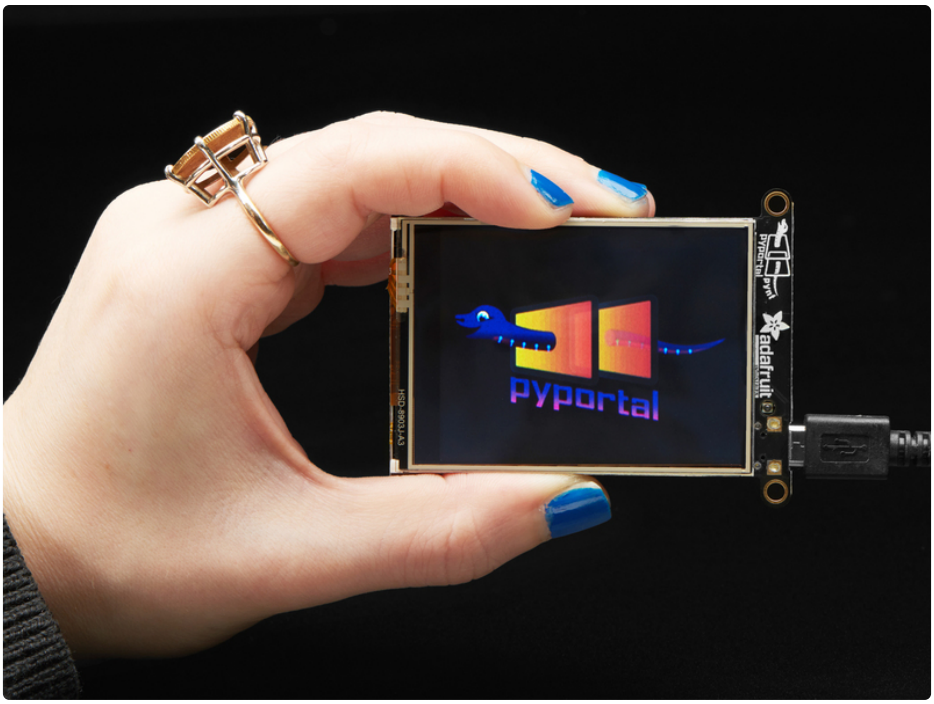
PyPortal, is our easy-to-use IoT device that allows you to create all the things for the “Internet of Things” in minutes. Make custom touch screen interface GUIs, all open-source, and Python-powered using tinyJSON / APIs to get news, stock values, weather, cat photos, and more – all over Wi-Fi with the latest technologies. Create little pocket universes of joy that connect to something good. Rotate it 90 degrees, it’s a web-connected conference badge #badgelife.



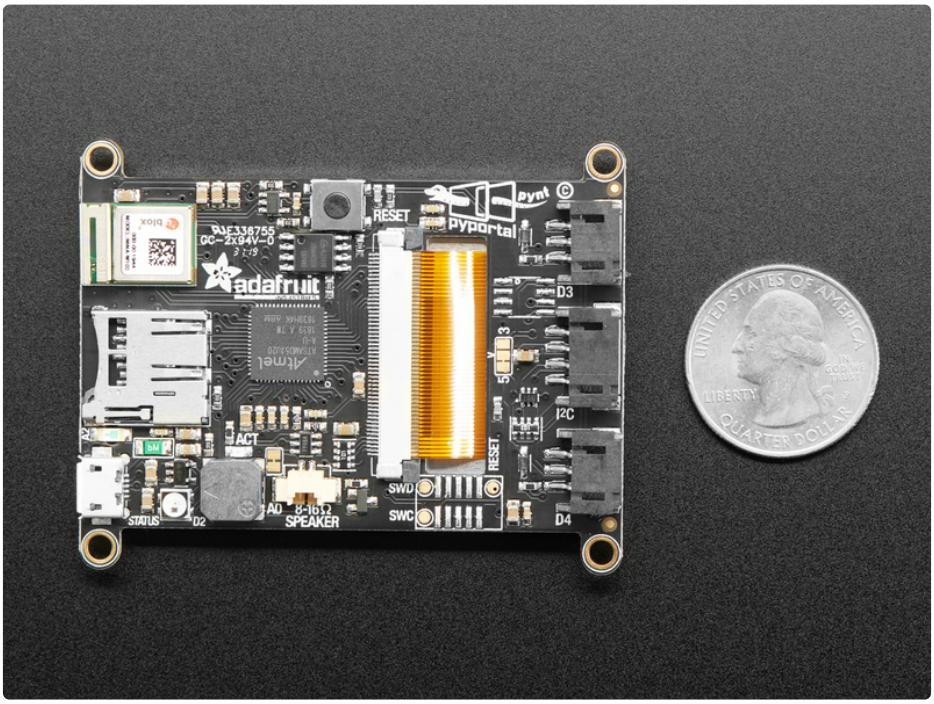
The PyPortal uses an ATMEL (Microchip) ATSAMD51J20, and an Espressif ESP32 Wi-Fi coprocessor with TLS/SSL (secure web) support built-in. PyPortal has a **3.2″ 320 x 240 color TFT** with resistive touch screen.

PyPortal includes: speaker, light sensor, temperature sensor, NeoPixel, microSD card slot, 8MB flash, plug-in ports for I2C, and 2 analog/digital pins. There are 3D files available for custom enclosures and the board has mounting holes which are also compatible with badge lanyard fasteners.

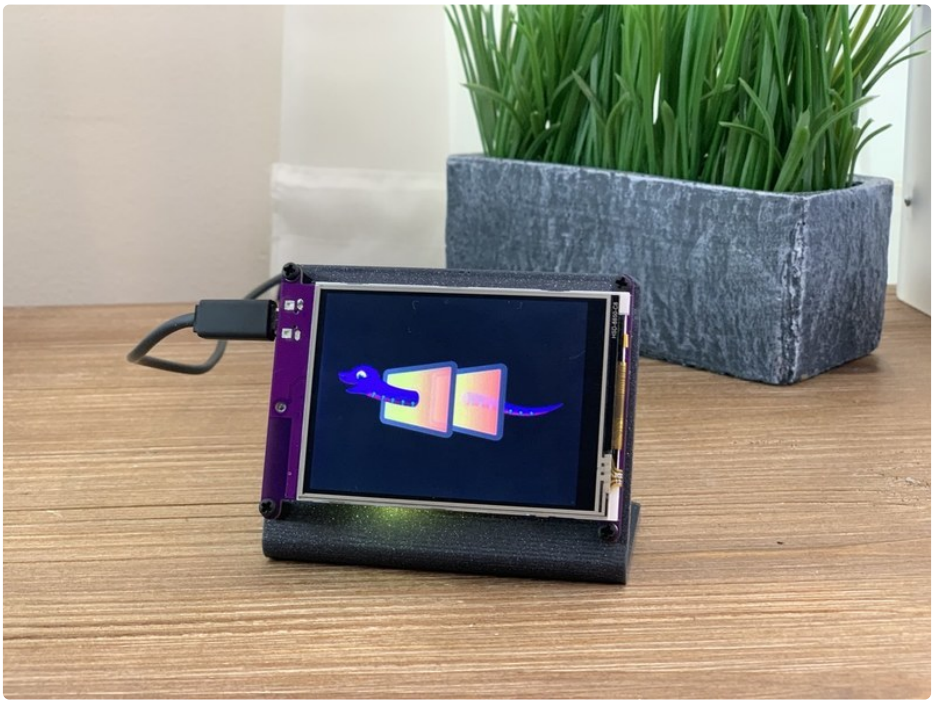
The **PyPortal Pynt** is the little sister to our popular PyPortal (https://adafru.it/pyportal) - zapped with a shink ray to take the design from a 3.2" diagonal down to 2.4" diagonal screen - but otherwise the same! PyPortal Pynt has a **2.4″ diagonal 320 x 240 color TFT** with resistive touch screen. **Compared to the original PyPortal, the Pynt does not include a ADT7410 temperature sensor. Other than the ADT7410, the Pynt's display, processor, STEMMA conectors and WiFi have the exact same wiring as the original 3.2" PyPortal so all Arduino/CircuitPython code will run exactly the same - just smaller!**



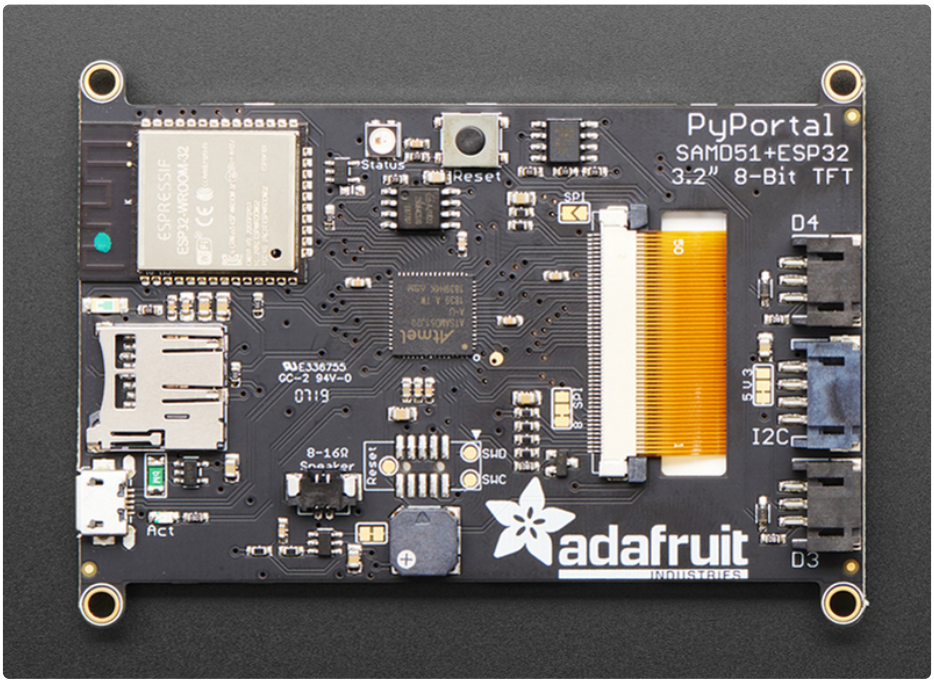
The M4 and ESP32 are a great couple - and each bring their own strengths to this board. The SAMD51 M4 has native USB so it can show up like a disk drive, act as a MIDI or HID keyboard/mouse, and of course bootload and debug over a serial port. It also has DACs, ADC, PWM, and tons of GPIO. Meanwhile, the ESP32 has secure WiFi capabilities, and plenty of Flash and RAM to buffer sockets. By letting the ESP32 focus on the complex TLS/SSL computation and socket buffering, it frees up the SAMD51 to act as the user interface. You get a great programming experience thanks to the native USB with files available for drag-n-drop, and you don't have to spend a ton of processor time and memory to do SSL encryption/decryption and certificate management. It's the best of both worlds!

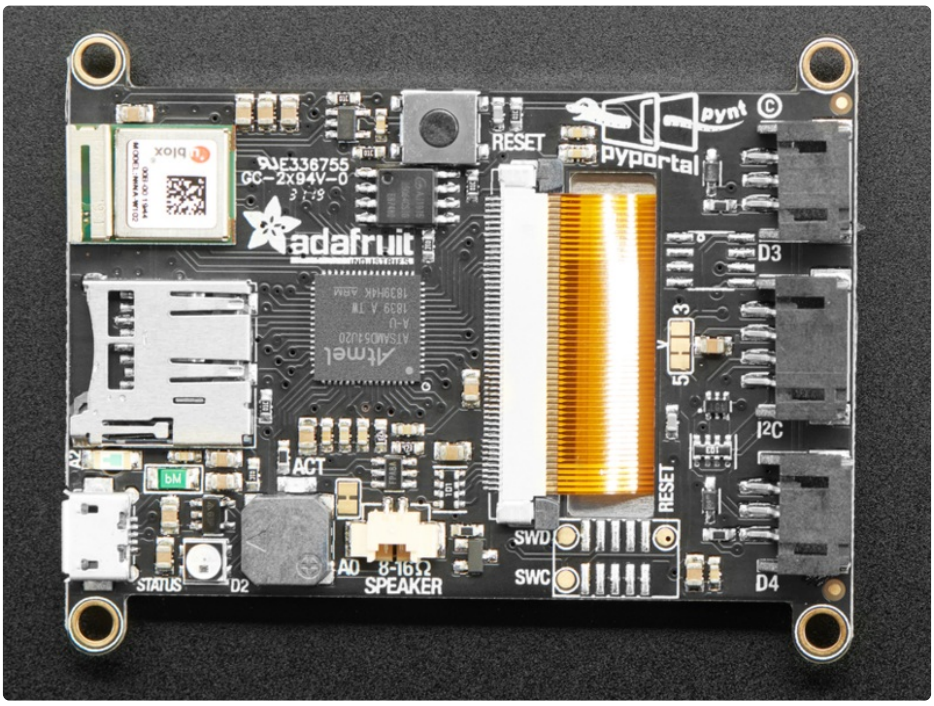


PyPortal is Open-Source hardware and Open-Source software, and it runs CircuitPython and Arduino code. Using CircuitPython, the device shows up as a USB drive and the code can be edited in any IDE, text editor, etc. You can have your IoT project up and running in minutes!



Pinouts



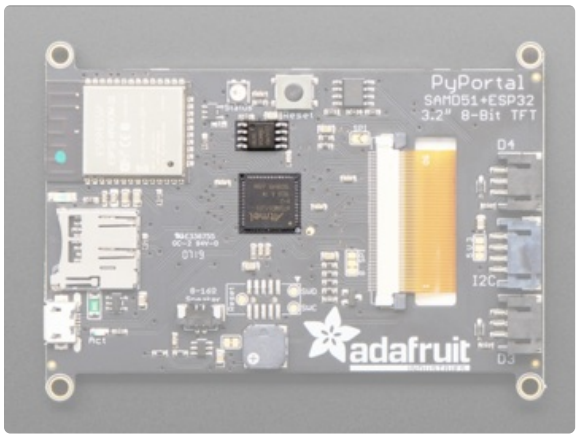


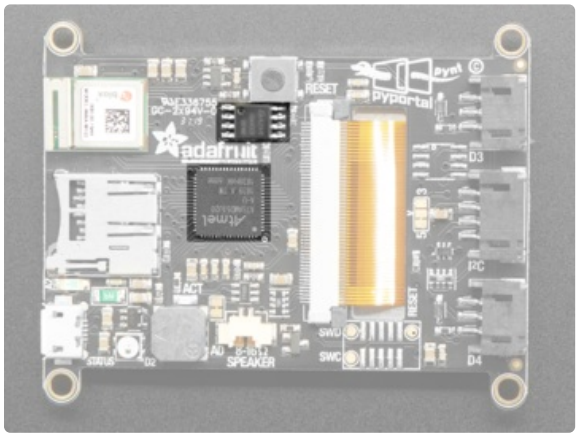
The PyPortal Pynt has the D3 and D4 sockets mislabeled, they should be swapped (to match the pyportal classic, above)

There are so many great features on the Adafruit PyPortal and PyPortal Pynt. Let's take a look at what's

available!

Microcontroller and Flash

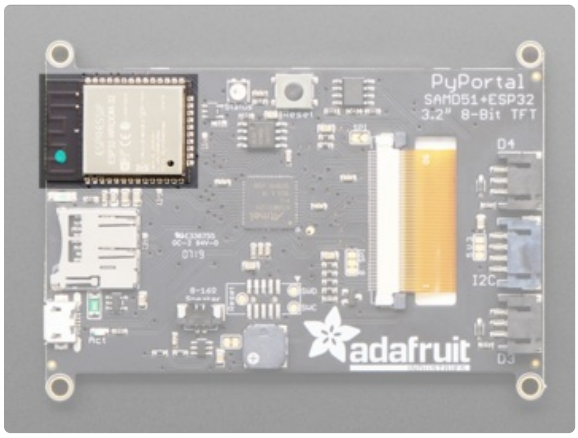


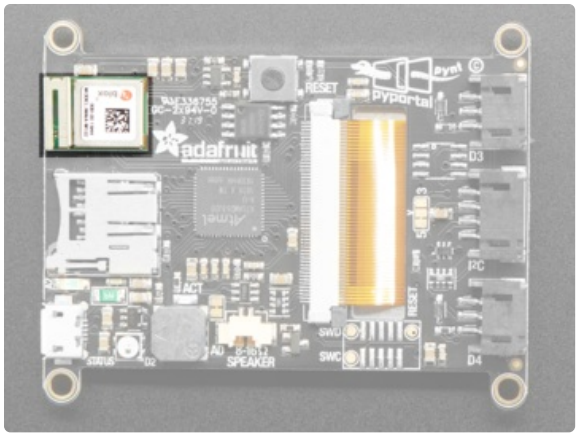


The main processor chip is the **ATSAMD51J20 Cortex M4** running at 120MHz with 3.3v logic/power. It has 1MB of Flash and 256KB of RAM.

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

WiFi





The WiFi capability uses an **Espressif ESP32 Wi-Fi coprocessor** with TLS/SSL support built-in.

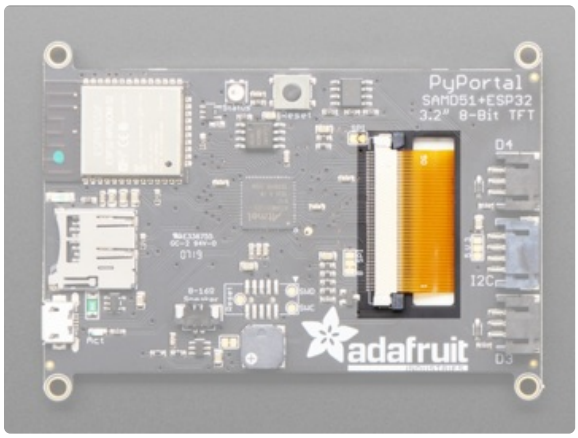
The ESP32 uses the SPI port for data, and also uses a CS pin (`board.ESP_CS` or Arduino `8`), Ready/Busy pin (`board.ESP_BUSY` or Arduino `5`), and reset pin (`board.ESP_RESET` or Arduino `7`)

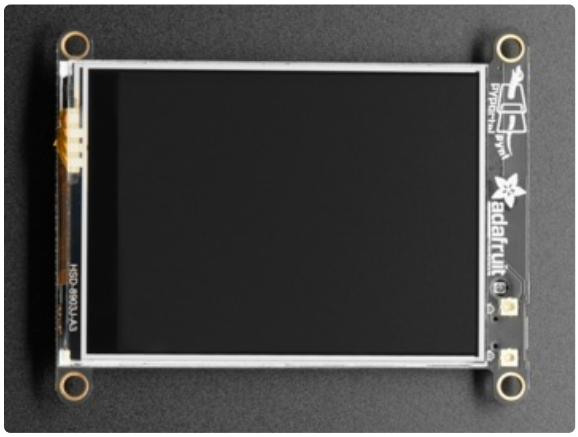
- For advanced use or reprogramming on Arduino, we also connect the main **RX/TX UART** to the ESP32 via `Serial1` . In CircuitPython use `board.ESP_RX` and `board.ESP_TX` . (CircuitPython 6.0.0 and earlier uses `board.RX` and `board.TX`).
- You can also connect to the ESP32 **RTS** pin (used in some serial contexts) on `board.ESP_RTS` or Arduino `51` .
- The ESP32 **GPIO0** pin for bootloader enable is connected to `board.ESP_GPIO0` or Arduino `6`

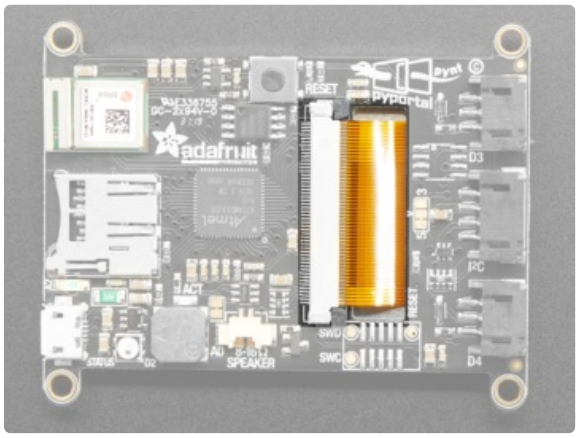
Display and Display Connector

On the front of the PyPortal is a **3.2″ 320 x 240 color TFT** with resistive touch screen! On the front of the Pynt is a **2.4″ diagonal 320 x 240 color TFT** with resistive touch screen!

On the back, there is a large connector near the middle, the **display connector**. It connects the display on the front to the board.







To give you the most data throughput we configure the screen for 8-bit interfacing. That means 8 data lines and a collection of 4 or 5 control lines. If you really want to use the screen in SPI mode, you can do so by soldering closed the **SPI** jumper and cutting/resoldering the **8/SPI** jumper over to the SPI side. That's for advanced users!

The touchscreen is fully analog/resistive. It can be read using our Arduino/CircuitPython drivers. The connections are as follows:

- YD on `board.TOUCH_YD` or Arduino `A4`
- XL on `board.TOUCH_XL` or Arduino `A5`
- YU on `board.TOUCH_YU` or Arduino `A6`
- XR on `board.TOUCH_XR` or Arduino `A7`

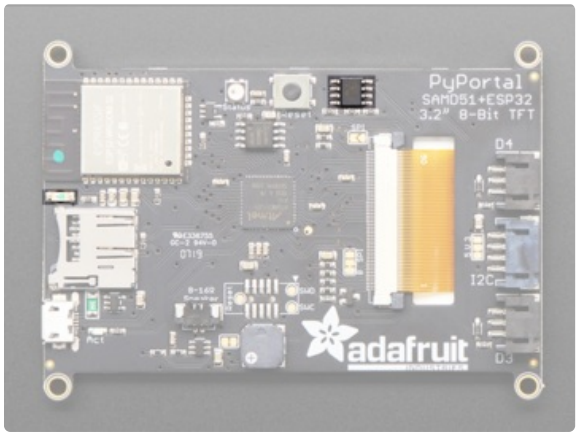
The 8 bit LCD interface is not exposed in CircuitPython (it's managed internally). In Arduino it's on Digital **34** thru **41**, which is on a PORTA 8-bit boundary (PA16-PA23) and can be used for DMA or fast port writes. This probably doesn't affect you.

There are multiple control pins

- **TFT Reset** - `board.TFT_RESET` or Arduino `24`
- **TFT WR** - `board.TFT_WR` or Arduino 25 (this is also the `board.TFT_DC` pin if using in SPI mode)
- **TFT RD** - `board.TFT_RD` or Arduino `9`
- **TFT RS** - `board.TFT_RS` or Arduino `10`
- **TFT CS** - `board.TFT_CS` or Arduino `11`
- **TFT TE** - `board.TFT_TE` or Arduino `12`

There is also a TFT backlight, transistor-connected to `board.TFT_BACKLIGHT` or Arduino `25`. You can PWM control it. There are 6 white LEDs connected in parallel, so having it be full on will draw quite a bit of current (over 100mA!)

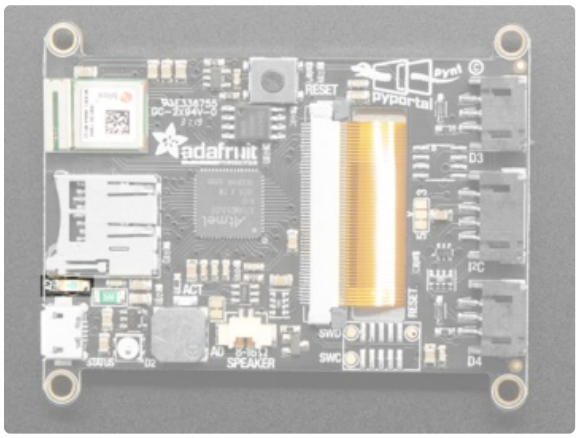
Sensors

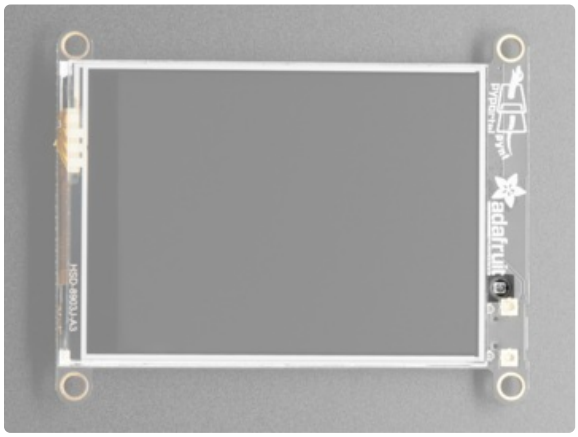


There are two built in sensors.

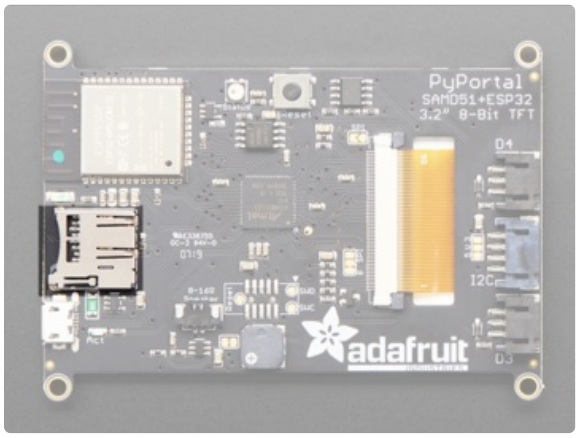
On the top of the PyPortal (not the Pynt) is the **ADT7410 Analog Devices temperature sensor** with 16-bit 0.0078°C temperature resolution and 0.5°C temperature tolerance. The sensor is `I2C` connected, use the Arduino or CircuitPython libraries to read it.

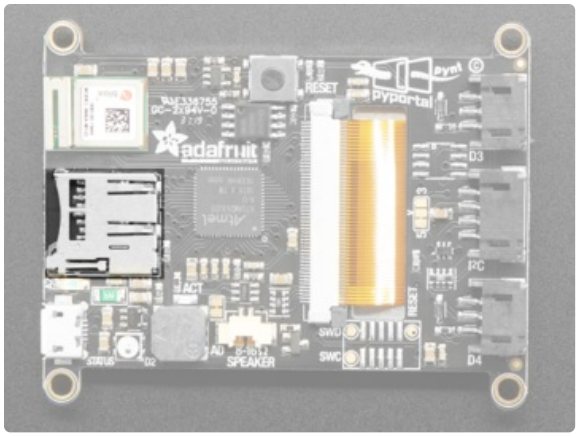
There is also an **ambient light sensor** on the side, which points through to the front, as seen in the second image. The light sensor is an analog input, connected to `board.LIGHT` (CircuitPython) or `A2` (Arduino) you can read it as any analog value ranging from 0 (dark) to 1023 (in Arduino) or 65535 (CircuitPython) when bright.





microSD Card Slot



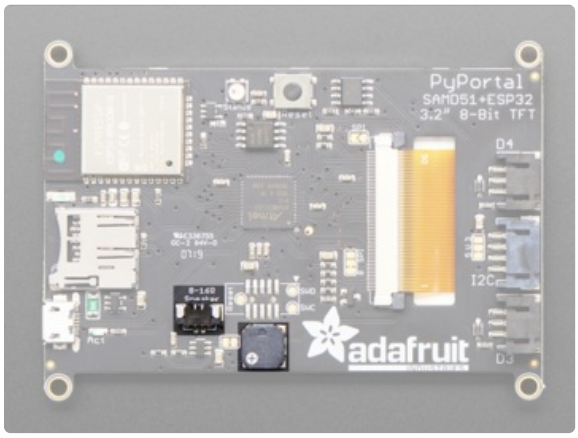


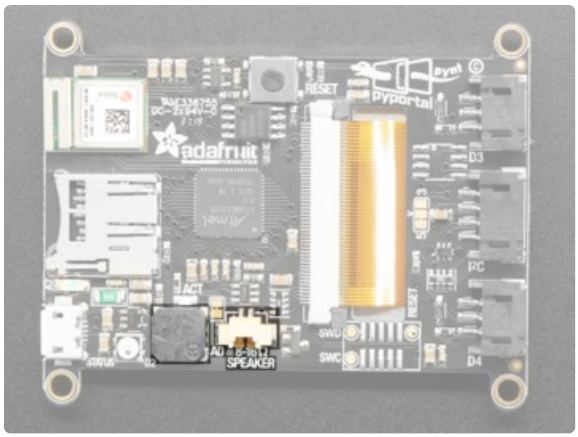
On the left side, there is a **microSD card slot**. A microSD card is the best way to add extra storage to your project and provide space for streams to be processed!

The SD card is on the main SPI port (shared with the ESP32) and a CS line. In CircuitPython, the CS pin is `board.SD_CS`. In Arduino it's digital `32`.

There is also a card detect pin on `board.SD_CARD_DETECT` (CircuitPython) or Arduino `33`

Speaker and Speaker Connector





There is a speaker and a speaker connector.

The grey squarish bit on the bottom is a **speaker**. There is a small class D amplifier connected to the speaker so it can get quite loud!

There is also a **speaker connector**, which is a Molex PicoBlade (https://adafru.it/C8p). You can attach one of the speakers available in the Adafruit shop, or solder a connector to your favorite speaker. If you do, cut the small solder jumper to the left of the buzzer so that you only have one speaker activated (and also it will be louder!)

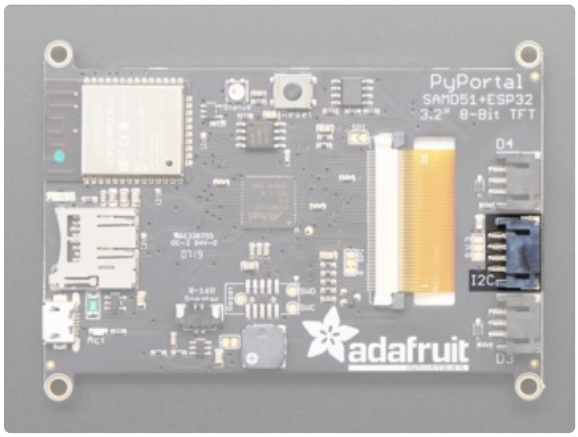
The speaker is connected to the DAC0 output from the SAMD51, via a class D amplifier. The analog output is known as `board.AUDIO_OUT` in CircuitPython. In Arduino it's `A0`.

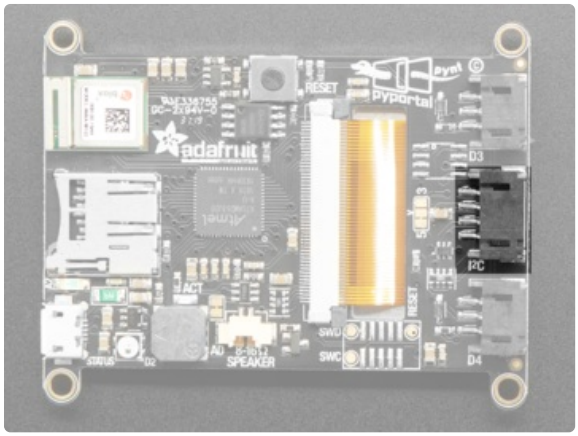
You can disable the speaker amplifier by setting the shutdown pin to output and low. It's on `board.SPEAKER_ENABLE` and Arduino `50`

I2C Connector

If using the I2C connector, you must cut the 5V VCC trace to the left and solder it to the 3V pad

instead - the SAMD51 does not like it if there are even light pullups to 5V and may hang on boot otherwise!

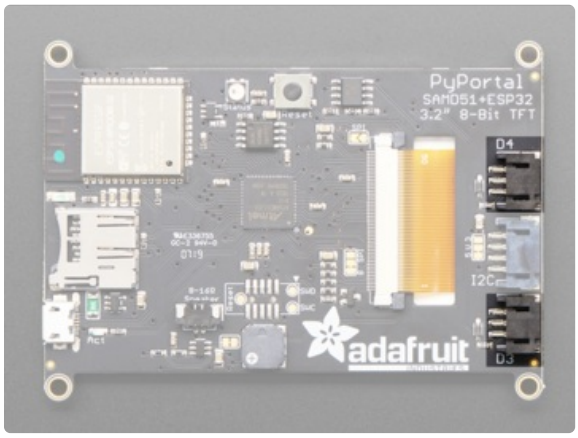


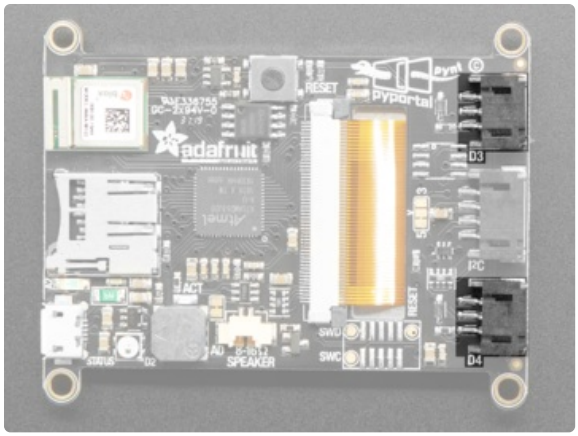


There is a 4-pin JST **I2C connector** in the center on the right, that is STEMMA and Grove compatible. The I2C has pullups to 3.3V power and is connected to the ADT7410 already.

The I2C connector defaults to 5V. There is a jumper you can cut or solder to change it between 5V and 3V.

Digital/Analog Connectors





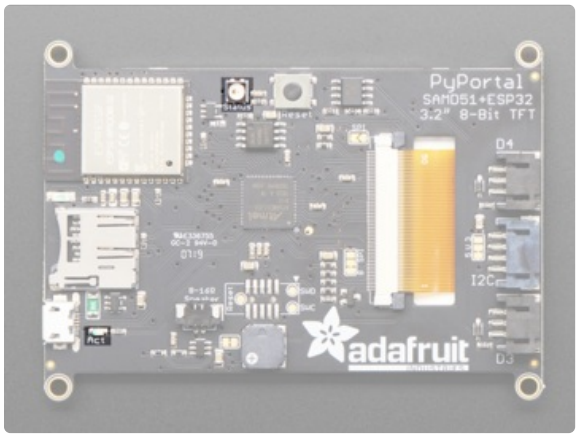
On the right side are two connectors labeled D3 and D4. These are **3-pin JST digital or analog connectors** for sensors or NeoPixels. These pins can be analog inputs or digital I/O.

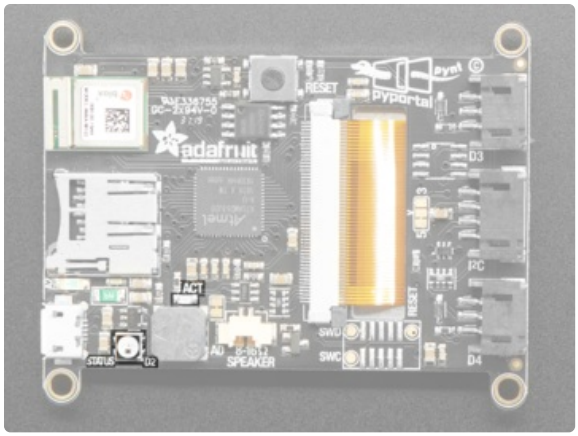
They have protection 1K resistors + 3.6V zener diodes so you can drive an LED directly from the output. Connect to them via `board.D3` and `board.D4` or Arduino `3` and `4`. For analog reading in arduino use **A1** for **D3** and **A3** for **D4** (yeah sorry it's not matchy!)

D3/A1 is the second DAC.

The PyPortal Pynt has the D3 and D4 sockets mislabeled, they should be swapped (to match the pyportal classic, above)

Status LED and NeoPixel



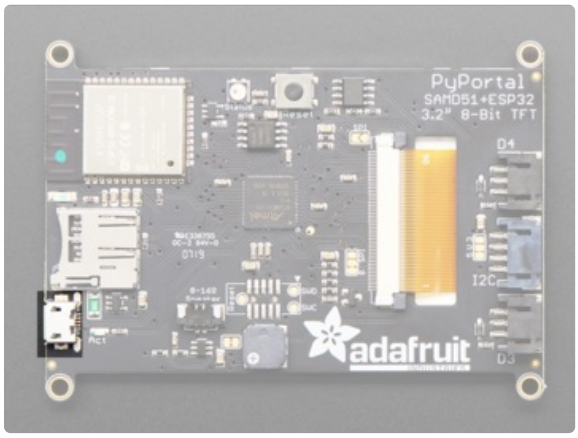


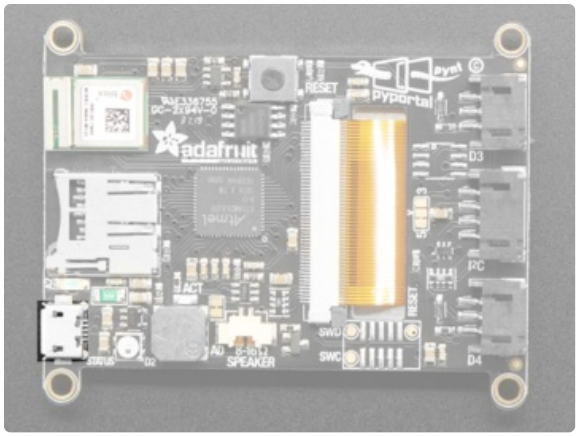
There are two LEDs on the board.

There is the RGB **status NeoPixel** labeled "STATUS". It is connected to `board.NEOPIXEL` or Arduino `2`

As well, there is the **D13 LED**. This is attached to `board.L` and Arduino `13`

USB Connector

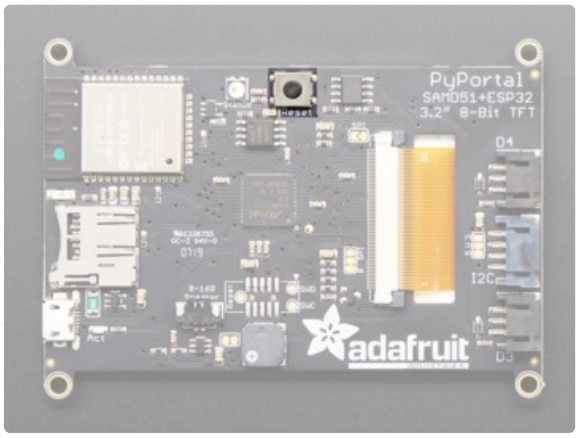


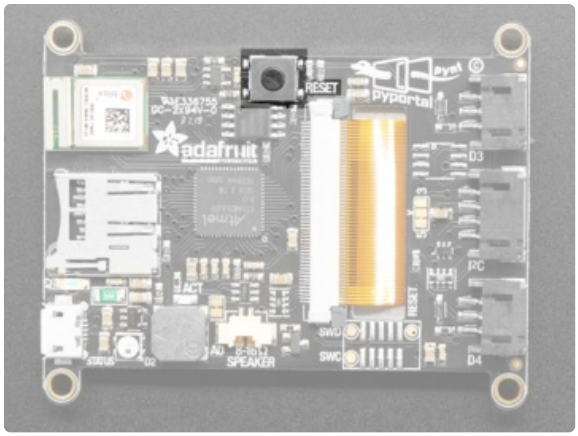


There is one USB port on the board.

On the left side, towards the bottom, is a **USB Micro** port, which is used for powering and programming the board.

Reset Button





The **reset button** is located on the top in the middle.

Click it once to re-start your firmware. Click twice to enter bootloader mode.

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!

Update the UF2 Bootloader

Update the Bootloader on your SAMD51 M4 board to prevent a somewhat rare problem of parts of internal flash being overwritten on power-up.

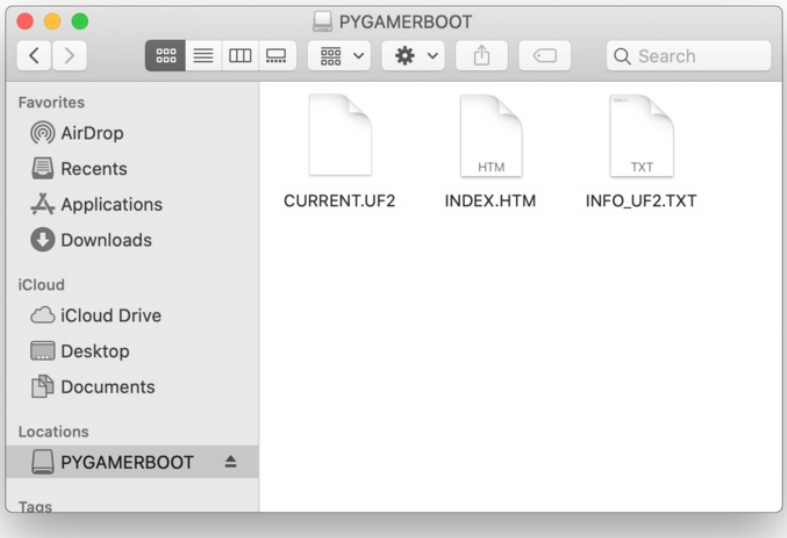
Your SAMD51 M4 board bootloader may need to be updated to fix an intermittent bug that can erase parts of internal flash.

Updating Your Bootloader

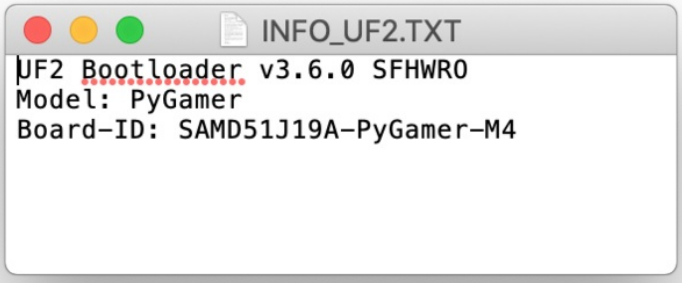
To see if you need to update your bootloader, get the UF2 boot drive to appear as a mounted drive on your computer, in a file browser window. If you're running MakeCode, click the reset button once. If you're running CircuitPython or an Arduino program, double-click the reset button.

When you see the **...BOOT** drive (**FEATHERBOOT**, **METROM4BOOT**, **ITSYM4BOOT**, **PORTALBOOT**, etc.) , click the drive in the file browser window and then double-click the **INFO_UF2.TXT** file to see what's inside.

The example screenshots below are for a PyGamer. What you see for your board will be largely the same except for the board name and the **BOOT** drive name.

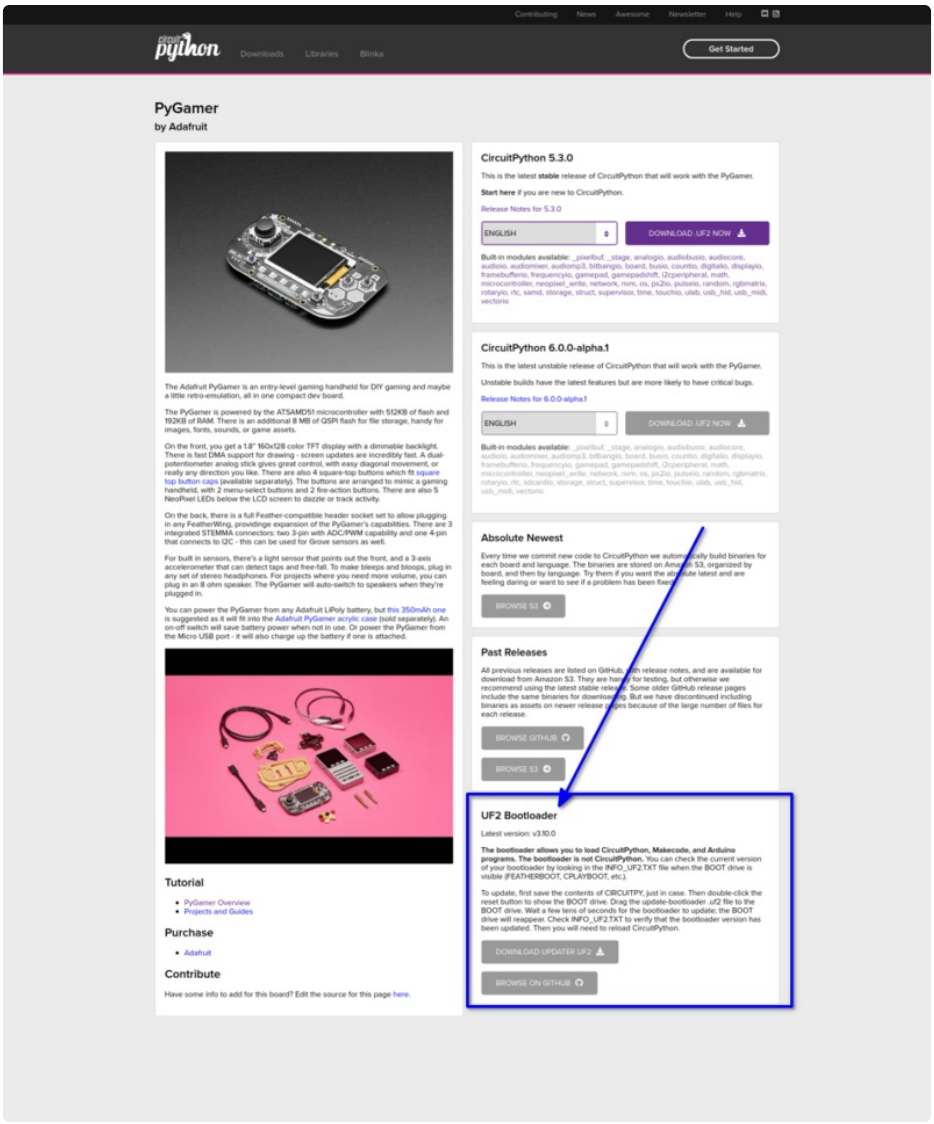


The bootloader version is listed in **INFO_UF2.TXT**. In this example, the version is **v3.6.0**.



If the bootloader version you see is older than ***v3.9.0***, you need to update. For instance, the bootloader above needs to be upgraded.

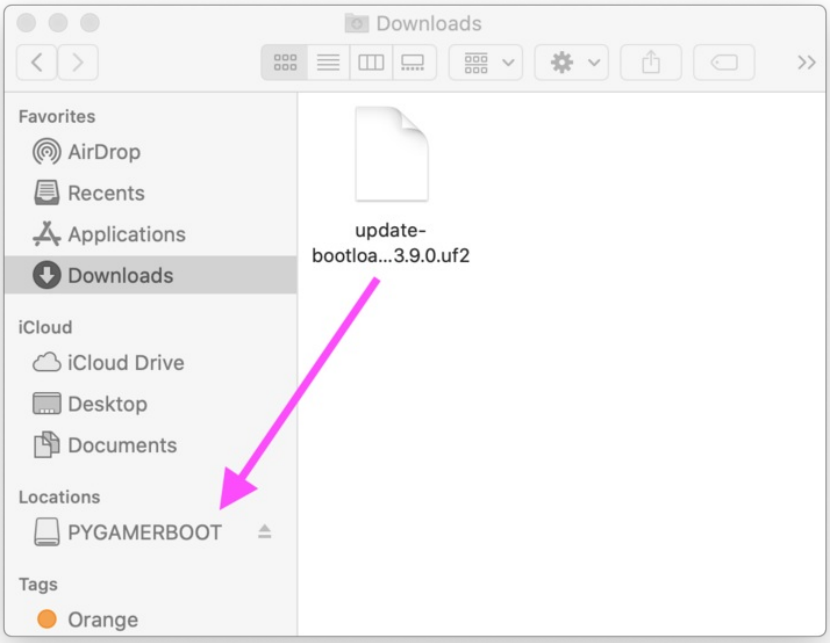
Download the latest version of the bootloader updater from the circuitpython.org Downloads page for your board.



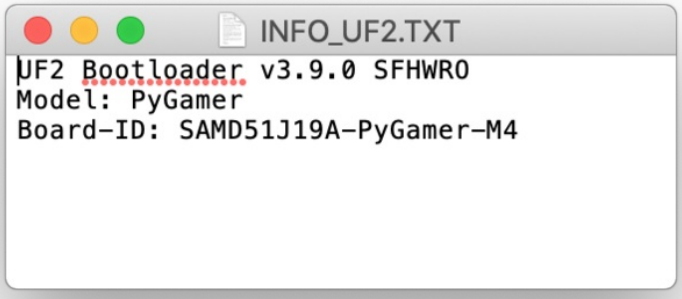
https://adafru.it/Em8

https://adafru.it/Em8

- The bootloader updater will be named **update-bootloader-*name_of_your_board*-v3.9.0.uf2** or some later version. Drag that file from your **Downloads** folder onto the **BOOT** drive:



After you drag the updater onto the boot drive, the red LED on the board will flicker and then blink slowly about five times. A few seconds later, the **BOOT** will appear in the Finder. After that, you can click on the **BOOT** drive and double-click **INFO_UF2.TXT** again to confirm you've updated the bootloader.



Updating Your PyPortal

Your PyPortal that came with AdaBox has older running firmware, libraries and software.

The files are the same for the PyPortal and the PyPortal Pynt.

Before you start, you'll NEED to update your PyPortal!

Step 1 - Update Firmware to Latest

Visit this page and follow the instructions to download and update the latest CircuitPython firmware. You will need to download the latest UF2 firmware file, double-click to enter the bootloader, then drag the UF2 over to the PORTALBOOT drive. (https://adafru.it/EnM)

Your PyPortal will no longer run the example code once you do this - that's OK! We have to finish the other two steps

Step 2 - Update Example Code to Latest

Your PyPortal may have come with an example Quotes demo, or perhaps its blank. Either way, you can install the latest Quotes demo package by clicking here to download a zip:

Note: These are the files that shipped with the Adabox 11 PyPortal. The libraries within this zip are out of date and not compatible with the current version of CircuitPython! The code will not run on the latest CircuitPython. Please use the updated information below to load the updates libraries on your PyPortal.

https://adafru.it/Env

https://adafru.it/Env

The following zip includes the latest libraries as of May 2020. Please visit https://circuitpython.org/libraries to download the latest bundle and update all of the libraries to the latest.

https://adafru.it/KVb

https://adafru.it/KVb

First, delete all the files from your CIRCUITPY drive (so you don't have any old lingering files)

Unzip this and go into the **boards/pyportal/5.x** folder. You will see files such as **code.py** and **pyportal_startup.bmp**. Copy over everything in the **boards/pyportal/5.x** folder. That means **code.py** and the **lib** folder will be in the 'root' directory of CIRCUITPY.

Step 3 - Check Display & Add WiFi Secrets

Once everything is fully copied, you will be prompted to edit **secrets.py**

Do that to enable WiFi support

Step 4 - If you are getting odd errors

If your filesystem somehow got corrupted, or you're getting unusual errors, try erasing the filesystem to clear out any corrupt files (https://adafru.it/Den), by:

- download the QSPI Eraser UF2 file (https://adafru.it/Den)
- load it onto the PyPortal by entering the bootloader and dragging it onto PORTALBOOT
- wait until the back LED goes from yellow to green
- Go to Step 1 to re-load the firmware and demo code!

Install 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/Egk

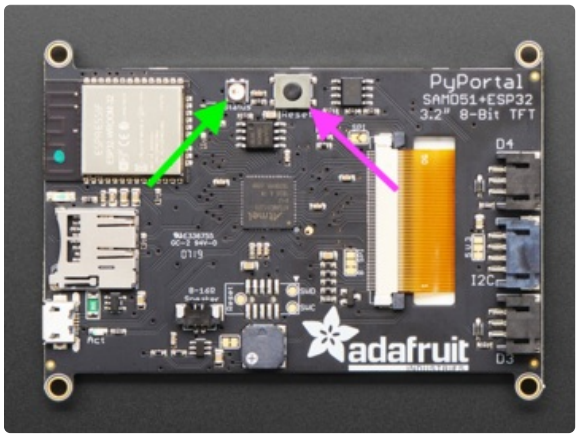
https://adafru.it/Egk

https://adafru.it/HFd

https://adafru.it/HFd

Click the link above to download the latest version of CircuitPython for the PyPortal.

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

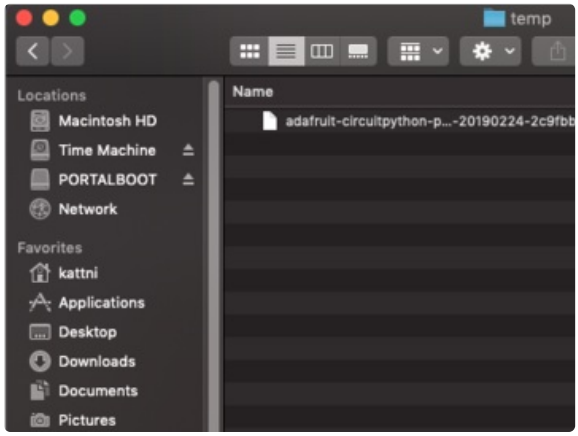


Plug your PyPortal into your computer using a known-good USB cable.

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.

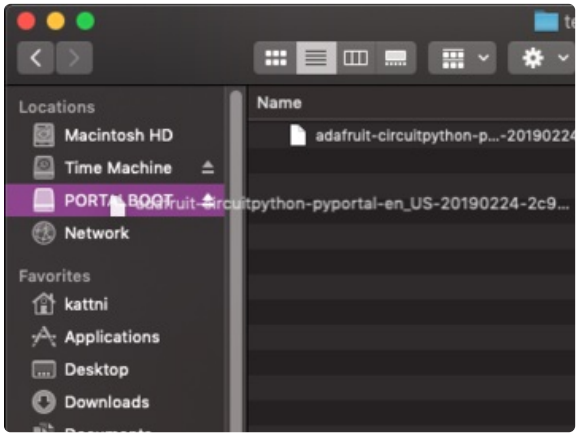
Double-click the **Reset** button on the top in the middle (magenta arrow) on your board, and you will see the NeoPixel RGB LED (green arrow) turn green. If it turns red, check the USB cable, try another USB port, etc. **Note:** The little red LED next to the USB connector will pulse red. That's ok!

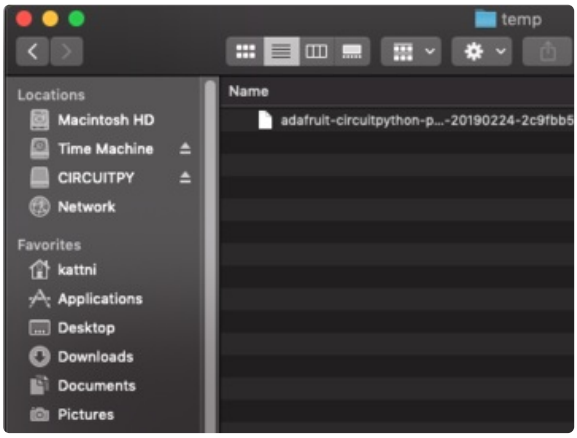
If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!



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

Drag the **adafruit-circuitpython-pyportal-<whatever>.uf2** file to **PORTALBOOT.**





The LED will flash. Then, the **PORTALBOOT** 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! :)

PyPortal Default Files

Click below to download a zip of the files that shipped on the PyPortal or PyPortal Pynt.

https://adafru.it/Env

https://adafru.it/Env

https://adafru.it/HFf

https://adafru.it/HFf

Installing Mu Editor

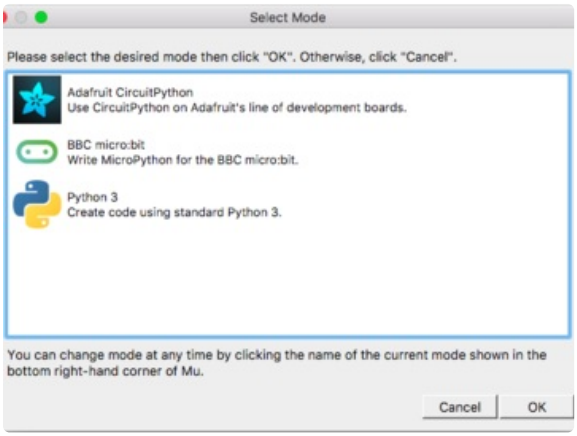
Mu is a simple code editor that works with the Adafruit CircuitPython boards. It's written in Python and works on Windows, MacOS, Linux and Raspberry Pi. The serial console is built right in so you get immediate feedback from your board's serial output!

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

Download and Install Mu

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

Using Mu



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

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

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

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

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

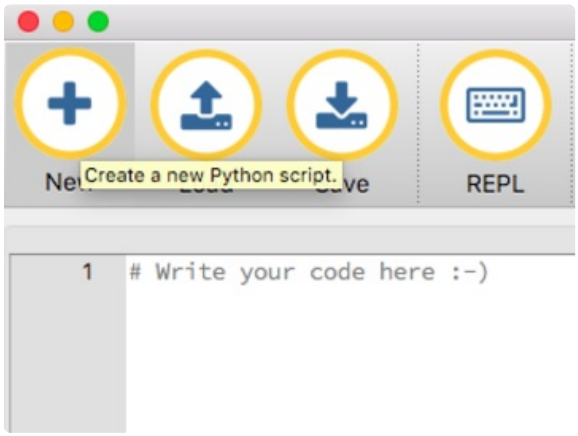
Creating and Editing Code

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

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

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

Creating Code



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

Copy and paste the following code into your editor:

```
import board
import digitalio
import time

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

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

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

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

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

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

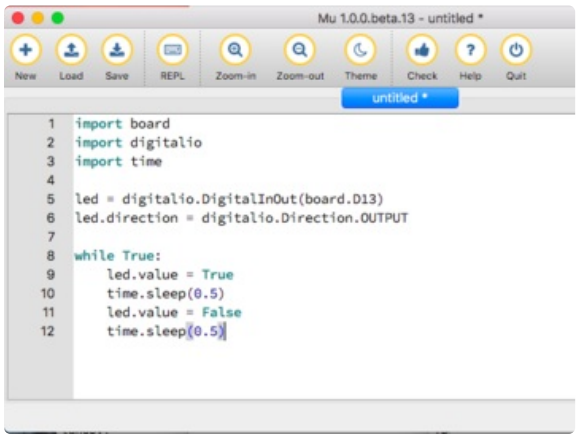
For Adafruit CLUE, you'll need to use `board.D17` instead of `board.LED`. The rest of the code remains the same. Make the following change to the `led =` line:

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

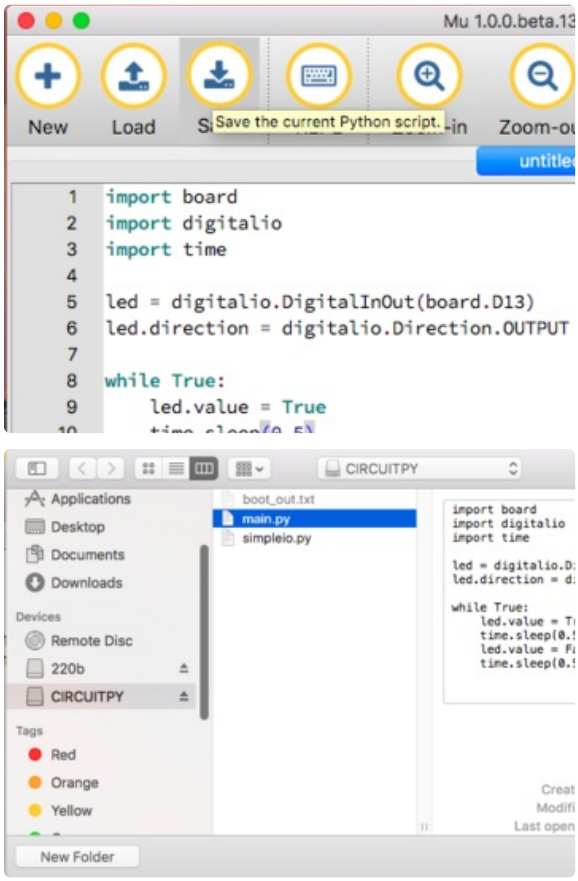
If you are using Adafruit ItsyBitsy nRF52840, you will need to edit the code to use board.BLUE_LED as shown below!

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

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



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



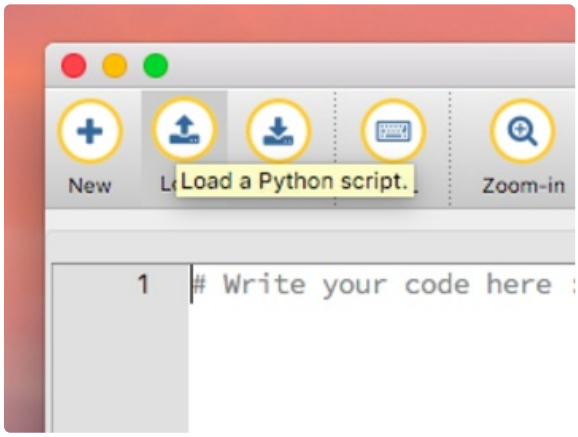
Save this file as **code.py** on your CIRCUITPY drive.

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

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

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

Editing Code



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

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

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

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

Don't Click Reset or Unplug!

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

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

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

There are a few ways to avoid this:

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

Recommended editors:

- **mu** (https://adafru.it/Be6) is an editor that safely writes all changes (it's also our recommended editor!)
- **emacs** (https://adafru.it/xNA) is also an editor that will fully write files on save (https://adafru.it/Be7)
- Sublime Text **(https://adafru.it/xNB)** safely writes all changes
- Visual Studio Code **(https://adafru.it/Be9)** appears to safely write all changes
- **gedit** on Linux appears to safely write all changes
- IDLE (https://adafru.it/IWB), in Python 3.8.1 or later, was fixed (https://adafru.it/IWD) to write all changes immediately
- thonny (https://adafru.it/Qb6) fully writes files on save

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

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

We *don't* recommend these editors:

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

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

2. Eject or Sync the Drive After Writing

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

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

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

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

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

Back to Editing Code...

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

```
import board
import digitalio
import time

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

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

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

Exploring Your First CircuitPython Program

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

Here is the original code again:

```
import board
import digitalio
import time

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

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

Imports & Libraries

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

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

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

Setting Up The LED

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

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

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

Loop-de-loops

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

Inside our loop, we have four items:

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

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

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

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

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

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

What Happens When My Code Finishes Running?

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

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

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

What if I don't have the loop?

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

The simplest loop would be:

```
while True:
    pass
```

And remember - you can press to exit the loop.

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

More Changes

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

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

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

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

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

Naming Your Program File

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

Connecting to the Serial Console

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

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

This line would result in:

```
Hello, world!
```

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

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

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

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

```
sudo apt purge modemmanager
```

Are you using Mu?

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

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

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

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

Setting Permissions on Linux

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

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

```
sudo adduser $USER dialout
```

After running the command above, reboot your machine to gain access to the group. On other Linux distributions, the group you need may be different. See Advanced Serial Console on Mac and

Linux (https://adafru.it/AAI) for details on how to add yourself to the right group.

Using Something Else?

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

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

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

Interacting with the Serial Console

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

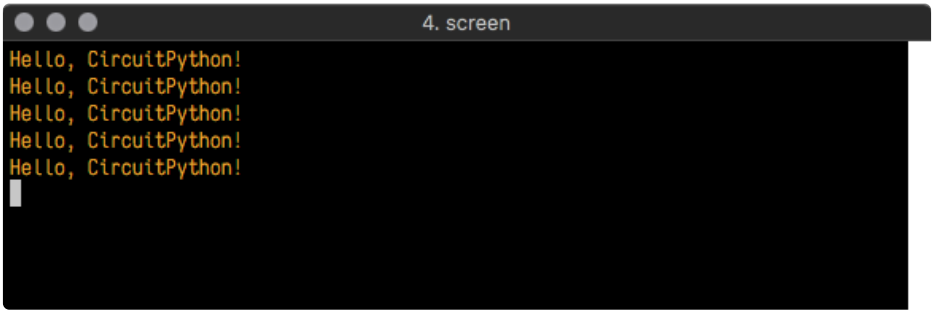
The code you wrote earlier has no output to the serial console. So, we're going to edit it to create some output.

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

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.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.

```
code.py
import board
import digitalio
import time

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

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

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

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

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

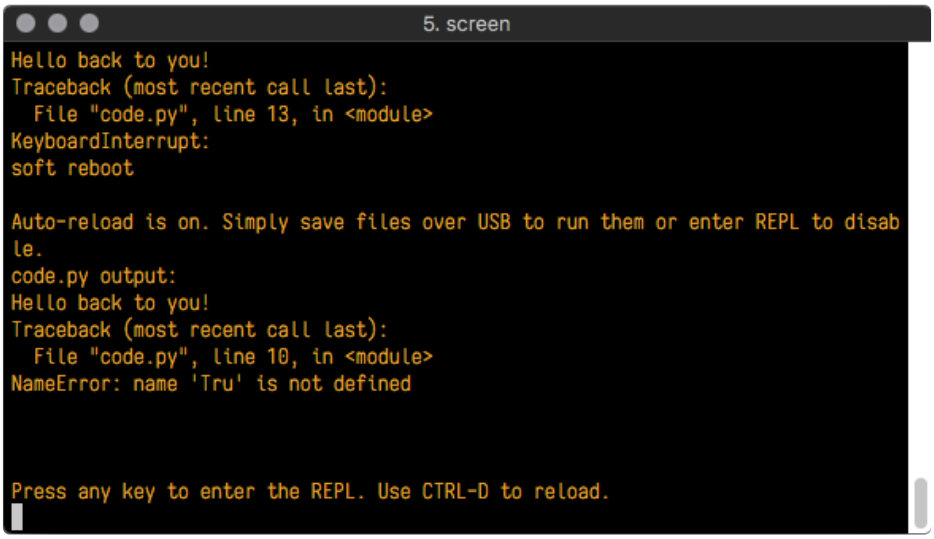
The `Traceback (most recent call last):` is telling you the last thing your board was doing before you saved your file. This is normal behavior and will happen every time the board resets. This is really handy for troubleshooting. Let's introduce an error so we can see how it is used.

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

```
code.py
import board
import digitalio
import time

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

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

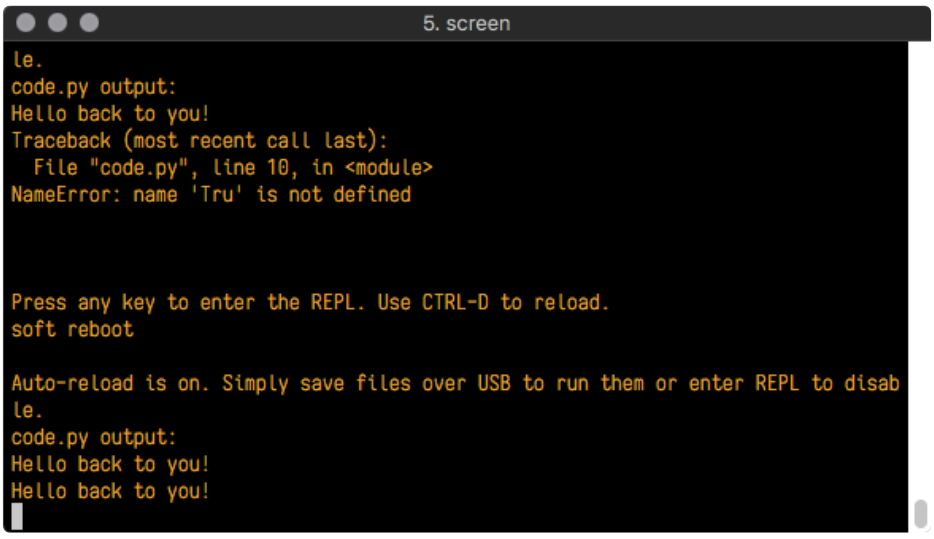
Save your file. You will notice that your red LED will stop blinking, and you may have a colored status LED blinking at you. This is because the code is no longer correct and can no longer run properly. We need to fix it!

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



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

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



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

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

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

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

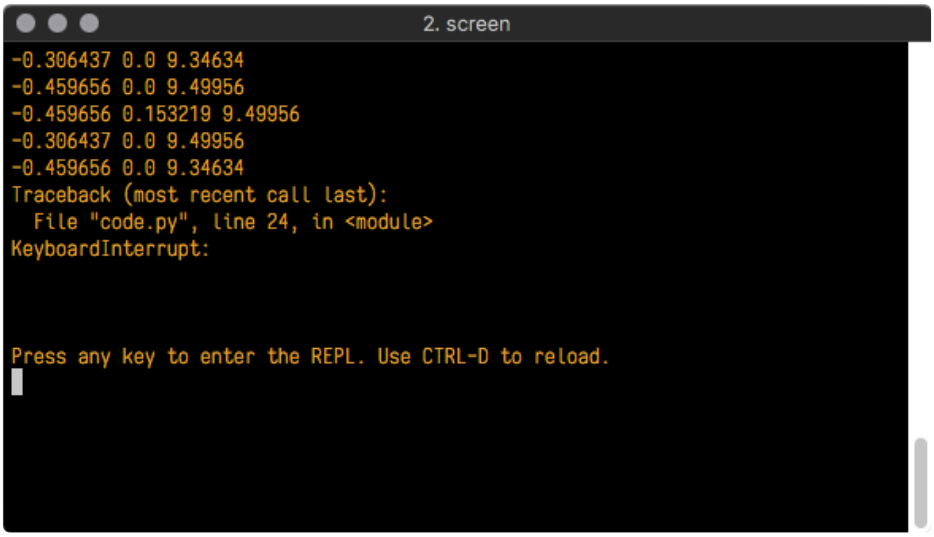
The REPL

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

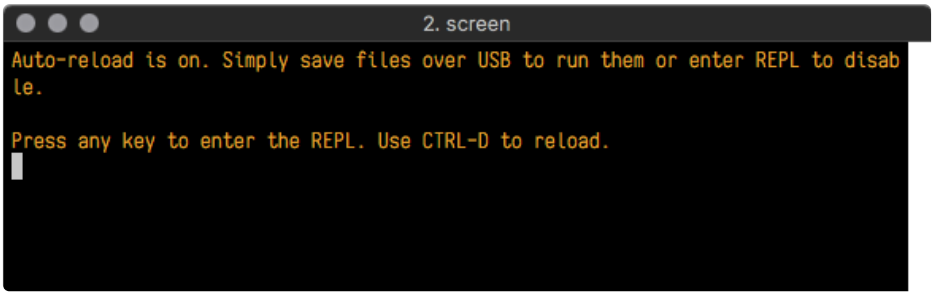
To use the REPL, you first need to be connected to the serial console. Once that connection has been established, you'll want to press **Ctrl** + **C**.

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

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



If there is no code running, you will enter the REPL immediately after pressing Ctrl + C. There is no information about what your board was doing before you interrupted it because there is no code running.



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

If you have trouble getting to the `>>>` prompt, try pressing Ctrl + C a few more times.

The first thing you get from the REPL is information about your board.

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

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

This is followed by the CircuitPython prompt.

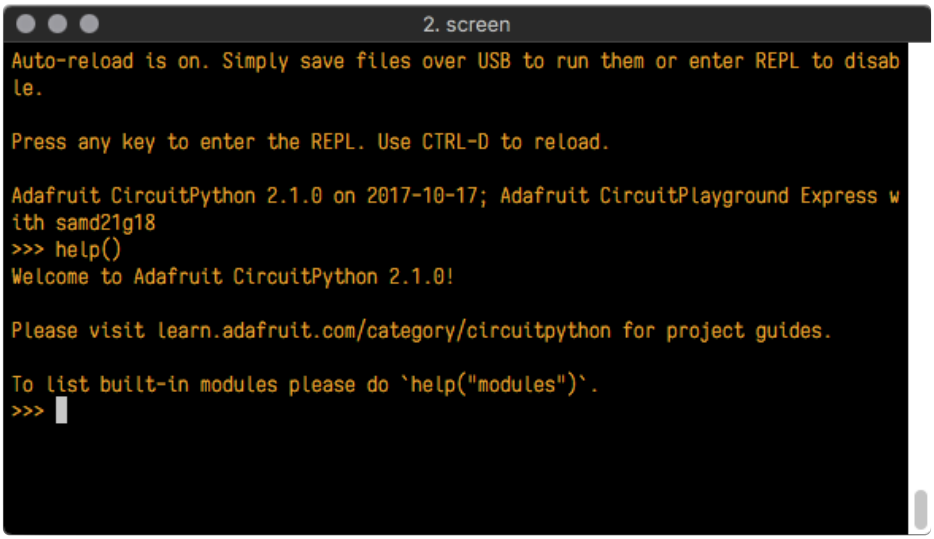
```
>>>
```

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

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

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

Then press enter. You should then see a message.

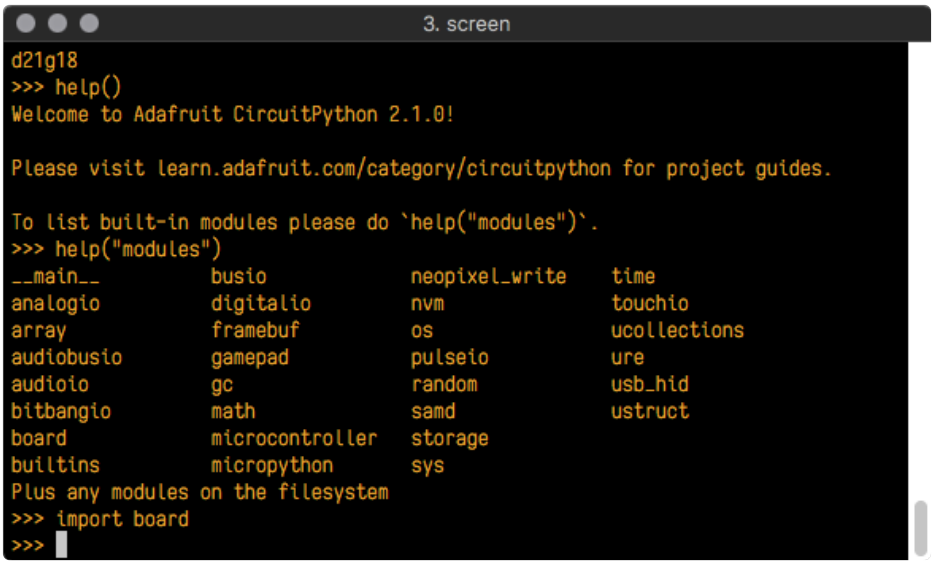


First part of the message is another reference to the version of CircuitPython you're using. Second, a URL for the CircuitPython related project guides. Then... wait. What's this? `To list built-in modules, please do `help("modules")`.` Remember the libraries you learned about while going through creating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

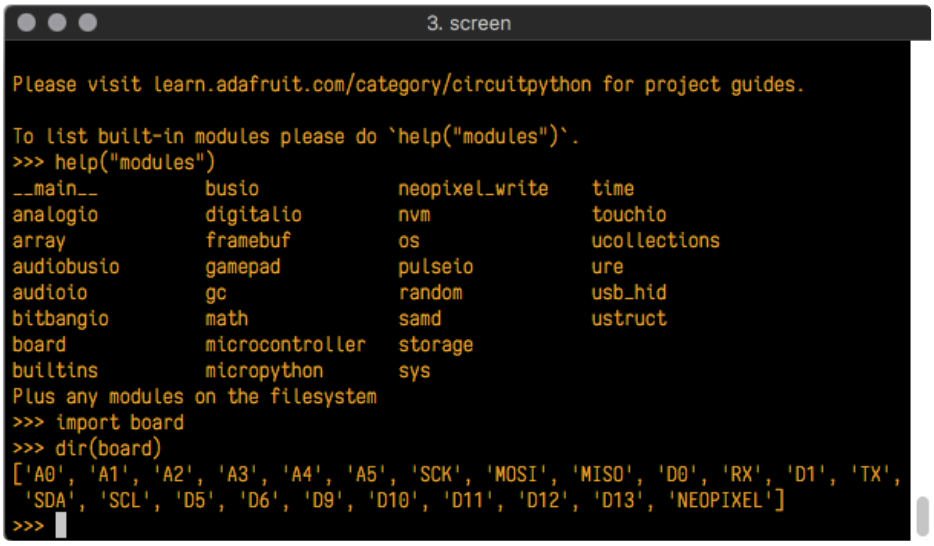
Type `help("modules")` into the REPL next to the prompt, and press enter.

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.

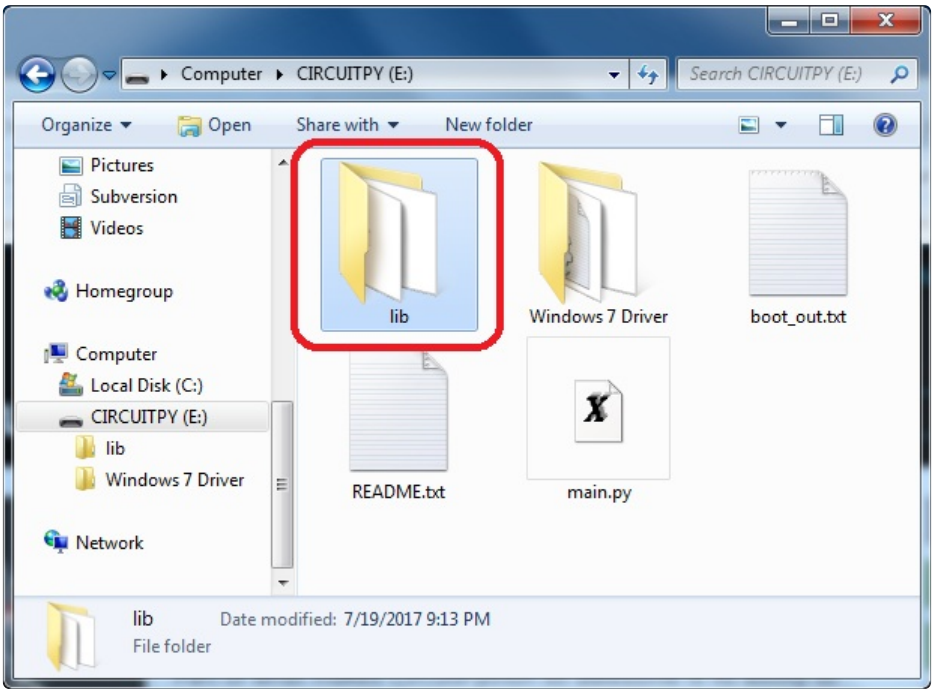
You can return to the REPL at any time!

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

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

libraries.

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

Installing the CircuitPython Library Bundle

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

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

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

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

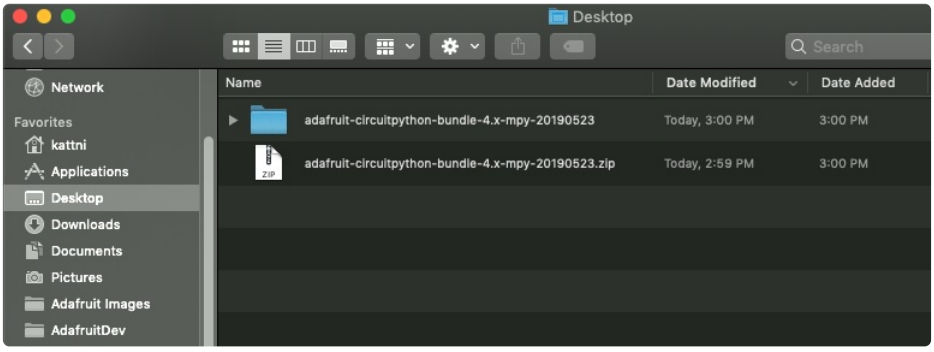
https://adafru.it/ENC

https://adafru.it/ENC

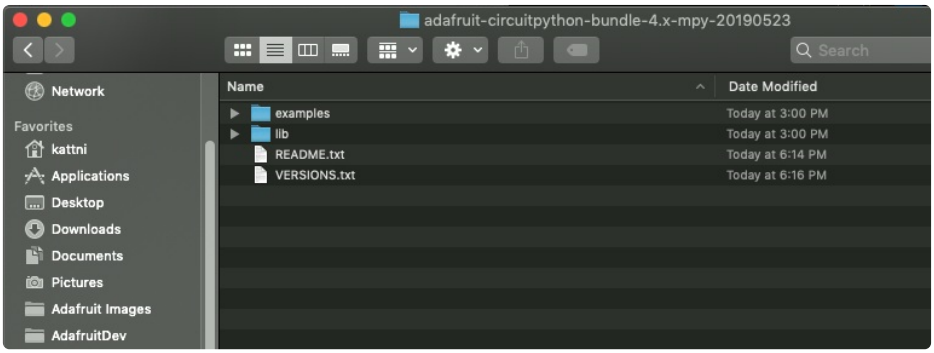
If you need another version, you can also visit the bundle release page (https://adafru.it/Ayy) which will let you select exactly what version you're looking for, as well as information about changes.

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

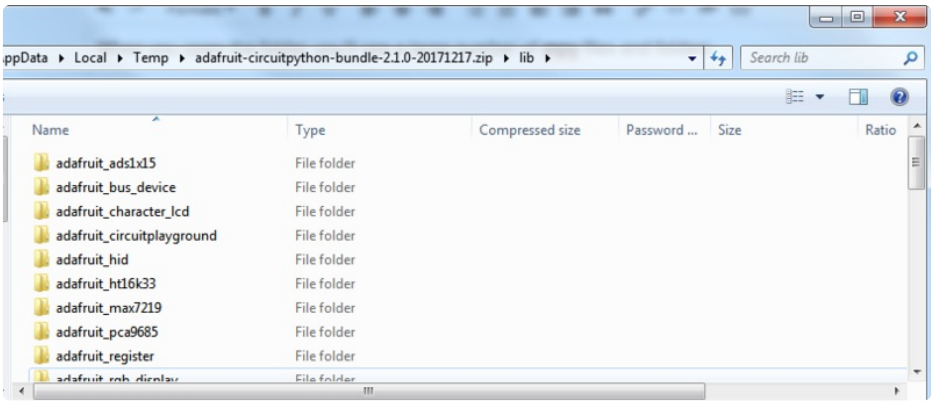
After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.



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



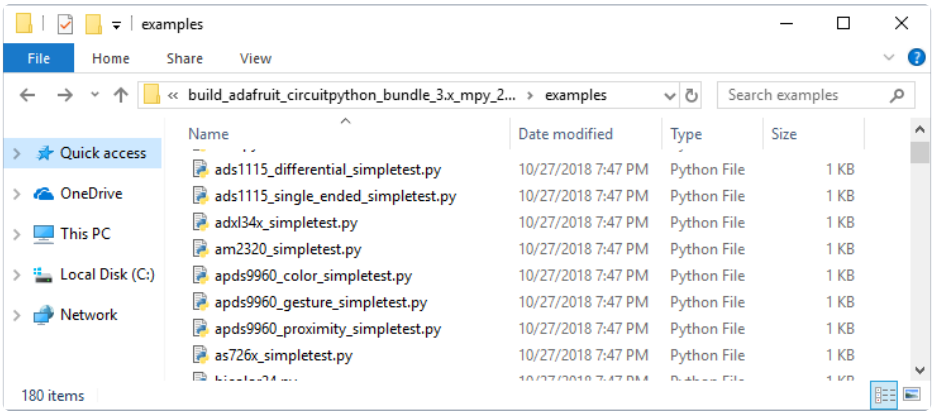
Now open the lib folder. When you open the folder, you'll see a large number of **mpy** files and folders



Example Files

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

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



Copying Libraries to Your Board

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

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

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

Example: `ImportError` Due to Missing Library

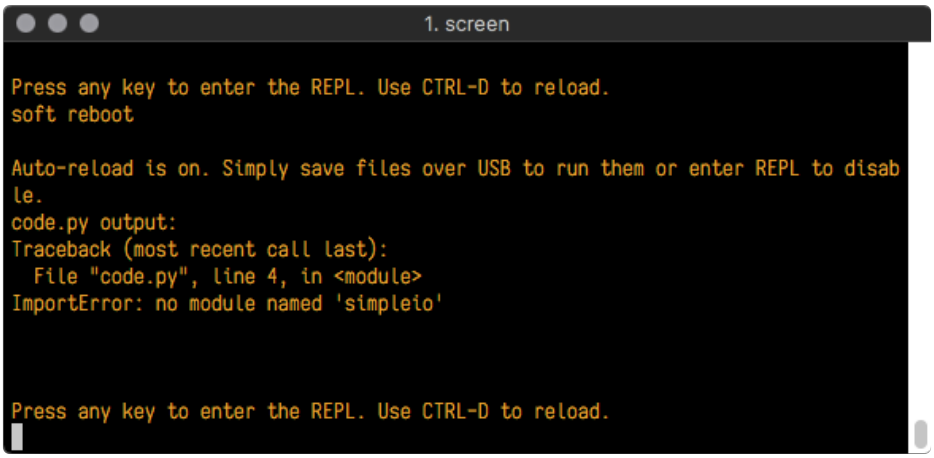
If you choose to load libraries as you need them, you may write up code that tries to use a library you haven't yet loaded. We're going to demonstrate what happens when you try to utilise a library that you don't have loaded on your board, and cover the steps required to resolve the issue.

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

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

```
import board
import time
import simpleio

led = simpleio.DigitalOut(board.D13)

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

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



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.

CircuitPython Pins and Modules

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

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

CircuitPython Pins

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

import board

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

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

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

Here is the output for the QT Py.

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

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

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

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

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

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

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

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

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

I2C, SPI, and UART

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

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

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

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

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

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

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

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

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

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

What Are All the Available Names?

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

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

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

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

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

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

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

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

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

Microcontroller Pin Names

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

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

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

CircuitPython Built-In Modules

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

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

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

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

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

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

Frequently Asked Questions

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

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

I have to continue using an older version of CircuitPython; where can I find compatible libraries?

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

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

Is ESP8266 or ESP32 supported in CircuitPython? Why not?

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

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

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

How do I connect to the Internet with CircuitPython?

If you'd like to add WiFi support, check out our guide on ESP32/ESP8266 as a co-processor. (https://adafru.it/Dwa)

Is there asyncio support in CircuitPython?

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

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

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

What is a `MemoryError`?

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

What do I do when I encounter a `MemoryError`?

Try resetting your board. Each time you reset the board, it reallocates the memory. While this is unlikely to

resolve your issue, it's a simple step and is worth trying.

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

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

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

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

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

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

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

You can download `mpy-cross` for your operating system from https://adafruit-circuit-python.s3.amazonaws.com/index.html?prefix=bin/mpy-cross/ (https://adafru.it/QDK). Almost any version will do. The format for .mpy files has not changed since CircuitPython 4.x.

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

How do I check how much memory I have free?

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

Will give you the number of bytes available for use.

Does CircuitPython support interrupts?

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

Does Feather M0 support WINC1500?

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

Can AVRs such as ATmega328 or ATmega2560 run CircuitPython?

No.

Commonly Used Acronyms

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

Troubleshooting

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

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

Always Run the Latest Version of CircuitPython and Libraries

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

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

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

I have to continue using CircuitPython 5.x, 4.x, 3.x or 2.x, where can I find compatible libraries?

We are no longer building or supporting the CircuitPython 2.x, 3.x, 4.x or 5.x library bundles. We highly encourage you to update CircuitPython to the latest version **(https://adafru.it/Em8) and use** the current version of the libraries **(https://adafru.it/ENC).** However, if for some reason you cannot update, you can find the last available 2.x build here (https://adafru.it/FJA), the last available 3.x build here (https://adafru.it/FJB), the last available 4.x build here (https://adafru.it/QDL), and the last available 5.x build here (https://adafru.it/QDJ).

CPLAYBOOT, TRINKETBOOT, FEATHERBOOT, or GEMMABOOT Drive Not Present

You may have a different board.

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

MacOS

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

Windows 10

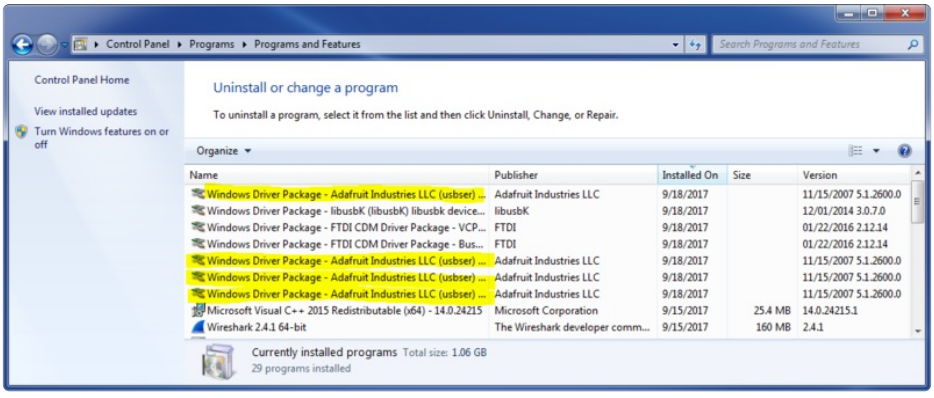
Did you install the Adafruit Windows Drivers package by mistake, or did you upgrade to Windows 10 with the driver package installed? You don't need to install this package on Windows 10 for most Adafruit boards. The old version (v1.5) can interfere with recognizing your device. Go to **Settings** ->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

Windows 7 or 8.1

Version 2.5.0.0 or later of the Adafruit Windows Drivers will fix the missing *boardname*BOOT drive problem on Windows 7 and 8.1. To resolve this, first uninstall the old versions of the drivers:

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

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

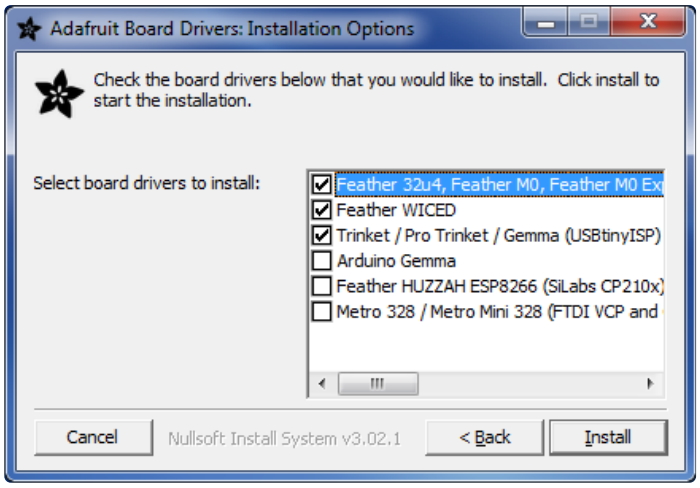


- Now install the new 2.5.0.0 (or higher) Adafruit Windows Drivers Package:

https://adafru.it/AB0

https://adafru.it/AB0

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



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

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

Windows Explorer Locks Up When Accessing `boardnameBOOT` Drive

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

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

uninstallation.

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

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

CIRCUITPY Drive Does Not Appear

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

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

Windows 7 and 8.1 Problems

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

Serial Console in Mu Not Displaying Anything

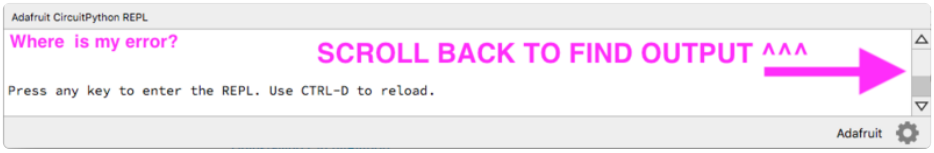
There are times when the serial console will accurately not display anything, such as, when no code is currently running, or when code with no serial output is already running before you open the console. However, if you find yourself in a situation where you feel it should be displaying something like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

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

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



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

More complex errors take even more lines!

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



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

CircuitPython RGB Status Light

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

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

Here's what the colors and blinking mean:

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

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

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

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

ValueError: Incompatible `.mpy` file.

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

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

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

CIRCUITPY Drive Issues

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

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

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

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

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

Easiest Way: Use `storage.erase_filesystem()`

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

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

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

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

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

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

1. Download the correct erase file:

https://adafru.it/AdI

https://adafru.it/AdI

https://adafru.it/AdJ

https://adafru.it/AdJ

https://adafru.it/EVK

https://adafru.it/EVK

https://adafru.it/AdK

https://adafru.it/AdK

https://adafru.it/EoM

https://adafru.it/EoM

https://adafru.it/DjD

https://adafru.it/DjD

https://adafru.it/DBA

https://adafru.it/DBA

https://adafru.it/Eca

https://adafru.it/Eca

https://adafru.it/Gnc

https://adafru.it/Gnc

https://adafru.it/GAN

https://adafru.it/GAN

https://adafru.it/GAO

https://adafru.it/GAO

https://adafru.it/Jat

https://adafru.it/Jat

https://adafru.it/Q5B

https://adafru.it/Q5B

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

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

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

If you haven't already downloaded the latest release of CircuitPython for your board, check out the

installation page (https://adafru.it/Amd). You'll also need to install your libraries and code!

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

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

1. Download the erase file:

https://adafru.it/AdL

https://adafru.it/AdL

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

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

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

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

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

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

Running Out of File Space on Non-Express Boards

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

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

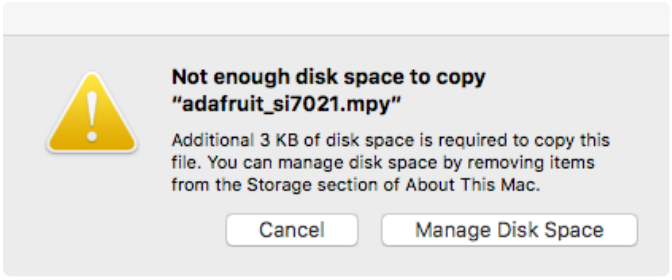
Delete something!

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

Use tabs

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

MacOS loves to add extra files.



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

Prevent & Remove MacOS Hidden Files

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

```
ls -l /Volumes
```

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

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

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

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

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

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

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

Copy Files on MacOS Without Creating Hidden Files

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

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

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

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

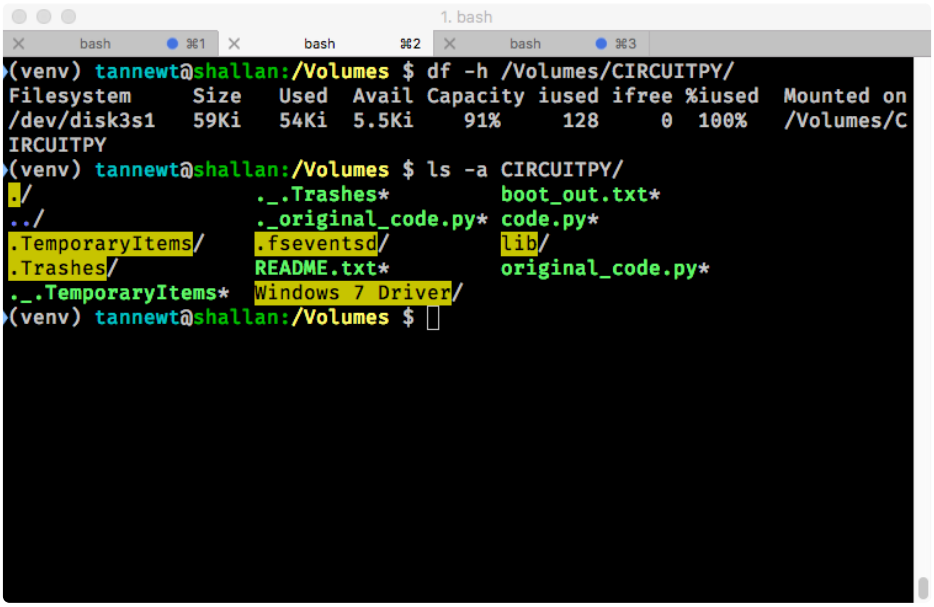
```
cp -rX folder_to_copy /Volumes/CIRCUITPY
```

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

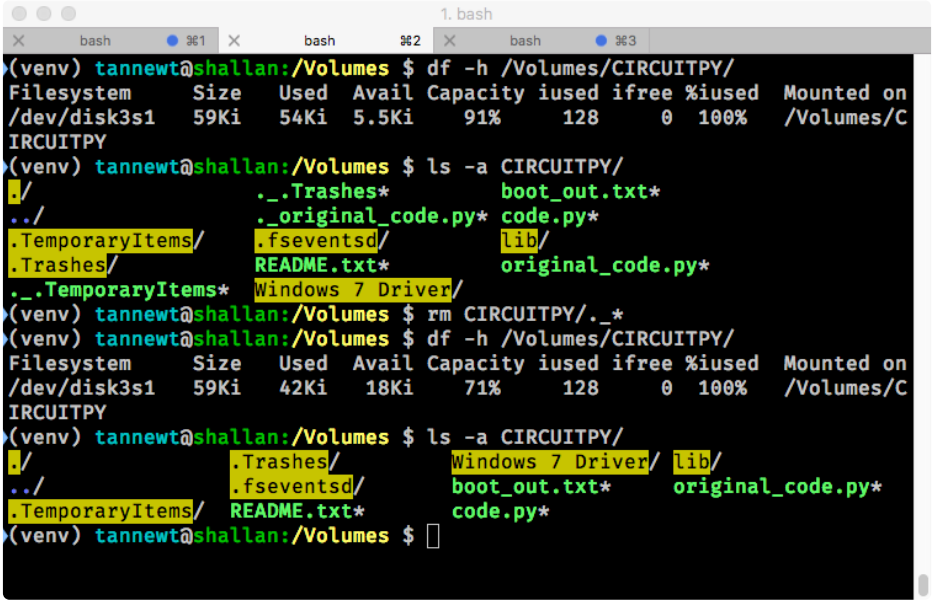
```
# if lib does not exist, you'll create a file named lib !
cp -X foo.mpy /Volumes/CIRCUITPY/lib
# This is safer, and will complain if a lib folder does not exist.
cp -X foo.mpy /Volumes/CIRCUITPY/lib/
```

Other MacOS Space-Saving Tips

If you'd like to see the amount of space used on the drive and manually delete hidden files here's how to do so. First list the amount of space used on the `CIRCUITPY` drive with the **df** command:



Lets remove the `._` files first.



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

Device locked up or boot looping

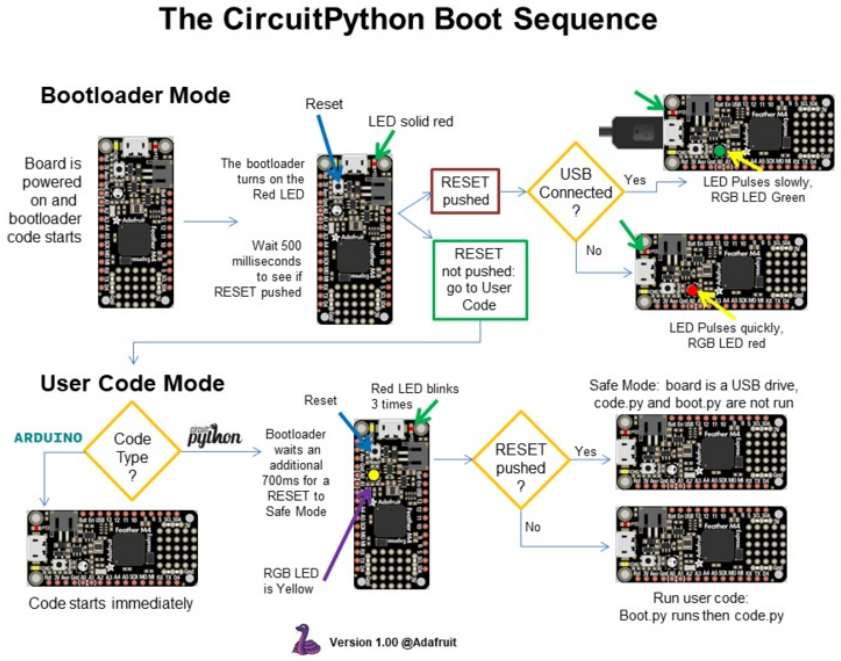
In rare cases, it may happen that something in your **code.py** or **boot.py** files causes the device to get locked up, or even go into a boot loop. These are not your everyday Python exceptions, typically it's the result of a deeper problem within CircuitPython. In this situation, it can be difficult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

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

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

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

Refer to the following diagram for boot sequence details:



Uninstalling CircuitPython

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

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

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

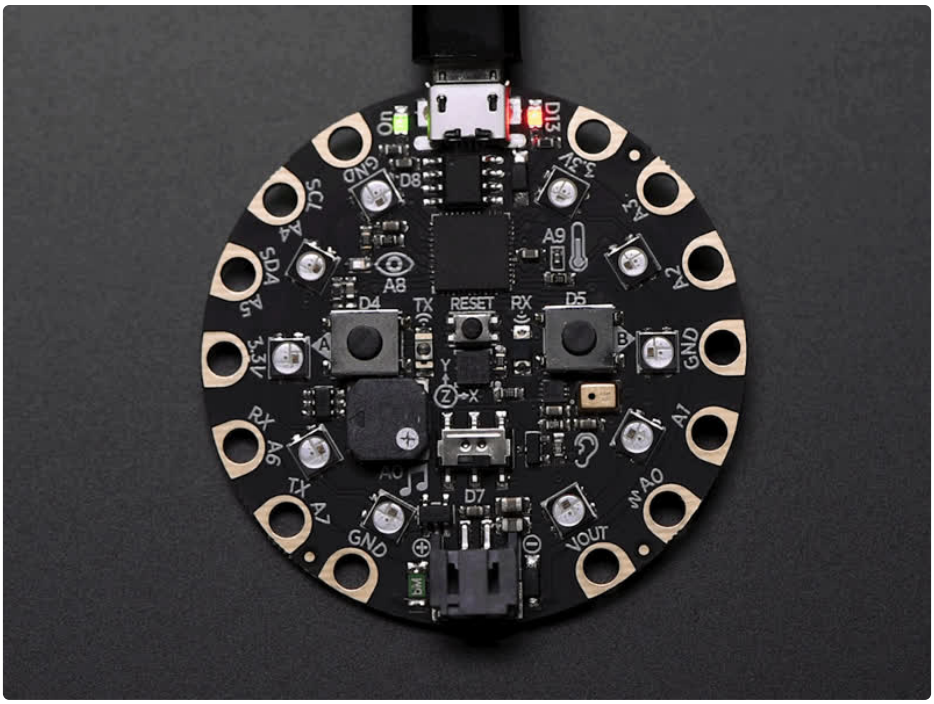
Backup Your Code

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

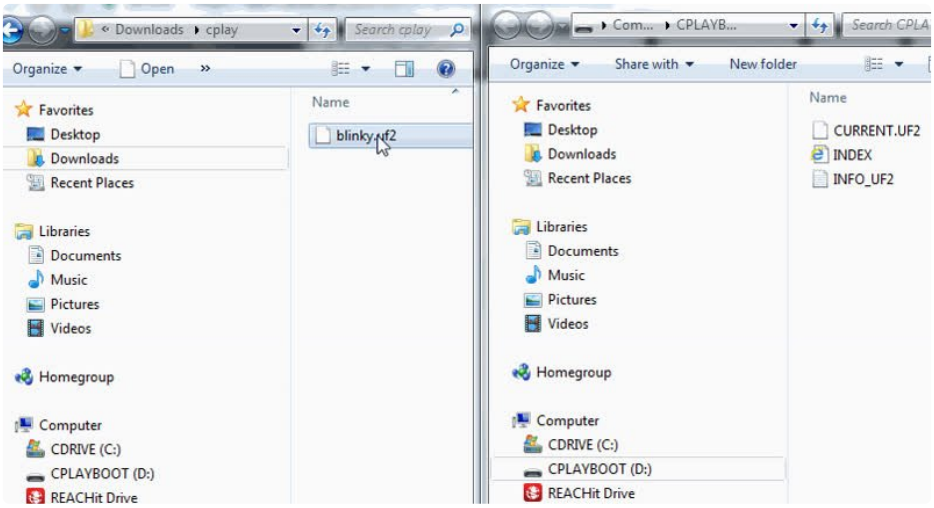
Moving Circuit Playground Express to MakeCode

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

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



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



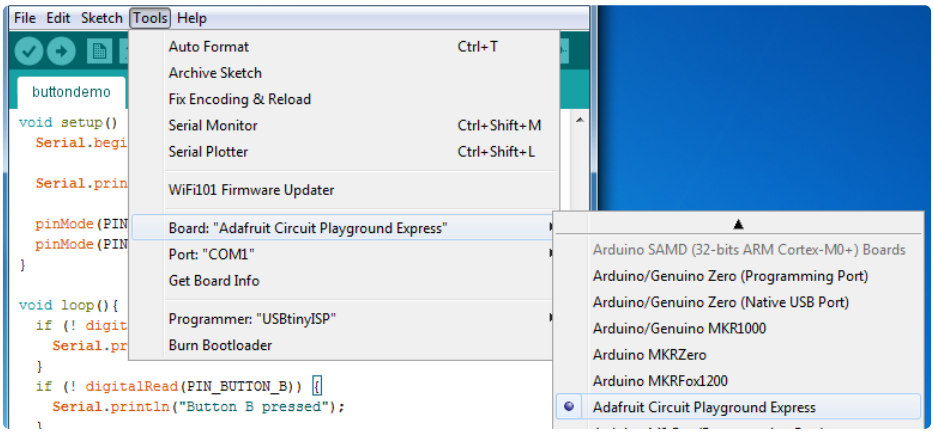
Your MakeCode is now running and CircuitPython has been removed. Going forward you only have to **single click** the reset button

Moving to Arduino

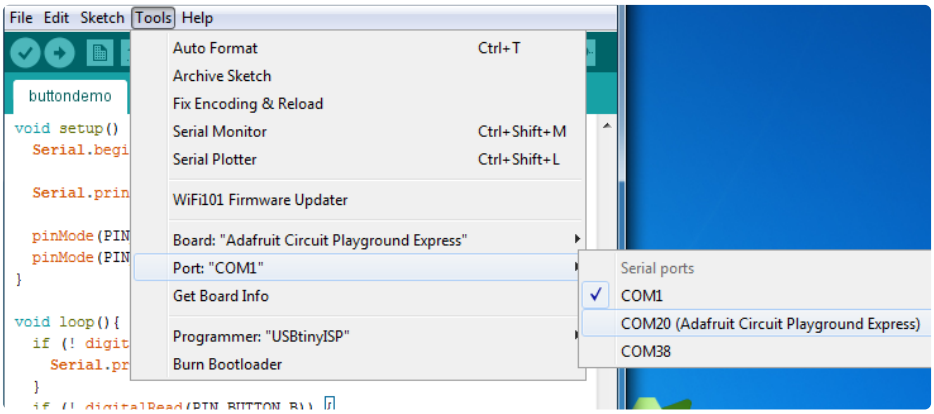
If you want to change your firmware to Arduino, it's also pretty easy.

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

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



Select the correct matching Port:



Create a new simple Blink sketch example:

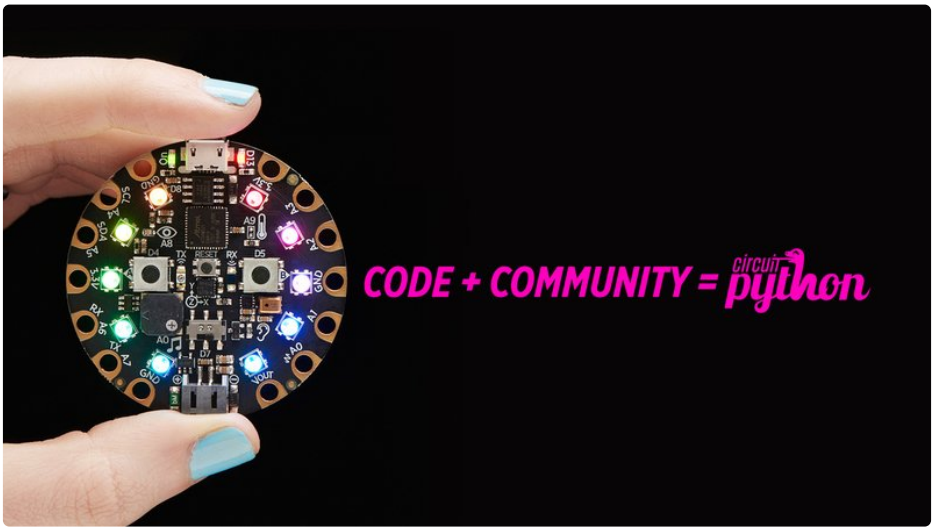
```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

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

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

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

automatically reset when you upload

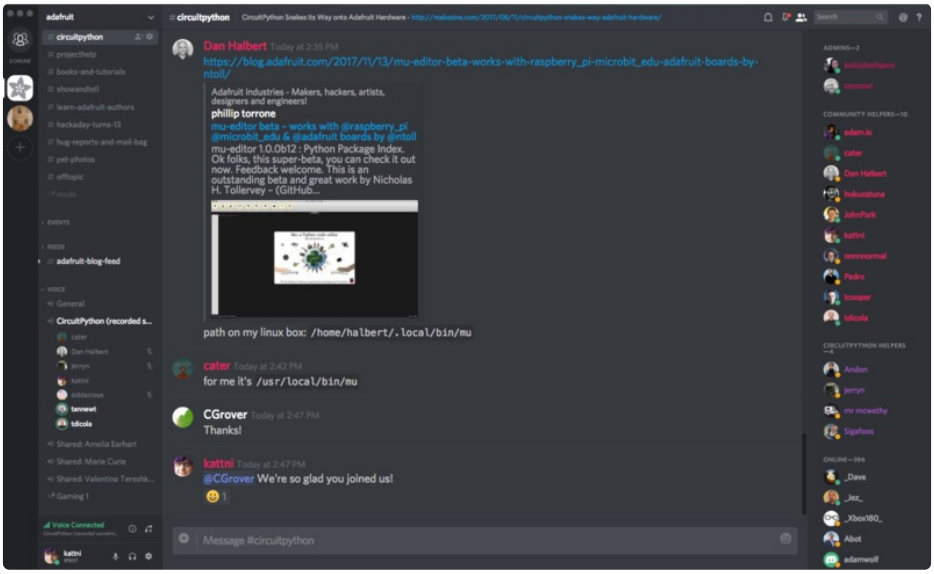
Welcome to the Community!



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

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

Adafruit Discord



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

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

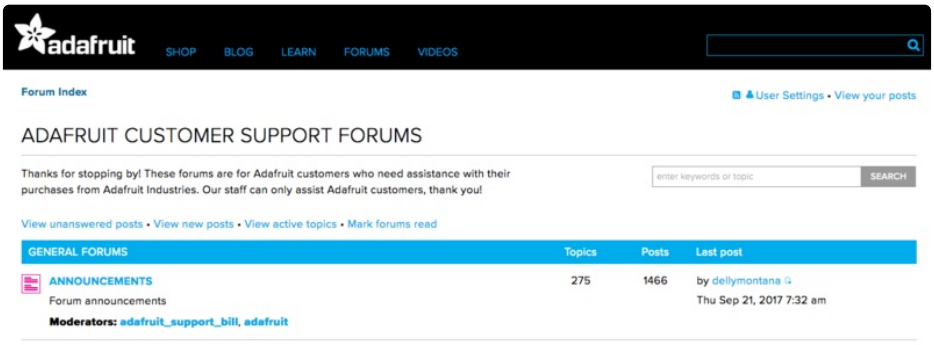
The help with CircuitPython channel is where to go with your CircuitPython questions. #help-with-circuitpython is there for new users and developers alike so feel free to ask a question or post a comment! Everyone of any experience level is welcome to join in on the conversation. We'd love to hear what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

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

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

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

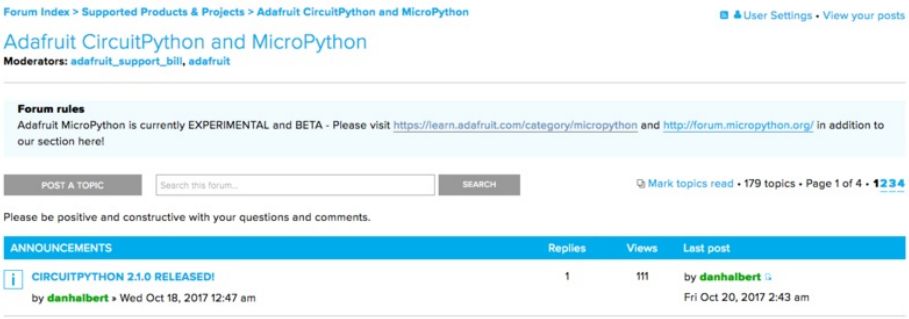
Adafruit Forums



The Adafruit Forums (https://adafru.it/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.

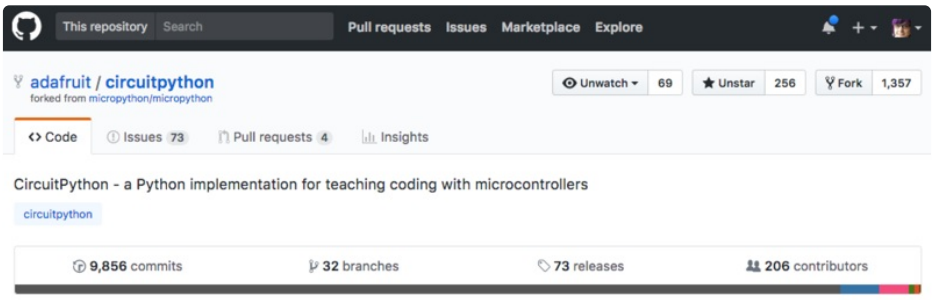
There are forum categories that cover all kinds of topics, including everything Adafruit. The Adafruit CircuitPython and MicroPython (https://adafru.it/xXA) category under "Supported Products & Projects" is the best place to post your CircuitPython questions.



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.

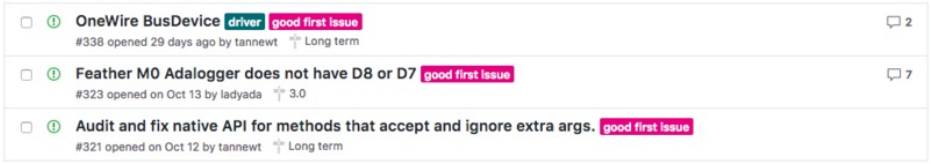
You might think you're just getting started, but you definitely know something that someone else doesn't. The great thing about the forums is that you can help others too! Everyone is welcome and encouraged to provide constructive feedback to any of the posted questions. This is an excellent way to contribute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



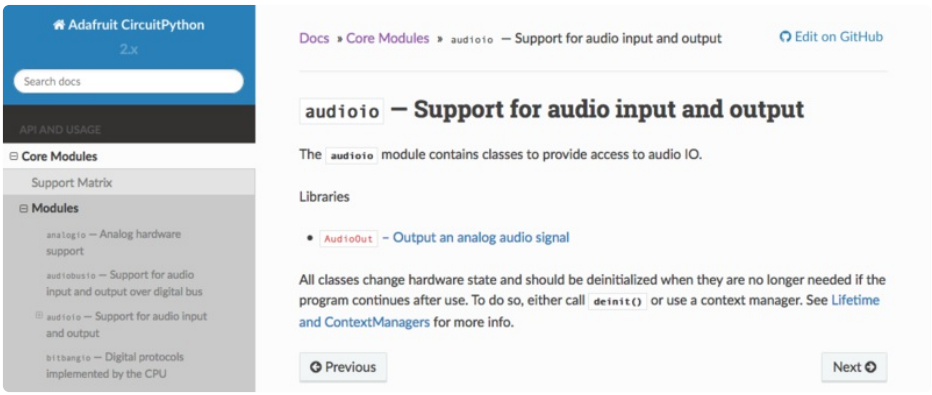
Already experienced and looking for a challenge? Checkout the rest of the issues list and you'll find plenty of ways to contribute. You'll find everything from new driver requests to core module updates. There's plenty of opportunities for everyone at any level!

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!

PyPortal CircuitPython Setup

To use all the amazing features of your PyPortal with CircuitPython, you must first install a number of libraries. This page covers that process.

Adafruit CircuitPython Bundle

Download the Adafruit CircuitPython Library Bundle. You can find the latest release here:

https://adafru.it/ENC

https://adafru.it/ENC

Download the **adafruit-circuitpython-bundle-*.x-mpy-*.zip** bundle zip file where ***.x MATCHES THE VERSION OF CIRCUITPYTHON YOU INSTALLED**, and unzip a folder of the same name. Inside you'll find a **lib** folder. You have two options:

- You can add the **lib** folder to your **CIRCUITPY** drive. This will ensure you have *all the drivers*. But it will take a bunch of space on the 8 MB disk
- Add each library as you need it, this will reduce the space usage but you'll need to put in a little more effort.

At a minimum we recommend the following libraries, in fact we more than recommend. They're basically required. So grab them and install them into **CIRCUITPY/lib** now!

- **adafruit_esp32spi** - This is the library that gives you internet access via the ESP32 using (you guessed it!) SPI transport. You need this for anything Internet
- **adafruit_requests** - This library allows us to perform HTTP requests and get responses back from servers. GET/POST/PUT/PATCH - they're all in here!
- **adafruit_pyportal** - This is our friendly wrapper library that does a lot of our projects, displays graphics and text, fetches data from the internet. Nearly all of our projects depend on it!
- **adafruit_portalbase** - This library is the base library that adafruit_pyportal library is built on top of.
- **adafruit_touchscreen** - a library for reading touches from the resistive touchscreen. Handles all the analog noodling, rotation and calibration for you.
- **adafruit_io** - this library helps connect the PyPortal to our free datalogging and viewing service
- **adafruit_imageload** - an image display helper, required for any graphics!
- **adafruit_display_text** - not surprisingly, it displays text on the screen
- **adafruit_bitmap_font** - we have fancy font support, and its easy to make new fonts. This library reads and parses font files.
- **adafruit_slideshow** - for making image slideshows - handy for quick display of graphics and sound
- **neopixel** - for controlling the onboard neopixel
- **adafruit_adt7410** - library to read the temperature from the on-board Analog Devices ADT7410 precision temperature sensor

- **adafruit_sdcard** - support for reading/writing data from the onboard SD card slot.
- **adafruit_bus_device** - low level support for I2C/SPI
- **adafruit_fakerequests** - This library allows you to create fake HTTP requests by using local files.

Internet Connect!

Once you have CircuitPython setup and libraries installed we can get your board connected to the Internet. Note that access to enterprise level secured WiFi networks is not currently supported, only WiFi networks that require SSID and password.

To get connected, you will need to start by creating a *secrets file*.

What's a secrets file?

We expect people to share tons of projects as they build CircuitPython WiFi widgets. What we want to avoid is people accidentally sharing their passwords or secret tokens and API keys. So, we designed all our examples to use a `secrets.py` file, that is in your **CIRCUITPY** drive, to hold secret/private/custom data. That way you can share your main project without worrying about accidentally sharing private stuff.

Your **secrets.py** file should look like this:

```
# This file is where you keep secret settings, passwords, and tokens!
# If you put them in the code you risk committing that info or sharing it

secrets = {
    'ssid' : 'home ssid',
    'password' : 'my password',
    'timezone' : "America/New_York", # http://worldtimeapi.org/timezones
    'github_token' : 'fawfj23rakjnfawiefa',
    'hackaday_token' : 'h4xx0rs3kret',
    }
```

Inside is a python dictionary named secrets with a line for each entry. Each entry has an entry name (say `'ssid'`) and then a colon to separate it from the entry key `'home ssid'` and finally a comma ,

At a minimum you'll need the `ssid` and `password` for your local WiFi setup. As you make projects you may need more tokens and keys, just add them one line at a time. See for example other tokens such as one for accessing github or the hackaday API. Other non-secret data like your timezone can also go here, just cause it's called secrets doesn't mean you can't have general customization data in there!

For the correct time zone string, look at http://worldtimeapi.org/timezones (https://adafru.it/EcP) and remember that if your city is not listed, look for a city in the same time zone, for example Boston, New York, Philadelphia, Washington DC, and Miami are all on the same time as New York.

Of course, don't share your **secrets.py** - keep that out of GitHub, Discord or other project-sharing sites.

Connect to WiFi

OK now you have your secrets setup - you can connect to the Internet. Lets use the ESP32SPI and the Requests libraries - [you'll need to visit the CircuitPython bundle and install](https://adafru.it/ENC):

- **adafruit_bus_device**
- **adafruit_esp32spi**
- **adafruit_requests**
- **neopixel**

Into your `lib` folder. Once that's done, load up the following example using Mu or your favorite editor:

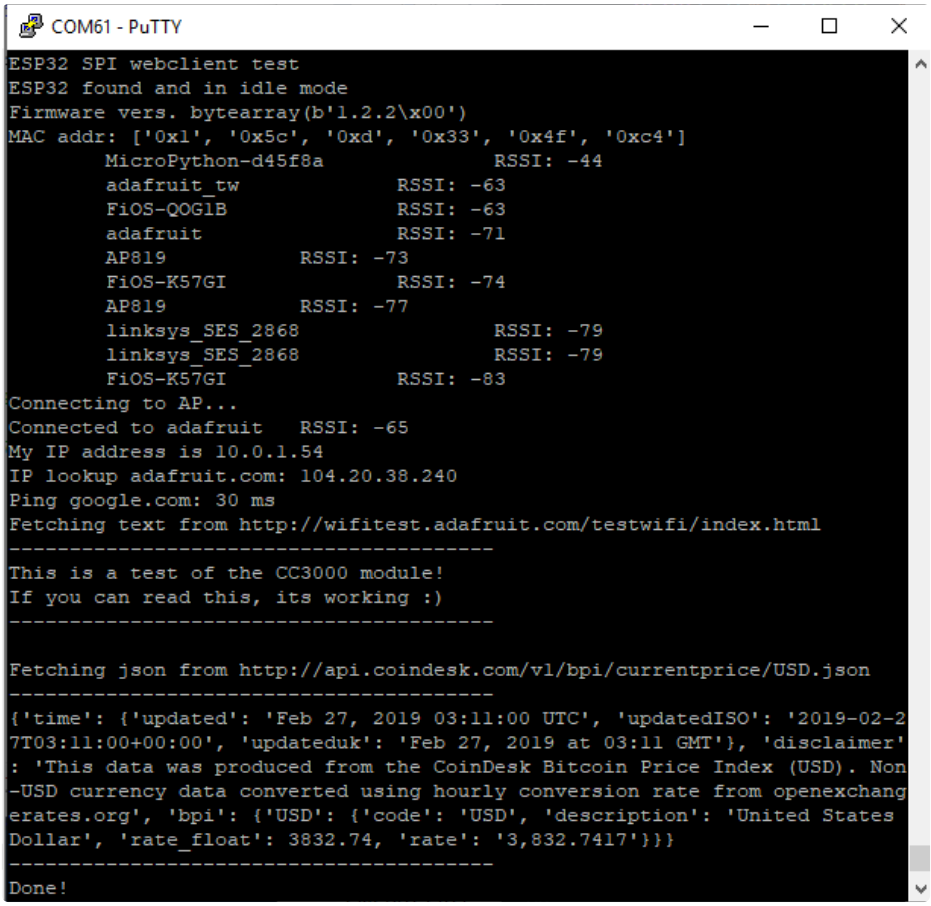
```
# SPDX-FileCopyrightText: 2019 ladyada for Adafruit Industries
# SPDX-License-Identifier: MIT

import board
import busio
from digitalio import DigitalInOut
import adafruit_requests as requests
import adafruit_esp32spi.adafruit_esp32spi_socket as socket
from adafruit_esp32spi import adafruit_esp32spi

# Get wifi details and more from a secrets.py file
try:
    from secrets import secrets
except ImportError:
    print("WiFi secrets are kept in secrets.py, please add them there!")
    raise

print("ESP32 SPI webclient test")

TEXT_URL = "http://wifitest.adafruit.com/testwifi/index.html"
JSON_URL = "http://api.coindesk.com/v1/bpi/currentprice/USD.json"


# If you are using a board with pre-defined ESP32 Pins:
esp32_cs = DigitalInOut(board.ESP_CS)
esp32_ready = DigitalInOut(board.ESP_BUSY)
esp32_reset = DigitalInOut(board.ESP_RESET)

# If you have an AirLift Shield:
# esp32_cs = DigitalInOut(board.D10)
# esp32_ready = DigitalInOut(board.D7)
# esp32_reset = DigitalInOut(board.D5)

# If you have an AirLift Featherwing or ItsyBitsy Airlift:
# esp32_cs = DigitalInOut(board.D13)
# esp32_ready = DigitalInOut(board.D11)
# esp32_reset = DigitalInOut(board.D12)

# If you have an externally connected ESP32:
# NOTE: You may need to change the pins to reflect your wiring
# esp32_cs = DigitalInOut(board.D9)
# esp32_ready = DigitalInOut(board.D10)
# esp32_reset = DigitalInOut(board.D5)
```

```
spi = busio.SPI(board.SCK, board.MOSI, board.MISO)
esp = adafruit_esp32spi.ESP_SPIcontrol(spi, esp32_cs, esp32_ready, esp32_reset)

requests.set_socket(socket, esp)

if esp.status == adafruit_esp32spi.WL_IDLE_STATUS:
    print("ESP32 found and in idle mode")
print("Firmware vers.", esp.firmware_version)
print("MAC addr:", [hex(i) for i in esp.MAC_address])

for ap in esp.scan_networks():
    print("\t%s\t\tRSSI: %d" % (str(ap["ssid"], "utf-8"), ap["rssi"]))

print("Connecting to AP...")
while not esp.is_connected:
    try:
        esp.connect_AP(secrets["ssid"], secrets["password"])
    except RuntimeError as e:
        print("could not connect to AP, retrying: ", e)
        continue
print("Connected to", str(esp.ssid, "utf-8"), "\tRSSI:", esp.rssi)
print("My IP address is", esp.pretty_ip(esp.ip_address))
print(
    "IP lookup adafruit.com: %s" % esp.pretty_ip(esp.get_host_by_name("adafruit.com"))
)
print("Ping google.com: %d ms" % esp.ping("google.com"))

# esp._debug = True
print("Fetching text from", TEXT_URL)
r = requests.get(TEXT_URL)
print("-" * 40)
print(r.text)
print("-" * 40)
r.close()

print()
print("Fetching json from", JSON_URL)
r = requests.get(JSON_URL)
print("-" * 40)
print(r.json())
print("-" * 40)
r.close()

print("Done!")
```

And save it to your board, with the name `code.py`.

Don't forget you'll also need to create the `secrets.py` file as seen above, with your WiFi ssid and password.

In a serial console, you should see something like the following. For more information about connecting with a serial console, view the guide Connecting to the Serial Console (https://adafru.it/Bec).



In order, the example code...

Initializes the ESP32 over SPI using the SPI port and 3 control pins:

```
esp32_cs = DigitalInOut(board.ESP_CS)
esp32_ready = DigitalInOut(board.ESP_BUSY)
esp32_reset = DigitalInOut(board.ESP_RESET)

spi = busio.SPI(board.SCK, board.MOSI, board.MISO)
esp = adafruit_esp32spi.ESP_SPIcontrol(spi, esp32_cs, esp32_ready, esp32_reset)
```

Tells our `requests` library the type of socket we're using (socket type varies by connectivity type - we'll be using the `adafruit_esp32spi_socket` for this example). We'll also set the interface to an `esp` object. This is a little bit of a hack, but it lets us use `requests` like CPython does.

```
requests.set_socket(socket, esp)
```

Verifies an ESP32 is found, checks the firmware and MAC address

```
if esp.status == adafruit_esp32spi.WL_IDLE_STATUS:
    print("ESP32 found and in idle mode")
print("Firmware vers.", esp.firmware_version)
print("MAC addr:", [hex(i) for i in esp.MAC_address])
```

Performs a scan of all access points it can see and prints out the name and signal strength:

```
for ap in esp.scan_networks():
    print("\t%s\t\tRSSI: %d" % (str(ap['ssid'], 'utf-8'), ap['rssi']))
```

Connects to the AP we've defined here, then prints out the local IP address, attempts to do a domain name lookup and ping google.com to check network connectivity (note sometimes the ping fails or takes a while, this isn't a big deal)

```
print("Connecting to AP...")
while not esp.is_connected:
    try:
        esp.connect_AP(secrets["ssid"], secrets["password"])
    except RuntimeError as e:
        print("could not connect to AP, retrying: ", e)
        continue
print("Connected to", str(esp.ssid, "utf-8"), "\tRSSI:", esp.rssi)
print("My IP address is", esp.pretty_ip(esp.ip_address))
print(
    "IP lookup adafruit.com: %s" % esp.pretty_ip(esp.get_host_by_name("adafruit.com"))
```

OK now we're getting to the really interesting part. With a SAMD51 or other large-RAM (well, over 32 KB) device, we can do a lot of neat tricks. Like for example we can implement an interface a lot like requests (https://adafru.it/E9o) - which makes getting data *really really easy*

To read in all the text from a web URL call `requests.get` - you can pass in `https` URLs for SSL connectivity

```
TEXT_URL = "http://wifitest.adafruit.com/testwifi/index.html"
print("Fetching text from", TEXT_URL)
r = requests.get(TEXT_URL)
print('-'*40)
print(r.text)
print('-'*40)
r.close()
```

Or, if the data is in structured JSON, you can get the json pre-parsed into a Python dictionary that can be easily queried or traversed. (Again, only for nRF52840, M4 and other high-RAM boards)

```
JSON_URL = "http://api.coindesk.com/v1/bpi/currentprice/USD.json"
print("Fetching json from", JSON_URL)
r = requests.get(JSON_URL)
print('-'*40)
print(r.json())
print('-'*40)
r.close()
```

Requests

We've written a requests-like (https://adafru.it/Kpa) library for web interfacing named Adafruit_CircuitPython_Requests (https://adafru.it/FpW). This library allows you to send HTTP/1.1 requests without "crafting" them and provides helpful methods for parsing the response from the server.

```
# SPDX-FileCopyrightText: 2021 ladyada for Adafruit Industries
# SPDX-License-Identifier: MIT

# adafruit_requests usage with an esp32spi_socket
import board
import busio
from digitalio import DigitalInOut
import adafruit_esp32spi.adafruit_esp32spi_socket as socket
from adafruit_esp32spi import adafruit_esp32spi
import adafruit_requests as requests

# Add a secrets.py to your filesystem that has a dictionary called secrets with "ssid" and
# "password" keys with your WiFi credentials. DO NOT share that file or commit it into Git or
other
# source control.
# pylint: disable=no-name-in-module,wrong-import-order
try:
    from secrets import secrets
except ImportError:
    print("WiFi secrets are kept in secrets.py, please add them there!")
    raise

# If you are using a board with pre-defined ESP32 Pins:
esp32_cs = DigitalInOut(board.ESP_CS)
esp32_ready = DigitalInOut(board.ESP_BUSY)
esp32_reset = DigitalInOut(board.ESP_RESET)

# If you have an externally connected ESP32:
# esp32_cs = DigitalInOut(board.D9)
# esp32_ready = DigitalInOut(board.D10)
# esp32_reset = DigitalInOut(board.D5)

# If you have an AirLift Featherwing or ItsyBitsy Airlift:
# esp32_cs = DigitalInOut(board.D13)
# esp32_ready = DigitalInOut(board.D11)
# esp32_reset = DigitalInOut(board.D12)

spi = busio.SPI(board.SCK, board.MOSI, board.MISO)
esp = adafruit_esp32spi.ESP_SPIcontrol(spi, esp32_cs, esp32_ready, esp32_reset)
```

```
print("Connecting to AP...")
while not esp.is_connected:
    try:
        esp.connect_AP(secrets["ssid"], secrets["password"])
    except RuntimeError as e:
        print("could not connect to AP, retrying: ", e)
        continue
print("Connected to", str(esp.ssid, "utf-8"), "\tRSSI:", esp.rssi)

# Initialize a requests object with a socket and esp32spi interface
socket.set_interface(esp)
requests.set_socket(socket, esp)

TEXT_URL = "http://wifitest.adafruit.com/testwifi/index.html"
JSON_GET_URL = "https://httpbin.org/get"
JSON_POST_URL = "https://httpbin.org/post"

print("Fetching text from %s" % TEXT_URL)
response = requests.get(TEXT_URL)
print("-" * 40)

print("Text Response: ", response.text)
print("-" * 40)
response.close()

print("Fetching JSON data from %s" % JSON_GET_URL)
response = requests.get(JSON_GET_URL)
print("-" * 40)

print("JSON Response: ", response.json())
print("-" * 40)
response.close()

data = "31F"
print("POSTing data to {0}: {1}".format(JSON_POST_URL, data))
response = requests.post(JSON_POST_URL, data=data)
print("-" * 40)

json_resp = response.json()
# Parse out the 'data' key from json_resp dict.
print("Data received from server:", json_resp["data"])
print("-" * 40)
response.close()

json_data = {"Date": "July 25, 2019"}
print("POSTing data to {0}: {1}".format(JSON_POST_URL, json_data))
response = requests.post(JSON_POST_URL, json=json_data)
print("-" * 40)

json_resp = response.json()
# Parse out the 'json' key from json_resp dict.
print("JSON Data received from server:", json_resp["json"])
print("-" * 40)
response.close()
```

The code first sets up the ESP32SPI interface. Then, it initializes a `request` object using an ESP32 `socket` and the `esp` object.

```
import board
import busio
from digitalio import DigitalInOut
import adafruit_esp32spi.adafruit_esp32spi_socket as socket
from adafruit_esp32spi import adafruit_esp32spi
import adafruit_requests as requests

# If you are using a board with pre-defined ESP32 Pins:
esp32_cs = DigitalInOut(board.ESP_CS)
esp32_ready = DigitalInOut(board.ESP_BUSY)
esp32_reset = DigitalInOut(board.ESP_RESET)

# If you have an externally connected ESP32:
# esp32_cs = DigitalInOut(board.D9)
# esp32_ready = DigitalInOut(board.D10)
# esp32_reset = DigitalInOut(board.D5)

spi = busio.SPI(board.SCK, board.MOSI, board.MISO)
esp = adafruit_esp32spi.ESP_SPIcontrol(spi, esp32_cs, esp32_ready, esp32_reset)

print("Connecting to AP...")
while not esp.is_connected:
    try:
        esp.connect_AP(b'MY_SSID_NAME', b'MY_SSID_PASSWORD')
    except RuntimeError as e:
        print("could not connect to AP, retrying: ",e)
        continue
print("Connected to", str(esp.ssid, 'utf-8'), "\tRSSI:", esp.rssi)

# Initialize a requests object with a socket and esp32spi interface
requests.set_socket(socket, esp)
```

HTTP GET with Requests

The code makes a HTTP GET request to Adafruit's WiFi testing website - http://wifitest.adafruit.com/testwifi/index.html (https://adafru.it/FpZ).

To do this, we'll pass the URL into `requests.get()`. We're also going to save the response *from* the server into a variable named `response`.

While we requested data from the server, we'd what the server responded with. Since we already saved the server's `response`, we can read it back. Luckily for us, **requests automatically decodes the server's response into human-readable text,** you can read it back by calling `response.text`.

Lastly, we'll perform a bit of cleanup by calling `response.close()`. This closes, deletes, and collect's the response's data.

```
print("Fetching text from %s"%TEXT_URL)
response = requests.get(TEXT_URL)
print('-'*40)

print("Text Response: ", response.text)
print('-'*40)
response.close()
```

While some servers respond with text, some respond with json-formatted data consisting of attribute–value pairs.

CircuitPython_Requests can convert a JSON-formatted response from a server into a CPython `dict.` object.

We can also fetch and parse **json** data. We'll send a HTTP get to a url we know returns a json-formatted response (instead of text data).

Then, the code calls `response.json()` to convert the response to a CPython `dict`.

```
print("Fetching JSON data from %s"%JSON_GET_URL)
response = requests.get(JSON_GET_URL)
print('-'*40)

print("JSON Response: ", response.json())
print('-'*40)
response.close()
```

HTTP POST with Requests

Requests can also **POST data to a server by calling the `requests.post` method, passing it a `data` value.**

```
data = '31F'
print("POSTing data to {0}: {1}".format(JSON_POST_URL, data))
response = requests.post(JSON_POST_URL, data=data)
print('-'*40)

json_resp = response.json()
# Parse out the 'data' key from json_resp dict.
print("Data received from server:", json_resp['data'])
print('-'*40)
response.close()
```

You can also post json-formatted data to a server by passing `json_data` into the `requests.post` method.

```
    json_data = {"Date" : "July 25, 2019"}
print("POSTing data to {0}: {1}".format(JSON_POST_URL, json_data))
response = requests.post(JSON_POST_URL, json=json_data)
print('-'*40)

json_resp = response.json()
# Parse out the 'json' key from json_resp dict.
print("JSON Data received from server:", json_resp['json'])
print('-'*40)
response.close()
```

Advanced Requests Usage

Want to send custom HTTP headers, parse the response as raw bytes, or handle a response's http status code in your CircuitPython code?

We've written an example to show advanced usage of the requests module below.

```
# SPDX-FileCopyrightText: 2021 ladyada for Adafruit Industries
# SPDX-License-Identifier: MIT

import board
import busio
from digitalio import DigitalInOut
import adafruit_esp32spi.adafruit_esp32spi_socket as socket
from adafruit_esp32spi import adafruit_esp32spi
import adafruit_requests as requests

# Add a secrets.py to your filesystem that has a dictionary called secrets with "ssid" and
# "password" keys with your WiFi credentials. DO NOT share that file or commit it into Git or
other
# source control.
# pylint: disable=no-name-in-module,wrong-import-order
try:
    from secrets import secrets
except ImportError:
    print("WiFi secrets are kept in secrets.py, please add them there!")
    raise

# If you are using a board with pre-defined ESP32 Pins:
esp32_cs = DigitalInOut(board.ESP_CS)
esp32_ready = DigitalInOut(board.ESP_BUSY)
esp32_reset = DigitalInOut(board.ESP_RESET)

# If you have an externally connected ESP32:
# esp32_cs = DigitalInOut(board.D9)
# esp32_ready = DigitalInOut(board.D10)
# esp32_reset = DigitalInOut(board.D5)

spi = busio.SPI(board.SCK, board.MOSI, board.MISO)
esp = adafruit_esp32spi.ESP_SPIcontrol(spi, esp32_cs, esp32_ready, esp32_reset)
```

```
print("Connecting to AP...")
while not esp.is_connected:
    try:
        esp.connect_AP(secrets["ssid"], secrets["password"])
    except RuntimeError as e:
        print("could not connect to AP, retrying: ", e)
        continue
print("Connected to", str(esp.ssid, "utf-8"), "\tRSSI:", esp.rssi)

# Initialize a requests object with a socket and esp32spi interface
socket.set_interface(esp)
requests.set_socket(socket, esp)

JSON_GET_URL = "http://httpbin.org/get"

# Define a custom header as a dict.
headers = {"user-agent": "blinka/1.0.0"}

print("Fetching JSON data from %s..." % JSON_GET_URL)
response = requests.get(JSON_GET_URL, headers=headers)
print("-" * 60)

json_data = response.json()
headers = json_data["headers"]
print("Response's Custom User-Agent Header: {0}".format(headers["User-Agent"]))
print("-" * 60)

# Read Response's HTTP status code
print("Response HTTP Status Code: ", response.status_code)
print("-" * 60)

# Close, delete and collect the response data
response.close()
```

WiFi Manager

That simpletest example works but it's a little finicky - you need to constantly check WiFi status and have many loops to manage connections and disconnections. For more advanced uses, we recommend using the WiFiManager object. It will wrap the connection/status/requests loop for you - reconnecting if WiFi drops, resetting the ESP32 if it gets into a bad state, etc.

Here's a more advanced example that shows the WiFi manager and also how to POST data with some extra headers:

```
# SPDX-FileCopyrightText: 2019 ladyada for Adafruit Industries
# SPDX-License-Identifier: MIT

import time
import board
import busio
from digitalio import DigitalInOut
import neopixel
from adafruit_esp32spi import adafruit_esp32spi
```

```
from adafruit_esp32spi import adafruit_esp32spi_wifimanager

print("ESP32 SPI webclient test")

# Get wifi details and more from a secrets.py file
try:
    from secrets import secrets
except ImportError:
    print("WiFi secrets are kept in secrets.py, please add them there!")
    raise

# If you are using a board with pre-defined ESP32 Pins:
esp32_cs = DigitalInOut(board.ESP_CS)
esp32_ready = DigitalInOut(board.ESP_BUSY)
esp32_reset = DigitalInOut(board.ESP_RESET)

# If you have an externally connected ESP32:
# esp32_cs = DigitalInOut(board.D9)
# esp32_ready = DigitalInOut(board.D10)
# esp32_reset = DigitalInOut(board.D5)

spi = busio.SPI(board.SCK, board.MOSI, board.MISO)
esp = adafruit_esp32spi.ESP_SPIcontrol(spi, esp32_cs, esp32_ready, esp32_reset)
"""Use below for Most Boards"""
status_light = neopixel.NeoPixel(
    board.NEOPIXEL, 1, brightness=0.2
)  # Uncomment for Most Boards
"""Uncomment below for ItsyBitsy M4"""
# status_light = dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI, 1, brightness=0.2)
# Uncomment below for an externally defined RGB LED
# import adafruit_rgbled
# from adafruit_esp32spi import PWMOut
# RED_LED = PWMOut.PWMOut(esp, 26)
# GREEN_LED = PWMOut.PWMOut(esp, 27)
# BLUE_LED = PWMOut.PWMOut(esp, 25)
# status_light = adafruit_rgbled.RGBLED(RED_LED, BLUE_LED, GREEN_LED)
wifi = adafruit_esp32spi_wifimanager.ESPSPI_WiFiManager(esp, secrets, status_light)

counter = 0

while True:
    try:
        print("Posting data...", end="")
        data = counter
        feed = "test"
        payload = {"value": data}
        response = wifi.post(
            "https://io.adafruit.com/api/v2/"
            + secrets["aio_username"]
            + "/feeds/"
            + feed
            + "/data",
            json=payload,
            headers={"X-AIO-KEY": secrets["aio_key"]},
        )
        print(response.json())
        response.close()
```

```
        counter = counter + 1
        print("OK")
    except (ValueError, RuntimeError) as e:
        print("Failed to get data, retrying\n", e)
        wifi.reset()
        continue
    response = None
    time.sleep(15)
```

You'll note here we use a secrets.py file to manage our SSID info. The wifimanager is given the ESP32 object, secrets and a neopixel for status indication.

Note, you'll need to add a some additional information to your secrets file so that the code can query the Adafruit IO API:

- `aio_username`
- `aio_key`

You can go to your adafruit.io View AIO Key link to get those two values and add them to the secrets file, which will now look something like this:

```
# This file is where you keep secret settings, passwords, and tokens!
# If you put them in the code you risk committing that info or sharing it

secrets = {
    'ssid' : '_your_ssid_',
    'password' : '_your_wifi_password_',
    'timezone' : "America/Los_Angeles", # http://worldtimeapi.org/timezones
    'aio_username' : '_your_aio_username_',
    'aio_key' : '_your_aio_key_',
    }
```

Next, **set up an Adafruit IO feed named `test`**

- If you do not know how to set up a feed, follow this page and come back when you've set up a feed named `test`. (https://adafru.it/f5k)

We can then have a simple loop for posting data to Adafruit IO without having to deal with connecting or initializing the hardware!

Take a look at your **test** feed on Adafruit.io and you'll see the value increase each time the CircuitPython board posts data to it!


/Feeds/test
test
30
25
20
15
10
5
0
test
+ Add Data
Download All Data
Filter
Prev First
page 1 of 1
Next
Created at
Value
Location
2019/02/27 1:38:37pm 30
2019/02/27 1:38:17pm 29
2019/02/27 1:37:55pm 28
2019/02/27 1:37:33pm 27
2019/02/27 1:37:11pm 26

CircuitPython BLE

CircuitPython BLE UART Example

It's easy to use Adafruit AirLift ESP32 co-processor boards for Bluetooth Low Energy (BLE) with CircuitPython. When you reset the ESP32, you can put it in WiFi mode (the default), or in BLE mode; you cannot use both modes simultaenously.

Here's a simple example of using BLE to connect CircuitPython with the Bluefruit Connect app. Use CircuitPython 6.0.0 or later.

Note: Don't confuse the **ESP32** with the **ESP32-S2**, which is a different module with a similar name. The ESP32-S2 does not support BLE.

Currently the AirLift support for CircuitPython only provides BLE peripheral support. BLE central is under development. So you cannot connect to BLE devices like Heart Rate monitors, etc., but you can act as a BLE peripheral yourself.

On-Board Airlift Co-Processor - No Wiring Needed

If you have an **Adafruit Metro M4 AirLift Lite**, an **Adafruit PyPortal** (regular, **Pynt** or **Titano**), an **Adafruit MatrixPortal**, or other Adafruit board with an onboard ESP32 co-processor, then everything is prewired for you, and the pins you need to use are predefined in CircuitPython.

Update the AirLift Firmware

You will need to update the AirLift's firmware to at least version 1.7.1. **Previous versions of the AirLift firmware do not support BLE.**

Follow the instructions in the guide below, and come back to this page when you've upgraded the AirLift's firmware:

https://adafru.it/FWs

https://adafru.it/FWs

Ensure the AirLift firmware is version 1.7.1 or higher for BLE to work.

Install CircuitPython Libraries

Make sure you are running the latest version of Adafruit CircuitPython (https://adafru.it/Amd) for your board; you'll need 6.0.0 or later.

Next you'll need to install the necessary libraries to use the hardware and BLE. Carefully follow the steps to find and install these libraries from Adafruit's CircuitPython library bundle (https://adafru.it/uap). Our CircuitPython starter guide has a great page on how to use the library bundle (https://adafru.it/ABU).

Install these libraries from the bundle:

- **adafruit_airlift**
- **adafruit_ble**

Before continuing make sure your board's **lib** folder or root filesystem has the **adafruit_airlift** and **adafruit_ble** folders copied over.

Install the Adafruit Bluefruit LE Connect App

The Adafruit Bluefruit LE Connect iOS and Android apps allow you to connect to BLE peripherals that provide a over-the-air "UART" service. Follow the instructions in the Bluefruit LE Connect Guide (https://adafru.it/Eg5) to download and install the app on your phone or tablet.

Copy and Adjust the Example Program

Copy the program below to the file **code.py** on **CIRCUITPY** on your board.

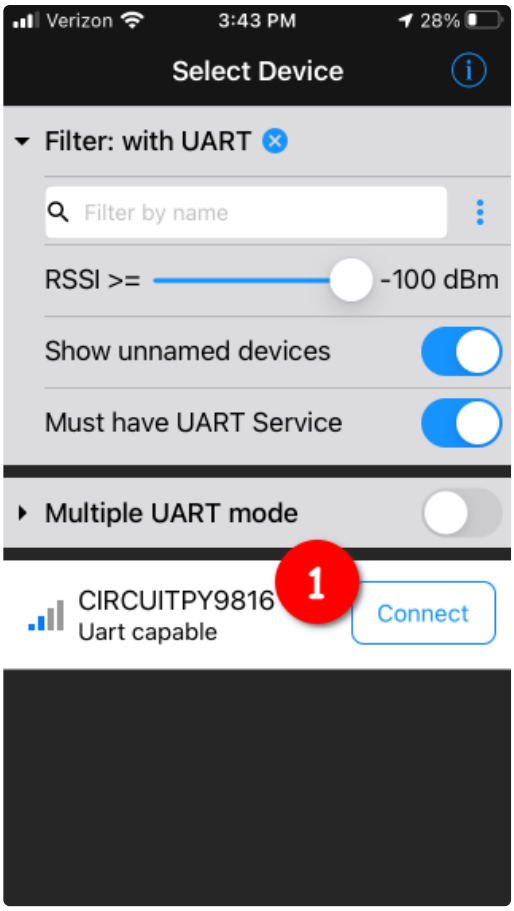
TAKE NOTE: Adjust the program as needed to suit the AirLift board you have. Comment and uncomment lines 12-39 below as necessary.

```
import board

from adafruit_ble import BLERadio
from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
from adafruit_ble.services.nordic import UARTService

from adafruit_airlift.esp32 import ESP32

# If you are using a Metro M4 Airlift Lite, PyPortal,
# or MatrixPortal, you can use the default pin settings.
# Leave this DEFAULT line uncommented.
esp32 = ESP32() # DEFAULT

# If you are using CircuitPython 6.0.0 or earlier,
# on PyPortal and PyPortal Titano only, use the pin settings
# below. Comment out the DEFAULT line above and uncomment
# the line below. For CircuitPython 6.1.0, the pin names
# have changed for these boards, and the DEFAULT line
# above is correct.
# esp32 = ESP32(tx_board_TX, rx_board_RX)
```

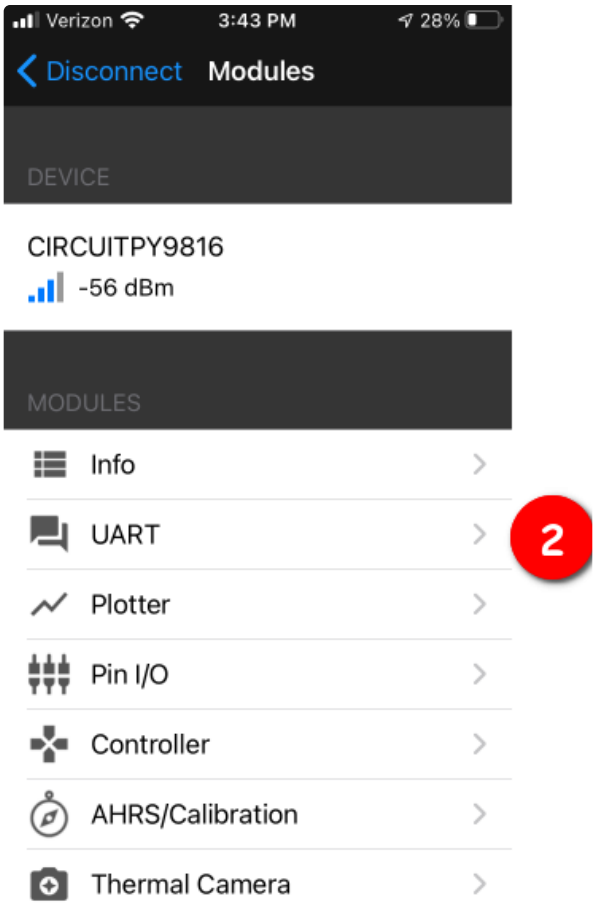
```
# esp32 = ESP32(tx=board.TX, rx=board.RX)

# If you are using an AirLift FeatherWing or AirLift Bitsy Add-On,
# use the pin settings below. Comment out the DEFAULT line above
# and uncomment the lines below.
# If you are using an AirLift Breakout, check that these
# choices match the wiring to your microcontroller board,
# or change them as appropriate.
# esp32 = ESP32(
#     reset=board.D12,
#     gpio0=board.D10,
#     busy=board.D11,
#     chip_select=board.D13,
#     tx=board.TX,
#     rx=board.RX,
# )

# If you are using an AirLift Shield,
# use the pin settings below. Comment out the DEFAULT line above
# and uncomment the lines below.
# esp32 = ESP32(
#     reset=board.D5,
#     gpio0=board.D6,
#     busy=board.D7,
#     chip_select=board.D10,
#     tx=board.TX,
#     rx=board.RX,
# )

adapter = esp32.start_bluetooth()

ble = BLERadio(adapter)
uart = UARTService()
advertisement = ProvideServicesAdvertisement(uart)

while True:
    ble.start_advertising(advertisement)
    print("waiting to connect")
    while not ble.connected:
        pass
    print("connected: trying to read input")
    while ble.connected:
        # Returns b'' if nothing was read.
        one_byte = uart.read(1)
        if one_byte:
            print(one_byte)
            uart.write(one_byte)
```

Talk to the AirLift via the Bluefruit LE Connect App

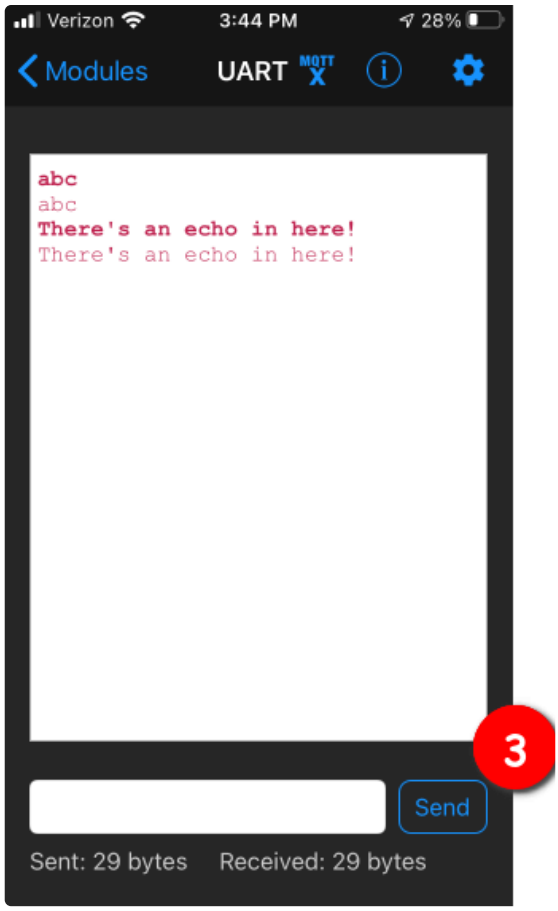
Start the Bluefruit LE Connect App on your phone or tablet. You should see a CIRCUITPY device available to connect to. Tap the Connect button (1):



You'll then see a list of Bluefruit Connect functions ("modules"). Choose the UART module (2):



On the UART module page, you can type a string and press Send (3). You'll see that string entered, and then see it echoed back (echoing is in gray).


Verizon
3:44 PM
28%
Modules
UART
MQTT
abc
abc
There's an echo in here!
There's an echo in here!
3
Send
Sent: 29 bytes
Received: 29 bytes

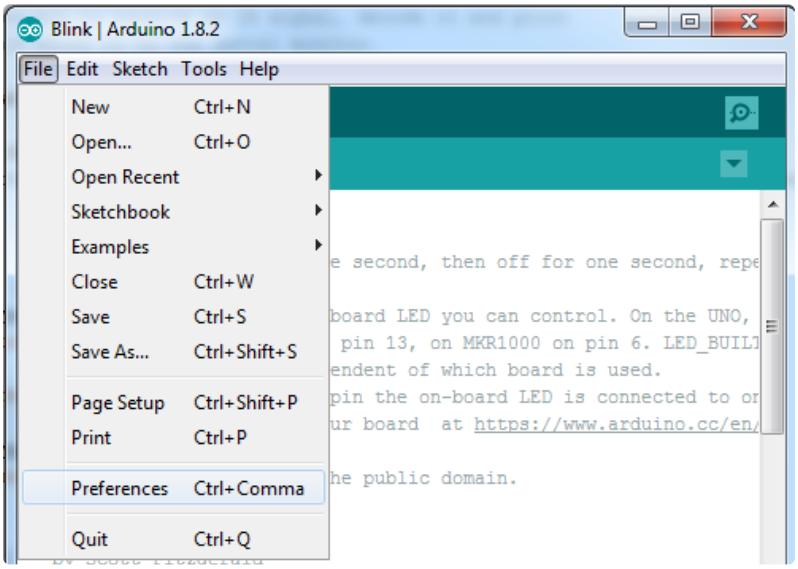
Arduino IDE Setup

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

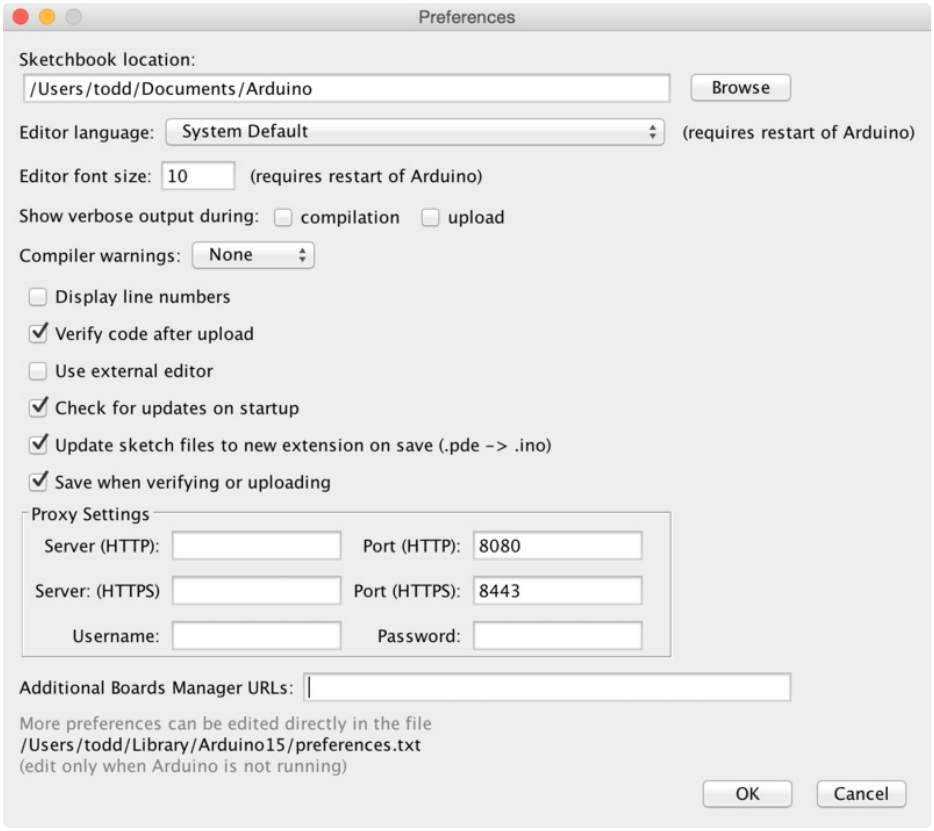
https://adafru.it/f1P

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*.



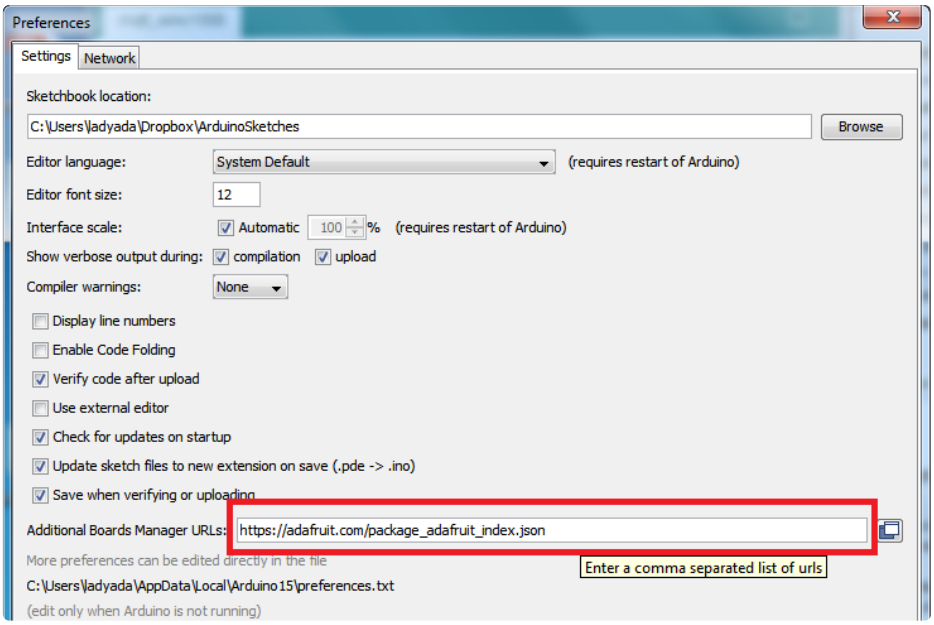
A dialog will pop up just like the one shown below.



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 (,)

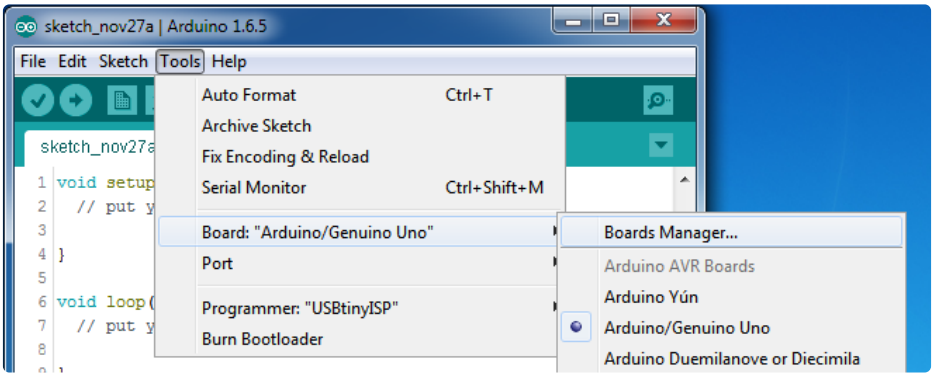
Once done click **OK** to save the new preference settings. Next we will look at installing boards with the Board Manager.

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

Using with Arduino IDE

The Feather/Metro/Gemma/QTPy/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.



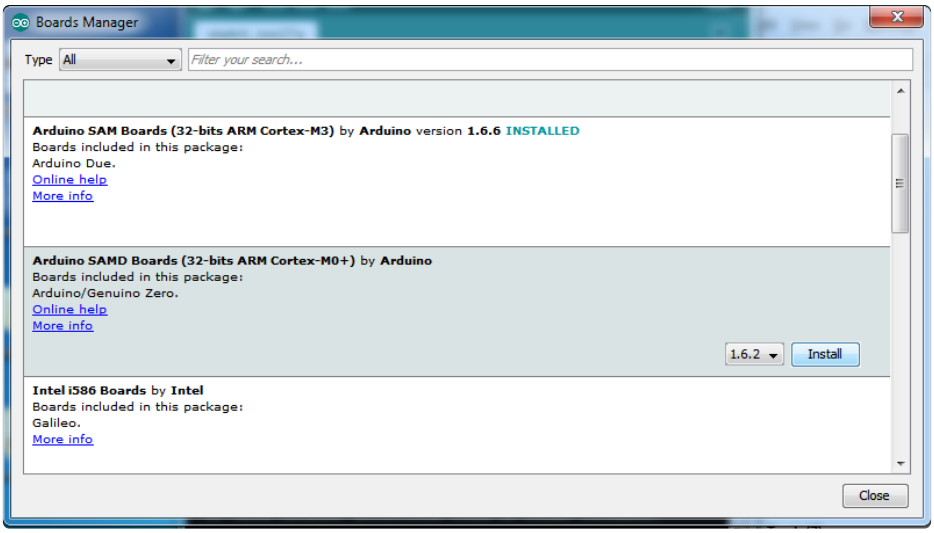
Once the Board Manager opens, click on the category drop down menu on the top left hand side of the window and select **All**. You will then be able to select and install the boards supplied by the URLs added to the preferences.

> 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**

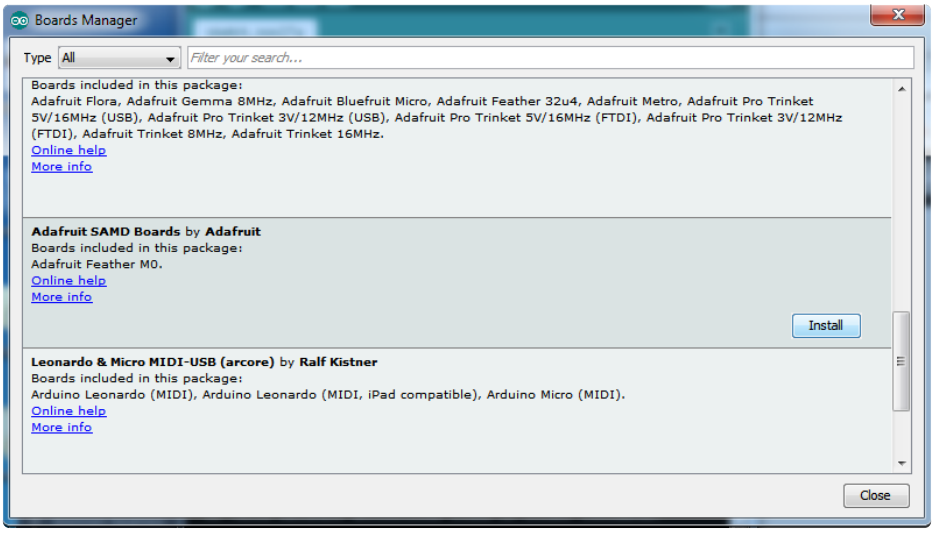


Install Adafruit SAMD

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**



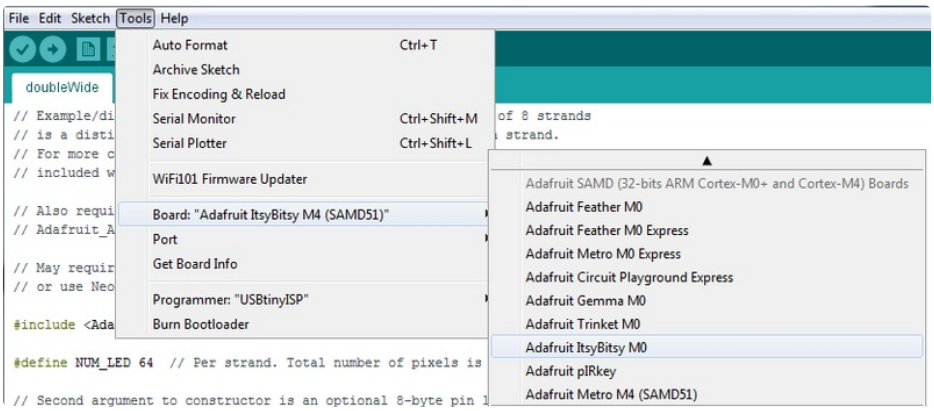
Even though in theory you don't need to - I recommend rebooting the IDE

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
- QT Py 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
- Grand Central M4 Express
- ItsyBitsy M4 Express
- Feather M4 Express
- Trellis M4 Express
- PyPortal M4
- PyPortal M4 Titano
- PyBadge M4 Express
- Metro M4 Airlift Lite
- PyGamer M4 Express
- MONSTER M4SK
- Hallowing M4
- MatrixPortal M4
- BLM Badge



Install Drivers (Windows 7 & 8 Only)

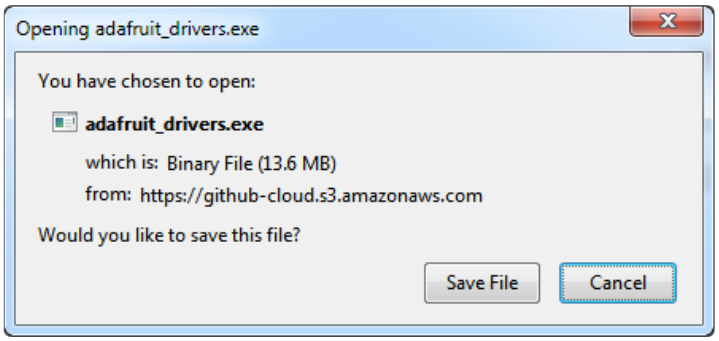
When you plug in the board, you'll need to possibly install a driver

Click below to download our Driver Installer

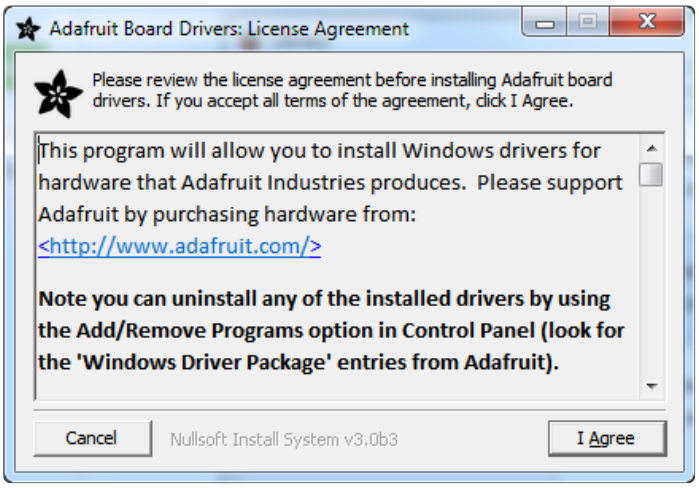
https://adafru.it/EC0

https://adafru.it/EC0

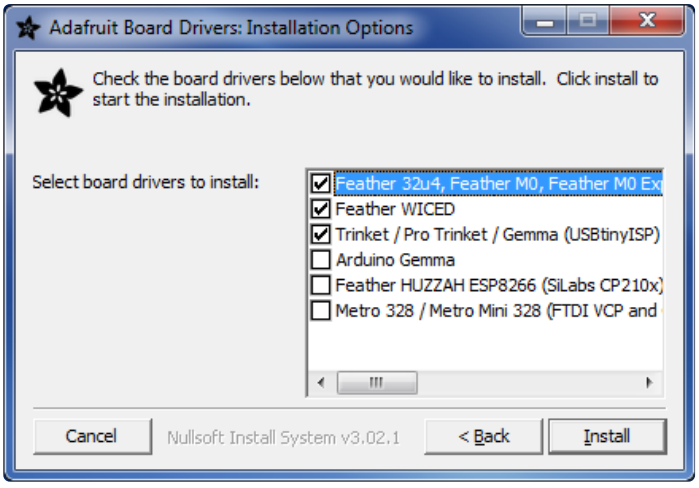
Download and run the installer



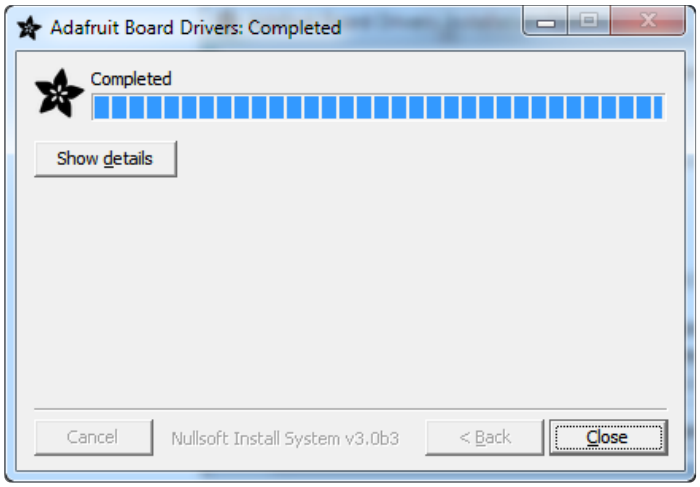
Run the installer! Since we bundle the SiLabs and FTDI drivers as well, you'll need to click through the license



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'

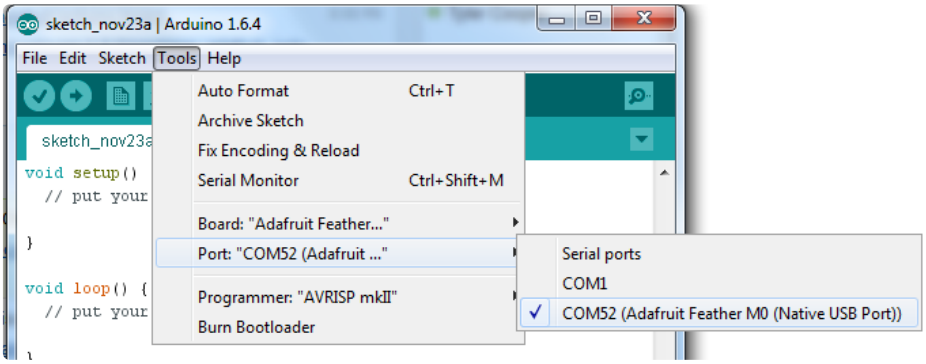


Blink

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!

Please note, the **QT Py and Trellis M4 Express** are two of our very few boards that does not have an onboard pin 13 LED so you can follow this section to practice uploading but you wont see an LED blink!



Now load up the Blink example

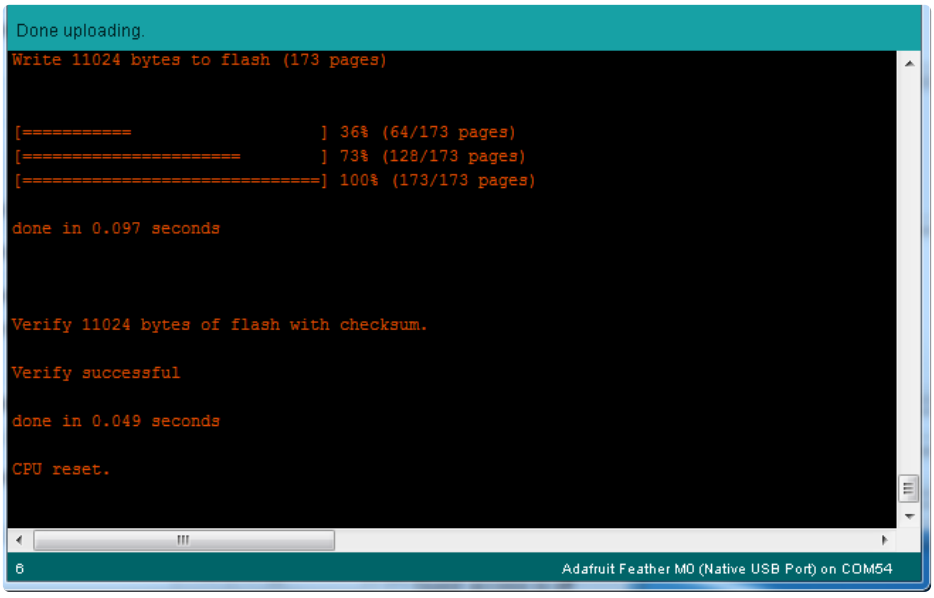
```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

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

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

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

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



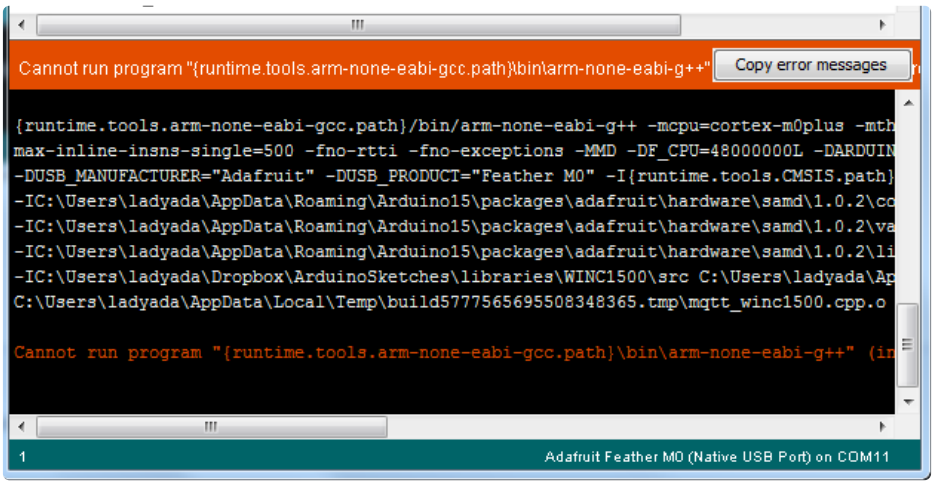
After uploading, you may see a message saying "Disk Not Ejected Properly" about the ...BOOT drive. You can ignore that message: it's an artifact of how the bootloader and uploading work.

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

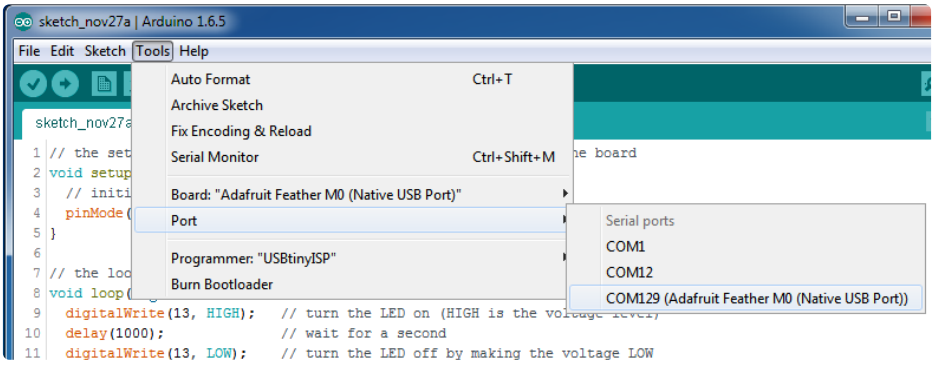


Manually bootloading

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 and/or RGB LED will be green, 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)

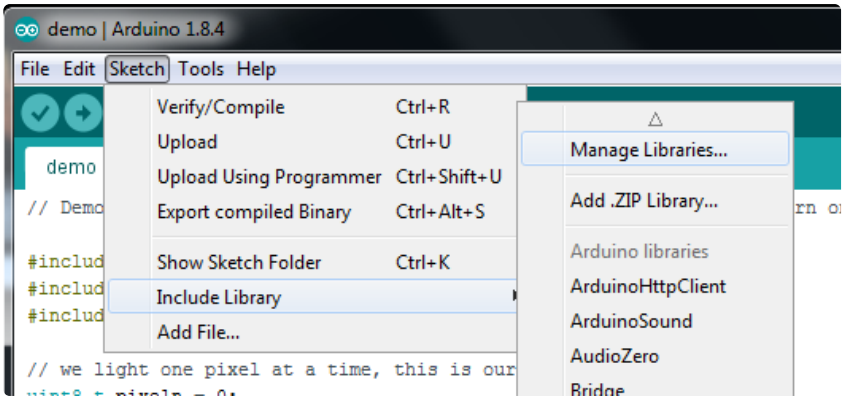
Arduino Libraries

OK now that you have Arduino IDE set up, drivers installed if necessary and you've practiced uploading code, you can start installing all the Libraries we'll be using to program it.

There's a lot of libraries!

Install Libraries

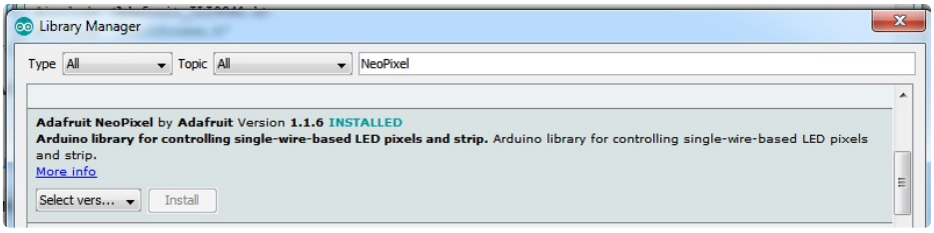
Open up the library manager...



And install the following libraries:

Adafruit NeoPixel

This will let you light up the status LED on the back

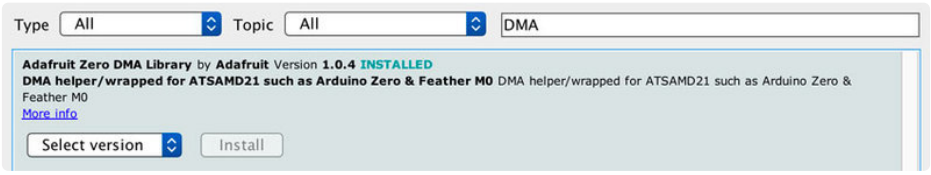


Adafruit SPIFlash

This will let you read/write to the onboard FLASH memory with super-fast QSPI support

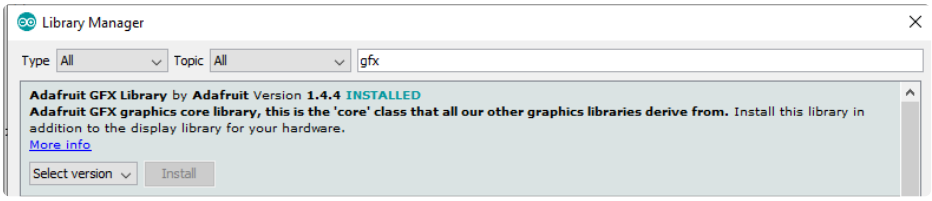
Adafruit Zero DMA

This is used by the Graphics Library if you choose to use DMA



Adafruit GFX

This is the graphics library used to draw to the screen



If using an older (pre-1.8.10) Arduino IDE, locate and install **Adafruit_BusIO** (newer versions do this automatically when installing Adafruit_GFX).

Adafruit ILI9341

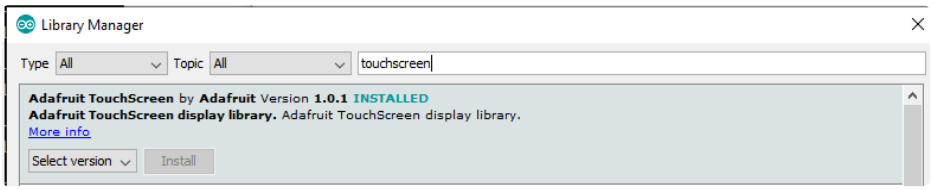
The display on the PyPortal!

Adafruit HX8357

The display on the PyPortal Titano!

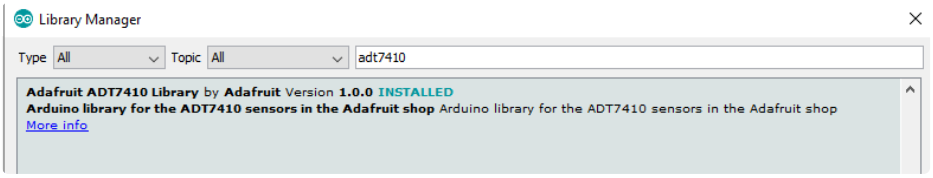
Adafruit Touchscreen

For reading touchscreen points on the resistive touchscreen



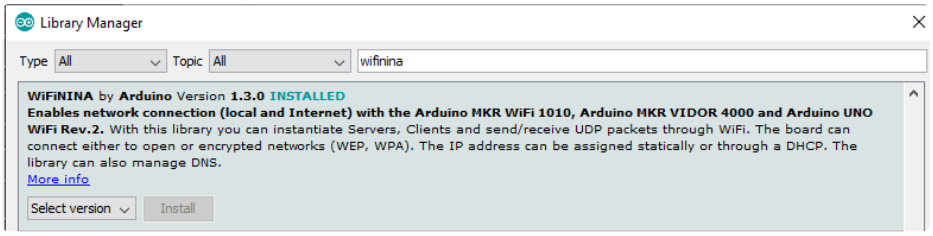
Analog Devices ADT7410

For reading temperature data from the onboard ADT7410



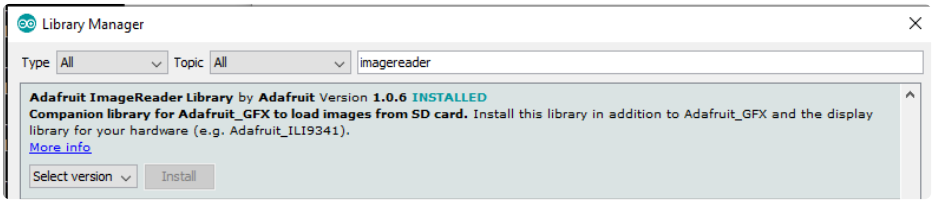
WiFiNINA

Will talk to the ESP32 WiFi co-processor to connect to the internet!



Adafruit ImageReader

For reading bitmaps from SD and displaying



Arduino Test

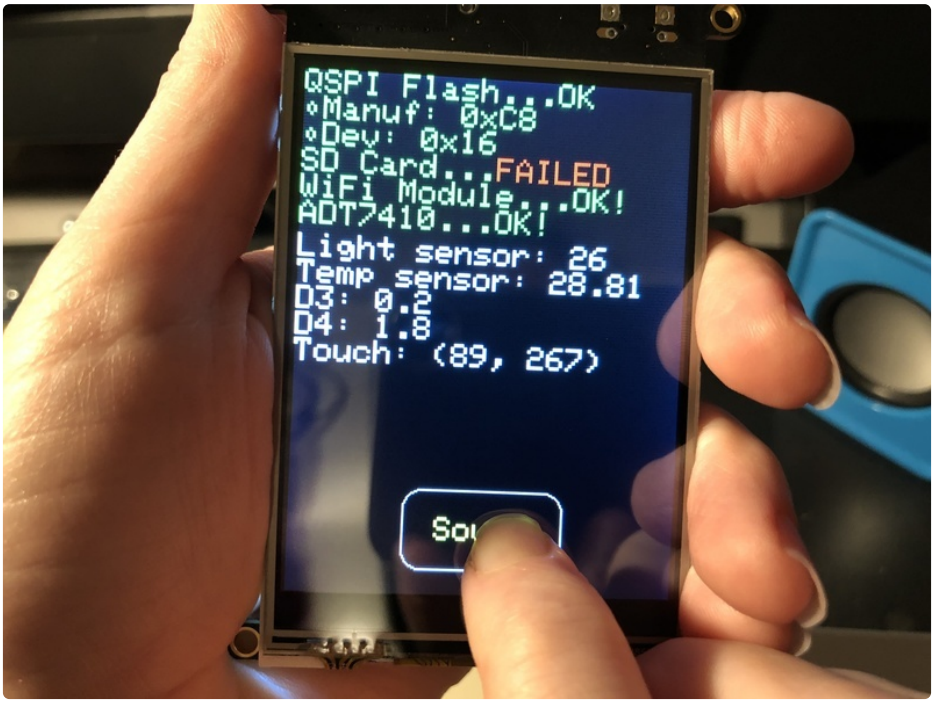
Once you've got the IDE installed and libraries in place you can run our test sketch. This will check all the hardware, and display it on the screen, its sort of a universal test because every part is checked. It's also a great reference if you want to know how to read the light sensor or initialize the touch screen.

It's normal to get `SD Card...Failed` if there is no SD card in the socket.

The light sensor value ranges from 0 (dark) to 1023 (bright)

The temperature sensor will heat up if the backlight is on for a while, that's also normal! To avoid self-heating turn or or lower down the backlight brightness

`D3` and `D4` measure the analog voltages on the 3 pin JST connectors. They'll be floating until some voltage is applied to them.



Click download project zip in the code window below, so you get both the arduino code and the coin.h (sound file) include

```
// This program does a test of all the hardware so you can get an example of how to read
// sensors, touchscreen, and display stuff!

#include "SPI.h"
#include "Adafruit_GFX.h"
#include "Adafruit_ILI9341.h"
#include <Adafruit_SPIFlash.h>
```

```
#include "Adafruit_ADT7410.h"
#include "TouchScreen.h"
#include <SdFat.h>
#include <WiFiNINA.h>
#include "coin.h"

#define RED_LED       13
#define TFT_RESET     24
#define TFT_BACKLIGHT 25
#define LIGHT_SENSOR  A2
#define SD_CS         32
#define SPKR_SHUTDOWN 50

#define TFT_D0        34 // Data bit 0 pin (MUST be on PORT byte boundary)
#define TFT_WR        26 // Write-strobe pin (CCL-inverted timer output)
#define TFT_DC        10 // Data/command pin
#define TFT_CS        11 // Chip-select pin
#define TFT_RST       24 // Reset pin
#define TFT_RD         9 // Read-strobe pin
#define TFT_BACKLIGHT 25
// ILI9341 with 8-bit parallel interface:
Adafruit_ILI9341 tft = Adafruit_ILI9341(tft8bitbus, TFT_D0, TFT_WR, TFT_DC, TFT_CS, TFT_RST,
TFT_RD);

Adafruit_FlashTransport_QSPI flashTransport(PIN_QSPI_SCK, PIN_QSPI_CS, PIN_QSPI_IO0,
PIN_QSPI_IO1, PIN_QSPI_IO2, PIN_QSPI_IO3);
Adafruit_SPIFlash flash(&flashTransport);

Adafruit_ADT7410 tempsensor = Adafruit_ADT7410();

#define YP A4  // must be an analog pin, use "An" notation!
#define XM A7  // must be an analog pin, use "An" notation!
#define YM A6   // can be a digital pin
#define XP A5   // can be a digital pin
TouchScreen ts = TouchScreen(XP, YP, XM, YM, 300);
#define X_MIN  750
#define X_MAX  325
#define Y_MIN  840
#define Y_MAX  240

Adafruit_GFX_Button coin = Adafruit_GFX_Button();
SdFat SD;

void setup() {
  Serial.begin(115200);
  //while (!Serial);

  Serial.println("All Test!");

  pinMode(RED_LED, OUTPUT);
  pinMode(TFT_BACKLIGHT, OUTPUT);
  digitalWrite(TFT_BACKLIGHT, HIGH);

  pinMode(TFT_RESET, OUTPUT);
  digitalWrite(TFT_RESET, HIGH);
  delay(10);
  digitalWrite(TFT_RESET, LOW);
```

```
  delay(10);
  digitalWrite(TFT_RESET, HIGH);
  delay(10);

  tft.begin();

  tft.fillScreen(ILI9341_BLACK);
  tft.setTextSize(2);
  tft.setTextColor(ILI9341_GREEN);
  tft.setTextWrap(true);
  tft.setCursor(0, 0);

  tft.print("QSPI Flash...");
  if (!flash.begin()){
    Serial.println("Could not find flash on QSPI bus!");
    tft.setTextColor(ILI9341_RED);
    tft.println("FAILED");
    while (1);
  }
  Serial.println("Reading QSPI ID");
  Serial.print("JEDEC ID: 0x"); Serial.println(flash.getJEDECID(), HEX);
  tft.setTextColor(ILI9341_GREEN);
  tft.print("QSPI Flash JEDEC 0x"); tft.println(flash.getJEDECID(), HEX);

  /*************** SD CARD */
  tft.setCursor(0, 48);
  tft.print("SD Card...");
  if (!SD.begin(SD_CS)) {
    Serial.println("Card init. failed!");
    tft.setTextColor(ILI9341_RED);
    tft.println("FAILED");
    tft.setTextColor(ILI9341_GREEN);
  } else {
    tft.println("OK!");
  }

  /*************** WiFi Module */

  tft.setCursor(0, 64);
  tft.print("WiFi Module...");
  WiFi.status();
  delay(100);
  if (WiFi.status() == WL_NO_MODULE) {
    Serial.println("ESP32 SPI not found");
    tft.setTextColor(ILI9341_RED);
    tft.println("FAILED");
    tft.setTextColor(ILI9341_GREEN);
  } else {
    Serial.println("ESP32 SPI mode found");
    tft.println("OK!");
  }

   /*************** Temperature sensor */
   tft.setCursor(0, 80);
   tft.print("ADT7410...");
   if (!tempsensor.begin()) {
    Serial.println("Couldn't find ADT7410!");
```

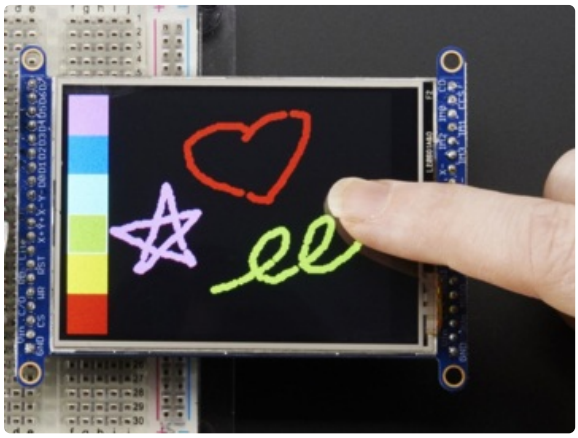
```
    tft.setTextColor(ILI9341_RED);
    tft.println("FAILED");
    tft.setTextColor(ILI9341_GREEN);
  } else {
    Serial.println("ADT7410 found");
    tft.println("OK!");
  }

  coin.initButton(&tft, 120, 280, 100, 50, ILI9341_WHITE, ILI9341_YELLOW, ILI9341_BLACK,
"Sound", 2);
  coin.drawButton();

  analogWriteResolution(12);
  analogWrite(A0, 128);
  pinMode(SPKR_SHUTDOWN, OUTPUT);
  digitalWrite(SPKR_SHUTDOWN, LOW);
}

void loop() {
  digitalWrite(RED_LED, HIGH);
  tft.setTextColor(ILI9341_WHITE);
  // read light sensor
  tft.fillRect(160, 100, 240, 16, ILI9341_BLACK);
  tft.setCursor(0, 100);
  uint16_t light = analogRead(LIGHT_SENSOR);
  Serial.print("light sensor: "); Serial.println(light);
  tft.print("Light sensor: "); tft.println(light);

  // read temp sensor
  tft.fillRect(150, 116, 240, 16, ILI9341_BLACK);
  tft.setCursor(0, 116);
  float temp = tempsensor.readTempC();
  Serial.print("temp sensor: "); Serial.println(temp, 2);
  tft.print("Temp sensor: "); tft.println(temp, 2);

  // externals
  tft.fillRect(0, 132, 240, 32, ILI9341_BLACK);
  tft.setCursor(0, 132);
  float d3 = (float)analogRead(A1) * 3.3 / 1024;
  float d4 = (float)analogRead(A3) * 3.3 / 1024;
  Serial.print("STEMMA: ");
  Serial.print(d3,1); Serial.print(", ");
  Serial.print(d4,1); Serial.println();
  tft.print("D3: "); tft.println(d3,1);
  tft.print("D4: "); tft.println(d4,1);

  tft.fillRect(80, 164, 240, 16, ILI9341_BLACK);
  tft.setCursor(0, 164);
  tft.print("Touch: ");

  TSPoint p = ts.getPoint();
  // we have some minimum pressure we consider 'valid'
  // pressure of 0 means no pressing!
  if (p.z > ts.pressureThreshhold) {
     Serial.print("X = "); Serial.print(p.x);
     Serial.print("\tY = "); Serial.print(p.y);
     Serial.print("\tPressure = "); Serial.println(p.z);
```

```
     int16_t x = map(p.x, X_MIN, X_MAX, 0, 240);
     int16_t y = map(p.y, Y_MIN, Y_MAX, 0, 320);
     tft.print("("); tft.print(x); tft.print(", "); tft.print(y); tft.println(")");
    if (coin.contains(x, y)) {
      Serial.println("Ding!");
      coin.press(true);
    } else {
      coin.press(false);
    }
  } else {
    coin.press(false);
  }
  if (coin.justPressed()) {
    coin.drawButton(true);
    digitalWrite(SPKR_SHUTDOWN, HIGH);

    uint32_t i, prior, usec = 1000000L / SAMPLE_RATE;
    prior = micros();
    for (uint32_t i=0; i<sizeof(coinaudio); i++) {
      uint32_t t;
      while((t = micros()) - prior < usec);
      analogWrite(A0, (uint16_t)coinaudio[i]);
      prior = t;
    }
    digitalWrite(SPKR_SHUTDOWN, LOW);
  }
  if (coin.justReleased()) {
    coin.drawButton(false);
  }
  digitalWrite(RED_LED, LOW);
  delay(20);
}
```

Graphics Demos

One of the nice things about how we set up the PyPortal is it uses a 8 bit parallel display connection, which can be driven much faster than classic SPI displays. Not only do you write 8 bits at a time instead of one, the SPI peripheral tops out at about 24MHz! Combined with DMA you can get surprising speeds. Another nifty hack is taking advantage of the 256KB of SRAM on the SAMD51J20 - you can buffer an entire 240x320 16-bit color bitmap and then draw the whole thing at once

TouchPaint



Our classic touchscreen painting demo is a good example of how to read touch points, convert the raw readings to calibrated coordinates and then draw something on the screen. It's in Adafruit_ILI9341 called **touchpaint_pyportal** (https://adafru.it/EiO)

Amiga Boing!

As featured above, this draws a checkered ball around the screen, a classic Amiga demo. Available in the Adafruit_ILI9341 library as the **pyportal_boing** demo (https://adafru.it/EiP). It's an intense demo showing how to calculate a scan line and render it post computation.

Mandelbrot

This is a good demo to show how we allocate a full display buffer, do all our calculations, then draw it all at once. Also in Adafruit_ILI9341 under example **mandelbrot** (https://adafru.it/EiQ)

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

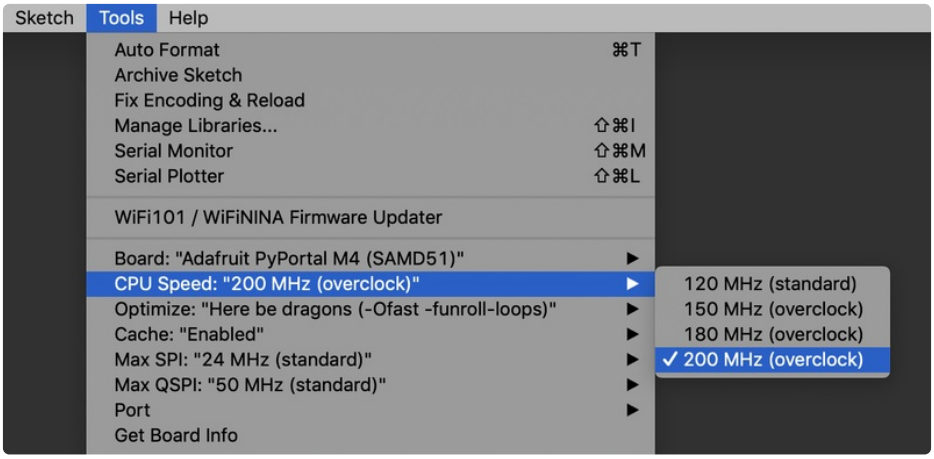
There's *a lot* of registers on the SAMD21, and you often are going through ASF or another framework to

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

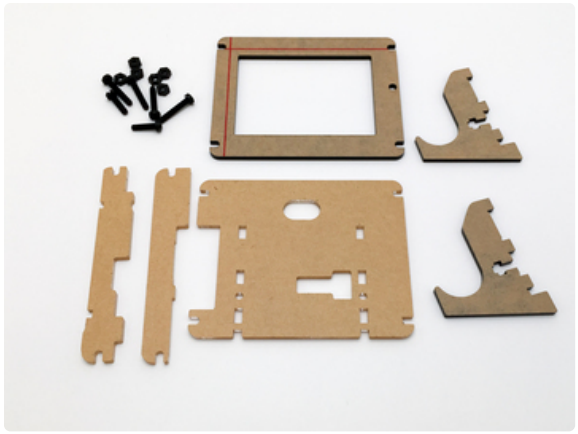
If you want to reduce power draw, some of our boards have an inductor so you can use the 1.8V buck converter instead of the built in linear regulator. If the board does have an inductor (see the schematic) you can add the line `SUPC->VREG.bit.SEL = 1;` to your code to switch to it. Note it will make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

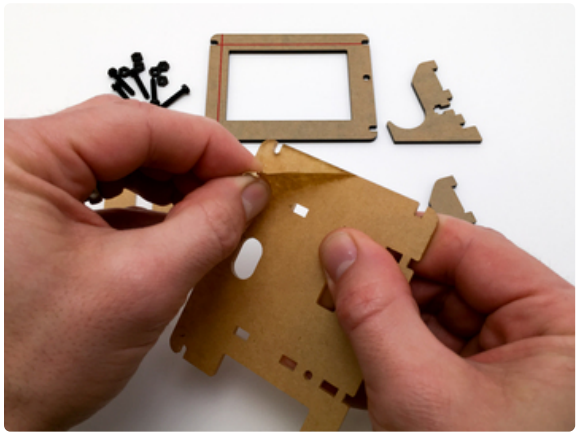
Build the PyPortal Stand

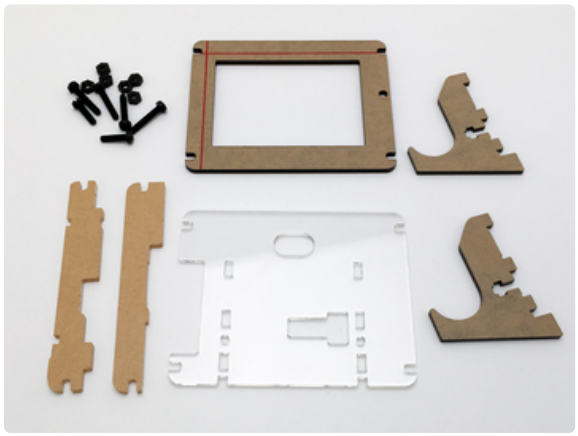
Here's how to assemble the laser cut acrylic stand for the PyPortal. The kit comes with six pieces of acrylic and six nylon screws and nuts.

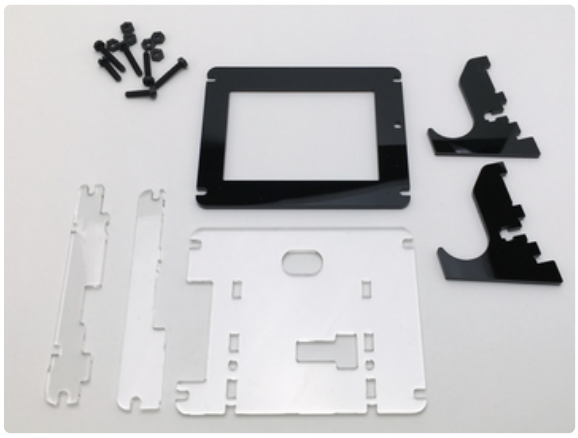
Prep

First, remove the protective paper from all of the acrylic

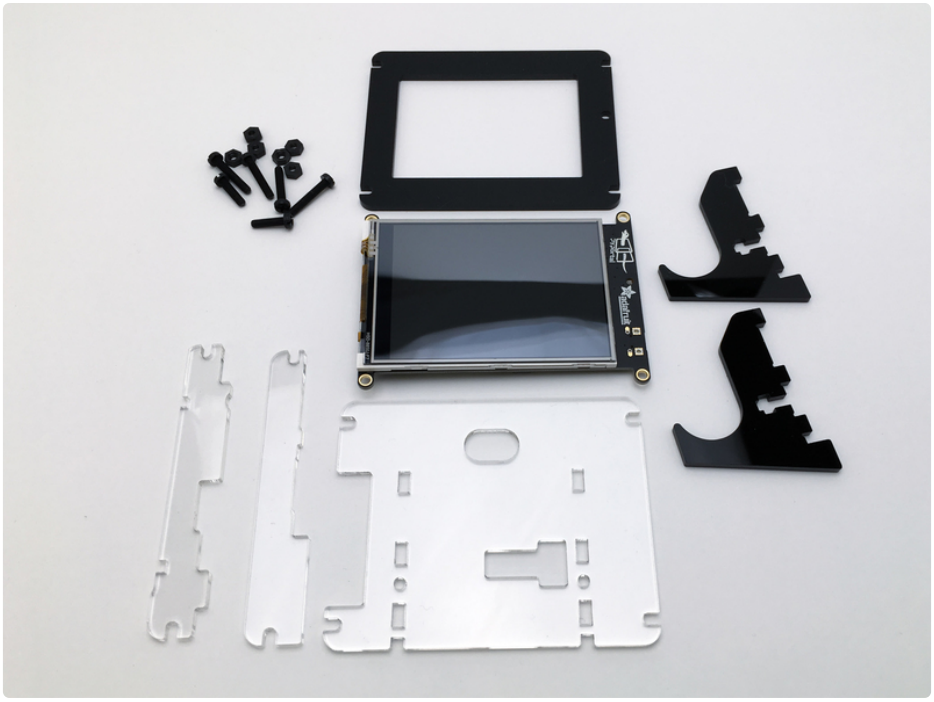


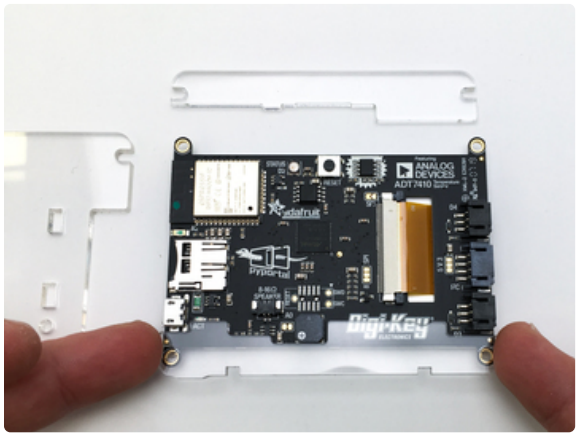






pieces.





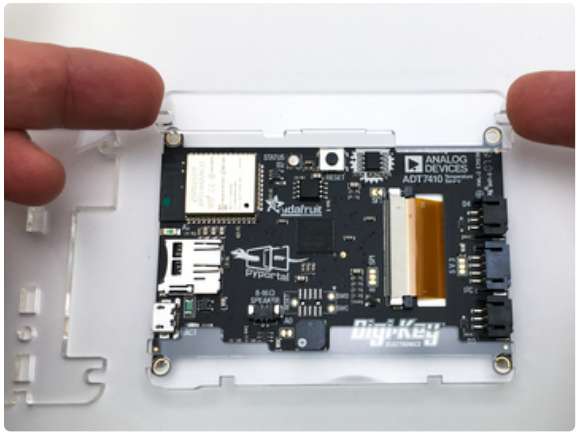
Sandwich

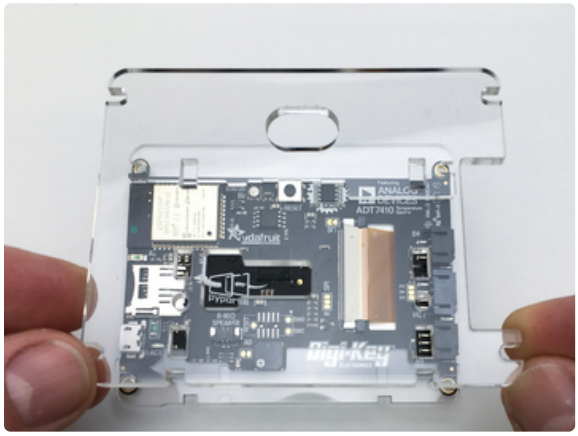
Next, do a dry fit of the three clear piece of acrylic on the back side of the PyPortal to get everything oriented properly.

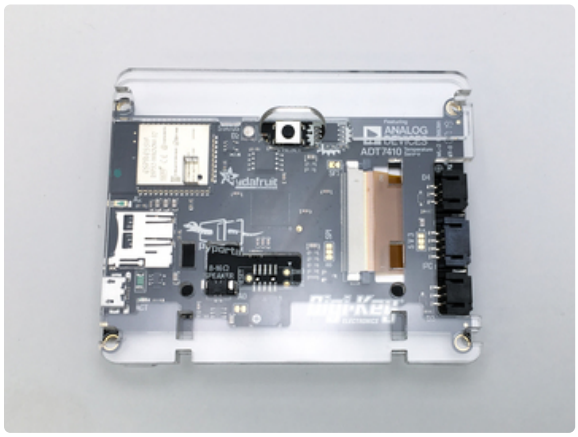
The two small pieces are used as spacers to allow clearance around some of the larger parts. Lay them onto the board first, as shown.

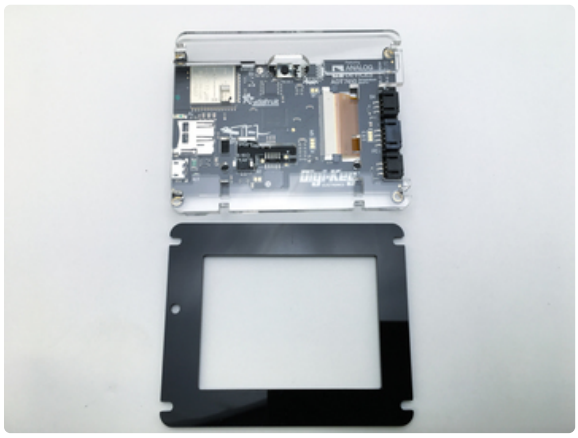
Then, place the large clear piece on top, making sure to align the hole for the reset and the cutout for the three JST ports.

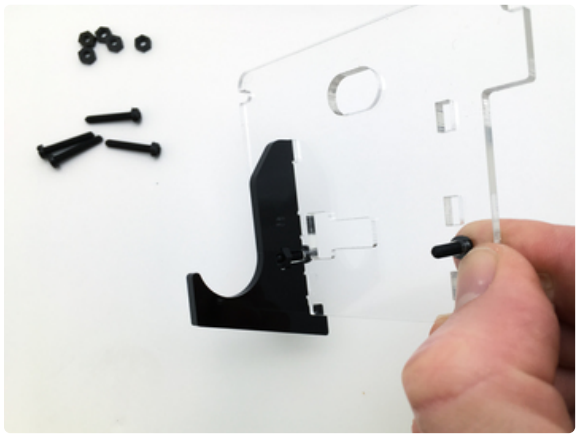
Complete the sandwich by placing the stack on top of the black front bezel with the hole for the light sensor oriented as shown here.









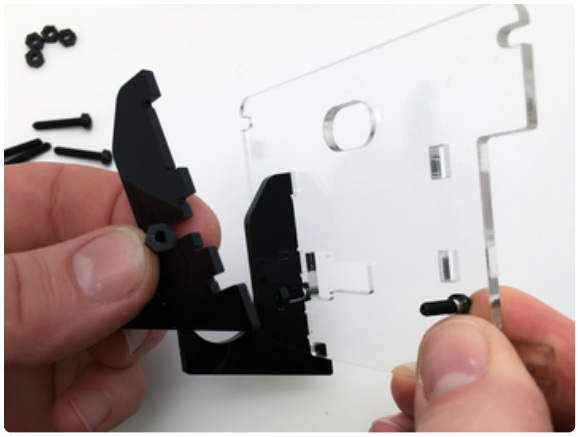


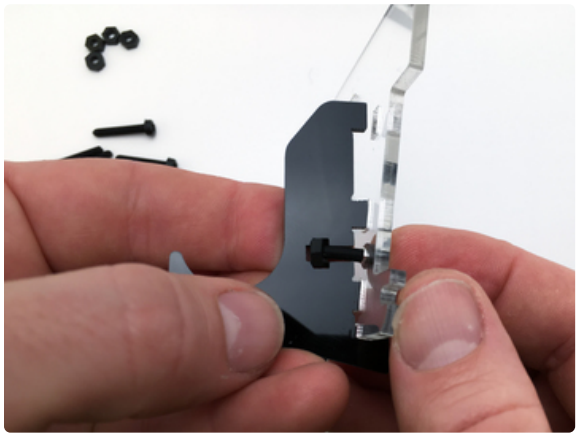
Legs

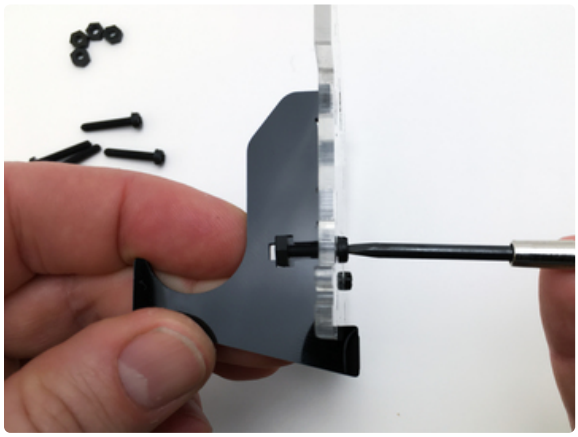
Now that the fit and orientation have been established, we'll install the legs.

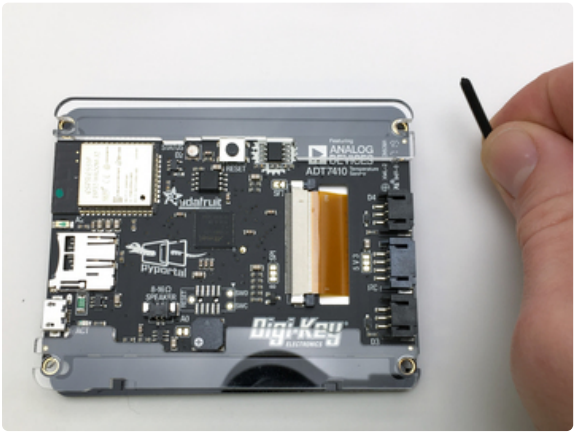
The two legs are identical. Pick one and slot it into the case back as shown.

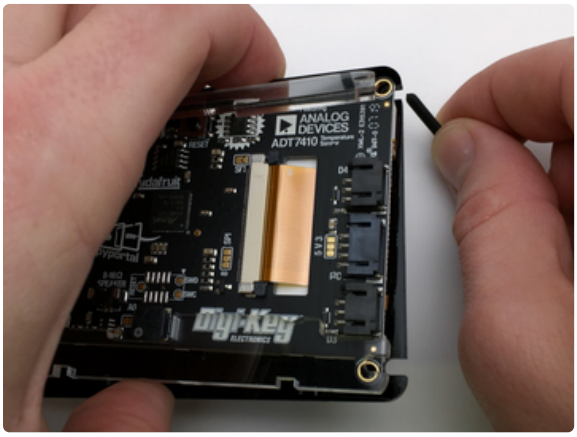
Place a nut into the captive slot of the leg and then feed a short screw through from the front of the clear acrylic case back. Fasten the screw (not too tight!) and then repeat for the second leg.







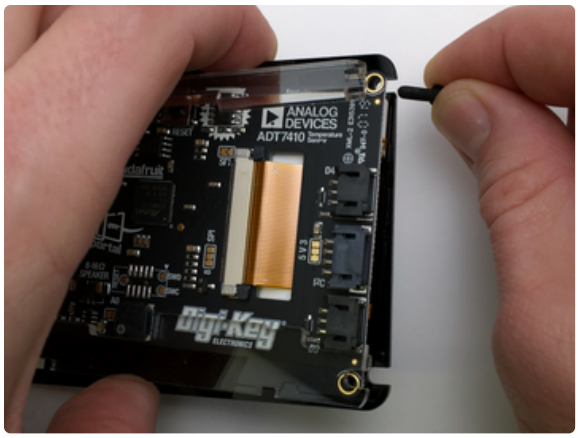


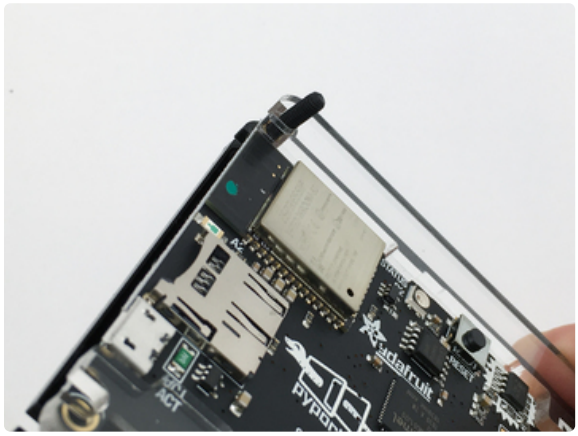


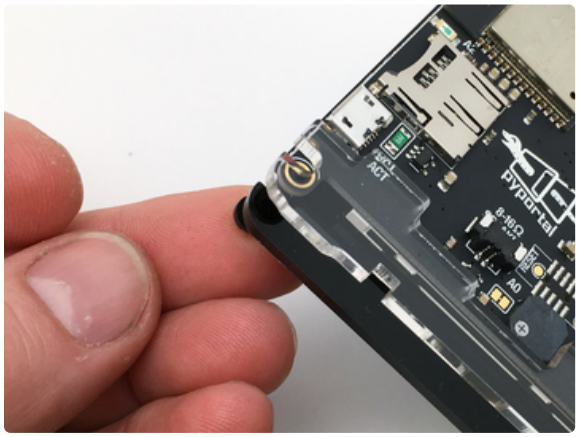
Add Long Screws

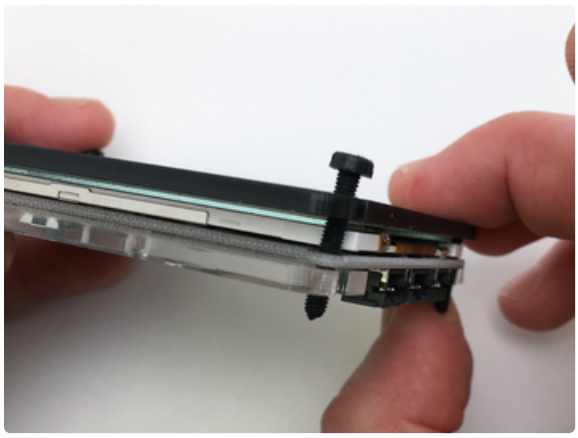
To put it all together, we'll use the four long screws to secure the entire acrylic - PyPortal - acrylic - acrylic sandwich!

Run the four long screws from the front to the back, as shown.

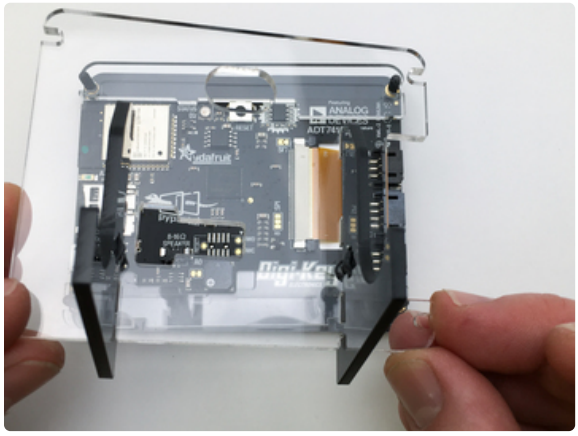

ANALOG DEVICES
ADT7410
adafruit
Digi-Key

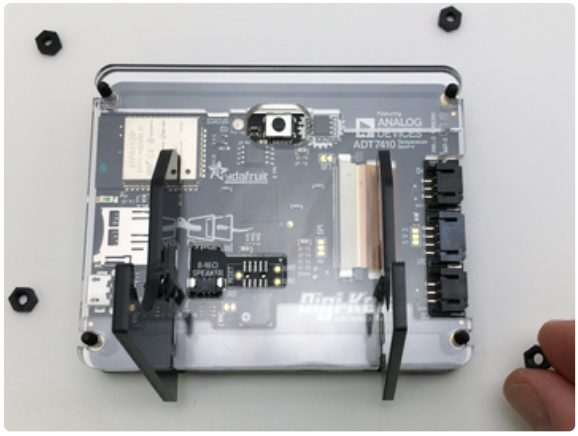

adafruit
pyportal

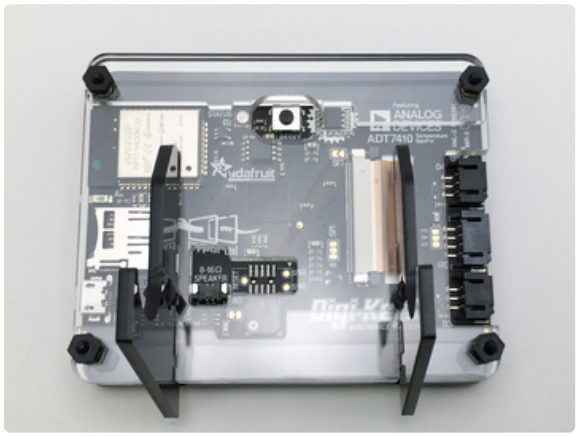

pyportal




STATUS
D2
RESET
Featuring
ANALOG
DEVICES
ADT7410
Temperature
Sensor
adafruit
pyportal
8-16Ω
SPEAKER
RESET
SWD
SWC
A0
A2
SPI
5V3
I2C
D4
D3
Digi-Key
ELECTRONICS



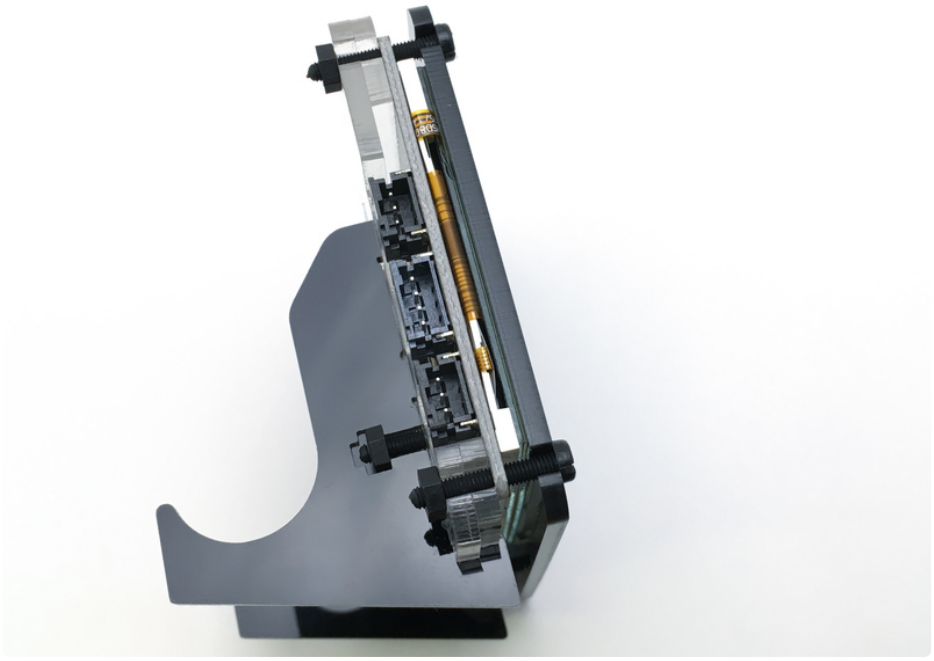




Screw It All Together

Finally, add the case back and legs assemblage and then thread on the four nuts to secure it all in place.

Be **careful not to over-tighten** the screws. Doing so can potentially crack the Pyportal display!



Bonus! Penny Roll Weight

If you'd like to give your PyPortal a bit of extra heft so it won't get pushed around on your desk, you can make a great weight for $0.50. A roll of 50 pennies does the trick! The legs are designed to hold a roll of coins perfectly!

Laser Cutter Files for PyPortal Stand

If you need to replace a piece or just want to make a spare for another PyPortal, here are the vector files for 1/8" (3mm) acrylic, in Adobe Illustrator format:

https://adafru.it/EqN

https://adafru.it/EqN

https://adafru.it/EqO

https://adafru.it/EqO

Updating ESP32 Firmware

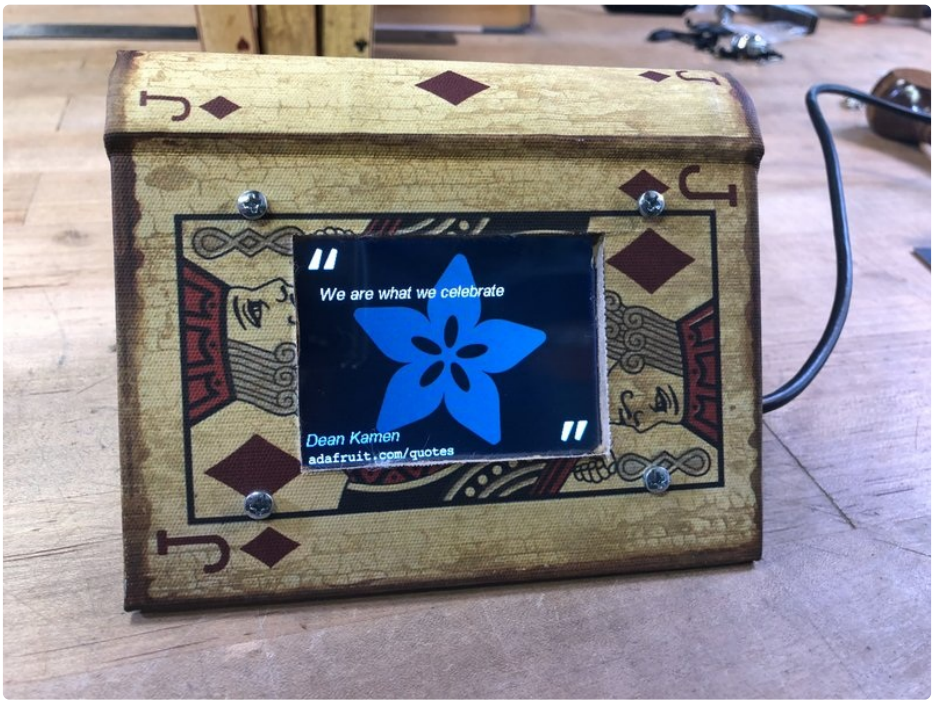
There may come a time when you want to update the firmware on the ESP32 itself. This isn't something we expect you'll do often if at all, but its good to know how if you need to.

We have a guide here which details the process of updating the ESP32 firmware on Airlift All-in-One boards (including the PyPortal, MatrixPortal, and Metro M4 AirLift) here... (https://adafru.it/FWs)

Parsing JSON

Parsing JSON from the Web

Here an example of how you can display text data from the web with PyPortal, by making an internet-connected quote book. (https://adafru.it/Hjb)



```
import time
import board
from adafruit_pyportal import PyPortal

# Set up where we'll be fetching data from
DATA_SOURCE = "https://www.adafruit.com/api/quotes.php"
QUOTE_LOCATION = [0, 'text']
AUTHOR_LOCATION = [0, 'author']

# the current working directory (where this file is)
cwd = ("/"+__file__).rsplit('/', 1)[0]
pyportal = PyPortal(url=DATA_SOURCE,
                    json_path=(QUOTE_LOCATION, AUTHOR_LOCATION),
                    status_neopixel=board.NEOPIXEL,
                    default_bg=cwd+"/quote_background.bmp",
                    text_font=cwd+"/fonts/Arial-ItalicMT-17.bdf",
                    text_position=((20, 120),  # quote location
                                   (5, 210)), # author location
                    text_color=(0xFFFFFF,  # quote text color
                                0x8080FF), # author text color
                    text_wrap=(35, # characters to wrap for quote
                               0), # no wrap for author
                    text_maxlen=(180, 30), # max text size for quote & author
                   )

# speed up projects with lots of text by preloading the font!
pyportal.preload_font()

while True:
    try:
        value = pyportal.fetch()
        print("Response is", value)
    except (ValueError, RuntimeError) as e:
        print("Some error occured, retrying! -", e)
    time.sleep(60)
```

JSON

The neat part is that the text is not coming from a file on the device (see how to do this next), but rather it is grabbed from a website!

Adafruit.com has a PHP script at the **adafruit.com/api/quotes.php** page. Each time it is requested, it returns a new quote from a large database of quotes.

In fact, you can run the same query the PyPortal does to see the results. Copy and paste this link: `https://www.adafruit.com/api/quotes.php`

into your browser and you'll see a result like this:

```
[
 {
  "text": "Science, my lad, is made up of mistakes, but they are mistakes which it is useful to make, bec
ause they lead little by little to the truth",
  "author": "Jules Verne"
 }
]
```

That result is the quote formatted as a **JSON** (JavaScript Object Notation) array. It is comprised of a single element with two keys: **text** and **author**.

- The value of the **text** key is `Science, my lad, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth`
- The value of the **author** key is `Jules Verne`

Since this JSON object has a consistent way to return the results to us, the code we're running on the PyPortal can easily parse the data and display it!

You can see how it's done in this part of **code.py**:

```
# Set up where we'll be fetching data from
DATA_SOURCE = "https://www.adafruit.com/api/quotes.php"
QUOTE_LOCATION = [0, 'text']
AUTHOR_LOCATION = [0, 'author']
```

Then, in the `pyportal` query we ask for the **text** and **author** name from that URL, and then use the `text_` arguments to set the `font`, `position`, `color`, `wrap`, and `maxlen` of the text when it is displayed.

```
pyportal = PyPortal(url=DATA_SOURCE,
                    json_path=(QUOTE_LOCATION, AUTHOR_LOCATION),
                    status_neopixel=board.NEOPIXEL,
                    default_bg=cwd+"quote_background.bmp",
                    text_font=cwd+"fonts/Arial-ItalicMT-17.bdf",
                    text_position=((20, 40),  # quote location
                                   (5, 190)), # author location
                    text_color=(0xFFFFFF,  # quote text color
                                0x8080FF), # author text color
                    text_wrap=(35, # characters to wrap for quote
                               0), # no wrap for author
                    text_maxlen=(180, 30), # max text size for quote & author
                    )
```

With all of this prepared, during the main loop of `while True:` the code will query the Adafruit quotes page for the JSON data, and display it, and then wait one minute until repeating the process.

Parsing local JSON files

If you would like to avoid pulling data from a web page or maybe you can't get access to a specific API key, you can use a "local" JSON file to pull data from.

To implement this local data sourcing method, create a new file and name it `local.txt`. Populate this file with the JSON data that you would like to use. For example, you could use the JSON data provided above and make sure the format of the data is the same. Save this file on the CIRCUITPY drive in the root.

You should not need to change anything in your code.

And that's it! The JSON data will now be pulled from this local file!

PyPortal Hardware FAQ

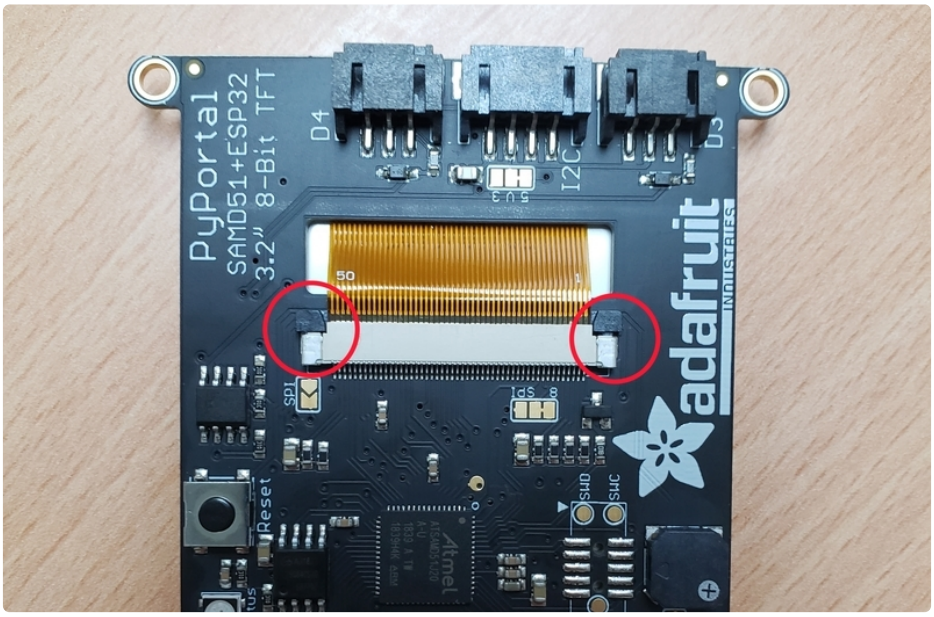
For CircuitPython-specific issues, see the CircuitPython software FAQ (https://adafru.it/HOE).

The PyPortal screen is all white or blank when powered on.

In shipping, the connector for the display may pop the retention tab(s) (red circles below). If you see one that is not in the position below, check to see if the orange display ribbon cable is straight. If so, gently push the tab down towards the main connector.

If you see the ribbon cable crooked, pop both tabs and slide the cable gently so it is in the connector like the picture and straight. then clip each side down.

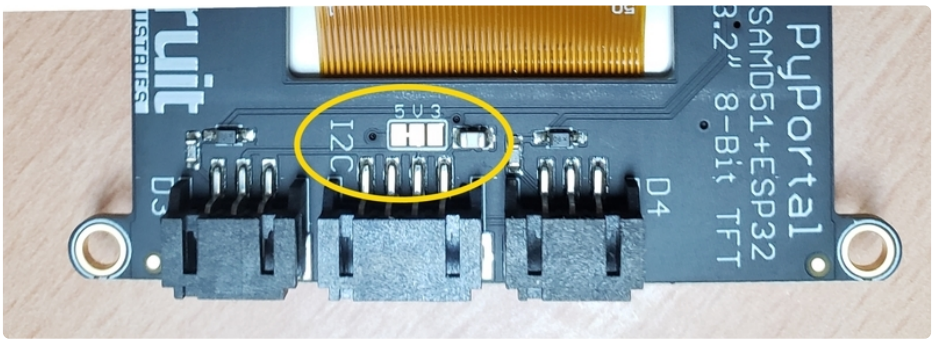
Repower the PyPortal afterwards and hopefully you'll see the CircuitPython boot text on the screen.



I'm using the STEMMA I2C connector to attach sensors and experiencing issues.

If you look at the pads circled in yellow below, there is a tiny connection between the V and 5 pads indicating a default of 5V power, which may cause issue in certain setups. If you are connecting one our sensors and seeing boot or other issues, try changing the voltage to 3.3V

If you want 3.3 volt power for your I2C connector, carefully cut that tiny trace between V and 5, then using a soldering iron connect the 3 and V pads. The PyPortal was not made to switch often between these two values so double check your I2C data sheets, some sensors can take 3.3 and 5V power so leaving it at 5 should be ok.



I'm seeing "AT" or other text being inserted at the REPL prompt.

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
```

What does the status NeoPixel indicate once my code.py is running?

Once your PyPortal boots up and successfully loads your **code.py** or **main.py,** the status NeoPixel will turn green briefly. Then, the NeoPixel will show one of the following color codes to indicate the status of the WiFi connection/activity or file operations:

- Red = not connected to WiFi
- Blue = connected to WiFi
- Yellow = fetching data

- Blue = got data
- Cyan = file opening

My PyPortal Pynt JST STEMMA sockets arent working!

The PyPortal Pynt has the D3 and D4 sockets mislabeled, they should be swapped (to match the pyportal classic)

Downloads

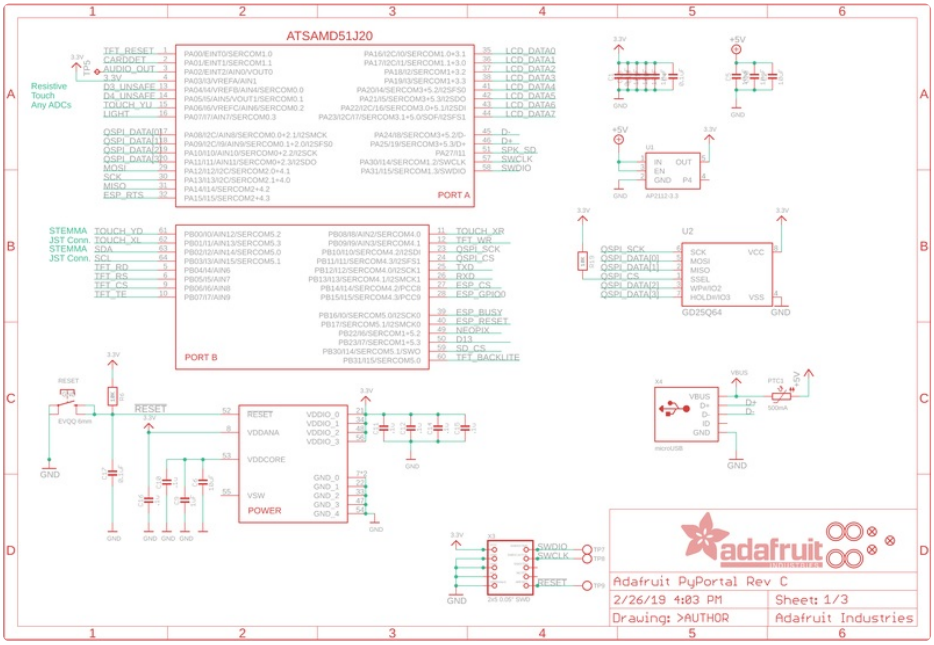
Files

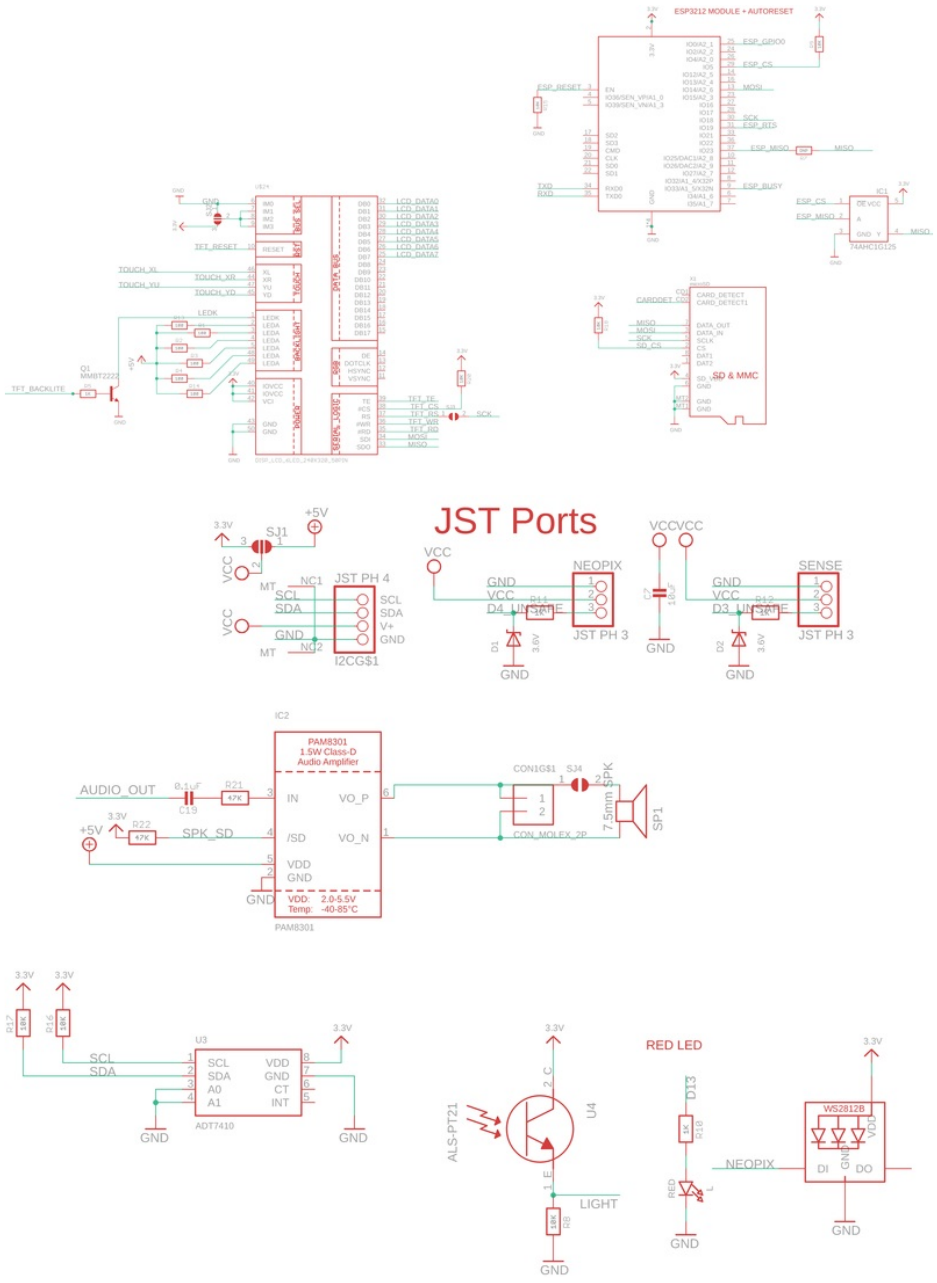
- ATSAMD51J20 datasheet (https://adafru.it/E9c)
- ADT7410 datasheet (https://adafru.it/DPv)
- EagleCAD PCB files on GitHub (https://adafru.it/E9d)
- Fritzing object in Adafruit Fritzing Library (https://adafru.it/E9e)
- Laser Cutter Files for PyPortal Stand (https://adafru.it/EqR)
- PyPortal 3D Models on GitHub (https://adafru.it/HUE)
- PyPortal Pynt 3D Models on GitHub (https://adafru.it/HUF)

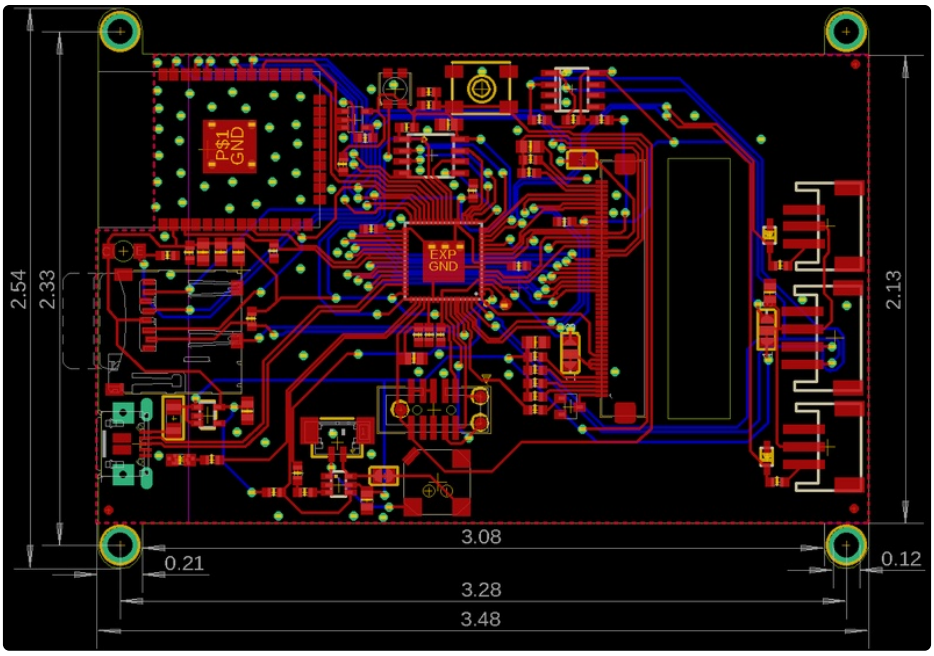
https://adafru.it/Jdl

https://adafru.it/Jdl

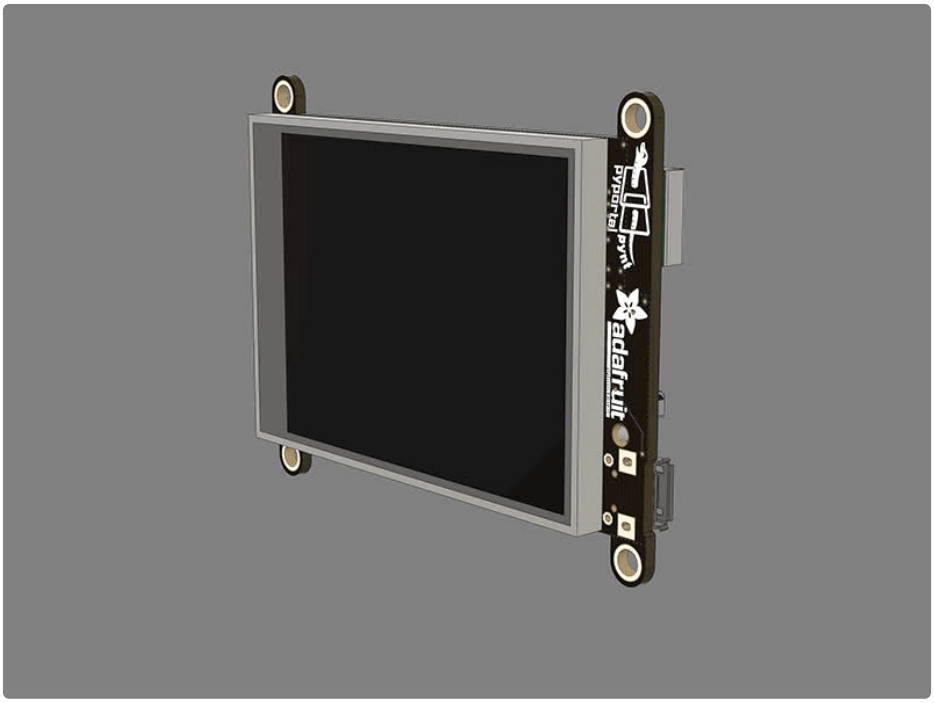
PyPortal Schematic and Fab Print



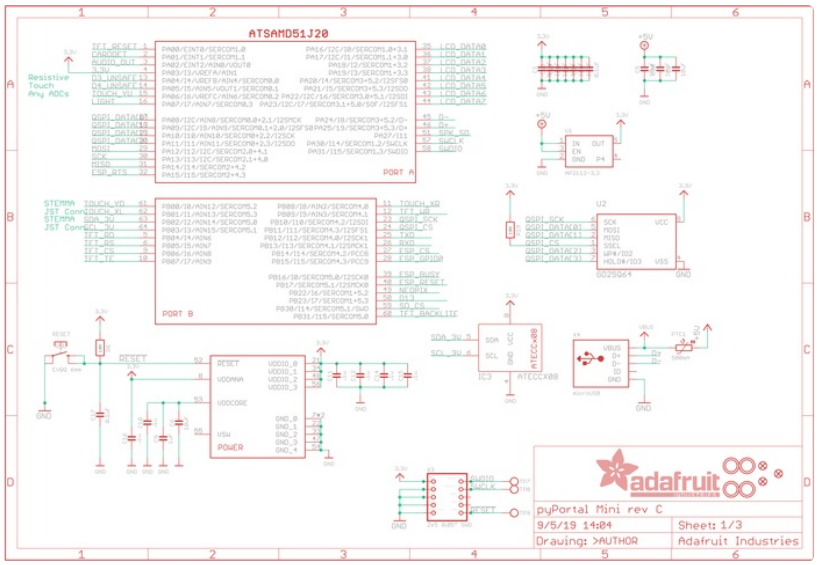

ESP3212 MODULE + AUTORESET
JST Ports
PAM8301
1.5W Class-D
Audio Amplifier
ADT7410
RED LED
SD & MMC

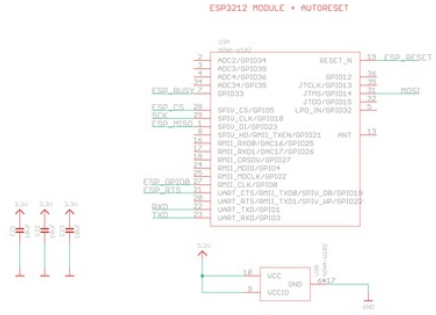


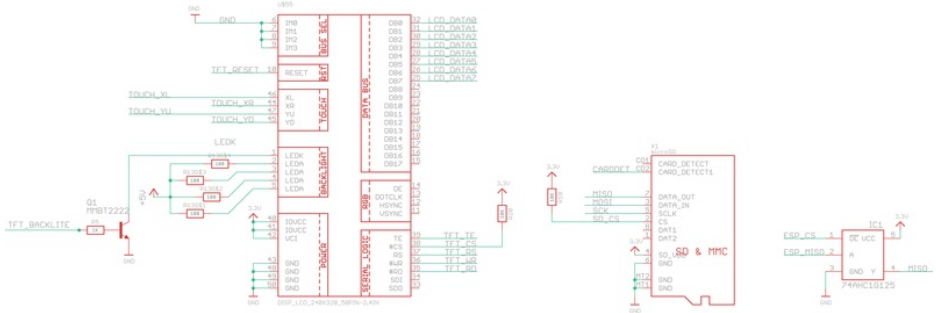
PyPortal Pynt 3D Model

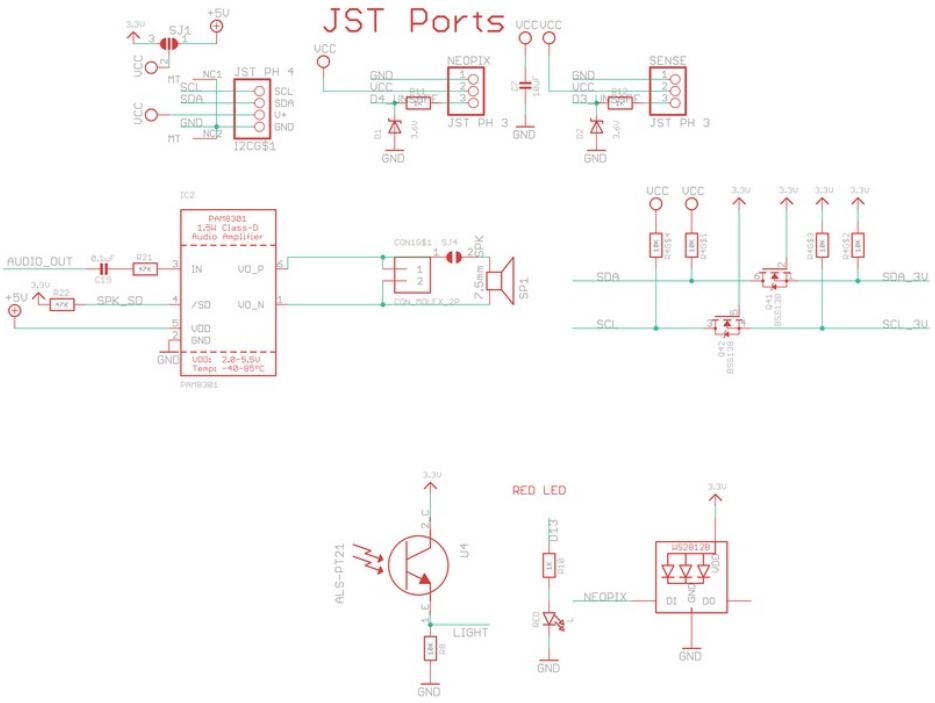


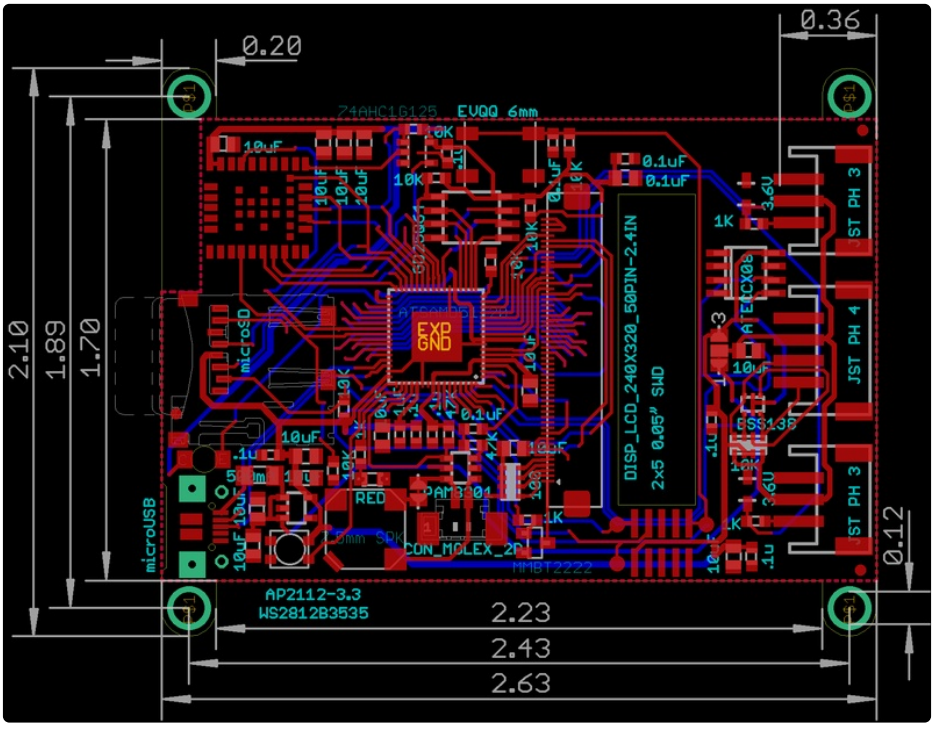
PyPortal Pynt Schematic and Fab Print


ATSAMD51J20
PORT A
PORT B
pyPortal Mini rev C
9/5/19 14:04
Sheet: 1/3
Drawing: >AUTHOR
Adafruit Industries


ESP3212 MODULE + AUTORESET


SD & MMC


JST Ports
JST PH 4
JST PH 3
NEOPIX
SENSE
PAM8301
1.5W Class-D
Audio Amplifier
AUDIO_OUT
SPK_SD
SDA
SCL
SDA_3V
SCL_3V
RED LED
ALS-PT21
LIGHT
WS28128
DI
DO


0.20
0.36
2.10
1.89
1.70
0.12
2.23
2.43
2.63
74AHC1G125
EVQQ 6mm
DISP_LCD_240X320_50PIN-2.4IN
2x5 0.05" SWD
microSD
microUSB
AP2112-3.3
WS2812B3535
PAM8301
CON_MOLEX_2P
NMBT2222
JST PH 3
JST PH 4