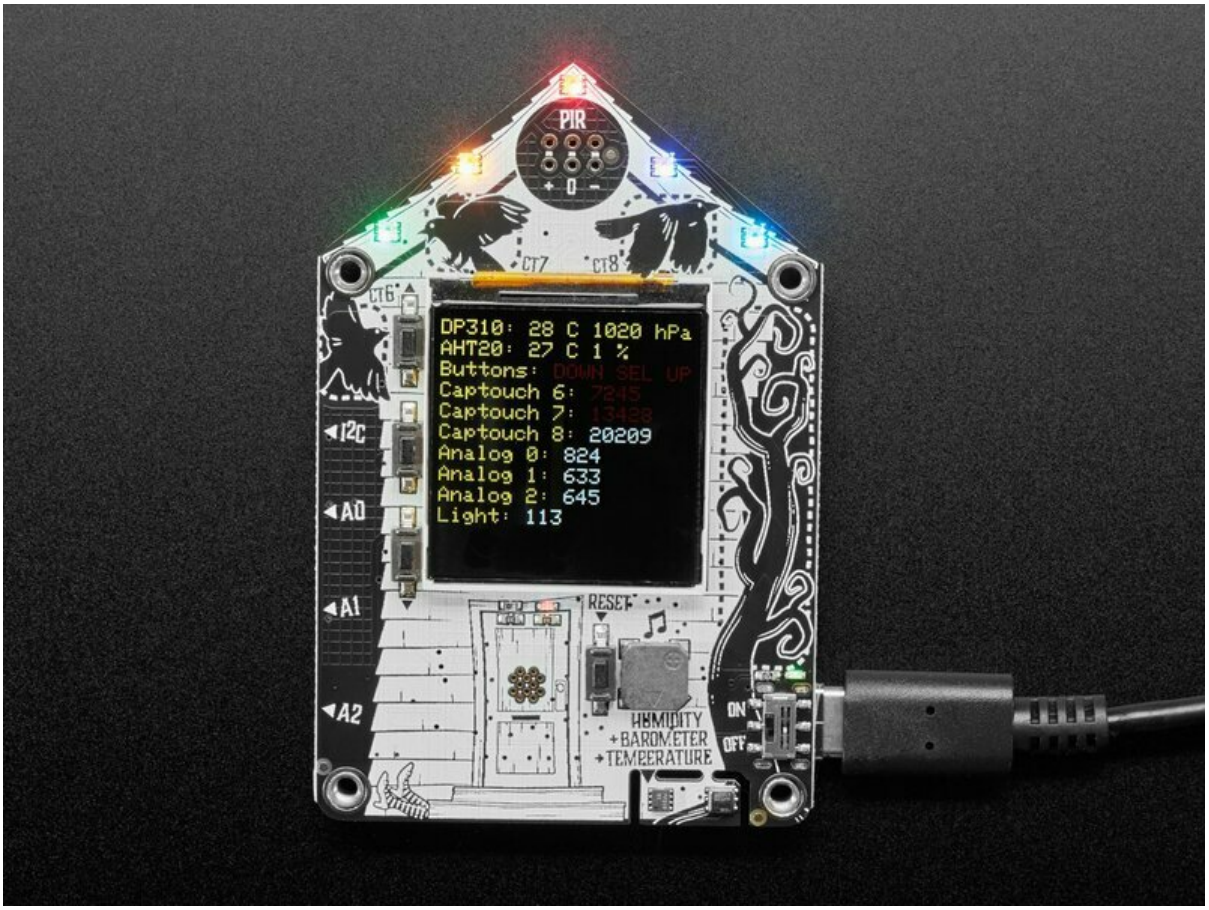
Adafruit FunHouse

Created by Melissa LeBlanc-Williams



https://learn.adafruit.com/adafruit-funhouse

Last updated on 2024-04-06 01:14:43 PM EDT

Table of Contents

Overview 7

Pinouts 10

- TFT Display and Display Connector
- Power
- ESP32-S2 WiFi Module
- DotStar LEDs and Red LED
- STEMMA QT
- Digital/Analog Connectors
- Speaker and Sensors
- Reset and Boot Buttons
- User Buttons
- Capacitive Touch Pads and Slider
- PIR Sensor Port

CircuitPython 19

- Set Up CircuitPython
- Option 1 - Load with UF2 Bootloader
- Option 2 - Use Chrome Browser To Upload BIN file
- Option 3 - Use esptool to load BIN file

CircuitPython Internet Test 23

- The settings.toml File

Getting The Date & Time 28

- Step 1) Make an Adafruit account
- Step 2) Sign into Adafruit IO
- Step 3) Get your Adafruit IO Key
- Step 4) Upload Test Python Code

FunHouse-Specific CircuitPython Libraries 31

- Get Latest Adafruit CircuitPython Bundle
- Secrets

Welcome To CircuitPython 33

- This guide will get you started with CircuitPython!

Installing the Mu Editor 33

- Download and Install Mu
- Starting Up Mu
- Using Mu

Creating and Editing Code 35

- Creating Code
- Editing Code
- Back to Editing Code...
- Naming Your Program File

Connecting to the Serial Console 40

- Are you using Mu?
- Serial Console Issues or Delays on Linux

• Setting Permissions on Linux
• Using Something Else?

Interacting with the Serial Console 43

The REPL 46

• Entering the REPL
• Interacting with the REPL
• Returning to the Serial Console

Advanced Serial Console on Windows 51

• Windows 7 and 8.1
• What's the COM?
• Install Putty

CircuitPython Libraries 54

• The Adafruit Learn Guide Project Bundle
• The Adafruit CircuitPython Library Bundle
• Downloading the Adafruit CircuitPython Library Bundle
• The CircuitPython Community Library Bundle
• Downloading the CircuitPython Community Library Bundle
• Understanding the Bundle
• Example Files
• Copying Libraries to Your Board
• Understanding Which Libraries to Install
• Example: ImportError Due to Missing Library
• Library Install on Non-Express Boards
• Updating CircuitPython Libraries and Examples
• CircUp CLI Tool

CircuitPython Pins and Modules 65

• CircuitPython Pins
• import board
• I2C, SPI, and UART
• What Are All the Available Names?
• Microcontroller Pin Names
• CircuitPython Built-In Modules

Advanced Serial Console on Mac 71

• What's the Port?
• Connect with screen

Frequently Asked Questions 73

• Using Older Versions
• Python Arithmetic
• Wireless Connectivity
• Asyncio and Interrupts
• Status RGB LED
• Memory Issues
• Unsupported Hardware

ESP32-S2 Bugs & Limitations 79

Troubleshooting 80

• Always Run the Latest Version of CircuitPython and Libraries

- I have to continue using CircuitPython 7.x or earlier. Where can I find compatible libraries?
- macOS Sonoma 14.x: Disk Errors Writing to CIRCUITPY
- Bootloader (boardnameBOOT) Drive Not Present
- Windows Explorer Locks Up When Accessing boardnameBOOT Drive
- Copying UF2 to boardnameBOOT Drive Hangs at 0% Copied
- CIRCUITPY Drive Does Not Appear or Disappears Quickly
- Device Errors or Problems on Windows
- Serial Console in Mu Not Displaying Anything
- code.py Restarts Constantly
- CircuitPython RGB Status Light
- CircuitPython 7.0.0 and Later
- CircuitPython 6.3.0 and earlier
- Serial console showing ValueError: Incompatible .mpy file
- CIRCUITPY Drive Issues
- Safe Mode
- To erase CIRCUITPY: storage.erase_filesystem()
- Erase CIRCUITPY Without Access to the REPL
- For the specific boards listed below:
- For SAMD21 non-Express boards that have a UF2 bootloader:
- For SAMD21 non-Express boards that do not have a UF2 bootloader:
- Running Out of File Space on SAMD21 Non-Express Boards
- Delete something!
- Use tabs
- On MacOS?
- Prevent & Remove MacOS Hidden Files
- Copy Files on MacOS Without Creating Hidden Files
- Other MacOS Space-Saving Tips
- Device Locked Up or Boot Looping

Welcome to the Community! 100

- Adafruit Discord
- CircuitPython.org
- Adafruit GitHub
- Adafruit Forums
- Read the Docs

CircuitPython Essentials 109

Blink 109

- LED Location
- Blinking an LED

Digital Input 111

- LED and Button
- Controlling the LED with a Button

Built-In DotStar LEDs 114

- DotStar Location
- DotStar Color and Brightness
- RGB LED Colors
- DotStar Rainbow

CPU Temperature 119

- Microcontroller Location
- Reading the Microcontroller Temperature

Arduino IDE Setup 122

Arduino Libraries 125

- Install Libraries
- Adafruit DotStar
- Adafruit GFX
- Adafruit ST7735 and ST7789
- Adafruit ImageReader
- Adafruit AHTX0
- Adafruit DPS310

Arduino Basics 127

- Using the Red LED
- Reading the Buttons
- Reading the Capacitive Touch Pads
- Using On-Board DotStars
- Using On-board Humidity and Temperature Sensor
- Using On-board Pressure Sensor
- Using the TFT Display

Arduino Self Test Example 133

WipperSnapper Setup 137

- What is WipperSnapper
- Sign up for Adafruit.io
- Add a New Device to Adafruit IO
- Feedback
- Troubleshooting
- "Uninstalling" WipperSnapper

WipperSnapper Essentials 143

DotStar LEDs 143

- Where are the DotStars on my board?
- Create a DotStar Component
- Set the DotStar's RGB Color
- Set DotStar Brightness
- DotStar FAQ

Read a Push-button 150

- Button Location
- Create a Push-button Component on Adafruit IO

Analog Input: Light Sensor 154

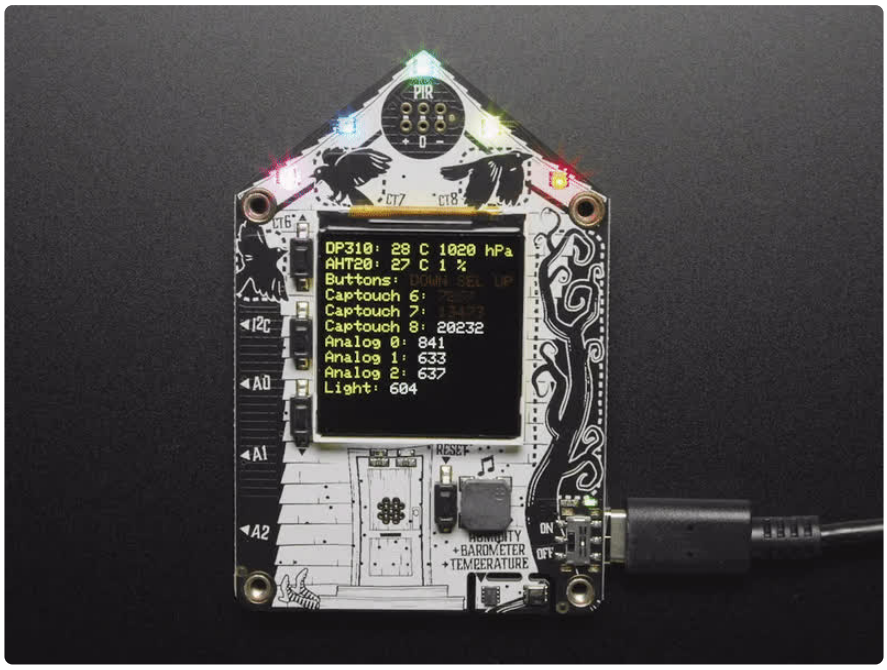
- Analog to Digital Converter (ADC)
- Light Sensor
- Where is the Light Sensor on my board?
- Create a Light Sensor Component
- Light Sensor Usage

I2C: On-board Sensors 160

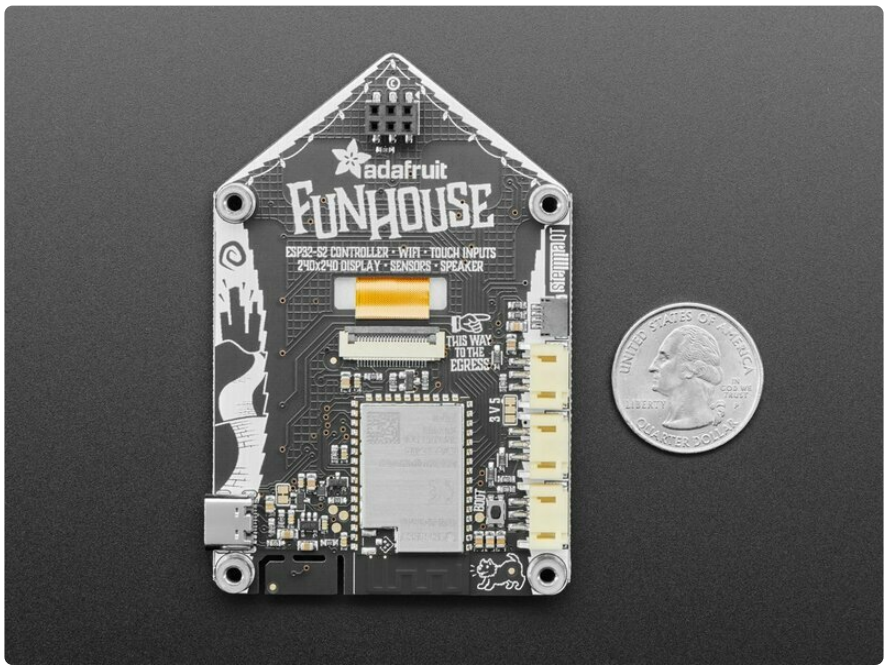
- Where are the I2C sensors on my board?
- Create AHT20 Sensor Component
- Read I2C Sensor Values
- Create DPS310 Component

I2C: External Sensor 168
• Parts
• Wiring
• Add an MCP9808 Component
• Read I2C Sensor Values
Piezo Speaker 174
• Piezo Buzzer Location
• Create a Piezo Buzzer Component
• Modify the Note
Factory Reset 177
• Factory Reset Firmware UF2
• Factory Reset and Bootloader Repair
• Download .bin and Enter Bootloader
• Step 1. Download the factory-reset-and-bootloader.bin file
• Step 2. Enter ROM bootloader mode
• The WebSerial ESPTool Method
• Connect
• Erase the Contents
• Program the ESP32-S2/S3
• The esptool Method (for advanced users)
• Install ESPTool.py
• Test the Installation
• Connect
• Erase the Flash
• Installing the Bootloader
• Reset the board
• Older Versions of Chrome
• The Flash an Arduino Sketch Method
• Arduino IDE Setup
• Load the Blink Sketch
Install UF2 Bootloader 189
Downloads 190
• Files:
• Schematic
• Fab Print
(OLD) WipperSnapper Usage 193
• Blink a LED
• Read a Push-Button
• Read an I2C Sensor
• Going Further

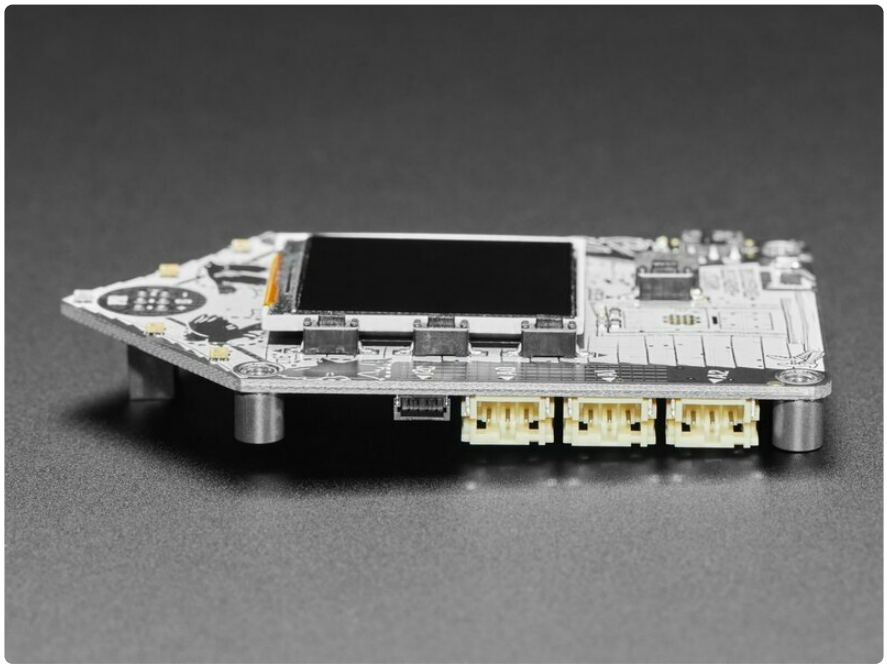
Overview



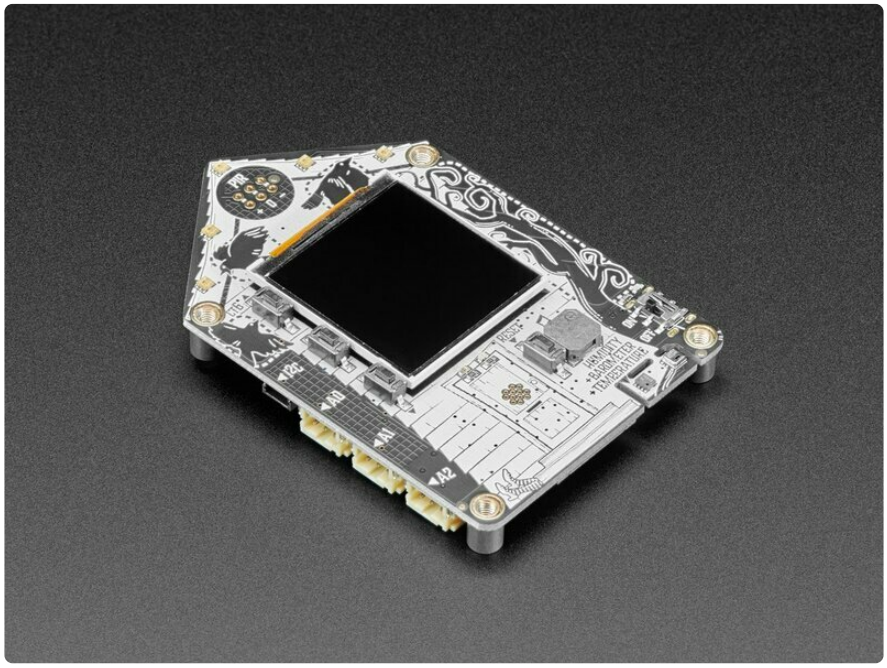
Home is where the heart is...it's also where we keep all our electronic bits. So why not wire it up with sensors and actuators to turn our house into an electronic wonderland. Whether it's tracking the environmental temperature and humidity in your laundry room, or notifying you when someone is detected in the kitchen, to sensing when a window was left open, or logging when your cat leaves through the pet door, this board is designed to make it way easy to make WiFi-connected home automation projects.



The main processor is the ESP32-S2, which has the advantage of the low cost and power of the ESP32 with the flexibility of CircuitPython support thanks to native USB support. There's also Arduino support for this chip, and many existing ESP32 projects seem to run as-is. Note this chip does not have BLE support, but for WiFi projects its great. You can use it to connect to IoT services or just the Internet in general, with SSL support and this module has plenty of PSRAM for any kind of data processing.

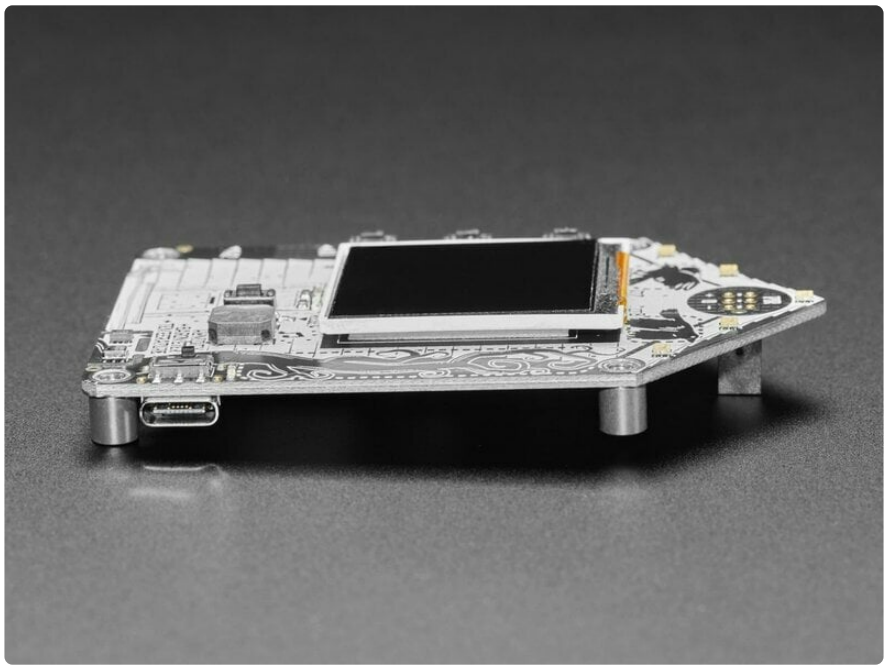


The board is designed to make it easy to wire up sensors with little or no soldering. There are built in sensors for light, pressure, humidity and temperature sensors. Three JST PH plugs allow for quick connection of STEMMA boards (https://adafru.it/RKb) that use digital or analog I/O, and there's a STEMMA QT port for any I2C devices. (https://adafru.it/NmD)

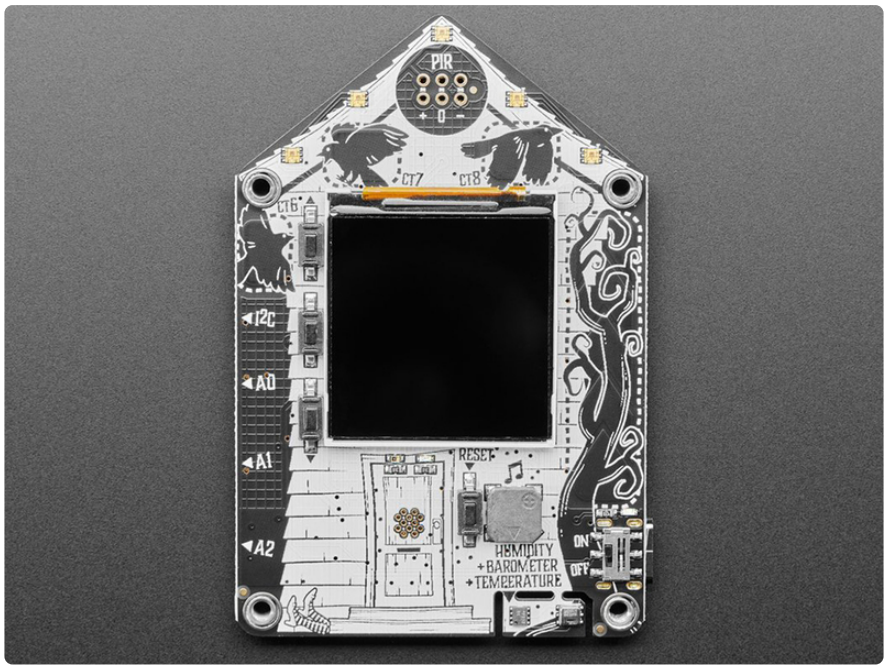


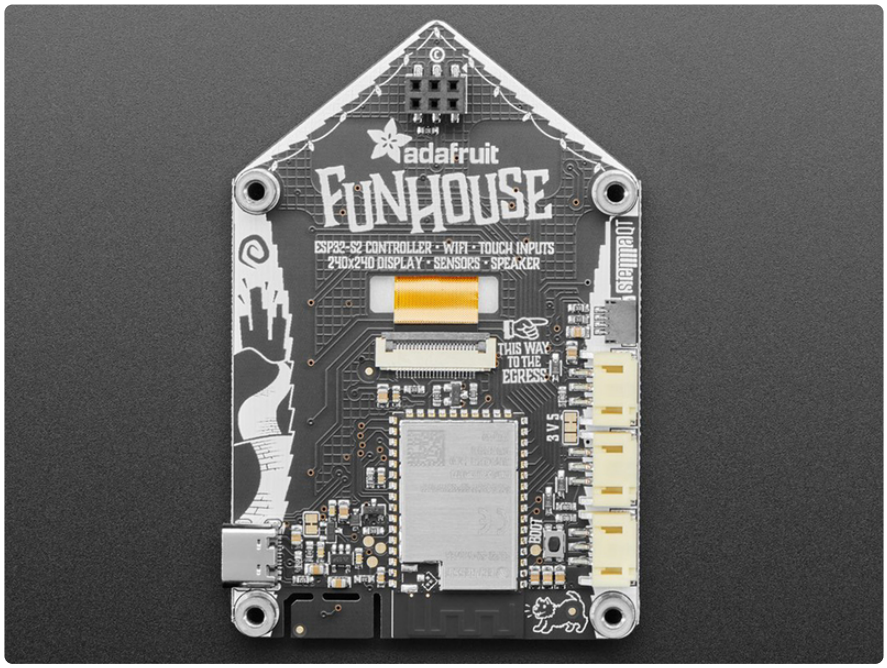
Here's what we included on this development board:

- **ESP32-S2 240MHz Tensilica processor** - the next generation of ESP32, now with native USB so it can act like a keyboard/mouse, MIDI device, disk drive, etc!
- **WROVER module** has FCC/CE certification and comes with 4 MByte of Flash and 2 MByte of PSRAM - you can have huge data buffers
- 1.54" Color TFT display with 240x240 pixels **(http://adafru.it/4421).** This petite display is one of our favorites, with SPI interface and a controllable backlight.
- **USB C** power and data connector
- **Five mini RGB DotStar LEDs** on the top, for animations or easily-visible notification
- **Three buttons** can be used to wake up the ESP32 from deep-sleep, or select different modes
- DPS310 **(http://adafru.it/4494) barometric pressure and temperature sensor**
- AHT20 **(http://adafru.it/4566) relative humidity and temperature sensor**
- Plug in socket for Mini PIR sensor **(http://adafru.it/4871)** (not included)
- Front facing **light sensor**
- **Speaker/Buzzer** can play tones and beeps for audible notification.
- **STEMMA QT** port for attaching all sorts of I2C devices (https://adafru.it/HMF)
- Three **STEMMA 3 pin JST** connectors for attaching NeoPixels (http://adafru.it/3919), speakers (http://adafru.it/3885), servos (http://adafru.it/4326) or relays (http://adafru.it/4409).
- Three **capacitive touch pads** and one **capacitive touch slider** with 5 elements.
- **On/Off switch**
- **Boot** and **Reset buttons** for re-programming

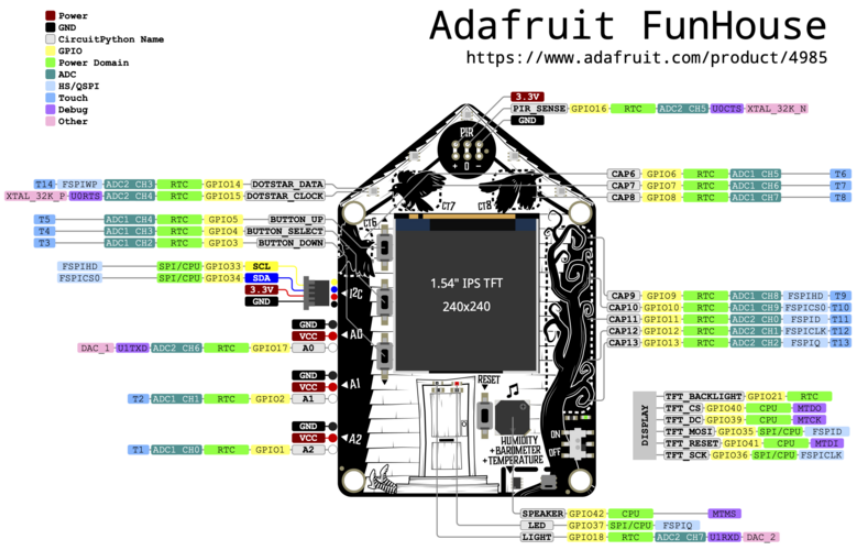


Pinouts



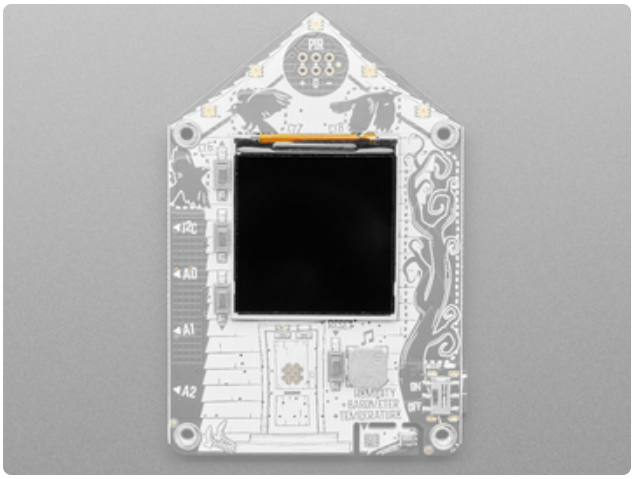


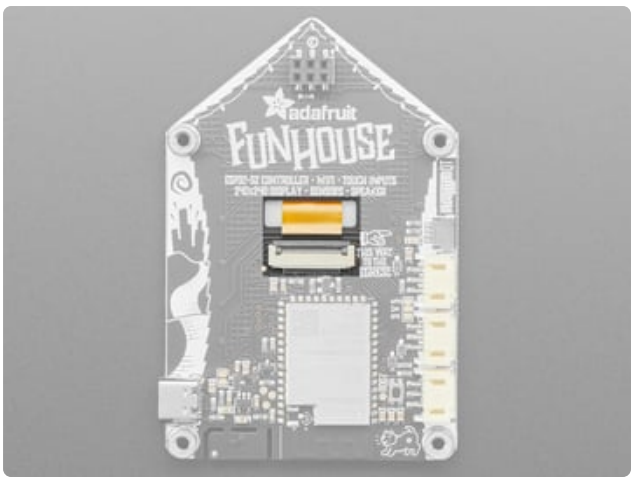
The Funhouse has a display and so many great features. It's packed with buttons, lights, connectors and sensors. Time to take a tour!



[Click here to view a PDF version of the pinout diagram](https://adafru.it/ZqC) (https://adafru.it/ZqC)

TFT Display and Display Connector

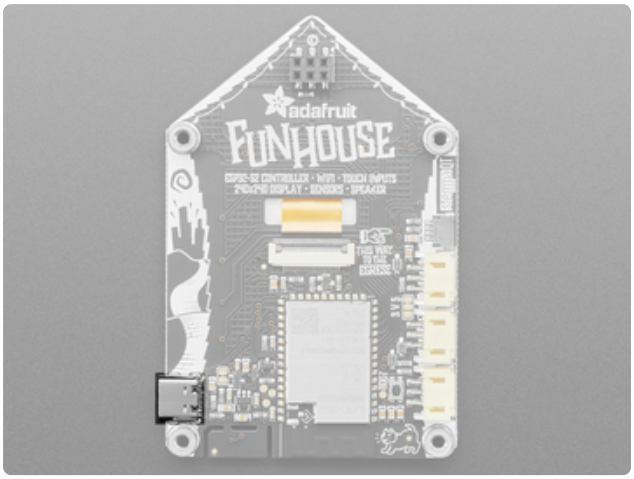


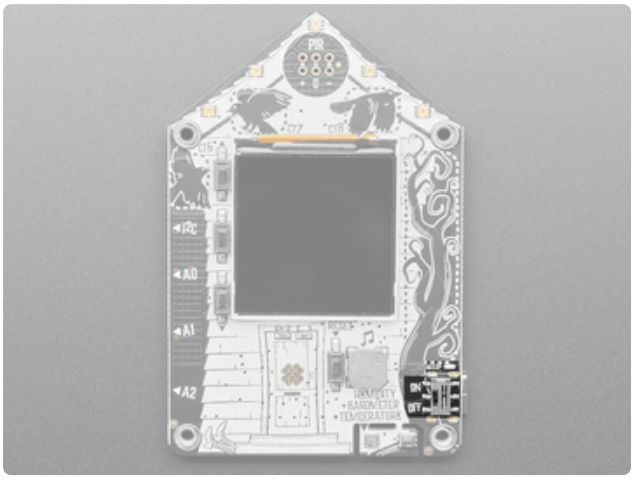


Front and center is a **1.54" ST7789-based IPS Wide Angle TFT Display with 240x240 pixels**. Each pixel is 16-bit full color.

On the back, the display cable goes through the board to the **display connector** on the back.

Power

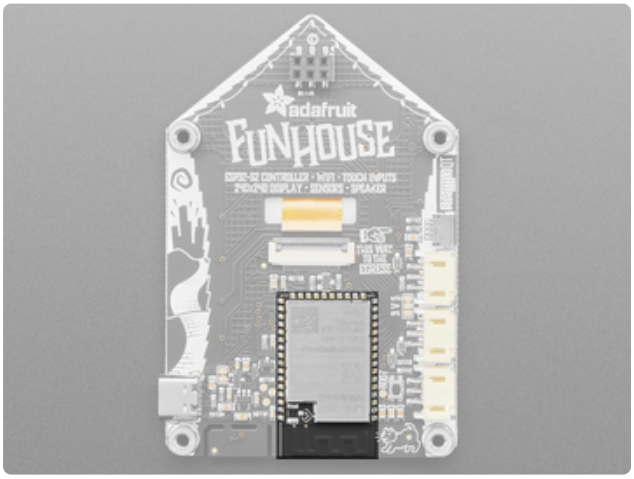




USB C port - This is used for both powering and programming the board. You can power it with any USB C cable and will request 5V from a USB C PD.

On/Off switch - This switch controls power to the board. If you plug in your board and nothing happens, make sure the switch is flipped to "ON"!

ESP32-S2 WiFi Module



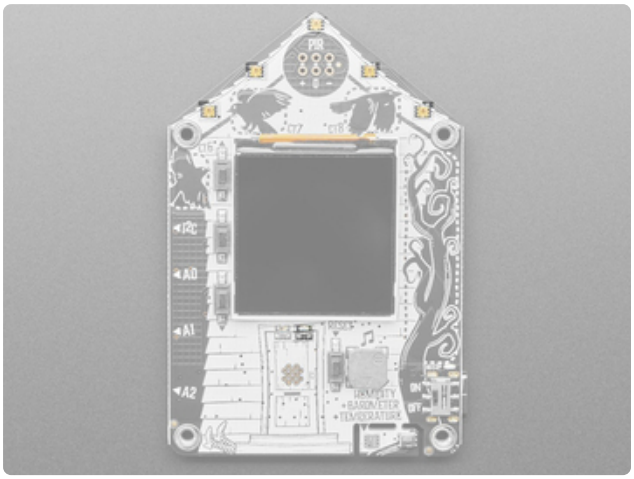
The **ESP32-S2 WROVER module**.

The ESP32-S2 is a highly-integrated, low-power, 2.4 GHz Wi-Fi System-on-Chip (SoC) solution that now has **built-in native USB** as well as some other interesting new technologies like Time of Flight distance measurements. With its state-of-the-art power and RF performance, this SoC is an ideal choice for a wide variety of application scenarios relating to the Internet of Things (IoT) (https://adafru.it/Bwq), wearable electronics (https://adafru.it/Osb), and smart homes.

Please note, this is a single-core 240 MHz chip, so it won't be as fast as ESP32's with dual-core. Also, there is no Bluetooth support. However, we are super excited about the ESP32-S2's native USB which unlocks a lot of capabilities for advanced interfacing! **This WROVER module comes with 4 MB flash and 2 MB PSRAM.**

The 4 MB of flash is inside the module and is used for **both** program firmware and filesystem storage. For example, in CircuitPython, we have 3 MB set aside for program firmware (this includes two OTA option spots as well) and a 1MB section for CircuitPython scripts and files.

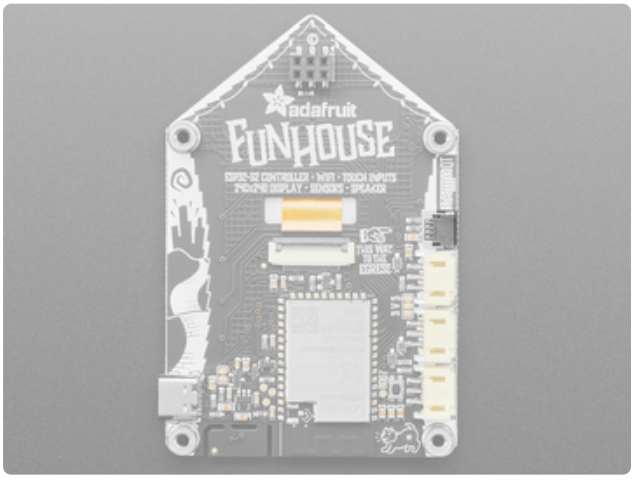
DotStar LEDs and Red LED



DotStar LEDs and red LED.

On the front of the board, along the top, are **five addressable RGB DotStar LEDs**, so you can light up the display with any color or pattern. You can use `DOTSTAR_CLOCK` or `GPIO15` for the Clock pin and `DOTSTAR_DATA` or `GPIO14` for the Data pin to control the DotStars.
Below the display, and slightly to the right, is a **red LED** positioned at the top right of the FunHouse door. It is user-controllable for blinky needs. You can blink this at any time. This LED is attached to `LED` or `GPIO37`.

STEMMA QT



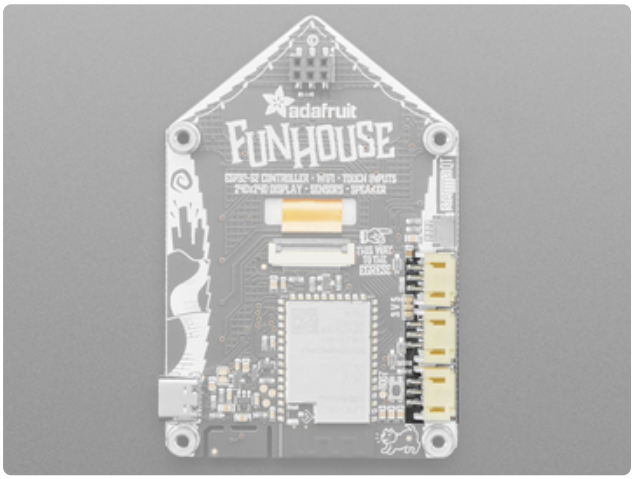
STEMMA QT **(https://adafru.it/Ft4)** - This JST SH 4-pin connector breaks out I2C (SCL, SDA, 3.3V, GND). It allows you to connect to various breakouts and sensors with **STEMMA QT** connectors (https://adafru.it/HMF) or to other things using assorted associated accessories (https://adafru.it/Ft6).

In CircuitPython, you can use the STEMMA connector
with `board.SCL` and `board.SDA`,
or `board.STEMMA_I2C()`.

Works great with any STEMMA QT or Qwiic sensor/device

You can also use it with Grove I2C devices thanks to this handy cable (http://adafru.it/4528)

Digital/Analog Connectors



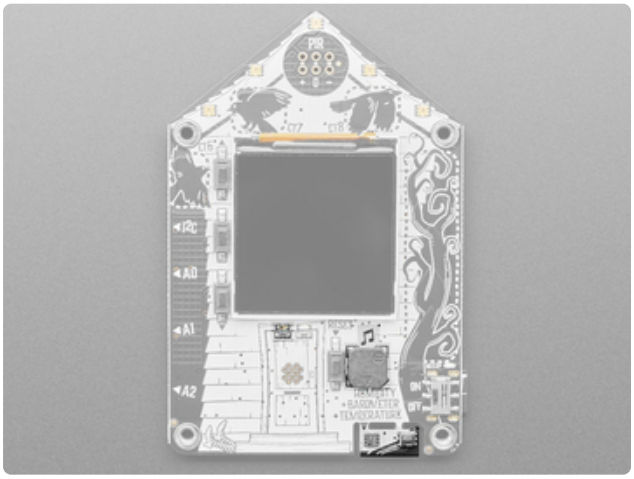
On the side are three connectors labeled **A0**, **A1**, and **A2**. These are **STEMMA 3 pin JST digital or analog connectors** for attaching NeoPixels (http://adafru.it/3919), speakers (http://adafru.it/3885), servos (http://adafru.it/4326) or relays (http://adafru.it/4409). These pins can be analog inputs or digital I/O. They are connected to `GPIO17` for `A0`, `GPIO2` for `A1`, and `GPIO1` for `A2`.

All three connectors have protection 1K resistors + 3.6V zener diodes so you can drive an LED directly from the output. The maximum current from these connectors is 200mA.

All three ports are 'true' analog outputs and all three can be used for PWM as well as analog inputs. The maximum input voltage is 2.6V, after which the zener diodes will kick in to drain excess voltage.

The power output is 5V by default, but a jumper can be cut/soldered to change it to 3.3V.

Speaker and Sensors



Towards the middle of the board, at the bottom right corner of the display, is a **speaker/buzzer** labeled with a musical note. This includes a mini class D amplifier on DAC output `GPIO42` and can play tones or lo-fi audio clips.

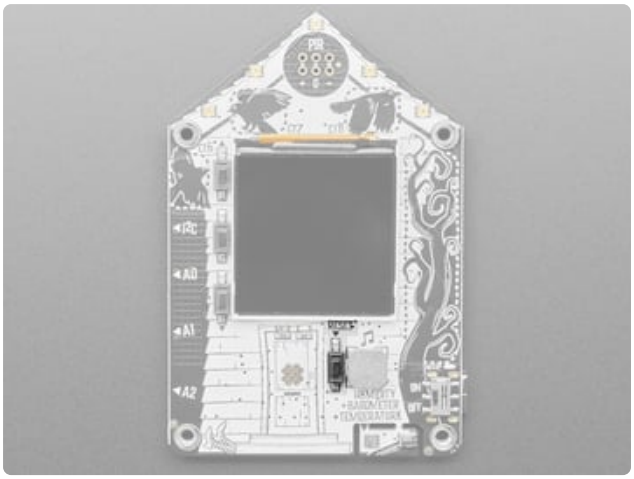
In the bottom right corner of the board, on the left side of the cutout region, is a **DPS310 pressure sensor**, that can be used to sense the barometric pressure It is connected to the I2C port and available on I2C address `0x77`.

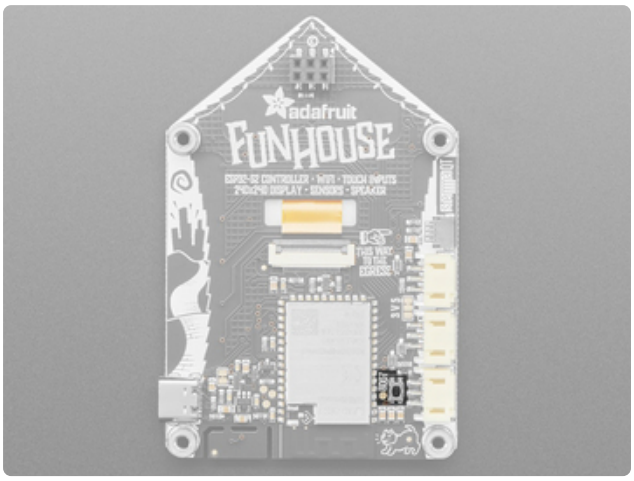
Also in the bottom right corner of the board, on the right side of the cutout region, is an **AHT20 Humidity and Temperature sensor**, that can be used to sense the humidity and temperature. It is connected to the I2C port and available on I2C address `0x38`.

Below the display, and slightly to the left, is a front-facing **light sensor** that is positioned at the top left of the FunHouse door. This is connected to `GPIO18`.

Note: The light sensor is influenced by the display's backlight, use some tape to block the light if needed. More info, and a great chart here (https://adafru.it/doW).

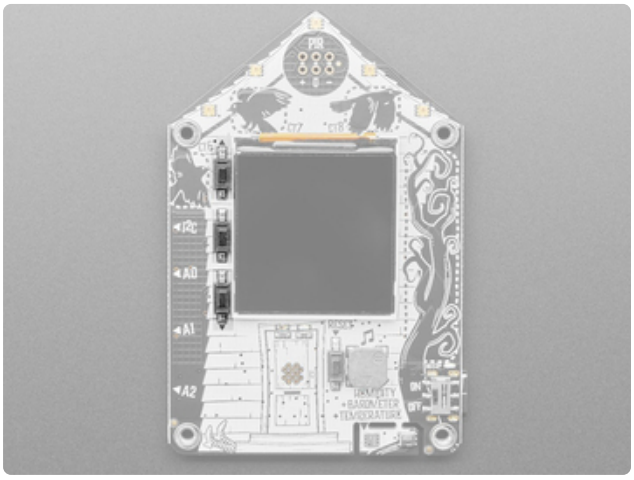
Reset and Boot Buttons





Reset button - The reset button is on the front below the display and to the right of the FunHouse Door.
Boot button - The boot button is on the back and situated between the Digital/ Analog Ports and the ESP32-S2 module. This is connected to **BOOT0** and can be used to put the board into ROM bootloader mode. To enter ROM bootloader mode, **hold down DFU button while clicking reset button mentioned above.** When in the ROM bootloader, you can upload code and query the chip using `esptool`.

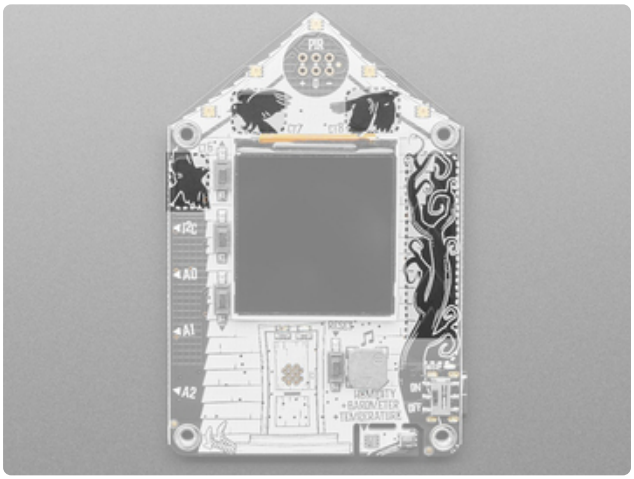
User Buttons



On the front of the board, to the left of the display, there are **three user-controllable buttons** labeled arrows for the top and bottom buttons. The buttons are on `BUTTON_DOWN` or `GPIO3`, `BUTTON_SELECT` or `GPIO4`, and `BUTTON_UP` or `GPIO5`. They can be used to wake up the ESP32-S2 from deep-sleep, or however you want to use them.

There are no pull-ups on board, use internal pulldowns for these pins - when the buttons are pressed the IO pin labeled is set to **HIGH**

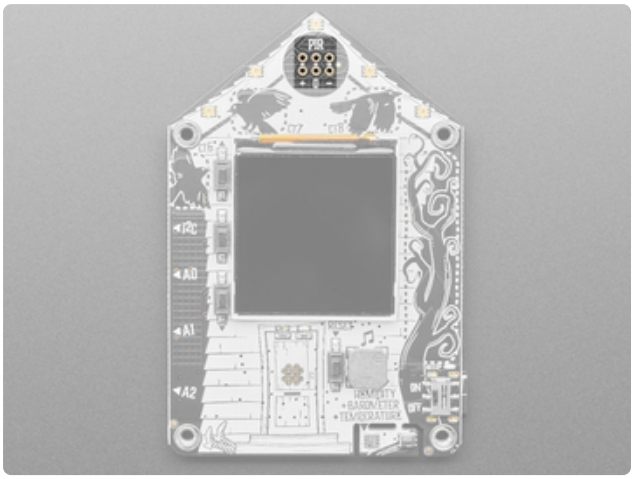
Capacitive Touch Pads and Slider

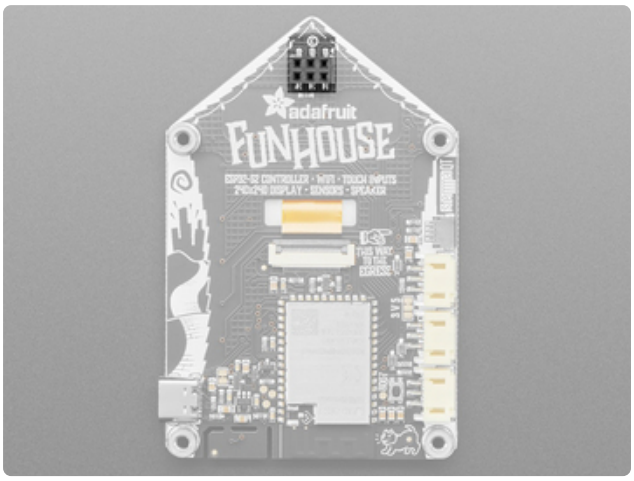


On the front of the board, to the left of the display and along the top, there are **three capacitive touch pads** with ravens on top labeled **CT6**, **CT7**, and **CT8**. The pads are on `CAP6` or `GPIO6`, `CAP7` or `GPIO7`, and `CAP8` or `GPIO8`. They can be used like buttons.

On the right side of the board, there is a **capacitive touch slider** made up of 5 capacitive touch pads. It has a tree on top. The pads are on `CAP9` or `GPIO9`, `CAP10` or `GPIO10`, `CAP11` or `GPIO11`, `CAP12` or `GPIO12`, and `CAP13` or `GPIO13`. They can be used as separate buttons or a positional slider.

PIR Sensor Port





On the front of the FunHouse is a location to add the PIR sensor. It is intended to be inserted through the front and into the connector on the back, but it could also work from the back. **When inserting, make sure the + and - symbols match up with the PIR sensor's markings or it will short out the board.**

The PIR sensor is connected to `PIR_SENSE` or `GPIO16`.

CircuitPython

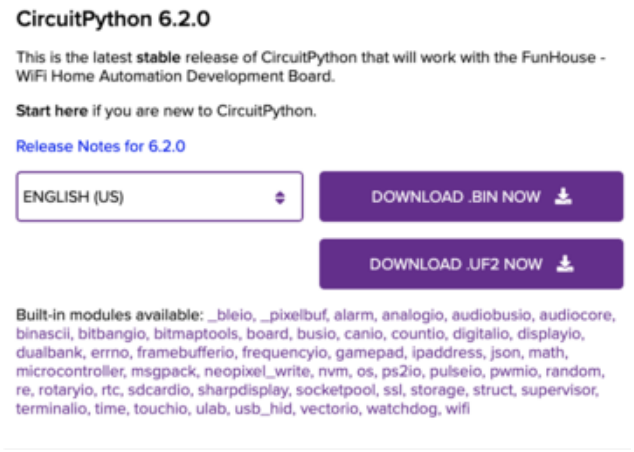
CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/f9W) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the **CIRCUITPY** drive to iterate.

Set Up CircuitPython

Follow the steps to get CircuitPython installed on your FunHouse.

Download the latest CircuitPython for your board from circuitpython.org

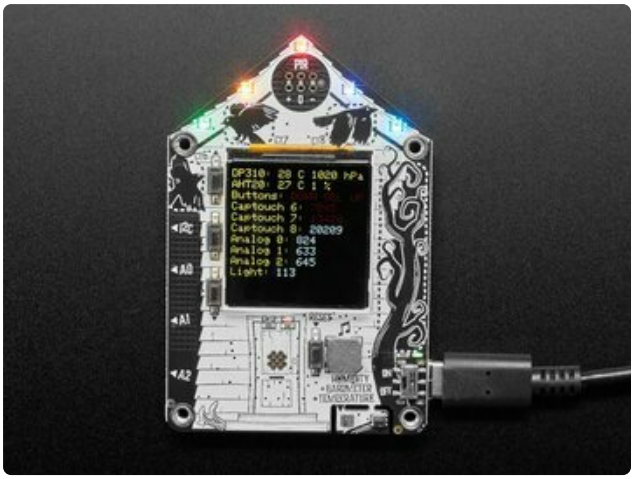
https://adafru.it/RKB



Click the link above and download the latest .BIN and .UF2 file

(depending on how you program the ESP32S2 board you may need one or the other, might as well get both)

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



Plug your FunHouse 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.

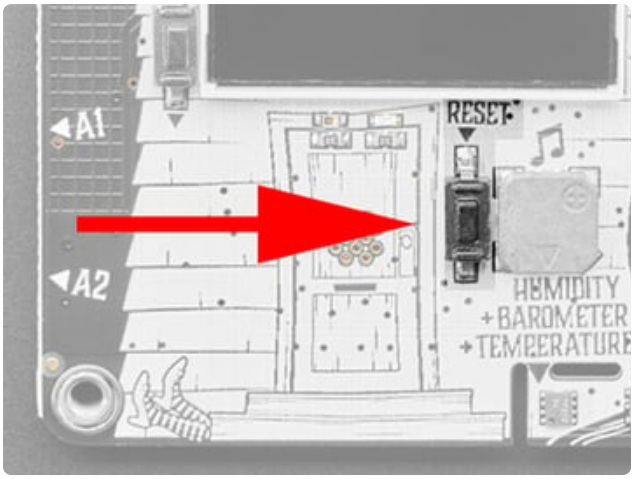
Option 1 - Load with UF2 Bootloader

This is by far the easiest way to load CircuitPython. **However it requires your board has the UF2 bootloader installed. Some early boards do not (we hadn't written UF2 yet!) - in which case you can load using the built in ROM bootloader.**

Still, try this first!

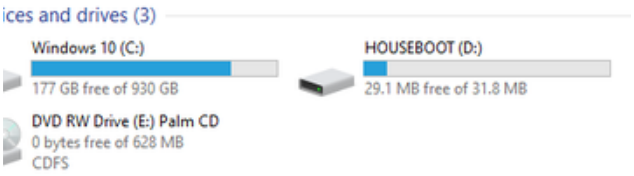
Try Launching UF2 Bootloader

Loading CircuitPython by drag-n-drop UF2 bootloader is the easier way and we recommend it.



Launch UF2 by **double-clicking** the Reset button (the one next to the USB C port). You may have to try a few times to get the timing right.

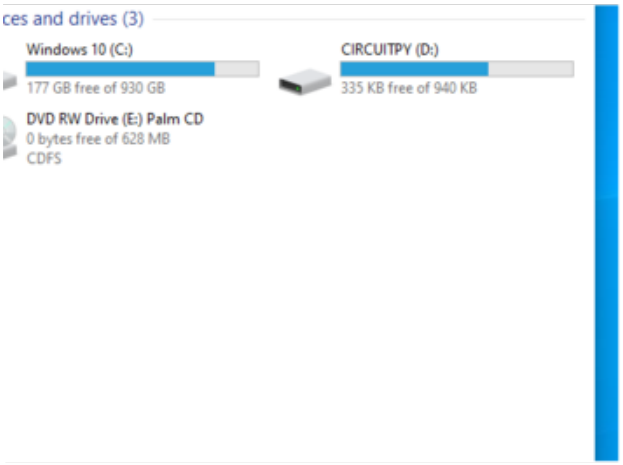
About a half second pause between clicks while the DotStars are purple seems to work well.



If the UF2 bootloader is installed, you will see a new disk drive appear called **HOUSEBOOT**

Copy the **UF2** file you downloaded at the first step of this tutorial onto the **HOUSEBOOT** drive

If you're using Windows and you get an error at the end of the file copy that says **Error from the file copy, Error 0x800701B1: A device which does not exist was specified.** You can ignore this error, the bootloader sometimes disconnects without telling Windows, the install completed just fine and you can continue. If its really annoying, you can also upgrade the bootloader (the latest version of the UF2 bootloader fixes this warning) (https://adafru.it/RLc)



Your board should auto-reset into CircuitPython, or you may need to press reset. A **CIRCUITPY** drive will appear. You're done! Go to the next pages.

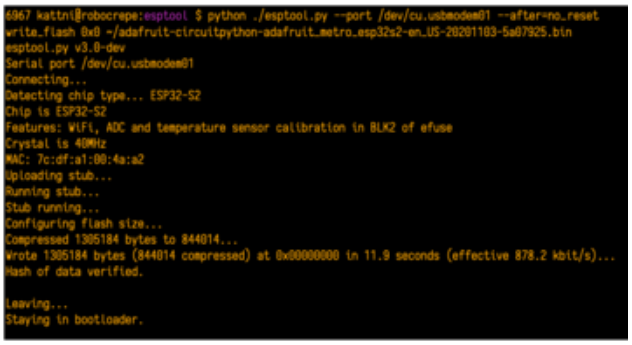
Option 2 - Use Chrome Browser To Upload BIN file

You will need to do a full erase prior to uploading new firmware.

The next best option is to try using the Chrome-browser version of esptool we have written. This is handy if you don't have Python on your computer, or something is really weird with your setup that makes esptool not run (which happens sometimes and isn't worth debugging!) You can follow along on the Install UF2 Bootloader (https://adafru.it/RLc) page and either load the UF2 bootloader and then come back to Option 1 on this page, or you can download the CircuitPython BIN file directly using the tool in the same manner as the bootloader.

Option 3 - Use esptool to load BIN file

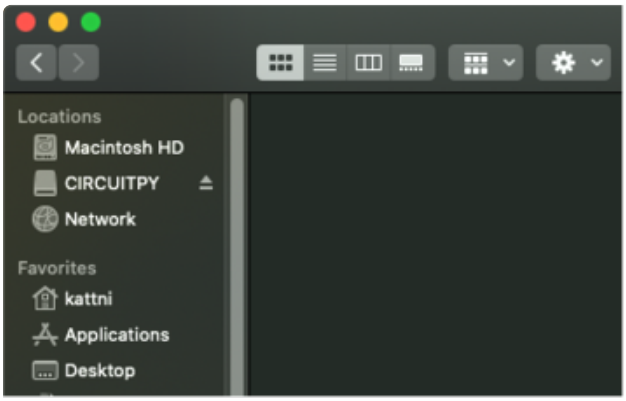
For more advanced users, you can upload with **esptool** to the ROM (hardware) bootloader instead!



Follow the initial steps found in the Run esptool and check connection section of the Install UF2 Bootloader page (https://adafru.it/RLc) to verify your environment is set up, your board is successfully connected, and which port it's using.

In the final command to write a binary file to the board, replace the port with your port, and replace "firmware.bin" with the the file you downloaded above.

The output should look something like the output in the image.



Press reset to exit the bootloader.

Your **CIRCUITPY** drive should appear!

You're all set! Go to the next pages.

CircuitPython Internet Test

One of the great things about the ESP32 is the built-in WiFi capabilities. This page covers the basics of getting connected using CircuitPython.

The first thing you need to do is update your **code.py** to the following. Click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, and copy the **entire lib folder** and the **code.py** file to your **CIRCUITPY** drive.

```
# SPDX-FileCopyrightText: 2020 Brent Rubell for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import os
import ipaddress
import ssl
import wifi
```

```
import socketpool
import adafruit_requests

# URLs to fetch from
TEXT_URL = "http://wifitest.adafruit.com/testwifi/index.html"
JSON_QUOTES_URL = "https://www.adafruit.com/api/quotes.php"
JSON_STARS_URL = "https://api.github.com/repos/adafruit/circuitpython"

print("ESP32-S2 WebClient Test")

print(f"My MAC address: {[hex(i) for i in wifi.radio.mac_address]}")

print("Available WiFi networks:")
for network in wifi.radio.start_scanning_networks():
    print("\t%s\t\tRSSI: %d\tChannel: %d" % (str(network.ssid, "utf-8"),
                                             network.rssi, network.channel))
wifi.radio.stop_scanning_networks()

print(f"Connecting to {os.getenv('CIRCUITPY_WIFI_SSID')}")
wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
os.getenv("CIRCUITPY_WIFI_PASSWORD"))
print(f"Connected to {os.getenv('CIRCUITPY_WIFI_SSID')}")
print(f"My IP address: {wifi.radio.ipv4_address}")

ping_ip = ipaddress.IPv4Address("8.8.8.8")
ping = wifi.radio.ping(ip=ping_ip)

# retry once if timed out
if ping is None:
    ping = wifi.radio.ping(ip=ping_ip)

if ping is None:
    print("Couldn't ping 'google.com' successfully")
else:
    # convert s to ms
    print(f"Pinging 'google.com' took: {ping * 1000} ms")

pool = socketpool.SocketPool(wifi.radio)
requests = adafruit_requests.Session(pool, ssl.create_default_context())

print(f"Fetching text from {TEXT_URL}")
response = requests.get(TEXT_URL)
print("-" * 40)
print(response.text)
print("-" * 40)

print(f"Fetching json from {JSON_QUOTES_URL}")
response = requests.get(JSON_QUOTES_URL)
print("-" * 40)
print(response.json())
print("-" * 40)

print()

print(f"Fetching and parsing json from {JSON_STARS_URL}")
response = requests.get(JSON_STARS_URL)
print("-" * 40)
print(f"CircuitPython GitHub Stars: {response.json()['stargazers_count']}")
print("-" * 40)

print("Done")
```

Your **CIRCUITPY** drive should resemble the following.

To get connected, the next thing you need to do is update the **settings.toml** file.

The settings.toml 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 **settings.toml** file, that is on your **CIRCUITPY** drive, to hold secret/private/custom data. That way you can share your main project without worrying about accidentally sharing private stuff.

If you have a fresh install of CircuitPython on your board, the initial **settings.toml** file on your **CIRCUITPY** drive is empty.

To get started, you can update the **settings.toml** on your **CIRCUITPY** drive to contain the following code.

```
# SPDX-FileCopyrightText: 2023 Adafruit Industries
#
# SPDX-License-Identifier: MIT

# This is where you store the credentials necessary for your code.
# The associated demo only requires WiFi, but you can include any
# credentials here, such as Adafruit IO username and key, etc.
CIRCUITPY_WIFI_SSID = "your-wifi-ssid"
CIRCUITPY_WIFI_PASSWORD = "your-wifi-password"
```

This file should contain a series of Python variables, each assigned to a string. Each variable should describe what it represents (say `wifi_ssid`), followed by an = (equals sign), followed by the data in the form of a Python string (such as `"my-wifi-password"` including the quote marks).

At a minimum you'll need to add/update your WiFi SSID and WiFi password, so do that now!

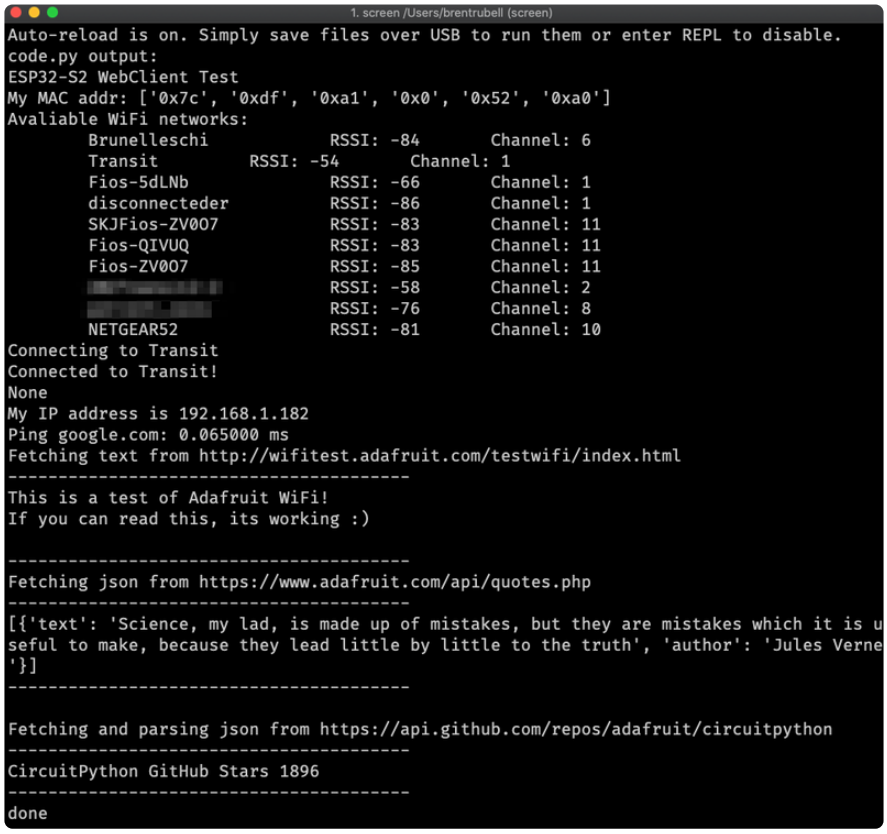
As you make projects you may need more tokens and keys, just add them one line at a time. See for example other tokens such as one for accessing GitHub or the Hackaday API. Other non-secret data like your timezone can also go here.

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 **settings.toml** - keep that out of GitHub, Discord or other project-sharing sites.

Don't share your settings.toml file! It has your passwords and API keys in it!

If you connect to the serial console, you should see something like the following:



In order, the example code...

Checks the ESP32's MAC address.

```
print(f"My MAC address: {[hex(i) for i in wifi.radio.mac_address]}")
```

Performs a scan of all access points and prints out the access point's name (SSID), signal strength (RSSI), and channel.

```
print("Available WiFi networks:")
for network in wifi.radio.start_scanning_networks():
    print("\t%s\t\tRSSI: %d\tChannel: %d" % (str(network.ssid, "utf-8"),
                                              network.rssi, network.channel))
wifi.radio.stop_scanning_networks()
```

Connects to the access point you defined in the **settings.toml** file, and prints out its local IP address.

```
print(f"Connecting to {os.getenv('WIFI_SSID')}")
wifi.radio.connect(os.getenv("WIFI_SSID"), os.getenv("WIFI_PASSWORD"))
print(f"Connected to {os.getenv('WIFI_SSID')}")
print(f"My IP address: {wifi.radio.ipv4_address}")
```

Attempts to ping a Google DNS server to test connectivity. If a ping fails, it returns `None`. Initial pings can sometimes fail for various reasons. So, if the initial ping is successful (`is not None`), it will print the echo speed in ms. If the initial ping fails, it will try one more time to ping, and then print the returned value. If the second ping fails, it will result in `"Ping google.com: None ms"` being printed to the serial console. Failure to ping does not always indicate a lack of connectivity, so the code will continue to run.

```
ping_ip = ipaddress.IPv4Address("8.8.8.8")
ping = wifi.radio.ping(ip=ping_ip) * 1000
if ping is not None:
    print(f"Ping google.com: {ping} ms")
else:
    ping = wifi.radio.ping(ip=ping_ip)
    print(f"Ping google.com: {ping} ms")
```

The code creates a socketpool using the wifi radio's available sockets. This is performed so we don't need to re-use sockets. Then, it initializes a a new instance of the requests (https://adafru.it/E9o) interface - which makes getting data from the internet really really easy.

```
pool = socketpool.SocketPool(wifi.radio)
requests = adafruit_requests.Session(pool, ssl.create_default_context())
```

To read in plain-text from a web URL, call `requests.get` - you may pass in either a http, or a https url for SSL connectivity.

```
print(f"Fetching text from {TEXT_URL}")
response = requests.get(TEXT_URL)
print("-" * 40)
print(response.text)
print("-" * 40)
```

Requests can also display a JSON-formatted response from a web URL using a call to `requests.get`.

```
print(f"Fetching json from {JSON_QUOTES_URL}")
response = requests.get(JSON_QUOTES_URL)
print("-" * 40)
print(response.json())
print("-" * 40)
```

Finally, you can fetch and parse a JSON URL using `requests.get`. This code snippet obtains the `stargazers_count` field from a call to the GitHub API.

```
print(f"Fetching and parsing json from {JSON_STARS_URL}")
response = requests.get(JSON_STARS_URL)
print("-" * 40)
print(f"CircuitPython GitHub Stars: {response.json()['stargazers_count']}")
print("-" * 40)
```

OK you now have your ESP32 board set up with a proper **settings.toml** file and can connect over the Internet. If not, check that your **settings.toml** file has the right SSID and password and retrace your steps until you get the Internet connectivity working!

Getting The Date & Time

A very common need for projects is to know the current date and time. Especially when you want to deep sleep until an event, or you want to change your display based on what day, time, date, etc. it is

Determining the correct local time is really really hard. There are various time zones, Daylight Savings dates, leap seconds, etc. Trying to get NTP time and then back-calculating what the local time is, is extraordinarily hard on a microcontroller just isn't worth the effort and it will get out of sync as laws change anyways.

For that reason, we have the free adafruit.io time service. **Free for anyone with a free adafruit.io account.** You do need an account because we have to keep accidentally mis-programmed-board from overwhelming adafruit.io and lock them out temporarily. Again, it's free!

There are other services like WorldTimeAPI, but we don't use those for our guides because they are nice people and we don't want to accidentally overload their site. Also, there's a chance it may eventually go down or also require an account.

Step 1) Make an Adafruit account

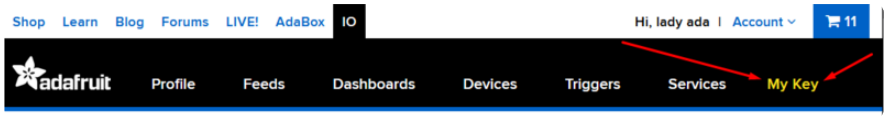
It's free! Visit https://accounts.adafruit.com/ (https://adafru.it/dyy) to register and make an account if you do not already have one

Step 2) Sign into Adafruit IO

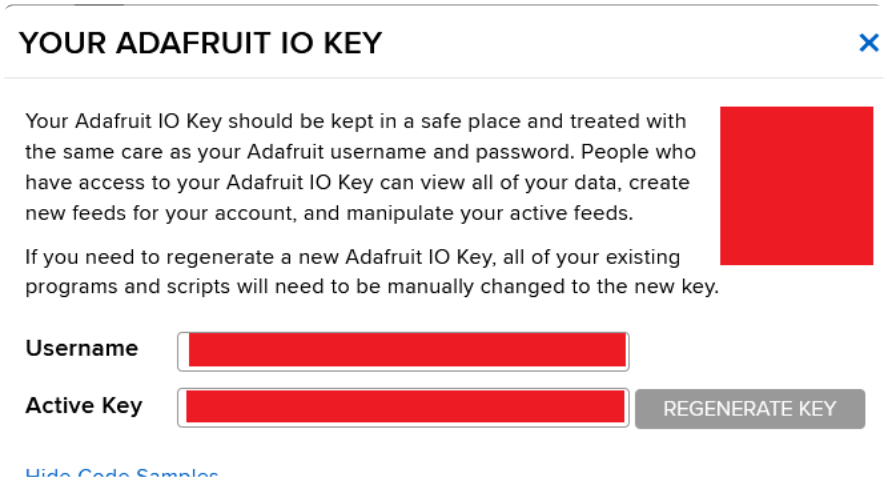
Head over to io.adafruit.com (https://adafru.it/fsU) and click **Sign In** to log into IO using your Adafruit account. It's free and fast to join.

Step 3) Get your Adafruit IO Key

Click on **My Key** in the top bar



You will get a popup with your **Username** and **Key** (In this screenshot, we've covered it with red blocks)



Go to the **settings.toml** file on your CIRCUITPY drive and add three lines for `AIO_USERNAME`, `ADAFRUIT_AIO_KEY` and `TIMEZONE` so you get something like the following:

```
# 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

CIRCUITPY_WIFI_SSID = "your-wifi-ssid"
CIRCUITPY_WIFI_PASSWORD = "your-wifi-password"
ADAFRUIT_AIO_USERAME = "your-adafruit-io-username"
ADAFTUIT_AIO_KEY = "your-adafruit-io-key"
```

```
# Timezone names from http://worldtimeapi.org/timezones
TIMEZONE="America/New_York"
```

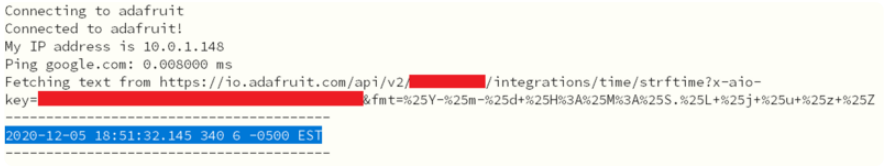
The timezone is optional, if you don't have that entry, adafruit.io will guess your timezone based on geographic IP address lookup. You can visit http://worldtimeapi.org/timezones (https://adafru.it/EcP) to see all the time zones available (even though we do not use Worldtime for time-keeping, we do use the same time zone table).

Step 4) Upload Test Python Code

This code is like the Internet Test code from before, but this time it will connect to adafruit.io and get the local time

```
import ipaddress
import os
import ssl
import wifi
import socketpool
import adafruit_requests
import secrets


# Get our username, key and desired timezone
ssid = os.getenv("CIRCUITPY_WIFI_SSID")
password = os.getenv("CIRCUITPY_WIFI_PASSWORD")
aio_username = os.getenv("ADAFRUIT_AIO_USERNAME")
aio_key = os.getenv("ADAFRUIT_AIO_KEY")
timezone = os.getenv("TIMEZONE")
TIME_URL = f"https://io.adafruit.com/api/v2/{aio_username}/integrations/time/
strftime?x-aio-key={aio_key}&amp;tz={timezone}")
TIME_URL += "&amp;fmt=%25Y-%25m-%25d+%25H%3A%25M%3A%25S.%25L+%25j+%25u+%25z+%25Z"

print("ESP32-S2 Adafruit IO Time test")

print("My MAC addr:", [hex(i) for i in wifi.radio.mac_address])

print("Available WiFi networks:")
for network in wifi.radio.start_scanning_networks():
    print("\t%s\t\tRSSI: %d\tChannel: %d" % (str(network.ssid, "utf-8"),
            network.rssi, network.channel))
wifi.radio.stop_scanning_networks()

print("Connecting to", ssid)
wifi.radio.connect(ssid, password)
print(f"Connected to {ssid}!")
print("My IP address is", wifi.radio.ipv4_address)

ipv4 = ipaddress.ip_address("8.8.4.4")
print("Ping google.com:", wifi.radio.ping(ipv4), "ms")

pool = socketpool.SocketPool(wifi.radio)
requests = adafruit_requests.Session(pool, ssl.create_default_context())

print("Fetching text from", TIME_URL)
response = requests.get(TIME_URL)
print("-" * 40)
print(response.text)
print("-" * 40)
```

After running this, you will see something like the below text. We have blocked out the part with the secret username and key data!



Note at the end you will get the date, time, and your timezone! If so, you have correctly configured your **settings.toml** and can continue to the next steps!

FunHouse-Specific CircuitPython Libraries

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

Get Latest Adafruit CircuitPython Bundle

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

Download the latest Library Bundle from circuitpython.org

https://adafru.it/ENC

Download the **adafruit-circuitpython-bundle-version-mpy-*.zip** bundle zip file, and unzip a folder of the same name. Inside you'll find a **lib** folder. The entire collection of libraries is too large to fit on the **CIRCUITPY** drive. Therefore, you'll need to copy the necessary libraries to your board individually.

At a minimum, the following libraries are required. Copy the following folders or .mpy files to the **lib** folder on your **CIRCUITPY** drive. **If the library is a folder, copy the entire folder** to the **lib** folder on your board.

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!

Library folders (copy the whole folder over to lib):

- **adafruit_funhouse** - This is a helper library designed for using all of the features of the FunHouse, including networking, buttons, DotStars, etc.

- **adafruit_portalbase** - This library is the base library that adafruit_funhouse is built on top of.
- **adafruit_bitmap_font** - There is fancy font support, and it's easy to make new fonts. This library reads and parses font files.
- **adafruit_display_text** - This library displays text on the screen.
- **adafruit_io** - This library helps connect the FunHouse to our free data logging and viewing service
- **adafruit_minimqtt** - MQTT library required for communicating with the MQTT Server

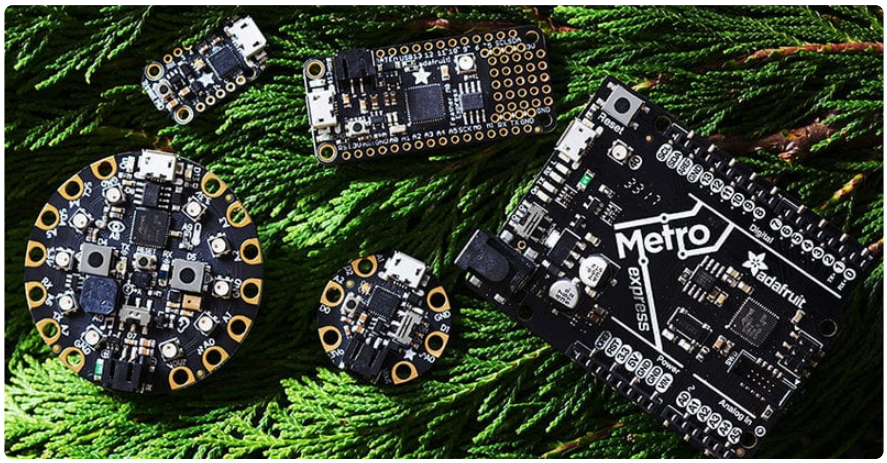
Library files:

- **adafruit_requests.mpy** - This library allows us to perform HTTP requests and get responses back from servers. GET/POST/PUT/PATCH - they're all in here!
- **adafruit_fakerequests.mpy** - This library allows you to create fake HTTP requests by using local files.
- **adafruit_miniqr.mpy** - QR creation library lets us add easy-to-scan 2D barcodes to the E-Ink display
- **adafruit_dotstar.mpy** - This library is used to control the onboard DotStars.
- **simpleio.mpy** - This library is used for tone generation.
- **adafruit_ahtx0.mpy** - This is used for the Humidity and Temperature Sensor
- **adafruit_dps310.mpy**- This is used for the Barometric Pressure Sensor

Secrets

Even if you aren't planning to go online with your FunHouse, you'll need to have a **settings.toml** file in the root directory (top level) of your **CIRCUITPY** drive. If you do not intend to connect to wireless, it does not need to have valid data in it. Here's more info on the secrets.toml file (https://adafru.it/P3b).

Welcome To CircuitPython



So, you've got a new **CircuitPython compatible board**. You plugged it in. Maybe it showed up as a disk drive called **CIRCUITPY**. Maybe it didn't! Either way, you need to know where to go from here. Well, this guide has you covered!

This guide will get you started with CircuitPython!

There are many amazing things about your new board. One of them is the ability to run CircuitPython. You may have seen that name on the Adafruit site (https://adafru.it/dAR) somewhere. Not sure what it is? This guide can help!

"But I've never coded in my life. There's no way I do it!" You absolutely can! CircuitPython is designed to help you learn from the ground up. If you're new to everything, this is the place to start!

This guide will walk you through how to get started with CircuitPython. You'll learn how to install CircuitPython, get updated to the newest version of CircuitPython, setup a serial connection, and edit your code. You'll learn some basics of how CircuitPython works, and about the CircuitPython libraries. You'll also find a list of frequently asked questions, and a series of troubleshooting steps if you run into any issues.

Welcome to CircuitPython!

Installing the 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

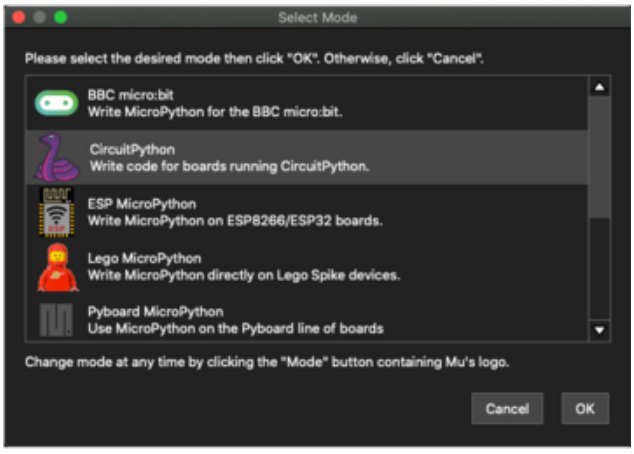
Download Mu from https://codewith.mu (https://adafru.it/Be6).

Click the **Download** link for downloads and installation instructions.

Click **Start Here** to find a wealth of other information, including extensive tutorials and and how-to's.

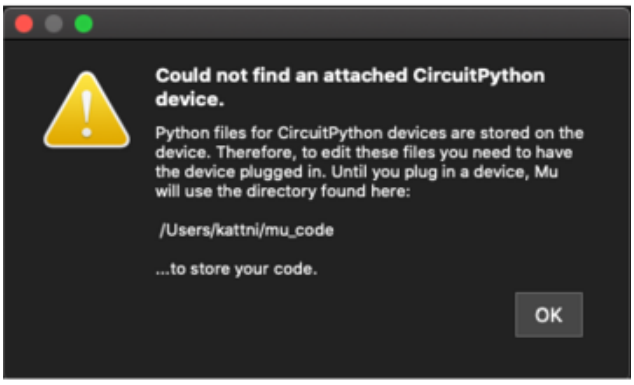
Windows users: due to the nature of MSI installers, please remove old versions of Mu before installing the latest version.

Starting Up 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 on startup, so if you do not have a CircuitPython board plugged in with a **CIRCUITPY** drive available, Mu will inform you where it will store any code you save until you plug in a board.

To avoid this warning, plug in a board and ensure that the **CIRCUITPY** drive is mounted before starting Mu.

Using 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

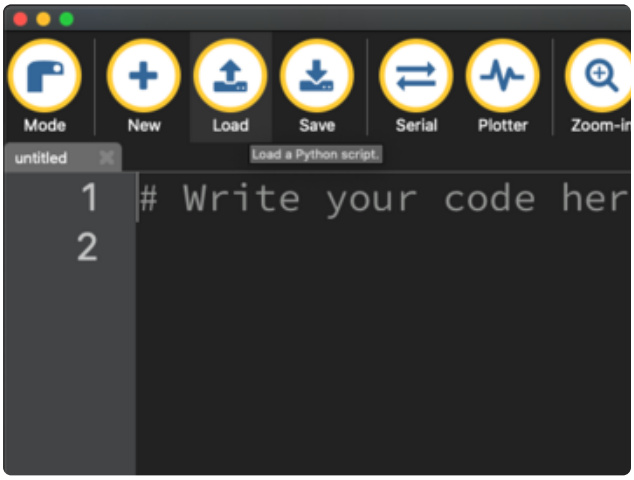
One of the best things about CircuitPython is how simple it is to get code up and running. This section covers how to create and edit your first CircuitPython program.

To create and edit code, all you'll need is an editor. There are many options. **Adafruit strongly recommends 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 a number of other editors that work quite well. The Recommended Editors page (https://adafru.it/Vue) has more details. Otherwise, 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 was formerly not a problem on macOS, but see the warning below.)

macOS Sonoma (14.x) introduced a bug that delays writes to small drives such as CIRCUITPY drives. This causes errors when saving files to CIRCUITPY. For a workaround, see https://learn.adafruit.com/welcome-to-circuitpython/troubleshooting#macos-sonoma-14-dot-x-disk-errors-writing-to-circuitpy-3160304

Creating Code



Installing CircuitPython generates a **code.py** file on your **CIRCUITPY** drive. To begin your own program, open your editor, and load the **code.py** file from the **CIRCUITPY** drive.

If you are using Mu, click the **Load** button in the button bar, navigate to the **CIRCUITPY** drive, and choose **code.py**.

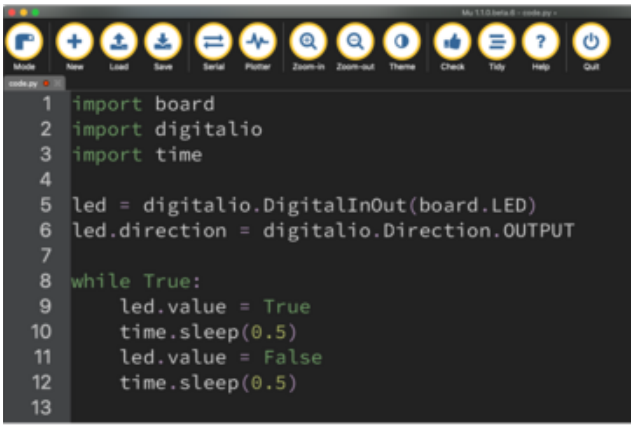
Copy and paste the following code into your editor:

```
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)
```

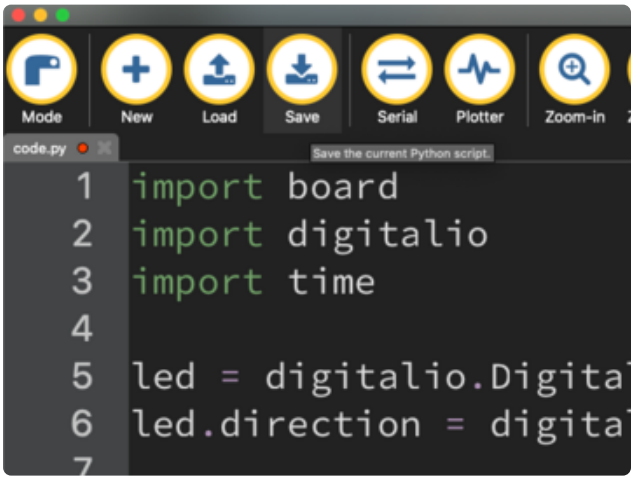
The KB2040, QT Py , Qualia, and the Trinkeys do not have a built-in little red LED! There is an addressable RGB NeoPixel LED. The above example will NOT work on the KB2040, QT Py, Qualia, or the Trinkeys!

If you're using a KB2040, QT Py, Quaila, or a Trinkey, or any other board without a single-color LED that can blink, please download the NeoPixel blink example (https://adafru.it/UDU).

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.



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



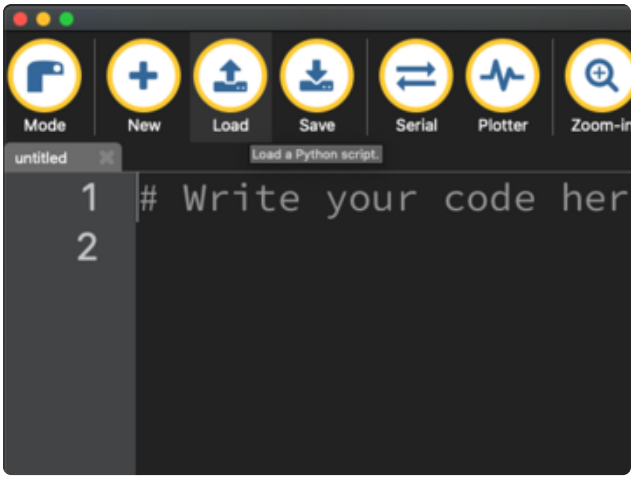
Save the **code.py** file on your **CIRCUITPY** drive.

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

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

On most boards you'll find a tiny red LED. On the ItsyBitsy nRF52840, you'll find a tiny blue LED. On QT Py M0, QT Py RP2040, Qualia, and the Trinkey series, you will find only an RGB NeoPixel LED.

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 one warning before you continue...

Don't click reset or unplug your board!

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. 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 couple of ways to avoid filesystem corruption.

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

Check out the Recommended Editors page (https://adafru.it/Vue) for details on different editing options.

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. You'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?

You 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 **code.py** is the recommended name for 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 (and Python) looks like this:

`print("Hello, world!")`

This line in your code.py would result in:

`Hello, world!`

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

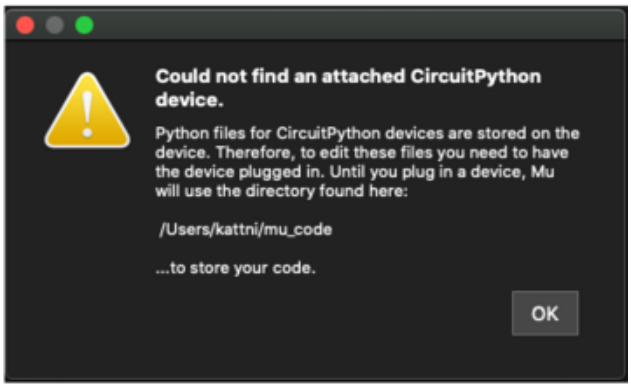
The serial console receives output from your CircuitPython board sent over USB and displays it so you can see it. This is necessary when you've included a print statement in your code and you'd like to see what you printed. It is also helpful for

troubleshooting errors, because your board will send errors and the serial console will display those too.

The serial console requires an editor that has a built in terminal, or a separate terminal program. A terminal is a program that gives you a text-based interface to perform various tasks.

Are you using Mu?

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



First, make sure your CircuitPython board is plugged in.

If you open Mu without a board plugged in, you may encounter the error seen here, letting you know no CircuitPython board was found and indicating where your code will be stored until you plug in a board.

If you are using Windows 7, make sure you installed the drivers (https://adafru.it/VuB).

Once you've opened Mu with your board plugged in, look for the **Serial** button in the button bar and click it.

The Mu window will split in two, horizontally, and display the serial console at the bottom.

If nothing appears in the serial console, it may mean your code is done running or has no print statements in it. Click into the serial console part of Mu, and press CTRL+D to reload.

Serial Console Issues or Delays on Linux

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 `modemmanager`, type the following command at a shell:

```
sudo apt purge modemmanager
```

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 the Advanced Serial Console on Linux (https://adafru.it/VAO) for details on how to add yourself to the right group.

Using Something Else?

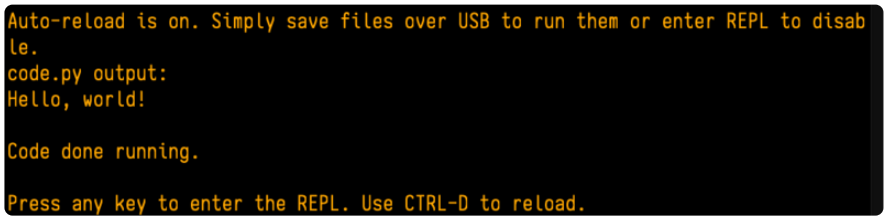
If you're not using Mu to edit, are using or if for some reason you are not a fan of its built in serial console, you can run the serial console from a separate program.

Windows requires you to download a terminal program. Check out the Advanced Serial Console on Windows page for more details. (https://adafru.it/AAH)

MacOS has Terminal built in, though there are other options available for download. Check the Advanced Serial Console on Mac page for more details. (https://adafru.it/AAI)

Linux has a terminal program built in, though other options are available for download. Check the Advanced Serial Console on Linux page for more details. (https://adafru.it/VAO)

Once connected, you'll see something like the following.



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, you're going to edit it to create some output.

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

```
import board
import digitalio
import time

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

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

```
    time.sleep(1)
    led.value = False
    time.sleep(1)
```

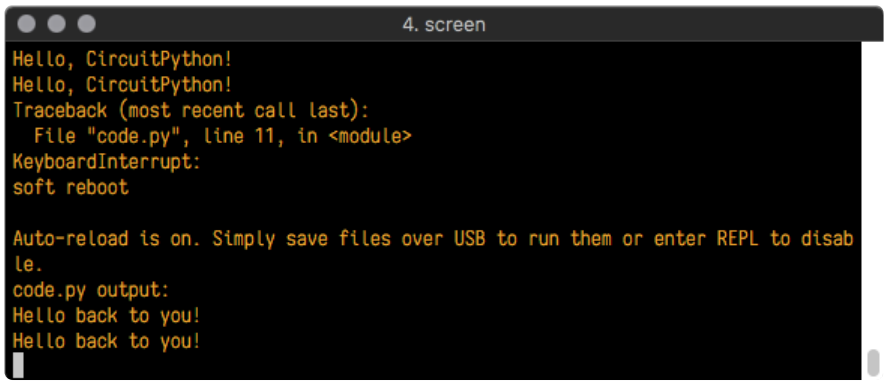
Save your file.

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

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

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.LED)
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!



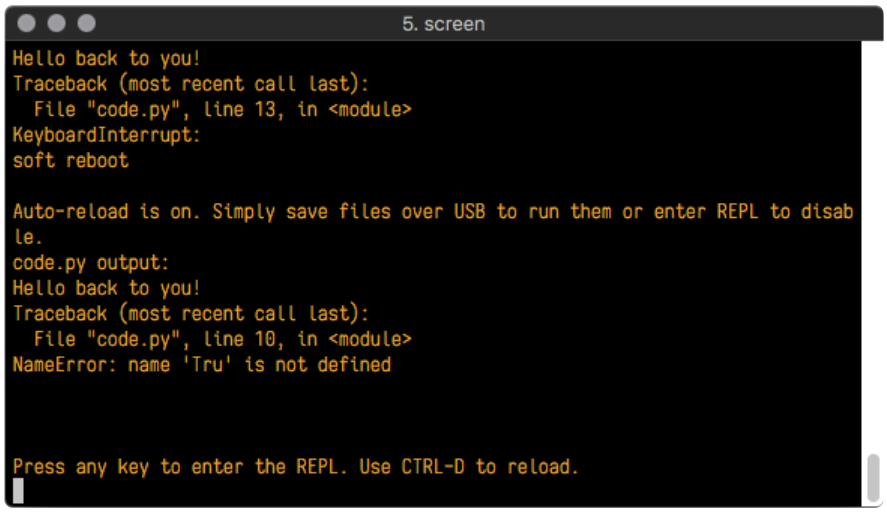
The `Traceback (most recent call last):` is telling you the last thing your board was doing before you saved your file. This is normal behavior and will happen every time the board resets. This is really handy for troubleshooting. Let's introduce an error so you can see how it is used.

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

```
import board
import digitalio
import time

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

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

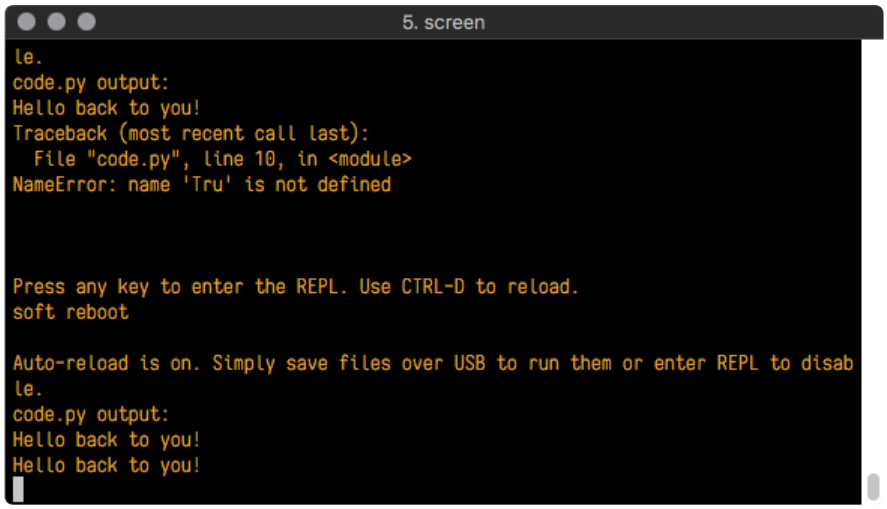
Save your file. You will notice that your red LED will stop blinking, and you may have a colored status LED blinking at you. This is because the code is no longer correct and can no longer run properly. You need to fix it!

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, which is called "print debugging". Essentially, 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

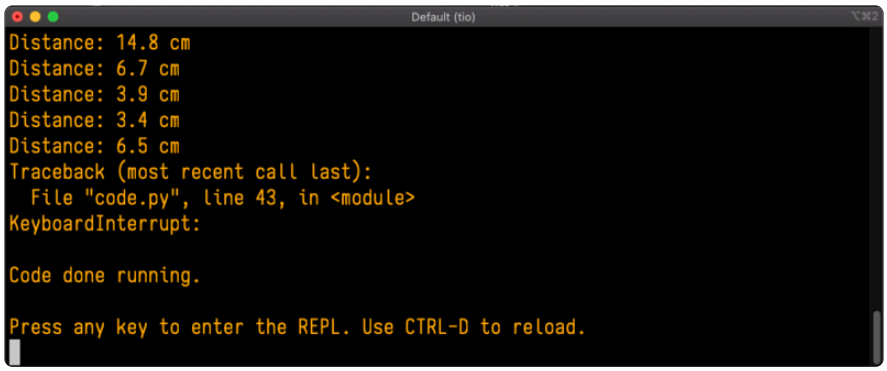
The other feature of the serial connection is the **R**ead-**E**valuate-**P**rint-**L**oop, or REPL. The REPL allows you to enter individual lines of code and have them run immediately. It's really handy if you're running into trouble with a particular program and can't figure out why. It's interactive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

Entering the REPL

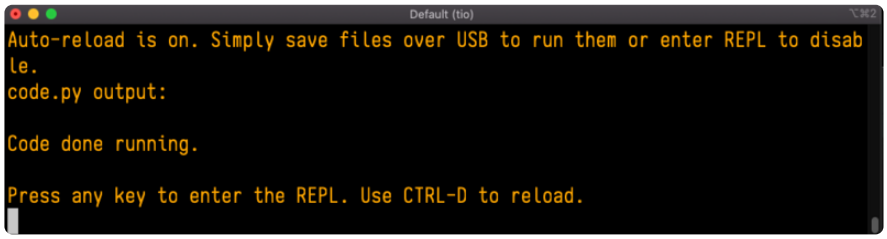
To use the REPL, you first need to be connected to the serial console. Once that connection has been established, you'll want to press **CTRL+C**.

If there is code running, in this case code measuring distance, 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.

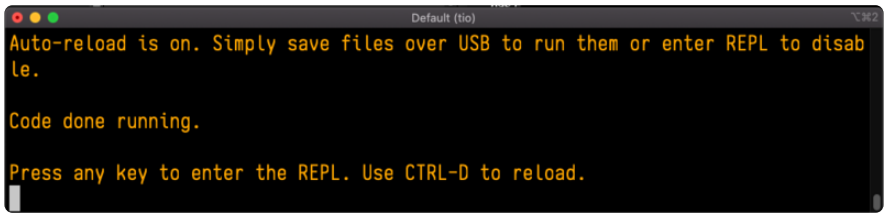
The `Traceback (most recent call last):` is telling you the last thing your board was doing before you pressed Ctrl + C and interrupted it. The `KeyboardInterrupt` is you pressing CTRL+C. This information can be handy when troubleshooting, but for now, don't worry about it. Just note that it is expected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



If your **code.py** file is empty or does not contain a loop, it will show an empty output and `Code done running.`. There is no information about what your board was doing before you interrupted it because there is no code running.



If you have no **code.py** on your **CIRCUITPY** drive, you will enter the REPL immediately after pressing CTRL+C. Again, there is no information about what your board was doing before you interrupted it because there is no code running.



Regardless, 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 7.0.0 on 2021-10-26; Adafruit Feather RP2040 with rp2040
```

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.

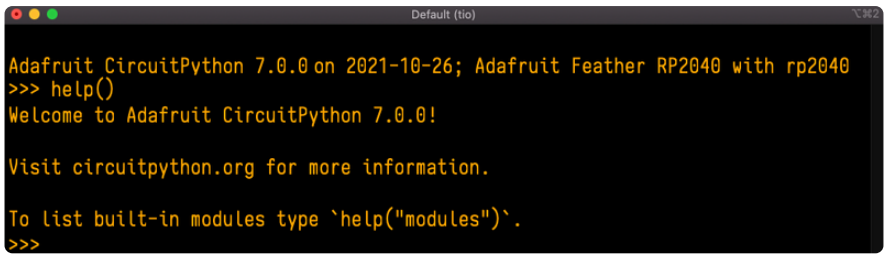
```
>>>
```

Interacting with the REPL

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

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

Then press enter. You should then see a message.



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

```
>>> help("modules")
__main__          board             micropython       storage
_bleio            builtins          msgpack           struct
adafruit_bus_device                 busio             neopixel_write    supervisor
adafruit_pixelbuf collections       onewireio         synthio
aesio             countio           os                sys
alarm             digitalio         paralleldisplay   terminalio
analogio          displayio         pulseio           time
array             errno             pwmio             touchio
atexit            fontio            qrio              traceback
audiobusio        framebufferio     rainbowio         ulab
audiocore         gc                random            usb_cdc
audiomixer        getpass           re                usb_hid
audiomp3          imagecapture      rgbmatrix         usb_midi
audiopwmio        io                rotaryio          vectorio
binascii          json              rp2pio            watchdog
bitbangio         keypad            rtc
bitmaptools       math              sdcardio
bitops            microcontroller   sharpdisplay
Plus any modules on the filesystem
>>>
```

This is a list of all the core modules built into CircuitPython, including `board`. Remember, `board` contains all of the pins on the board that you can use in your code. From the REPL, you are able to see that list!

Type `import board` into the REPL and press enter. It'll go to a new prompt. It might look like nothing happened, but that's not the case! If you recall, the `import` statement simply tells the code to expect to do something with that module. In this case, it's telling the REPL that you plan to do something with that module.

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

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

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

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

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

Every programmer in every programming language starts with a piece of code that says, "Hello, World." You'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. 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 modules are available and explore those modules.

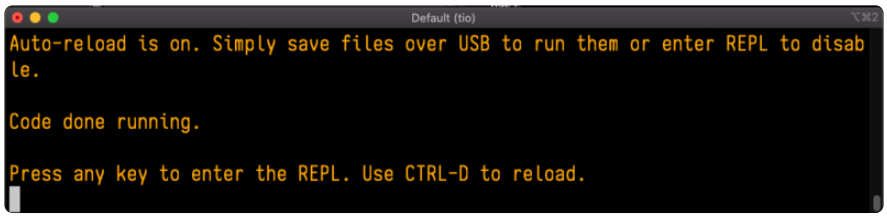
Try typing more into the REPL to see what happens!

Everything typed into the REPL is ephemeral. Once you reload the REPL or return to the serial console, nothing you typed will be retained in any memory space. So be sure to save any desired code you wrote somewhere else, or you'll lose it when you leave the current REPL instance!

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!



Advanced Serial Console on Windows

Windows 7 and 8.1

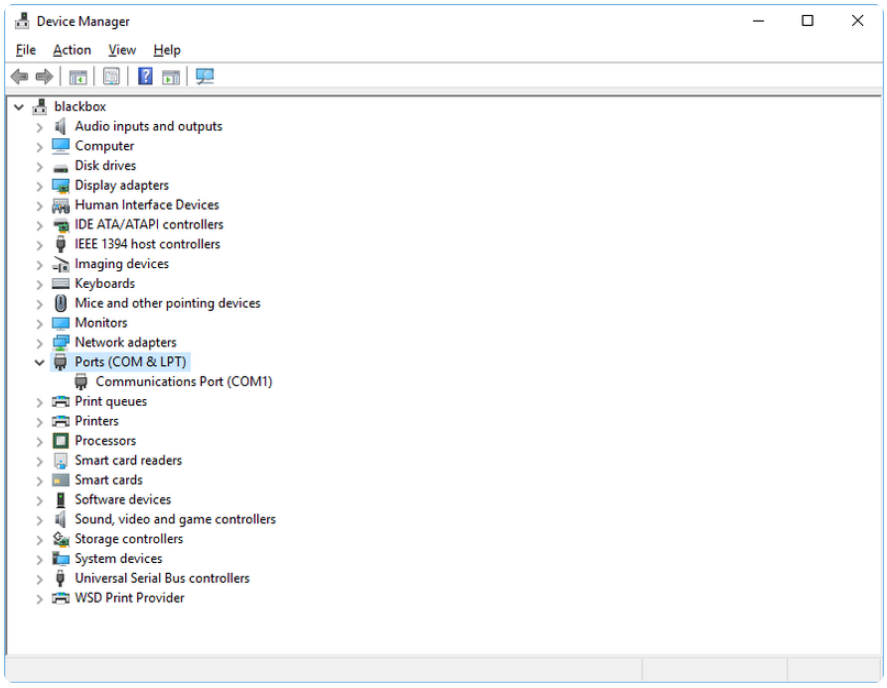
If you're using Windows 7 (or 8 or 8.1), you'll need to install drivers. See the Windows 7 and 8.1 Drivers page (https://adafru.it/VuB) for details. You will not need to install drivers on Mac, Linux or Windows 10.

You are strongly encouraged to upgrade to Windows 10 if you are still using Windows 7 or Windows 8 or 8.1. Windows 7 has reached end-of-life and no longer receives security updates. A free upgrade to Windows 10 is still available (https://adafru.it/RWc).

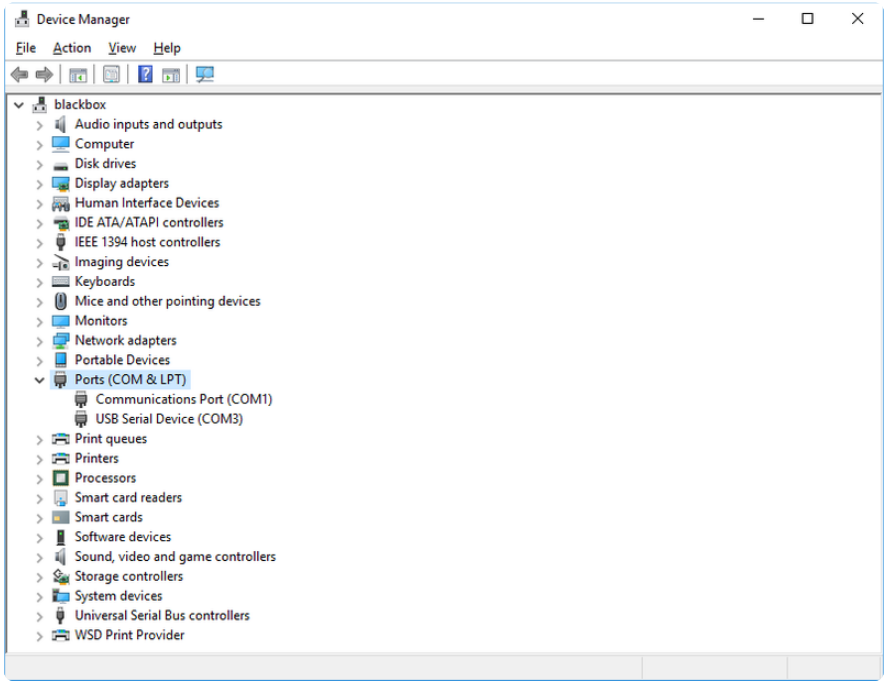
What's the COM?

First, you'll want to find out which serial port your board is using. When you plug your board in to USB on your computer, it connects to a serial port. The port is like a door through which your board can communicate with your computer using USB.

You'll use Windows Device Manager to determine which port the board is using. The easiest way to determine which port the board is using is to first check **without** the board plugged in. Open Device Manager. Click on Ports (COM & LPT). You should find something already in that list with (COM#) after it where # is a number.



Now plug in your board. The Device Manager list will refresh and a new item will appear under Ports (COM & LPT). You'll find a different (COM#) after this item in the list.



Sometimes the item will refer to the name of the board. Other times it may be called something like USB Serial Device, as seen in the image above. Either way, there is a new (COM#) following the name. This is the port your board is using.

Install Putty

If you're using Windows, you'll need to download a terminal program. You're going to use PuTTY.

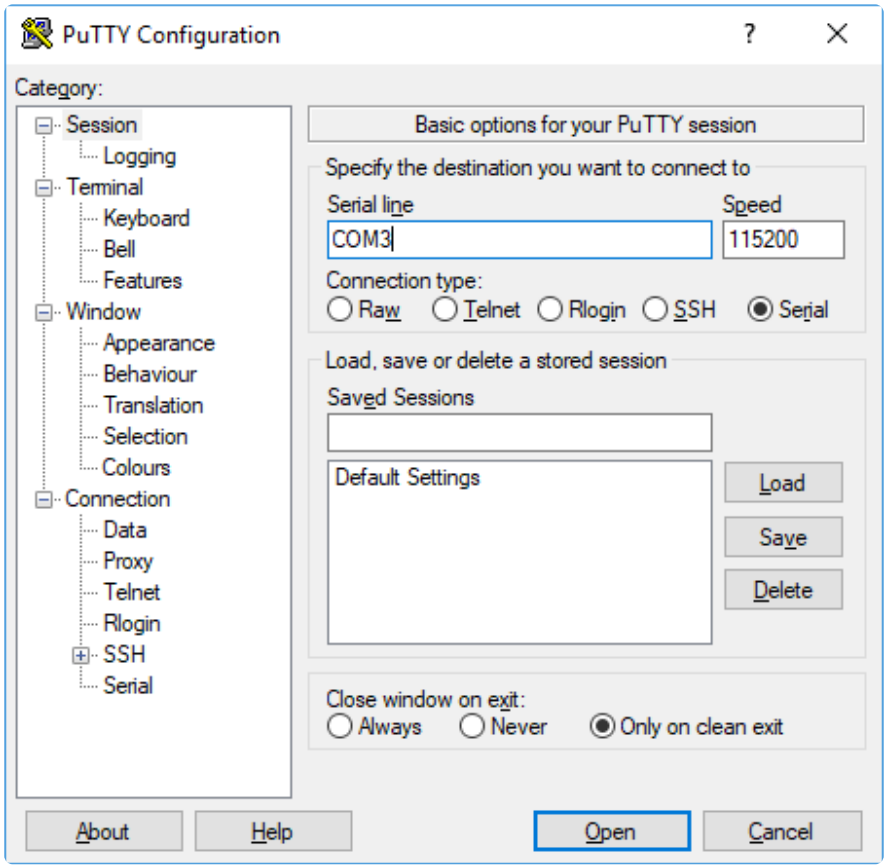
The first thing to do is download the latest version of PuTTY (https://adafru.it/Bf1). You'll want to download the Windows installer file. It is most likely that you'll need the 64-bit version. Download the file and install the program on your machine. If you run into issues, you can try downloading the 32-bit version instead. However, the 64-bit version will work on most PCs.

Now you need to open PuTTY.

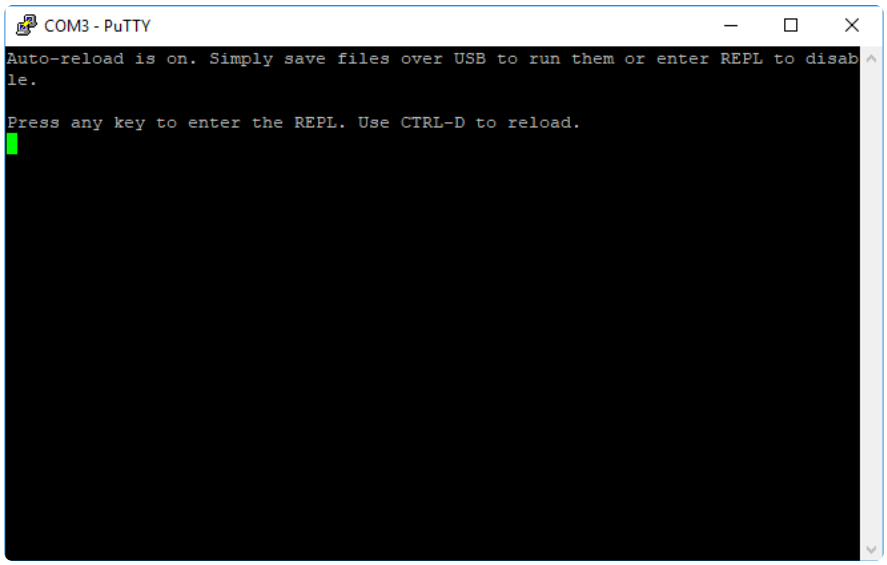
- Under **Connection type:** choose the button next to **Serial**.
- In the box under **Serial line**, enter the serial port you found that your board is using.
- In the box under **Speed**, enter 115200. This called the baud rate, which is the speed in bits per second that data is sent over the serial connection. For boards

with built in USB it doesn't matter so much but for ESP8266 and other board with a separate chip, the speed required by the board is 115200 bits per second. So you might as well just use 115200!

If you want to save those settings for later, use the options under **Load, save or delete a stored session.** Enter a name in the box under **Saved Sessions**, and click the **Save** button on the right.



Once your settings are entered, you're ready to connect to the serial console. Click "Open" at the bottom of the window. A new window will open.



If no code is running, the window will either be blank or will look like the window above. Now you're ready to see the results of your code.

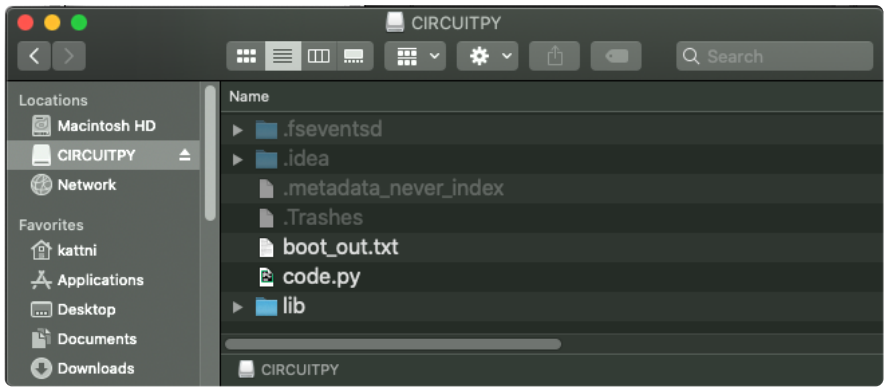
Great job! You've connected to the serial console!

CircuitPython Libraries

As CircuitPython development continues and there are new releases, Adafruit 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 great 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 an excellent reference for how it all should work. In Python terms, you can place our library files in the **lib** directory because it's part of the Python path by default.

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

The bundle and the library releases on GitHub 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.

Due to the regular updates and space constraints, Adafruit does not ship boards with the entire bundle. Therefore, you will need to load the libraries you need when you begin working with your board. You can find example code in the guides for your board that depends on external libraries.

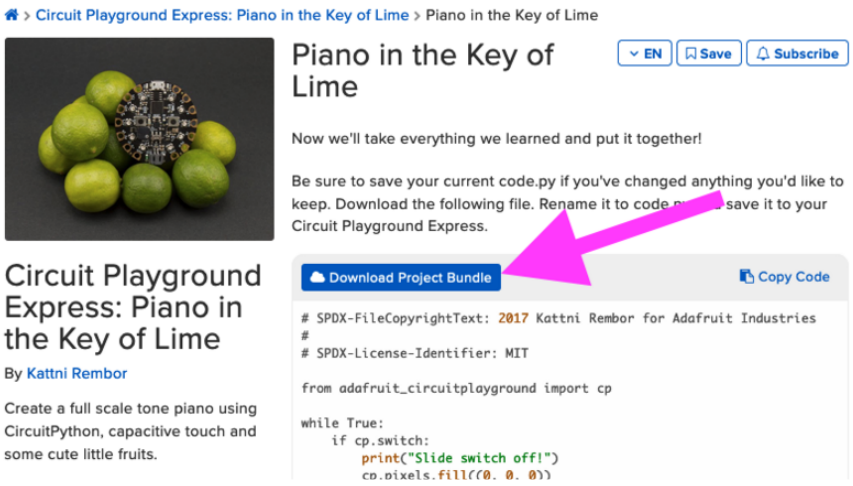
Either way, as you start to explore CircuitPython, you'll want to know how to get libraries on board.

The Adafruit Learn Guide Project Bundle

The quickest and easiest way to get going with a project from the Adafruit Learn System is by utilising the Project Bundle. Most guides now have a **Download Project Bundle** button available at the top of the full code example embed. This button downloads all the necessary files, including images, etc., to get the guide project up and running. Simply click, open the resulting zip, copy over the right files, and you're good to go!

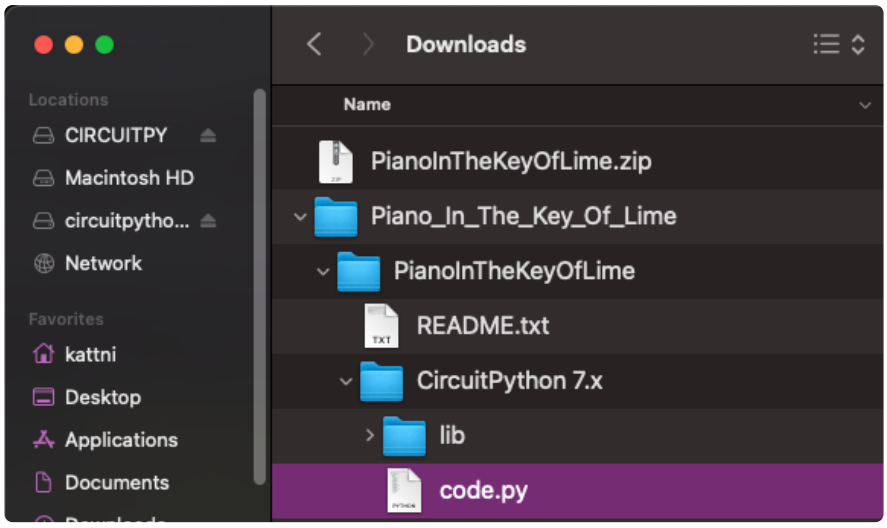
The first step is to find the Download Project Bundle button in the guide you're working on.

The Download Project Bundle button is only available on full demo code embedded from GitHub in a Learn guide. Code snippets will NOT have the button available.



When you copy the contents of the Project Bundle to your CIRCUITPY drive, it will replace all the existing content! If you don't want to lose anything, ensure you copy your current code to your computer before you copy over the new Project Bundle content!

The Download Project Bundle button downloads a zip file. This zip contains a series of directories, nested within which is the **code.py**, any applicable assets like images or audio, and the **lib/** folder containing all the necessary libraries. The following zip was downloaded from the Piano in the Key of Lime guide.



The Piano in the Key of Lime guide was chosen as an example. That guide is specific to Circuit Playground Express, and cannot be used on all boards. Do not

expect to download that exact bundle and have it work on your non-CPX microcontroller.

When you open the zip, you'll find some nested directories. Navigate through them until you find what you need. You'll eventually find a directory for your CircuitPython version (in this case, 7.x). In the version directory, you'll find the file and directory you need: **code.py** and **lib/**. Once you find the content you need, you can copy it all over to your **CIRCUITPY** drive, replacing any files already on the drive with the files from the freshly downloaded zip.

In some cases, there will be other files such as audio or images in the same directory as code.py and lib/. Make sure you include all the files when you copy things over!

Once you copy over all the relevant files, the project should begin running! If you find that the project is not running as expected, make sure you've copied ALL of the project files onto your microcontroller board.

That's all there is to using the Project Bundle!

The Adafruit CircuitPython Library Bundle

Adafruit provides CircuitPython libraries for much of the hardware they provide, including sensors, breakouts and more. To eliminate the need for searching for each library individually, the libraries are available together in the Adafruit CircuitPython Library Bundle. The bundle contains all the files needed to use each library.

Downloading the Adafruit CircuitPython Library Bundle

You can download the latest Adafruit CircuitPython Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

Match up the bundle version with the version of CircuitPython you are running. For example, you would download the 6.x library bundle if you're running any version of CircuitPython 6, or the 7.x library bundle if you're running any version of CircuitPython 7, etc. If you mix libraries with major CircuitPython versions, you will get incompatible mpy errors due to changes in library interfaces possible during major version changes.

Click to visit circuitpython.org for the latest Adafruit CircuitPython Library Bundle

https://adafru.it/ENC

Download the bundle version that matches your CircuitPython firmware version. If you don't know the version, check the version info in **boot_out.txt** file on the **CIRCUITPY** drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.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.

The CircuitPython Community Library Bundle

The CircuitPython Community Library Bundle is made up of libraries written and provided by members of the CircuitPython community. These libraries are often written when community members encountered hardware not supported in the Adafruit Bundle, or to support a personal project. The authors all chose to submit these libraries to the Community Bundle make them available to the community.

These libraries are maintained by their authors and are not supported by Adafruit. As you would with any library, if you run into problems, feel free to file an issue on the GitHub repo for the library. Bear in mind, though, that most of these libraries are supported by a single person and you should be patient about receiving a response. Remember, these folks are not paid by Adafruit, and are volunteering their personal time when possible to provide support.

Downloading the CircuitPython Community Library Bundle

You can download the latest CircuitPython Community Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

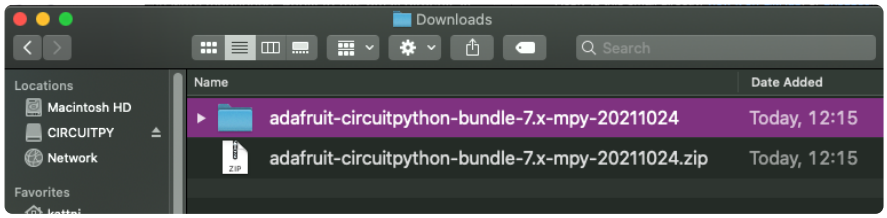
Click for the latest CircuitPython Community Library Bundle release

https://adafru.it/VCn

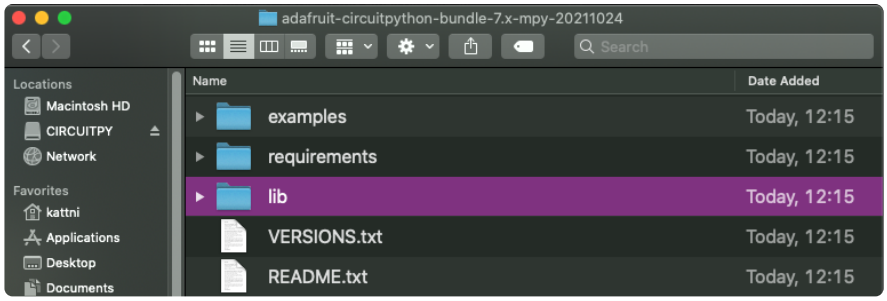
The link takes you to the latest release of the CircuitPython Community Library Bundle on GitHub. There are multiple versions of the bundle available. **Download the bundle version that matches your CircuitPython firmware version.** If you don't know the version, check the version info in **boot_out.txt** file on the **CIRCUITPY** drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.x library bundle.

Understanding the Bundle

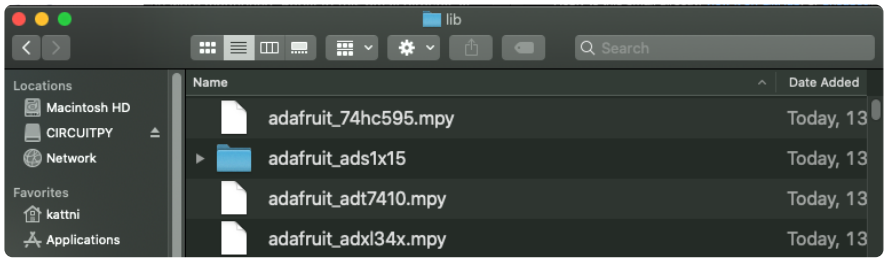
After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.



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.



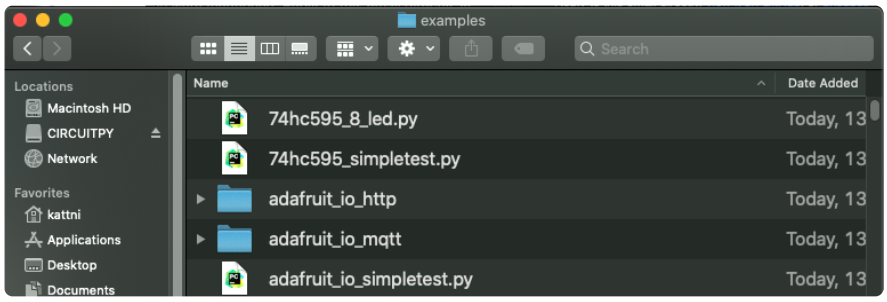
Now open the lib folder. When you open the folder, you'll see a large number of **.mpy** files, and folders.



Example Files

All example files from each library are now included in the bundles in an **examples** directory (as seen above), 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 open the **lib** folder on your **CIRCUITPY** drive. 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**.

If the library is a directory with multiple **.mpy** files in it, be sure to **copy the entire folder to CIRCUITPY/lib**.

This also applies to example files. Open the **examples** folder you extracted from the downloaded zip, and copy the applicable file to your **CIRCUITPY** drive. Then, rename it to **code.py** to run it.

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

Understanding Which Libraries to Install

You now know how to load libraries on to your CircuitPython-compatible microcontroller board. You may now be wondering, how do you know which libraries you need to install? Unfortunately, it's not always straightforward. Fortunately, there is

an obvious place to start, and a relatively simple way to figure out the rest. First up: the best place to start.

When you look at most CircuitPython examples, you'll see they begin with one or more `import` statements. These typically look like the following:

- `import library_or_module`

However, `import` statements can also sometimes look like the following:

- `from library_or_module import name`
- `from library_or_module.subpackage import name`
- `from library_or_module import name as local_name`

They can also have more complicated formats, such as including a `try` / `except` block, etc.

The important thing to know is that **an `import` statement will always include the name of the module or library that you're importing**.

Therefore, the best place to start is by reading through the `import` statements.

Here is an example import list for you to work with in this section. There is no setup or other code shown here, as the purpose of this section involves only the import list.

```
import time
import board
import neopixel
import adafruit_lis3dh
import usb_hid
from adafruit_hid.consumer_control import ConsumerControl
from adafruit_hid.consumer_control_code import ConsumerControlCode
```

Keep in mind, not all imported items are libraries. Some of them are almost always built-in CircuitPython modules. How do you know the difference? Time to visit the REPL.

In the Interacting with the REPL section (https://adafru.it/Awz) on The REPL page (https://adafru.it/Awz) in this guide, the `help("modules")` command is discussed. This command provides a list of all of the built-in modules available in CircuitPython for your board. So, if you connect to the serial console on your board, and enter the REPL, you can run `help("modules")` to see what modules are available for your board. Then, as you read through the `import` statements, you can,

for the purposes of figuring out which libraries to load, ignore the statement that import modules.

The following is the list of modules built into CircuitPython for the Feather RP2040. Your list may look similar or be anything down to a significant subset of this list for smaller boards.

Now that you know what you're looking for, it's time to read through the import statements. The first two, `time` and `board`, are on the modules list above, so they're built-in.

The next one, `neopixel`, is not on the module list. That means it's your first library! So, you would head over to the bundle zip you downloaded, and search for **neopixel**. There is a **neopixel.mpy** file in the bundle zip. Copy it over to the **lib** folder on your **CIRCUITPY** drive. The following one, `adafruit_lis3dh`, is also not on the module list. Follow the same process for **adafruit_lis3dh**, where you'll find **adafruit_lis3dh.mpy**, and copy that over.

The fifth one is `usb_hid`, and it is in the modules list, so it is built in. Often all of the built-in modules come first in the import list, but sometimes they don't! Don't assume that everything after the first library is also a library, and verify each import with the modules list to be sure. Otherwise, you'll search the bundle and come up empty!

The final two imports are not as clear. Remember, when `import` statements are formatted like this, the first thing after the `from` is the library name. In this case, the library name is `adafruit_hid`. A search of the bundle will find an **adafruit_hid folder**. When a library is a folder, you must copy the **entire folder and its contents as it is in the bundle** to the **lib** folder on your **CIRCUITPY** drive. In this case, you would copy the entire **adafruit_hid** folder to your **CIRCUITPY/lib** folder.

Notice that there are two imports that begin with `adafruit_hid`. Sometimes you will need to import more than one thing from the same library. Regardless of how many times you import the same library, you only need to load the library by copying over the **adafruit_hid** folder once.

That is how you can use your example code to figure out what libraries to load on your CircuitPython-compatible board!

There are cases, however, where libraries require other libraries internally. The internally required library is called a dependency. In the event of library dependencies, the easiest way to figure out what other libraries are required is to connect to the serial console and follow along with the `ImportError` printed there. The following is a very simple example of an `ImportError`, but the concept is the same for any missing library.

Example: `ImportError` Due to Missing Library

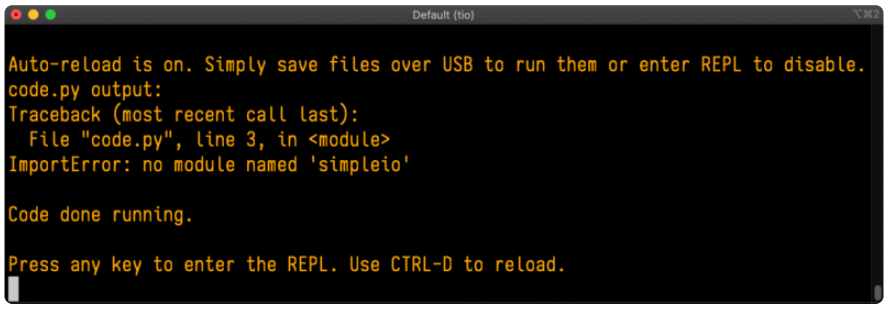
If you choose to load libraries as you need them, or you're starting fresh with an existing example, you may end up with code that tries to use a library you haven't yet loaded. This section will demonstrate what happens when you try to utilise a library that you don't have loaded on your board, and cover the steps required to resolve the issue.

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

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

```
import board
import time
import simpleio

led = simpleio.DigitalOut(board.LED)

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

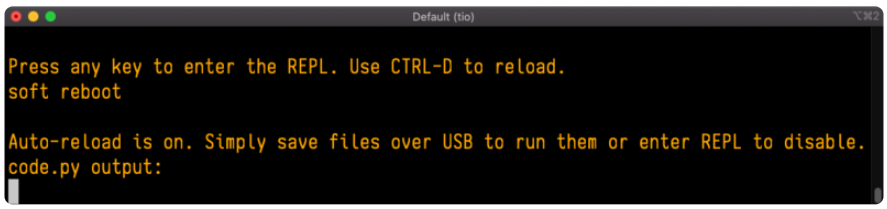
Save this file. Nothing happens to your board. Let's check the serial console to see what's going on.



You have an `ImportError`. It says there is `no module named 'simpleio'`. That's the one you just included in your 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 you'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 an M0 non-Express board such as Trinket M0, Gemma M0, QT Py M0, or one of the M0 Trinkeys, you'll want to follow the same steps in the example above to install libraries as you need them. Remember, you don't need to wait for an `ImportError` if you know what library you added to your code. Open the library bundle you downloaded, find the library you need, and drag it to the **lib** folder on your **CIRCUITPY** drive.

You can still end up running out of space on your M0 non-Express board 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 suggestions on the Troubleshooting page (https://adafru.it/Den).

Updating CircuitPython Libraries and 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.

CircUp CLI Tool

There is a command line interface (CLI) utility called CircUp (https://adafru.it/Tfi) that can be used to easily install and update libraries on your device. Follow the directions on the install page within the CircUp learn guide (https://adafru.it/-Ad). Once you've got it installed you run the command `circup update` in a terminal to interactively update all libraries on the connected CircuitPython device. See the usage page in the CircUp guide (https://adafru.it/-Ah) for a full list of functionality

---

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 SAMD21. **You may have a different board, and this list will vary, based on the board.**

```
>>> 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 SAMD21 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 SAMD21, 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 SAMD21 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())
```

The board.I2C(), board.SPI(), and board.UART() singletons do not exist on all boards. They exist if there are board markings for the default pins for those devices.

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, first, connect to the serial console.

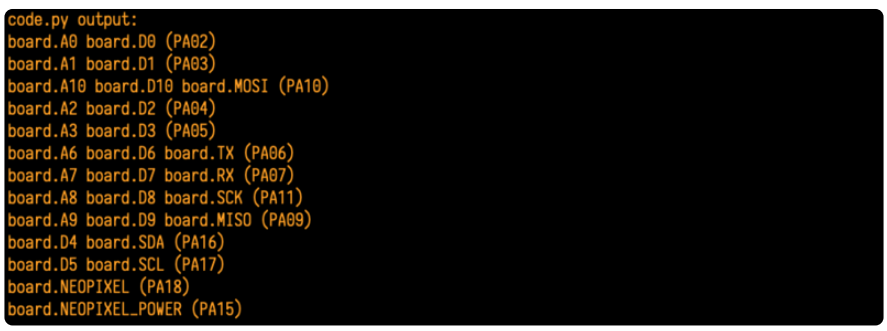
In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Essentials/Pin_Map_Script/** and then click on

the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:

```
# SPDX-FileCopyrightText: 2020 anecdata for Adafruit Industries
# SPDX-FileCopyrightText: 2021 Neradoc for Adafruit Industries
# SPDX-FileCopyrightText: 2021-2023 Kattni Rembor for Adafruit Industries
# SPDX-FileCopyrightText: 2023 Dan Halbert for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython Essentials Pin Map Script"""
import microcontroller
import board
try:
    import cyw43  # raspberrypi
except ImportError:
    cyw43 = None

board_pins = []
for pin in dir(microcontroller.pin):
    if (isinstance(getattr(microcontroller.pin, pin), microcontroller.Pin) or
        (cyw43 and isinstance(getattr(microcontroller.pin, pin), cyw43.CywPin))):
        pins = []
        for alias in dir(board):
            if getattr(board, alias) is getattr(microcontroller.pin, pin):
                pins.append(f"board.{alias}")
        # Add the original GPIO name, in parentheses.
        if pins:
            # Only include pins that are in board.
            pins.append(f"({str(pin)})")
            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 SAMD21:



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 (PA02)`. This means that you can access pin **A0** in CircuitPython using both `board.A0` and `board.D0`.

The pins in parentheses are the microcontroller pin names. See the next section for more info on those.

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 SAMD21 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

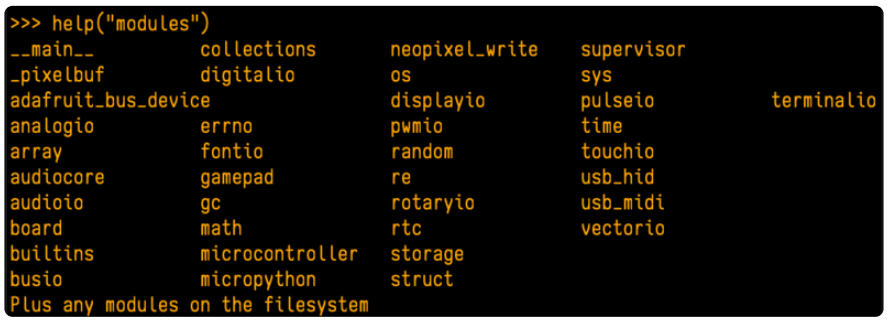
There is a set of modules used in most CircuitPython programs. One or more of these modules is always used in projects involving hardware. Often hardware requires installing a separate library from the Adafruit CircuitPython Bundle. But, if you try to find `board` or `digitalio` in the same bundle, you'll 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")
```



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

Advanced Serial Console on Mac

Connecting to the serial console on Mac does not require installing any drivers or extra software. You'll use a terminal program to find your board, and `screen` to connect to it. Terminal and `screen` both come installed by default.

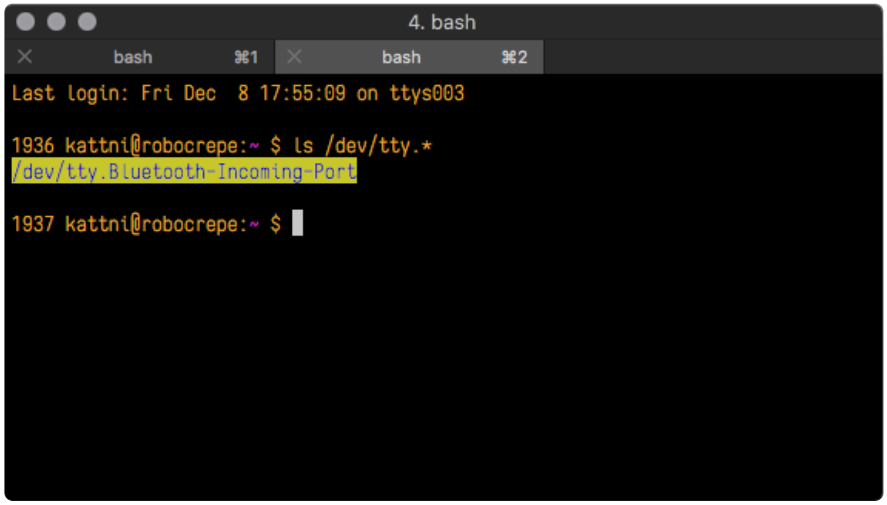
What's the Port?

First you'll want to find out which serial port your board is using. When you plug your board in to USB on your computer, it connects to a serial port. The port is like a door through which your board can communicate with your computer using USB.

The easiest way to determine which port the board is using is to first check **without** the board plugged in. Open Terminal and type the following:

```
ls /dev/tty.*
```

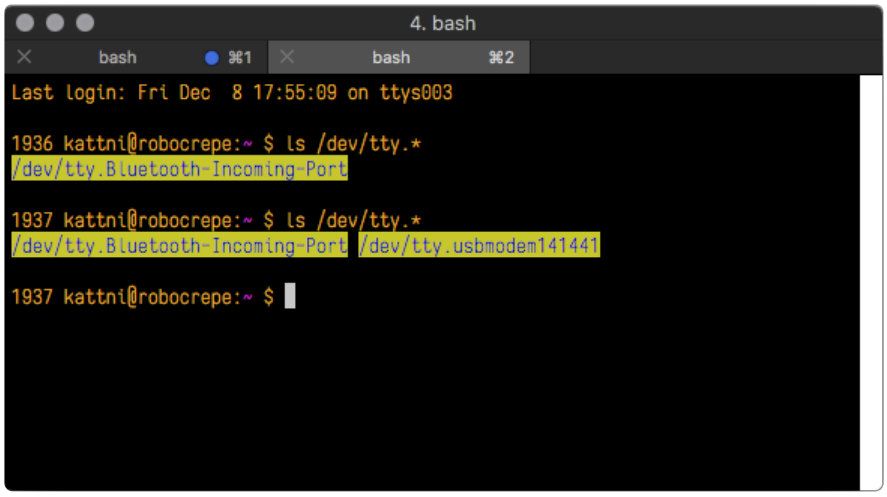
Each serial connection shows up in the `/dev/` directory. It has a name that starts with `tty.`. The command `ls` shows you a list of items in a directory. You can use `*` as a wildcard, to search for files that start with the same letters but end in something different. In this case, you're asking to see all of the listings in `/dev/` that start with `tty.` and end in anything. This will show us the current serial connections.



Now, plug your board. In Terminal, type:

```
ls /dev/tty.*
```

This will show you the current serial connections, which will now include your board.



A new listing has appeared called `/dev/tty.usbmodem141441`.
The `tty.usbmodem141441` part of this listing is the name the example board is using. Yours will be called something similar.

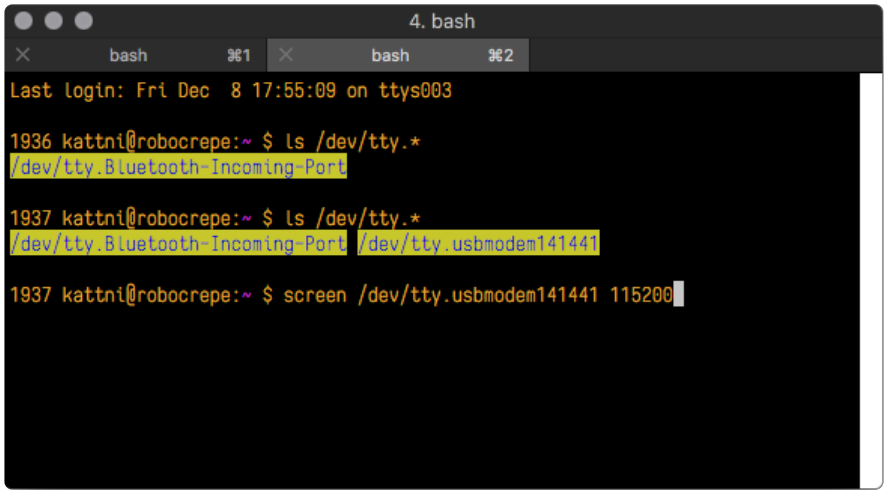
Using Linux, a new listing has appeared called `/dev/ttyACM0`. The `ttyACM0` part of this listing is the name the example board is using. Yours will be called something similar.

Connect with screen

Now that you know the name your board is using, you're ready connect to the serial console. You're going to use a command called `screen`. The `screen` command is included with MacOS. To connect to the serial console, use Terminal. Type the following command, replacing `board_name` with the name you found your board is using:

```
screen /dev/tty.board_name 115200
```

The first part of this establishes using the `screen` command. The second part tells screen the name of the board you're trying to use. The third part tells screen what baud rate to use for the serial connection. The baud rate is the speed in bits per second that data is sent over the serial connection. In this case, the speed required by the board is 115200 bits per second.



Press enter to run the command. It will open in the same window. If no code is running, the window will be blank. Otherwise, you'll see the output of your code.

Great job! You've connected to the serial console!

Frequently Asked Questions

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

What are some common acronyms to know?

CP or CPy = CircuitPython (https://adafru.it/KJD)
CPC = Circuit Playground Classic (http://adafru.it/3000) (does not run CircuitPython)
CPX = Circuit Playground Express (http://adafru.it/3333)
CPB = Circuit Playground Bluefruit (http://adafru.it/4333)

Using Older Versions

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

I have to continue using CircuitPython 7.x or earlier. Where can I find compatible libraries?

We are no longer building or supporting the CircuitPython 7.x or earlier 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, here are the last available library bundles for older versions:

- 2.x bundle (https://adafru.it/FJA)
- 3.x bundle (https://adafru.it/FJB)
- 4.x bundle (https://adafru.it/QDL)
- 5.x bundle (https://adafru.it/QDJ)
- 6.x bundle (https://adafru.it/Xmf)
- 7.x bundle (https://adafru.it/18e9)

Python Arithmetic

Does CircuitPython support floating-point numbers?

All CircuitPython boards support floating point arithmetic, even if the microcontroller chip does not support floating point in hardware. Floating point numbers are stored in 30 bits, with an 8-bit exponent and a 22-bit mantissa. Note that this is two bits less than standard 32-bit single-precision floats. You will get about 5-1/2 digits of decimal precision.

(The **broadcom** port may provide 64-bit floats in some cases.)

Does CircuitPython support long integers, like regular Python?

Python long integers (integers of arbitrary size) are available on most builds, except those on boards with the smallest available firmware size. On these boards, integers are stored in 31 bits.

Boards without long integer support are mostly SAMD21 ("M0") boards without an external flash chip, such as the Adafruit Gemma M0, Trinket M0, QT Py M0, and the Trinkey series. There are also a number of third-party boards in this category. There are also a few small STM third-party boards without long integer support.

`time.localtime()`, `time.mktime()`, `time.time()`, and `time.monotonic_ns()` are available only on builds with long integers.

Wireless Connectivity

How do I connect to the Internet with CircuitPython?

If you'd like to include WiFi in your project, your best bet is to use a board that is running natively on ESP32 chipsets - those have WiFi built in!

If your development board has an SPI port and at least 4 additional pins, you can check out this guide (https://adafru.it/F5X) on using AirLift with CircuitPython - extra wiring is required and some boards like the MacroPad or NeoTrellis do not have enough available pins to add the hardware support.

For further project examples, and guides about using AirLift with specific hardware, check out the Adafruit Learn System (https://adafru.it/VBr).

How do I do BLE (Bluetooth Low Energy) with CircuitPython?

The nRF52840 and nRF52833 boards have the most complete BLE implementation. Your program can act as both a BLE central and peripheral. As a central, you can scan for advertisements, and connect to an advertising board. As a peripheral, you can advertise, and you can create services available to a central. Pairing and bonding are supported.

ESP32-C3 and ESP32-S3 boards currently provide an incomplete (https://adafru.it/11Au) BLE implementation. Your program can act as a central, and connect to a peripheral. You can advertise, but you cannot create services. You cannot advertise anonymously. Pairing and bonding are not supported.

The ESP32 could provide a similar implementation, but it is not yet available. Note that the ESP32-S2 does not have Bluetooth capability.

On most other boards with adequate firmware space, BLE is available for use with AirLift (https://adafru.it/11Av) or other NINA-FW-based co-processors. Some boards have this coprocessor on board, such as the PyPortal (https://adafru.it/11Aw). Currently, this implementation only supports acting as a BLE peripheral. Scanning and connecting as a central are not yet implemented. Bonding and pairing are not supported.

Are there other ways to communicate by radio with CircuitPython?

Check out Adafruit's RFM boards (https://adafru.it/11Ay)for simple radio communication supported by CircuitPython, which can be used over distances of 100m to over a km, depending on the version. The RFM SAMD21 M0 boards can be used, but they were not designed for CircuitPython, and have limited RAM and flash space; using the RFM breakouts or FeatherWings with more capable boards will be easier.

Asyncio and Interrupts

Is there asyncio support in CircuitPython?

There is support for asyncio starting with CircuitPython 7.1.0, on all boards except the smallest SAMD21 builds. Read about using it in the Cooperative Multitasking in CircuitPython (https://adafru.it/XnA) Guide.

Does CircuitPython support interrupts?

No. CircuitPython does not currently support interrupts - please use asyncio for multitasking / 'threaded' control of your code

Status RGB LED

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)

Memory Issues

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.

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 here (https://adafru.it/QDK). Builds are available for Windows, macOS, x64 Linux, and Raspberry Pi Linux. Choose the latest `mpy-cross` whose version matches the version of CircuitPython you are using.

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

How do I check how much memory I have free?

Run the following to see the number of bytes available for use:

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

Unsupported Hardware

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://adafru.it/CiG)!

As of CircuitPython 8.x we have started to support ESP32 and ESP32-C3 and have added a WiFi workflow for wireless coding! (https://adafru.it/10JF)

We also support ESP32-S2 & ESP32-S3, which have native USB.

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.

ESP32-S2 Bugs & Limitations

Nobody likes bugs, but all nontrivial software and hardware has some. The master list of problems is the Issues list on github (https://adafru.it/188d).

I2C at 100 kHz bus frequency runs slowly

The default I2C bus clock speed is 100 kHz (100000) . At that rate, the ESP32-S2 will leave 10ms (https://adafru.it/ZCc) gaps between I2C transactions. This can slow down your I2C interactions considerably, such as when you are controlling a stepper motor with a PCA9685 controller.

Raising the I2C bus frequency to 125 kHz (125000) or higher fixes this problem. If your I2C peripheral can handle higher frequencies, you can use 400 kHz (400000) or even in some cases 1 MHz (1000000).

Note that `board.I2C()` creates an I2C bus that runs at 100 kHz. The bus frequency cannot be changed.. To create an I2C bus on the default I2C pins that runs at a different frequency, you must use `busio.I2C(board.SCL, board.SDA, frequency=)`.

No DAC-based audio output

Current versions of the ESP-IDF SDK do not have the required APIs for DAC-based audio output. Once a future version of ESP-IDF that adds it, it will be possible to implement DAC-based AudioOut in CircuitPython.

Workaround: PWMOut can create tones and buzzes.

Workaround: I2SOut audio is currently being developed and will work with boards such as the I2S 3W Class D Amplifier Breakout - MAX98357A (http://adafru.it/3006).

Deep Sleep & Wake-up sources

ESP32-S2 has hardware limitations on what kind of "pin alarms" can wake it. The following combinations are possible:

- EITHER one or two pins that wake from deep sleep when they are pulled LOW
- OR an arbitrary number of pins that wake from deep sleep when they are pulled HIGH, and optionally one pin that wakes from deep sleep when pulled LOW

This means that "wake" buttons should be wired so that pressing them pulls HIGH and a pull DOWN resistor is used with the pin. However, in some hardware designs including the original MagTag, the integrated buttons are pulled LOW when pressed and so only 1 or 2 buttons can be selected to wake the MagTag.

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 CircuitPython development continues and there are new releases, Adafruit 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 CircuitPython development continues and there are new releases, Adafruit 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 new versions of CircuitPython are released, Adafruit 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 7.x or earlier. Where can I find compatible libraries?

Adafruit is no longer building or supporting the CircuitPython 7.x or earlier library bundles. You are highly encourged 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, links to the previous bundles are available in the FAQ (https://adafru.it/FwY).

macOS Sonoma 14.x: Disk Errors Writing to CIRCUITPY

macOS Sonoma before 14.4 beta 2 takes many seconds to complete writes to small FAT drives, 8MB or smaller. This causes errors when writing to CIRCUITPY. The best solution is to remount the CIRCUITPY drive after it is automatically mounted. Or consider downgrading back to Ventura if that works for you. This problem is being tracked in CircuitPython GitHub issue 8449 (https://adafru.it/18ea).

Here is a shell script to do this remount conveniently (courtesy @czei in GitHub (https://adafru.it/18ea)). Copy the code here into a file named, say, **remount-CIRCUITPY.sh**. Place the file in a directory on your PATH, or in some other convenient place.

macOS Sonoma 14.4 beta and after does not have the problem above, but does take an inordinately long time to write to FAT drives of size 1GB or less (40 times longer than 2GB drives). This problem is being tracked in CircuitPython GitHub issue 8918 (https://adafru.it/19iD).

```
#!/bin/sh
#
# This works around bug where, by default, macOS 14.x writes part of a file
# immediately, and then doesn't update the directory for 20-60 seconds, causing
# the file system to be corrupted.
#

disky=`df | grep CIRCUITPY | cut -d" " -f1`
sudo umount /Volumes/CIRCUITPY
sudo mkdir /Volumes/CIRCUITPY
sleep 2
sudo mount -v -o noasync -t msdos $disky /Volumes/CIRCUITPY
```

Then in a Terminal window, do this to make this script executable:

```
chmod +x remount-CIRCUITPY.sh
```

Place the file in a directory on your `PATH`, or in some other convenient place.

Now, each time you plug in or reset your CIRCUITPY board, run the file **remount-CIRCUITPY.sh**. You can run it in a Terminal window or you may be able to place it on the desktop or in your dock to run it just by double-clicking.

This will be something of a nuisance but it is the safest solution.

This problem is being tracked in this CircuitPython issue (https://adafru.it/18ea).

Bootloader (boardnameBOOT) Drive Not Present

You may have a different board.

Only Adafruit Express boards and the SAMD21 non-Express boards ship with the UF2 bootloader (https://adafru.it/zbX)installed. The Feather M0 Basic, Feather M0 Adalogger, and similar boards use a regular Arduino-compatible bootloader, which does not show a **boardnameBOOT** drive.

MakeCode

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

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

To use a CircuitPython-compatible board with Windows 7 or 8.1, you must install a driver. Installation instructions are available here (https://adafru.it/VuB).

Windows 7 and 8.1 have reached end of life. It is recommended (https://adafru.it/Amd) that you upgrade to Windows 10 if possible; an upgrade is probably still free for you. Check here (https://adafru.it/Amd).

The Windows Drivers installer was last updated in November 2020 (v2.5.0.0) . Windows 7 drivers for CircuitPython boards released since then, including RP2040 boards, are not available. There are no plans to release drivers for new boards. The boards work fine on Windows 10.

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

Let us know in the Adafruit support forums (https://adafru.it/jlf) 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 that 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**: There have been 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 or Disappears Quickly

Kaspersky anti-virus can block the appearance of the **CIRCUITPY** drive. There has not yet been settings change discovered 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.

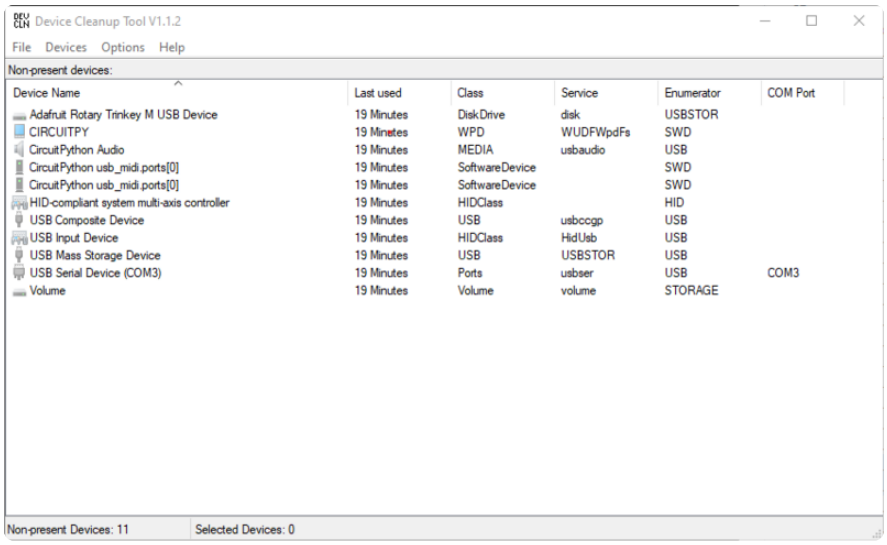
Sophos Endpoint security software can cause CIRCUITPY to disappear (https://adafru.it/ELr) and the BOOT drive to reappear. It is not clear what causes this behavior.

Samsung Magician can cause CIRCUITPY to disappear (reported here (https://adafru.it/18eb) and here (https://adafru.it/18ec)).

Device Errors or Problems on Windows

Windows can become confused about USB device installations. This is particularly true of Windows 7 and 8.1. It is recommended (https://adafru.it/Amd) that you upgrade to Windows 10 if possible; an upgrade is probably still free for you: see this link (https://adafru.it/V2a).

If not, try cleaning up your USB devices. Use Uwe Sieber's Device Cleanup Tool (https://adafru.it/RWd) (on that page, scroll down to "Device Cleanup Tool"). Download and unzip the tool. Unplug all the boards and other USB devices you want to clean up. Run the tool as Administrator. You will see a listing like this, probably with many more devices. It is listing all the USB devices that are not currently attached.



Select all the devices you want to remove, and then press Delete. It is usually safe just to select everything. Any device that is removed will get a fresh install when you plug it in. Using the Device Cleanup Tool also discards all the COM port assignments for the unplugged boards. If you have used many Arduino and CircuitPython boards, you have probably seen higher and higher COM port numbers used, seemingly without end. This will fix that problem.

Serial Console in Mu Not Displaying Anything

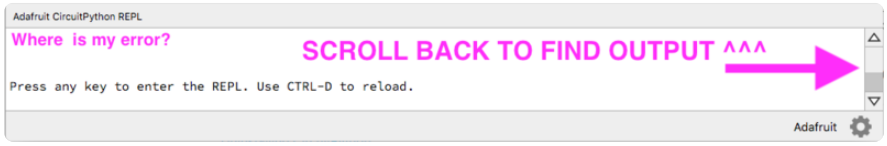
There are times when the serial console will accurately not display anything, such as, when no code is currently running, or when code with no serial output is already running before you open the console. However, if you find yourself in a situation where you feel it should be displaying something like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

code.py Restarts Constantly

CircuitPython will restart **code.py** if you or your computer writes to something on the CIRCUITPY drive. This feature is called auto-reload, and lets you test a change to your program immediately.

Some utility programs, such as backup, anti-virus, or disk-checking apps, will write to the CIRCUITPY as part of their operation. Sometimes they do this very frequently, causing constant restarts.

Acronis True Image and related Acronis programs on Windows are known to cause this problem. It is possible to prevent this by disabling the " (https://adafru.it/XDZ)Acronis Managed Machine Service Mini" (https://adafru.it/XDZ).

If you cannot stop whatever is causing the writes, you can disable auto-reload by putting this code in **boot.py** or **code.py**:

```
import supervisor

supervisor.runtime.autoreload = False
```

CircuitPython RGB Status Light

Nearly all 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. In versions before 7.0.0, they do NOT indicate any status while running CircuitPython.

CircuitPython 7.0.0 and Later

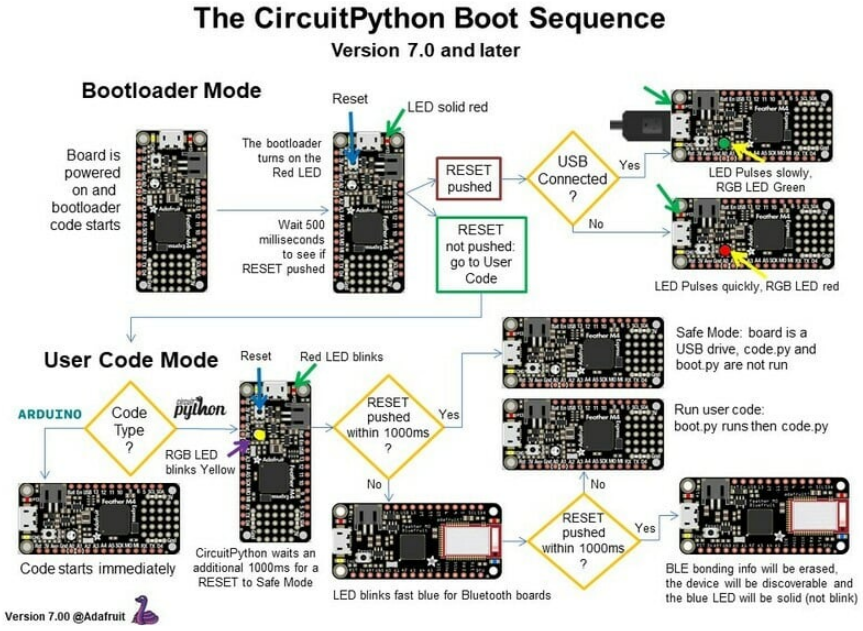
The status LED blinks were changed in CircuitPython 7.0.0 in order to save battery power and simplify the blinks. These blink patterns will occur on single color LEDs when the board does not have any RGB LEDs. Speed and blink count also vary for this reason.

On start up, the LED will blink **YELLOW** multiple times for 1 second. Pressing the RESET button (or on Espressif, the BOOT button) during this time will restart the board and then enter safe mode. On Bluetooth capable boards, after the yellow blinks, there will be a set of faster blue blinks. Pressing reset during the **BLUE** blinks will clear Bluetooth information and start the device in discoverable mode, so it can be used with a BLE code editor.

Once started, CircuitPython will blink a pattern every 5 seconds when no user code is running to indicate why the code stopped:

- 1 **GREEN** blink: Code finished without error.
- 2 **RED** blinks: Code ended due to an exception. Check the serial console for details.
- 3 **YELLOW** blinks: CircuitPython is in safe mode. No user code was run. Check the serial console for safe mode reason.

When in the REPL, CircuitPython will set the status LED to **WHITE**. You can change the LED color from the REPL. The status indicator will not persist on non-NeoPixel or DotStar LEDs.



CircuitPython 6.3.0 and earlier

Here's what the colors and blinking mean:

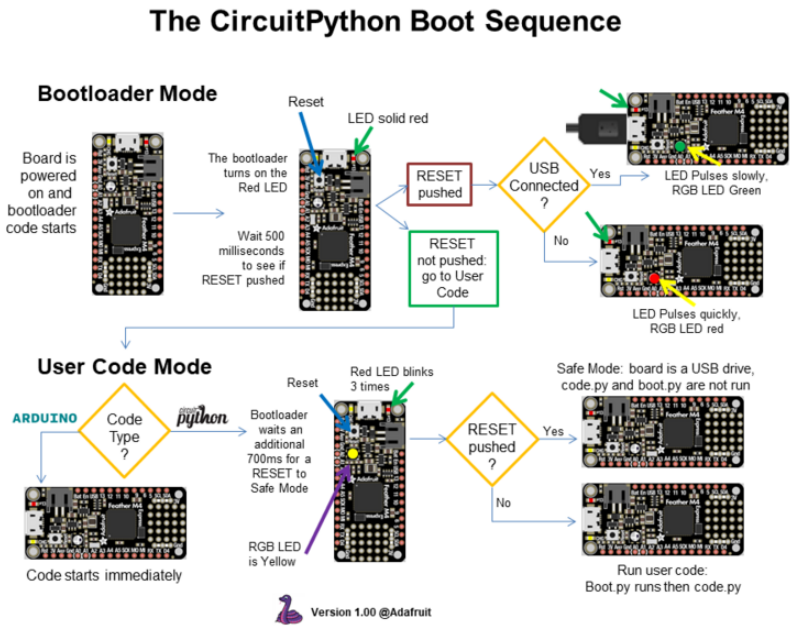
- steady **GREEN**: **code.py** (or **code.txt**, **main.py**, or **main.txt**) is running
- pulsing **GREEN**: **code.py** (etc.) has finished or does not exist
- steady **YELLOW** at start up: (4.0.0-alpha.5 and newer) CircuitPython is waiting for a reset to indicate that it should start in safe mode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



Serial console showing `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 6.x and 7.x, 2.x and 3.x, and 1.x and 2.x.

So, for instance, if you upgraded to CircuitPython 7.x from 6.x you'll need to download a newer version of the library that triggered the error on `import`. All libraries are available in the Adafruit bundle (https://adafru.it/y8E).

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. 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, though it is more common on Windows.

Be aware, if you have used Arduino to program your board, CircuitPython is no longer able to provide the USB services. You will need to reload CircuitPython to resolve this situation.

The easiest first step is to reload CircuitPython. Double-tap reset on the board so you get a **boardnameBOOT** drive rather than a **CIRCUITPY** drive, and copy the latest version of CircuitPython (**.uf2**) back to the board. This may restore **CIRCUITPY** functionality.

If reloading CircuitPython does not resolve your issue, the next step is to try putting the board into safe mode.

Safe Mode

Whether you've run into a situation where you can no longer edit your **code.py** on your **CIRCUITPY** drive, your board has gotten into a state where **CIRCUITPY** is read-only, or you have turned off the **CIRCUITPY** drive altogether, safe mode can help.

Safe mode in CircuitPython does not run any user code on startup, and disables auto-reload. This means a few things. First, safe mode bypasses any code in **boot.py** (where you can set **CIRCUITPY** read-only or turn it off completely). Second, it does not run the code in **code.py**. And finally, it does not automatically soft-reload when data is written to the **CIRCUITPY** drive.

Therefore, whatever you may have done to put your board in a non-interactive state, safe mode gives you the opportunity to correct it without losing all of the data on the **CIRCUITPY** drive.

Entering Safe Mode in CircuitPython 7.x and Later

You can enter safe by pressing reset during the right time when the board boots. Immediately after the board starts up or resets, it waits one second. On some boards, the onboard status LED will blink yellow during that time. If you press reset during that one second period, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a "slow" double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

Entering Safe Mode in CircuitPython 6.x

You can enter safe by pressing reset during the right time when the board boots.. Immediately after the board starts up or resets, it waits 700ms. On some boards, the

onboard status LED (highlighted in green above) will turn solid yellow during this time. If you press reset during that 700ms, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a slow double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

In Safe Mode

Once you've entered safe mode successfully in CircuitPython 6.x, the LED will pulse yellow.

If you successfully enter safe mode on CircuitPython 7.x, the LED will intermittently blink yellow three times.

If you connect to the serial console, you'll find the following message.

```
Auto-reload is off.
Running in safe mode! Not running saved code.

CircuitPython is in safe mode because you pressed the reset button during boot.
Press again to exit safe mode.

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

You can now edit the contents of the **CIRCUITPY** drive. Remember, your code will not run until you press the reset button, or unplug and plug in your board, to get out of safe mode.

At this point, you'll want to remove any user code in **code.py** and, if present, the **boot.py** file from **CIRCUITPY**. Once removed, tap the reset button, or unplug and plug in your board, to restart CircuitPython. This will restart the board and may resolve your drive issues. If resolved, you can begin coding again as usual.

If safe mode does not resolve your issue, 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.

To erase CIRCUITPY: `storage.erase_filesystem()`

CircuitPython includes a built-in function to erase and reformat the filesystem. If you have a version of CircuitPython older than 2.3.0 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 the following into the REPL:

```
&gt;&gt;&gt; import storage
&gt;&gt;&gt; storage.erase_filesystem()
```

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

Erase CIRCUITPY Without Access to the REPL

If you can't access the REPL, or you're running a version of CircuitPython previous to 2.3.0 and you don't want to upgrade, there are options available for some specific boards.

The options listed below are considered to be the "old way" of erasing your board. The method shown above using the REPL is highly recommended as the best method for erasing your board.

If at all possible, it is recommended to use the REPL to erase your CIRCUITPY drive. The REPL method is explained above.

For the specific boards listed below:

If the board you are trying to erase is listed below, follow the steps to use the file to erase your board.

1. Download the correct erase file:

Circuit Playground Express

https://adafru.it/AdI

Feather M0 Express

https://adafru.it/AdJ

Feather M4 Express

https://adafru.it/EVK

Metro M0 Express

https://adafru.it/AdK

Metro M4 Express QSPI Eraser

https://adafru.it/EoM

Trellis M4 Express (QSPI)

https://adafru.it/DjD

Grand Central M4 Express (QSPI)

https://adafru.it/DBA

PyPortal M4 Express (QSPI)

https://adafru.it/Eca

Circuit Playground Bluefruit (QSPI)

https://adafru.it/Gnc

Monster M4SK (QSPI)

https://adafru.it/GAN

PyBadge/PyGamer QSPI Eraser.UF2

https://adafru.it/GAO

CLUE_Flash_Erase.UF2

https://adafru.it/Jat

Matrix_Portal_M4_(QSPI).UF2

https://adafru.it/Q5B

RP2040 boards (flash_nuke.uf2)

https://adafru.it/18ed

2. Double-click the reset button on the board to bring up the **boardnameBOOT** drive.
3. Drag the erase **.uf2** file to the **boardnameBOOT** drive.
4. The status LED will turn yellow or blue, indicating the erase has started.
5. After approximately 15 seconds, the status LED will light up green. On the NeoTrellis M4 this is the first NeoPixel on the grid
6. Double-click the reset button on the board to bring up the **boardnameBOOT** drive.
7. Drag the appropriate latest release of CircuitPython (https://adafru.it/Em8) **.uf2** file to the **boardnameBOOT** drive.

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

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

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

For SAMD21 non-Express boards that have a UF2 bootloader:

Any SAMD21-based microcontroller that does not have external flash available is considered a SAMD21 non-Express board. Non-Express boards that have a UF2 bootloader include Trinket M0, GEMMA M0, QT Py M0, and the SAMD21-based Trinkey boards.

If you are trying to erase a SAMD21 non-Express board, follow these steps to erase your board.

1. Download the erase file:

SAMD21 non-Express Boards

https://adafru.it/VB-

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

reappear.

5. Drag the appropriate latest release CircuitPython (https://adafru.it/Em8) **.uf2** file to the boardnameBOOT drive.

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

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

For SAMD21 non-Express boards that do not have a UF2 bootloader:

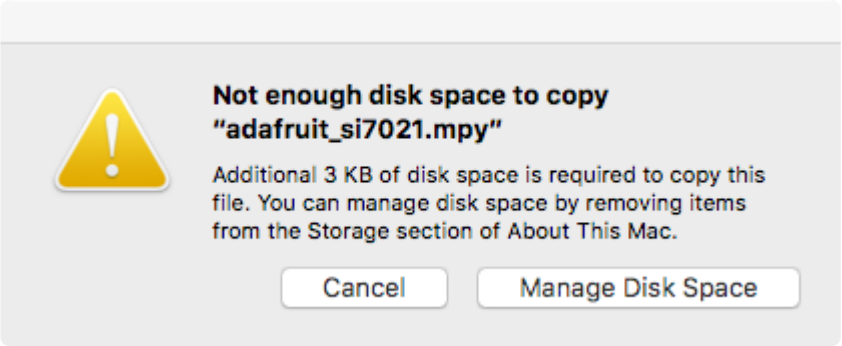
Any SAMD21-based microcontroller that does not have external flash available is considered a SAMD21 non-Express board. Non-Express boards that do **not** have a UF2 bootloader include the Feather M0 Basic Proto, Feather Adalogger, or the Arduino Zero.

If you are trying to erase a non-Express board that does not have a UF2 bootloader, 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 SAMD21 Non-Express Boards

Any SAMD21-based microcontroller that does not have external flash available is considered a SAMD21 non-Express board. This includes boards like the Trinket M0, GEMMA M0, QT Py M0, and the SAMD21-based Trinkey 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 number of ways to free up space.



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.

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. It's ~12KiB or so.

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, that is recommended 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 you're counting bytes.

On MacOS?

MacOS loves to generate hidden 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 **file_name.mpy** file to the board use a command like:

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

(Replace **file_name.mpy** with the name of the file you want to copy.)

Or to copy a folder and all of the files and folders contained within, use a command like:

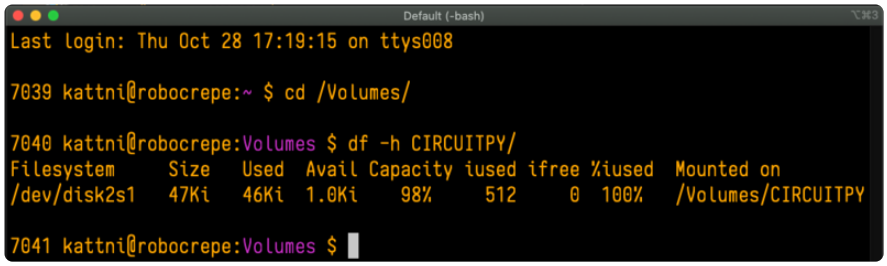
```
cp -rX folder_to_copy /Volumes/CIRCUITPY
```

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

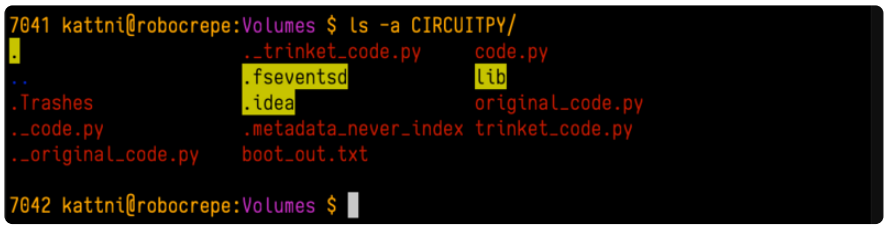
```
# if lib does not exist, you'll create a file named lib !
cp -X file_name.mpy /Volumes/CIRCUITPY/lib
# This is safer, and will complain if a lib folder does not exist.
cp -X file_name.mpy /Volumes/CIRCUITPY/lib/
```

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, move into the **Volumes/** directory with `cd /Volumes/`, and then list the amount of space used on the **CIRCUITPY** drive with the `df` command.



That's not very much space left! The next step is to show a list of the files currently on the **CIRCUITPY** drive, including the hidden files, using the `ls` command. You cannot use Finder to do this, you must do it via command line!



There are a few of the hidden files that MacOS loves to generate, all of which begin with a **._** before the file name. Remove the **._** files using the `rm` command. You can remove them all once by running `rm CIRCUITPY/._*`. The `*` acts as a wildcard to apply the command to everything that begins with **._** at the same time.

Finally, you can run `df` again to see the current space used.

```
7043 kattni@robocrepe:Volumes $ df -h CIRCUITPY/
Filesystem     Size   Used  Avail Capacity iused ifree %iused  Mounted on
/dev/disk2s1   47Ki   34Ki   13Ki    73%     512     0  100%   /Volumes/CIRCUITPY

7044 kattni@robocrepe:Volumes $
```

Nice! You have 12Ki 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. A boot loop occurs when the board reboots repeatedly and never fully loads. These are not caused by 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.

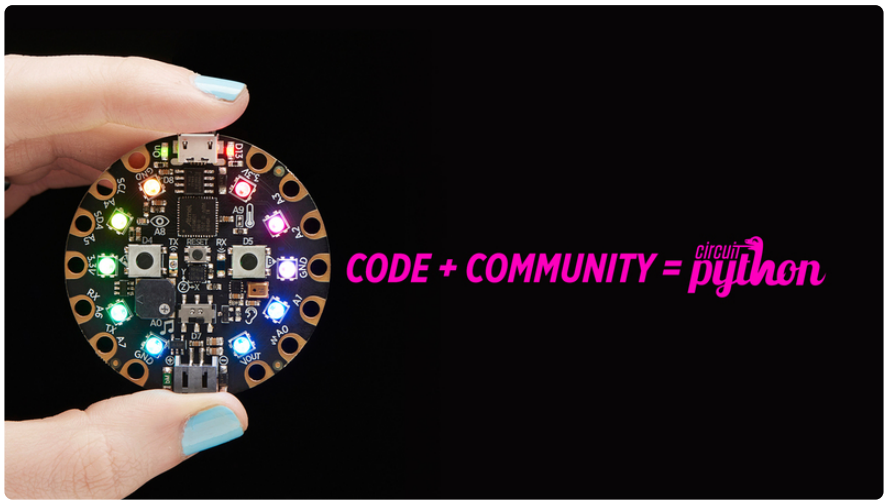
The method used to manually enter safe mode can be different for different devices. It is also very similar to the method used for getting into bootloader mode, which is a different thing. So it can take a few tries to get the timing right. If you end up in bootloader mode, no problem, you can try again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

For most devices:
Press the reset button, and then when the RGB status LED blinks yellow, press the reset button again. Since your reaction time may not be that fast, try a "slow" double click, to catch the yellow LED on the second click.

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 diagrams above for boot sequence details.

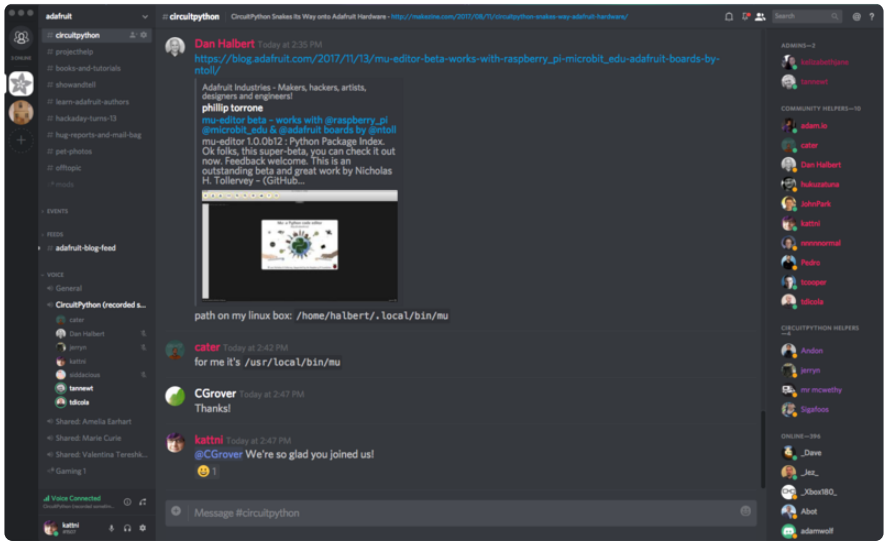
Welcome to the Community!



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. Whether this is your first microcontroller board or you're a seasoned software engineer, you have something important to offer the Adafruit CircuitPython community. This page highlights 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 #show-and-tell 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.

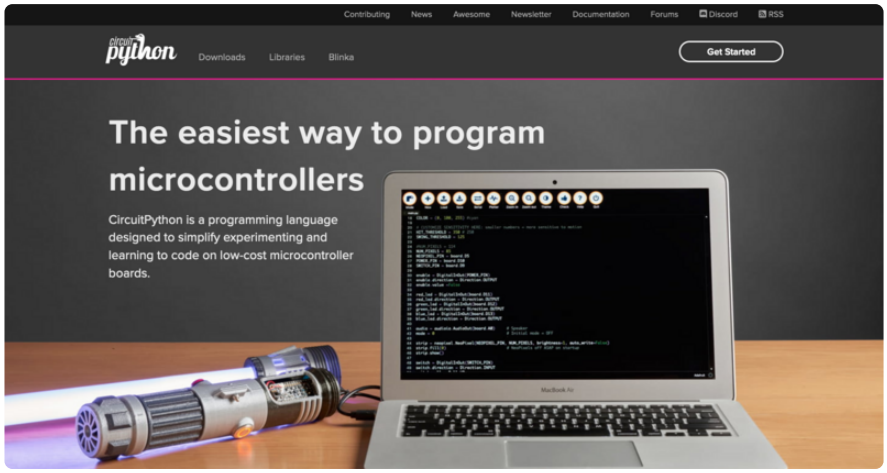
The help with CircuitPython channel is where to go with your CircuitPython questions. #help-with-circuitpython is there for new users and developers alike so feel free to ask a question or post a comment! Everyone of any experience level is welcome to join in on the conversation. Your contributions are important! The #circuitpython-dev 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.

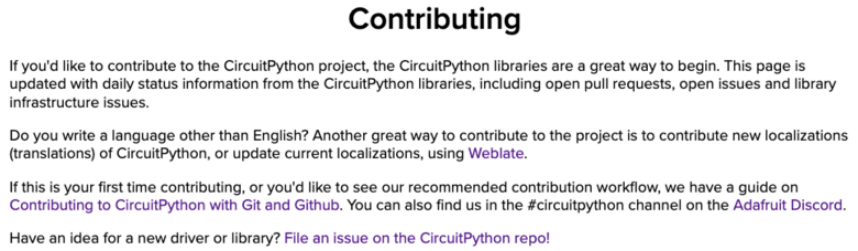
The Adafruit Discord is the 24x7x365 hackerspace that you can bring your granddaughter to.

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

CircuitPython.org



Beyond the Adafruit Learn System, which you are viewing right now, the best place to find information about CircuitPython is circuitpython.org (https://adafru.it/KJD). Everything you need to get started with your new microcontroller and beyond is available. You can do things like download CircuitPython for your microcontroller (https://adafru.it/Em8) or download the latest CircuitPython Library bundle (https://adafru.it/ENC), or check out which single board computers support Blinka (https://adafru.it/EA8). You can also get to various other CircuitPython related things like Awesome CircuitPython or the Python for Microcontrollers newsletter. This is all incredibly useful, but it isn't necessarily community related. So why is it included here? The Contributing page (https://adafru.it/VD7).



Contributing

If you'd like to contribute to the CircuitPython project, the CircuitPython libraries are a great way to begin. This page is updated with daily status information from the CircuitPython libraries, including open pull requests, open issues and library infrastructure issues.

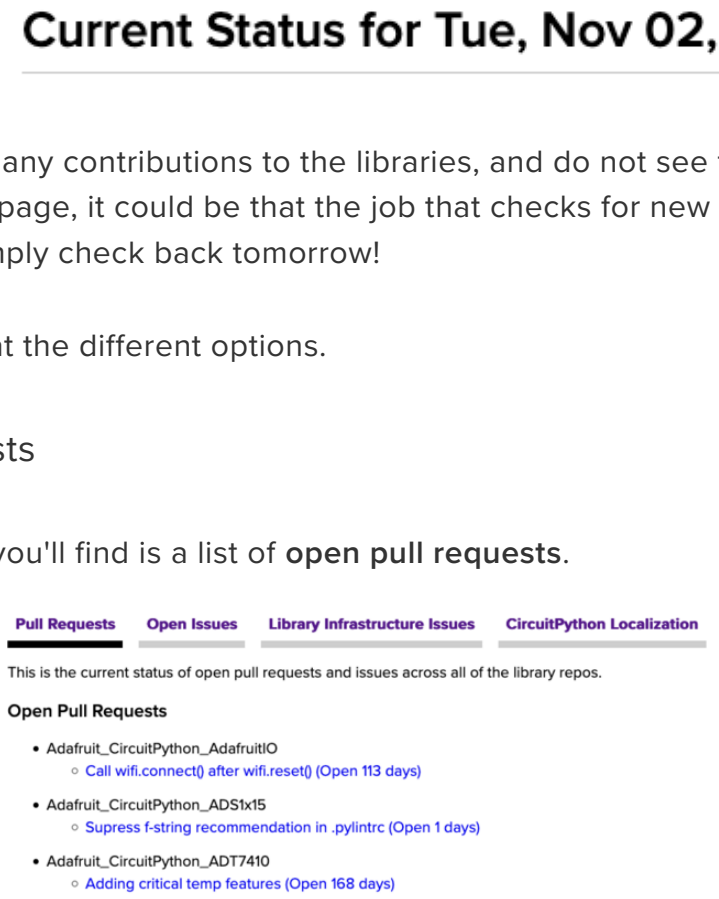
Do you write a language other than English? Another great way to contribute to the project is to contribute new localizations (translations) of CircuitPython, or update current localizations, using Weblate.

If this is your first time contributing, or you'd like to see our recommended contribution workflow, we have a guide on Contributing to CircuitPython with Git and Github. You can also find us in the #circuitpython channel on the Adafruit Discord.

Have an idea for a new driver or library? File an issue on the CircuitPython repo!

CircuitPython itself is written in C. However, all of the Adafruit CircuitPython libraries are written in Python. If you're interested in contributing to CircuitPython on the Python side of things, check out circuitpython.org/contributing (https://adafru.it/VD7). You'll find information pertaining to every Adafruit CircuitPython library GitHub repository, giving you the opportunity to join the community by finding a contributing option that works for you.

Note the date on the page next to **Current Status for**:



If you submit any contributions to the libraries, and do not see them reflected on the Contributing page, it could be that the job that checks for new updates hasn't yet run for today. Simply check back tomorrow!

Now, a look at the different options.

Pull Requests

The first tab you'll find is a list of **open pull requests**.

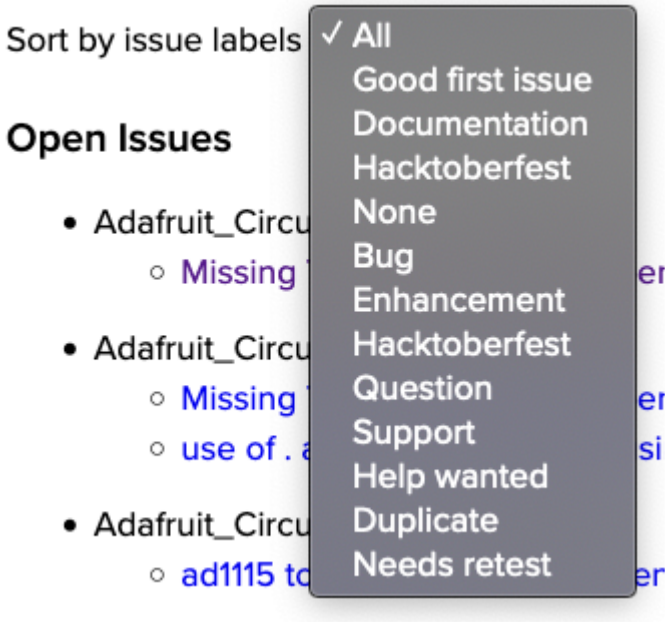
GitHub pull requests, or PRs, are opened when folks have added something to an Adafruit CircuitPython library GitHub repo, and are asking for Adafruit to add, or merge, their changes into the main library code. For PRs to be merged, they must first be reviewed. Reviewing is a great way to contribute! Take a look at the list of open pull requests, and pick one that interests you. If you have the hardware, you can test code changes. If you don't, you can still check the code updates for syntax. In the case of documentation updates, you can verify the information, or check it for spelling and grammar. Once you've checked out the update, you can leave a comment letting us know that you took a look. Once you've done that for a while, and you're more comfortable with it, you can consider joining the CircuitPythonLibrarians review team. The more reviewers we have, the more authors we can support. Reviewing is a crucial part of an open source ecosystem, CircuitPython included.

Open Issues

The second tab you'll find is a list of **open issues**.

GitHub issues are filed for a number of reasons, including when there is a bug in the library or example code, or when someone wants to make a feature request. Issues are a great way to find an opportunity to contribute directly to the libraries by updating code or documentation. If you're interested in contributing code or documentation, take a look at the open issues and find one that interests you.

If you're not sure where to start, you can search the issues by label. Labels are applied to issues to make the goal easier to identify at a first glance, or to indicate the difficulty level of the issue. Click on the dropdown next to "Sort by issue labels" to see the list of available labels, and click on one to choose it.



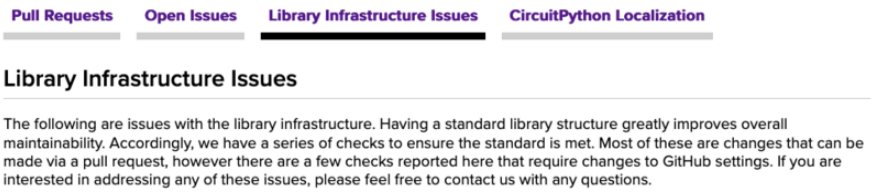
If you're new to everything, new to contributing to open source, or new to contributing to the CircuitPython project, you can choose "Good first issue". Issues with that label are well defined, with a finite scope, and are intended to be easy for someone new to figure out.

If you're looking for something a little more complicated, consider "Bug" or "Enhancement". The Bug label is applied to issues that pertain to problems or failures found in the library. The Enhancement label is applied to feature requests.

Don't let the process intimidate you. If you're new to Git and GitHub, there is a guide (https://adafru.it/Dkh) to walk you through the entire process. As well, there are always folks available on Discord () to answer questions.

Library Infrastructure Issues

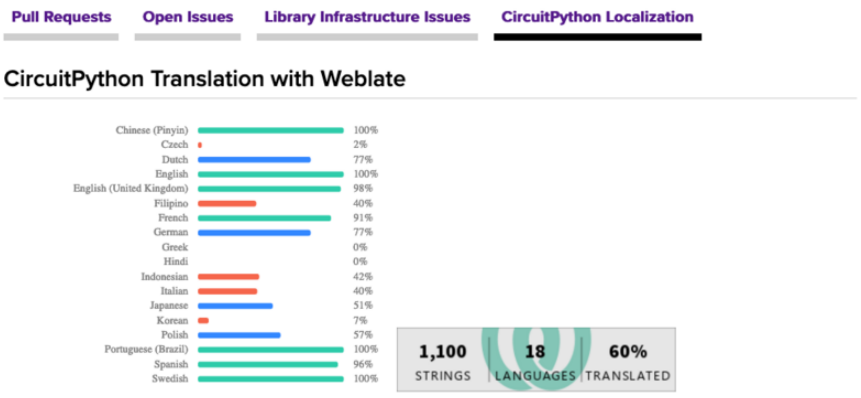
The third tab you'll find is a list of **library infrastructure issues**.



This section is generated by a script that runs checks on the libraries, and then reports back where there may be issues. It is made up of a list of subsections each containing links to the repositories that are experiencing that particular issue. This page is available mostly for internal use, but you may find some opportunities to contribute on this page. If there's an issue listed that sounds like something you could help with, mention it on Discord, or file an issue on GitHub indicating you're working to resolve that issue. Others can reply either way to let you know what the scope of it might be, and help you resolve it if necessary.

CircuitPython Localization

The fourth tab you'll find is the **CircuitPython Localization** tab.

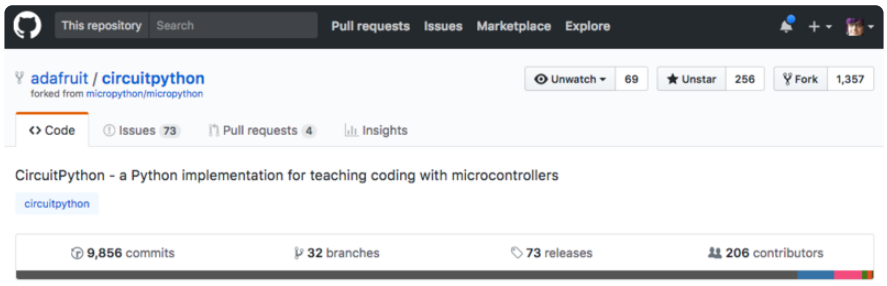


If you speak another language, you can help translate CircuitPython! The translations apply to informational and error messages that are within the CircuitPython core. It means that folks who do not speak English have the opportunity to have these messages shown to them in their own language when using CircuitPython. This is incredibly important to provide the best experience possible for all users.

CircuitPython uses Weblate to translate, which makes it much simpler to contribute translations. You will still need to know some CircuitPython-specific practices and a few basics about coding strings, but as with any CircuitPython contributions, folks are there to help.

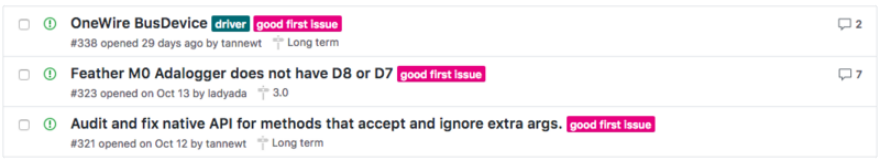
Regardless of your skill level, or how you want to contribute to the CircuitPython project, there is an opportunity available. The Contributing page (https://adafru.it/VD7) is an excellent place to start!

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 the CircuitPython project. The CircuitPython core is written in C. The libraries are written in Python. GitHub is the best source of ways to contribute to the CircuitPython core (https://adafru.it/tB7), and the CircuitPython libraries (https://adafru.it/VFv). 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. For the CircuitPython core, head over to the CircuitPython repository on GitHub, click on "Issues (https://adafru.it/tBb)", and you'll find a list that includes issues labeled "good first issue (https://adafru.it/188e)". For the libraries, head over to the Contributing page Issues list (https://adafru.it/VFv), and use the drop down menu to search for "good first issue (https://adafru.it/VFw)". These issues are things that have been 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. If you need help getting started with GitHub, there is an excellent guide on Contributing to CircuitPython with Git and GitHub (https://adafru.it/Dkh).



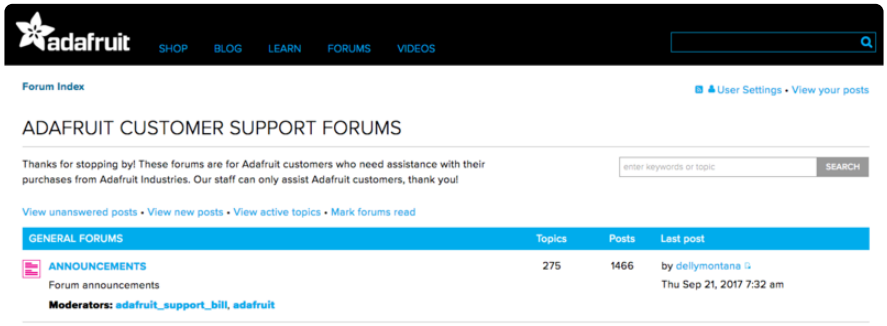
Already experienced and looking for a challenge? Checkout the rest of either issues list and you'll find plenty of ways to contribute. You'll find all sorts of things, from new driver requests, to library bugs, to core module updates. There's plenty of opportunities for everyone at any level!

When working with or using CircuitPython or the CircuitPython libraries, you may find problems. If you find a bug, that's great! The team loves bugs! Posting a detailed issue to GitHub is an invaluable way to contribute to improving CircuitPython. For CircuitPython itself, file an issue here (https://adafru.it/tBb). For the libraries, file an issue on the specific library repository on GitHub. 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 stable and unstable releases is a very important part of contributing CircuitPython. The developers can't possibly find all the problems themselves! They 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!

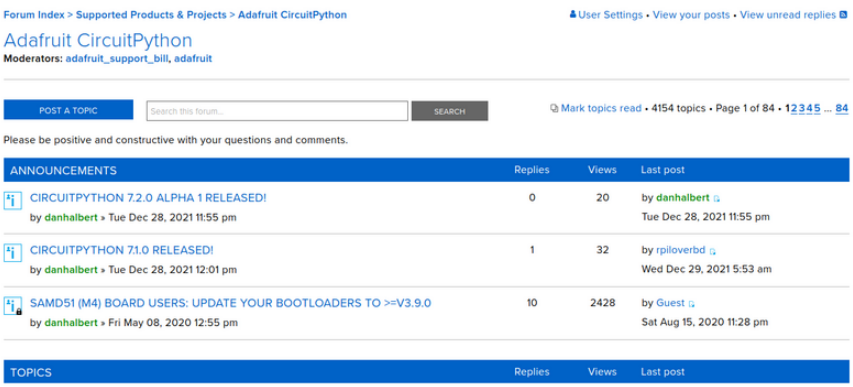
Adafruit Forums



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.

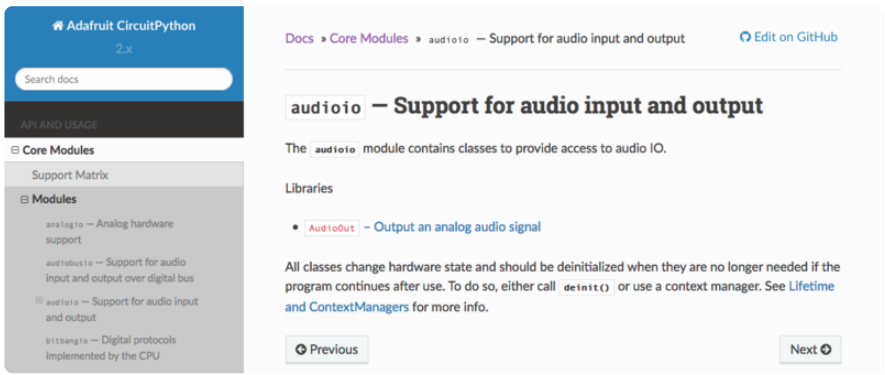
There are forum categories that cover all kinds of topics, including everything Adafruit. The Adafruit CircuitPython (https://adafru.it/xXA) category under "Supported Products & Projects" is the best place to post your CircuitPython questions.



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!

Read the Docs



Read the Docs (https://adafru.it/Beg) is a an excellent resource for a more detailed look at the CircuitPython core and the CircuitPython libraries. This is where you'll find

things like API documentation and example code. For an in depth look at viewing and understanding Read the Docs, check out the CircuitPython Documentation (https://adafru.it/VFx) page!

CircuitPython Essentials

You've been introduced to CircuitPython, and worked through getting everything set up. What's next? CircuitPython Essentials!

There are a number of core modules built into CircuitPython, which can be used along side the many CircuitPython libraries available. The following pages demonstrate some of these modules. Each page presents a different concept including a code example with an explanation. All of the examples are designed to work with your microcontroller board.

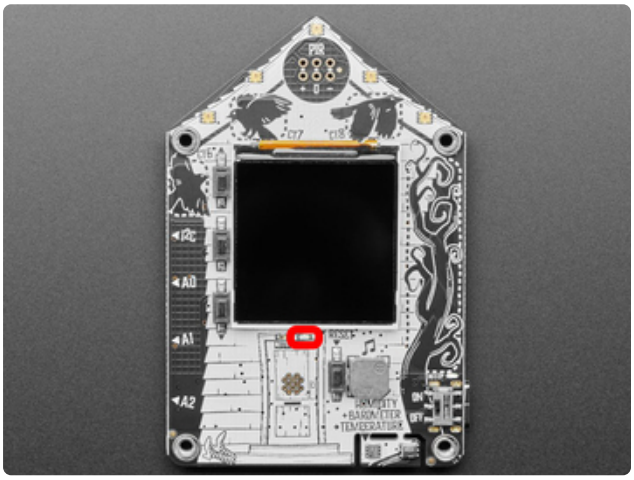
Time to get started learning the CircuitPython essentials!

Blink

In learning any programming language, you often begin with some sort of `Hello, World!` program. In CircuitPython, Hello, World! is blinking an LED. Blink is one of the simplest programs in CircuitPython. It involves three built-in modules, two lines of set up, and a short loop. Despite its simplicity, it shows you many of the basic concepts

needed for most CircuitPython programs, and provides a solid basis for more complex projects. Time to get blinky!

LED Location



The build-in red LED (indicated in red in the image) is located in the upper right of the FunHouse door, towards the center of the board on the front.

Blinking an LED

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Templates/blink/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython Blink Example - the CircuitPython 'Hello, World!'"""
import time
import board
import digitalio

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 built-in LED begins blinking!

Note that the code is a little less "Pythonic" than it could be. It could also be written as `led.value = not led.value` with a single `time.sleep(0.5)`. That way is more difficult to understand if you're new to programming, so the example is a bit longer than it needed to be to make it easier to read.

It's important to understand what is going on in this program.

First you `import` three modules: `time`, `board` and `digitalio`. This makes these modules available for use in your code. All three are built-in to CircuitPython, so you don't need to download anything to get started.

Next, you set up the LED. To interact with hardware in CircuitPython, your code must let the board know where to look for the hardware and what to do with it. So, you create a `digitalio.DigitalInOut()` object, provide it the LED pin using the `board` module, and save it to the variable `led`. Then, you tell the pin to act as an `OUTPUT`.

Finally, you create a `while True:` loop. This means all the code inside the loop will repeat indefinitely. Inside the loop, you set `led.value = True` which powers on the LED. Then, you use `time.sleep(0.5)` to tell the code to wait half a second before moving on to the next line. The next line sets `led.value = False` which turns the LED off. Then you use another `time.sleep(0.5)` to wait half a second before starting the loop over again.

With only a small update, you can control the blink speed. The blink speed is controlled by the amount of time you tell the code to wait before moving on using `time.sleep()`. The example uses `0.5`, which is one half of one second. Try increasing or decreasing these values to see how the blinking changes.

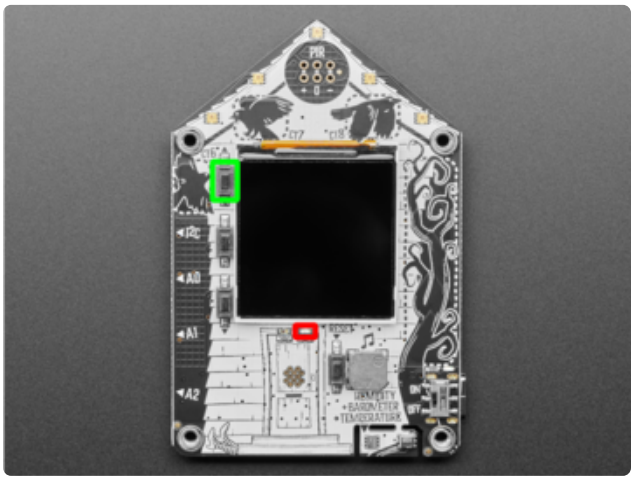
That's all there is to blinking an LED using CircuitPython!

Digital Input

The CircuitPython `digitalio` module has many applications. The basic Blink program sets up the LED as a digital output. You can just as easily set up a **digital input** such as a button to control the LED. This example builds on the basic Blink

example, but now includes setup for a button switch. Instead of using the `time` module to blink the LED, it uses the status of the button switch to control whether the LED is turned on or off.

LED and Button



The red led (indicated by the red box in the image) is located in the upper right of the FunHouse door, towards the center of the board on the front.
The top button (indicated by the green box in the image) is located near the upper left corner of the display below the up arrow on the board silk.

Controlling the LED with a Button

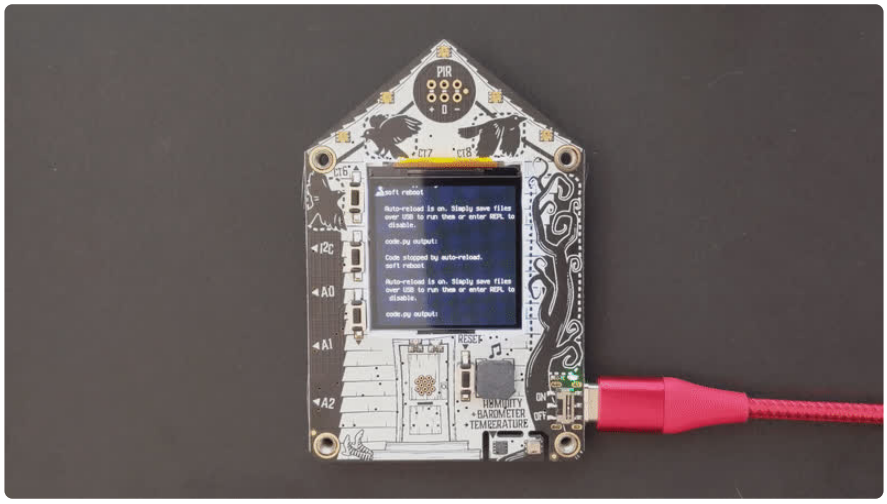
In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **Adafruit_FunHouse/digital_input_led/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT
#
"""CircuitPython Digital Input Example for FunHouse"""
import board
import digitalio

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

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

while True:
    if not button.value:
        led.value = False
    else:
        led.value = True
```

Now, press the button. The LED lights up! Let go of the button and the LED turns off.



Note that the code is a little less "Pythonic" than it could be. It could also be written as `led.value = not button.value`. That way is more difficult to understand if you're new to programming, so the example is a bit longer than it needed to be to make it easier to read.

First you `import` two modules: `board` and `digitalio`. This makes these modules available for use in your code. Both are built-in to CircuitPython, so you don't need to download anything to get started.

Next, you set up the LED. To interact with hardware in CircuitPython, your code must let the board know where to look for the hardware and what to do with it. So, you create a `digitalio.DigitalInOut()` object, provide it the LED pin using the `board` module, and save it to the variable `led`. Then, you tell the pin to act as an `OUTPUT`.

You include setup for the button as well. It is similar to the LED setup, except the button is an `INPUT`, and requires a pull up.

Inside the loop, you check to see if the button is pressed, and if so, turn on the LED. Otherwise the LED is off.

That's all there is to controlling an LED with a button switch!

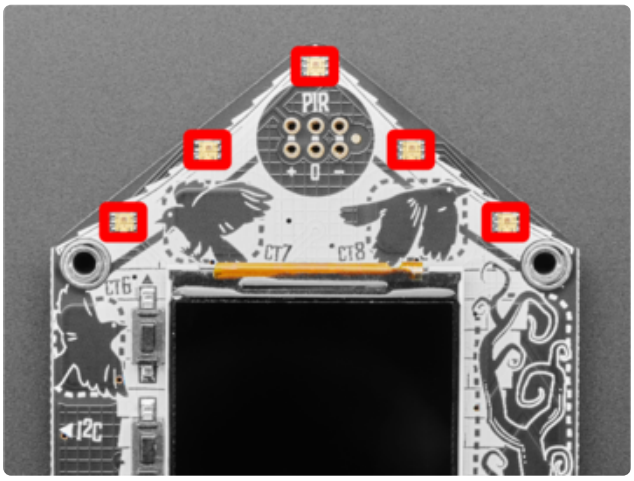
Built-In DotStar LEDs

Your board has multiple built-in RGB DotStar LEDs. You can use CircuitPython code to control the color and brightness of these LEDs. They are also used to indicate the bootloader status.

A DotStar refers to any 2-wire serial LED, typically APA102, but also possibly SK9822. Along side a driver chip, DotStars have have three LEDs: RGB DotStars have a red LED, a blue LED and a green LED, and white DotStars have three white LEDs. The LEDs on your microcontroller are RGB DotStars! DotStars operate over a generic 2-wire SPI bus, which means they aren't as strict about timing. They allow for extremely fast data and PWM rates so they're suitable for POV displays. They can be used individually (as in the built-in LED on your board), or chained together in strips or other creative form factors. DotStars do not light up on their own; they must be connected to a microcontroller. They require two pins, data and clock, to operate. The response time is faster when connected to a hardware SPI pair of pins, but will work connected to any two digital pins. You do not need to worry about connecting the DotStars because they're built into your microcontroller!

This page will cover using CircuitPython to control the RGB DotStars built into your microcontroller. You'll learn how to change the color and brightness, and how to make a rainbow. Time to get started!

DotStar Location



On the FunHouse, the DotStar LEDs (indicated by red boxes in the image) are spread out evenly along the top edges of the board, beginning at the top center, and then along the two edges to the mounting holes.

DotStar Color and Brightness

To use the built-in DotStars on your board, you need to first install the DotStar library into the **lib** folder on your **CIRCUITPY** drive.

Then you need to update **code.py**.

Click the **Download Project Bundle** button below to download the necessary libraries and the code.py file in a zip file. Extract the contents of the zip file, and copy the **entire lib folder** and the **code.py file** to your **CIRCUITPY** drive.

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT
#
"""CircuitPython DotStar red, green, blue example for FunHouse"""
import time
import board
import adafruit_dotstar

dots = adafruit_dotstar.DotStar(board.DOTSTAR_CLOCK, board.DOTSTAR_DATA, 5)
dots.brightness = 0.3

while True:
    dots.fill((255, 0, 0))
    time.sleep(0.5)
    dots.fill((0, 255, 0))
    time.sleep(0.5)
    dots.fill((0, 0, 255))
    time.sleep(0.5)
```

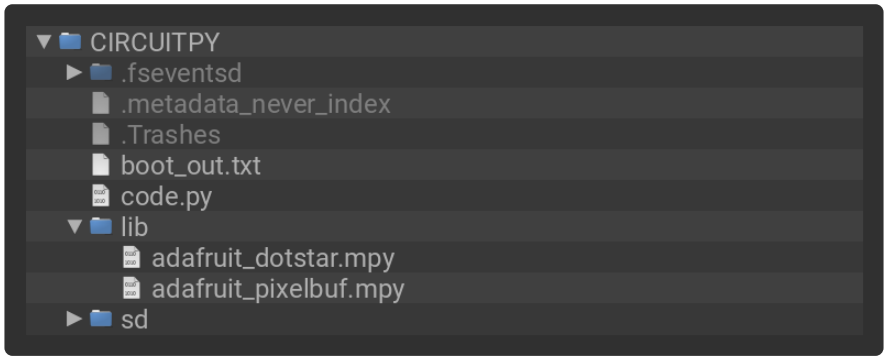
Your **CIRCUITPY** drive contents should resemble the image below.

You should have in / of the **CIRCUITPY** drive:

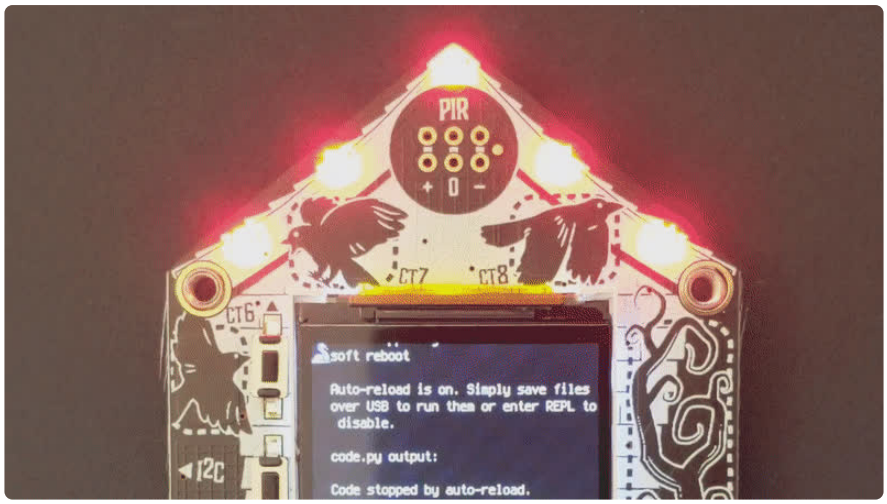
- **code.py**

And in the **lib** folder on your **CIRCUITPY** drive:

- **adafruit_bus_device/**
- **adafruit_dotstar.mpy**
- **adafruit_pixelbuf.mpy**



The DotStar LEDs being flashing red, green and blue!



If your DotStars do not start flashing red, green and blue, make sure you've copied all the necessary files and folders to the CIRCUITPY drive!

First you import the necessary modules, `time` and `board`, and the necessary library, `adafruit_dotstar`. This makes these modules and libraries available for use in your code. The first two are modules built-in to CircuitPython, so you don't need to download anything to use those. The `adafruit_dotstar` library is separate, which is why you needed to install it before getting started.

Next, you set up the Dotstar LEDs. To interact with hardware in CircuitPython, your code must let the board know where to look for the hardware and what to do with it. So, you create a `adafruit_dotstar.DotStar()` object, provide it the DotStar LED pins using the `board` module, and tell it the number of LEDs. You save this object to the variable `pixels`.

Then, you set the DotStar brightness using the `brightness` attribute. `brightness` expects float between `0` and `1.0`. A float is essentially a number with a decimal in it. The brightness value represents a percentage of maximum brightness; `0` is 0% and `1.0` is 100%. Therefore, setting `pixel.brightness = 0.3` sets the brightness to 30%. The default brightness, which is to say the brightness if you don't explicitly set it, is `1.0`. The default is really bright! That is why there is an option available to easily change the brightness.

Inside the loop, you turn the DotStars red for 0.5 seconds, green for 0.5 seconds, and blue for 0.5 seconds.

To turn the DotStars red, you "fill" them with an RGB value. Check out the section below for details on RGB colors. The RGB value for red is `(255, 0, 0)`. Note that the RGB value includes the parentheses. The `fill()` attribute expects the full RGB

value including those parentheses. That is why there are two pairs of parentheses in the code.

You can change the RGB values to change the colors that the DotStars cycle through. Check out the list below for some examples. You can make any color of the rainbow with the right RGB value combination!

That's all there is to changing the color and setting the brightness of the built-in DotStar LEDs!

RGB LED Colors

RGB LED colors are set using a combination of **r**ed, **g**reen, and **b**lue, in the form of an (**R**, **G**, **B**) tuple. Each member of the tuple is set to a number between 0 and 255 that determines the amount of each color present. Red, green and blue in different combinations can create all the colors in the rainbow! So, for example, to set an LED to red, the tuple would be `(255, 0, 0)`, which has the maximum level of red, and no green or blue. Green would be `(0, 255, 0)`, etc. For the colors between, you set a combination, such as cyan which is `(0, 255, 255)`, with equal amounts of green and blue. If you increase all values to the same level, you get white! If you decrease all the values to 0, you turn the LED off.

Common colors include:

- red: `(255, 0, 0)`
- green: `(0, 255, 0)`
- blue: `(0, 0, 255)`
- cyan: `(0, 255, 255)`
- purple: `(255, 0, 255)`
- yellow: `(255, 255, 0)`
- white: `(255, 255, 255)`
- black (off): `(0, 0, 0)`

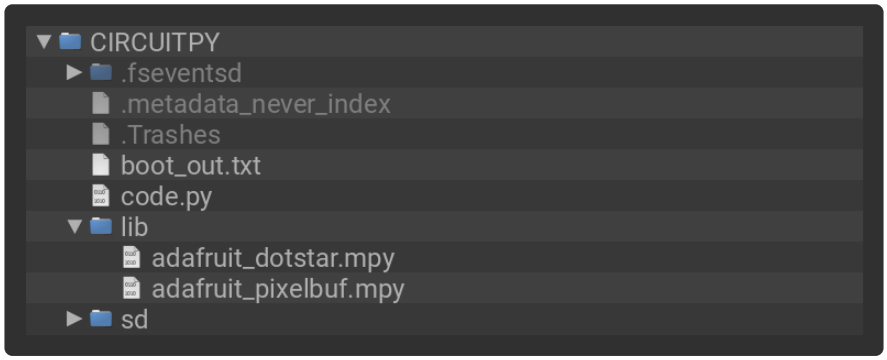
DotStar Rainbow

You should have already installed the library necessary to use the built-in DotStar LEDs. If not, follow the steps at the beginning of the DotStar Color and Brightness section to install it.

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip

file, open the directory **Adafruit_FunHouse/dotstar_rainbow/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:




```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT
#
"""CircuitPython DotStar rainbow example for FunHouse"""
import time
import board
import adafruit_dotstar
from rainbowio import colorwheel

dots = adafruit_dotstar.DotStar(board.DOTSTAR_CLOCK, board.DOTSTAR_DATA, 5,
auto_write=False)
dots.brightness = 0.3


def rainbow(delay):
    for color_value in range(255):
        for led in range(5):
            pixel_index = (led * 256 // 5) + color_value
            dots[led] = colorwheel(pixel_index & 255)
        dots.show()
        time.sleep(delay)


while True:
    rainbow(0.01)
```

The DotStars display a rainbow cycle!

This example builds on the previous example.

First, you import the same three modules and libraries. In addition to those, you import `colorwheel`.

The DotStar hardware setup is similar, but you now also set `auto_write` to `False`. This means that now the DotStar won't change unless you explicitly tell it to by calling `show()`. This is necessary for this example to speed up the rainbow animation. Brightness setting is the same.

Next, you have the `rainbow()` helper function. This helper displays the rainbow cycle. It expects a `delay` in seconds. The higher the number of seconds provided for `delay`, the slower the rainbow will cycle. The helper cycles through the values of the color wheel to create a rainbow of colors.

Inside the loop, you call the rainbow helper with a 0.01 second delay, by including `rainbow(0.01)`.

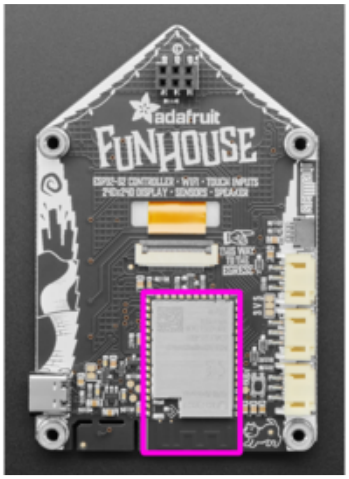
That's all there is to making rainbows using the built-in DotStar LEDs!

CPU Temperature

There is a temperature sensor built into the CPU on your microcontroller board. It reads the internal CPU temperature, which varies depending on how long the board has been running or how intense your code is.

CircuitPython makes it really simple to read this data from the temperature sensor built into the microcontroller. Using the built-in `microcontroller` module, you can easily read the temperature.

Microcontroller Location



The microcontroller on the FunHouse is located on the bottom of the back of the board, towards the center.

Reading the Microcontroller Temperature

The data is read using two lines of code. All necessary modules are built into CircuitPython, so you don't need to download any extra files to get started.

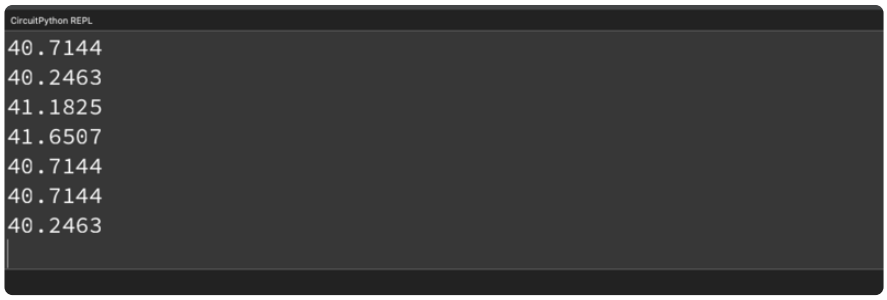
Connect to the serial console (https://adafru.it/Bec), and then update your **code.py** to the following.

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Templates/cpu_temperature/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython CPU temperature example in Celsius"""
import time
import microcontroller

while True:
```

```
    print(microcontroller.cpu.temperature)
    time.sleep(0.15)
```



The CPU temperature in Celsius is printed out to the serial console!

Try putting your finger on the microcontroller to see the temperature change.

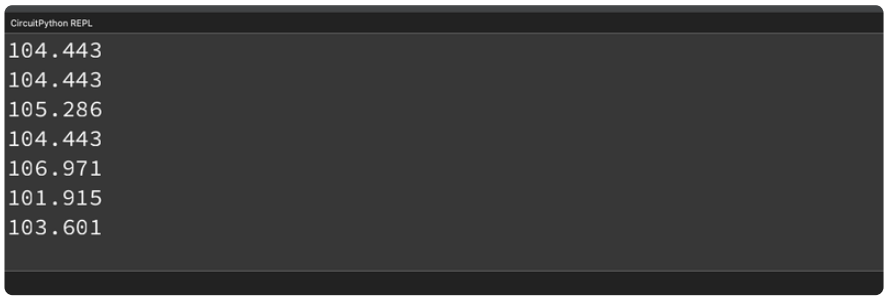
The code is simple. First you import two modules: `time` and `microcontroller`. Then, inside the loop, you print the microcontroller CPU temperature, and the `time.sleep()` slows down the print enough to be readable. That's it!

You can easily print out the temperature in Fahrenheit by adding a little math to your code, using this simple formula: Celsius * (9/5) + 32.

In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **CircuitPython_Templates/cpu_temperature_f/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT
"""CircuitPython CPU temperature example in Fahrenheit"""
import time
import microcontroller

while True:
    print(microcontroller.cpu.temperature * (9 / 5) + 32)
    time.sleep(0.15)
```



The CPU temperature in Fahrenheit is printed out to the serial console!

That's all there is to reading the CPU temperature using CircuitPython!

Arduino IDE Setup

The ESP32-S2/S3 bootloader does not have USB serial support for Windows 7 or 8. (See https://github.com/espressif/arduino-esp32/issues/5994) please update to version 10 which is supported by espressif! Alternatively you can try this community-crafted Windows 7 driver (https://github.com/kutukvpavel/Esp32-Win7-VCP-drivers)

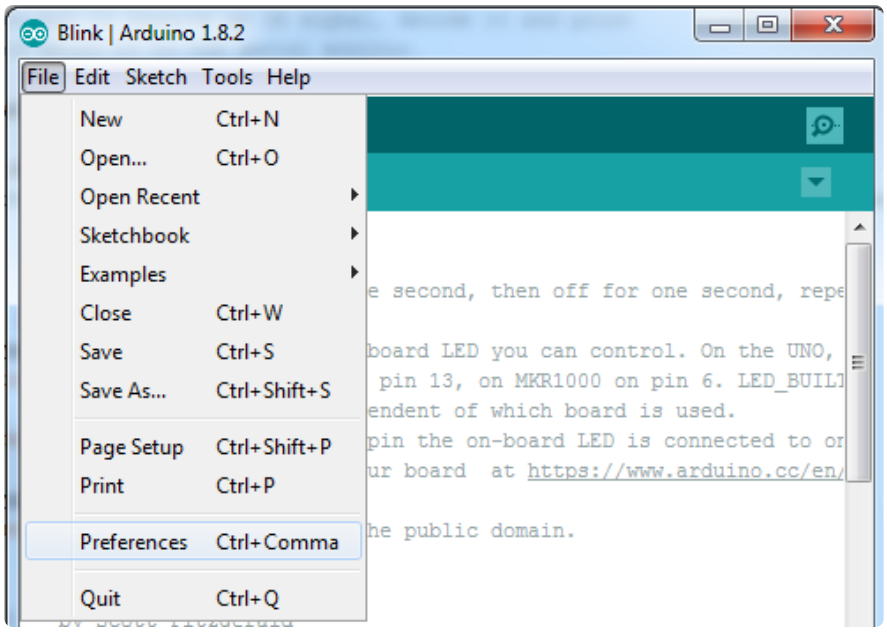
The first thing you will need to do is to download the latest release of the Arduino IDE. You will need to be using **version 1.8** or higher for this guide

Arduino IDE Download

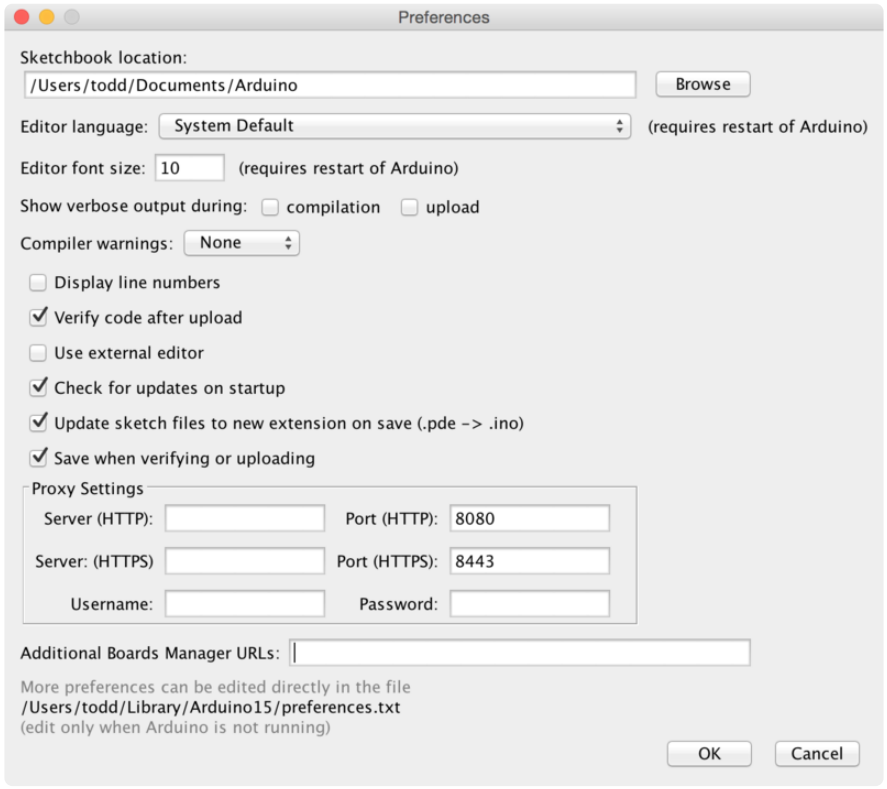
https://adafru.it/f1P

To use the ESP32-S2/S3 with Arduino, you'll need to follow the steps below for your operating system. You can also check out the Espressif Arduino repository for the most up to date details on how to install it (https://adafru.it/weF).

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.

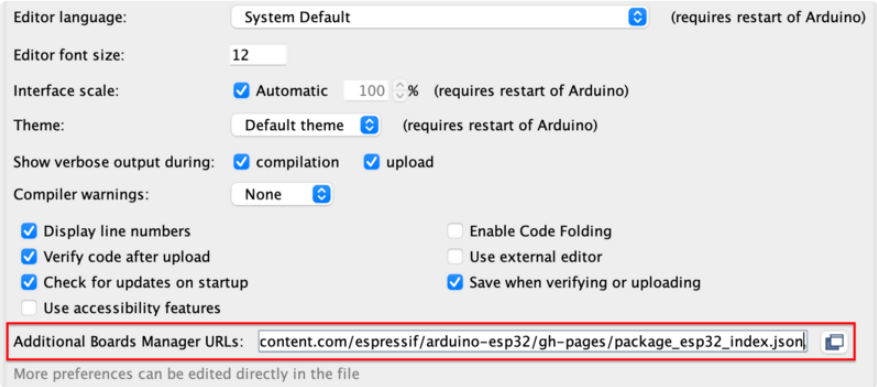


We will be adding a URL to the new **Additional Boards Manager URLs** option. The list of URLs is comma separated, and you will only have to add each URL once. New Adafruit boards and updates to existing boards will automatically be picked up by the Board Manager each time it is opened. The URLs point 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://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json
```



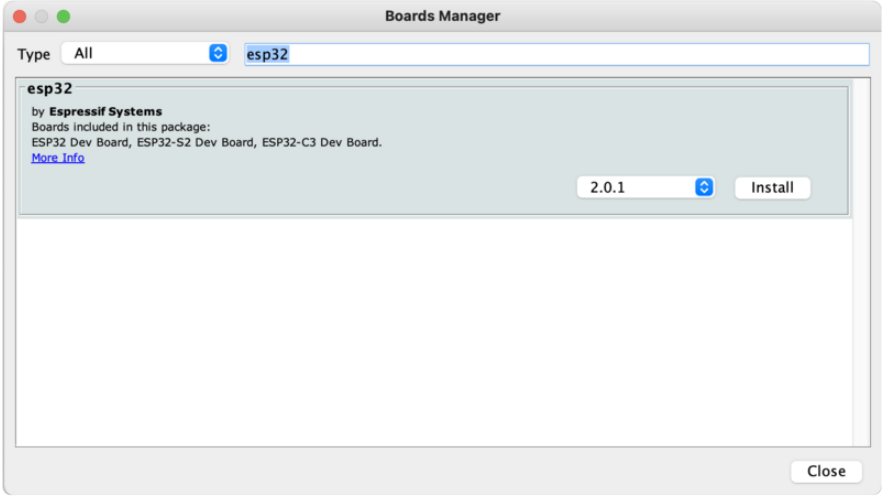
If you're an advanced hacker and want the 'bleeding edge' release that may have fixes (or bugs!) you can check out the dev url instead:

```
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_dev_index.json
```

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.

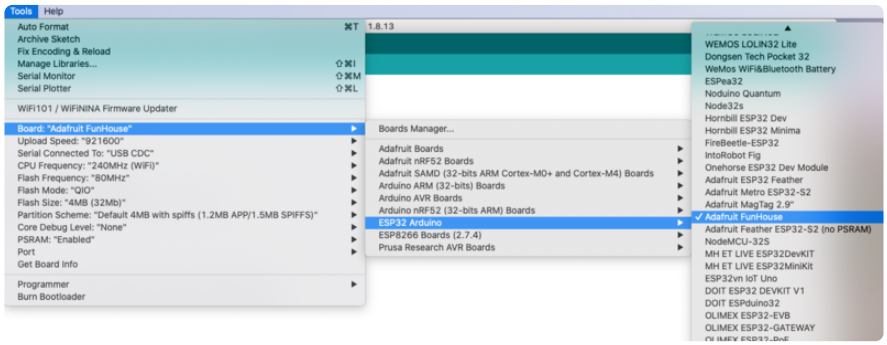
The next step is to actually install the Board Support Package (BSP). Go to the **Tools → Board → Board Manager** submenu. A dialog should come up with various BSPs. Search for **esp32**.



Click the **Install** button and wait for it to finish. Once it is finished, you can close the dialog.

In the **Tools → Board** submenu you should see **ESP32 Arduino** and in that dropdown it should contain the ESP32 boards along with all the latest ESP32-S2/S3 boards.

Look for the board called Adafruit FunHouse.



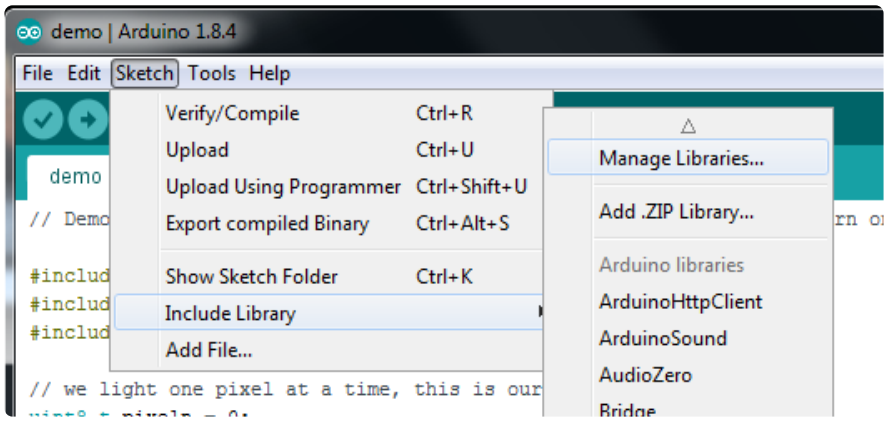
Arduino Libraries

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

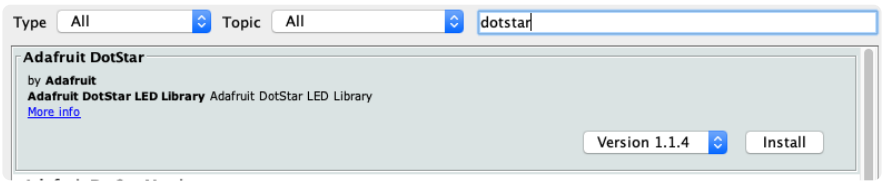
Open up the library manager...



And install the following libraries:

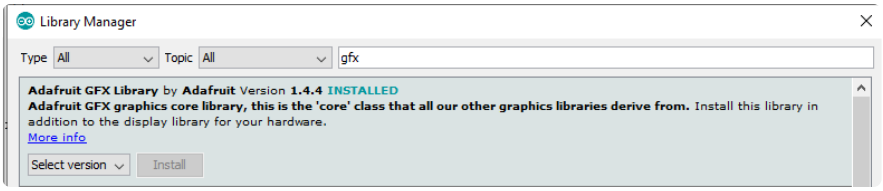
Adafruit DotStar

This will let you light up the status LEDs on the front



Adafruit GFX

This is the graphics library used to draw to the screen



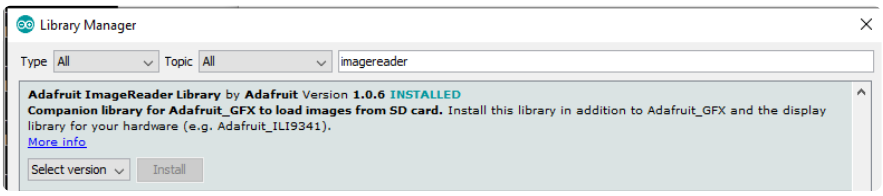
If using an older (pre-1.8.10) Arduino IDE, locate and install **Adafruit_BusIO** (newer versions do this automatically when installing Adafruit_GFX).

Adafruit ST7735 and ST7789

For using the display.

Adafruit ImageReader

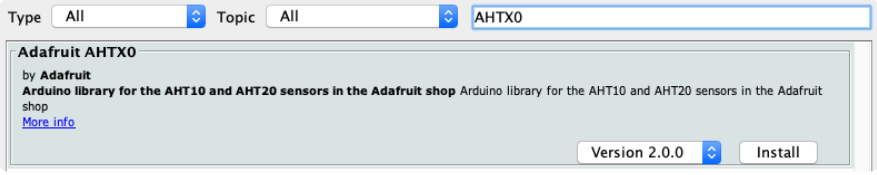
For reading bitmaps from SD and displaying



Adafruit AHTX0

For using the Humidity and Temperature Sensor.

Make sure to type the number Zero and not the Letter O when searching.



Adafruit DPS310

For using the Pressure Sensor

Arduino Basics

Arduino support for the ESP32-S2 is relatively new right now, so we recommend using CircuitPython!

Once you have Arduino installed and set up and you can upload simple blink sketches, you can move on to using each element of the FunHouse board.

Using the Red LED

It's always good to blink the LED when you want to verify if something is happening on your board. The LED is on IO #13, but we recommend you use the `LED_BUILTIN` macro and you can use this simple sketch example to blink the LED:

```
void setup() {
  // initialize built in LED pin as an output.
  pinMode(LED_BUILTIN, OUTPUT);
  // initialize USB serial converter so we have a port created
  Serial.begin();
}

// the loop function runs over and over again forever
```

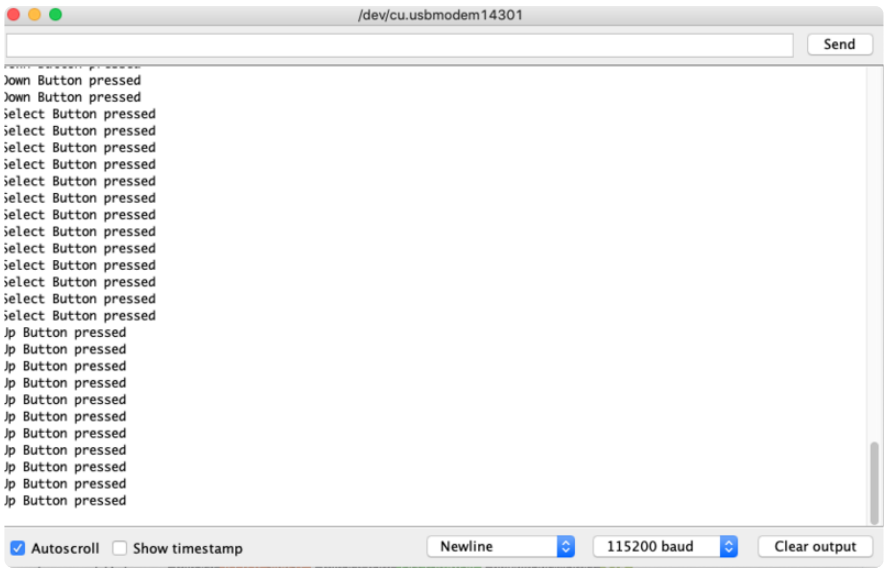
```
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                       // wait for a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);                       // wait for a second
}
```

Reading the Buttons

There are three buttons on the front of the FunHouse - they're connected to digital pins IO **3**, **4**, and **5**. However, we recommend you use the constants `BUTTON_DOWN`, `BUTTON_SELECT`, `BUTTON_UP`.

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

  pinMode(BUTTON_DOWN, INPUT_PULLDOWN);
  pinMode(BUTTON_SELECT, INPUT_PULLDOWN);
  pinMode(BUTTON_UP, INPUT_PULLDOWN);
}

void loop() {
  if (digitalRead(BUTTON_DOWN)) {
    Serial.println("Down Button pressed");
  }
  if (digitalRead(BUTTON_SELECT)) {
    Serial.println("Select Button pressed");
  }
  if (digitalRead(BUTTON_UP)) {
    Serial.println("Up Button pressed");
  }

  // small debugging delay
  delay(10);
}
```

Open the serial console and press buttons to see the serial output printed!



Reading the Capacitive Touch Pads

To read the value of the capacitive touch pads, you can use the `touchRead()` function, which is part of the ESP32 package. To use it, you only need to provide the GPIO pin number. The FunHouse uses IO #6, #7, and #8 for the button style touch pads and #9 through #13 along the slider.

So to get the value of IO #7, you would use the following code:

```
uint16_t touchread;

touchread = touchRead(7);
if (touchread &gt; 20000 ) {
  // Do Something
}
```

You may want to try adjusting the threshold for your needs. A more complete example can be found on the **Arduino Self Test Example** page.

Using On-Board DotStars

There are 4 DotStar LEDs on pin IO #14 and #15 (we recommend using the macro `PIN_DOTSTAR_DATA` and `PIN_DOTSTAR_CLOCK`).

Here's an example sketch that initializes the DotStar LEDs and color cycles them through a rainbow of different colors.

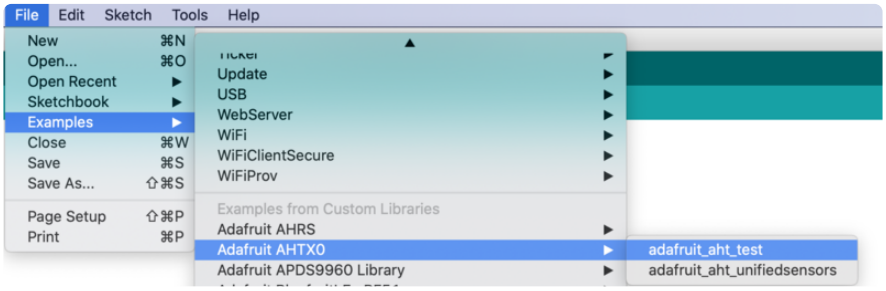
```
// SPDX-FileCopyrightText: 2021 Melissa LeBlanc-Williams for Adafruit Industries
//
// SPDX-License-Identifier: MIT

#include <Adafruit_DotStar.h>

#define NUM_DOTSTAR 5

// LEDs!
Adafruit_DotStar pixels(NUM_DOTSTAR, PIN_DOTSTAR_DATA, PIN_DOTSTAR_CLOCK,
DOTSTAR_BRG);

uint16_t firstPixelHue = 0;
uint8_t LED_dutycycle = 0;

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

  pinMode(LED_BUILTIN, OUTPUT);

#if ESP_IDF_VERSION >= ESP_IDF_VERSION_VAL(5, 1, 1)
  ledcAttach(LED_BUILTIN, 5000, 8);
#else
  ledcSetup(0, 5000, 8);
  ledcAttachPin(LED_BUILTIN, 0);
#endif
```

```
  pixels.begin(); // Initialize pins for output
  pixels.show();  // Turn all LEDs off ASAP
  pixels.setBrightness(20);
}


void loop() {
  Serial.println("Hello!");

  // pulse red LED
#if ESP_IDF_VERSION >= ESP_IDF_VERSION_VAL(5, 1, 1)
  ledcWrite(LED_BUILTIN, LED_dutycycle++);
#else
  ledcWrite(0, LED_dutycycle++);
#endif

  // rainbow dotstars
  for (int i=0; i<pixels.numPixels(); i++) { // For each pixel in strip...
      int pixelHue = firstPixelHue + (i * 65536L / pixels.numPixels());
      pixels.setPixelColor(i, pixels.gamma32(pixels.ColorHSV(pixelHue)));
  }
  pixels.show(); // Update strip with new contents
  firstPixelHue += 256;

  delay(15);
}


void rainbow(int wait) {
  for(long firstPixelHue = 0; firstPixelHue < 5*65536; firstPixelHue += 256) {
    for(int i=0; i<pixels.numPixels(); i++) { // For each pixel in strip...
      int pixelHue = firstPixelHue + (i * 65536L / pixels.numPixels());
      pixels.setPixelColor(i, pixels.gamma32(pixels.ColorHSV(pixelHue)));
    }
    pixels.show(); // Update strip with new contents
    delay(wait);  // Pause for a moment
  }
}
```

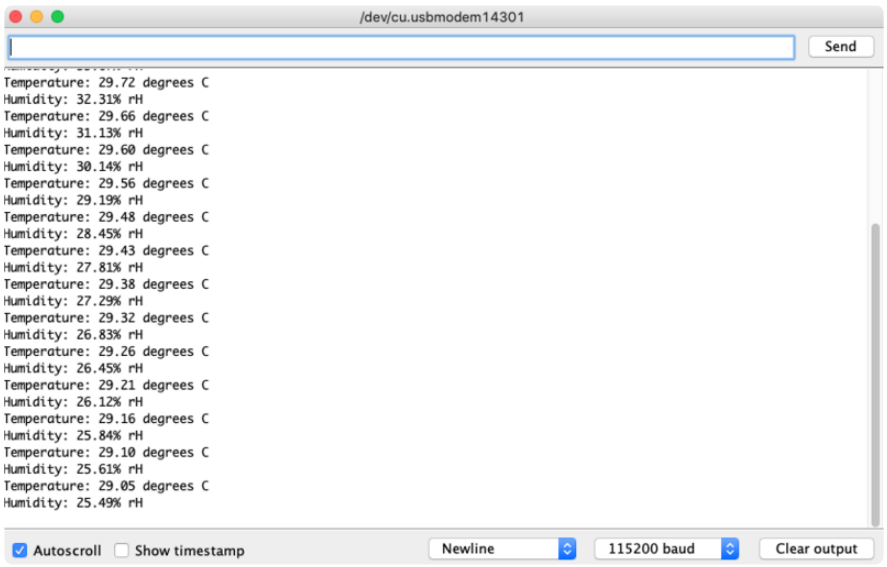
Using On-board Humidity and Temperature Sensor

There's a pre-soldered humidity and temperature sensor that you can use.

You can test the AHTX0 by loading the included **adafruit_aht_test** in the Arduino library



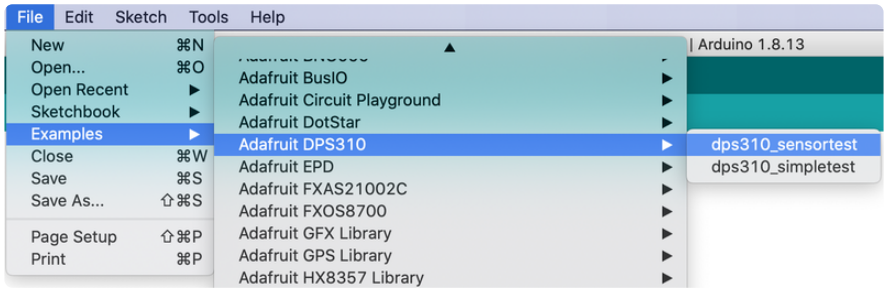
Now you can upload, reset, and check the serial port for temperature and humidity data!



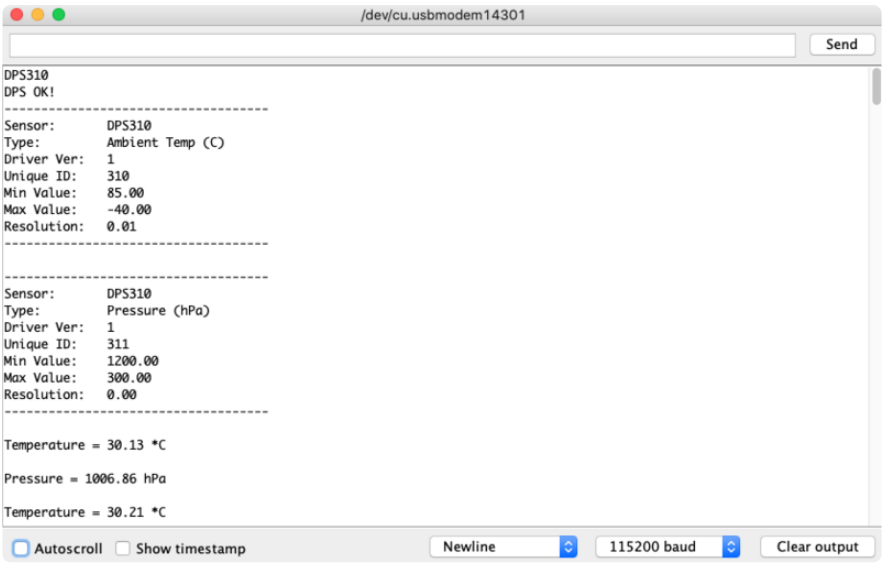
Using On-board Pressure Sensor

There's a pre-soldered pressure sensor that you can use.

You can test the DPS310 by loading the included **adafruit_sensor_test** in the Arduino library



Now you can upload, reset, and check the serial port for ambient temperature and pressure data!

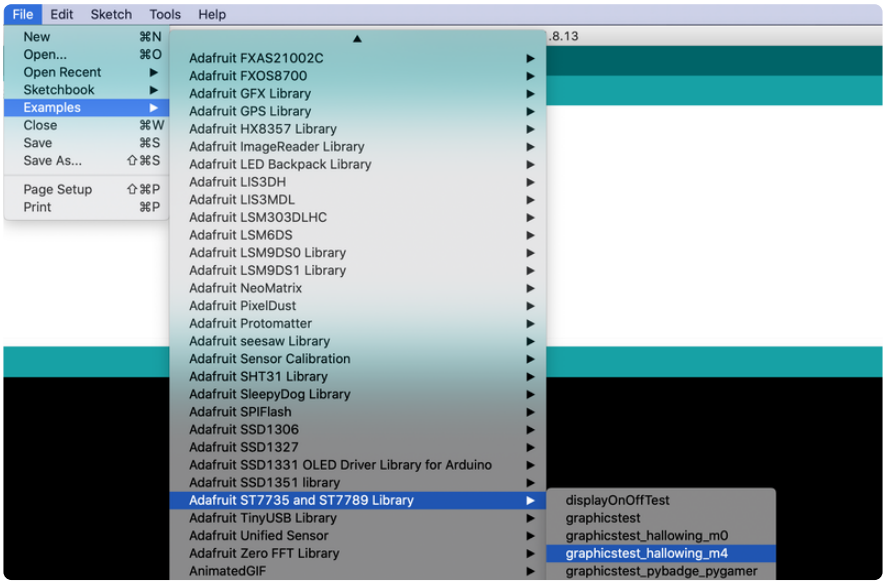


Using the TFT Display

You've been so patient, it's time to draw to the display!

We'll be using a demo that was written for the HalloWing M4, which has the same display, so only a couple of minor changes are needed.

From the **Adafruit ST7735 and ST7789 Library** folder, open the **graphicstest_hallowing_m4** example



Remove the pin definitions near the top since they are now part of the FunHouse Board Support Package.

```
#define TFT_CS        44 // PyBadge/PyGamer display control pins: chip select
#define TFT_RST       46 // Display reset
```

```
#define TFT_DC        45 // Display data/command select
#define TFT_BACKLIGHT 47 // Display backlight pin
```

Change the initialization line to the following:

```
Adafruit_ST7789 tft = Adafruit_ST7789(TFT_CS, TFT_DC, TFT_RESET);
```

You can now upload the example to your FunHouse to see it display various graphics and text tests.

For more information on how to display graphics, and text, check out the Adafruit GFX guide (https://adafru.it/doL)

Arduino Self Test Example

The **selftest.ino** sketch will initialize all the sensors and respond to the buttons and capacitive touch pads. Go ahead and upload it to your FunHouse and start pressing buttons. This will also slowly color-cycle the DotStar LEDs and blink the red LED.

FunHouse_SelfTest.UF2

https://adafru.it/RWB

```
// SPDX-FileCopyrightText: 2021 Melissa LeBlanc-Williams for Adafruit Industries
//
// SPDX-License-Identifier: MIT

#include <Adafruit_DotStar.h>
#include <Adafruit_GFX.h>    // Core graphics library
#include <Adafruit_ST7789.h> // Hardware-specific library for ST7789
#include <Adafruit_DPS310.h>
#include <Adafruit_AHTX0.h>

#define NUM_DOTSTAR 5
#define BG_COLOR ST77XX_BLACK
#define ST77XX_GREY 0x8410   // Colors are in RGB565 format

// display!
Adafruit_ST7789 tft = Adafruit_ST7789(TFT_CS, TFT_DC, TFT_RESET);
// LEDs!
Adafruit_DotStar pixels(NUM_DOTSTAR, PIN_DOTSTAR_DATA, PIN_DOTSTAR_CLOCK,
DOTSTAR_BRG);
// sensors!
Adafruit_DPS310 dps;
Adafruit_AHTX0 aht;

uint8_t LED_dutycycle = 0;
uint16_t firstPixelHue = 0;

void setup() {
  //while (!Serial);
  Serial.begin(115200);
  delay(100);

  pixels.begin(); // Initialize pins for output
  pixels.show();  // Turn all LEDs off ASAP
```

```
  pixels.setBrightness(20);

  pinMode(BUTTON_DOWN, INPUT_PULLDOWN);
  pinMode(BUTTON_SELECT, INPUT_PULLDOWN);
  pinMode(BUTTON_UP, INPUT_PULLDOWN);

  //analogReadResolution(13);

  tft.init(240, 240);                // Initialize ST7789 screen
  pinMode(TFT_BACKLIGHT, OUTPUT);
  digitalWrite(TFT_BACKLIGHT, HIGH); // Backlight on

  tft.fillScreen(BG_COLOR);
  tft.setTextSize(2);
  tft.setTextColor(ST77XX_YELLOW);
  tft.setTextWrap(false);

  // check DPS!
  tft.setCursor(0, 0);
  tft.setTextColor(ST77XX_YELLOW);
  tft.print("DP310? ");


  if (! dps.begin_I2C()) {
    tft.setTextColor(ST77XX_RED);
    tft.println("FAIL!");
    while (1) delay(100);
  }
  tft.setTextColor(ST77XX_GREEN);
  tft.println("OK!");
  dps.configurePressure(DPS310_64HZ, DPS310_64SAMPLES);
  dps.configureTemperature(DPS310_64HZ, DPS310_64SAMPLES);

  // check AHT!
  tft.setCursor(0, 20);
  tft.setTextColor(ST77XX_YELLOW);
  tft.print("AHT20? ");

  if (! aht.begin()) {
    tft.setTextColor(ST77XX_RED);
    tft.println("FAIL!");
    while (1) delay(100);
  }
  tft.setTextColor(ST77XX_GREEN);
  tft.println("OK!");

  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(SPEAKER, OUTPUT);

#if ESP_IDF_VERSION >= ESP_IDF_VERSION_VAL(5, 1, 1)
  ledcAttach(LED_BUILTIN, 2000, 8);
  ledcAttach(SPEAKER, 2000, 8);
  ledcWrite(SPEAKER, 0);
#else
  ledcSetup(0, 2000, 8);
  ledcAttachPin(LED_BUILTIN, 0);

  ledcSetup(1, 2000, 8);
  ledcAttachPin(SPEAKER, 1);
  ledcWrite(1, 0);
#endif
}



void loop() {


  /********************* sensors    */
```

```
  sensors_event_t humidity, temp, pressure;

  tft.setCursor(0, 0);
  tft.setTextColor(ST77XX_YELLOW, BG_COLOR);
  dps.getEvents(&temp, &pressure);

  tft.print("DP310: ");
  tft.print(temp.temperature, 0);
  tft.print(" C ");
  tft.print(pressure.pressure, 0);
  tft.print(" hPa");
  tft.println("              ");
  Serial.printf("DPS310: %0.1f *C  %0.2f hPa\n", temp.temperature,
pressure.pressure);


  tft.setCursor(0, 20);
  tft.setTextColor(ST77XX_YELLOW, BG_COLOR);
  aht.getEvent(&humidity, &temp);

  tft.print("AHT20: ");
  tft.print(temp.temperature, 0);
  tft.print(" C ");
  tft.print(humidity.relative_humidity, 0);
  tft.print(" %");
  tft.println("              ");
  Serial.printf("AHT20: %0.1f *C  %0.2f rH\n", temp.temperature,
humidity.relative_humidity);


  /****************** BUTTONS */
  tft.setCursor(0, 40);
  tft.setTextColor(ST77XX_YELLOW);
  tft.print("Buttons: ");
  if (! digitalRead(BUTTON_DOWN)) {
    tft.setTextColor(ST77XX_GREY);
  } else {
    Serial.println("DOWN pressed");
    tft.setTextColor(ST77XX_WHITE);
  }
  tft.print("DOWN ");

  if (! digitalRead(BUTTON_SELECT)) {
    tft.setTextColor(ST77XX_GREY);
  } else {
    Serial.println("SELECT pressed");
    tft.setTextColor(ST77XX_WHITE);
  }
  tft.print("SEL ");

  if (! digitalRead(BUTTON_UP)) {
    tft.setTextColor(ST77XX_GREY);
  } else {
    Serial.println("UP pressed");
    tft.setTextColor(ST77XX_WHITE);
  }
  tft.println("UP");

  /************************** CAPACITIVE */
  uint16_t touchread;

  tft.setCursor(0, 60);
  tft.setTextColor(ST77XX_YELLOW, BG_COLOR);
  tft.print("Captouch 6: ");
  touchread = touchRead(6);
  if (touchread < 10000 ) {
    tft.setTextColor(ST77XX_GREY, BG_COLOR);
  } else {
    tft.setTextColor(ST77XX_WHITE, BG_COLOR);
```

```
}
tft.print(touchread);
tft.println("          ");
Serial.printf("Captouch #6 reading: %d\n", touchread);

tft.setCursor(0, 80);
tft.setTextColor(ST77XX_YELLOW, BG_COLOR);
tft.print("Captouch 7: ");
touchread = touchRead(7);
if (touchread < 20000 ) {
  tft.setTextColor(ST77XX_GREY, BG_COLOR);
} else {
  tft.setTextColor(ST77XX_WHITE, BG_COLOR);
}
tft.print(touchread);
tft.println("          ");
Serial.printf("Captouch #7 reading: %d\n", touchread);


tft.setCursor(0, 100);
tft.setTextColor(ST77XX_YELLOW, BG_COLOR);
tft.print("Captouch 8: ");
touchread = touchRead(8);
if (touchread < 20000 ) {
  tft.setTextColor(ST77XX_GREY, BG_COLOR);
} else {
  tft.setTextColor(ST77XX_WHITE, BG_COLOR);
}
tft.print(touchread);
tft.println("          ");
Serial.printf("Captouch #8 reading: %d\n", touchread);


/************************* ANALOG READ */
uint16_t analogread;

tft.setCursor(0, 120);
tft.setTextColor(ST77XX_YELLOW, BG_COLOR);
tft.print("Analog 0: ");
analogread = analogRead(A0);
if (analogread < 8000 ) {
  tft.setTextColor(ST77XX_WHITE, BG_COLOR);
} else {
  tft.setTextColor(ST77XX_RED, BG_COLOR);
}
tft.print(analogread);
tft.println("    ");
Serial.printf("Analog A0 reading: %d\n", analogread);


tft.setCursor(0, 140);
tft.setTextColor(ST77XX_YELLOW, BG_COLOR);
tft.print("Analog 1: ");
analogread = analogRead(A1);
if (analogread < 8000 ) {
  tft.setTextColor(ST77XX_WHITE, BG_COLOR);
} else {
  tft.setTextColor(ST77XX_RED, BG_COLOR);
}
tft.print(analogread);
tft.println("    ");
Serial.printf("Analog A1 reading: %d\n", analogread);


tft.setCursor(0, 160);
tft.setTextColor(ST77XX_YELLOW, BG_COLOR);
tft.print("Analog 2: ");
analogread = analogRead(A2);
if (analogread < 8000 ) {
```

```
    tft.setTextColor(ST77XX_WHITE, BG_COLOR);
  } else {
    tft.setTextColor(ST77XX_RED, BG_COLOR);
  }
  tft.print(analogread);
  tft.println("    ");
  Serial.printf("Analog A2 reading: %d\n", analogread);

  tft.setCursor(0, 180);
  tft.setTextColor(ST77XX_YELLOW, BG_COLOR);
  tft.print("Light: ");
  analogread = analogRead(A3);
  tft.setTextColor(ST77XX_WHITE, BG_COLOR);
  tft.print(analogread);
  tft.println("    ");
  Serial.printf("Light sensor reading: %d\n", analogread);

  /************************* Beep! */
  if (digitalRead(BUTTON_SELECT)) {
     Serial.println("** Beep! ***");
     fhtone(SPEAKER, 988.0, 100.0);  // tone1 - B5
     fhtone(SPEAKER, 1319.0, 200.0); // tone2 - E6
     delay(100);
     //fhtone(SPEAKER, 2000.0, 100.0);
  }

  /************************* LEDs */
  // pulse red LED
#if ESP_IDF_VERSION >= ESP_IDF_VERSION_VAL(5, 1, 1)
  ledcWrite(LED_BUILTIN, LED_dutycycle);
#else
  ledcWrite(0, LED_dutycycle);
#endif
  LED_dutycycle += 32;

  // rainbow dotstars
  for (int i=0; i<pixels.numPixels(); i++) { // For each pixel in strip...
      int pixelHue = firstPixelHue + (i * 65536L / pixels.numPixels());
      pixels.setPixelColor(i, pixels.gamma32(pixels.ColorHSV(pixelHue)));
  }
  pixels.show(); // Update strip with new contents
  firstPixelHue += 256;
}


void fhtone(uint8_t pin, float frequency, float duration) {
#if ESP_IDF_VERSION >= ESP_IDF_VERSION_VAL(5, 1, 1)
  ledcAttach(SPEAKER, frequency, 8);
  ledcWrite(SPEAKER, 128);
  delay(duration);
  ledcWrite(SPEAKER, 0);
#else
  ledcSetup(1, frequency, 8);
  ledcAttachPin(pin, 1);
  ledcWrite(1, 128);
  delay(duration);
  ledcWrite(1, 0);
#endif
}
```

WipperSnapper Setup

The WipperSnapper firmware and ecosystem are in BETA and are actively being developed to add functionality, more boards, more sensors, and fix bugs. We

encourage you to try out WipperSnapper with the understanding that it is not final release software and is still in development.

If you encounter any bugs, glitches, or difficulties during the beta period, or with this guide, please contact us via http://io.adafruit.com/support

What is WipperSnapper

WipperSnapper is a firmware designed to turn any WiFi-capable board into an Internet-of-Things device without programming a single line of code. WipperSnapper connects to Adafruit IO (https://adafru.it/fsU), a web platform designed (by Adafruit! (https://adafru.it/Bo5)) to display, respond, and interact with your project's data.

Simply load the WipperSnapper firmware onto your board, add credentials, and plug it into power. Your board will automatically register itself with your Adafruit IO account.

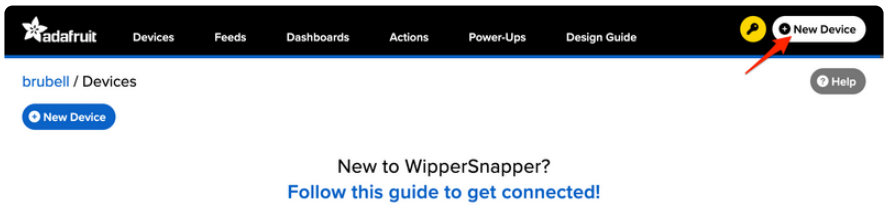
From there, you can add components to your board such as buttons, switches, potentiometers, sensors, and more! Components are dynamically added to hardware, so you can immediately start interacting, logging, and streaming the data your projects produce without writing code.

Sign up for Adafruit.io

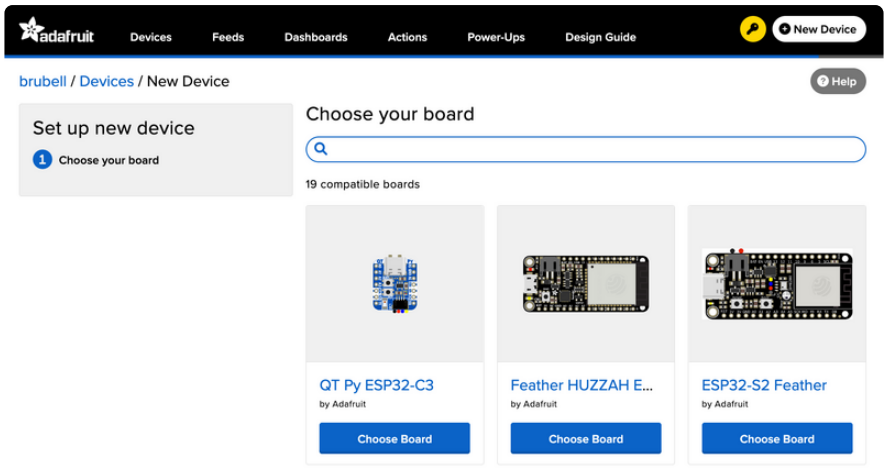
You will need an Adafruit IO account to use WipperSnapper on your board. If you do not already have one, head over to io.adafruit.com (https://adafru.it/fsU) to create a free account.

Add a New Device to Adafruit IO

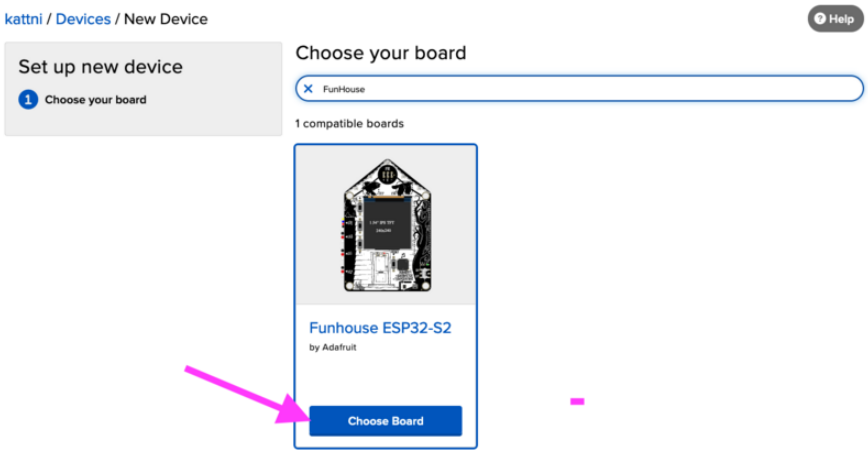
Log into your Adafruit IO (https://adafru.it/fsU) account. Click the New Device button at the top of the page.



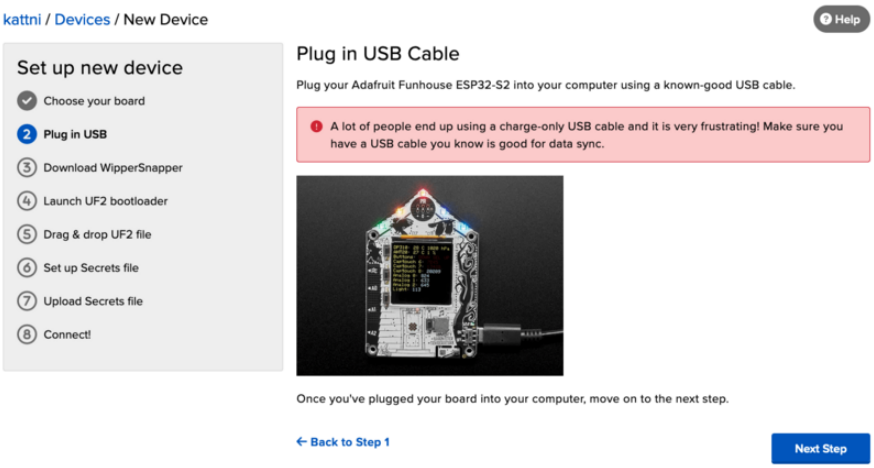
After clicking New Device, you should be on the board selector page. This page displays every board that is compatible with the WipperSnapper firmware.



In the board selector page's search bar, search for the FunHouse. Once you've located the board you'd like to install WipperSnapper on, click the **Choose Board** button to bring you to the self-guided installation wizard.

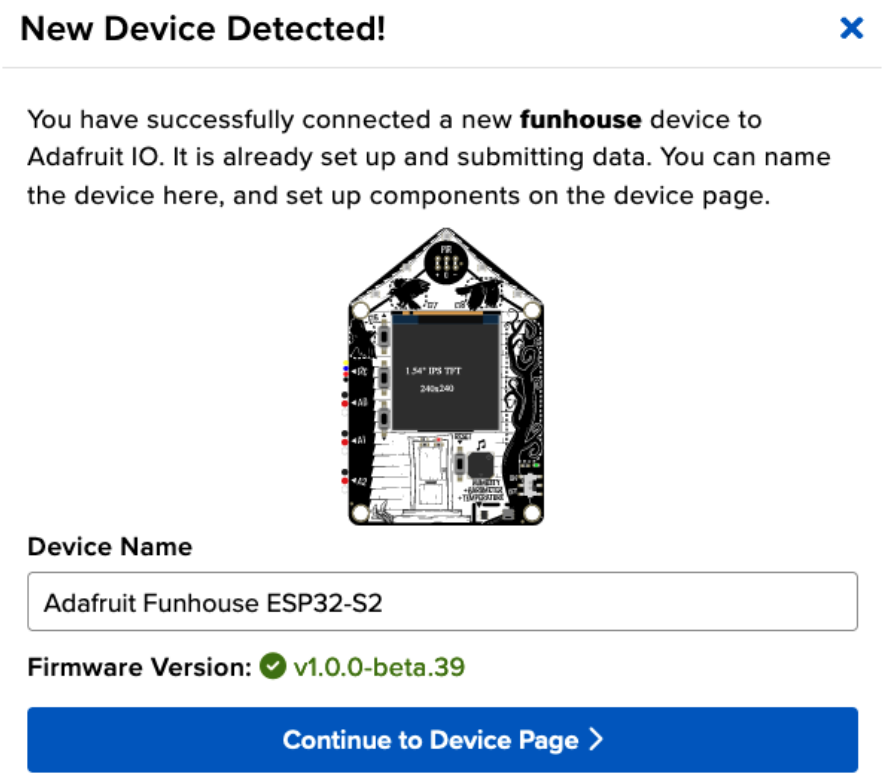


Follow the step-by-step instructions on the page to install Wippersnapper on your device and connect it to Adafruit IO.



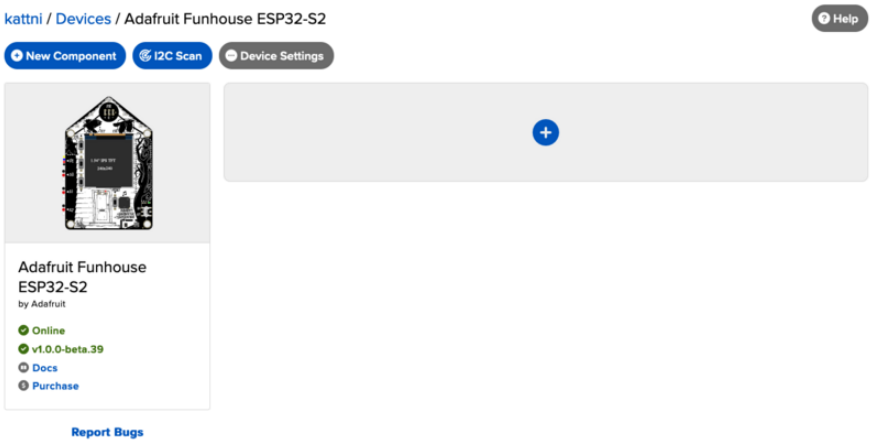
If the installation was successful, a popover should appear displaying that your board has successfully been detected by Adafruit IO.

Give your board a name and click "Continue to Device Page".



You should be brought to your board's device page.

Next, Visit this guide's **WipperSnapper Essentials** pages to learn how to interact with your board using Adafruit IO.



Feedback

Adafruit.io WipperSnapper is in **beta** and you can help improve it!

If you have suggestions or general feedback about the installation process - visit https://io.adafruit.com/support (https://adafru.it/Sgb), click "Contact Adafruit IO Support" and select "I have feedback or suggestions for the WipperSnapper Beta".

Troubleshooting

If you encountered an issue during installation, please try the steps below first.

If you're still unable to resolve the issue, or if your issue is not listed below, get in touch with us directly at https://io.adafruit.com/support (https://adafru.it/Sgb). Make sure to click "Contact Adafruit IO Support" and select "There is an issue with WipperSnapper. Something is broken!"

I don't see my board on Adafruit IO, it is stuck connecting to WiFi

First, make sure that you selected the correct board on the board selector.

Next, please make sure that you entered your WiFi credentials properly, there are no spaces/special characters in either your network name (SSID) or password, and that you are connected to a 2.4GHz wireless network.

If you're still unable to connect your board to WiFi, please make a new post on the WipperSnapper technical support forum with the error you're experiencing, the LED colors which are blinking, and the board you're using. (https://adafru.it/V6a)

I don't see my board on Adafruit IO, it is stuck "Registering with Adafruit IO"

Try hard-resetting your board by unplugging it from USB power and plugging it back in.

If the error is still occurring, please make a new post on the WipperSnapper technical support forum with information about what you're experiencing, the LED colors which are blinking (if applicable), and the board you're using. (https://adafru.it/V6a)

"Uninstalling" WipperSnapper

WipperSnapper firmware is an application that is loaded onto your board. There is nothing to "uninstall". However, you may want to "move" your board from running

WipperSnapper to running Arduino or CircuitPython. You also may need to restore your board to the state it was shipped to you from the Adafruit factory.

Moving from WipperSnapper to CircuitPython

Follow the steps on the Installing CircuitPython page (https://adafru.it/Amd) to install CircuitPython on your board running WipperSnapper.

- If you are unable to double-tap the RST button to enter the UF2 bootloader, follow the "Factory Resetting a WipperSnapper Board" instructions below.

Uploading this sketch will overwrite WipperSnapper. If you want to re-install WipperSnapper, follow the instructions at the top of this page.

Moving from WipperSnapper to Arduino

If you want to use your board with Arduino, you will use the Arduino IDE to load any sketch onto your board.

First, follow the page below to set up your Arduino IDE environment for use with your board.

Setup Arduino IDE

https://adafru.it/10eC

Then, follow the page below to upload the "Arduino Blink" sketch to your board.

Upload Arduino Blink/Self Test Sketch

https://adafru.it/10eD

Uploading this sketch will overwrite WipperSnapper. If you want to re-install WipperSnapper, follow the instructions at the top of this page.

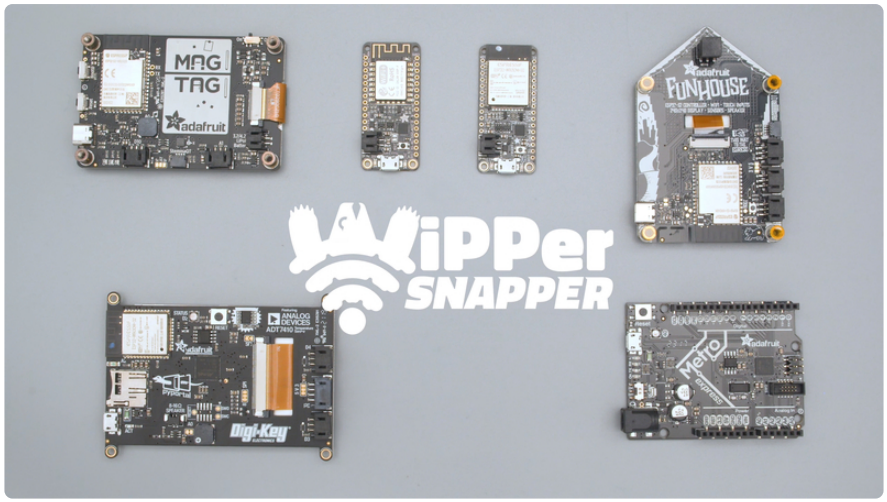
Factory Resetting a WipperSnapper Board

Sometimes, hardware gets into a state that requires it to be "restored" to the original state it shipped in. If you'd like to get your board back to its original factory state, follow the guide below.

Factory Reset Adafruit FunHouse

https://adafru.it/XRc

WipperSnapper Essentials



You've installed WipperSnapper firmware on your board and connected it to Adafruit IO. Next, to learn how to use Adafruit IO!

The Adafruit IO supports a large number of components. Components are physical parts such as buttons, switches, sensors, servos, LEDs, RGB LEDs, and more.

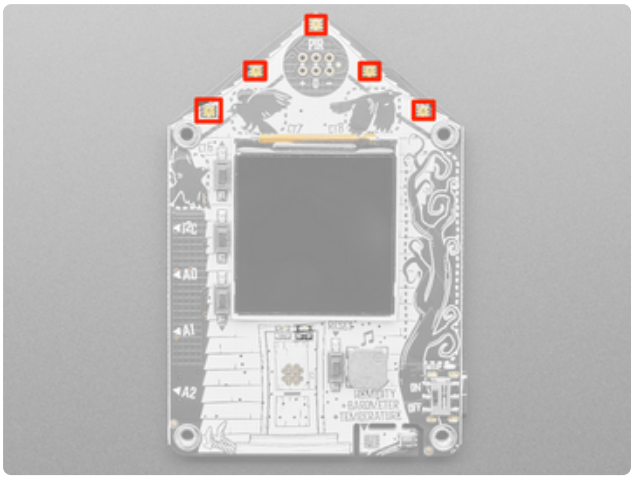
The following pages will get you up and running with WipperSnapper as you interact with your board's built-in components, such as read the value of a push button, send the value of an I2C sensor to the internet, and wirelessly control colorful LEDs.

DotStar LEDs

Your board has multiple APA102 (DotStar, in Adafruit jargon) RGB LEDs built in. Boards running the WipperSnapper firmware can be wirelessly controlled by Adafruit IO to interact with Dotstars.

On this page, you'll learn how to change the color and brightness of the DotStars built into your board from Adafruit IO.

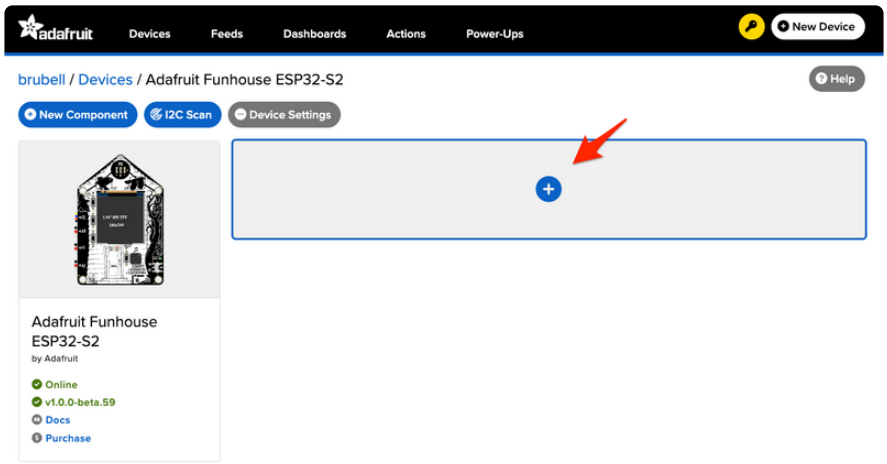
Where are the DotStars on my board?

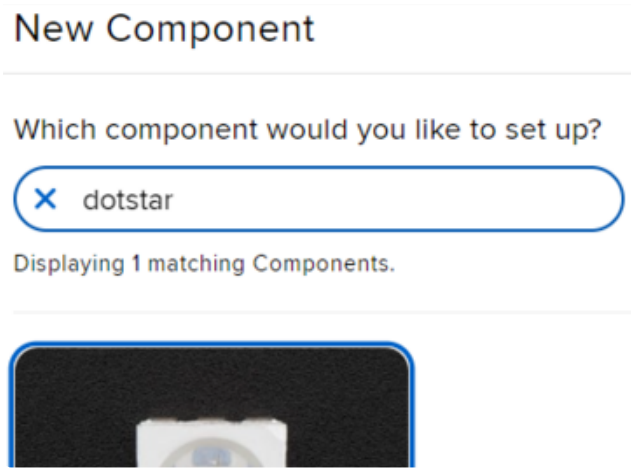


The FunHouse has 5 DotStar RGB LEDs (highlighted in red) located at the top of the board.

Create a DotStar Component

On the device page, click the New Component (or `+`) button to open the component picker.





Search for the component name by entering `dotstar` into the text box on the component picker, the list of components should update as soon as you stop typing

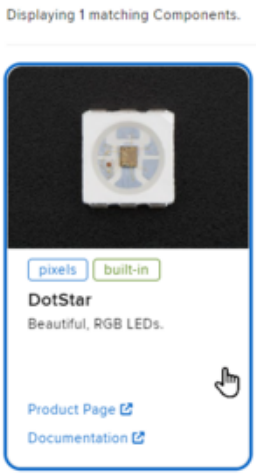
Filtering and searching for components

WipperSnapper supports such a large number of components we added filtering! Try searching for various keywords, like:

- component names: `aht20`, `servo`, `buzzer`, `button`, `dotstar`, etc
- sensor types: `light`, `temperature`, `pressure`, `humidity`, etc
- interface: `i2c`, `uart`, `ds18x20`, `pin`, etc (also I2C addresses e.g. `0x44`)
- vendor: `Adafruit`, `ASAIR`, `Infineon`, `Bosch`, `Honeywell`, `Sensirion`, etc

We've also added product and documentation links to every component, follow the links beneath the component descriptions to be taken to the appropriate product page or Learn-Guide

---



Select the **DotStar** from the list of results to go to the component configuration page.

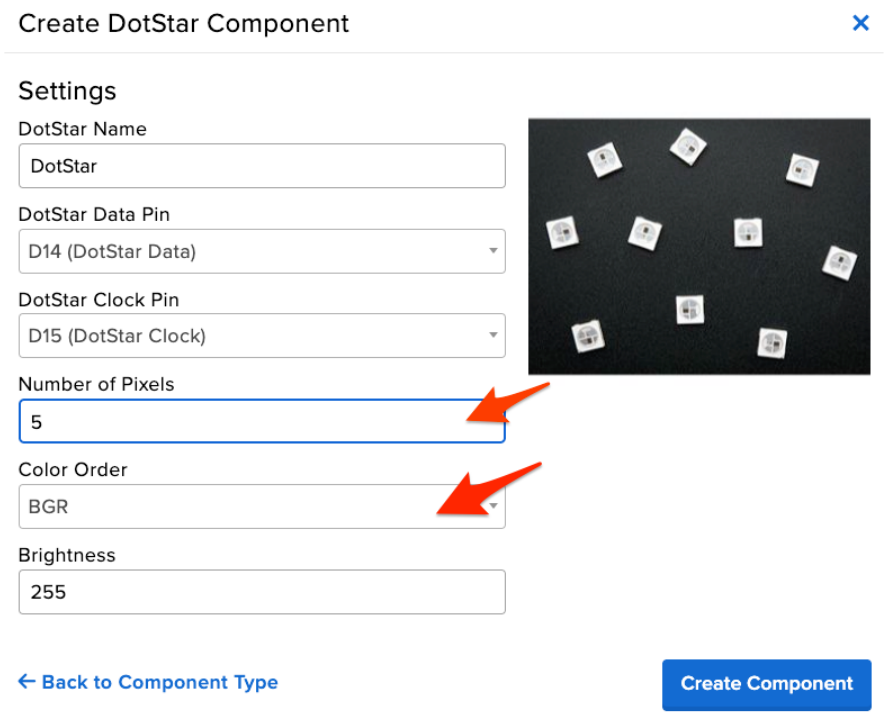
There will be a back button if you select the wrong component, and you can use the Edit component icon (⚙) on the device page to update the component configuration in the future.

The board's DotStar data and clock pins are automatically found and selected.

The FunHouse contains five DotStar pixels. Set the **Number of Pixels** to 5.

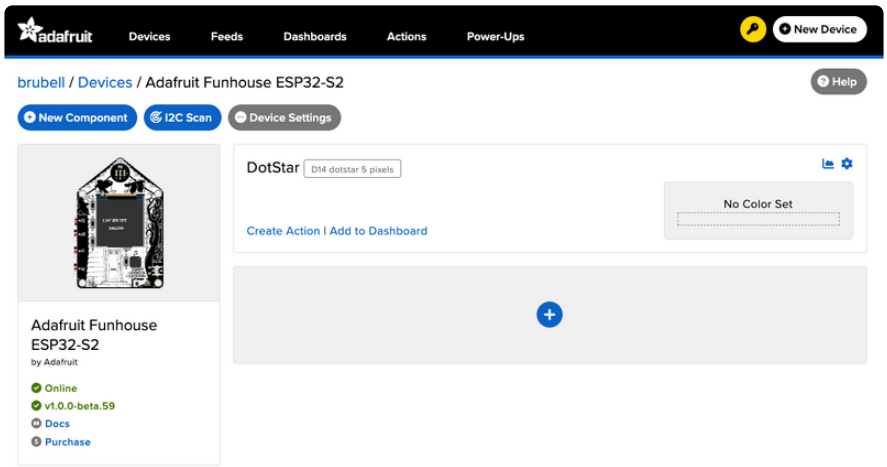
The color order used by the FunHouse's DotStar strand is not the default BRG ordering. Set the **Color Order** to `BGR`.

Click **Create Component**



Behind the scenes, Adafruit IO sends a command to your board running WipperSnapper firmware telling it to initialize a new DotStar strand with the settings from the form.

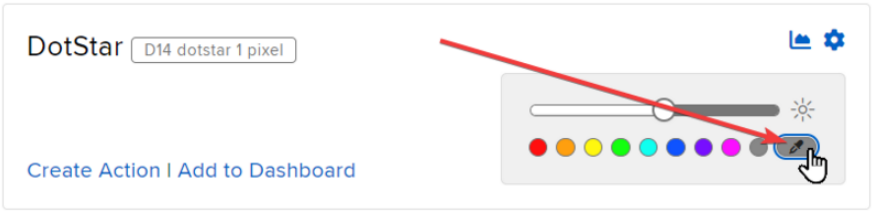
The Device page shows the DotStar component.



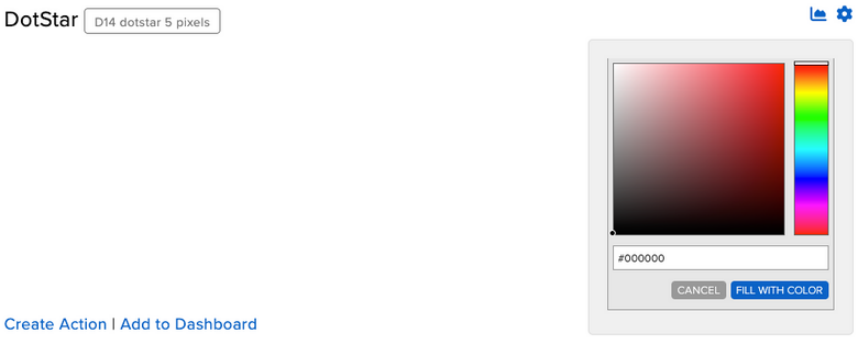
Set the DotStar's RGB Color

Since no colors have been set yet, the color picker's default value is `#000000` (black in hex color code) and appears "off". Let's change that to make the DotStars shine brightly!

On the device page, click the color dropper at the end of the color swatch list.



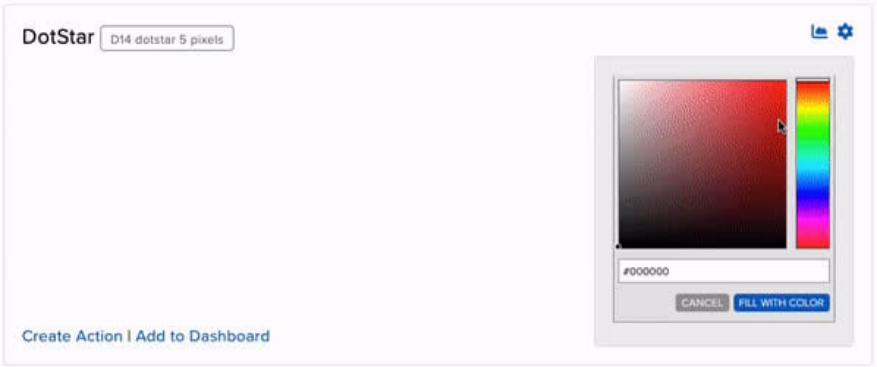
The Dotstar component should expand, revealing its color picker.



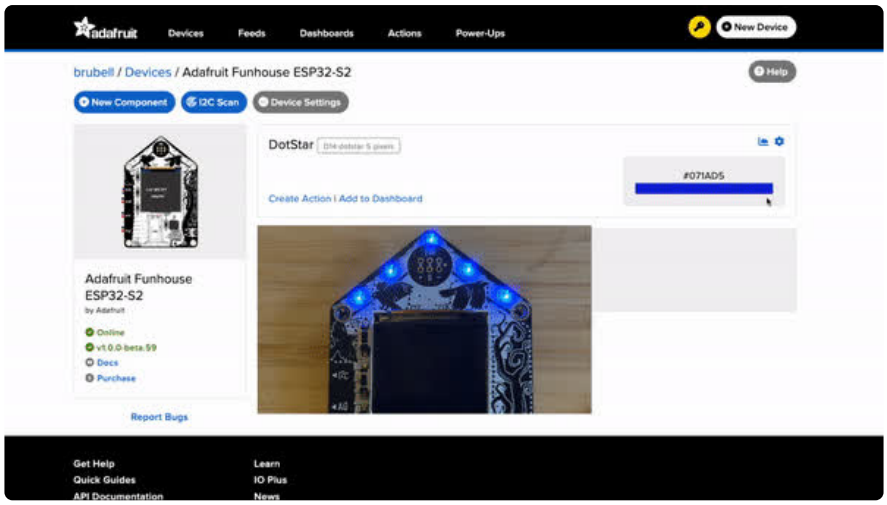
Hex Colors 101

The color picker on Adafruit IO uses hex color codes to represent Red, Green, and Blue values. For example, `#FF0000` is the hex color code for the color red. The colors (`#FF0000`) red component is `FF` (255 translated to decimal), the green component is `00` and the blue component is `00`. Translated to RGB format, the color is `RGB (255, 0, 0)`.

Using the color picker, or by manually entering a hex color code, select a color.

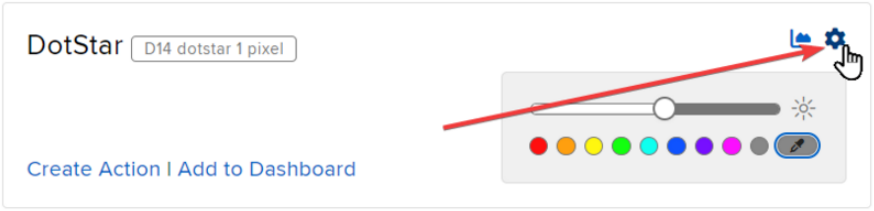


When you're ready to set the color of your device's DotStars, click **FILL WITH COLOR**. The DotStars on your board will glow with the color you selected!



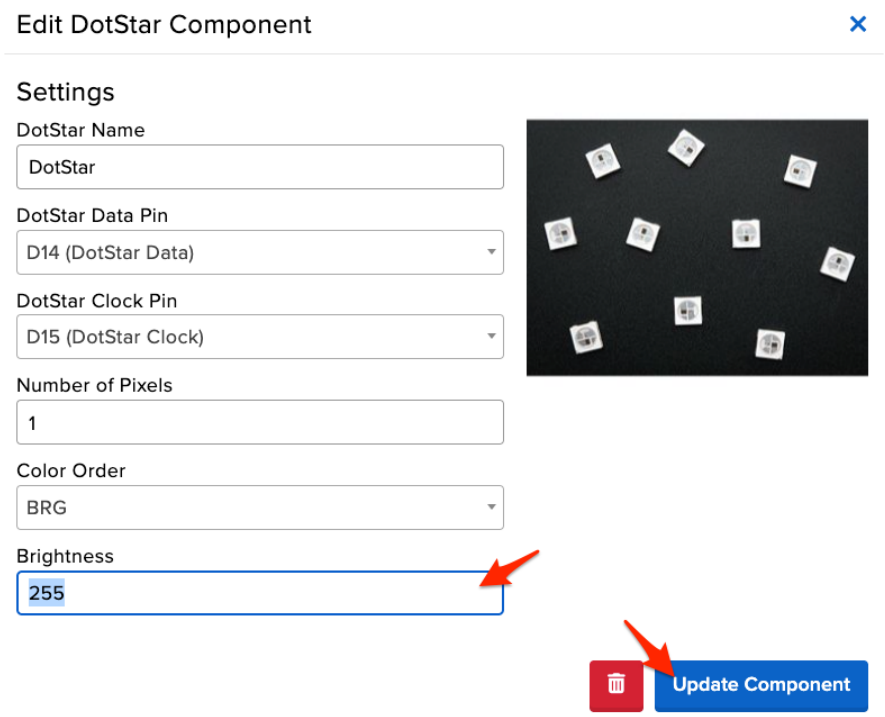
Set DotStar Brightness

If the DotStar strand is too bright (or too dim), you can change the overall brightness. Click the gear/cog icon on the DotStar component to open its settings.



On the DotStar component form, set Brightness to a value between 0 (fully off) and 255 (full brightness).

Click the **Update Component** button to send the updated configuration to your device.



DotStar FAQ

I'm getting the wrong colors. Red and blue are swapped!

Different versions of DotStar LEDs expect to receive color data in a different order... and occasionally it may change if it improves production efficiency or yield.

If you are having this issue - Try changing the Color Order setting (within the DotStar component's settings) until you find one that works with your hardware.

Why does the color on my DotStar not look exactly like it does on the color picker?

WipperSnapper firmware automatically performs gamma correction using the `gamma32` function in the Adafruit_DotStar library (https://adafru.it/18pB). However, it may not be completely identical to the gamma correction used by your operating system or monitor.

- For more information about Gamma Correction, visit this guide... (https://adafru.it/w2B)

I want to set the colors of individual pixels on my DotStar strand

There are future plans for setting the colors of individual pixels, but at this time DotStar support on WipperSnapper currently only supports filling an entire strand of pixels with one color.

Does this support DotStar Matrixes?

WipperSnapper does not currently support NxN matrices of DotStars at this time.

Read a Push-button

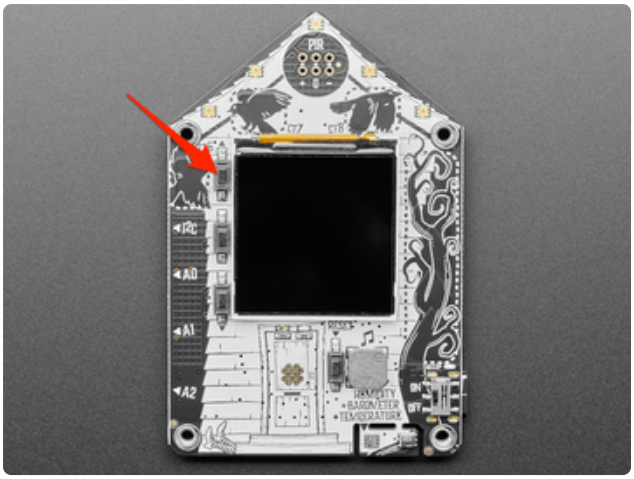
This demo shows reading the state of a push-button from WipperSnapper. But the same kind of control can be used for reading switches, break beam sensors, and other digital sensors.

You can configure a board running WipperSnapper to read data from standard input buttons, switches, or digital sensors, and send the value to Adafruit IO.

From Adafruit IO, you will configure one of the pushbuttons on your board as a push button component. Then, when the button is pressed (or released), a value will be published to Adafruit IO.

Button Location

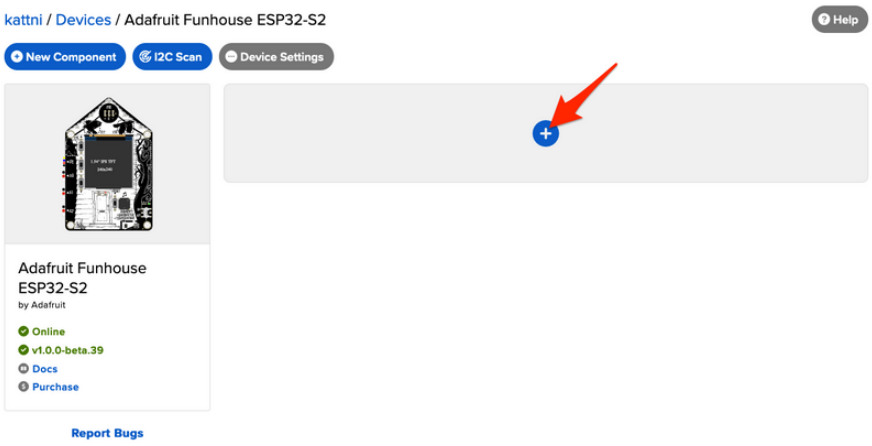
This example uses the board's built-in push-button and internal pull-up resistor instead of wiring a push-button up.

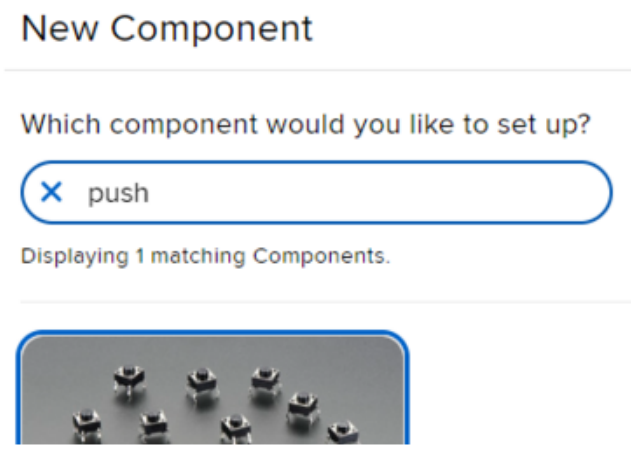


On the front of the board, to the left of the display, there are **three user-controllable buttons** labeled arrows for the top and bottom buttons. In the example on this page, we're going to use the top button (pointed to by the red arrow).

Create a Push-button Component on Adafruit IO

On the device page, click the New Component (or "+") button to open the component picker.





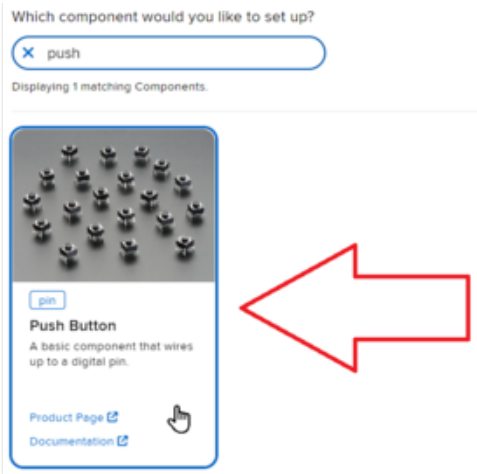
Search for the component name by entering `push` into the text box on the component picker, the list of components should update as soon as you stop typing.

Filtering and searching for components

Since WipperSnapper supports such a large number of components, you can use filtering. Try searching for various keywords, like:

- component names: `aht20`, `servo`, `buzzer`, `button`, `potentiometer`, etc
- sensor types: `light`, `temperature`, `pressure`, `humidity`, etc
- interface: `i2c`, `uart`, `ds18x20`, `pin`, etc (also I2C addresses e.g. `0x44`)
- vendor: `Adafruit`, `ASAIR`, `Infineon`, `Bosch`, `Honeywell`, `Sensirion`, etc

There are also added product and documentation links for every component. Follow the links beneath the component descriptions to be taken to the appropriate product page or Learn Guide.

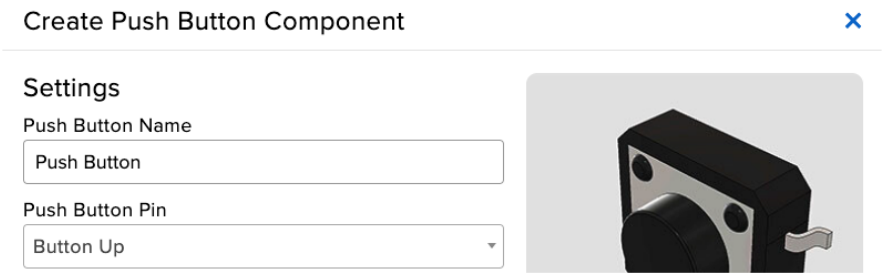


Select the **Push Button** from the list of results to go to the component configuration page.

There will be a back button if you select the wrong component, and you can use the Edit component icon (⚙) on the device page to update the component configuration in the future.

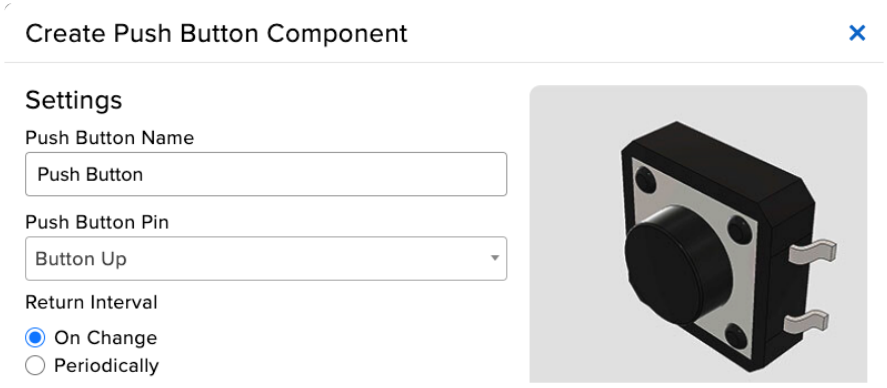
The "Create Push Button Component" form presents you with options for configuring the push button.

Start by selecting the board's pin connected to the push button.

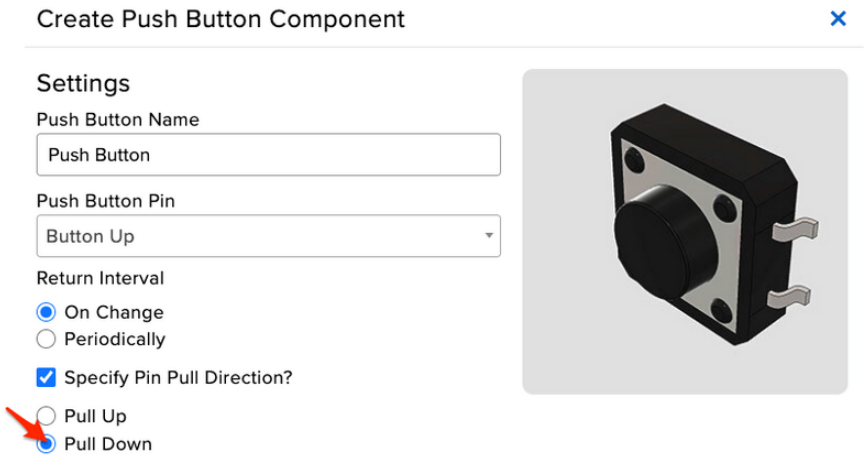


The Return Interval dictates how frequently the value of the push-button will be sent from the board to Adafruit IO.

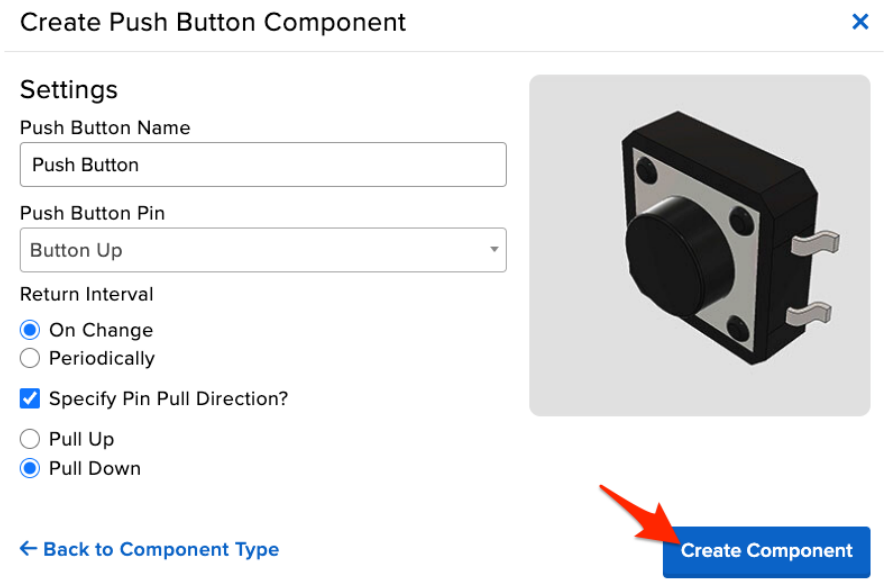
For this example, you will configure the push button's value to be only sent when the value changes (i.e. when it's either pressed or depressed).



Finally, check the **Specify Pin Pull Direction** checkbox and select the pull direction.

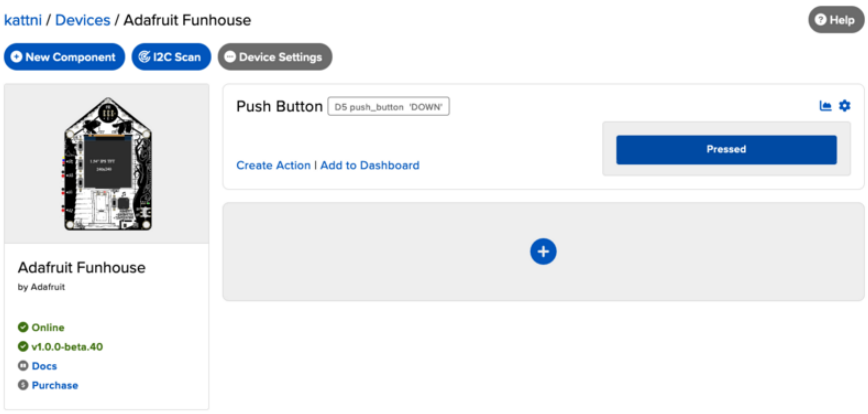


Make sure the form's settings look like the following screenshot. Then, click **Create Component**.

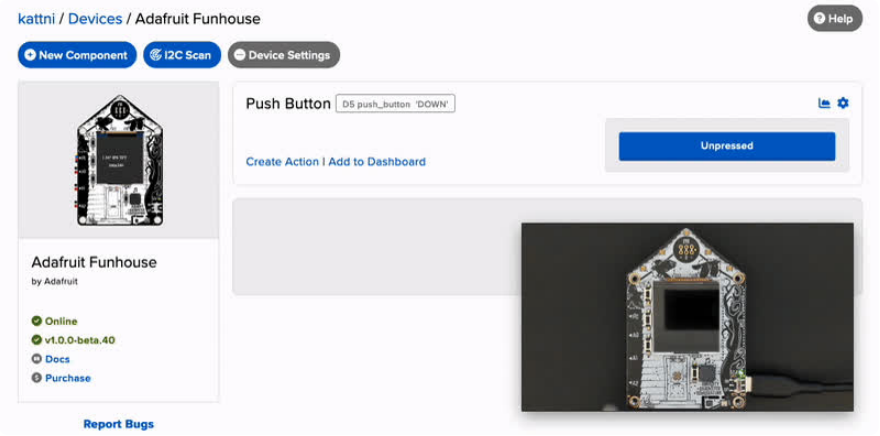


Adafruit IO sends a command to your WipperSnapper board, telling it to configure the GPIO pin you selected to behave as a digital input pin and to enable it to pull up the internal resistor.

Your board's page should also show the new push-button component.



Push the button on your board to change the value of the push-button component on Adafruit IO.



Analog Input: Light Sensor

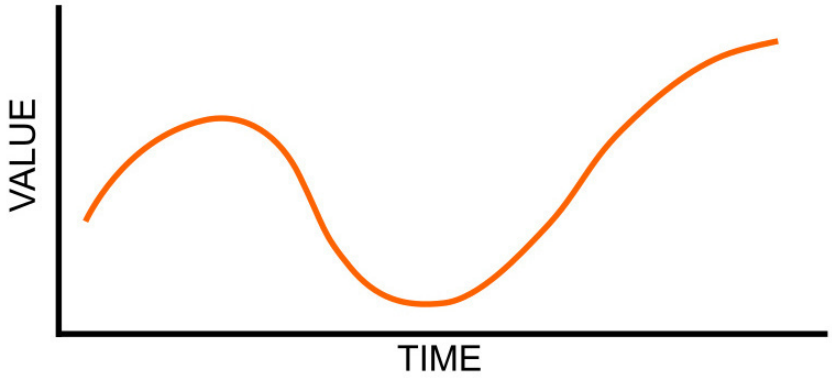
This page demonstrates using a light sensor as an analog input. However, the same process can be used for other analog input components on Adafruit IO such as the potentiometer.

Your microcontroller board has both digital and analog signal capabilities. Some pins are analog, some are digital, and some are capable of both. Check the **Pinouts** page in this guide for details about your board.

Analog signals are different from digital signals in that they can be any voltage and can vary continuously and smoothly between voltages. An analog signal is like a dimmer switch on a light, whereas a digital signal is like a simple on/off switch.

Digital signals only can ever have two states, they are either are **on** (high logic level voltage like 3.3V) or **off** (low logic level voltage like 0V / ground).

By contrast, analog signals can be any voltage in-between on and off, such as 1.8V or 0.001V or 2.98V and so on.



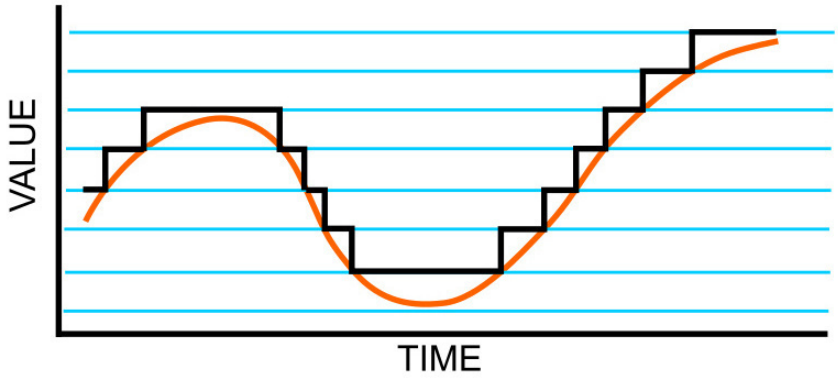
Analog signals are continuous values which means they can be an infinite number of different voltages. Think of analog signals like a floating point or fractional number, they can smoothly transiting to any in-between value like 1.8V, 1.81V, 1.801V, 1.8001V, 1.80001V and so forth to infinity.

Many devices use analog signals, in particular sensors typically output an analog signal or voltage that varies based on something being sensed like light, heat, humidity, etc.

Analog to Digital Converter (ADC)

An analog-to-digital-converter, or ADC, is the key to reading analog signals and voltages with a microcontroller. An ADC is a device that reads the voltage of an analog signal and converts it into a digital, or numeric, value. The microcontroller can't read analog signals directly, so the analog signal is first converted into a numeric value by the ADC.

The black line below shows a digital signal over time, and the red line shows the converted analog signal over the same amount of time.



Once that analog signal has been converted by the ADC, the microcontroller can use those digital values any way you like!

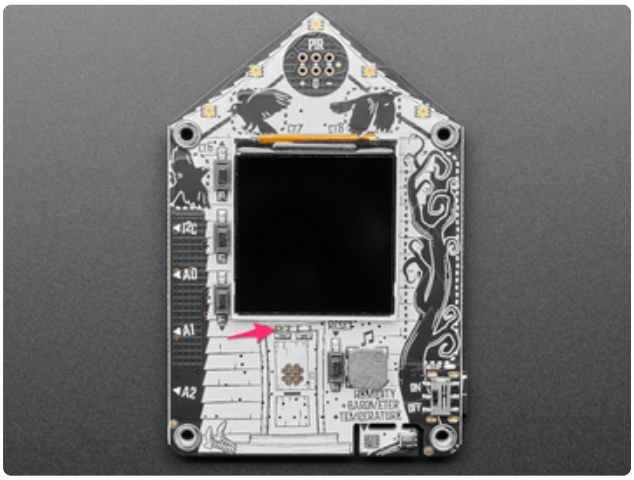
Light Sensor

A light sensor (also known as a CdS cell, light-dependent resistor, or photoresistor) detects light. They change their resistive value (in ohms, Ω) depending on how much light shines into the photocell.

When a light sensor is exposed to more light, the resistance decreases. When it is exposed to less light, the resistance increases.

By using a light sensor wired in a specific way (as a voltage divider), we can turn resistance into voltage. That change is then read by your board's Analog-to-Digital converter and sent to Adafruit IO.

Where is the Light Sensor on my board?

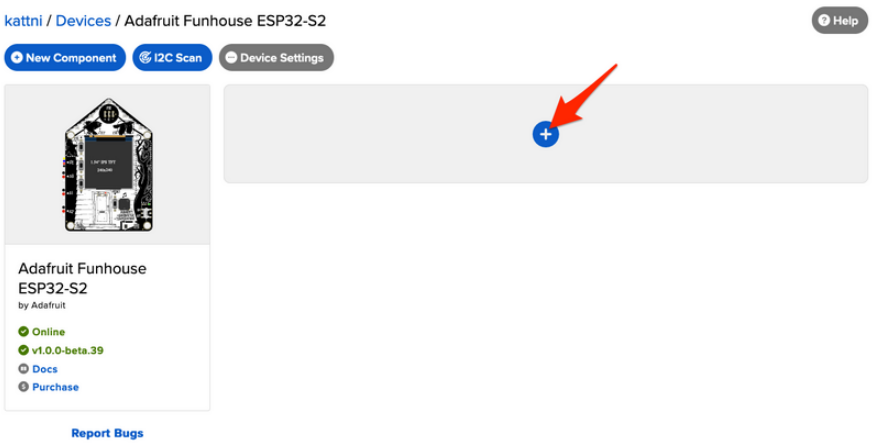


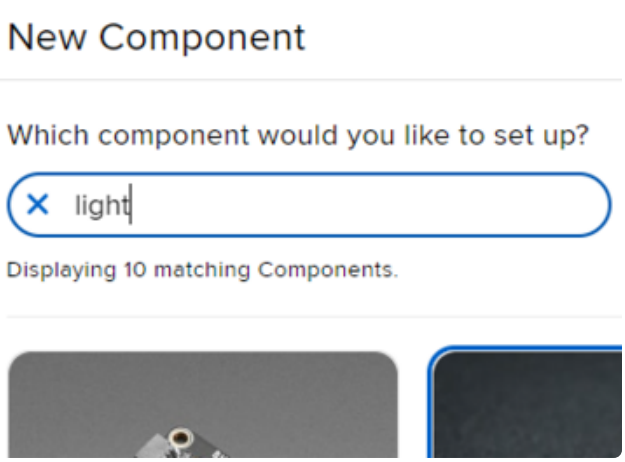
Below the display, and slightly to the left, is a front-facing **light sensor** that is positioned at the top left of the FunHouse door.

Note: The light sensor on the FunHouse is influenced by the display's backlight, use some tape to block the light if needed. More info, and a great chart here: https://forums.adafruit.com/viewtopic.php?f=19&t=179236

Create a Light Sensor Component

On the device page, click the New Component (or "+") button to open the component picker.





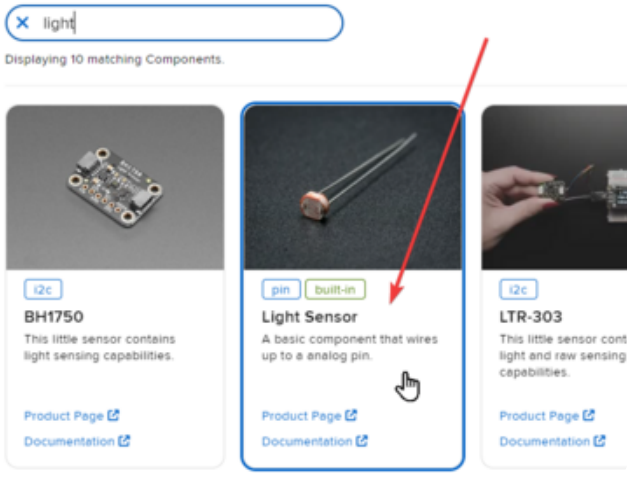
Search for the component name by entering `light` into the text box on the component picker, the list of components should update as soon as you stop typing.

Filtering and searching for components

WipperSnapper supports such a large number of components we added filtering! Try searching for various keywords, like:

- component names: `aht20`, `servo`, `buzzer`, `button`, `relay`, etc
- sensor types: `light`, `temperature`, `pressure`, `humidity`, etc
- interface: `i2c`, `uart`, `ds18x20`, `pin`, etc (also I2C addresses e.g. `0x44`)
- vendor: `Adafruit`, `ASAIR`, `Infineon`, `Bosch`, `Honeywell`, `Sensirion`, etc

We've also added product and documentation links to every component, follow the links beneath the component descriptions to be taken to the appropriate product page or Learn-Guide.

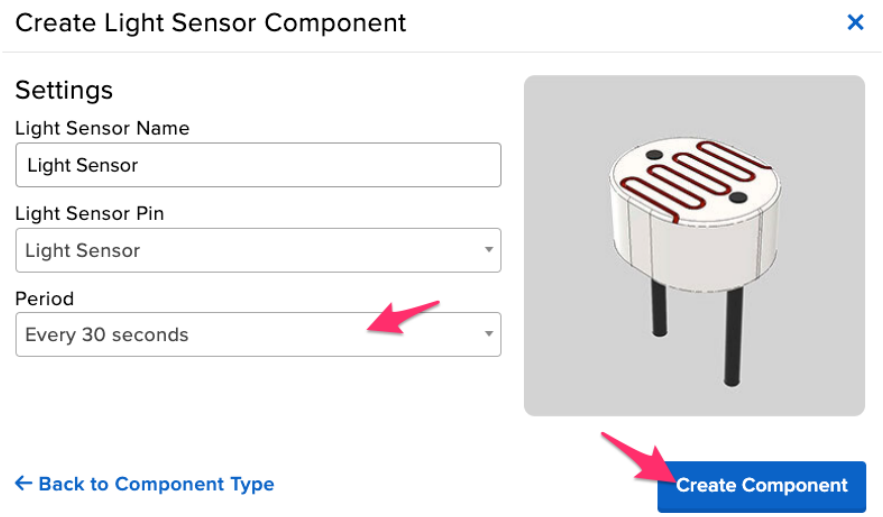


Select the **Light Sensor** from the list of results to go to the component configuration page.

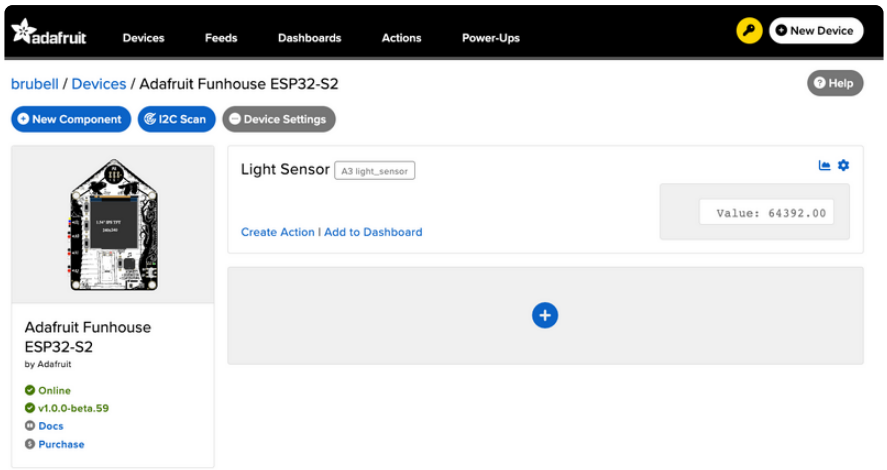
There will be a back button if you select the wrong component, and you can use the Edit component icon (⚙) on the device page to update the component configuration in the future.

The name and pin for the light sensor on your board are automatically selected. The **Period** determines how frequently the light sensor's value will be checked and sent to Adafruit IO. Set it to check the light sensor value every 30 seconds.

Click Create Component.

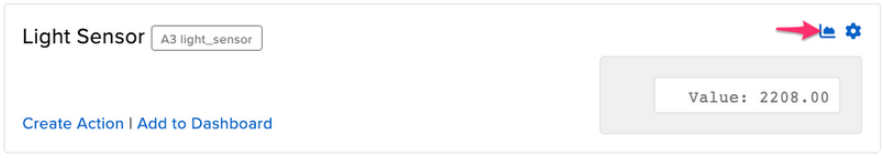


The device page shows a new light sensor component. The value of this component will change every 30 seconds.

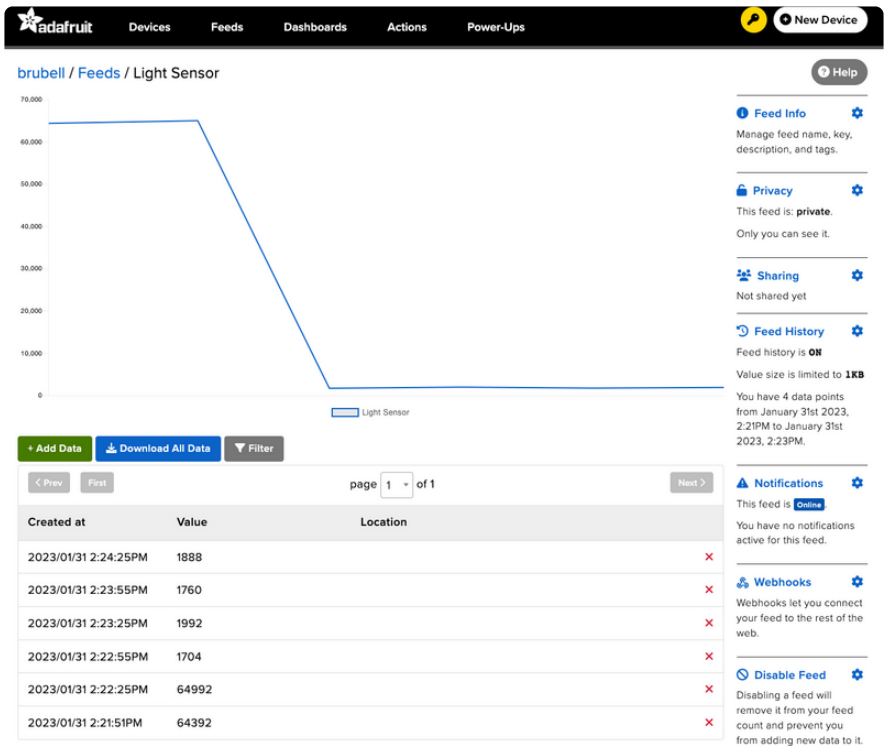


Light Sensor Usage

To test the light sensor, try covering the light sensor with a piece of paper. Navigate to the feed page by clicking the graph icon on the top right corner of the light sensor component.



On the light sensor feed page, you'll be able to observe a graph of the light sensor values as they change over time.



I2C: On-board Sensors

Inter-Integrated Circuit, aka **I2C**, is a two-wire protocol for connecting sensors and "devices" to a microcontroller. A large number of sensors, including the ones sold by Adafruit, use I2C to communicate.

Typically, using I2C with a microcontroller involves programming. Adafruit IO and WipperSnapper let you configure a microcontroller to read data from an I2C sensor and publish that data to the internet without writing code.

The WipperSnapper firmware supports a number of I2C sensors, viewable in list format here (https://adafru.it/Zbq).

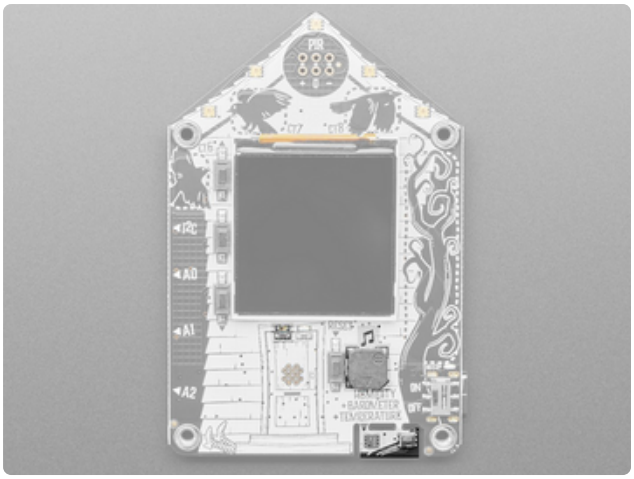
- If you do not see the I2C sensor you're attempting to use with WipperSnapper, we have a guide on adding a component to Adafruit IO WipperSnapper here (https://adafru.it/Zbr).

The process for adding an I2C component to your board running WipperSnapper is similar for most sensors.

On this page, you'll learn how to configure an I2C sensor built into a development board to send data to Adafruit IO. Then you'll learn how to locate, interpret, and download the data produced by your sensors.

Where are the I2C sensors on my board?

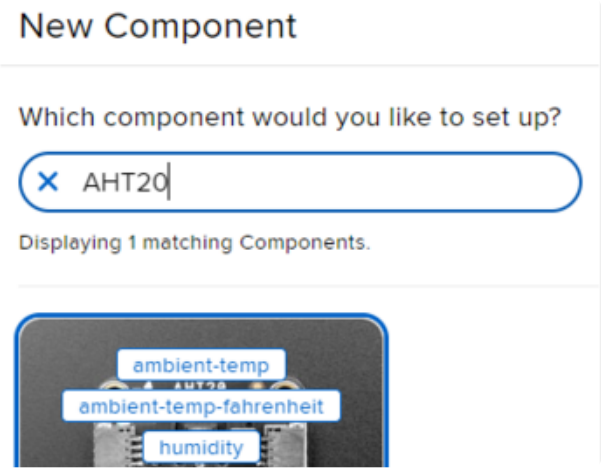
Your board has multiple I2C sensors built-in meaning that there's no wiring required!



In the bottom right corner of the board, on the left side of the cutout region (highlighted in **yellow**), is a **DPS310 pressure sensor**, that can be used to sense the barometric pressure. It is connected to the I2C port and available on I2C address `0x77`.
Also in the bottom right corner of the board, on the right side of the cutout region (highlighted in **red**), is an **AHT20 Humidity and Temperature sensor**, that can be used to sense the humidity and temperature. It is connected to the I2C port and available on I2C address `0x38`.

Create AHT20 Sensor Component

On the device page, click the New Component (or "+") button to open the component picker.



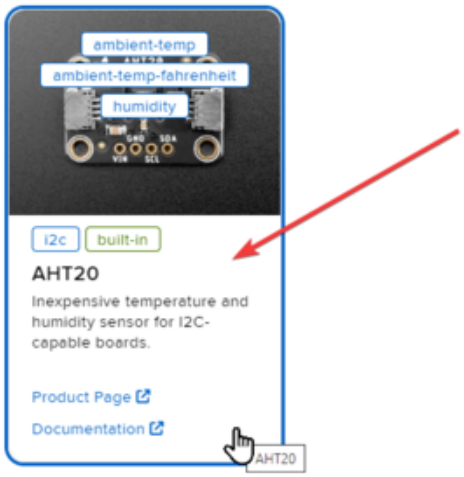
Search for the component name by entering `AHT20` into the text box on the component picker, the list of components should update as soon as you stop typing.

Filtering and searching for components

WipperSnapper supports such a large number of components we added filtering! Try searching for various keywords, like:

- component names: `aht20`, `servo`, `buzzer`, `button`, `potentiometer`, etc
- sensor types: `light`, `temperature`, `pressure`, `humidity`, etc
- interface: `i2c`, `uart`, `ds18x20`, `pin`, etc (also I2C addresses e.g. `0x44`)
- vendor: `Adafruit`, `ASAIR`, `Infineon`, `Bosch`, `Honeywell`, `Sensirion`, etc

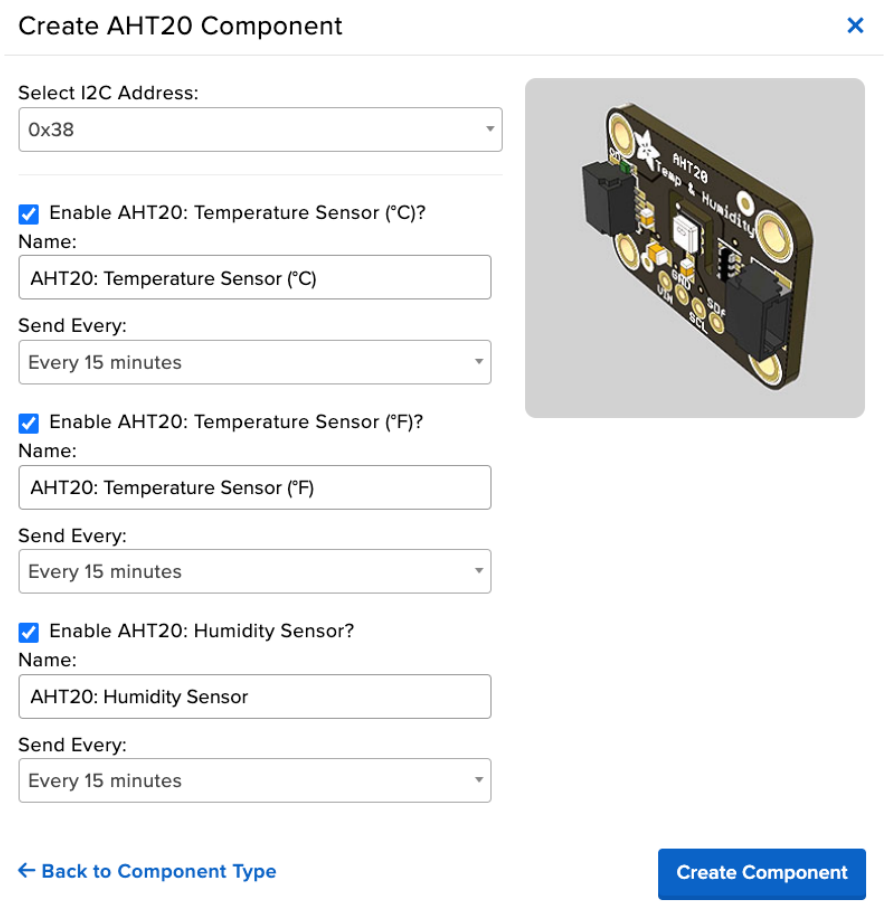
We've also added product and documentation links to every component, follow the links beneath the component descriptions to be taken to the appropriate product page or Learn-Guide.



Select the **AHT20** from the list of results to go to the component configuration page.

There will be a back button if you select the wrong component, and you can use the Edit component icon (⚙) on the device page to update the component configuration in the future.

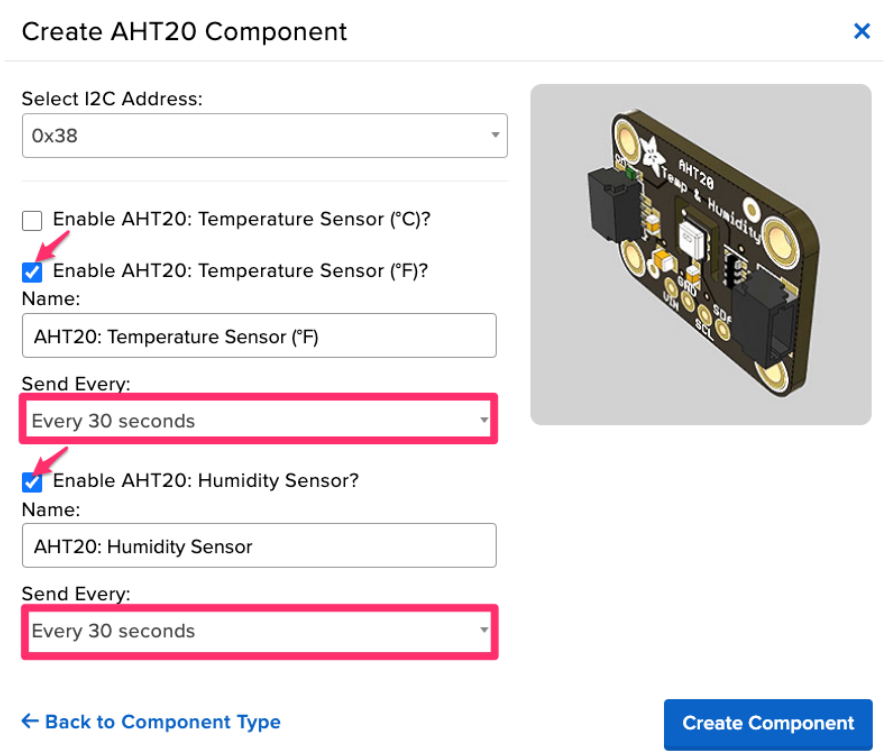
On the component configuration page, the AHT20's I2C sensor address should be listed along with the sensor's settings.



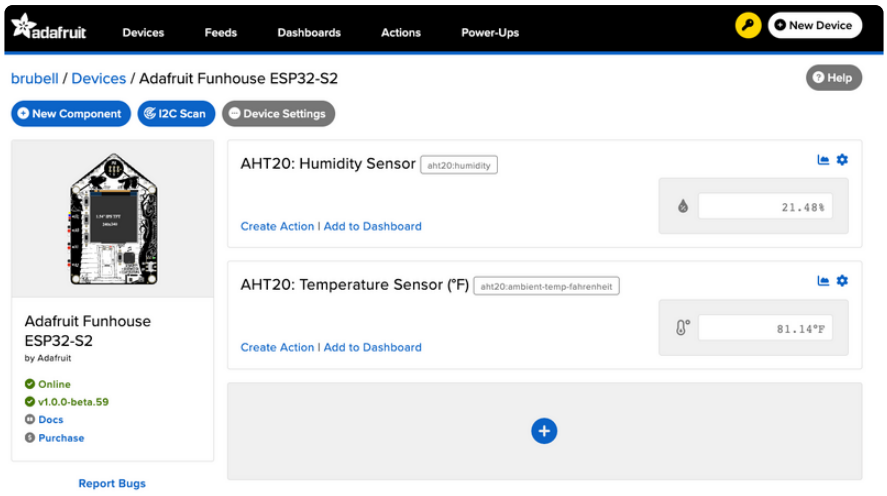
The AHT20 sensor can measure ambient temperature and relative humidity. This page has individual options for reading the ambient temperature, in either Celsius or Fahrenheit, and the relative humidity. You may select the readings which are appropriate to your application and region.

The **Send Every** option is specific to each sensor measurement. This option will tell the Feather how often it should read from the AHT20 sensor and send the data to Adafruit IO. Measurements can range from every 30 seconds to every 24 hours.

For this example, set the **Send Every** interval for both seconds to **Every 30 seconds.** Click **Create Component**.



The board page should now show the AHT20 component you created. After the interval you configured elapses, the WipperSnapper firmware running on your board automatically reads values from the sensor and sends them to Adafruit IO.

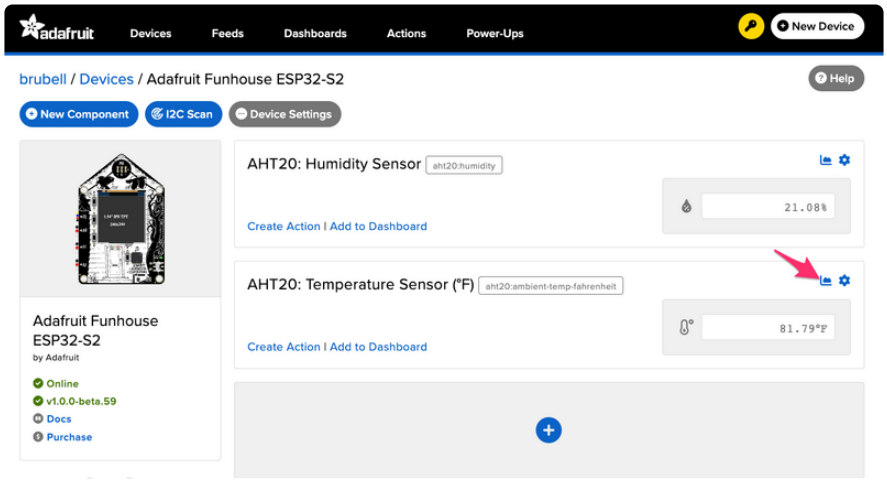


Read I2C Sensor Values

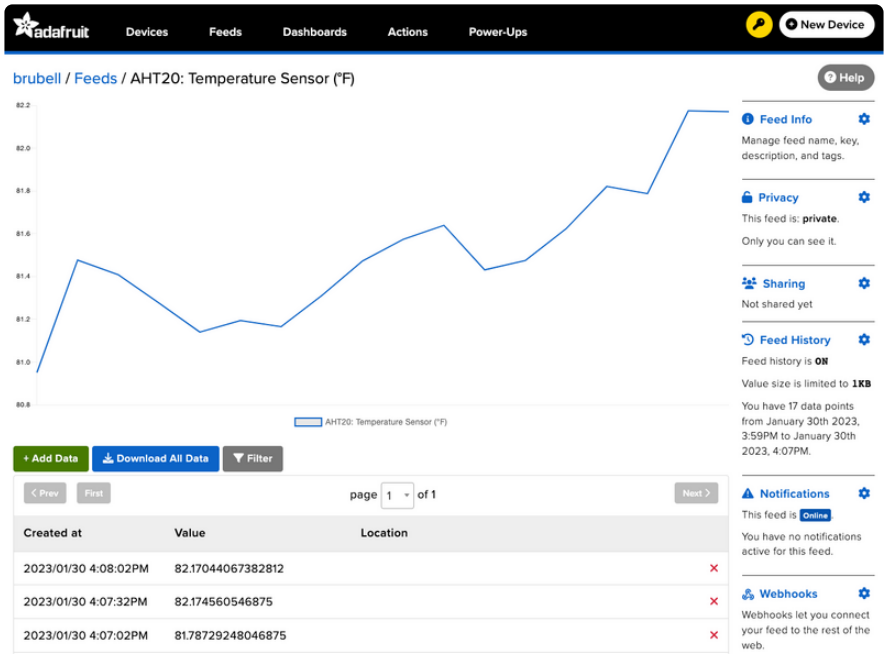
Now to take a look behind the scenes at a powerful element of using Adafruit IO and WipperSnapper. When a new component is created on Adafruit IO, an Adafruit IO Feed (https://adafru.it/ioA) is also created. This Feed holds your sensor component's values for long-term storage (30 days of storage for Adafruit IO Free and 60 days for Adafruit IO Plus plans).

Aside from holding the **values** read by a sensor, the component feed also holds **metadata** about the data pushed to Adafruit IO. This includes settings for whether the data is public or private, what license the stored sensor data falls under, and a general description of the data.

Now look at the AHT20's temperature sensor feed. To navigate to a component's feed, click on the chart icon in the upper-right-hand corner of the component.



On the component's feed page, you'll each data point read by your sensor and when they were reported to Adafruit IO.



Doing more with your sensor's Adafruit IO Feed

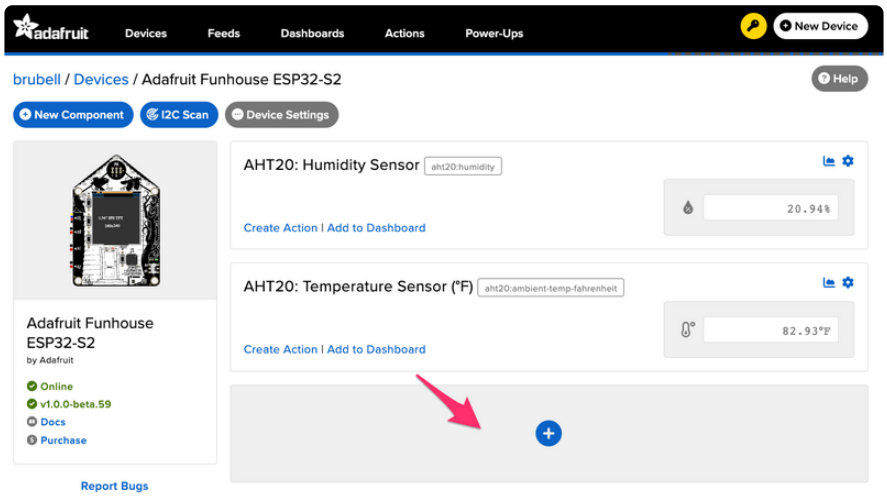
We've only scratched the surface of what Adafruit IO Feeds can accomplish for your IoT projects. For a complete overview of Adafruit IO Feeds, including tasks like

downloading feed data, sharing a feed, removing erroneous data points from a feed, and more, head over to the "Adafruit IO Basics: Feed" learning guide (https://adafru.it/ioA).

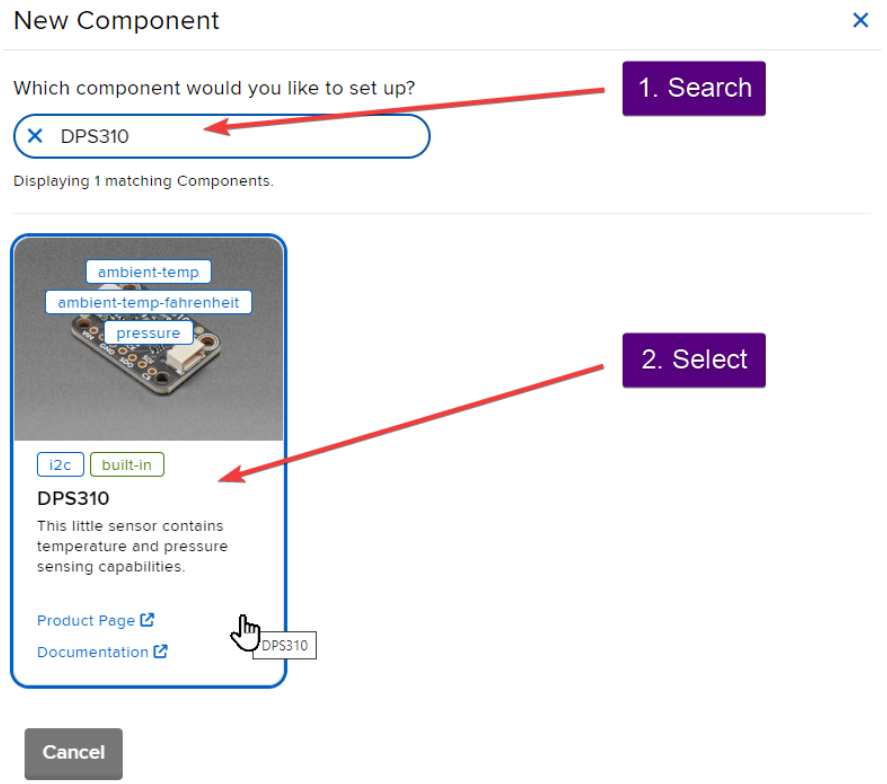
Create DPS310 Component

The process of creating an Adafruit IO component for the FunHouse's DPS310 sensor is similar to the process we followed above for the AHT20.

On the device page, click the New Component (or "+") button to open the component picker.



Adafruit IO supports a large amount of components. To quickly find your sensor, type `DPS310` into the search bar, then select the **DPS310** component.

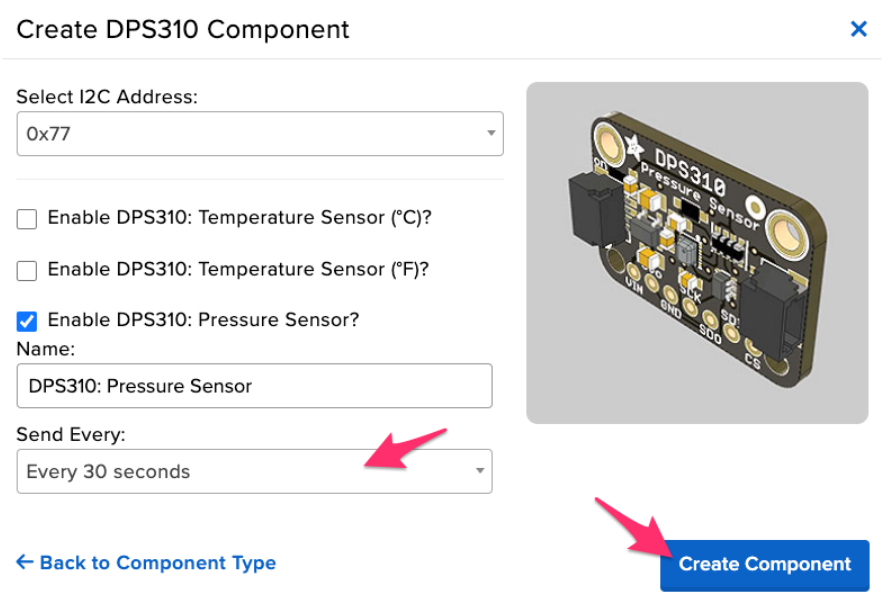


The DPS310 sensor can measure barometric pressure and/or temperature. Select the sensor readings which are appropriate to your application and region.

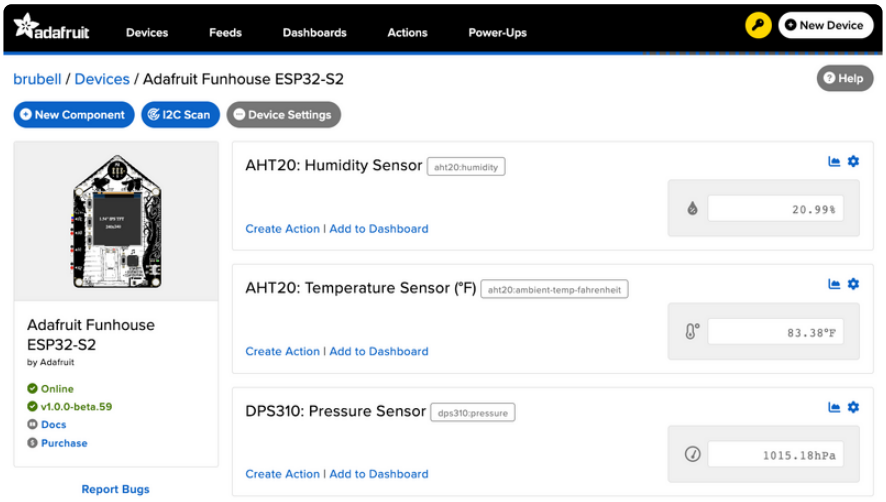
Since we previously set up the DPS310 to measure ambient temperature, we're only selecting the barometric pressure option.

The **Send Every** option is specific to each sensor measurement. This option will tell the Feather how often it should read from the DPS310 sensor and send the data to Adafruit IO. Measurements can range from every 30 seconds to every 24 hours.

For this example, set the **Send Every** interval for both seconds to **Every 30 seconds.** Click **Create Component**.



The board page should now show the DPS310 component you created. After the interval you configured elapses, the WipperSnapper firmware running on your board automatically reads values from the sensor and sends them to Adafruit IO.



I2C: External Sensor

While this page uses the "MCP9808 High Accuracy I2C Temperature Sensor Breakout", the process for adding an I2C sensor to your board running WipperSnapper is similar for all I2C sensors.

Inter-Integrated Circuit, aka **I2C**, is a two-wire protocol for connecting sensors and "devices" to a microcontroller. A large number of sensors, including the ones sold by Adafruit, use I2C to communicate.

Typically, using I2C with a microcontroller involves programming. Adafruit IO and WipperSnapper let you configure a microcontroller to read data from an I2C sensor and publish that data to the internet without writing code.

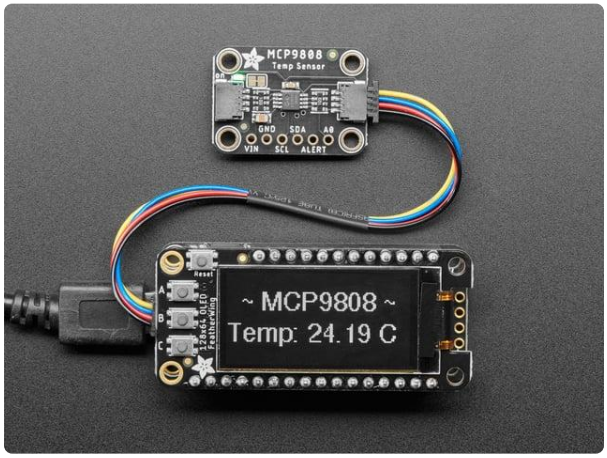
The WipperSnapper firmware supports a number of I2C sensors, viewable in list format here (https://adafru.it/Zbq).

- If you do not see the I2C sensor you're attempting to use with WipperSnapper, Adafruit has a guide on adding a component to Adafruit IO WipperSnapper here (https://adafru.it/Zbr).

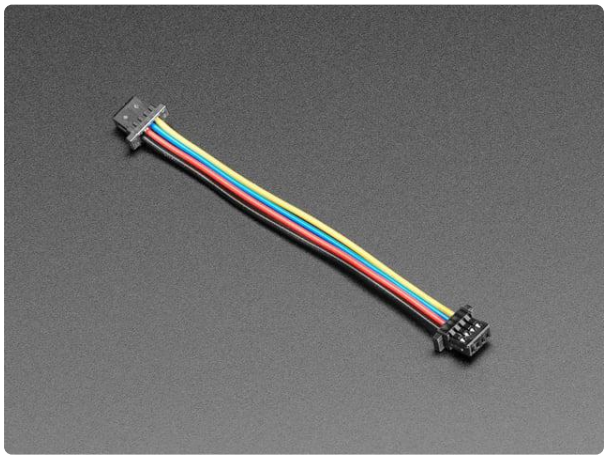
On this page, you'll learn how to wire up an I2C sensor to your board. Then, you'll create a new component on Adafruit IO for your I2C sensor and send the sensor values to Adafruit IO. Finally, you'll learn how to locate, interpret, and download the data produced by your sensors.

Parts

You will need the following parts to complete this page:

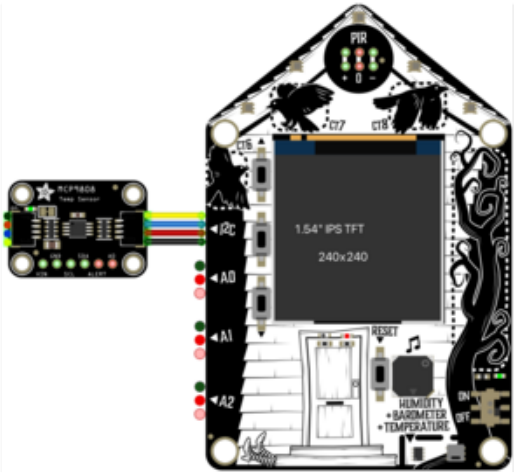


Adafruit MCP9808 High Accuracy I2C Temperature Sensor Breakout
The MCP9808 digital temperature sensor is one of the more accurate/precise we've ever seen, with a typical accuracy of ±0.25°C over the sensor's -40°C to...
https://www.adafruit.com/product/5027



STEMMA QT / Qwiic JST SH 4-Pin Cable - 50mm Long
This 4-wire cable is 50mm / 1.9" long and fitted with JST SH female 4-pin connectors on both ends. Compared with the chunkier JST PH these are 1mm pitch instead of 2mm, but...
https://www.adafruit.com/product/4399

Wiring



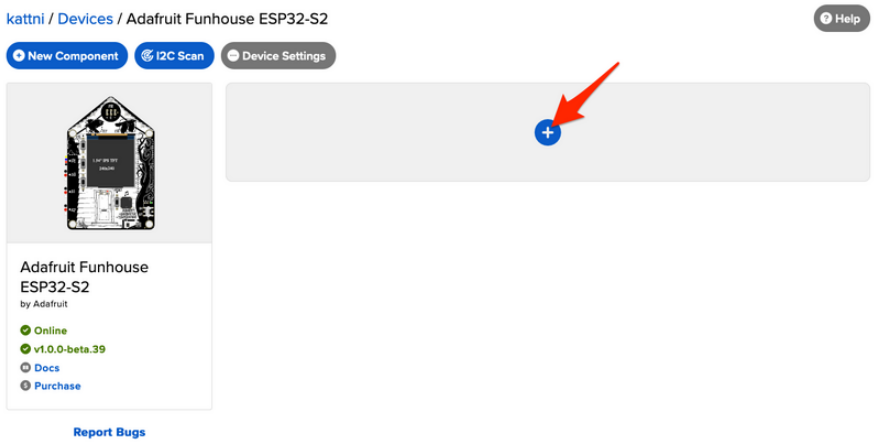
If you're using a STEMMA QT to STEMMA QT cable:

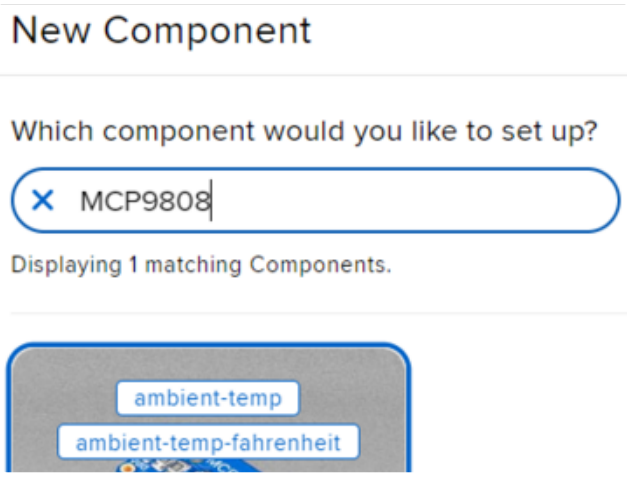
Board STEMMA QT Port to **MCP9808's STEMMA QT Port**
If you're using a breadboard:

Board power to **MCP9808 VIN**
Board ground to **MCP9808 GND**
Board SCL to **MCP9808 SCL**
Board SDA to **MCP9808 SDA**

Add an MCP9808 Component

On the device page, click the New Component (or "+") button to open the component picker.





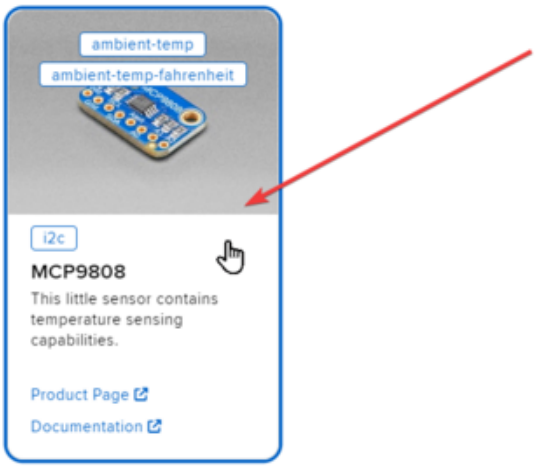
Search for the component name by entering `MCP9808` into the text box on the component picker, the list of components should update as soon as you stop typing.

Filtering and searching for components

Since WipperSnapper supports such a large number of components, there is keyword filtering. Try searching for various keywords, like:

- component names: `aht20`, `servo`, `buzzer`, `button`, `potentiometer`, etc
- sensor types: `light`, `temperature`, `pressure`, `humidity`, etc
- interface: `i2c`, `uart`, `ds18x20`, `pin`, etc (also I2C addresses e.g. `0x44`)
- vendor: `Adafruit`, `ASAIR`, `Infineon`, `Bosch`, `Honeywell`, `Sensirion`, etc

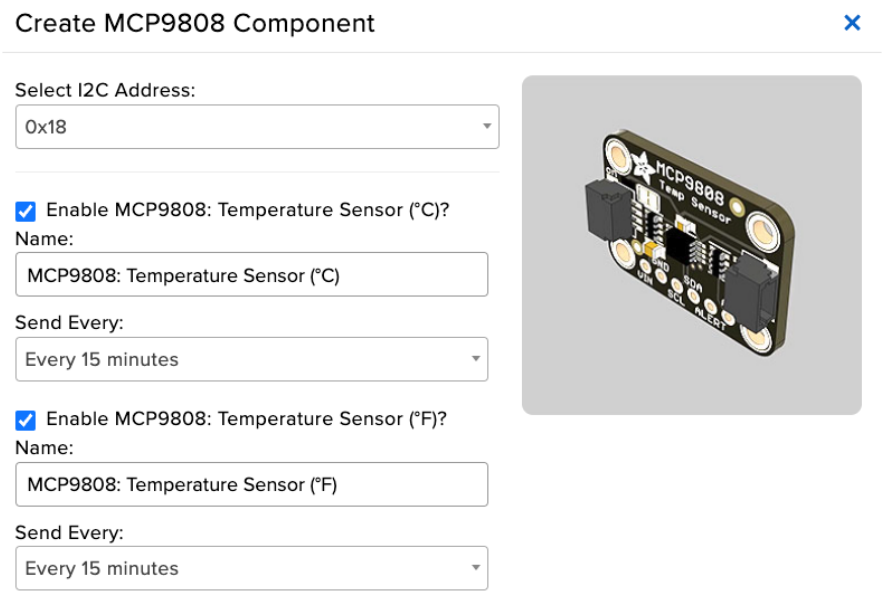
There are added product and documentation links for every component, follow the links beneath the component descriptions to be taken to the appropriate product page or Learn Guide.



Select the **MCP9808** from the list of results to go to the component configuration page.

There will be a back button if you select the wrong component, and you can use the Edit component icon (⚙) on the device page to update the component configuration in the future.

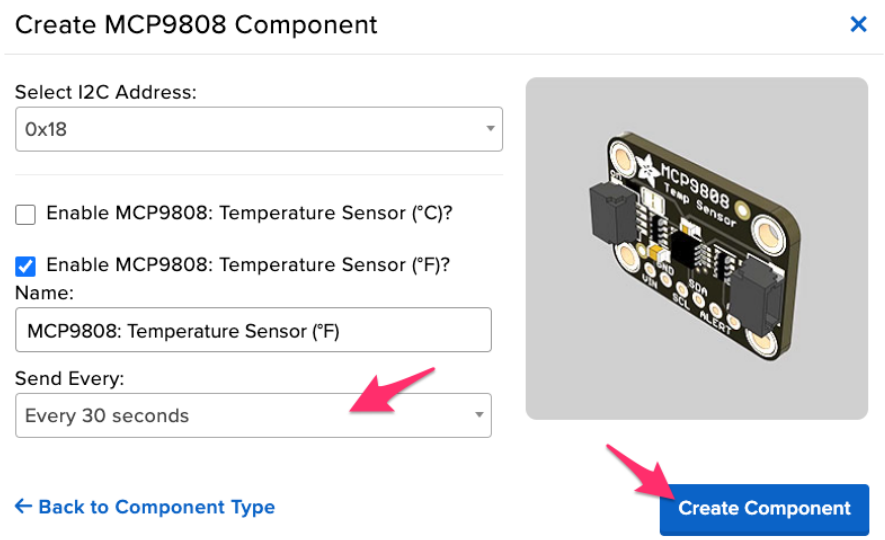
On the component configuration page, the MCP9808's I2C sensor address should be listed along with the sensor's settings.



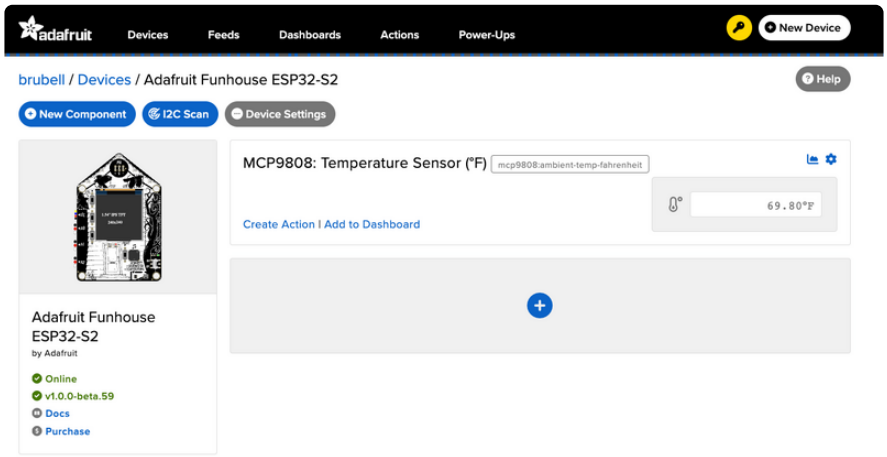
The MCP9808 sensor can measure ambient temperature. This page has individual options for reading the ambient temperature, in either Celsius or Fahrenheit. You may select the readings which are appropriate to your application and region.

The **Send Every** option is specific to each sensor measurement. This option will tell the board how often it should read from the sensor and send the data to Adafruit IO. Measurements can range from every 30 seconds to every 24 hours.

For this example, set the **Send Every** interval for both seconds to **Every 30 seconds.** Click **Create Component**.



The board page should now show the MCP9808 component you created. After the interval you configured elapses, the WipperSnapper firmware running on your board automatically reads values from the sensor and sends them to Adafruit IO.



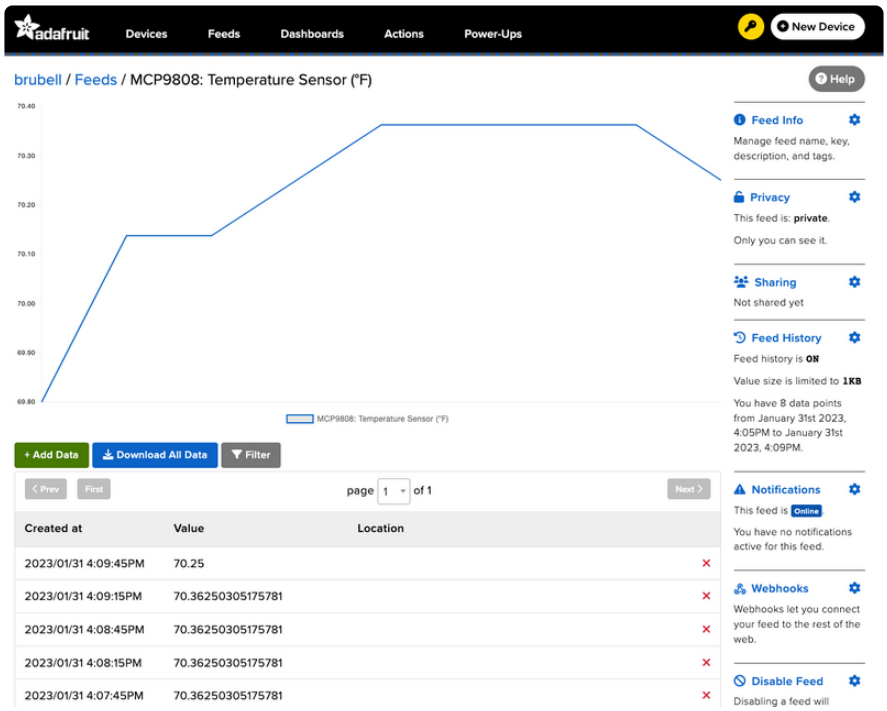
Read I2C Sensor Values

Now to look behind the scenes at a powerful element of using Adafruit IO and WipperSnapper. When a new component is created on Adafruit IO, an Adafruit IO Feed (https://adafru.it/ioA) is also created. This Feed holds your sensor component values for long-term storage (30 days of storage for Adafruit IO Free and 60 days for Adafruit IO Plus plans).

Aside from holding the **values** read by a sensor, the component's feed also holds **metadata** about the data pushed to Adafruit IO. This includes settings for whether the data is public or private, what license the stored sensor data falls under, and a general description of the data.

Next, to look at the sensor temperature feed. To navigate to a component's feed, click on the chart icon in the upper-right-hand corner of the component.

On the component's feed page, you'll each data point read by your sensor and when they were reported to Adafruit IO.



Doing more with your sensor's Adafruit IO Feed

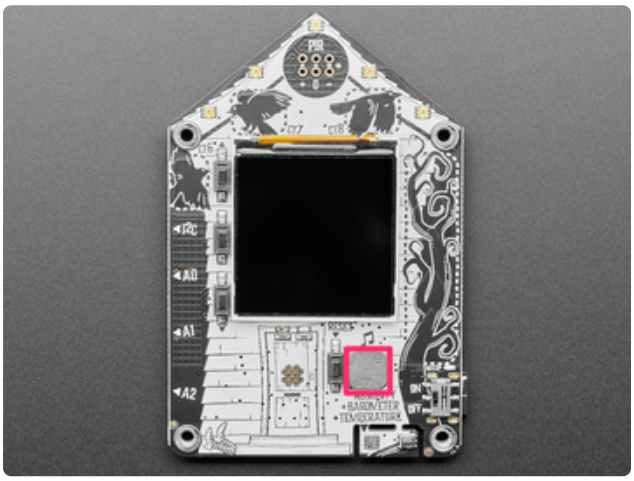
This only scratches the surface of what Adafruit IO Feeds can accomplish for your IoT projects. For a complete overview of Adafruit IO Feeds, including tasks like downloading feed data, sharing a feed, removing erroneous data points from a feed, and more, head over to the "Adafruit IO Basics: Feed" learning guide (https://adafru.it/ioA).

Piezo Speaker

Piezo buzzers are simple devices that can generate basic beeps and tones. They work by using a piezo crystal, a special material that changes shape when voltage is applied to it. If the crystal pushes against a diaphragm, like a tiny speaker cone, it can generate a pressure wave that the human ear picks up as sound. Simply change the frequency of the voltage sent to the piezo and it will start generating sounds by changing shape very quickly!

This page will explain configuring a piezo buzzer using Adafruit IO and WipperSnapper, making the piezo "buzz", and changing its sound.

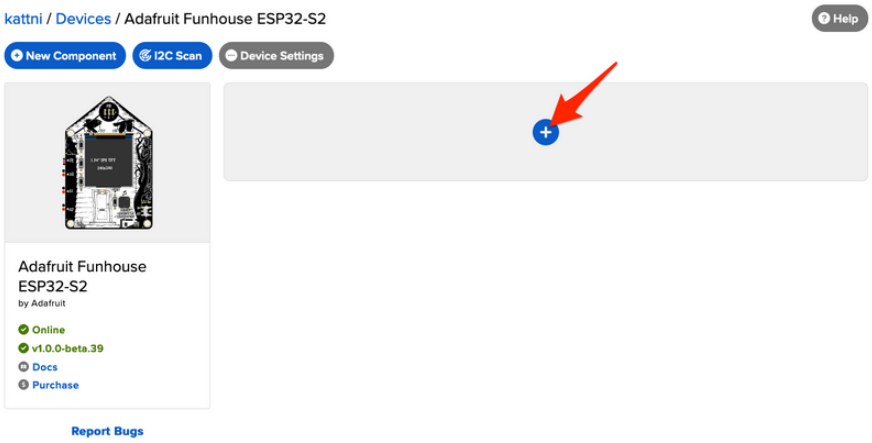
Piezo Buzzer Location

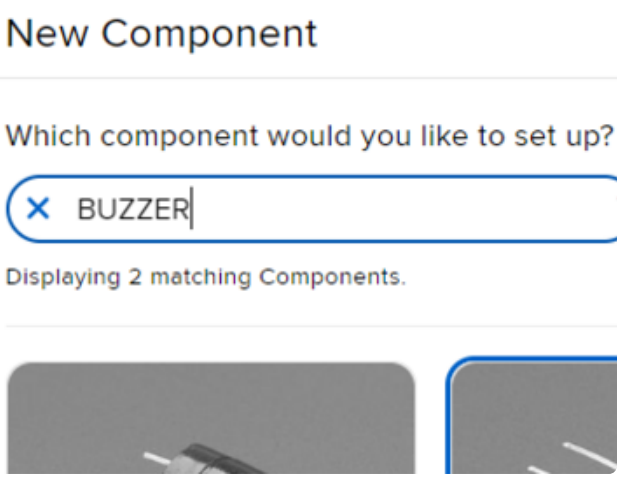


The Piezo buzzer is located on the right side of the FunHouse, highlighted in pink.

Create a Piezo Buzzer Component

On the device page, click the New Component (or "+") button to open the component picker.





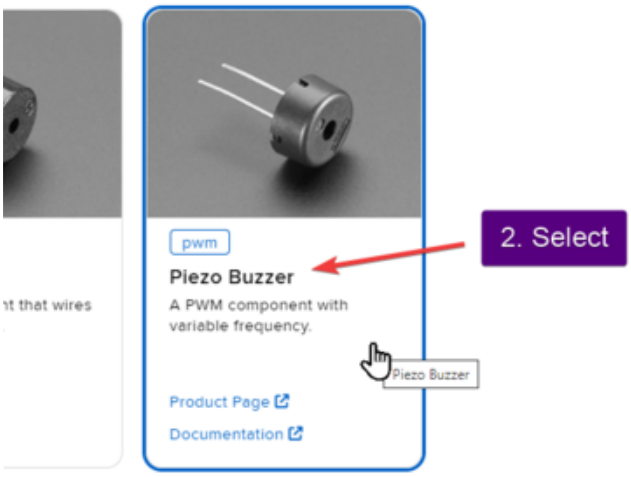
Search for the component name by entering `BUZZER` into the text box on the component picker, the list of components should update as soon as you stop typing.

Filtering and searching for components

WipperSnapper supports such a large number of components we added filtering! Try searching for various keywords, like:

- component names: `aht20`, `servo`, `buzzer`, `button`, `piezo`, etc
- sensor types: `light`, `temperature`, `pressure`, `humidity`, etc
- interface: `i2c`, `uart`, `ds18x20`, `pin`, etc (also I2C addresses e.g. `0x44`)
- vendor: `Adafruit`, `ASAIR`, `Infineon`, `Bosch`, `Honeywell`, `Sensirion`, etc

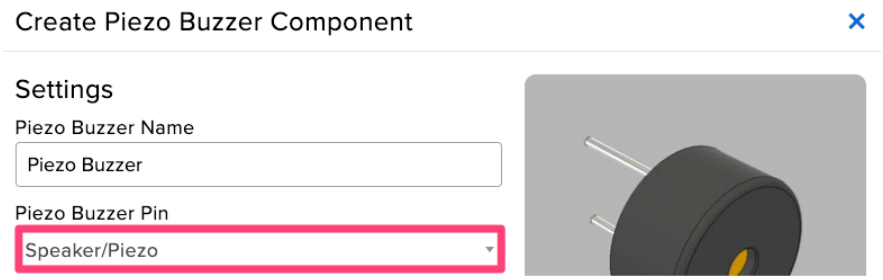
We've also added product and documentation links to every component, follow the links beneath the component descriptions to be taken to the appropriate product page or Learn-Guide.



Select the **Piezo Buzzer** from the list of results to go to the component configuration page.

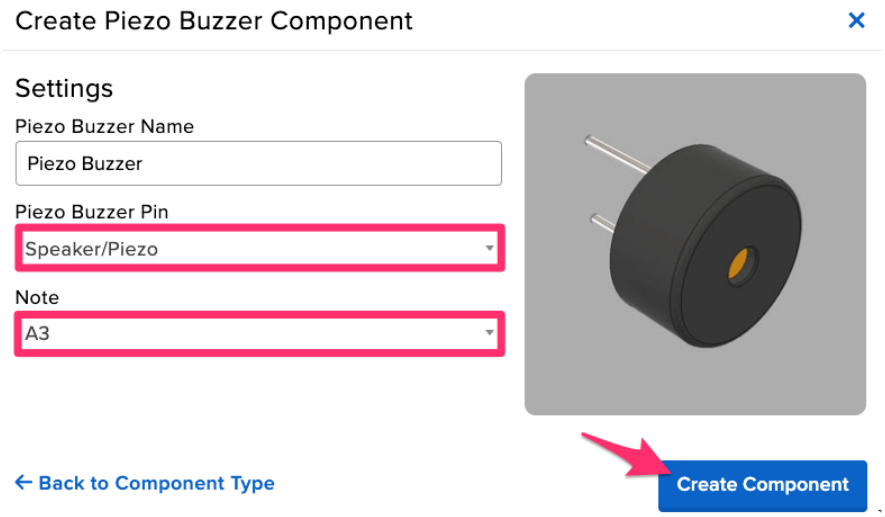
There will be a back button if you select the wrong component, and you can use the Edit component icon (⚙) on the device page to update the component configuration in the future.

On the Create Piezo Buzzer form, under the Piezo Buzzer pin, select "Speaker/Piezo".

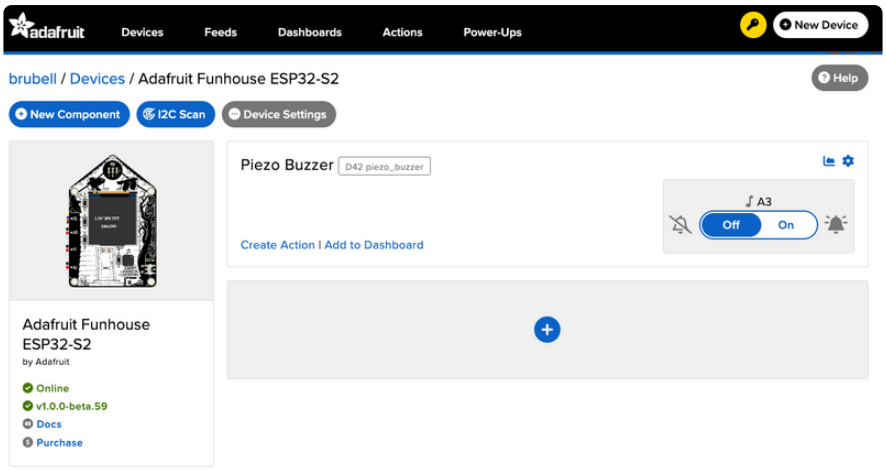


WipperSnapper matches musical notes to frequencies, in Hertz (there's a handy hertz-to-note table on this website (https://adafru.it/19DW)).

Select a musical note you'd like the Piezo to play when it's activated. Then, click Create Component.

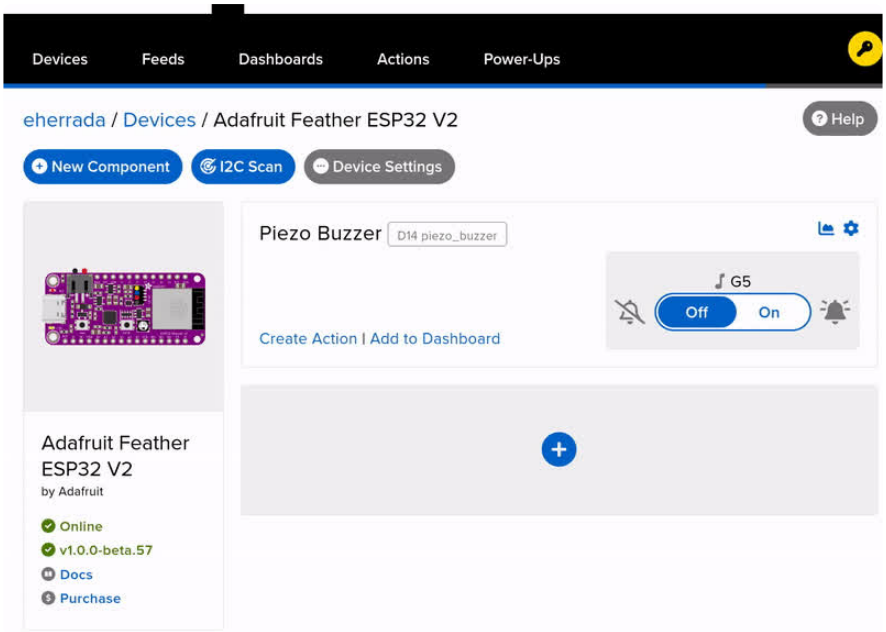


The device page displays a new piezo buzzer component. Toggling the piezo buzzer component's switch "on" causes the Piezo buzzer to play the selected note. Toggling the switch "off" stops the Piezo buzzer.



Modify the Note

Is your buzzer's tone too high-pitched or too quiet? To modify the note played by the buzzer, click the cog on the top-right of the Wippersnapper piezo component. Select a new tone from the dropdown labeled Note.



Factory Reset

The FunHouse microcontroller ships running a full sensor test and DotStar swirl. The sensor data is shown on the display. It's lovely, but you probably had other plans for the board. As you start working with your board, you may want to return to the original code to begin again, or you may find your board gets into a bad state. Either way, this page has you covered.

You're probably used to seeing the **HOUSEBOOT** drive when loading CircuitPython or Arduino. The **HOUSEBOOT** drive is part of the UF2 bootloader, and allows you to drag and drop files, such as CircuitPython. However, on the ESP32-S2 the UF2 bootloader can become damaged.

Factory Reset Firmware UF2

If you have a bootloader still installed - which means you can double-click to get the **HOUSEBOOT** drive to appear, then you can simply drag this UF2 file over to the **BOOT** drive.

To enter bootloader mode, plug in the board into a USB cable with data/sync capability. Press the reset button once, wait till the RGB LED turns purple, then press the reset button again. Then drag this file over:

Click to download the FunHouse Factory Reset UF2

https://adafru.it/XRa

Your board is now back to its factory-shipped state! You can now begin again with your plans for your board.

Factory Reset and Bootloader Repair

What if you tried double-tapping the reset button, and you still can't get into the UF2 bootloader? Whether your board shipped without the UF2 bootloader, or something damaged it, this section has you covered.

There is no bootloader protection for the UF2 bootloader. That means it is possible to erase or damage the UF2 bootloader, especially if you upload an Arduino sketch to an ESP32-S2/S3 board that doesn't "know" there's a bootloader it should not overwrite!

It turns out, however, the ESP32-S2/S3 comes with a second bootloader: the ROM bootloader. Thanks to the ROM bootloader, you don't have to worry about damaging the UF2 bootloader. The ROM bootloader can never be disabled or erased, so its always there if you need it! You can simply re-load the UF2 bootloader from the ROM bootloader.

Completing a factory reset will erase your board's firmware which is also used for storing CircuitPython/Arduino/Files! Be sure to back up your data first.

There are two ways to do a factory reset and bootloader repair. The first is using WebSerial through a Chromium-based browser, and the second is using `esptool` via command line. **We highly recommend using WebSerial through Chrome/Chromium.**

The next section walks you through the prerequisite steps needed for both methods.

Download .bin and Enter Bootloader

Step 1. Download the factory-reset-and-bootloader.bin file

Save the following file wherever is convenient for you. You will need to access it from the WebSerial ESPTool.

Note that this file is approximately 3MB. This is not because the bootloader is 3MB, it is because the bootloader is near the end of the available flash. Most of the file is empty but its easier to program if you use a combined file.

Click to download funhouse-factory-reset-and-bootloader.bin

https://adafru.it/XRb

Step 2. Enter ROM bootloader mode

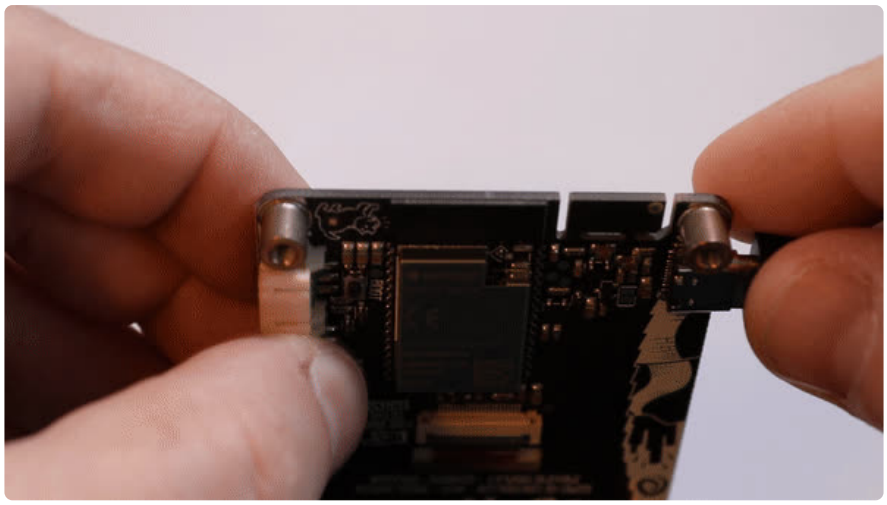
Entering the ROM bootloader is easy. Complete the following steps.

Before you start, make sure your ESP32-S2/S3 is plugged into USB port to your computer using a data/sync cable. Charge-only cables will not work!

Turn on the On/Off switch - check that you see the OK light on so you know the board is powered, a prerequisite!

To enter the bootloader:

1. **Press and hold the BOOT/DFU button down.** Don't let go of it yet!
2. **Press and release the Reset button.** You should still have the BOOT/DFU button pressed while you do this.
3. **Now you can release the BOOT/DFU button.**



No USB drive will appear when you've entered the ROM bootloader. This is normal!

Now that you've downloaded the .bin file and entered the bootloader, you're ready to continue with the factory reset and bootloader repair process. The next two sections walk you through using WebSerial and `esptool`.

The WebSerial ESPTool Method

We highly recommend using WebSerial ESPTool method to perform a factory reset and bootloader repair. However, if you'd rather use esptool via command line, you can skip this section.

This method uses the WebSerial ESPTool through Chrome or a Chromium-based browser. The WebSerial ESPTool was designed to be a web-capable option for programming ESP32-S2/S3 boards. It allows you to erase the contents of the microcontroller and program up to four files at different offsets.

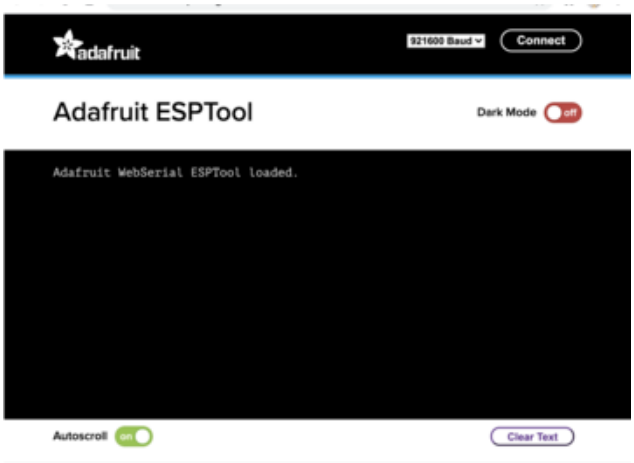
You will have to use a Chromium browser (like Chrome, Opera, Edge...) for this to work, Safari and Firefox, etc. are not supported because we need Web Serial and only Chromium is supporting it to the level needed.

Follow the steps to complete the factory reset.

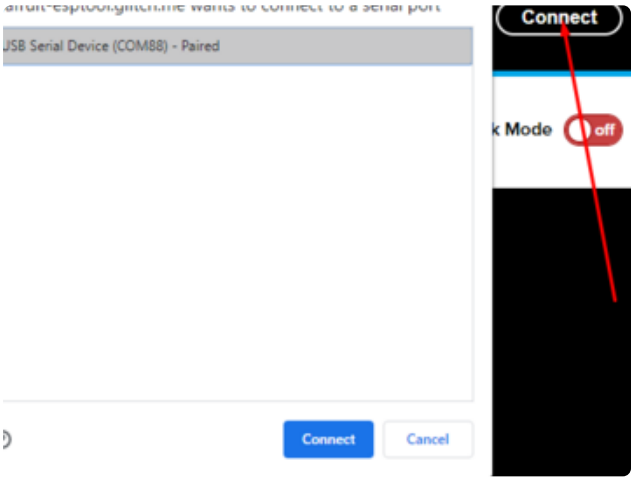
If you're using Chrome 88 or older, see the Older Versions of Chrome section at the end of this page for instructions on enabling Web Serial.

Connect

You should have plugged in **only the ESP32-S2/S3 that you intend to flash**. That way there's no confusion in picking the proper port when it's time!



In the **Chrome browser** visit https://adafruit.github.io/Adafruit_WebSerial_ESPTool/ (https://adafru.it/PMB). You should see something like the image shown.



Press the **Connect** button in the top right of the web browser. You will get a pop up asking you to select the COM or Serial port.

Remember, you should remove all other USB devices so only the ESP32-S2/S3 board is attached, that way there's no confusion over multiple ports!

On some systems, such as MacOS, there may be additional system ports that appear in the list.

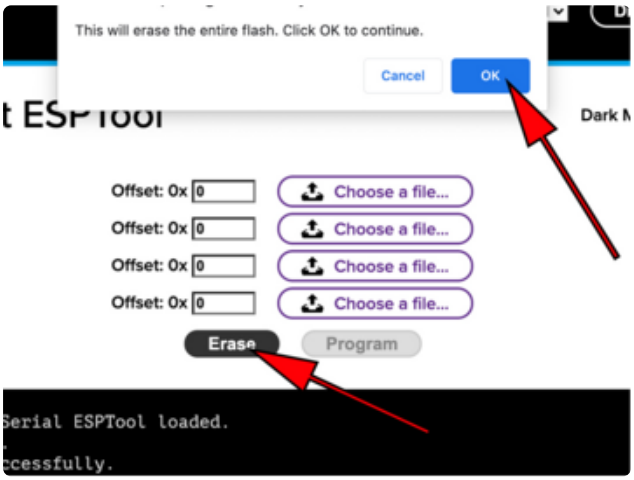
```
ESP Web Flasher loaded.
Connecting...
Connected successfully.
Try hard reset.
Chip type ESP32-S2
Connected to ESP32-S2
MAC Address: 7C:DF:A1:06:8D:D0
Uploading stub...
Running stub...
Stub is now running...
Detecting Flash Size
FlashId: 0x164020
Flash Manufacturer: 20
Flash Device: 4016
Auto-detected Flash size: 4MB
```

The JavaScript code will now try to connect to the ROM bootloader. It may timeout for a bit until it succeeds. On success, you will see that it is **Connected** and will print out a unique **MAC address** identifying the board along with other information that was detected.

Once you have successfully connected, the command toolbar will appear.

Erase the Contents

This will erase everything on your board! If you have access, and wish to keep any code, now is the time to ensure you've backed up everything.



To erase the contents, click the Erase button. You will be prompted whether you want to continue. Click OK to continue or if you changed your mind, just click cancel.

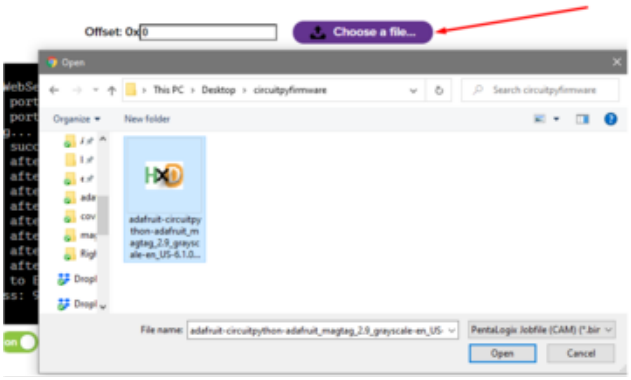
You'll see "Erasing flash memory. Please wait..." This will eventually be followed by "Finished." and the amount of time it took to erase.

Do not disconnect! Immediately continue on to programming the ESP32-S2/S3.

Do not disconnect after erasing! Immediately continue on to the next step!

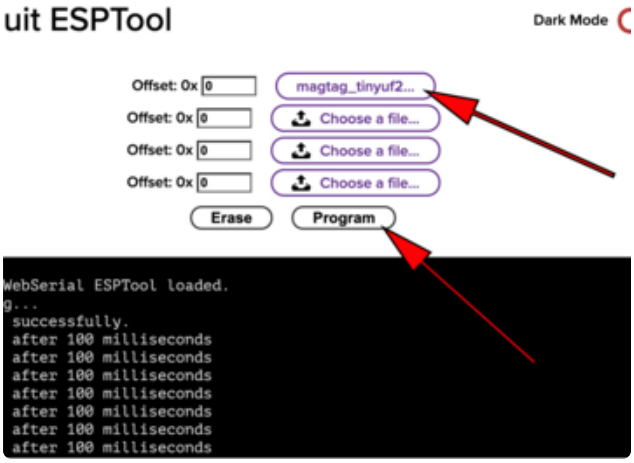
Program the ESP32-S2/S3

Programming the microcontroller can be done with up to four files at different locations, but with the board-specific **factory-reset.bin** file, which you should have downloaded under **Step 1** on this page, you only need to use one file.

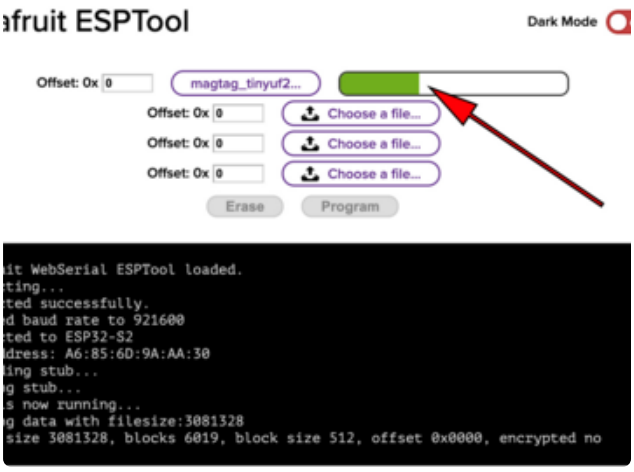


Click on the first **Choose a file...**. (The tool will only attempt to program buttons with a file and a unique location.) Then, select the *-**factory-reset.bin** file you downloaded in Step 1 that matches your board.

Verify that the **Offset** box next to the file location you used is (0x) **0**.



Once you choose a file, the button text will change to match your filename. You can then select the **Program** button to begin flashing.



A progress bar will appear and after a minute or two, you will have written the firmware.

Once completed, you can skip down to the section titled Reset the Board.

The `esptool` Method (for advanced users)

If you used WebSerial ESPTool, you do not need to complete the steps in this section!

Once you have entered ROM bootloader mode, you can then use Espressif's esptool program (https://adafru.it/E9p) to communicate with the chip! `esptool` is the 'official' programming tool and is the most common/complete way to program an ESP chip.

Install ESPTool.py

You will need to use the command line / Terminal to install and run `esptool`.

You will also need to have pip and Python installed (any version!).

Install the latest version using pip (you may be able to run `pip` without the `3` depending on your setup):

```
pip3 install --upgrade esptool
```

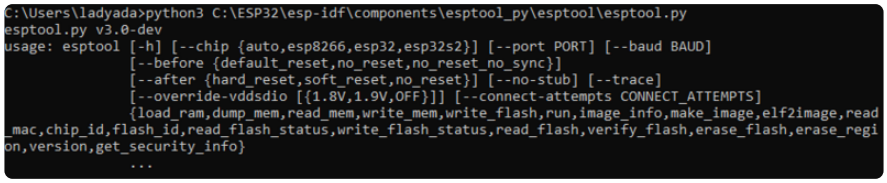
Then, you can run:

```
esptool.py
```

Make sure you are running esptool v3.0 or higher, which adds ESP32-S2/S3 support.

Test the Installation

Run `esptool.py` in a new terminal/command line and verify you get something like the below:

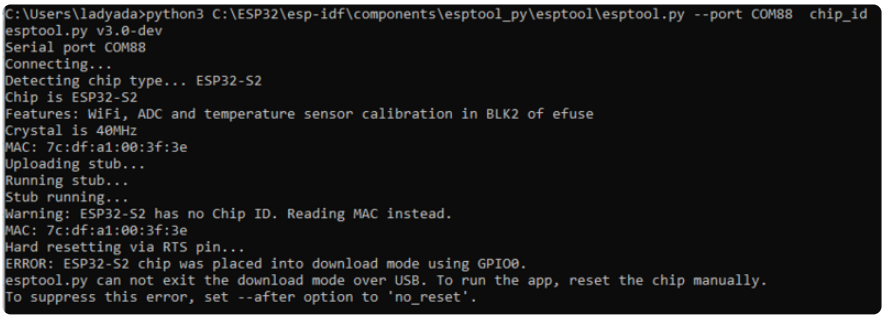


Connect

Run the following command, replacing the identifier after `--port` with the `COMxx`, `/dev/cu.usbmodemxx` or `/dev/ttySxx` you found above.

```
esptool.py --port COM88 chip_id
```

You should get a notice that it connected over that port and found an ESP32-S2/S3.

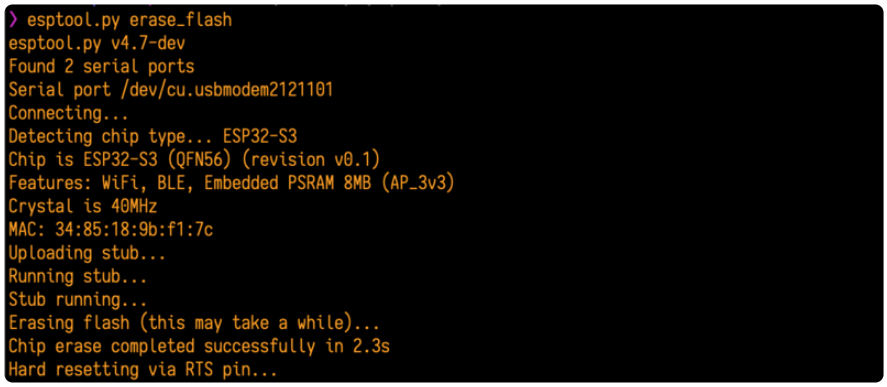


Erase the Flash

Before programming the board, it is a good idea to erase the flash. Run the following command.

```
esptool.py erase_flash
```

You must be connected (by running the command in the previous section) for this command to work as shown.



Installing the Bootloader

Run this command and replace the serial port name with your matching port and the file you just downloaded

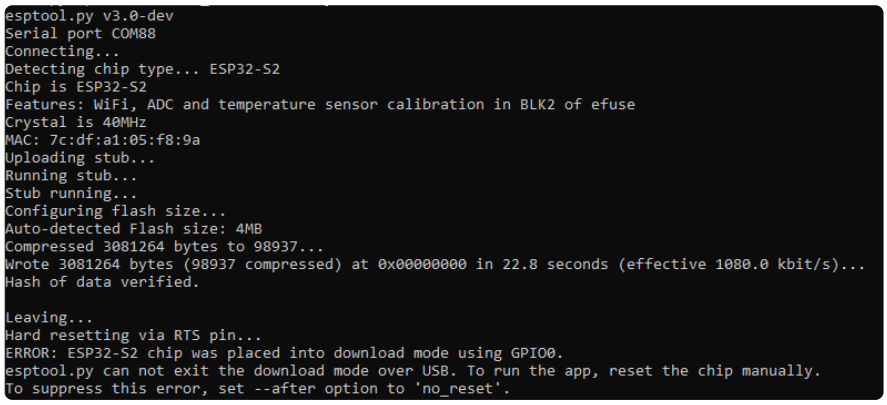
```
esptool.py --port COM88 write_flash 0x0 tinyuf2_combo.bin
```

Don't forget to change the `--port` name to match.

Adjust the bootloader filename accordingly if it differs from tinyuf2_combo.bin.

There might be a bit of a 'wait' when programming, where it doesn't seem like it's working. Give it a minute, it has to erase the old flash code which can cause it to seem like it's not running.

You'll finally get an output like this:



Once completed, you can continue to the next section.

Reset the board

Now that you've reprogrammed the board, you need to reset it to continue. Click the reset button to launch the new firmware.

The display will show the FunHouse sensor data, and the DotStar LEDs will light up in a rainbow swirl.

You've successfully returned your board to a factory reset state!

Older Versions of Chrome

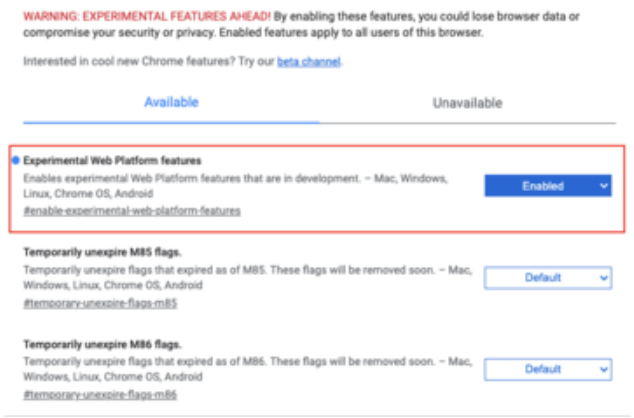
As of chrome 89, Web Serial is already enabled, so this step is only necessary on older browsers.

We suggest updating to Chrome 89 or newer, as Web Serial is enabled by default.

If you must continue using an older version of Chrome, follow these steps to enable Web Serial.

If you receive an error like the one shown when you visit the WebSerial ESPTool site, you're likely running an older version of Chrome.

You must be using Chrome 78 or later to use Web Serial.



To enable Web Serial in Chrome versions 78 through 88:

Visit **chrome://flags** from within Chrome.
Find and enable the **Experimental Web Platform features**
Restart Chrome

The Flash an Arduino Sketch Method

This section outlines flashing an Arduino sketch onto your ESP32-S2/S3 board, which automatically installs the UF2 bootloader as well.

Arduino IDE Setup

If you don't already have the Arduino IDE installed, the first thing you will need to do is to download the latest release of the Arduino IDE. ESP32-S2/S3 requires **version 1.8** or higher. Click the link to download the latest.

Arduino IDE Download

https://adafru.it/Pd5

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** > **Preferences** menu in Windows or Linux, or the **Arduino** > **Preferences** menu on OS X.

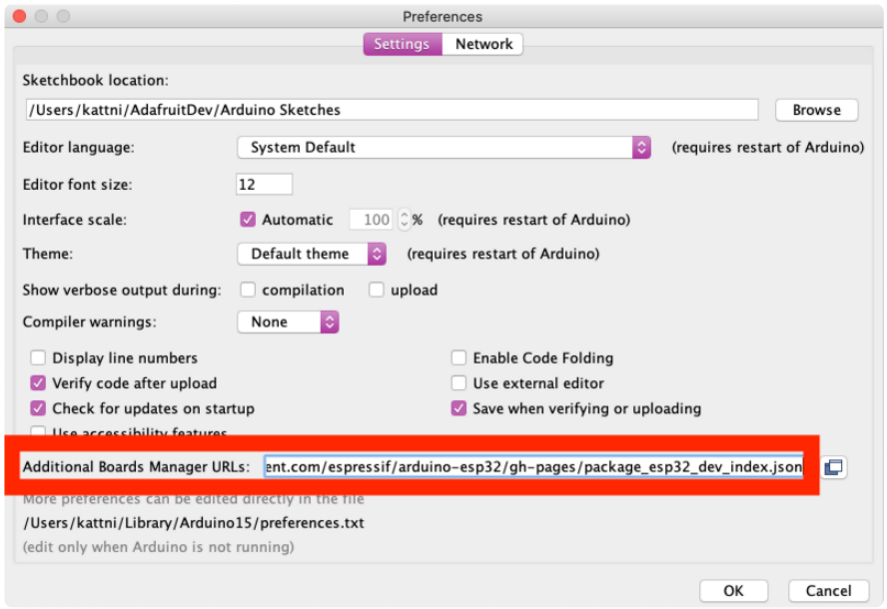
The **Preferences** window will open.

In the **Additional Boards Manager URLs** field, you'll want to add a new URL. The list of URLs is comma separated, and you will only have to add each URL once. The URLs point to index files that the Board Manager uses to build the list of available & installed boards.

Copy the following URL.

```
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_dev_index.json
```

Add the URL to the the **Additional Boards Manager URLs** field (highlighted in red below).



Click **OK** to save and close **Preferences**.

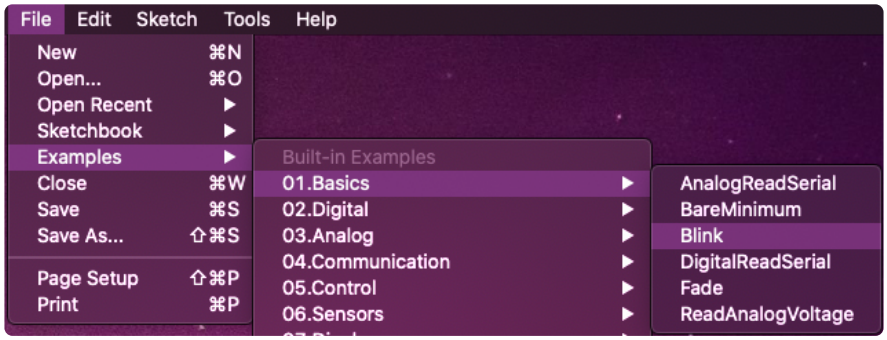
In the **Tools** > **Boards** menu you should see the **ESP32 Arduino** menu. In the expanded menu, it should contain the ESP32 boards along with all the latest ESP32-S2 boards.

Now that your IDE is setup, you can continue on to loading the sketch.

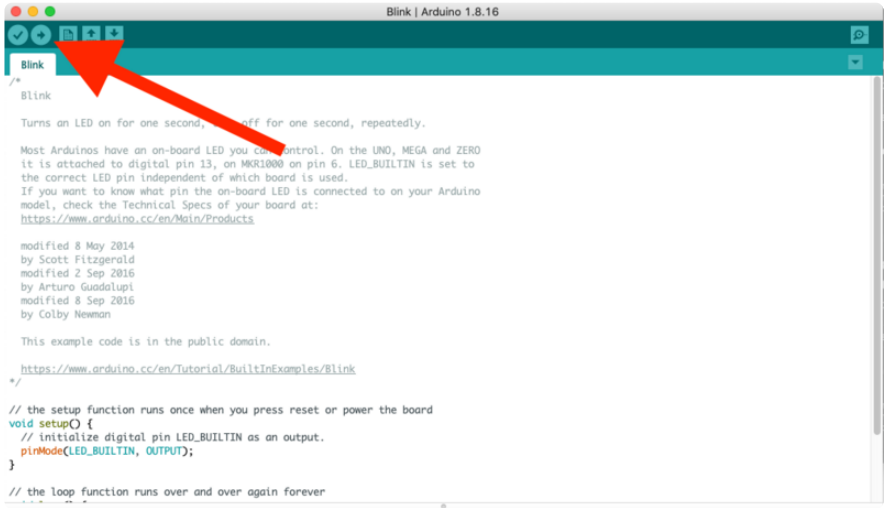
Load the Blink Sketch

In the **Tools** > **Boards** menu you should see the **ESP32 Arduino** menu. In the expanded menu, look for the menu option for the **Adafruit FunHouse**, and click on it to choose it.

Open the Blink sketch by clicking through **File** > **Examples** > **01.Basics** > **Blink**.



Once open, click Upload from the sketch window.



Once successfully uploaded, the little red LED will begin blinking once every second. At that point, you can now enter the bootloader.

Install UF2 Bootloader

If your board has a UF2 bootloader, you do not need to follow the steps on this page. Try to enter the UF2 bootloader before continuing! Double-tap the reset button to do so.

The Adafruit FunHouse ships with a UF2 bootloader which allows the board to show up as **HOUSEBOOT** when you double-tap the reset button, and enables you to drag and drop UF2 files to update the firmware.

On ESP32-S2/S3, there is no bootloader protection for the UF2 bootloader. That means it is possible to erase or damage the UF2 bootloader, especially if you upload an Arduino sketch to an ESP32-S2/S3 board that doesn't "know" there's a bootloader it should not overwrite!

It turns out, however, the ESP32-S2/S3 comes with a second bootloader: the ROM bootloader. Thanks to the ROM bootloader, you don't have to worry about damaging the UF2 bootloader. The ROM bootloader can never be disabled or erased, so its always there if you need it! You can simply re-load the UF2 bootloader from the ROM bootloader.

If your UF2 bootloader ends up damaged or overwritten, you can follow the steps found in the Factory Reset and Bootloader Repair (https://adafru.it/XRc) section of the Factory Reset page in this guide.

Once completed, you'll return to where the board was when you opened the package. Then you'll be back in business, and able to continue with your existing plans!

Downloads

Files:

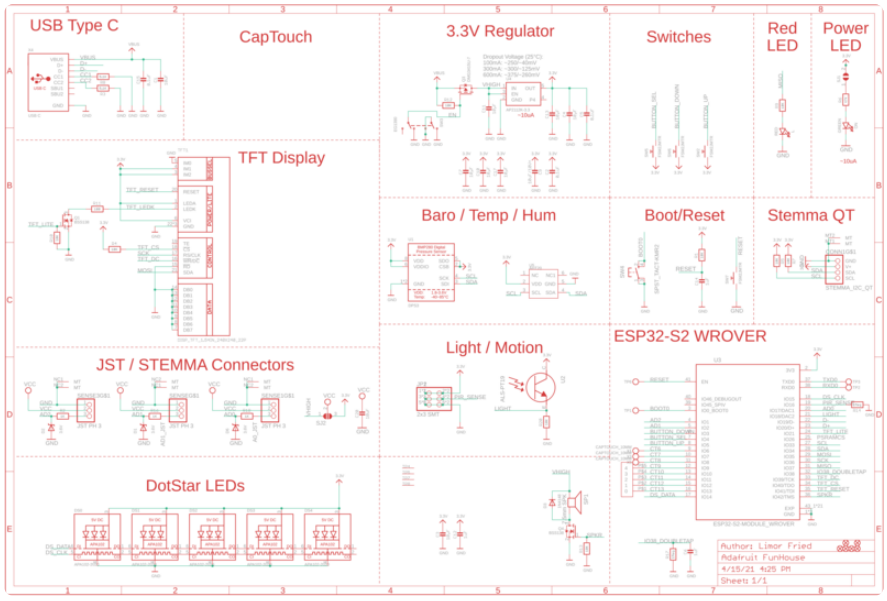
- ESP32-S2 product page with resources (https://adafru.it/OpE)
- ESP32-S2 datasheet (https://adafru.it/OpF)
- ESP32-S2 WROVER datasheet (https://adafru.it/Oqa)
- ESP32-S2 Technical Reference (https://adafru.it/Oqb)
- DPS310 datasheet (https://adafru.it/RKC)
- AHT20 datasheet (https://adafru.it/LAm)
- EagleCAD PCB files on GitHub (https://adafru.it/RKD)
- 3D Models on GitHub (https://adafru.it/Sbt)
- Fritzing object in the Adafruit Fritzing Library (https://adafru.it/RKE)
- PDF for FunHouse pinout diagram (https://adafru.it/ZqC)

SVG for FunHouse pinout diagram

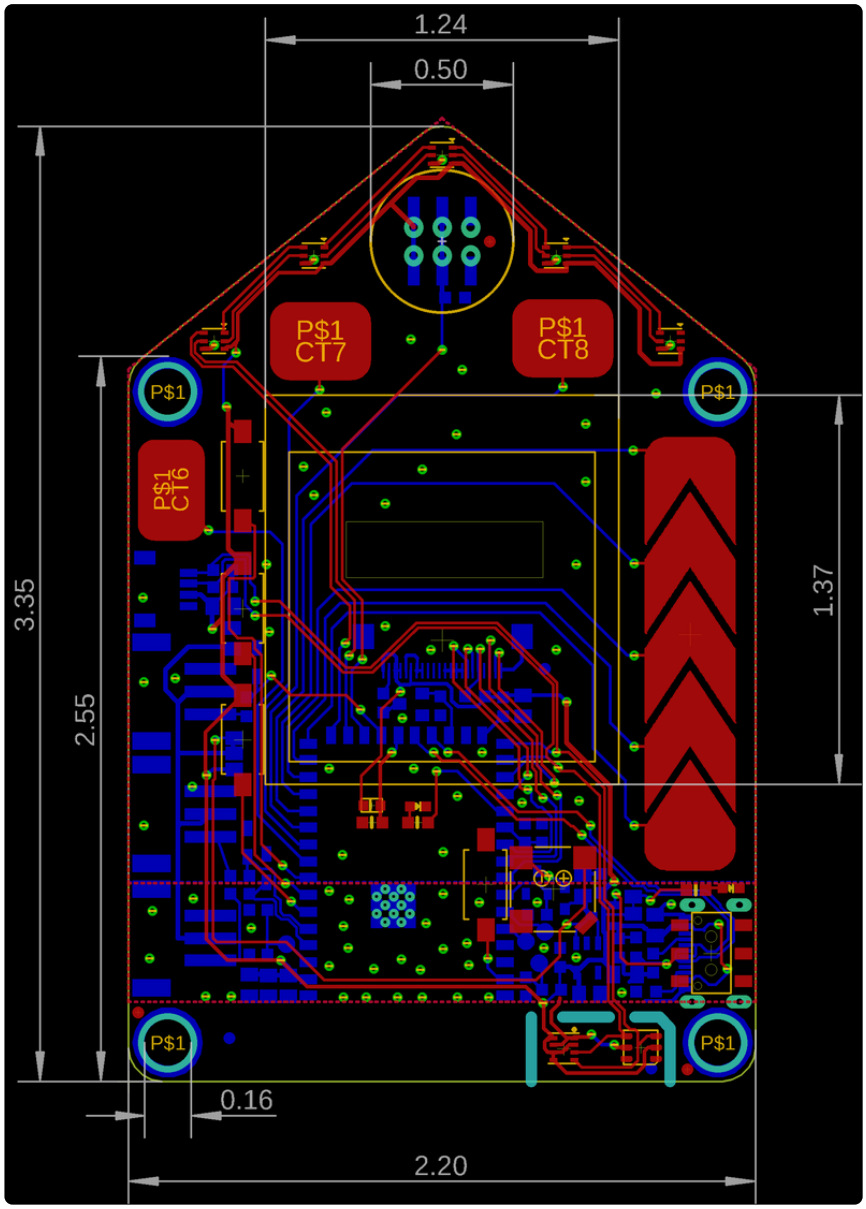
https://adafru.it/ZqD

Schematic

The schematic shows a BMP280, but the FunHouse has a DPS310, which is pin-compatible.



Fab Print



Here's a design file for a mounting bracket -- you can use it as a template for cutting your own by hand, with a laser cutter or mill, or as a jumping off point for modeling one for 3D printing.

FunHouse-wall-mount.svg

https://adafru.it/Sfy

(OLD) WipperSnapper Usage

This page assumes that you have installed WipperSnapper on your board and registered it with the Adafruit.io website. If you have not done this yet, please go back to the previous page in this guide and connect your board.

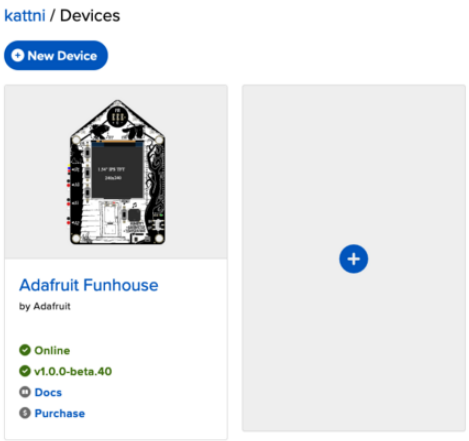
Blink a LED

One of the first programs you typically write to get used to embedded programming is a sketch that repeatably blinks an LED. IoT projects are wireless so after completing this section, you'll be able to turn on (or off) the LED built into your board from anywhere in the world.

In this demo, we show controlling an LED from Adafruit IO. But the same kind of control can be used for relays, lights, motors, or solenoids.

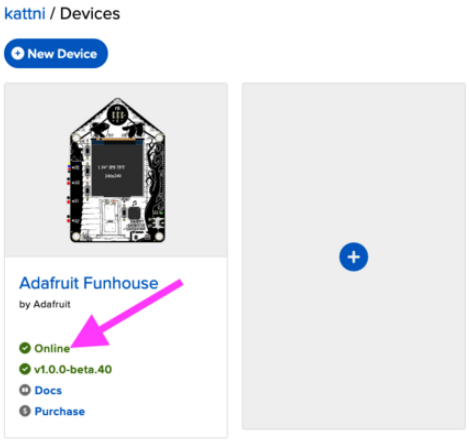
Navigate to io.adafruit.com/wippersnapper (https://adafru.it/TAu). You should see the board you just connected to Adafruit IO listed on this page.

- If you do not see your board - go back to the previous setup page and ensure you have registered it with Adafruit IO

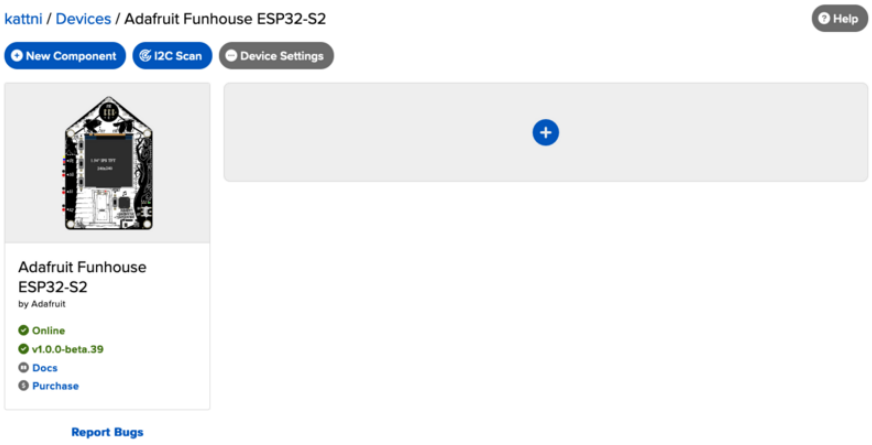


Make sure the board's tile says Online in green text, indicating that it's online and communicating with Adafruit IO.

- If the board appears offline on the website but was previously connected, press the Reset (RST) button (or unplug the USB, and plug it back in) to force the board to reboot.



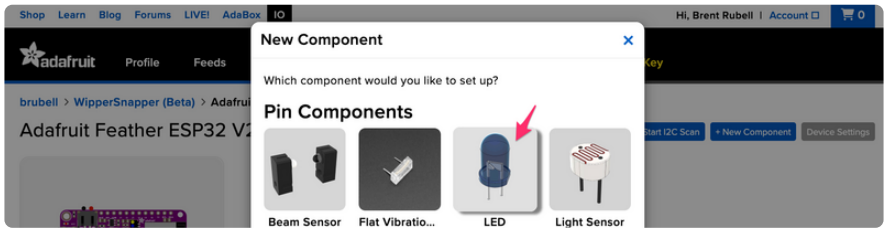
Click the board to navigate to its page.



From the device page, click the + **New Component (or +)** button.

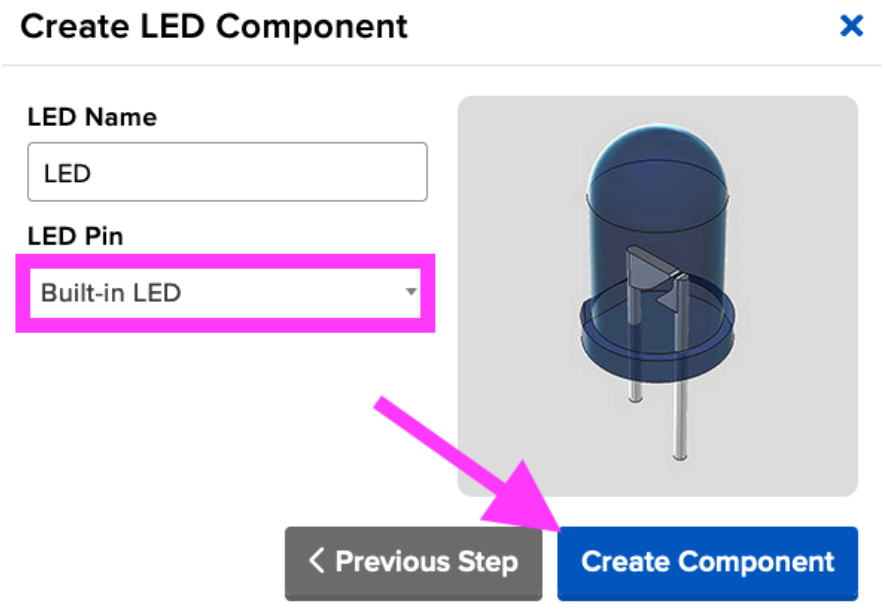
The Component Picker lists all the sensors and actuators which can be used with the WipperSnapper firmware.

Click the LED icon.



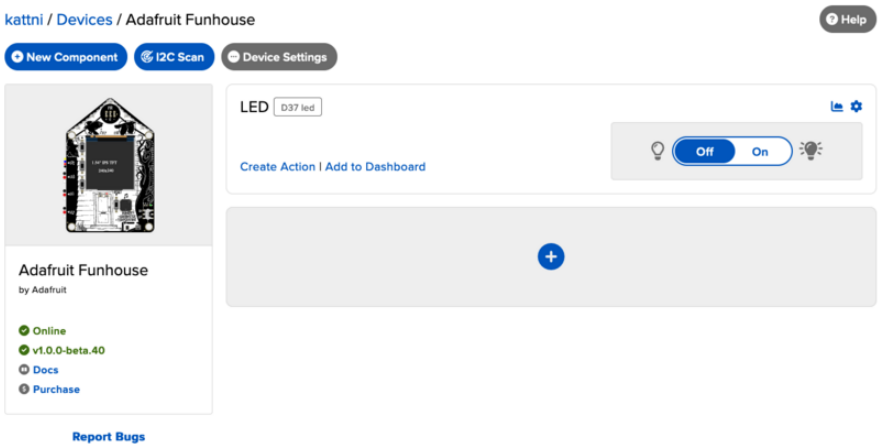
Microcontroller boards contain GPIO pins that can be configured either as an input or an output. The "Create LED Component" screen tells WipperSnapper to configure a general-purpose output pin connected to the LED on your board as a digital output so you can turn the LED on or off.

The FunHouse has a built-in LED located at **GPIO #37 (Built-in LED).** Select this pin as the LED Pin and click **Create Component**

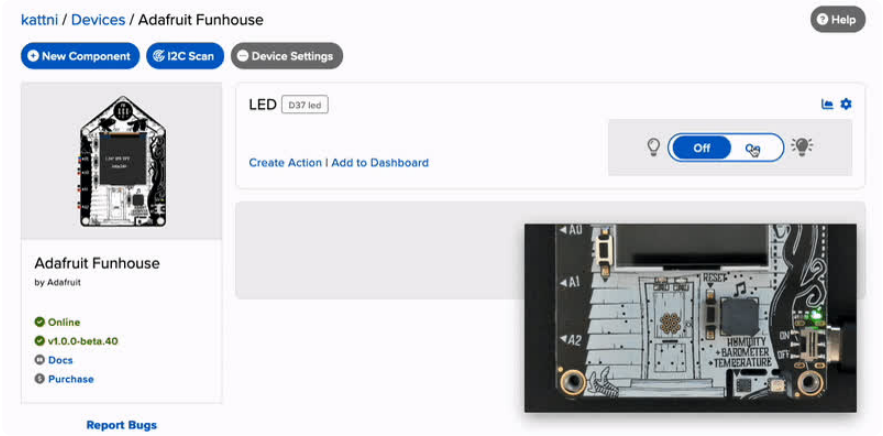


Behind the scenes, Adafruit IO sends send a command to your board running WipperSnapper telling it to configure the GPIO pin as a digital output.

Your board's page on Adafruit IO shows a new LED component.



On the board page, **toggle the LED component by clicking the toggle switch.** This should turn your board's built-in LED on or off.



Read a Push-Button

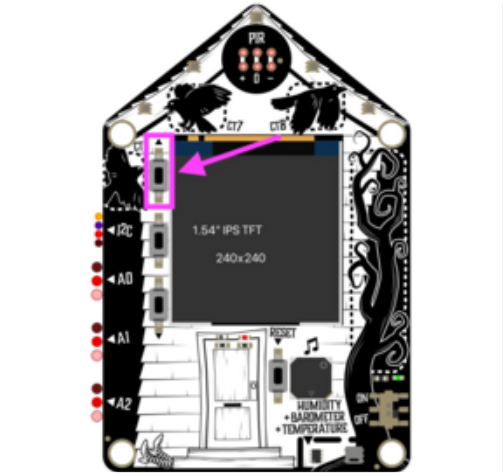
You can also configure a board running WipperSnapper to read data from standard input buttons, switches, or digital sensors, and send the value to Adafruit IO.

Let's wire up a push button to your board and configure it to publish a value to Adafruit IO when the button has been pressed or released.

In this demo, we show reading the state of a push-button from WipperSnapper. But the same kind of control can be used for reading switches, break beam sensors, and other digital sensors.

Wiring

We'll be using the board's internal pull-up resistors instead of a physical resistor.

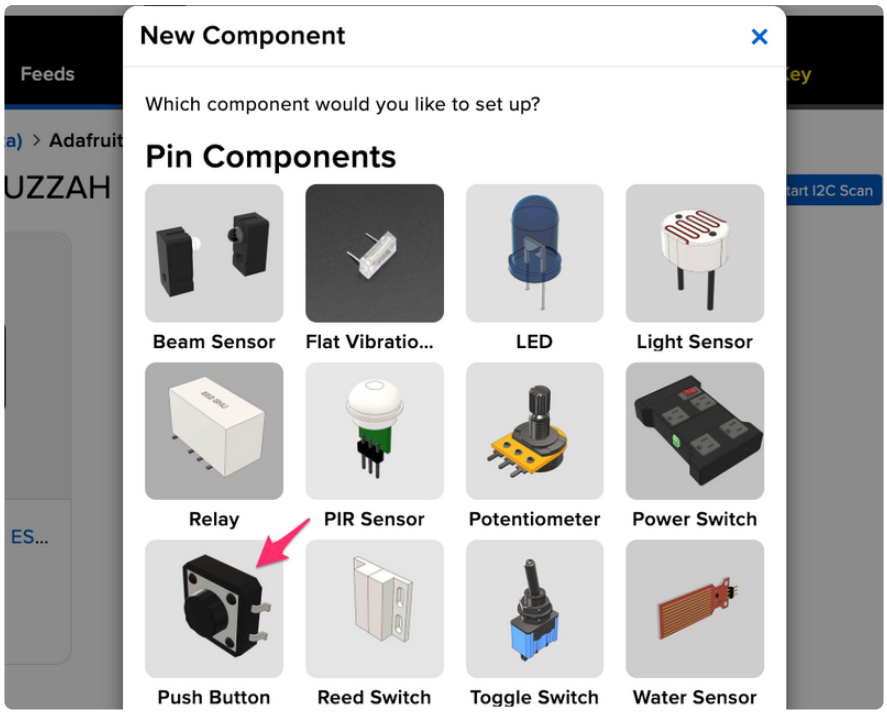


The **up button** is located on the FunHouse near the top left corner of the built-in display, next to the CT6 bird on the board silk.

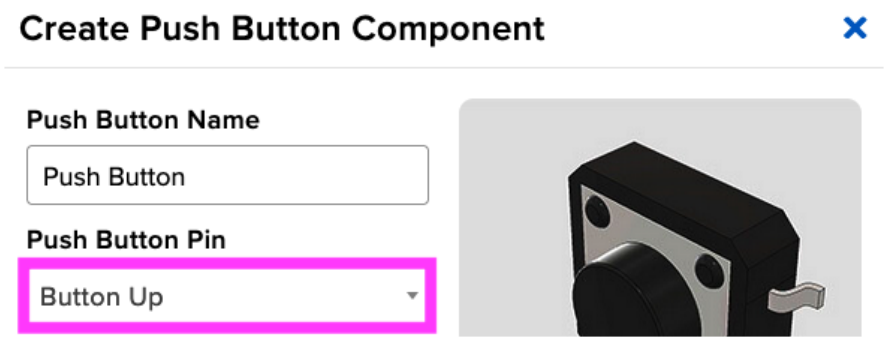
Usage

On the device page, click + **New Component**.

From the component picker, select the **Push Button**.



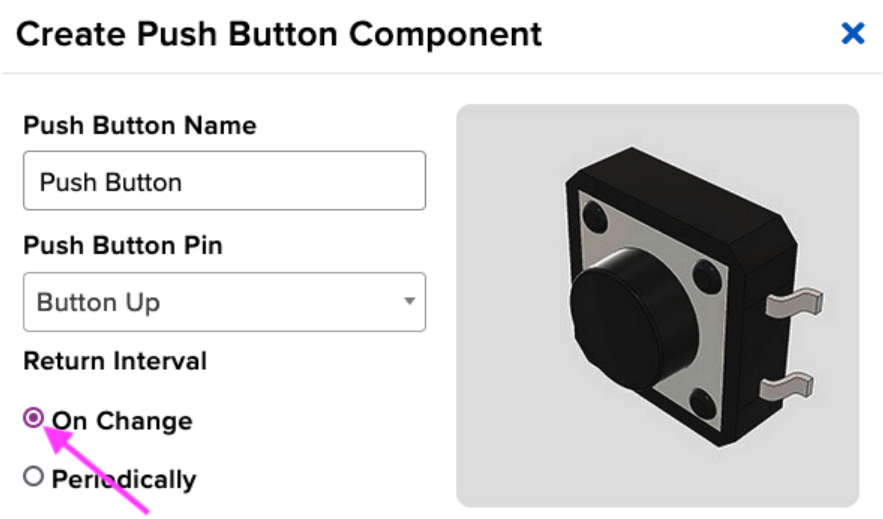
The next screen presents you with options for configuring the push button. Start by selecting the board's pin you connected to the push button.



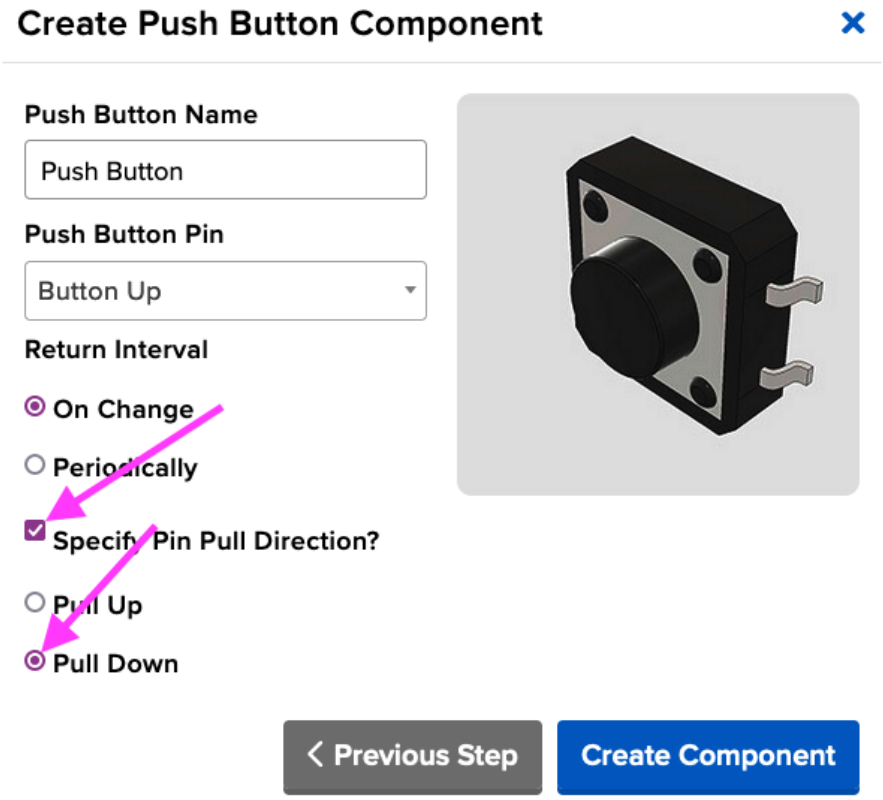
The Return Interval dictates how frequently the value of the push-button will be sent from the board to Adafruit IO.

For this example, you will configure the push button's value to be only sent when the value changes (i.e: when it's either pressed or depressed).

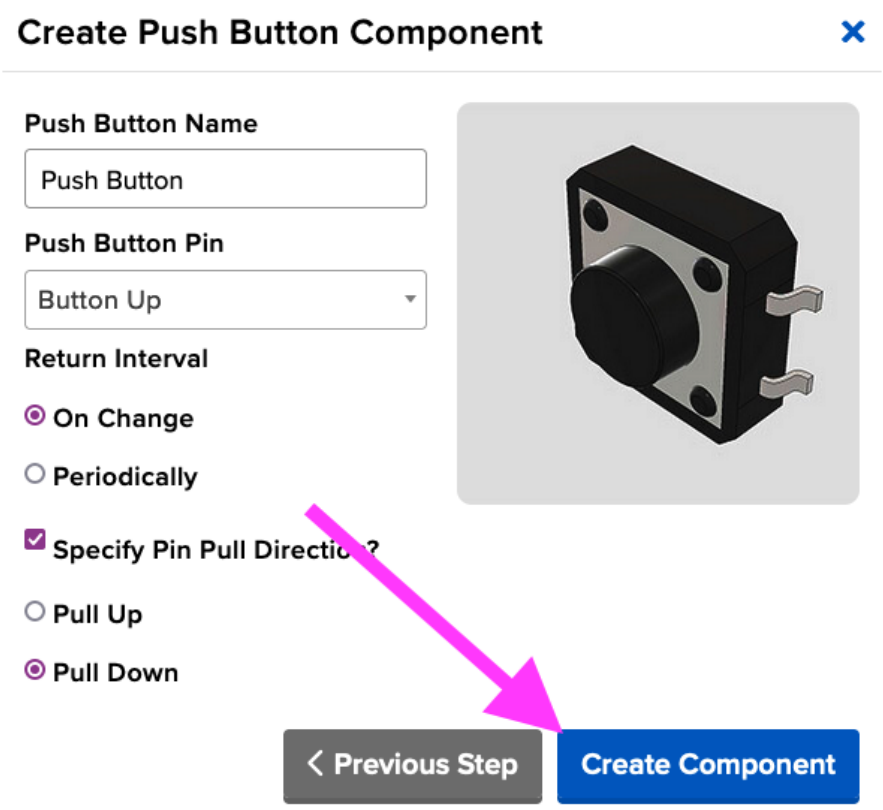
Select **On Change**



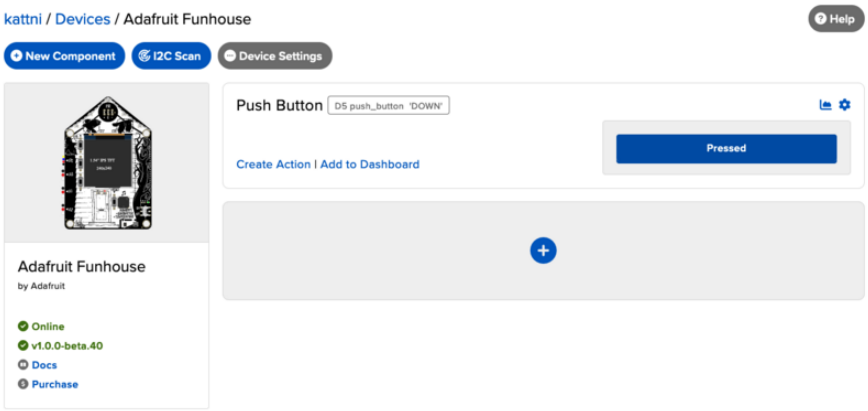
Finally, check the **Specify Pin Pull Direction** checkbox and select **Pull Down** to turn on the board's internal pulldown resistor.



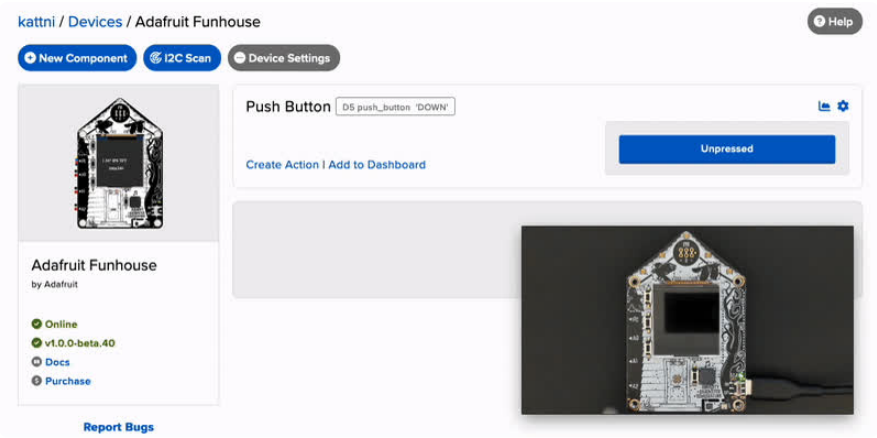
Make sure the form's settings look like the following screenshot. Then, click **Create Component**.



Adafruit IO sends a command to your WipperSnapper board, telling it to configure the GPIO pin you selected to behave as a digital input pin and to enable it to pull up the internal resistor. Your board's page should also show the new push-button component.



Push the button to change the value of the component on Adafruit IO.



Read an I2C Sensor

Inter-Integrate Circuit, aka **I2C**, is a two-wire protocol for connecting sensors and "devices" to a microcontroller. A large number of sensors, including the ones sold by Adafruit, use I2C to communicate.

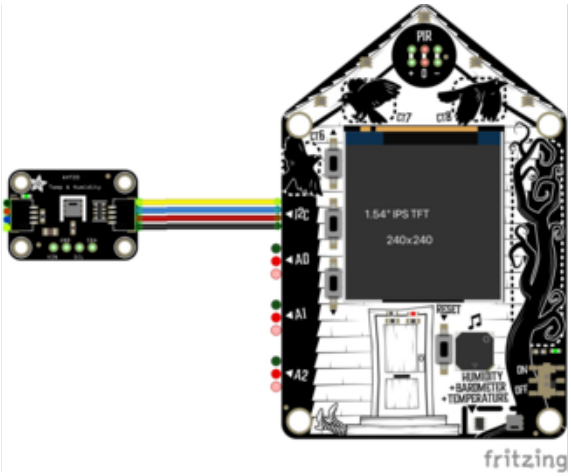
Typically, using I2C with a microcontroller involves programming. Adafruit IO and WipperSnapper let you configure a microcontroller to read data from an I2C sensor and publish that data to the internet without writing code.

The WipperSnapper firmware supports a number of I2C sensors, viewable in list format here (https://adafru.it/Zbq). If you do not see the I2C sensor you're attempting to use with WipperSnapper - we have a guide on adding a component to Adafruit IO WipperSnapper here (https://adafru.it/Zbr).

The process for adding an I2C component to your board running WipperSnapper is similar for most sensors. For this section, we're using the Adafruit AHT20 (http://adafru.it/4566), an inexpensive sensor that can read ambient temperature and humidity.

Wiring

First, wire up an AHT20 sensor to your board exactly as follows. Here is an example of the AHT20 wired using I2C with a STEMMA QT cable (no soldering required) (http://adafru.it/4209).



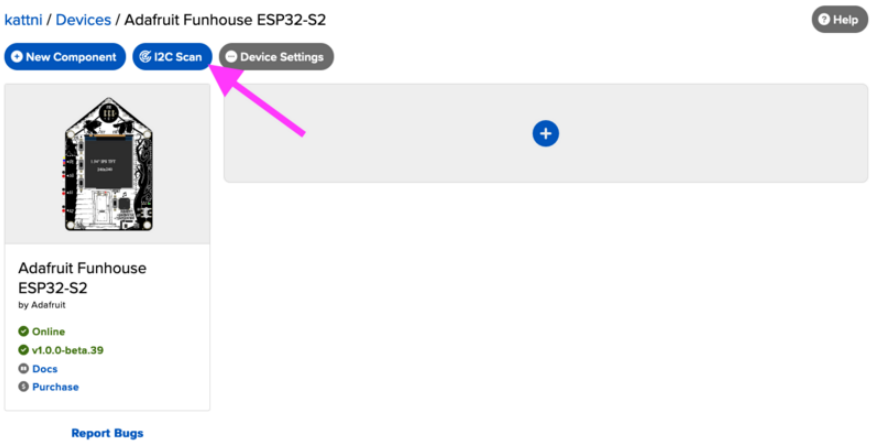
Simply connect a STEMMA QT to STEMMA QT cable from the STEMMA QT port on your board to a STEMMA QT port on your sensor.

Scan I2C Bus

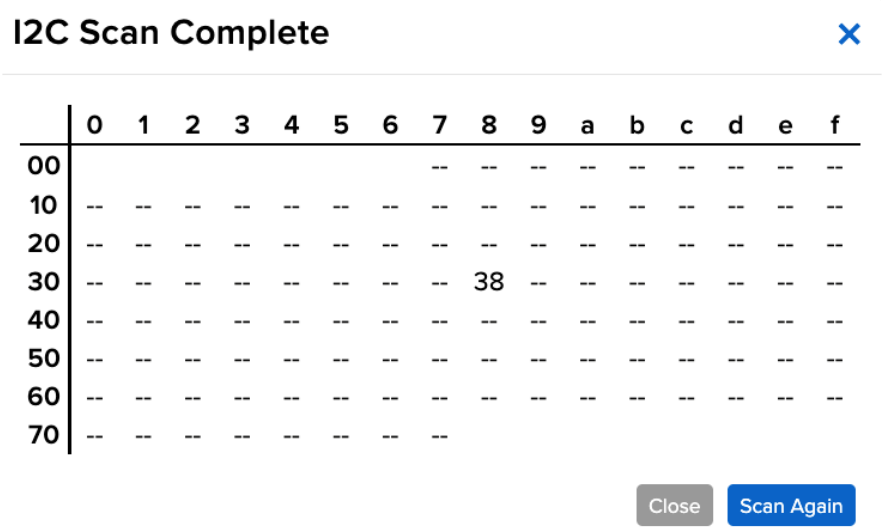
First, ensure that you've correctly wired the AHT20 sensor to your board by performing an I2C scan to detect the I2C device on the bus.

On the board page, click **Start I2C Scan**.

- If you do not see this button, double-check that your board shows as Online.



You should see a new pop-up showing a list of the I2C addresses detected by an I2C scan. If wired correctly, the AHT20's default I2C address of `0x38` appear in the I2C scan list.



I don't see the I2C sensor's address in the list

First, double-check the connection and/or wiring between the sensor and the board.

Then, reset the board and let it re-connect to Adafruit IO WipperSnapper.

Create the Sensor Component

Now that you know the sensor can be detected by the board, it's time to configure and create the sensor on Adafruit IO.

On the board page, add a new component to your board by clicking the + **button** or the + **New Component** button.

The Component Picker lists all the available components, sensors, and parts that can be used with WipperSnapper.

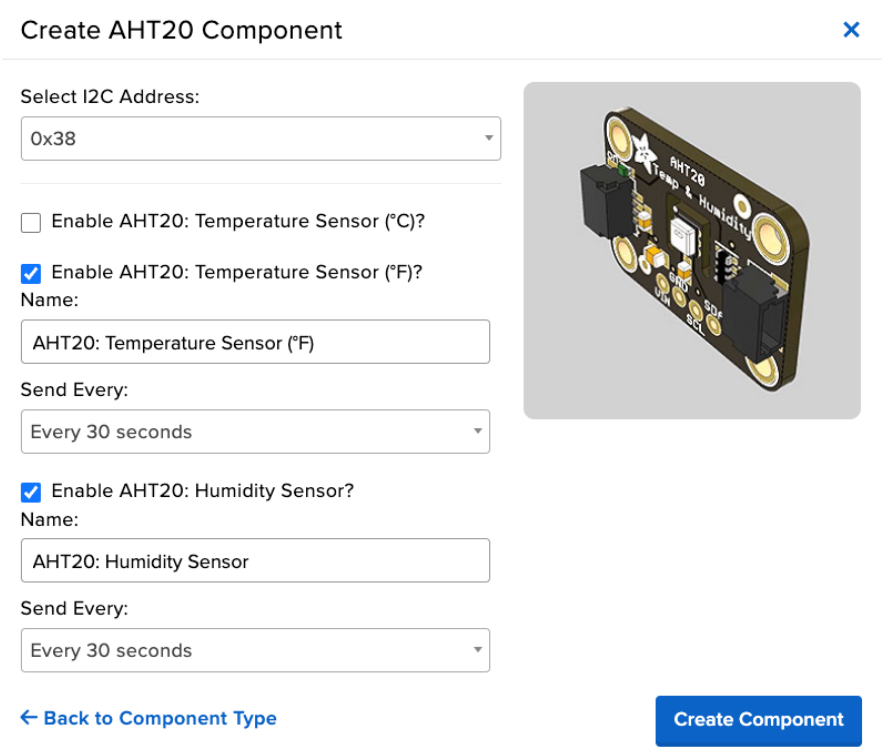
Under the **I2C Components** header, click **AHT20.**

On the component configuration page, the AHT20's I2C sensor address should be listed along with the sensor's settings.

The AHT20 sensor can measure ambient temperature and relative humidity. This page has individual options for reading the ambient temperature, in either degree Celsius or degree Fahrenheit, and the relative humidity. You may select the readings which are appropriate to your application and region.

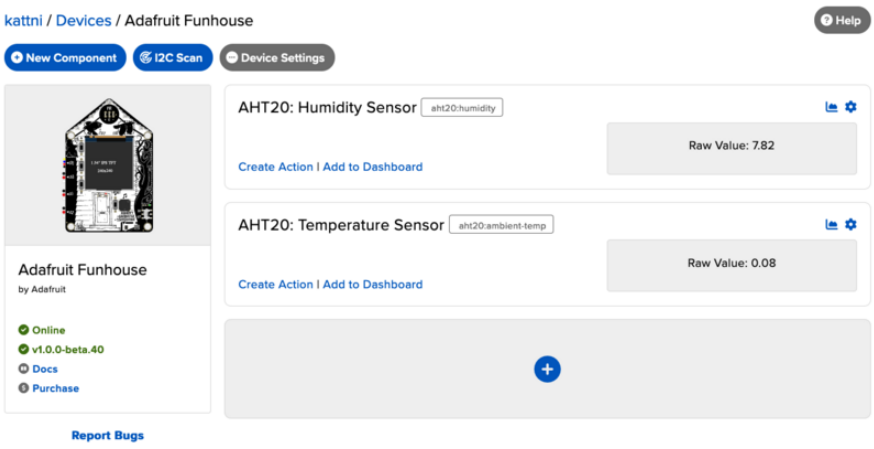
The **Send Every** option is specific to each sensor measurement. This option will tell the Feather how often it should read from the AHT20 sensor and send the data to Adafruit IO. Measurements can range from every 30 seconds to every 24 hours.

For this example, set the **Send Every** interval for both seconds to **Every 30 seconds** and click **Create Component**.



The board page should now show the AHT20 component you created.

After the interval you configured elapses, WipperSnapper automatically reads values from the sensor and sends them to Adafruit IO.



Going Further

Want to learn more about Adafruit IO WipperSnapper? We have more complex projects on the Adafruit Learning System (https://adafru.it/Yyd).