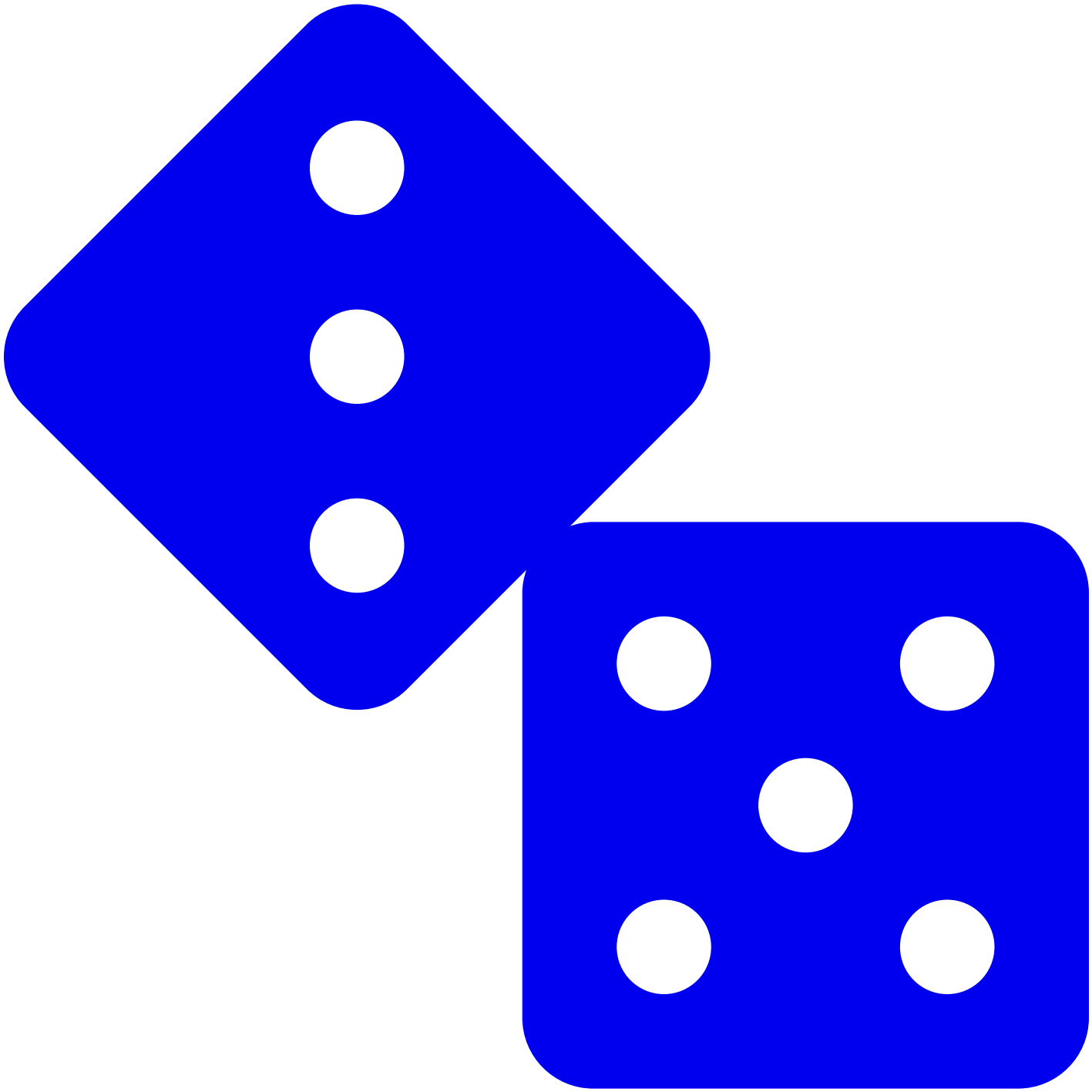
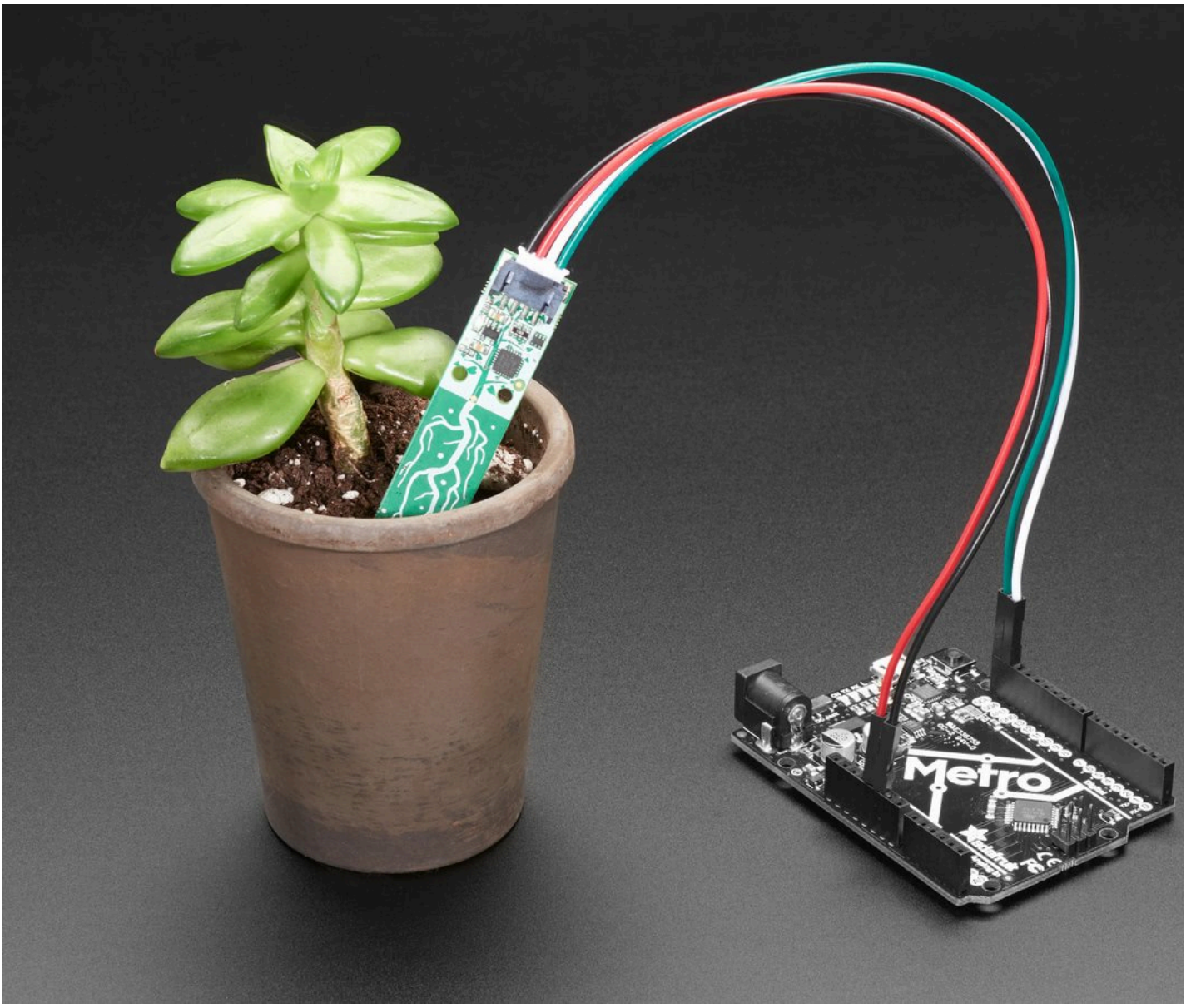


- Explore & Learn

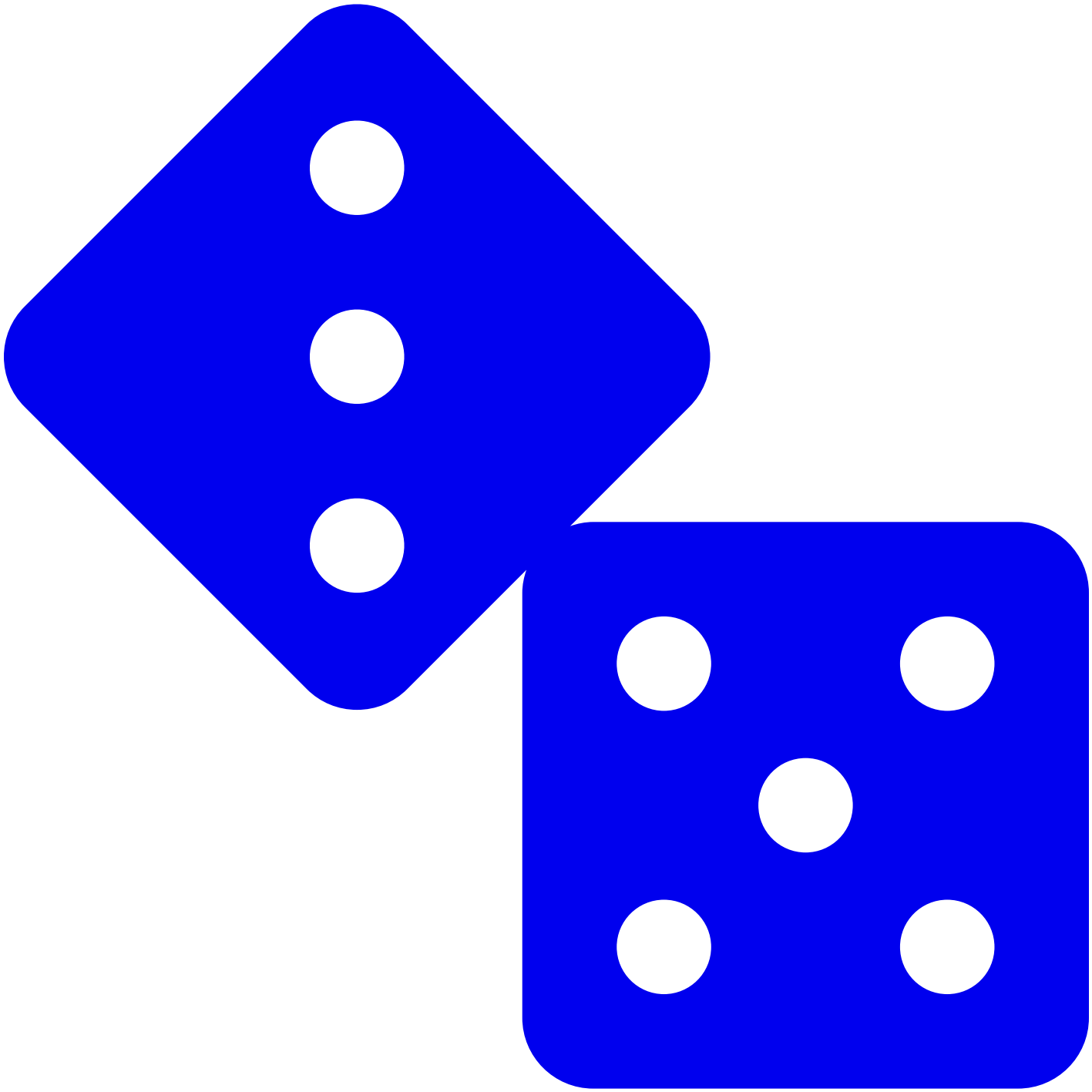
Learn Categories

- 3D Printing
- AdaBox
- Adafruit Products
- Arduino Compatibles
- Breakout Boards
- Circuit Playground
- CircuitPython
- CLUE
- Community Support
- Components
- Crickit
- Customer & Partner Projects
- Development Boards
- Educators
- EL Wire/Tape/Panel
- Feather
- Gaming
- Hacks
- Internet of Things - IOT
- LCDs & Displays
- LEDs
- Machine Learning
- MakeCode
- Maker Business
- micro:bit
- Microcontrollers
- Programming
- Raspberry Pi
- Robotics & CNC
- Sensors
- STEMMA
- Tools
- Trellis
- Wearables



Explore
Groups to get your gears turning
Explore Groups

- New Guides
- Playground



Introducing Feather RP2040 SCORPIO

Introducing Feather RP2040 SCORPIO

by Phillip Burgess and 2 other contributors
Contributors:

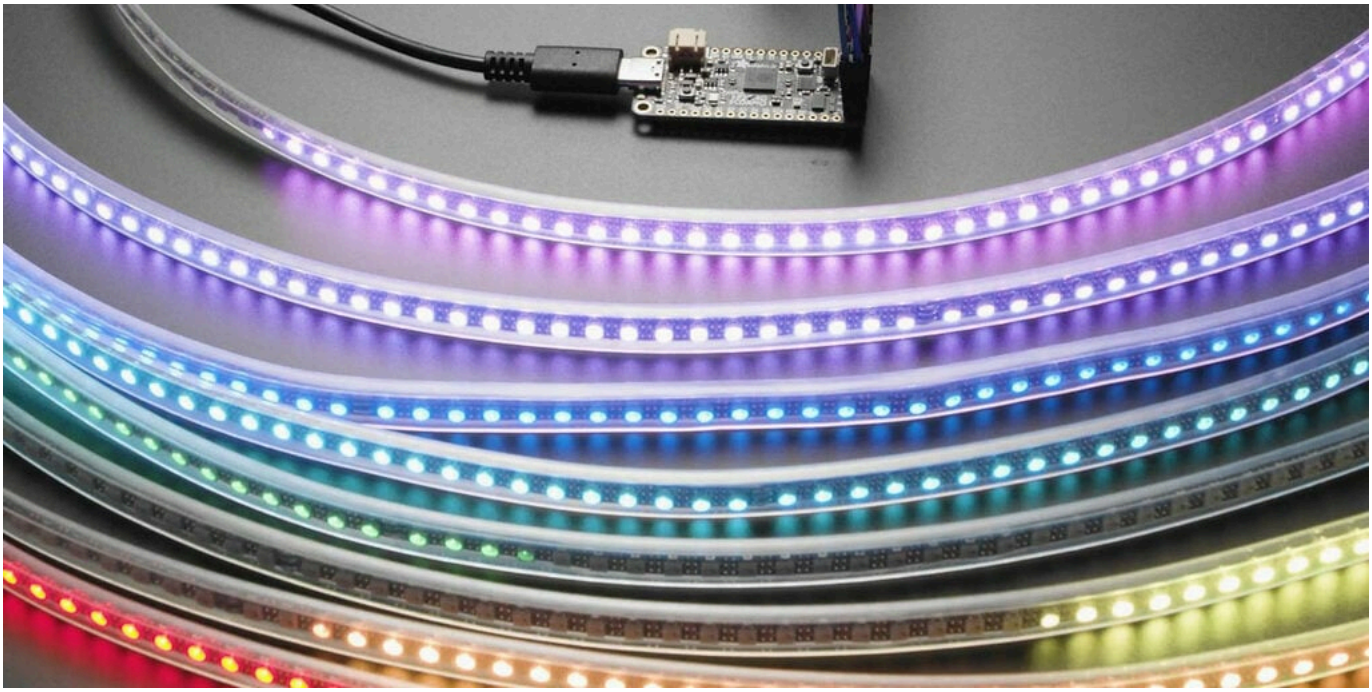
- lady ada
- Kattni Rembor

published December 27, 2022, last edited January 22, 2025
last major update January 05, 2023
posted in Adafruit Products LEDs/ LED Pixels CircuitPython Feather/ Feather Boards STEMMA

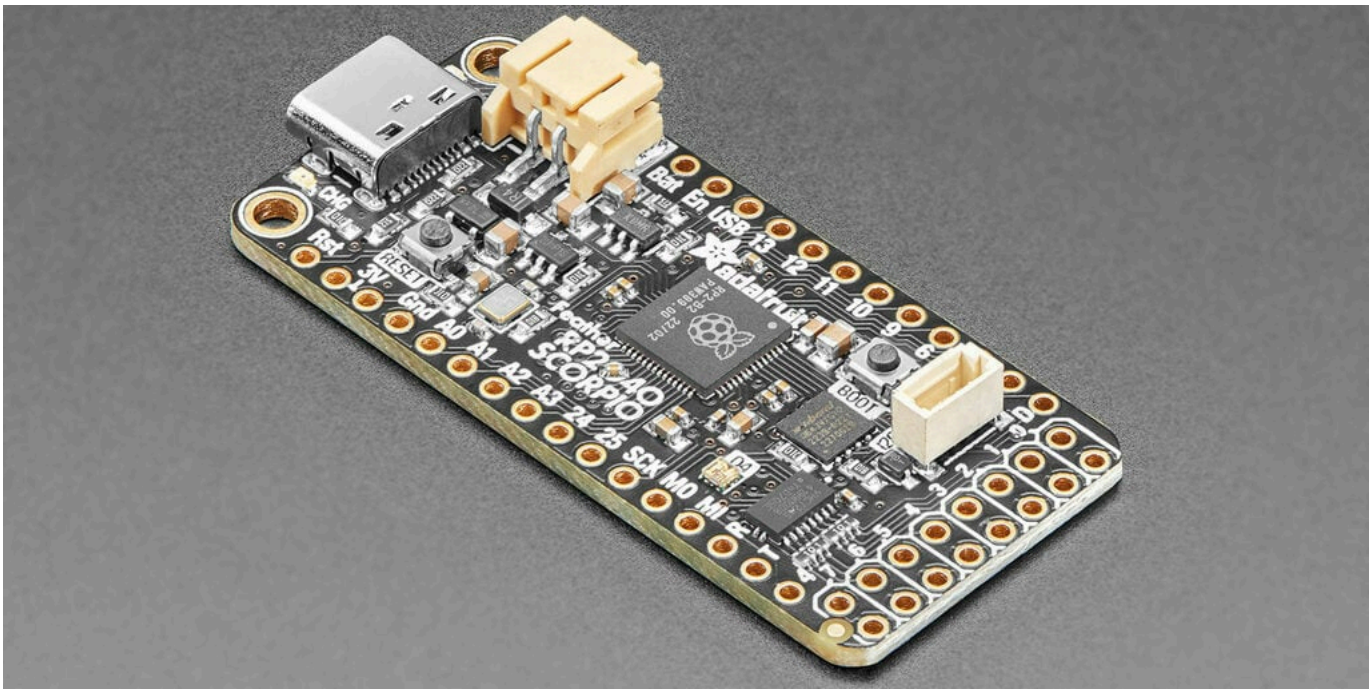
Primary Products

Adafruit Feather RP2040 SCORPIO - 8 Channel NeoPixel Driver
$14.50
Add to Cart
77
Beginner
Product guide
1

Overview



If there is one thing Adafruit is known for, its mega-blinky-fun-rainbow-LEDs. We just love sticking NeoPixels anywhere and everywhere. When we saw the new "PIO" feature of the RP2040 from Raspberry Pi, we knew it would be perfect for driving huge numbers of NeoPixels. So we created the **Adafruit Feather RP2040 SCORPIO** specifically for NeoPixel (WS2812-compatible) control, but also good for various other PIO-based projects that want to take advantage of the Feather pinout with an additional 8 consecutive outputs (or inputs).



The RP2040 PIO state machine is ideal for NeoPixel control: it can generate perfect waveforms, with up to 8 outputs concurrently, all through DMA. That means no processor time is wasted to bit-bang the LED data. Just set up the buffer and tell the PIO peripheral to "make it so." It issues that data to the 8 outputs without delay while your code can continue to read buttons, play music, run CircuitPython — whatever you like!

SCORPIO has a clever pinout, where all the standard Feather pins are the same as the GPIO numbers, plus the standard I2C, SPI and UART lines — and theres *still* enough pins left over to have an extra 8 consecutive pins for PIO usage on GPIO16 through GPIO23.

To make NeoPixel usage glitch-free there is a 3V-to-5V level shifter so the output logic is 5V. If you want 3V signals, the shifter voltage is adjustable with a jumper on the bottom. It's also possible to flip the direction of the level shifter to make the 8 I/O pins *inputs* — say for making a logic analyzer — with a directional jumper selection also on the bottom of the PCB.

The RP2040 SCORPIO also has a *ton* of RAM, 264KB, making it trivial to buffer huge numbers of NeoPixels…*several thousand* if needed. In fact there's so much RAM you can even *dither* the pixels to for finer brightness control, for better-looking LEDs at low brightness or for gamma correction.

We have NeoPXL8 driver code available in Arduino and CircuitPython, so you can jump immediately to making beautiful artworks driven by the Adafruit SCORPIO.

- Measures 2.0" x 0.9" x 0.28" (50.8mm x 22.8mm x 7mm) without headers soldered in
- Light as a (large?) feather - 5 grams

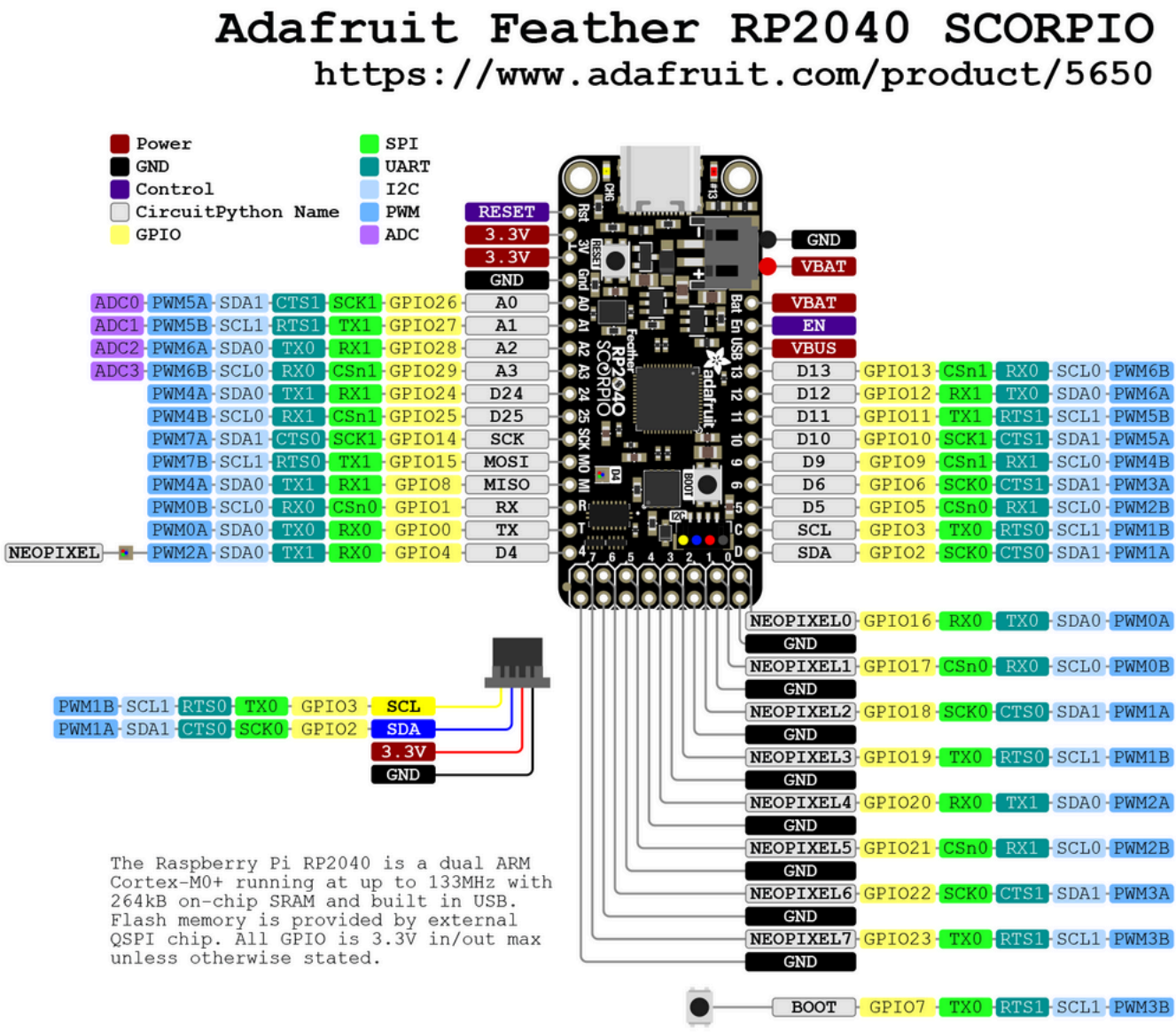
- RP2040 32-bit Cortex M0+ dual core running at ~125 MHz @ 3.3V logic and power
- 264 KB RAM
- **8 MB SPI FLASH** chip for storing files and CircuitPython/MicroPython code storage. No EEPROM
- **Tons of GPIO! 21 x GPIO pins with following capabilities:**
  - **Four** 12-bit ADCs (one more than Pico)
  - Two I2C, Two SPI, and two UART peripherals, we label one for the "main" in standard Feather locations
  - 16 x PWM outputs - for servos, LEDs, etc.
- **Plus:**
  - 8 x consecutive GPIO outputs (or inputs) with 5V level shifting for NeoPixel control
- **Built-in 200mA+ lipoly charger** with charging status indicator LED
- **Pin #13 red LED** for general purpose blinking
- **RGB NeoPixel** for full-color indication on **D4**
- On-board **STEMMA QT connector** that lets you quickly connect any Qwiic, STEMMA QT or Grove I2C devices with no soldering!
- **Both Reset button and Bootloader select button for quick restarts** (no unplugging-replugging to relaunch code). Bootloader button is also available as user-input button on GPIO #7
- 3.3V Power/enable pin
- 2 mounting holes
- 12 MHz crystal for perfect timing
- 3.3V regulator with 500mA peak current output
- **USB Type C connector** lets you access built-in ROM USB bootloader and serial port debugging

Page last edited March 08, 2024

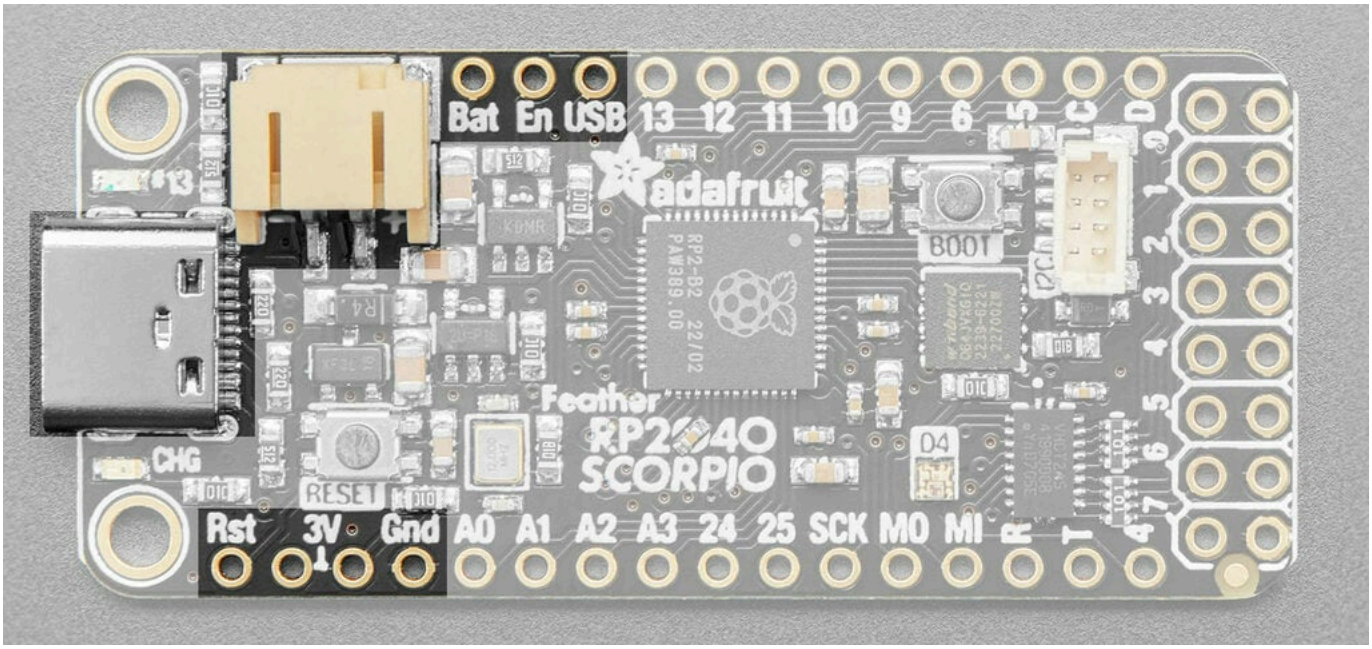
Text editor powered by tinymce.

Pinouts

Adafruit's *Feather RP2040 SCORPIO* has many pins, ports and features. This page takes you on a tour of the board!

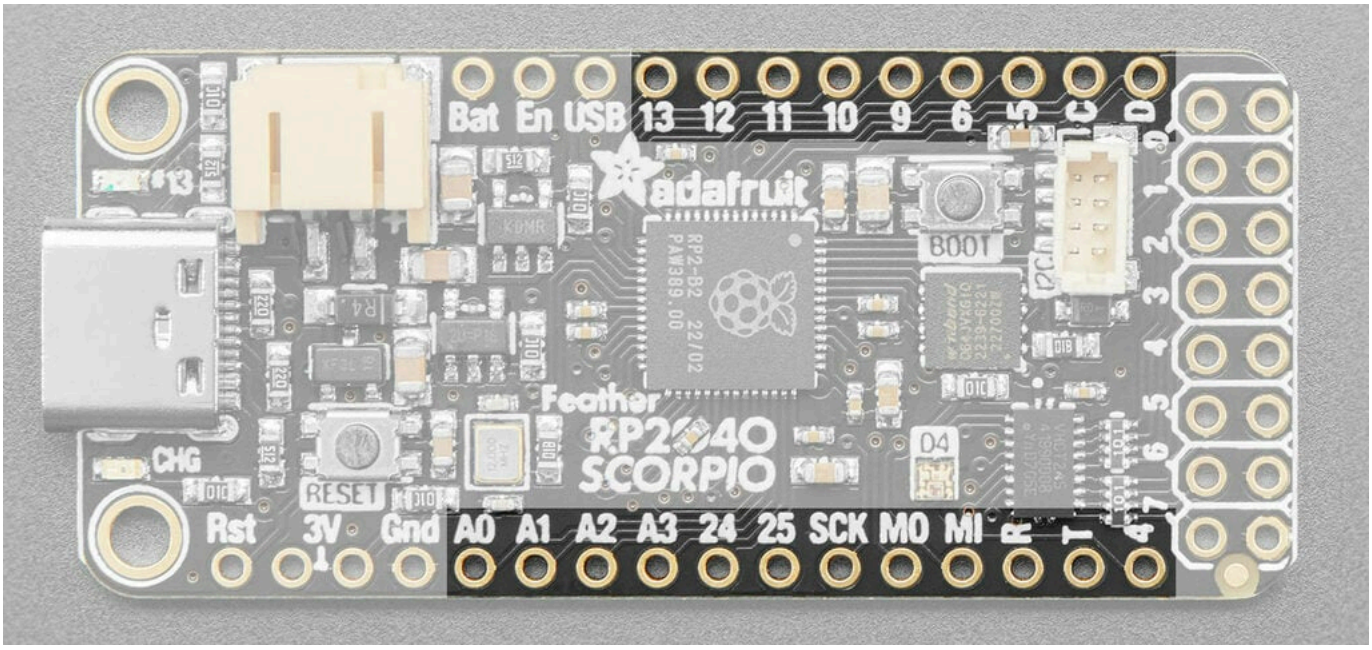


Power Pins and Connections



- **USB C connector** - This is used for power and data. Connect to your computer via a USB C cable to update firmware and edit code.
- **LiPoly Battery connector** - This 2-pin JST PH connector allows you to plug in lithium-polymer batteries to power the Feather. The Feather is also capable of charging batteries plugged into this port via USB.
- **GND** - This is the common ground for all power and logic.
- **BAT** - This is the positive voltage to/from the 2-pin JST jack for the optional LiPoly battery.
- **USB** - This is the positive voltage to/from the USB C jack, if USB is connected.
- **EN** - This is the 3.3V regulator's enable pin. This has a pull-up resistor so the board defaults to “on.” Connecting this to ground (typically through a toggle switch) powers off most of the board (battery charging continues to function).
- **3.3V** - These pins are the output from the 3.3V regulator, which can supply up to 500mA.

Logic Pins



I2C and SPI on RP2040

The RP2040 is capable of handling I2C, SPI and UART (serial) on many pins. However, there are really only two peripherals each of I2C, SPI and UART: I2C0 and I2C1, SPI0 and SPI1, and UART0 and UART1. So while many pins are *capable* of I2C, SPI and UART, you can only *do* two at a time, and only on separate peripherals, 0 and 1. I2C, SPI and UART peripherals are included and numbered below.

PWM on RP2040

The RP2040 supports PWM on all pins. However, it is not capable of PWM on all pins at the same time. There are 8 PWM “slices” (0 through 7), each with two outputs, A and B. Each pin on the Feather is assigned a PWM slice and output. For example, A0 is PWM5 A, which means it is *first* output of the *sixth* slice. With careful planning you can have up to **16** PWM instances on the Feather RP2040. The important thing to know is that **you cannot use the same slice and output more than once at the**

same time. So, if you have a PWM object on pin A0, you cannot also put a PWM object on D10, because they are both PWM5 A. The PWM slices and outputs are indicated below.

Analog Pins

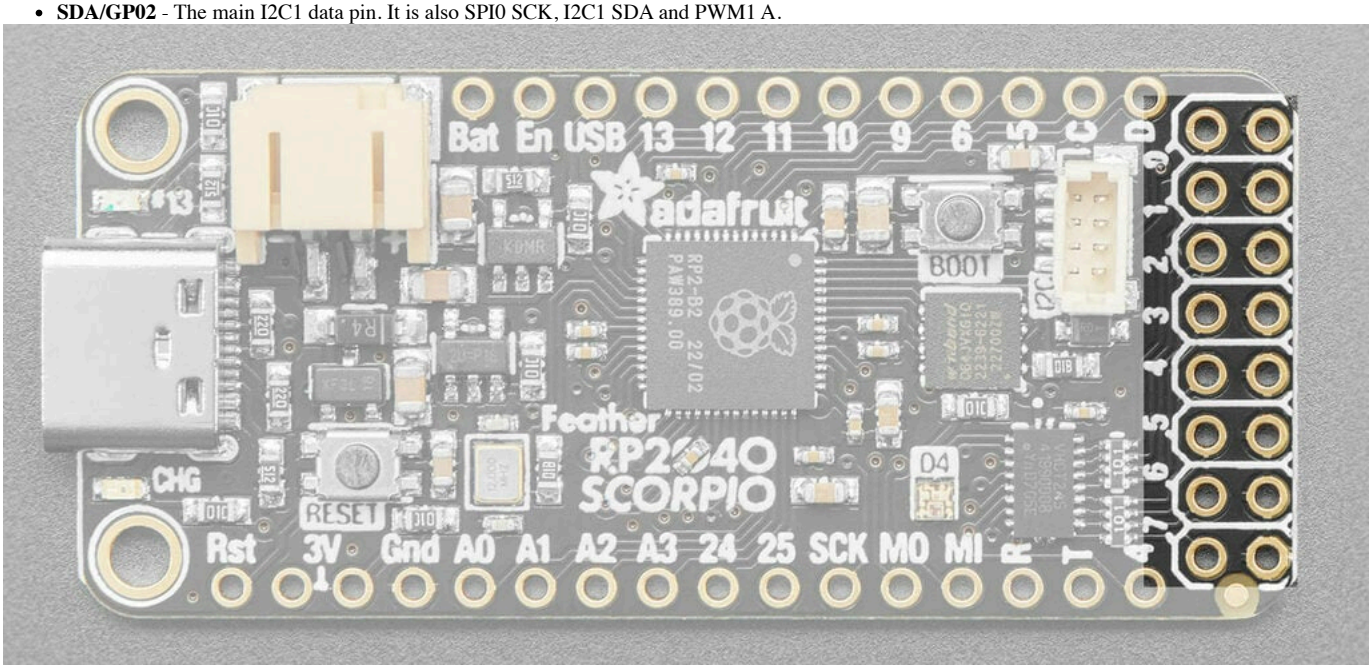
The RP2040 has four ADC inputs. These pins are the only pins capable of handling analog input, though they all can also do digital input or output.

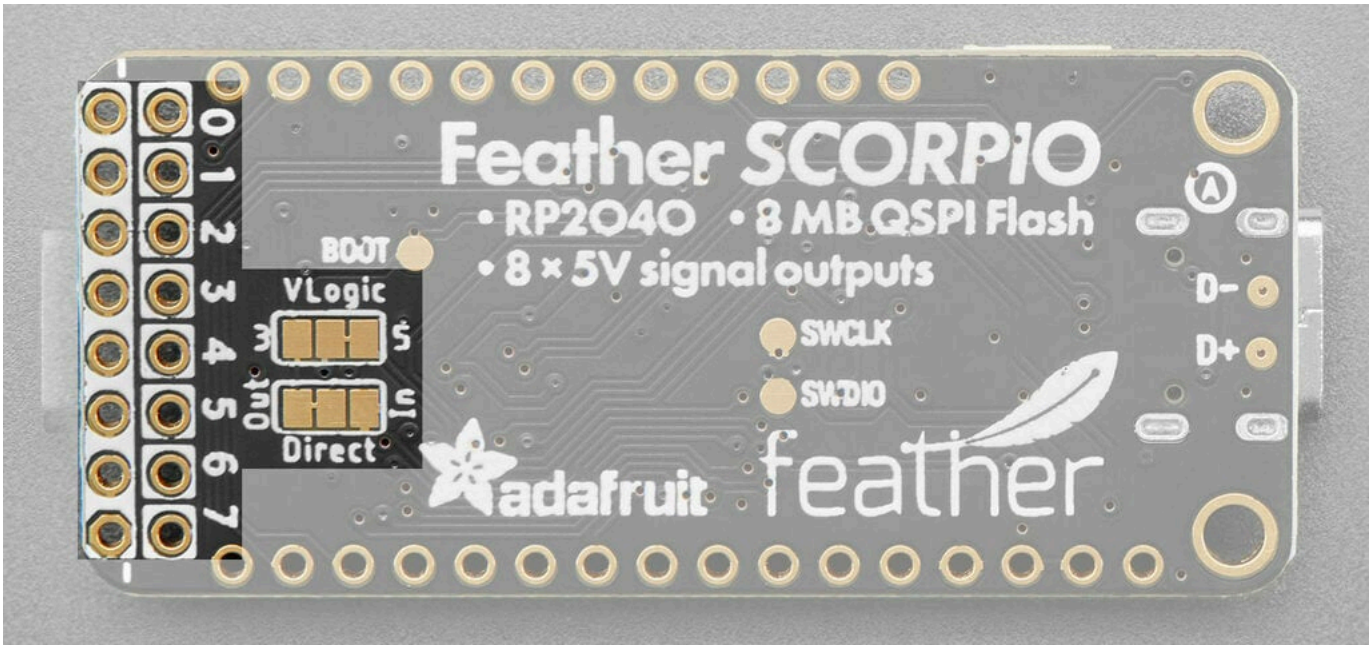
- **A0/GP26** - This pin is ADC0. It is also SPI1 SCK, I2C1 SDA and PWM5 A.
- **A1/GP27** - This pin is ADC1. It is also SPI1 MOSI, I2C1 SCL and PWM5 B.
- **A2/GP28** - This pin is ADC2. It is also SPI1 MISO, I2C1 SDA and PWM6 A.
- **A3/GP29** - This pin is ADC3. It is also SPI1 CS, I2C0 SCL and PWM6 B.

Digital Pins

These are the digital I/O pins along the two longer edges of the board; the “Feather header” pins. They all have multiple capabilities.

- **D24/GP24** - Digital I/O pin 24. It is also UART1 TX, I2C0 SDA, and PWM4 A.
- **D25/GP25** - Digital I/O pin 25. It is also UART1 RX, I2C0 SCL, and PWM4 B.
- **SCK/GP14** - The main SPI1 SCK. It is also I2C1 SDA and PWM7 A.
- **MO/GP15** - The main SPI1 MOSI. It is also I2C1 SCL and PWM7 B.
- **MI/GP08** - The main SPI1 MISO. It is also UART1 TX, I2C0 SDA and PWM4 A.
- **RX/GP01** - The main UART0 RX pin. It is also I2C0 SCL, SPI0 CS and PWM0 B.
- **TX/GP00** - The main UART0 TX pin. It is also I2C0 SDA, SPI0 MISO and PWM0 A.
- **D4/GP04** - Digital I/O pin 4. It is also SPI0 MISO, I2C0 SDA and PWM2 A — but these special peripheral functions should be avoided on this pin as it’s also used for controlling the onboard **NeoPixel**!
- **D13/GP13** - Digital I/O pin 13. It is also SPI1 CS, UART0 RX, I2C0 SCL and PWM6 B.
- **D12/GP12** - Digital I/O pin 12. It is also SPI1 MISO, UART0 TX, I2C0 SDA and PWM6 A.
- **D11/GP11** - Digital I/O pin 11. It is also SPI1 MOSI, I2C1 SCL and PWM5 B.
- **D10/GP10** - Digital I/O pin 10. It is also SPI1 SCK, I2C1 SDA and PWM5 A.
- **D9/GP09** - Digital I/O pin 9. It is also SPI1 CS, UART1 RX, I2C0 SCL and PWM4 B.
- **D6/GP06** - Digital I/O pin 6. It is also SPI0 SCK, I2C1 SDA and PWM3 A.
- **D5/GP05** - Digital I/O pin 5. It is also SPI0 CS, I2C0 SCL and PWM2 B.
- **SCL/GP03** - The main I2C1 clock pin. It is also SPI0 MOSI, I2C1 SCL and PWM1 B.
- **SDA/GP02** - The main I2C1 data pin. It is also SPI0 SCK, I2C1 SDA and PWM1 A.





SCORPIO Header

This special 8x2 board-end header warrants its own in-depth page later. But in brief: the 8 "outer" pins are all grounds, the 8 "inner" are:

- **NEOPIXEL0/GP16** - Digital I/O pin 16. Also SPI0 MISO, UART0 TX, I2C0 SDA and PWM0 A.
- **NEOPIXEL1/GP17** - Digital I/O pin 17. Also SPI0 CS, UART0 RX, I2C0 SCL and PWM0 B.
- **NEOPIXEL2/GP18** - Digital I/O pin 18. Also SPI0 SCK, I2C1 SDA and PWM1 A.
- **NEOPIXEL3/GP19** - Digital I/O pin 19. Also SPI0 MOSI, I2C1 SCL and PWM1 B.
- **NEOPIXEL4/GP20** - Digital I/O pin 20. Also SPI0 MISO, UART1 TX, I2C0 SDA and PWM2 A.
- **NEOPIXEL5/GP21** - Digital I/O pin 21. Also SPI0 CS, UART1 RX, I2C0 SCL and PWM2 B.
- **NEOPIXEL6/GP22** - Digital I/O pin 22. Also SPI0 SCK, I2C1 SDA and PWM3 A.
- **NEOPIXEL7/GP23** - Digital I/O pin 23. Also SPI0 MOSI, I2C1 SCL and PWM3 B.

Some configurable pads are present to select logic voltage and direction on these pins, explained on the **8X I/O Header** page.

Pin Numbering

Starting with a clean slate for SCORPIO allowed fixing a minor annoyance compared to prior Feathers: on all of the normal Feather pins along the longer edges of the board, the silkscreen labels, CircuitPython pin names and Arduino digital pin numbers are now synchronized. CircuitPython's pin D4 is Arduino's pin 4 is "4" printed on the board. This should simplify translating code for this board between the two languages, plus the RP2040 SDK.

The pins along SCORPIO's short edge are an exception. These are labeled 0–7 on the board, named NEOPIXEL0–NEOPIXEL7 in CircuitPython, and are digital pins 16–23 in Arduino.

CircuitPython code written for the original non-SCORPIO Feather RP2040 should carry over to SCORPIO without changes. Non-SCORPIO Feather RP2040 Arduino code may need changes if it uses digitalRead() or digitalWrite() on pins 6–8 (now 4–6 on SCORPIO) or any special peripherals (I2C, PWM, etc.) on these pins or the SCK/MO/MI pins.

CircuitPython I2C, SPI and UART

Note that in CircuitPython, there is a board object each for I2C, SPI and UART that use the pins labeled on the Feather. You can use these objects to initialise these peripherals in your code.

- `board.STEMMA_I2C()` uses SCL/SDA
- `board.SPI()` uses SCK/MO/MI
- `board.UART()` uses RX/TX

GPIO Pins by Pin Functionality

Primary pin functions marked on the Feather RP2040 SCORPIO silk are bold.

I2C Pins

- I2C0 SCL: A3, D25, RX, D13, D9, D5, NEOPIXEL1/D17, NEOPIXEL5/D21
- I2C0 SDA: A2, D24, MISO, TX, D12, NEOPIXEL0/D16, NEOPIXEL4/D20
- I2C1 SCL: **SCL**, A1, MOSI, D11, NEOPIXEL3/D19, NEOPIXEL7/D23
- I2C1 SDA: **SDA**, A0, SCK, D10, D6, NEOPIXEL2/D18, NEOPIXEL6/D22

SPI Pins

- SPI0 SCK: D6, NEOPIXEL2/D18, NEOPIXEL6/D22
- SPI0 MOSI: SCL, NEOPIXEL3/D19, NEOPIXEL7/D23

- SPI0 MISO: TX, D4, NEOPIXEL0/D16, NEOPIXEL4/D20
- SPI0 CS: RX
- SPI1 SCK: **SCK**, A0, D10
- SPI1 MOSI: **MO**, A1, D11
- SPI1 MISO: **MI**, A2, D24, D12
- SPI1 CS: A3, D25, D13, D9

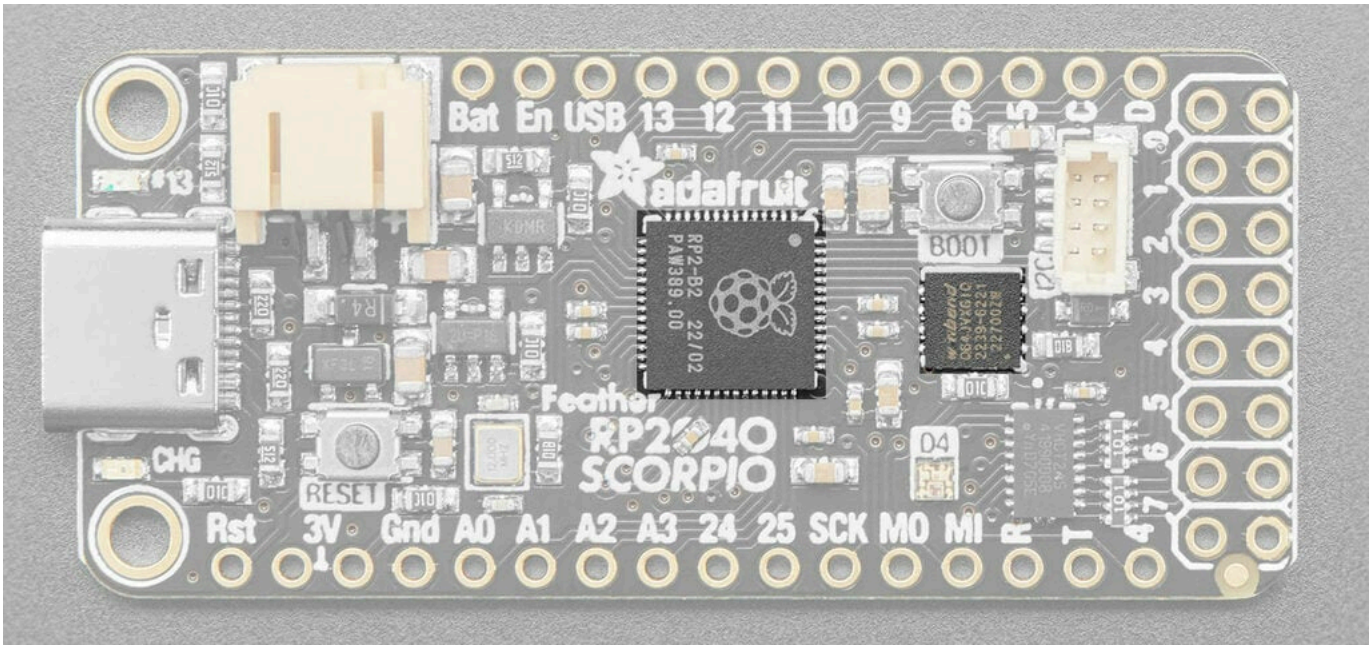
UART Pins

- UART0 TX: **T** (TX), A2, D12, NEOPIXEL0/D16
- UART0 RX: **R** (RX), A3, D13, NEOPIXEL1/D17
- UART1 TX: D24, MISO, D4, NEOPIXEL4/D20
- UART1 RX: D25, D9, D5, NEOPIXEL5/D21

PWM Pins

- PWM0 A: TX, NEOPIXEL0/D16
- PWM0 B: RX, NEOPIXEL1/D17
- PWM1 A: SDA, NEOPIXEL2/D18
- PWM1 B: SCL, NEOPIXEL3/D19
- PWM2 A: D4, NEOPIXEL4/D20
- PWM2 B: D5, NEOPIXEL5/D21
- PWM3 A: D6, NEOPIXEL6/D22
- PWM3 B: NEOPIXEL7/D23
- PWM4 A: D24, MISO
- PWM4 B: D25, D9
- PWM5 A: A0, D10
- PWM5 B: A1, D11
- PWM6 A: A2, D12
- PWM6 B: A3, D13
- PWM7 A: SCK
- PWM7 B: MOSI

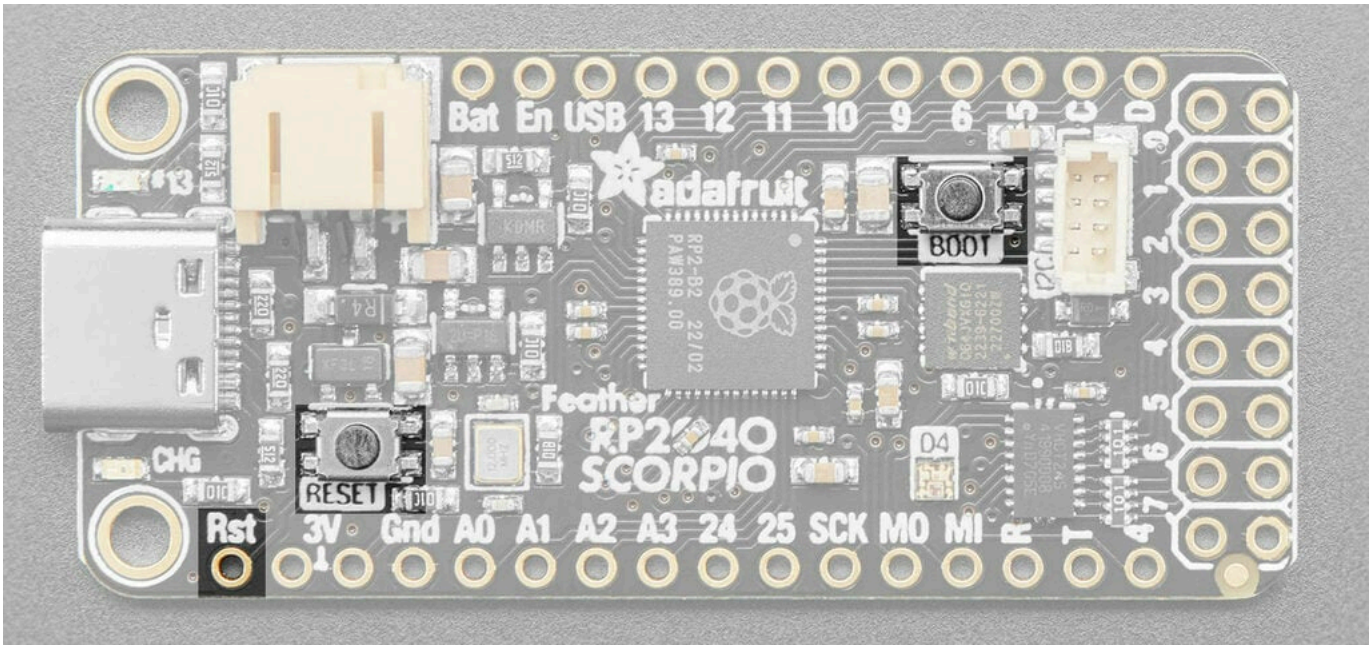
Microcontroller and Flash



The square towards the middle is the **RP2040 microcontroller**, the "brains" of the Feather RP2040 SCORPIO board.

The square near the BOOT button is the **QSPI Flash**. It is connected to 6 dedicated pins that are *not* brought out on the GPIO pads. This way you don't have to worry about the SPI flash colliding with other devices on the main SPI connection. Plus, QSPI is at least an *order of magnitude* faster than plain SPI flash.

Buttons and RST Pin



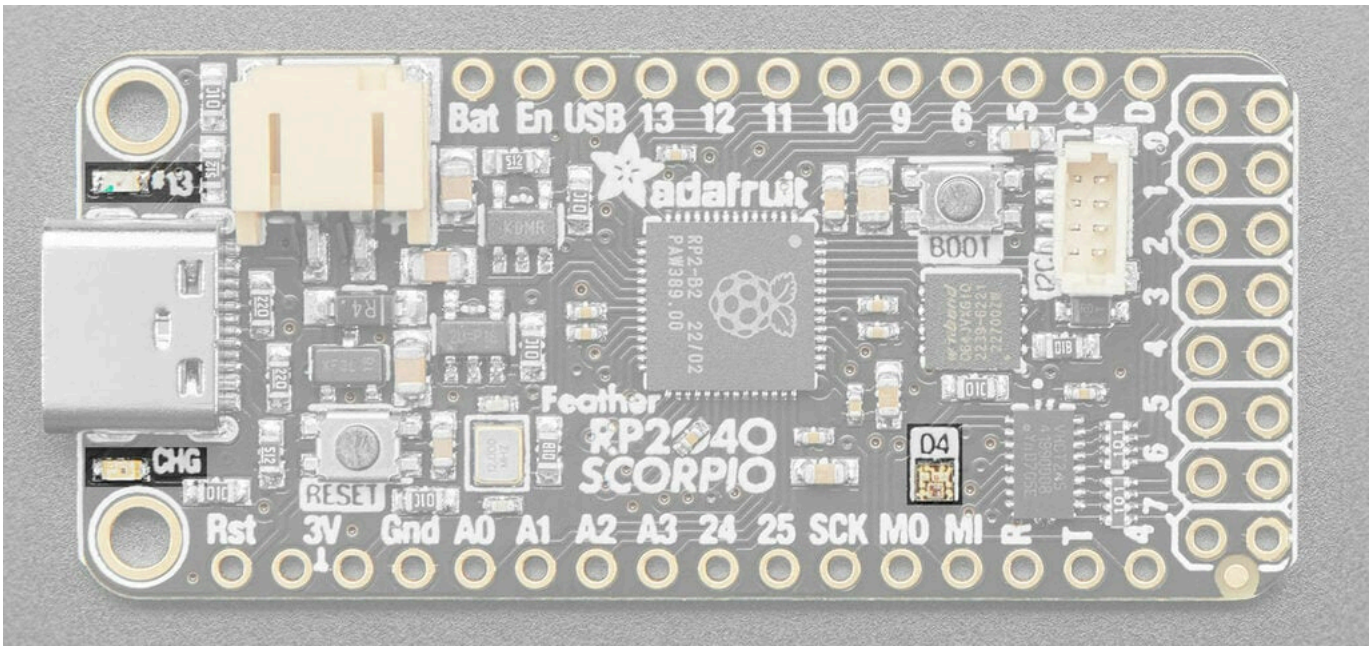
The Feather RP2040 SCORPIO has two buttons.

The **BOOT** button is used to enter the bootloader to install/update CircuitPython, and sometimes before uploading Arduino code. To enter the bootloader, press and hold BOOT and then power up the board (either by plugging it into USB or pressing RESET). Other times, when code is running on the board, BOOT (aka digital pin 7 in Arduino) can be used as an input if you don't want to wire up a button on another pin.

The **RESET button** restarts the board and helps enter the bootloader. You can click it to reset the board without unplugging the USB cable or battery.

The **RST pin** is can be used to reset the board. Momentarily tie to ground manually to reset the board.

LEDs

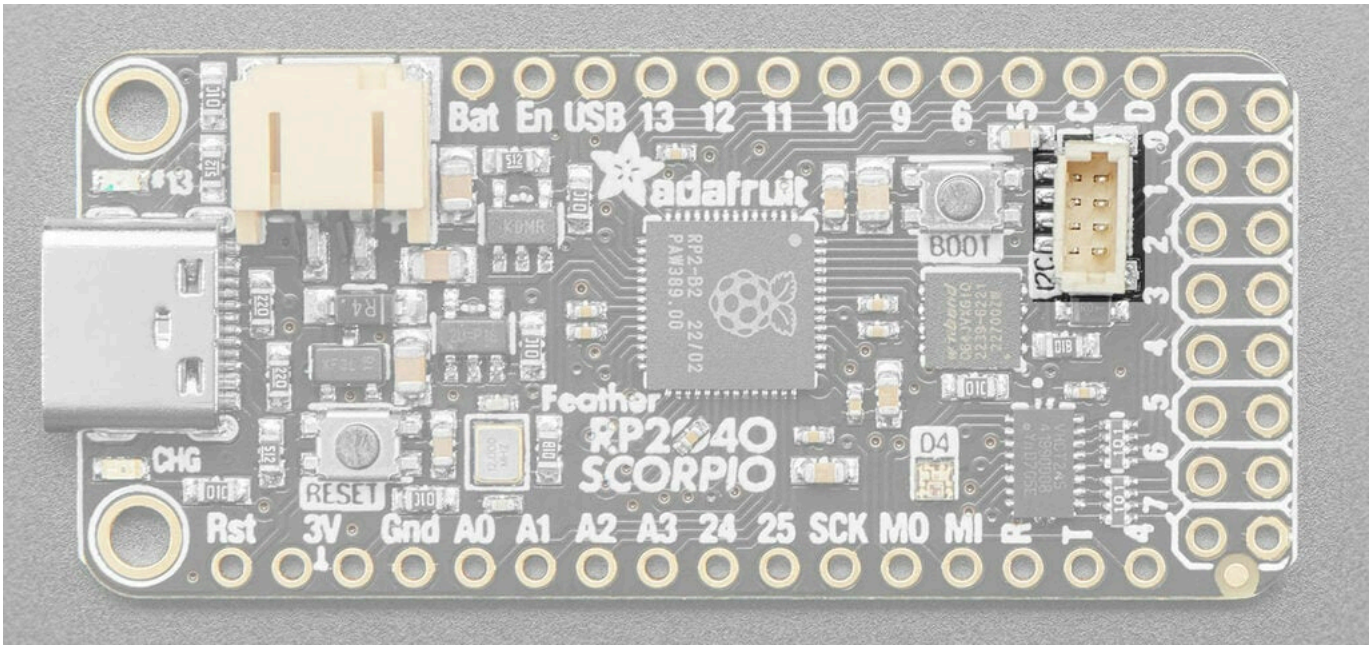


Above the pin labels for MO and MI is the status **NeoPixel LED**. In CircuitPython, the NeoPixel is `board.NEOPIXEL` and the library for it is here and in the bundle. In Arduino it's digital pin 4. The NeoPixel is powered by the 3.3V power supply but that hasn't shown to make a big difference in brightness or color. In CircuitPython, the LED is used to indicate the runtime status.

Below the USB C connector is the **CHG LED**. This indicates the charge status of a connected lipoly battery, if one is present and USB is connected. It is amber while charging, and green when fully charged. Note, it's normal for this LED to flicker when no battery is in place, that's the charge circuitry trying to detect whether a battery is there or not.

Above the USB C connector is the **D13 LED**. This little red LED is controllable in CircuitPython code using `board.LED`, or digital pin 13 in Arduino. Also, this LED will pulse when the board is in bootloader mode.

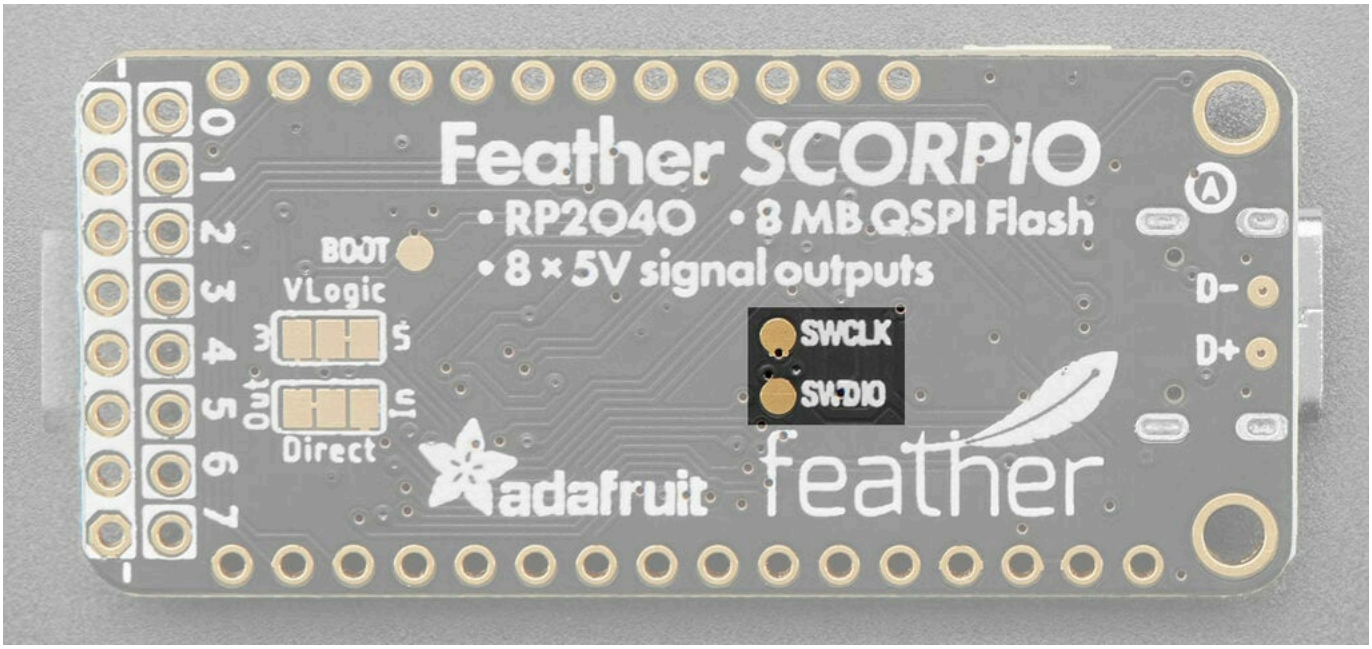
STEMMA QT



The Feather RP2040 comes with a built in **STEMMA QT connector**! This means you can connect up all sorts of I2C sensors and breakouts, no soldering required! This connector uses the SCL and SDA pins for I2C, which ends up being the RP2040's I2C1 peripheral. In CircuitPython, you can use the STEMMA connector with `board.SCL` and `board.SDA`, or `board.STEMMA_I2C()`. In Arduino it is `Wire` (*not* Wire1).

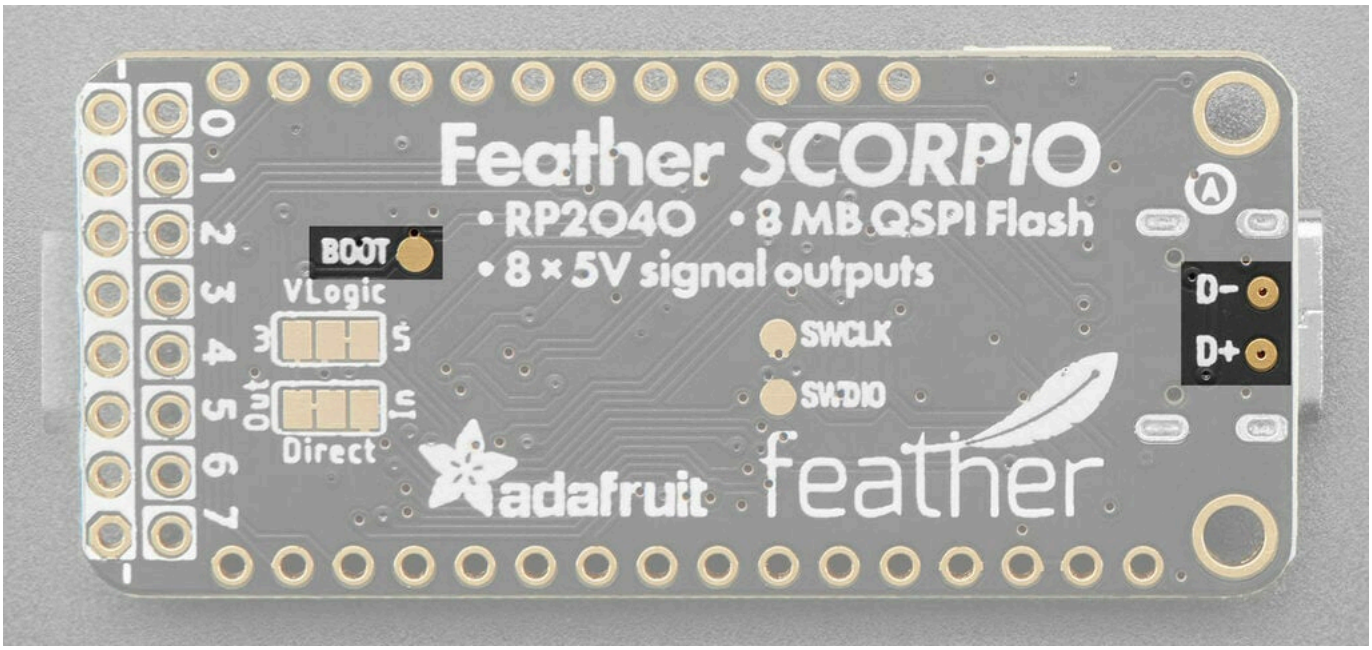
STEMMA QT / Qwiic JST SH 4-pin Cable - 100mm Long
This 4-wire cable is a little over 100mm / 4" long and fitted with JST-SH female 4-pin connectors on both ends. Compared with the chunkier JST-PH these are 1mm pitch instead of...
$0.95
In Stock
Add to Cart
Add to Wishlist
ledmesh
+ New List

Debug Interface

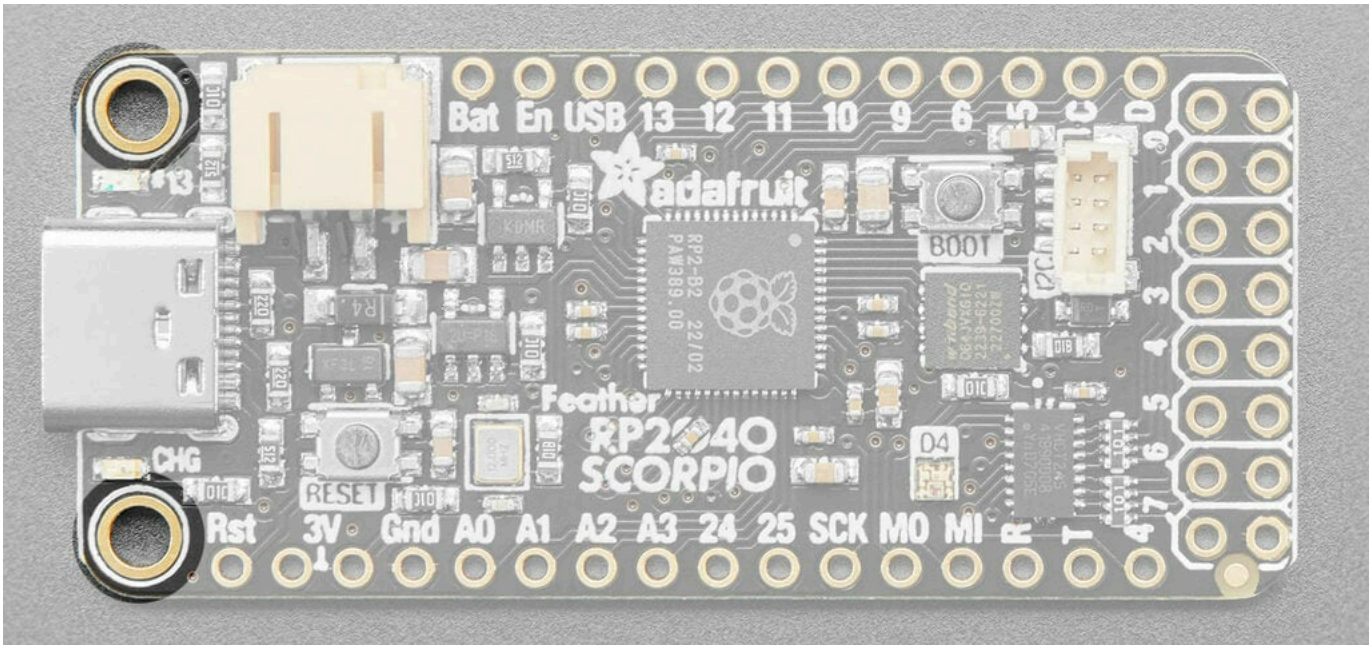


On the back of the board are pads for the **SWCLK** and **SWDIO** pins. They provide access to the internal Serial Wire Debug multi-drop bus, which provides debug access to both processors, and can be used to download code.

Miscellany



The back side of the board has pads to access the BOOT switch line and the USB D+ and D- pins. This is all esoteric but perhaps somebody has a use.



Feather RP2040 SCORPIO has only **two mounting holes**, at the USB end of the board. Adafruit Feather boards normally have *four* holes at the corners, but two were sacrificed here to accommodate the board-end SCORPIO header while maintaining standard Feather dimensions.

Page last edited March 08, 2024

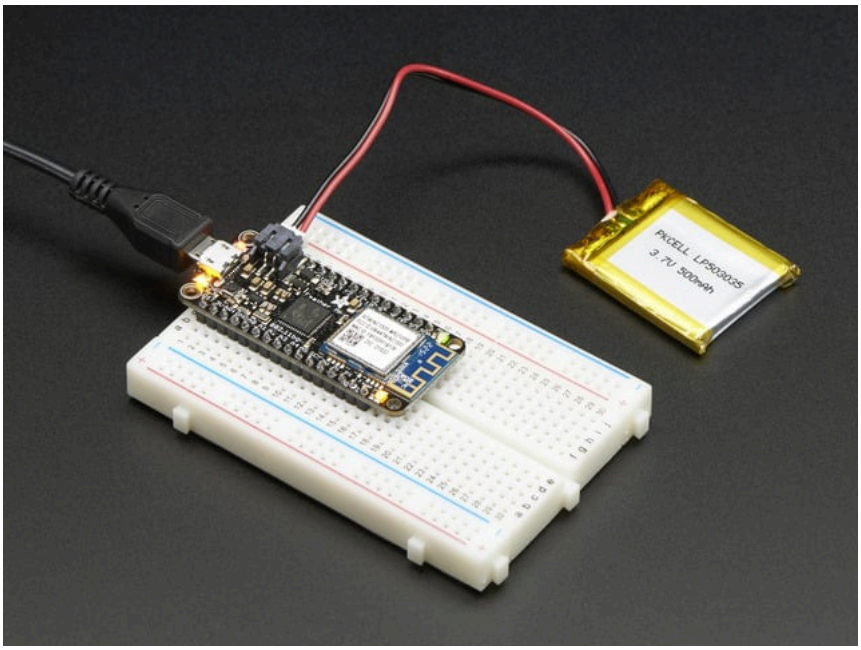
Text editor powered by tinymce.

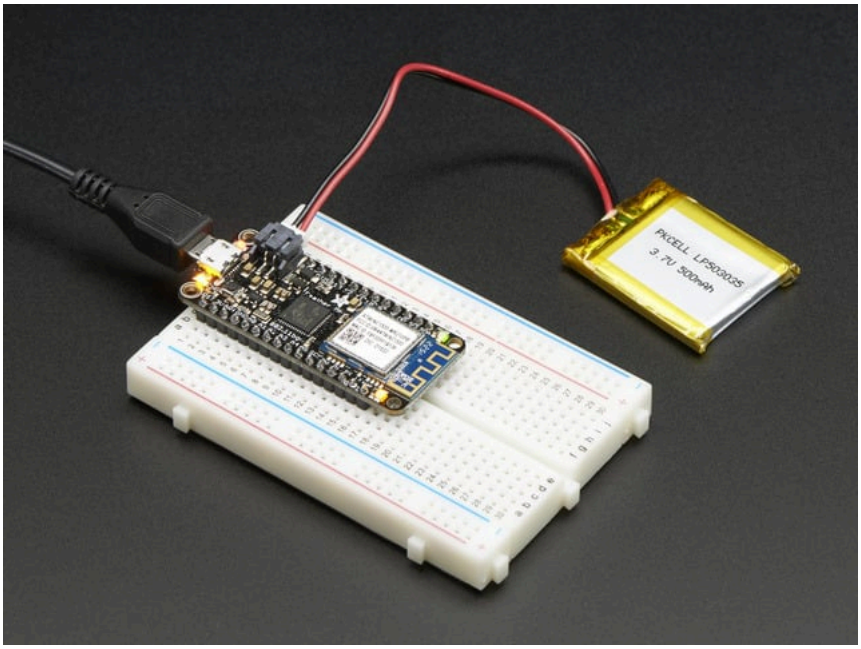
Assembly

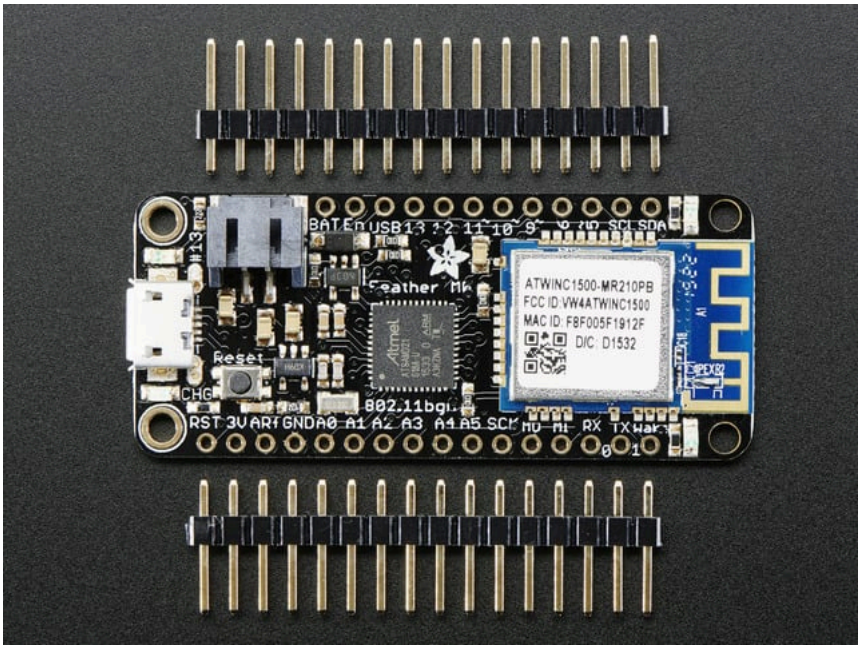
We ship Feathers fully tested but without headers attached - this gives you the most flexibility on choosing how to use and configure your Feather

Header Options!

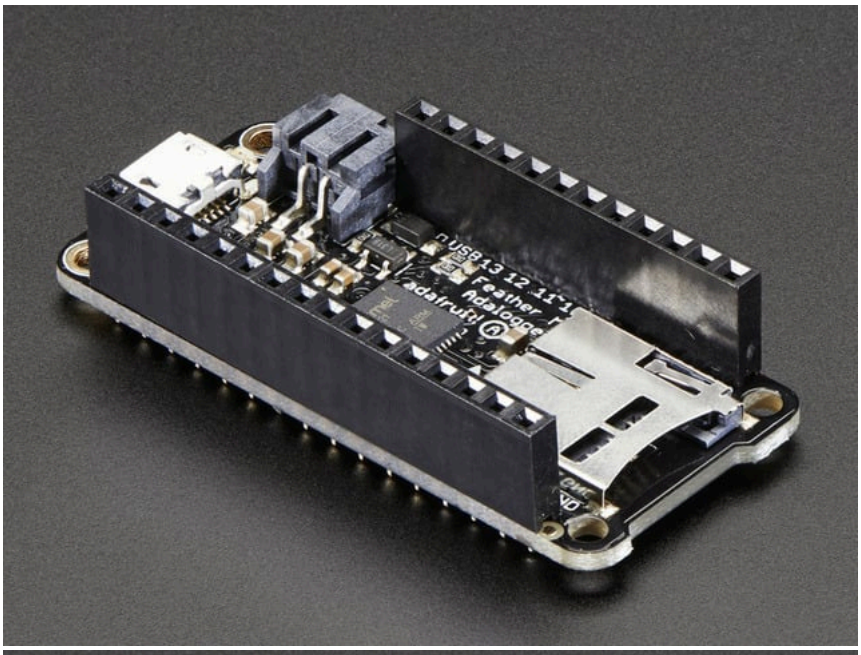
Before you go gung-ho on soldering, there's a few options to consider!

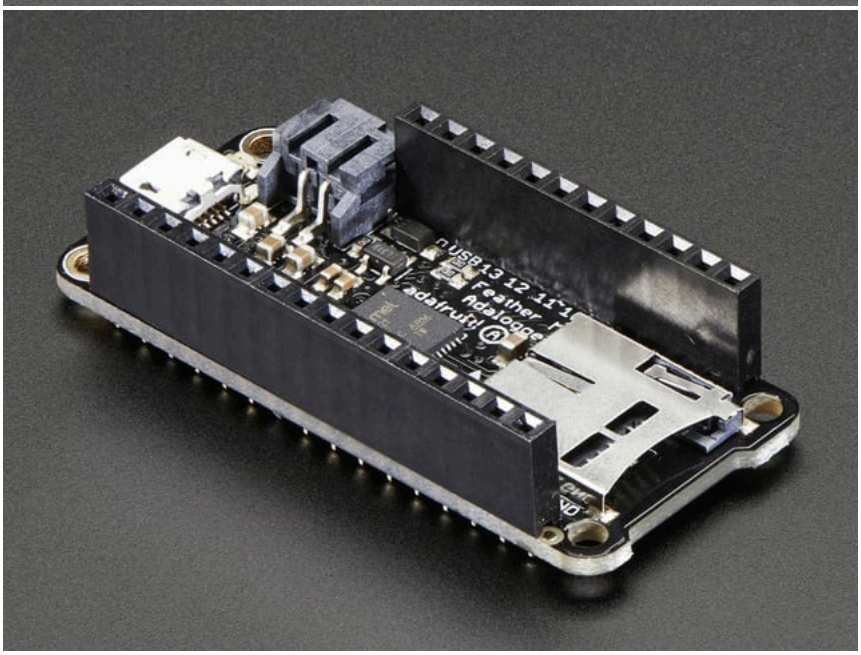


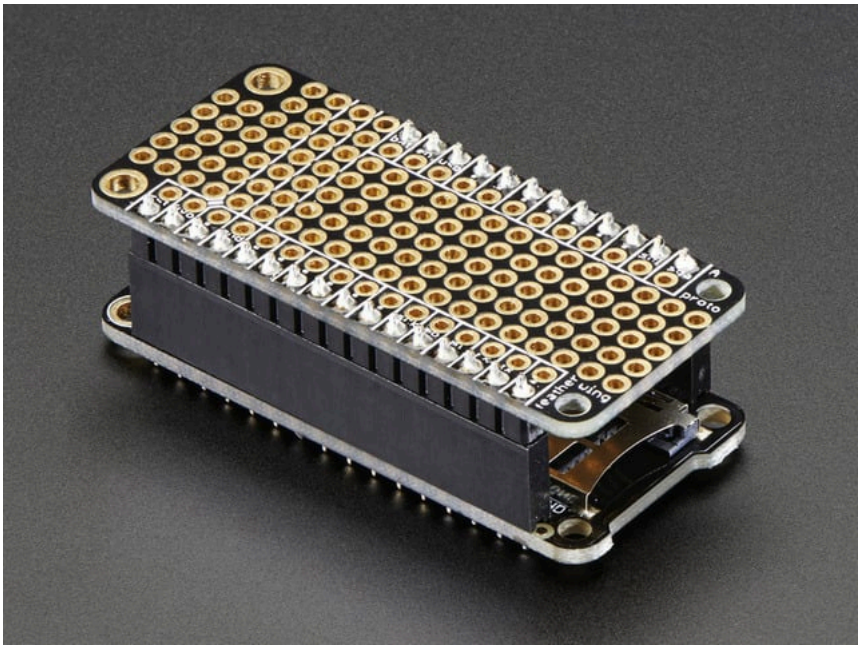


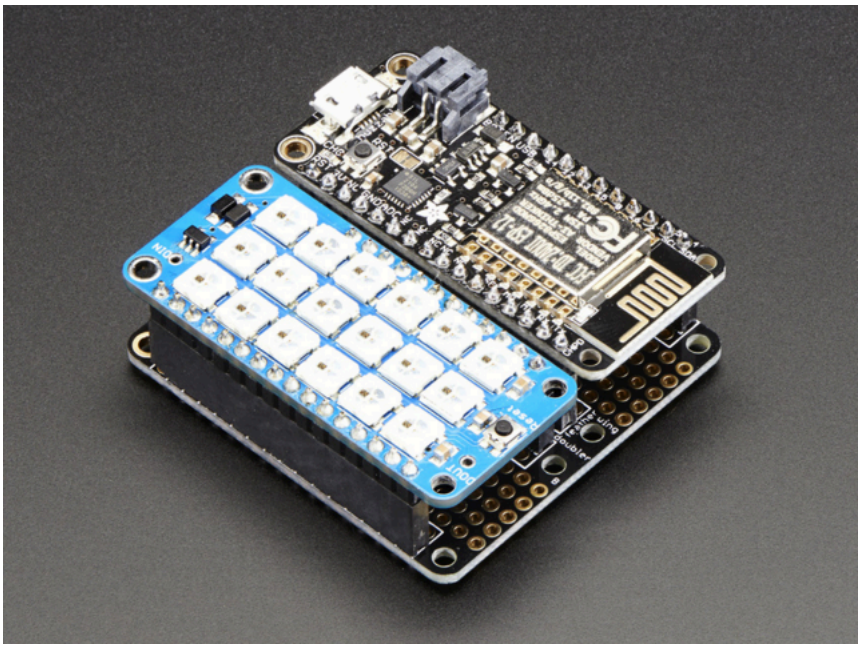


The first option is soldering in plain male headers, this lets you plug in the Feather into a solderless breadboard









Another option is to go with socket female headers. This won't let you plug the Feather into a breadboard but it will let you attach featherwings very easily

A few Feather boards require access to top-side components like buttons or connectors, making stacking impractical. Sometimes you can stack in the opposite order—FeatherWing underneath—or, if *both* Feather and Wing require top-side access, place the boards side-by-side with a FeatherWing Doubler or Tripler.

We also have 'slim' versions of the female headers, that are a little shorter and give a more compact shape

Finally, there's the "Stacking Header" option. This one is sort of the best-of-both-worlds. You get the ability to plug into a solderless breadboard *and* plug a featherwing on top. But its a little bulky

Soldering in Plain Headers

Prepare the header strip:

Cut the strip to length if necessary. It will be easier to solder if you insert it into a breadboard - **long pins down**

Add the breakout board:

Place the breakout board over the pins so that the short pins poke through the breakout pads

And Solder!

Be sure to solder all pins for reliable electrical contact.

(For tips on soldering, be sure to check out our Guide to Excellent Soldering).

Solder the other strip as well.

You're done! Check your solder joints visually and continue onto the next steps

Soldering on Female Header

Tape In Place

For sockets you'll want to tape them in place so when you flip over the board they don't fall out

Flip & Tack Solder

After flipping over, solder one or two points on each strip, to 'tack' the header in place

And Solder!

Be sure to solder all pins for reliable electrical contact.

(For tips on soldering, be sure to check out our Guide to Excellent Soldering).

You're done! Check your solder joints visually and continue onto the next steps

Page last edited March 08, 2024

Text editor powered by tinymce.

8X I/O Header

SCORPIO's secret sauce is the **8x2 header** at the end of the board, providing **eight sequential general-purpose inputs/outputs** (not shared by any other pins on the board) with a signal ground for each. Paired with the RP2040 chip's **PIO** peripheral, SCORPIO is uniquely suited to certain types of parallel logic projects…large-scale NeoPixel installations, logic analyzers, etc. The mounting holes normally present at this end of a Feather board were sacrificed to make space for this header, there to simplify hookups compared to breadboard wires or a separate stacked FeatherWing.

The "inner" row of eight pins (labeled 0–7) connect to **GPIO pins 16–23** through a **logic level shifter**, so this can safely interface with either **3.3V or 5V** devices.

The "outer" row of eight pins are all **ground** points. Typically these are used so SCORPIO and a separately-powered circuit have a common 0V point of reference, though in some situations it's okay to use these as a return when powering modest circuits (see "Powering SCORPIO NeoPixel Projects" page).

Input or Output? 5V or 3V?

From the factory, SCORPIO is configured for **5V output** on these eight pins (all other I/O pins work at 3.3V logic as usual). **Solder pads** on the back of the board allow reconfiguring this for **8 inputs** and/or **3.3V logic**.

Reconfiguring either setting requires **cutting a PCB trace** between adjacent pads using a hobby knife or file, then bridging the opposite pads with a dot of **solder**.

Looking at the back of SCORPIO with the text rightside-up…

The **VLogic** pads select **3.3V or 5V** operation. By default—**5V** operation—the right two pads are connected with a PCB trace. To switch to **3.3V** logic, **cut the PCB trace** between the center and right pad, then bridge the center and left pads with solder.

The **Direct** pads select **input or output** operation of the board-end header (other pins are unaffected). By default—**output** operation—the left two pads are connected with a PCB trace. To switch to **input** operation, **cut the PCB** trace between the center and left pad, then bridge the center and right pads with solder.

All 8 board-end pins are either **input or output**; there is **no mixed-direction** operation. If some application required *switching* all 8 between input and output modes, one could cut the Direct–Out PCB trace and solder a small wire from the center pad to an unused GPIO pin, then in code set that pin HIGH for output or LOW for input across the 8 pins. This is *not* built-in as it's *super esoteric* and every last RP2040 GPIO pin was already assigned!

SCORPIO's 8x2 header is *not* suitable for driving HUB75 RGB LED matrices; the arrangement of data and ground pins is altogether different. Use an RGB Matrix FeatherWing for that.

Coincidentally the 8 pins are sequential PWM channels (slice 0–3, outputs A & B), so this might find uses in multiple-servo projects.

Making 8x2 Connections

SCORPIO arrives with both straight and right-angle 8x2 pin headers, either of which can be soldered in place. Looking from the top side of the board (with all the components), the **upper row of pins are I/O**, the **lower row is all ground**. Let's talk about the SCORPIO side of an overall circuit. With a header installed, there are a few options for making connections here…

Individual **female jumper wires** (F–F or F–M) pressed into the header might be adequate for **quick prototyping**…but in real-world use, with vibration or just moving a project from one place to another, these can get accidentally **dislodged too easily**. Instead, for **robust** installations, consider…

A 16-pin IDC ribbon cable with 8x2 headers—the exact same type used for HUB75 RGB LED matrices—carries all 8 signals and grounds on alternate wires in a tidy bundle.

Our Large Dual Row Wire Housing Pack for DIY Jumper Cables contains five 8x2 housings. These can be paired with Premium Female/Female Raw Custom Jumper Wires as part of an overall wiring harness. One end of each wire securely clicks into place in the housing.

In rare situations that rely on other ground paths on the SCORPIO board, the Large Single Row Housing Pack for DIY Jumper Cables can be used, and connect to only the upper row of pins. **If in doubt, use the dual-row type.**

If you have the parts and tools it's easy enough to crimp a DIY version of these cables…but for onesy-twosey projects, the ribbon cable or jumper housing kits above are economical and the most time-consuming steps are already done.

Choose Your Own Adventure

So far we've talked about the SCORPIO side of the connection…but what's the other end of this cable supposed to connect to?

There is no one-size-fits-all answer. This is where you come in.

If controlling 8 concurrent strands of NeoPixels, for example: each of those data wires will connect to the "DIN" pin of the first pixel in each strand, and each of the grounds to the strands' grounds (the latter will usually be 2-way connections, as they usually also need to connect to power supply ground).

The "Powering SCORPIO NeoPixel Projects" section of this guide shows some example circuits.

Most often, you'll *cut the 16-pin cable in half* (whether a ribbon cable or DIY jumper cable), strip the ends of each wire, and construct some sort of **custom wiring harness**, distributing data and power to all of the NeoPixels in your project. In addition to the parts mentioned above, this involves **additional wire and some soldering and heat-shrink tubing** and perhaps some JST-SM connectors for the strips and a DC barrel jack for power (or using half of a DC extension cable).

Here's one such DIY wiring harness. This one's built from scrap bin parts so it doesn't visually match the Adafruit parts above, but the principles are exactly the same. Eight JST-SM plugs (left) connect to NeoPixel strips. Power and ground from each (in two groups of four, as multi-way splits are challenging to solder) lead to a DC barrel jack (top left, for a 5V power supply), and ground and data for each are joined to alternating pins of a ribbon cable (socket at right). Notice the inordinate amount of heat-shrink tube covering the soldered connections and also to keep bundles un-frayed and tidy:

Depending how power is distributed in your circuit, you might not *need* to make all 8 ground connections, but there is no harm in doing so and this usually keeps the wiring more orderly and secure: no loose ends and all 16 wires are populated and gripping the header.

Page last edited March 08, 2024

Text editor powered by tinymce.

Power Management

Battery + USB Power

We wanted to make our Feather boards easy to power both when connected to a computer as well as via battery.

There's **two ways to power** a Feather:

1. You can connect with a USB cable (just plug into the jack) and the Feather will regulate the 5V USB down to 3.3V.
2. You can also connect a 4.2/3.7V Lithium Polymer (LiPo/LiPoly) or Lithium Ion (LiIon) battery to the JST jack. This will let the Feather run on a rechargeable battery.

When the USB power is powered, it will automatically switch over to USB for power, as well as start charging the battery (if attached). This happens 'hot-swap' style so you can always keep the LiPoly connected as a 'backup' power that will only get used when USB power is lost.

The JST connector polarity is matched to Adafruit LiPoly batteries. Using wrong polarity batteries can destroy your Feather. Many customers try to save money by purchasing Lipoly batteries from Amazon only to find that they plug them in and the Feather is destroyed!

The above shows the USB-C jack (left), LiPoly JST jack (top left), as well as the 3.3V regulator and changeover diode (below the JST jack) and the LiPoly charging circuitry (to the right of the JST jack).

There's also a **CHG** LED next to the USB jack, which will light up while the battery is charging. This LED might also flicker if the battery is not connected, it's normal.

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

Power Supplies

You have a lot of power supply options here! We bring out the **BAT** pin, which is tied to the LiPoly JST connector, as well as **USB** which is the +5V from USB if connected. We also have the **3V** pin which has the output from the 3.3V regulator. We use a 500mA peak regulator. While you can get 500mA from it, you can't do it continuously from 5V as it will overheat the regulator.

Measuring Battery

If you're running off of a battery, chances are you wanna know what the voltage is at! That way you can tell when the battery needs recharging. LiPoly batteries are 'maxed out' at 4.2V and stick around 3.7V for much of the battery life, then slowly sink down to 3.2V or so before the protection circuitry cuts it off. By measuring the voltage you can quickly tell when you're heading below 3.7V.

Note that unlike other Feathers, we do not have an ADC connected to a battery monitor. Reason being there's only 4 ADCs and we didn't want to use one precious ADC for a battery monitor. You can create a resistor divider from BAT to GND with two 10K resistors and connect the middle to one of the ADC pins on a breadboard.

ENable pin

If you'd like to turn off the 3.3V regulator, you can do that with the **EN**(able) pin. Simply tie this pin to **Ground** and it will disable the 3V regulator. The **BAT** and **USB** pins will still be powered.

Alternative Power Options

The two primary ways for powering a feather are a 3.7/4.2V LiPo battery plugged into the JST port *or* a USB power cable.

If you need other ways to power the Feather, here's what we recommend:

- For permanent installations, a 5V 1A USB wall adapter will let you plug in a USB cable for reliable power
- For mobile use, where you don't want a LiPoly, use a USB battery pack!
- If you have a higher voltage power supply, use a 5V buck converter and wire it to a USB cable's 5V and GND input

Here's what you cannot do:

- **Do not use alkaline or NiMH batteries** and connect to the battery port - this will destroy the LiPoly charger
- **Do not use 7.4V RC batteries on the battery port** - this will destroy the board

The Feather *is not designed for external power supplies* - this is a design decision to make the board compact and low cost. It is not recommended, but technically possible:

- **Connect an external 3.3V power supply to the 3V and GND pins.** Not recommended, this may cause unexpected behavior and the **EN** pin will no longer work. Also this doesn't provide power on **BAT** or **USB** and some Feathers/Wings use those pins for high current usages. You may end up damaging your Feather.
- **Connect an external 5V power supply to the USB and GND pins.** Not recommended, this may cause unexpected behavior when plugging in the USB port because you will be back-powering the USB port, which *could* confuse or damage your computer.

Page last edited March 08, 2024

Text editor powered by tinymce.

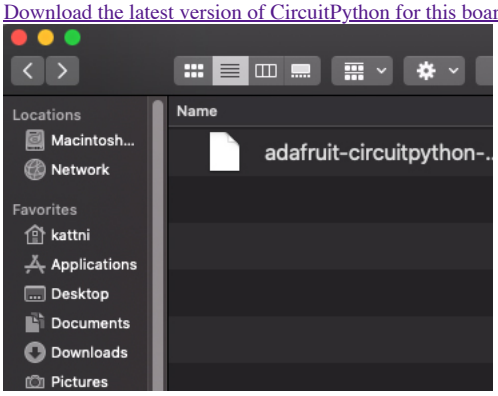
Install CircuitPython

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

CircuitPython Quickstart

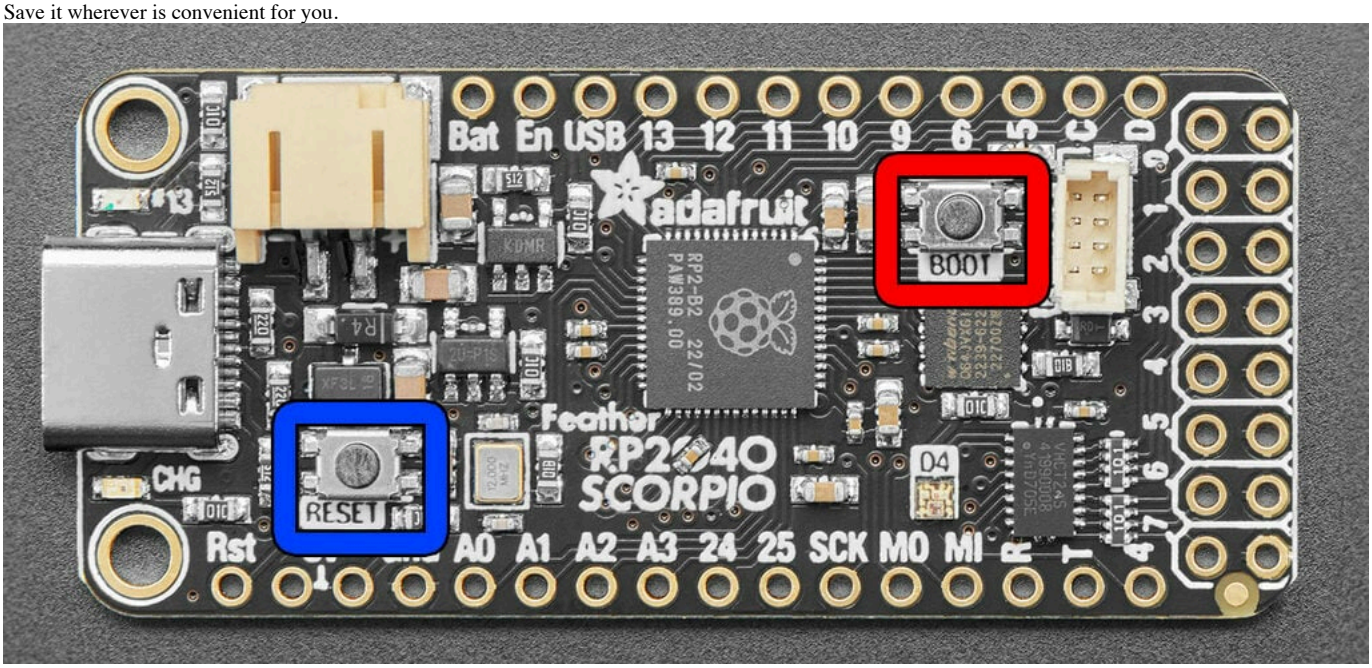
Follow this step-by-step to quickly get CircuitPython running on your board.

Download the latest version of CircuitPython for this board via circuitpython.org



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

Save it wherever is convenient for you.

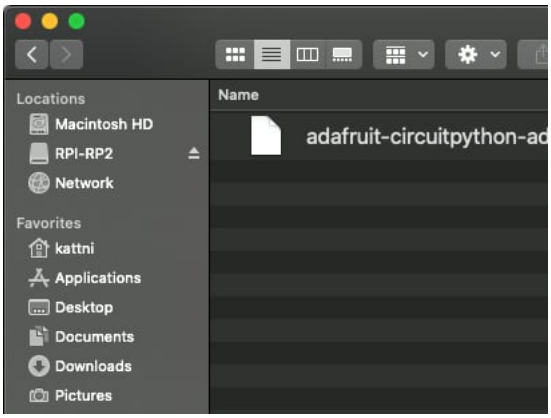


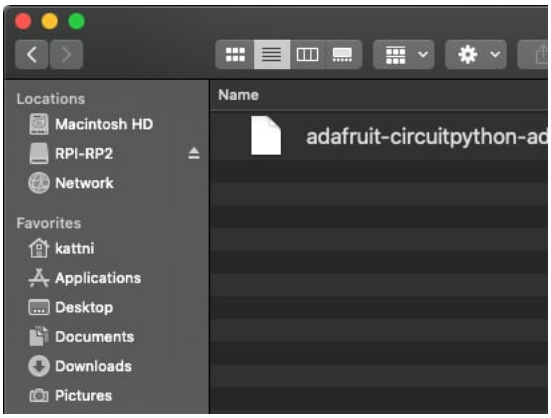
To enter the bootloader, hold down the **BOOT/BOOTSEL button** (highlighted in red above), and while continuing to hold it (don't let go!), press and release the **reset button** (highlighted in red or blue above). **Continue to hold the BOOT/BOOTSEL button until the RPI-RP2 drive appears!**

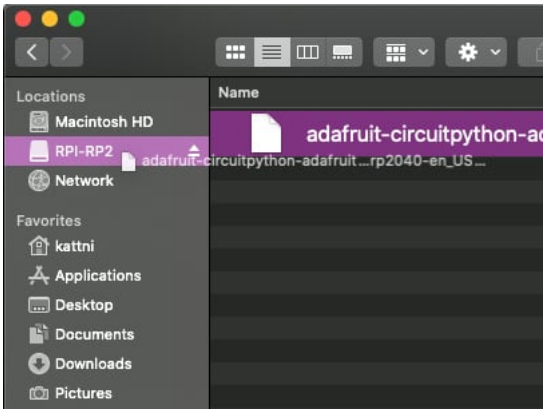
If the drive does not appear, release all the buttons, and then repeat the process above.

You can also start with your board unplugged from USB, press and hold the BOOTSEL button (highlighted in red above), continue to hold it while plugging it into USB, and wait for the drive to appear before releasing the button.

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

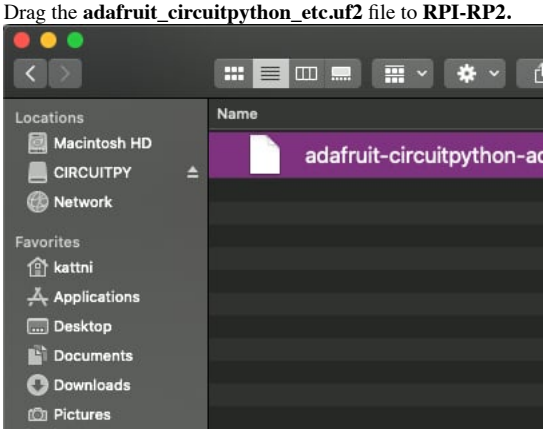






You will see a new disk drive appear called **RPI-RP2**.

Drag the **adafruit_circuitpython_etc.uf2** file to **RPI-RP2.**



The **RPI-RP2** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

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

Safe Mode

You want to edit your **code.py** or modify the files on your **CIRCUITPY** drive, but find that you can't. Perhaps your board has gotten into a state where **CIRCUITPY** is read-only. You may have turned off the **CIRCUITPY** drive altogether. Whatever the reason, 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

To enter safe mode when using CircuitPython, plug in your board or hit reset (highlighted in red above). Immediately after the board starts up or resets, it waits 1000ms. On some boards, the onboard status LED (highlighted in green above) will blink yellow during that time. If you press reset during that 1000ms, 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

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

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

Copy Text

```
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*.

Flash Resetting UF2

If your board ever gets into a really *weird* state and CIRCUITPY doesn't show up as a disk drive after installing CircuitPython, try loading this 'nuke' UF2 to RPI-RP2. which will do a 'deep clean' on your Flash Memory. **You will lose all the files on the board**, but at least you'll be able to revive it! After loading this UF2, follow the steps above to re-install CircuitPython.
Download flash erasing "nuke" UF2

Page last edited March 08, 2024

Text editor powered by tinymce.

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.

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...

Page last edited March 08, 2024

Text editor powered by tinymce.

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 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 14.1 introduced a bug that delays writes to small drives such as CIRCUITPY drives. This caused errors when saving files to CIRCUITPY. There is a workaround. The bug was fixed in Sonoma 14.4, but at the cost of greatly slowed writes to drives 1GB or smaller.

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:

Download File
Copy Code

```
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.

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 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 page of every board guide to get your board up and running again.

If you are having trouble saving code on Windows 10, try including this code snippet at the top of code.py:

Download File
Copy Code

```
import supervisor
supervisor.runtime.autoreload = False
```

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:

Download File
Copy Code

```
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:

Download File
Copy Code

```
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.

Page last edited March 08, 2024

Text editor powered by tinymce.

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.

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:

Copy Text

```
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:

Copy Text

```
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 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.

MacOS has Terminal built in, though there are other options available for download. Check the Advanced Serial Console on Mac page for more details.

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.

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

Page last edited March 08, 2024

Text editor powered by tinymce.

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:

Download File
Copy Code

```
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.

Download File
Copy Code

```
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`

Download File
Copy Code

```
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!

Page last edited March 08, 2024

Text editor powered by tinymce.

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.

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.

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.

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

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

Page last edited March 08, 2024

Text editor powered by tinymce.

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:

Download File
Copy Code

```
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.**

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? 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.

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:

Download File
Copy Code

```
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:

Download File
Copy Code

```
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:

Download File
Copy Code

```
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:

Download Project Bundle
Copy Code

```
# 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)
```

View on GitHub

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.

Microcontroller pin names for QT Py SAMD21.

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 and the Python-like modules included here. 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, 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.

Download File
Copy Code

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

help("modules") results for QT Py

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

Page last edited March 08, 2024

Text editor powered by tinymce.

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 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

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

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.

Download File
Copy Code

```
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 on The REPL page 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.

Download File
Copy Code

```
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.

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 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. 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 for a full list of functionality

Page last edited March 08, 2024

Text editor powered by tinymce.

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
CPC = Circuit Playground Classic (does not run CircuitPython)
CPX = Circuit Playground Express
CPB = Circuit Playground Bluefruit

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 8.x or earlier. Where can I find compatible libraries?

We are no longer building or supporting the CircuitPython 8.x or earlier library bundles. We highly encourage you to update CircuitPython to the latest version and use the current version of the libraries. However, if for some reason you cannot update, here are the last available library bundles for older versions:

- 2.x bundle
- 3.x bundle
- 4.x bundle
- 5.x bundle
- 6.x bundle
- 7.x bundle
- 8.x bundle

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

nRF52840, nRF52833, and **as of CircuitPython 9.1.0**, ESP32, ESP32-C3, and ESP32-S3 boards (with 8MB) 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.

Most Espressif boards with only 4MB of flash do not have enough room to include BLE in CircuitPython 9. Check the Module Support Matrix to see if your board has support for `_bleio`. CircuitPython 10 is planned to support _bleio on Espressif boards with 4MB flash.

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 or other NINA-FW-based co-processors. Some boards have this coprocessor on board, such as the PyPortal. 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 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 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!

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 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. 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.

On macOS and Linux, after you download **mpy-cross**, you must make the the file executable by doing `chmod +x name-of-the-mpy-cross-executable`.

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!

As of CircuitPython 8.x we have started to support ESP32 and ESP32-C3 and have added a WiFi workflow for wireless coding!

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.

Page last edited March 08, 2024

Text editor powered by tinymce.

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. Everything you need to get started with your new microcontroller and beyond is available. You can do things like download CircuitPython for your microcontroller or download the latest CircuitPython Library bundle, or check out which single board computers support Blinka. 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.

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. 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 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 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, and the CircuitPython libraries. If you need an account, visit https://github.com/ and sign up.

If you're new to GitHub or programming in general, there are great opportunities for you. For the CircuitPython core, head over to the CircuitPython repository on GitHub, click on "Issues", and you'll find a list that includes issues labeled "good first issue". For the libraries, head over to the Contributing page Issues list, and use the drop down menu to search for "good first issue". 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.

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. 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 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 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 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 page!

Page last edited March 08, 2024

Text editor powered by tinymce.

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 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.

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. 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!

Page last edited March 08, 2024

Text editor powered by tinymce.

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.

Comando screen en MacOS usando ruta de tarjeta de ejemplo

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!

Page last edited March 08, 2024

Text editor powered by tinymce.

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..**

You need to download the CircuitPython Library Bundle that matches your version of CircuitPython. **Please update CircuitPython and then download the latest bundle**.

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 and use the current version of the libraries. However, if for some reason you cannot update, links to the previous bundles are available in the FAQ.

macOS Sonoma before 14.4: Errors Writing to CIRCUITPY
macOS 14.4 - 15.1: Slow Writes to CIRCUITPY

macOS Sonoma before 14.4 took many seconds to complete writes to small FAT drives, 8MB or smaller. This causes errors when writing to CIRCUITPY. The best solution was to remount the CIRCUITPY drive after it is automatically mounted. Or consider downgrading back to Ventura if that works for you. This problem was tracked in CircuitPython GitHub issue 8449.

Below is a shell script to do this remount conveniently (courtesy @czei in GitHub). 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 and versions of macOS before Sequoia 15.2 did not have the problem above, but did take an inordinately long time to write to FAT drives of size 1GB or less (40 times longer than 2GB drives). As of macOS 15.2, writes are no longer very slow. This problem was tracked in CircuitPython GitHub issue 8918.

Download File
Copy Code

```
#!/bin/sh
#
# This works around bug where, by default,
# macOS 14.x before 14.4 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:

Copy Text

```
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.

Bootloader (*boardname*BOOT) Drive Not Present

You may have a different board.

Only Adafruit Express boards and the SAMD21 non-Express boards ship with the UF2 bootloader installed. The Feather M0 Basic, Feather M0 Adalogger, and similar boards use a regular Arduino-compatible bootloader, which does not show a ***boardname*BOOT** drive.

MakeCode

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

macOS

DriveDx and its accompanything **SAT SMART Driver** can interfere with seeing the BOOT drive. See this forum post for how to fix the problem.

Windows 10 or later

Did you install the Adafruit Windows Drivers package by mistake, or did you upgrade to Windows 10 or later with the driver package installed? You don't need to install this package on Windows 10 or 11for 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

Windows 7 and 8.1 have reached end of life. It is recommended that you upgrade to Windows 10 or 11 if possible. Drivers are available for some older CircuitPython boards, but there are no plans to release drivers for newer boards.

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 newer boards. The boards work fine on Windows 10 and later.

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

Let us know in the Adafruit support forums or on the Adafruit Discord if this does not work for you!

Windows Explorer Locks Up When Accessing *boardname*BOOT Drive

On Windows, several third-party programs that can cause issues. The symptom is that you try to access the ***boardname*BOOT** 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 *boardname*BOOT 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 ***boardname*BOOT** 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 and the BOOT drive to reappear. It is not clear what causes this behavior.

Samsung Magician can cause CIRCUITPY to disappear (reported here and here).

Device Errors or Problems on Windows

Windows can become confused about USB device installations. Try cleaning up your USB devices. Use Uwe Sieber's Device Cleanup Tool (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!

Download File
Copy Code

```
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 "Acronis Managed Machine Service Mini".

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

Download File
Copy Code

```
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.

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 ***boardname*BOOT** 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 0.7 seconds. On some boards, the onboard status LED (highlighted in green above) will turn solid yellow during this time. If you press reset during that 0.7 seconds, 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.

Copy Text

```
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 to do this.

1. Connect to the CircuitPython REPL using Mu or a terminal program.
2. Type the following into the REPL:

Copy Text

```
>>> import storage
>>> 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
Feather M0 Express
Feather M4 Express
Metro M0 Express
Metro M4 Express QSPI Eraser
Trellis M4 Express (QSPI)
Grand Central M4 Express (QSPI)
PyPortal M4 Express (QSPI)
Circuit Playground Bluefruit (QSPI)
Monster M4SK (QSPI)
PyBadge/PyGamer QSPI Eraser.UF2
CLUE_Flash_Erase.UF2
Matrix_Portal_M4_(QSPI).UF2
RP2040 boards (flash_nuke.uf2)

2. Double-click the reset button on the board to bring up the ***boardname*BOOT** drive.
3. Drag the erase **.uf2** file to the ***boardname*BOOT** drive.
4. The 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 ***boardname*BOOT** drive.
7. Drag the appropriate latest release of CircuitPython **.uf2** file to the ***boardname*BOOT** drive.

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

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

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

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

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

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

Download File
Copy Code

```
ls -l /Volumes
```

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

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

Copy Text

```
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:

Copy Text

```
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:

Copy Text

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

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

Copy Text

```
# 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.

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.

For more information on safe mode and how to enter safe mode, see the Safe Mode section on this page.

Page last edited March 08, 2024

Text editor powered by tinymce.

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!

Page last edited March 08, 2024

Text editor powered by tinymce.

Blink

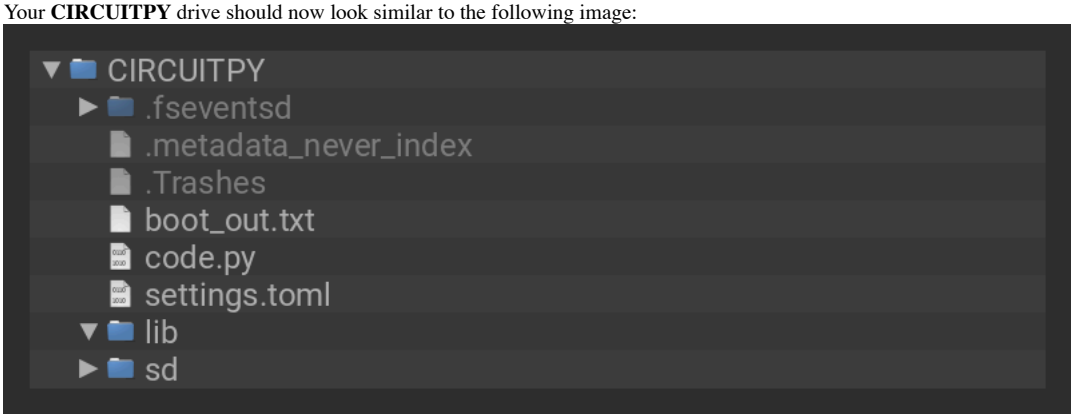
In learning any programming language, you often begin with some sort of `Hello, World!` program. In CircuitPython, Hello, World! is blinking an LED. Blink is one of the simplest programs in CircuitPython. It involves three built-in modules, two lines of set up, and a short loop. Despite its simplicity,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

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:



Download Project Bundle
Copy Code

```
# 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)
```

View on GitHub

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!

Page last edited March 08, 2024

Text editor powered by tinymce.

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

Controlling the LED with a Button

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!

Page last edited March 08, 2024

Text editor powered by tinymce.

Using adafruit_neopxl8

For boards like Feather RP2040 SCORPIO, use the `adafruit_neopxl8` library to control the LEDs in CircuitPython, rather than the standard `neopixel` library.

The library works much like the standard neopixel library, but it can control from 1 to 8 LED strands. Just import it, and then create an instance of the NeoPXL8 class:

Download File
Copy Code

```
import board
from adafruit_neopxl8 import NeoPxl8

# Customize for your strands here
num_strands = 8
strand_length = 30
first_led_pin = board.NEOPIXEL0

num_pixels = num_strands * strand_length
pixels = NeoPxl8(
    first_led_pin,
    num_pixels,
    num_strands=num_strands,
    auto_write=False,
    brightness=0.50,
)
```

"Background" writing

The `adafruit_neopxl8` library is fast for two reasons. Not only does it send the data for up to 8 strands at once, it also sends the data "in the background", while your CircuitPython code can do things like check for button presses & update animations. With care in coding the rest of your CircuitPython program, you can get some really high update rates for your LED displays.

About pixel numbering

In CircuitPython's NeoPXL8, the pixels from all the strands are interleaved. In other words, say you have a 8 strands of 30 LEDs. The first 30 pixels are the pixels in the first strand; the next 30 pixels are the second pixel from each strand; and so on. Or, to put it another way, the pixels of the first strand are the numbers in `range(0,30)`, the pixels of the second strand are `range(30, 60)`, etc.

You can use the PixelMap feature of the LED Animation Library to build an object that lets you address each strand individually:

Download File
Copy Code

```
from adafruit_led_animation.helper import PixelMap

def strand(n):
    return PixelMap(
        pixels,
        range(n * strand_length, (n + 1) * strand_length),
        individual_pixels=True,
    )

strands = [strand(i) for i in range(num_strands)]
```

Now, you can refer to `strands[S][P]` to mean `strand #S, pixel #P`, or use `strand[S]` to refer to the whole strand—for instance, to call the `fill` method or to use with an LED animation. Note that you want to call the strand function just once per strand and store the result. You can store them all in a list, like this example does; or you could give them names—for instance, if you you have 4 strands on the 4 sides of a window you might want to write `top=strand(0)` and `bottom=strand(1)`, for instance. Any of the other pixel mapping helpers from the LED Animation Library work equally well with NeoPXL8.

Using Animations with NeoPXL8

Download File
Copy Code

```
import rainbowio
from adafruit_led_animation.animation.comet import Comet
from adafruit_led_animation.group import AnimationGroup

# For each strand, create a comet animation of a different color
animations = [
```

```
    Comet(strand, 0.02, rainbowio.colorwheel(3 * 32 * i), ring=True)
    for i, strand in enumerate(strands)
]

# Group them so we can run them all at once
animations = AnimationGroup(*animations)

while True:
    animations.animate()
```

Here's the full example code. Use the 'download project bundle' link to get all the files you need to run it:

Download Project Bundle
Copy Code

```
# SPDX-FileCopyrightText: 2022 Jeff Epler
#
# SPDX-License-Identifier: Unlicense

import adafruit_ticks
import board
import rainbowio
from adafruit_led_animation.animation.comet import Comet
from adafruit_led_animation.group import AnimationGroup
from adafruit_led_animation.helper import PixelMap

from adafruit_neopxl8 import NeoPxl8

# Customize for your strands here
num_strands = 8
strand_length = 30
first_led_pin = board.NEOPIXEL0

num_pixels = num_strands * strand_length

# Make the object to control the pixels
pixels = NeoPxl8(
    first_led_pin,
    num_pixels,
    num_strands=num_strands,
    auto_write=False,
    brightness=0.50,
)


def strand(n):
    return PixelMap(
        pixels,
        range(n * strand_length, (n + 1) * strand_length),
        individual_pixels=True,
    )


# Create the 8 virtual strands
strands = [strand(i) for i in range(num_strands)]

# For each strand, create a comet animation of a different color
animations = [
    Comet(strand, 0.02, rainbowio.colorwheel(3 * 32 * i), ring=True)
    for i, strand in enumerate(strands)
]

# Advance the animations by varying amounts so that they become staggered
for i, animation in enumerate(animations):
    animation._tail_start = 30 * 5 * i // 8

# Group them so we can run them all at once
animations = AnimationGroup(*animations)
```

```
# Run the animations and report on the speed in frame per secodn
t0 = adafruit_ticks.ticks_ms()
frame_count = 0
while True:
    animations.animate()
    frame_count += 1
    t1 = adafruit_ticks.ticks_ms()
    dt = adafruit_ticks.ticks_diff(t1, t0)
    if dt > 1000:
        print(f"{frame_count * 1000/dt:.1f}fps")
        t0 = t1
        frame_count = 0
```

View on GitHub

Page last edited March 08, 2024

Text editor powered by tinymce.

Arduino IDE Setup

The **Arduino Philhower core** provides support for RP2040 microcontroller boards. This page covers getting your Arduino IDE set up to include your board.

Arduino IDE Download

The first thing you will need to do is to download the latest release of the Arduino IDE. The Philhower core requires **version 1.8** or higher.

Arduino IDE Download

Download and install it to your computer.

Once installed, open the Arduino IDE.

Adding the Philhower Board Manager URL

In the Arduino IDE, and navigate to the **Preferences** window. You can access it through **File > Preferences** on Windows or Linux, or **Arduino > Preferences** 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://github.com/earlephilhower/arduino-pico/releases/download/global/package_rp2040_index.json
```

Add the URL to the the **Additional Boards Manager URLs** field (highlighted in red below).

Click **OK** to save and close **Preferences**.

Add Board Support Package

In the Arduino IDE, click on **Tools > Board > Boards Manager**. If you have previously selected a board, the **Board** menu item may have a board name after it.

In the **Boards Manager**, search for RP2040. Scroll down to the **Raspberry Pi Pico/RP2040 by Earle F Philhower, III** entry. Click **Install** to install it.

Installing a new board package can take a few minutes. Don't click Cancel!

Once installation is complete, click **Close** to close the Boards Manager.

Choose Your Board

In the **Tools > Boards** menu, you should now see **Raspberry Pi RP2040 Boards** (possibly followed by a version number).

Navigate to the **Raspberry Pi RP2040 Boards** menu. You will see the available boards listed.

From the **Raspberry Pi RP2040 Boards** rollover menu, select **Adafruit Feather RP2040 SCORPIO** (*not* Feather RP2040 above it…*must* choose the SCORPIO variant to correctly use this board).

If there is no serial Port available in the dropdown, or an invalid one appears - don't worry about it! The RP2040 does not actually use a serial port to upload, so its OK if it does not appear if in manual bootload mode. You will see a serial port appear after uploading your first sketch.

Now you're ready to begin using Arduino with your RP2040 board!

Page last edited March 08, 2024

Text editor powered by tinymce.

Arduino Usage

Now that you've set up the Arduino IDE with the Philhower RP2040 Arduino core, you're ready to start using Arduino with your RP2040.

RP2040 Arduino Pins

There is no pin remapping for Arduino on the RP2040. Therefore, the pin names on the top of the board are **not** the pin names used for Arduino. The Arduino pin names are the RP2040 GPIO pin names.

To find the Arduino pin name, check the PrettyPins diagram found on the Pinouts page. Each GPIO pin in the diagram has a **GPIOx** pin name listed, where **x** is the pin number. The Arduino pin name is the number following **GPIO**. For example, **GPIO1** would be Arduino pin `1`.

Most pin numbers are also indicated on the board silkscreen, but the board-end **8X I/O header** is a special exception: labeled 0–7, these are Arduino pins `16` through `23`.

Choose Your Board

Navigate to the **Tools > Boards > Raspberry Pi RP2040 Boards** menu. The Raspberry PI RP2040 Boards menu name may be followed by a version number.

Once the menu has expanded, you will see three different versions the Feather RP2040 SCORPIO available: **Feather RP2040 SCORPIO**, **Feather RP2040 SCORPIO (Picoprobe)**, and **Feather RP2040 SCORPIO (pico-debug)**. Unless you are specifically familiar with the other two, always choose **Feather RP2040 SCORPIO**.

Choose **Feather RP2040 SCORPIO** from the menu (*not* Feather RP2040 above it…*must* choose the SCORPIO variant to correctly use this board).

Load the Blink Sketch

Begin by plugging in your board to your computer, and wait a moment for it to be recognised by the OS. It will create a COM/serial port that you can now select from the **Tools > Port** menu dropdown.

Open the Blink sketch by clicking through **File > Examples > 01.Basics > Blink**.

Click Upload. A successful upload will result in text similar to the following.

Once complete, the little red LED will begin blinking once every second! Try changing up the `delay()` timing to change the rate at which the LED blinks.

Manually Enter the Bootloader

If you get into a state with the bootloader where you can no longer upload a sketch, or you have uploaded code that crashes and doesn't auto-reboot into the bootloader, you may have to manually enter the bootloader.

To enter the bootloader, hold down the **BOOT button**, and while continuing to hold it (don't let go!), press and release the **reset button**. **Continue to hold the BOOT button until the RPI-RP2 drive appears!**

Once the RPI-RP2 drive shows up, your board is in bootloader mode. There will not be a port available in bootloader mode, this is expected.

Once you see RPI-RP2 drive, make sure you are no longer holding down any buttons (reset or boot0 button).

Now, click Upload on your sketch to try again.

Page last edited March 08, 2024

Text editor powered by tinymce.

Blink

The first and most basic program you can upload to your Arduino is the classic Blink sketch. This takes something on the board and makes it, well, blink! On and off. It's a great way to make sure everything is working and you're uploading your sketch to the right board and right configuration.

When all else fails, you can always come back to Blink!

Pre-Flight Check: Get Arduino IDE & Hardware Set Up

This lesson assumes you have Arduino IDE set up. This is a generalized checklist, some elements may not apply to your hardware. If you haven't yet, check the previous steps in the guide to make sure you:

- **Install the very latest Arduino IDE for Desktop** (not all boards are supported by the Web IDE so we don't recommend it).
- **Install any board support packages (BSP) required for your hardware.** Some boards are built in defaults on the IDE, but lots are not! You may need to install plug-in support which is called the BSP.

- **Get a Data/Sync USB cable for connecting your hardware.** A *significant* amount of problems folks have stem from not having a USB cable with data pins. Yes, these cursed cables roam the land, making your life hard. If you find a USB cable that doesn't work for data/sync, *throw it away immediately*! There is no need to keep it around, cables are very inexpensive these days.
- **Install any drivers required** - If you have a board with a FTDI or CP210x chip, you may need to get separate drivers. If your board has native USB, it probably doesn't need anything. After installing, reboot to make sure the driver sinks in.
- **Connect the board to your computer.** If your board has a power LED, make sure its lit. Is there a power switch? Make sure its turned On!

Start up Arduino IDE and Select Board/Port

OK now you are prepared! Open the Arduino IDE on your computer. Now you have to tell the IDE what board you are using, and how you want to connect to it.

In the IDE find the **Tools** menu. You will use this to select the board. If you switch boards, you *must switch the selection!* So always double-check before you upload code in a new session.

New Blink Sketch

OK lets make a new blink sketch! From the **File** menu, select **New**

Then in the new window, copy and paste this text:

Download File
Copy Code

```
int led = LED_BUILTIN;

void setup() {
  // Some boards work best if we also make a serial connection
  Serial.begin(115200);

  // set LED to be an output pin
  pinMode(led, OUTPUT);
}

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

  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(500);                // wait for a half second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(500);                // wait for a half second
}
```

Note that in this example, we are not only blinking the LED but also printing to the Serial monitor, think of it as a little bonus to test the serial connection.

One note you'll see is that we reference the LED with the constant `LED_BUILTIN` rather than a number. That's because, historically, the built in LED was on pin 13 for Arduinos. But in the decades since, boards don't always have a pin 13, or maybe it could not be used for an LED. So the LED could have moved to another pin. It's best to use `LED_BUILTIN` so you don't get the pin number confused!

Verify (Compile) Sketch

OK now you can click the Verify button to convert the sketch into binary data to be uploaded to the board.

Note that Verifying a sketch is the same as Compiling a sketch - so we will use the words interchangeably

During verification/compilation, the computer will do a bunch of work to collect all the libraries and code and the results will appear in the bottom window of the IDE.

If something went wrong with compilation, you will get red warning/error text in the bottom window letting you know what the error was. It will also highlight the line with an error.

For example, here I had the wrong board selected - and the selected board does not have a built in LED!

Here's another common error, in my haste I forgot to add a `;` at the end of a line. The compiler warns me that it's looking for one - note that the error is actually a few lines up!

Turning on detailed compilation warnings and output can be very helpful sometimes - Its in Preferences under "Show Verbose Output During:" and check the Compilation button. If you ever need to get help from others, be sure to do this and then provide all the text that is output. It can assist in nailing down what happened!

On success you will see something like this white text output and the message **Done compiling.** in the message area.

Upload Sketch

Once the code is verified/compiling cleanly you can upload it to your board. Click the **Upload** button.

The IDE will try to compile the sketch again for good measure, then it will try to connect to the board and upload a the file.

This is actually one of the hardest parts for beginners because it's where a lot of things can go wrong.

However, lets start with what it looks like on success! Here's what your board upload process looks like when it goes right:

Often times you will get a warning like this, which is kind of vague:

`No device found on COM66` (or whatever port is selected)
`An error occurred while uploading the sketch`

This could be a few things.

First up, check again that you have the correct board selected! Many electronics boards have very similar names or look, and often times folks grab a board different from what they thought.

If you're positive the right board is selected, we recommend the next step is to put the board into manual bootloading mode.

Native USB and manual bootloading

Historically, microcontroller boards contained two chips: the main micro chip (say, ATmega328 or ESP8266 or ESP32) and a separate chip for USB interface that would be used for bootloading (a CH430, FT232, CP210x, etc). With these older designs, the microcontroller is put into a bootloading state for uploading code by the separate chip. It allows for easier uploading but is more expensive as two chips are needed, and also the microcontroller can't act like a keyboard or disk drive.

Modern chips often have 'native' USB - that means that there is no separate chip for USB interface. It's all in one! Great for cost savings, simplicity of design, reduced size and more control. However, it means the chip must be self-aware enough to be able to put *itself* into bootload/upload mode on its own. That's fine 99% of the time but is very likely you will at some point get the board into an odd state that makes it too confused to bootload.

A lot of beginners have a little freakout the first time this happens, they think the board is ruined or 'bricked' - it's almost certainly not, it is just crashed and/or confused. You may need to perform a little trick to get the board back into a good state, at which point you won't need to manually bootload again.

Before continuing we *really, really* suggest turning on **Verbose Upload** messages, it will help in this process because you will be able to see what the IDE is trying to do. It's a checkbox in the **Preferences** menu.

Enter Manual Bootload Mode

OK now you know it's probably time to try manual bootloading. No problem! Here is how you do that for this board:

Once you are in manual bootload mode, go to the Tools menu, and make sure you have selected the bootloader serial port. **It is almost certain that the serial port has changed now that the bootloader is enabled**

Now you can try uploading again!

Did you remember to select the new Port in the Tools menu since the bootloader port has changed?

This time, you should have success!

After uploading this way, be sure to **click the reset button** - it sort of makes sure that the board got a good reset and will come back to life nicely.

After uploading with Manual Bootloader - don't forget to re-select the old Port again

It's also a good idea to try to re-upload the sketch again now that you've performed a manual bootload to get the chip into a good state. It should perform an auto-reset the second time, so you don't have to manually bootload again.

Finally, a Blink!

OK it was a journey but now we're here and you can enjoy your blinking LED. Next up, try to change the delay between blinks and re-upload. It's a good way to make sure your upload process is smooth and practiced.

Page last edited March 08, 2024

Text editor powered by tinymce.

I2C Scan Test

A lot of sensors, displays, and devices can connect over I2C. I2C is a 2-wire 'bus' that allows multiple devices to all connect on one set of pins so it's very convenient for wiring!

When using your board, you'll probably want to connect up I2C devices, and it can be a little tricky the first time. The best way to debug I2C is go through a checklist and then perform an I2C scan

Common I2C Connectivity Issues

- **Have you connected four wires (at a minimum) for each I2C device?** Power the device with whatever is the logic level of your microcontroller board (probably 3.3V), then a ground wire, and a SCL clock wire, and and a SDA data wire.
- **If you're using a STEMMA QT board - check if the power LED is lit.** It's usually a green LED to the left side of the board.

- **Does the STEMMA QT/I2C port have switchable power or pullups?** To reduce power, some boards have the ability to cut power to I2C devices or the pullup resistors. Check the documentation if you have to do something special to turn on the power or pullups.
- **If you are using a DIY I2C device, do you have pullup resistors?** Many boards do *not* have pullup resistors built in and they are *required!* We suggest any common 2.2K to 10K resistors. You'll need two: one each connects from SDA to positive power, and SCL to positive power. Again, positive power (a.k.a VCC, VDD or V+) is often 3.3V
- **Do you have an address collision?** You can only have *one* board per address. So you cannot, say, connect two AHT20's to one I2C port because they have the same address and will interfere. Check the sensor or documentation for the address. Sometimes there are ways to adjust the address.
- **Does your board have multiple I2C ports?** Historically, boards only came with one. But nowadays you can have two or even three! This can help solve the "hey, but what if I want two devices with the same address" problem: just put one on each bus.
- **Are you hot-plugging devices?** I2C does *not* support dynamic re-connection, you cannot connect and disconnect sensors as you please. They should all be connected on boot and not change. (Only exception is if you're using a hot-plug assistant but that'll cost you).
- **Are you keeping the total bus length reasonable?** I2C was designed for maybe 6" max length. We like to push that with plug-n-play cables, but really please keep them as short as possible! (Only exception is if you're using an active bus extender).

Perform an I2C scan!

Install TestBed Library

To scan I2C, the Adafruit TestBed library is used. This library and example just makes the scan a little easier to run because it takes care of some of the basics. You will need to add support by installing the library. Good news: it is *very easy* to do it. Go to the **Arduino Library Manager.**

Search for **TestBed** and install the **Adafruit TestBed** library

Now open up the I2C Scan example

Download File
Copy Code

```
// SPDX-FileCopyrightText: 2023 Carter Nelson for Adafruit Industries
//
// SPDX-License-Identifier: MIT
// ----------------------------------------
// i2c_scanner
//
// Modified from https://playground.arduino.cc/Main/I2cScanner/
// ----------------------------------------

#include <Wire.h>

// Set I2C bus to use: Wire, Wire1, etc.
#define WIRE Wire

void setup() {
  WIRE.begin();

  Serial.begin(9600);
  while (!Serial)
     delay(10);
  Serial.println("\nI2C Scanner");
}


void loop() {
  byte error, address;
  int nDevices;

  Serial.println("Scanning...");

  nDevices = 0;
  for(address = 1; address < 127; address++ )
  {
    // The i2c_scanner uses the return value of
    // the Write.endTransmisstion to see if
    // a device did acknowledge to the address.
    WIRE.beginTransmission(address);
    error = WIRE.endTransmission();

    if (error == 0)
    {
      Serial.print("I2C device found at address 0x");
      if (address<16)
        Serial.print("0");
      Serial.print(address,HEX);
      Serial.println("  !");

      nDevices++;
    }
    else if (error==4)
    {
      Serial.print("Unknown error at address 0x");
      if (address<16)
        Serial.print("0");
      Serial.println(address,HEX);
    }
  }
  if (nDevices == 0)
```

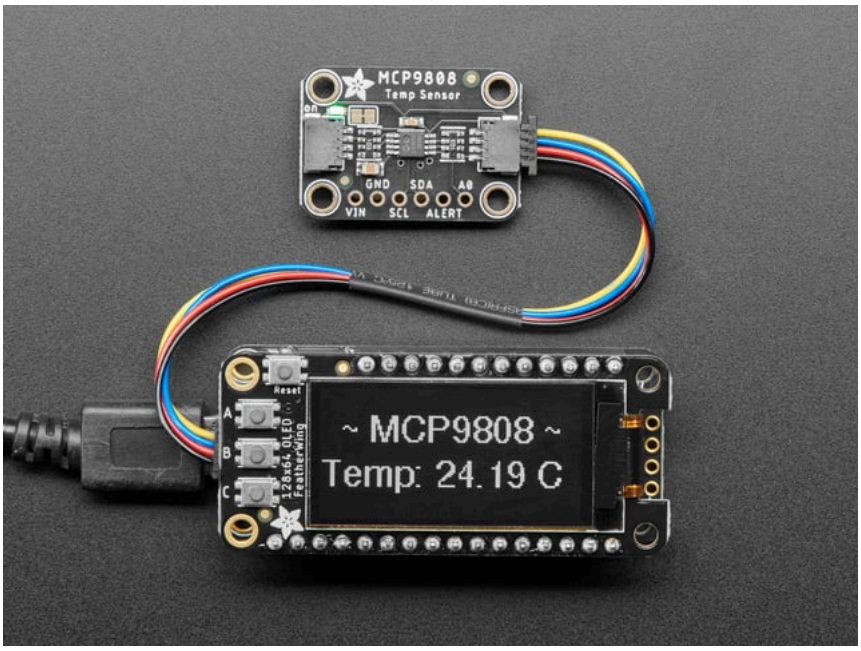
```
    Serial.println("No I2C devices found\n");
  else
    Serial.println("done\n");

  delay(5000);           // wait 5 seconds for next scan
}
```

View on GitHub

Wire up I2C device

While the examples here will be using the Adafruit MCP9808, a high accuracy temperature sensor, the overall process is the same for just about any I2C sensor or device.

The first thing you'll want to do is get the sensor connected so your board has I2C to talk to.



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...
guides with product
$4.95
In Stock
Add to Cart
Add to Wishlist
ledmesh
+ New List

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...
$0.95
In Stock

Add to Cart
Add to Wishlist
ledmesh
+ New List

Wiring the MCP9808

The MCP9808 comes with a STEMMA QT connector, which makes wiring it up quite simple and solder-free.

Now upload the scanning sketch to your microcontroller and open the serial port to see the output. You should see something like this:

Page last edited March 08, 2024

Text editor powered by tinymce.

Adafruit_NeoPXL8

This is the important one!

Using the 8-way output requires installing the **Adafruit_NeoPXL8** library in addition to **Adafruit_NeoPixel**. You can find these through the **Arduino Library Manager** (Sketch→Include Library→Manage Libraries…)

Adafruit_NeoPXL8 library use is explained in its own guide:

Adafruit NeoPXL8 FeatherWing and Library

Obviously the FeatherWing and ESP32 pages are not relevant to this particular hardware, but please *do* read the **Overview**, **RP2040**, **SCORPIO** and **NeoPXL8HDR** pages.

Page last edited March 08, 2024

Text editor powered by tinymce.

Powering SCORPIO NeoPixel Projects

Correctly powering NeoPixel projects — especially larger ones as the pixel count gets into the hundreds or even *thousands* — sometimes seems like a mystical dark art. The next few pages will help make some sense of this.

The confusion arises in that folks often want hard numbers. In the code domain, memory and bits are *finite and digital;* it's simply a *hard fact* whether X number of pixels can fit in Y amount of RAM.

Power isn't like that; it's *analog*. We know some *possible per-pixel minimums* and *maximums*, and some guiding *rules of thumb*, but it *never* works out into firm X-into-Y numbers. Doing this well requires some conceptualizing and some testing and measuring, perhaps on a limited smaller scale to start.

These other guides can provide some insights:

- **Adafruit NeoPixel Überguide** — especially the *Powering NeoPixels* page
- **Sipping Power with NeoPixels**
- **1,500 NeoPixel LED Curtain with Raspberry Pi and Fadecandy** — especially the *Power Topology* page

Some essential concepts can be distilled as:

- *Every* NeoPixel that's powered, even if "off" (set to color 0,0,0), draws a ***minimum amount of current*** to run the internal logic. The exact amount varies a bit among different WS2812-compatible LEDs, but a reasonable rule of thumb is **"about 1 milliamp per NeoPixel."** That is, 100 "off" NeoPixels might use around 100 mA of current. It's usually a bit less, but 1 mA provides a safety margin and keeps the back-of-envelope math simple.
- The **maximum** current per pixel (lit white at full brightness) *also* varies among devices, but a common rule of thumb here is **60 mA per RGB pixel, 80 mA for RGBW.** Again, reality will often be somewhat below these values, but they're easy and provide a comfortable margin.
- The *actual* current will be **somewhere between these two values,** and it just **isn't possible to get an accurate prediction,** especially with lots of animation. The "Sipping Power" guide mentioned above explains how to choose colors and program animation to skew more toward the minimum power estimates.
- The **current rating** of any conductor or power source — a length of wire, a battery, a copper PCB trace, a USB socket — is a *comfortable estimate* based on *continuous current draw* in typical setting (e.g. in open air). Most can deliver *more current* for *brief periods*, as long as it balances out over time. In a sense, *any* conductor is like a fuse, and the key here is to keep it in the safe zone. It's about *thermal equilibrium*.

There are **instruments** that can help in determining power draw; most **multimeters** have a current measuring mode, a decent **workbench power supply** will show actual current drawn, and so forth.

Caution with multimeters: it's common for these to have a **fuse** that trips at some number of milliamps or a couple amps at most. Check the manual. Large projects will surely exceed this rating, so it's often prudent to first **build a small section** of your design, measure and iterate on that, *then* scale up to the full project.

If you don't yet have these tools, a simple test is to feel if the board or any wires start getting uncomfortably warm to the touch. If so, the pixels are pulling more current than the conductors can safely supply. **Stop immediately** and plan more robust power distribution.

Page last edited March 08, 2024

Text editor powered by tinymce.

LiPoly Battery (JST)

Small projects — up to a few dozen NeoPixels, *maybe* a couple hundred if you're *extremely selective* about animation — can be powered with a lithium-polymer battery connected to SCORPIO's JST socket, and recharged via the USB port. This is the **most compact option** for fully **self-contained** projects, such as **wearables** and **cosplay**.

Before proceeding, consider:

- As with all of these examples, you'll need to do some **wiring** and **soldering** to distribute power to all of the NeoPixel chains.
- One should understand the **safety and durability issues surrounding bare LiPoly cells.** If this raises red flags, powering from USB (next page) might be preferable.
- To turn the circuit off, you'll need a beefy **DPDT switch** (double pole, double throw).
- **There are a number of other "gotchas" with this power arrangement, and one should have an intuitive understanding of issues discussed below. Consider using one of the other approaches if at all possible. This is only for the smallest, tightest situations.**

Here's a schematic diagram of how the parts connect. It's laid out to make all the connections clear, though the actual physical wiring will likely take a different shape… and might have JST plugs on the strips, etc.:

This shows NeoPixel strips, but *any NeoPixel-compatible part* could be used (rings, dot strands, etc.), and in any combination. Most software (e.g. NeoPXL8 library) has an additional constraint that they're **all RGB** or **all RGBW**; can't mix the two types. Also, you don't have to use all 8 outputs; fewer is fine if that's all your project needs.

Remember that **this arrangement is only suitable for smaller projects with few pixels lit; perhaps 1 Amp (1000 mA) overall.** See the top page of this section regarding minimum and maximum per-LED current guidelines. This might suffice for a self-contained build of the Ooze Master 3000 project, where only a small fraction of LEDs are lit at any time. **It does not "scale up."**

It's unusual in that we're using the ground connections at the end of the board as part of the power distribution; in larger, more power-hungry projects, power distribution to the LEDs is a separate thing, and those pads are only used for ground *reference* between board and power supply, not as current-carrying members themselves.

Positive voltage must be **split** from SCORPIO's **BAT** pin to the 5V input on each of the NeoPixel chains — either wiring up a splitter or chaining one to next, or using part of a Perma-Proto board or a power distribution bus. The NeoPixels in this case are powered straight off the battery — nominally around 3.7V give or take — rather than the typical 5V, but this works fine.

You don't *have* to run the ground wires this way. If it's easier in your project to split or chain as done with the positive voltage, do that and use just one ground pin.

Notice the DPDT switch is used in a "staggered" configuration. In one position, only EN and GND are connected, powering off the board and disconnecting power to the NeoPixels (so they don't drain the battery when "off"). In the other position, only BAT and 5V (to NeoPixel chains) are connected. Charge the battery with the switch in the "off" position.

Most small toggle switches are **insufficient** for this setup due to their **limited DC current rating**, sometimes just milliamps. Check the product info or datasheet, or use Digi-Key's parametric parts search. DPDT switches rated for **≥1A** are not inexpensive!

LiPoly batteries themselves have best practices for current draw that must be observed. Product info might mention something like a "C/10" or "C/3" discharge rate (e.g. can safely draw 1/10 or 1/3 of capacity per hour, respectively). The ones we sell have built-in protection against over-current.

When uploading new code to the board, the power switch must be in the "on" position. This presents a *situation:* the NeoPixels are being powered from ~3.7V, but logic signals from the board are shifted to USB's 5V, exceeding the pixels' supply voltage. All present-generation NeoPixels (SK9812 LEDs) have input protection and work fine this way…but other "compatible" pixels (WS2812, etc.), and old first-generation NeoPixels, *may or may not*. For maximum reliability in this situation, *disconnect the battery when uploading new code,* and conversely, set the switch to the "off" position when charging the battery.

Page last edited March 08, 2024

Text editor powered by tinymce.

Single USB Port

Again for **small** projects — dozens or a couple hundred NeoPixels with careful animation — powering the whole project through SCORPIO's **USB port** is possible, and is the simplest of all the circuits. Paired with an inexpensive and ubiquitous **USB power bank** you might already have, this is another good option for wearables and cosplay, albeit a little bulkier than the prior example.

Before proceeding, consider:

- As with all of these examples, you'll need to do some **wiring** and **soldering** to distribute power to all of the NeoPixel chains.
- **There are still a few "gotchas" with this power arrangement. Read through to see if this is adequate for your project aspirations.**

Here's a schematic diagram of how the parts connect. It's laid out to make all the connections clear, though the actual physical wiring will likely take a different shape… and might have JST plugs on the strips, etc.:

This shows NeoPixel strips, but *any NeoPixel-compatible part* could be used (rings, dot strands, etc.), and in any combination. Most software (e.g. NeoPXL8 library) has an additional constraint that they're **all RGB** or **all RGBW**; can't mix the two types. Also, you don't have to use all 8 outputs; fewer is fine if that's all your project needs.

Remember that **this arrangement is only suitable for smaller projects with few pixels lit; perhaps 1–2 Amps (1000–2000 mA) overall.** See the top page of this section regarding minimum and maximum per-LED current guidelines. This might suffice for a self-contained build of the Ooze Master 3000 project, where only a small fraction of LEDs are lit at any time. **It does not "scale up" very much.**

It's unusual in that we're using the ground connections at the end of the board as part of the power distribution; in larger, more power-hungry projects, power distribution to the LEDs is a separate thing, and those pads are only used for ground *reference* between board and power supply, not as current-carrying members themselves.

Positive voltage must be **split** from SCORPIO's **USB** pin to the 5V input on each of the NeoPixel chains — either wiring up a splitter or chaining one to next, or using part of a Perma-Proto board or a power distribution bus.

You don't *have* to run the ground wires this way. If it's easier in your project to split or chain as done with the positive voltage, do that and use just one ground pin.

The main "gotcha" of this configuration comes when it's time to upload new code to the board. Many computers' USB ports can only provide a maximum of 500 mA current…any more and they shut the port right off. If the board's already programmed to show animation when powered on, it might immediately exceed this, get disconnected and then can't accept code.

One (sometimes) solution is to use a **USB 3.0** (or later) port or powered hub, which can usually deliver **900 mA**, which may suffice depending on the already-programmed animation.

More consistently reliable is to **hold down SCORPIO's BOOT button** when connecting to USB, which jumps straight into the bootloader; NeoPixel animation code never has a chance to start. Though "off" NeoPixels still draw a little current, a modest-sized project isn't likely to exceed 500 mA in this state. Once programmed, power-hungry animation might then cause the computer to shut down the USB port…but that's okay, just unplug and use a USB power bank at that point.

The other "gotcha" can occur if exceeding the USB power bank's current limit. Most can reliably deliver about 1.5 Amps (1500 mA) from a USB-A port, some up to 2.5A. Exceed this for more than a couple seconds, and usually a protection circuit will kick in, shutting off power. **On some power banks, the only way to reset the protection circuit is to plug it into a charger.** So if this happens "in the field," your SCORPIO project might be dead in the water…super embarrassing at the cosplay contest! **Therefore:** do *a lot* of testing, work on animation code to keep it well within limits of your power supply…or consider one of the other example power circuits.

Page last edited March 08, 2024

Text editor powered by tinymce.

5V DC Power Source

For **larger** projects — hundreds or thousands of pixels, or even with smaller numbers but with bright, power-hungry animation — powering through the SCORPIO board isn't practical and the NeoPixels must be powered externally. The power source might be a 5V DC power brick, or a larger battery that can deliver ample current.

Here's a schematic diagram of how the parts connect. It's laid out to make all the connections clear, though the actual physical wiring will likely take a different shape… and might have JST plugs on the strips, etc.:

Again this example shows NeoPixel strips, but *any NeoPixel-compatible part* could be used.

The ground pins at the end of the board are no longer used for power distribution. As current needs increase, NeoPixels should tap *directly* into a 5V source and ground. A *single* ground connection between board and pixels ensures a common point of reference, but isn't for *delivering* power.

The 5V DC power source for the NeoPixels might be a power brick, bench power supply, or large battery (the nominal 3.7V from a LiPoly cell is sufficient, it doesn't really *need* to be a full 5 Volts). Something like a barrel jack connector might be used here for convenience.

Power to the SCORPIO board can be provided one of three ways:

- **(A) above:** Through the USB port, either from a USB hub, phone charger, or cutting up a USB cable to tap into the same 5V supply as the NeoPixels. This is usually simplest.
- **(B):** a LiPoly battery plugged into the JST connector. Perhaps for portable projects, if you don't want to tap into the same power source as the NeoPixels.
- **(C):** connect USB and GND to the same 5V supply as the NeoPixels. If you go this route, unplug the power supply when connecting USB to program the board. The extra wire between NeoPixel ground and edge-of-board ground isn't needed in this case, since everything's in the same domain.

If it's your first time building a large NeoPixel project, *please* familiarize yourself with the **Power Topology** page of the 1,500 NeoPixel LED Curtain with Raspberry Pi and Fadecandy guide. The key point here is to use adequate conductors for carrying power. Wires that are hot to the touch are a fire hazard and a sure indication of inadequate gauge.

Page last edited March 08, 2024

Text editor powered by tinymce.

Split Power

Really big projects might exceed the current rating of a single power supply. If a larger single power source isn't available or is beyond one's budget, it's possible to **split up power demands among multiple sources**.

This isn't just for big architectural installations! It's also handy with really extravagant wearables, cosplay, art bikes and other mobile contraptions. USB power banks are affordable, ubiquitous and reasonably safe…but each USB-A port can only sustain 1–2 Amps of current. By splitting demands among multiple ports and/or multiple power banks, ambitious portable projects can be build with relative ease (albeit with lots of batteries to top off).

The number and capacity of power supplies will require some estimation on your part, and testing and measuring smaller sections of a project in isolation to determine overall needs and how to split things up.

Let's imagine a hypothetical situation where it's been determined that power requirements can be split across three sources (whether that's power bricks or USB batteries is immaterial). One power supply will feed two strips, the other two will feed three strips each. (Conceivably a project could be split *all eight ways,* but let's keep the example manageable!)

Here's a schematic diagram of how the parts connect. It's laid out to make all the connections clear, though the actual physical wiring will likely take a different shape… and might have JST plugs on the strips, etc.:

As usual, the example shows NeoPixel strips, but *any NeoPixel-compatible part* could be used.

Notice there are three groups of strips, each group powered from its own 5V supply; there are no direct connections from one group or one power supply to the next, each is in isolation.

However…each group has a single **ground connection to the SCORPIO board** (shown as gray wires). As explained on the prior page, this isn't for delivering power, but to provide a common point of reference through the whole circuit.

Also like the prior page, power to the SCORPIO board can be provided one of three ways:

- **(A) above:** Through the USB port, either from a USB hub, phone charger, or (with USB power banks) a USB cable into a second or third port on the battery. This is usually simplest.
- **(B):** a LiPoly battery plugged into the JST connector. Perhaps for portable projects, if you don't want to tap into the same power source as the NeoPixels.
- **(C):** connect USB and GND to one of the NeoPixel 5V supplies. If you go this route, unplug the power supply when connecting USB to program the board.

Read the "Single USB Port" page regarding the **protection circuit** in USB power banks. Allow some safe overhead; don't "redline" the circuit or your dramatic reveal might get cancelled!

If it's your first time building a large NeoPixel project, *please* familiarize yourself with the **Power Topology** page of the 1,500 NeoPixel LED Curtain with Raspberry Pi and Fadecandy guide. The key point here is to use adequate conductors for carrying power. Wires that are hot to the touch are a fire hazard and a sure indication of inadequate gauge.

Page last edited March 08, 2024

Text editor powered by tinymce.

FAQ

Q: How Many NeoPixels Can I Control With This?

A: Buckle Up…

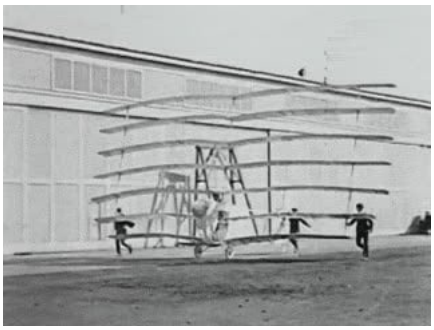
In Douglas Adams' *The Hitchhiker's Guide to the Galaxy,* the mind-bogglingly powerful supercomputer *Deep Thought* is tasked to find once and for all The Ultimate Answer to Life, the Universe and Everything. Running for 7.5 million years, Deep Thought finally issues an answer — 42, carefully checked and verified — and suggests if the result isn't satisfactory then *maybe they're asking the wrong question*.

"How many NeoPixels" is very much like that. Folks want a concrete answer, like *42,* but there are *so many layers of nuance* it's just not that simple.

If you'll settle for a **quick but imprecise answer**, let's say the **NeoPXL8** library running on **Feather RP2040 SCORPIO** can reasonably manage up to about **10,000 RGB NeoPixels**, or **6,000 RGBW pixels**. **NeoPXL8HDR**, about 1/10th that — **1,000 RGB** or **600 RGBW pixels**. These are very round "ish" figures involving a *lot* of hand waving and rough napkin math, and assume the pixels are fed power from an external source, not through the board's USB-C port. Particular situations could yield different outcomes, either more or fewer. So let's peel back some of those layers…

By way of **metaphor**…consider this diagram of aerodynamic forces from every 6th grade science book.

The point of this illustration is that flight is *never* dictated by just one simple pass/fail number. A *continual evaluation of all these changing forces* determines whether the airplane takes flight…



…or just rolls along the ground…maybe snapping and folding in a heap for comedic effect.

Some of the "forces" that all play into the success (or failure) of large-scale NeoPixel projects include:

- **RAM**
- **Signal bandwidth**
- **CPU speed**
- **Power**
- **Cost**
- **Subjective human perception**

RAM

In the classic plain-vanilla ***Adafruit_NeoPixel*** library, each **RGB** pixel requires **3 bytes** of RAM. **RGBW** pixels are **4 bytes** each. The popular Arduino Uno microcontroller board of days past provides **1.5 kilobytes** of RAM. Just keeping the Arduino "core" and user code running takes some overhead, so there's **a little over 1K free** for things like NeoPixels…let's say around **350** of the **RGB** variety, or **250 RGBW**, to use very rounded figures.

In the ***Adafruit_NeoPXL8*** library, pixels require at least ***twice*** the space: **6 bytes** per **RGB** pixel, or **8 per RGBW**. This is because there's *two copies* of the pixel data: one in "original NeoPixel" format for speedy access, and an internal copy where all the data has been shuffled around for SCORPIO's concurrent 8-way output. ***Adafruit_NeoPXL8HDR*** requires still *more,* about **5X** the base amount.

All this additional data would seem like a drag, except that contemporary microcontrollers are packed with *inordinate* amounts of RAM! The **RP2040** chip at the heart of Feather SCORPIO has **264 kilobytes**. So even in the doubled-up NeoPXL8 format, there's a theoretical *potential* for nearly **45,000 RGB** NeoPixels, or **33,000 RGBW**… *way beyond* our simple 10,000/6,000 answer earlier. We dial it back because **at this scale, all the other "forces" play a much larger role than RAM alone…**

Signal Bandwidth

A string of NeoPixels receives data at a constant rate…and not a terribly fast one. Bits are sent down the wire, bucket-brigade style, at **800,000 bits per second (800 Kbps)**. For most runs of modest length, this works fine, but as projects grow it can become a bottleneck. NeoPXL8 was *specifically written* to tackle this bottleneck by splitting the problem into 8 smaller ones…but if a project *really* grows in scope, we're back to square one (or like, 8 *square ones* concurrently).

800 Kbps is equal to **1.25 microseconds** per bit (1.25 µS/bit). Each **RGB** NeoPixel expects **24 bits** of color data (**30 µS**). **RGBW** pixels, **32** bits (**40 µS**). And then, to mark the end of data, there's a **300 µS** pause with no activity on the line.

If we were to max this out to that *theoretically possible* 45,000 pixel figure (eight strands of 5,625 RGB NeoPixels each), *every single time there's an update,* it would take 169,050 microseconds to issue that much data. Over 1/6 of a second. Meaning it could *never* animate anything that large faster than **six frames per second (6 FPS)**. That's not good. But what even *is* good?

Subjective Human Perception

Going out-of-sequence for a moment, let's consider that last figure. Why is 6 FPS inadequate? It *isn't necessarily*. That all depends what you're trying to achieve…it just happens that *most* NeoPixel projects are aiming for **smooth continuous motion**.

What constitutes "smooth" is a topic of much debate, and you may have heard figures like 24, 25, 30, 50 or 60 frames per second tossed about. *None* of these are Sacred Golden Things handed down from Mount Olympus…just various rates standardized upon in film and TV. The hotness in gaming right now is 120 FPS monitors. Meanwhile, a lot of really top-notch traditional animation is shot "on twos," meaning just 12 frames per second…an order of magnitude difference! It's all so variable and in reality you don't have to hit *any* of these numbers, nor even remain constant.

It's simply subjective. What looks good? What's *smooth enough?*

Since it's been relied upon in film for close to a century now, let's generalize that "around 24 FPS usually starts to looks nice." *More* often looks *better, less* might be *adequate,* but let's say it "looks nice." Rounding up to a mathematically-simpler 25 FPS, each frame is then visible for 40,000 µS…and *all* the NeoPixel data for *each* frame needs to be transmitted within that time window. Subtracting the 300 µS "latch time," and dividing by 30 µS per RGB pixel yields 1,323 pixels per strand, so 10,584 total at 25 FPS. We round to **10,000** because it's easier to say, and because it's really just an approximate *aspirational* target…it's highly unlikely anything will *actually* build up quite this large, as **there's still other forces to consider**. The board and code could "reasonably manage" this many NeoPixels and animate them "well enough," so that's the source of our answer…but really that's just the *start* of an uphill climb…

CPU Speed

So you might *possess* 10,000+ NeoPixels, a suitable power source, and the hardware might be *capable of sending* all that data many times per second…but where is that data coming from? It has to be *generated by your own code* on the microcontroller…often by visiting every single pixel. Can you even *calculate* that many pixels in that short a time? It often depends on how complex the animation is.

Working to your **advantage**, the NeoPXL8 library uses **direct memory access (DMA)** to transmit all the data *without CPU intervention*…the transfer proceeds in the background while the CPU is totally free to calculate pixels for the next frame. This is in contrast to the classic NeoPixel library, where everything stops during these updates.

Working **against** you…first, a small amount of overhead is needed to "shuffle" the pixel data you generate into the format that goes down the wires. It's *not a lot* of time, but it's *present*. Second, a bigger factor is just the *math needed to calculate the animation* for all those pixels…and progressively fewer microseconds available to do this at higher frame rates. Not insurmountable, but typically either the animation complexity or the number of pixels has to give…often it's some compromise between these. 10,000 might be *optimistic,* but it's *reasonable* as a soft limit to toss around and answer the most frequently-asked question.

If you program some gorgeous animation but it's just a *little* too complex and the frame rate is *slightly less* than you'd like, keep in mind that the RP2040 microcontroller is very accepting of overclocking (**Tools→CPU Speed** menu … even 200 MHz is usually robust, though higher speeds might get unreliable), and you can also try different compiler optimization settings (**Tools→Optimize**), though these often have unintended consequences.

If you have an *idea* for a large project but aren't certain it can scale up due to this potential bottleneck, try writing some code *as if* you had all those pixels connected, but in reality nothing is there yet. Count frames and elapsed time (using `millis()`) to gauge the average frame rate. Or, if you have some NeoPixels around, try connecting fewer or shorter runs (but with the code configured for full size) and see if this smaller section looks good or if animation is too choppy. If the results are encouraging, then proceed to build upward!

Power

More than anything else, *this* is what kneecaps really large projects. You might have all the speed and frame rate issues worked out, but just *getting enough power* distributed through massive NeoPixel projects (and doing it ***safely***) can be really challenging. Without careful consideration of overall brightness and color across several thousand LEDs, this could even trip a typical house circuit breaker. This is why "10,000" is a "theoretical" maximum — we don't actually *recommend* taking it that far, or even half that, but it's *possible* and ***you asked for a number.***

A couple of other guides go into more detail on this topic:

- **On scaling *up:*** **1,500 NeoPixel LED Curtain**
- **On scaling *back:*** **Sipping Power With NeoPixels**

NeoPXL8HDR

NeoPXL8HDR takes a different approach. Rather than more and more pixels, it takes **fewer** pixels and pulls tricks to make them **look nicer**. This refreshes the NeoPixels *hundreds of times per second* to "dither" colors and brightness. But…as discussed in the Signal Bandwidth section above…there's an inherent **data bottleneck**. Too many pixels and the effect slows down and falls apart.

This is why the recommended maximums are **1/10** as much: **1,000 RGB** pixels (as eight 125 pixel runs) or **600 RGBW** (8x75). But **these are not actually hard limits**, it's just where the dithering effect subjectively starts to come apart. You *can* go higher by dialing back the precision to 11 or even 10 bits, it's just not as satisfying. And shorter runs will always look better because they refresh (dither) faster.

Red Herring

All of this is why "How many NeoPixels" might be a distraction; the wrong question. Big numbers *catch attention,* and SCORPIO is capable of *going there,* but it's worth further asking if those big numbers are really *necessary* for your art and your soul.

Funny thing…NeoPXL8, NeoPXL8HDR, SCORPIO…these weren't really motivated by *large* projects, but by *topologically problematic* ones. Wearables, cosplay, stage costumes for example…there was a tendency for creators to wrap one massively long run of NeoPixels around a body. But given bodies' range of motion and flexibility, breaks are nearly inevitable. Splitting this into multiple runs can't *prevent* breaks, but when they happen, the problem is *localized,* not catastrophic…

We get it, **twinkling lights elicit smiles!** But this *doesn't necessarily scale linearly*. 200 NeoPixels might have greater impact than 100, but it's *unlikely* to be fully *twice* the impact. It seems the folks most impressed by massive NeoPixel projects are *other folks who've struggled to built massive projects*. Consider your audience, and how much (or little) is really needed to make *them* happy.

Source: stuffman-art on Tumblr

It's funny that, as engineer types, we tend to glorify simplicity and minimalism — efficient and bloat-free code, the barebones circuit designs of Wozniak or Sinclair, the geometric purity of a Dieter Rams radio — but **toss some NeoPixels in front of us and we lose all control.** SCORPIO addresses some practical barriers to bigger projects, but in doing so opens doors to a new set of issues. If you find yourself unhealthily obsessing over "how many," it might be time to challenge yourself to how *few.*

If anyone asks though, just tell them 10,000 NeoPixels. Saves time, sounds cool.

Page last edited March 08, 2024

Text editor powered by tinymce.

Downloads

Files:

- RP2040 Datasheet
- 3D Models on GitHub
- Fritzing file

SCORPIO PrettyPins SVG

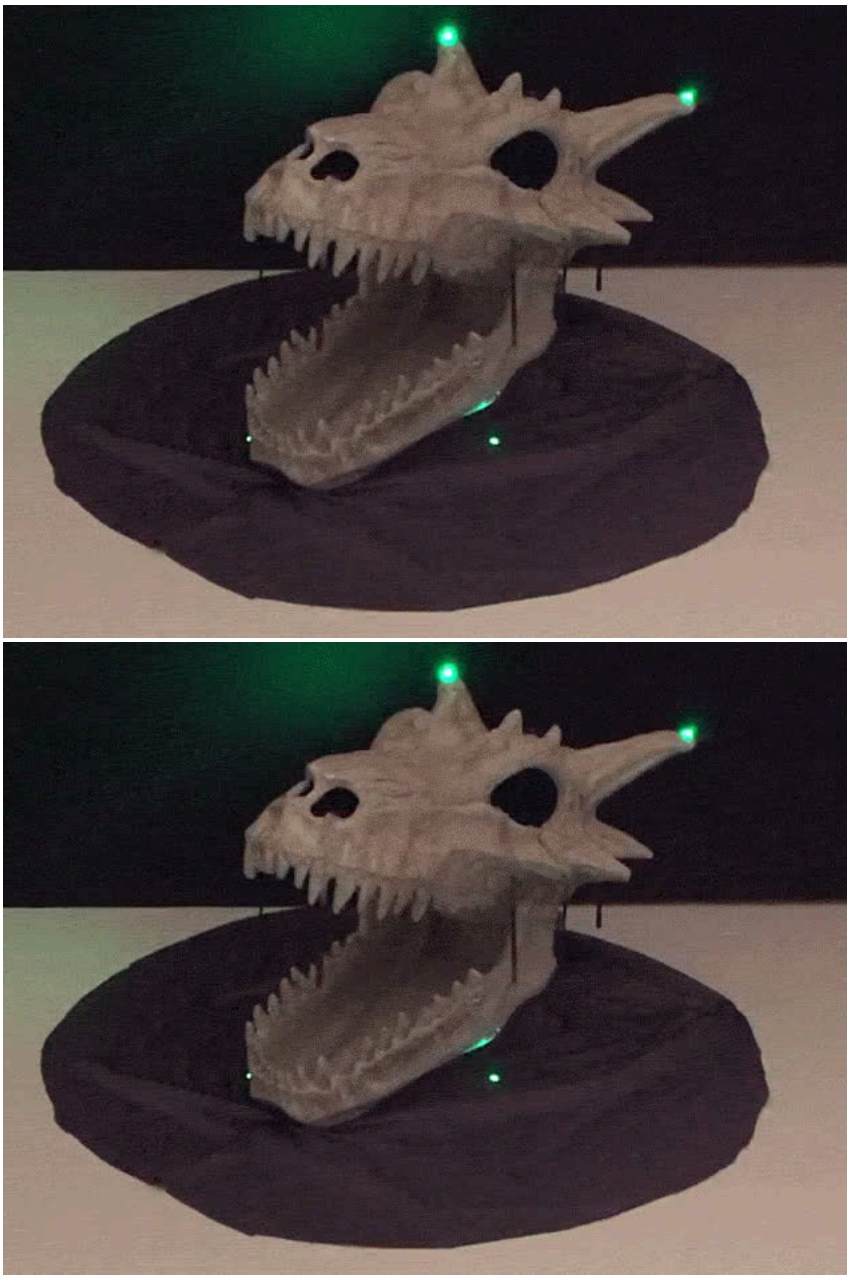
Schematic

Fab Print

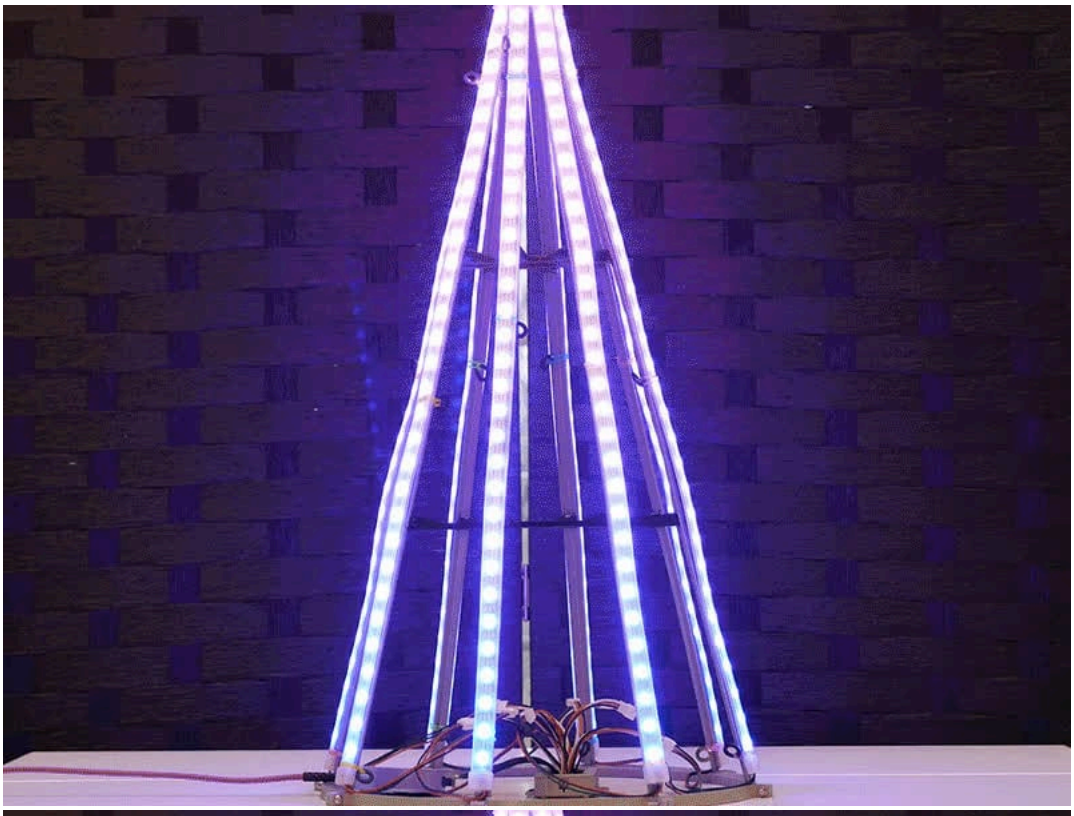
Page last edited March 08, 2024

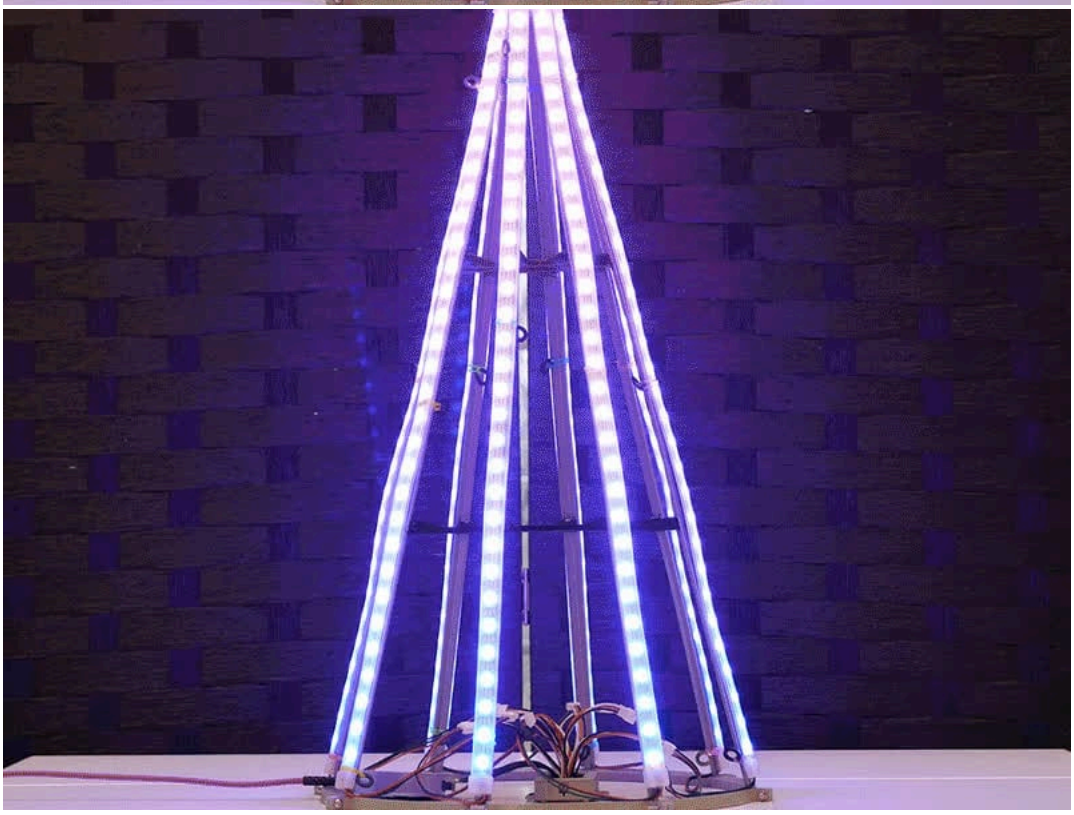
Text editor powered by tinymce.
Related Guides

Holiday Icicle Lights with Flair
By Phillip Burgess
beginner

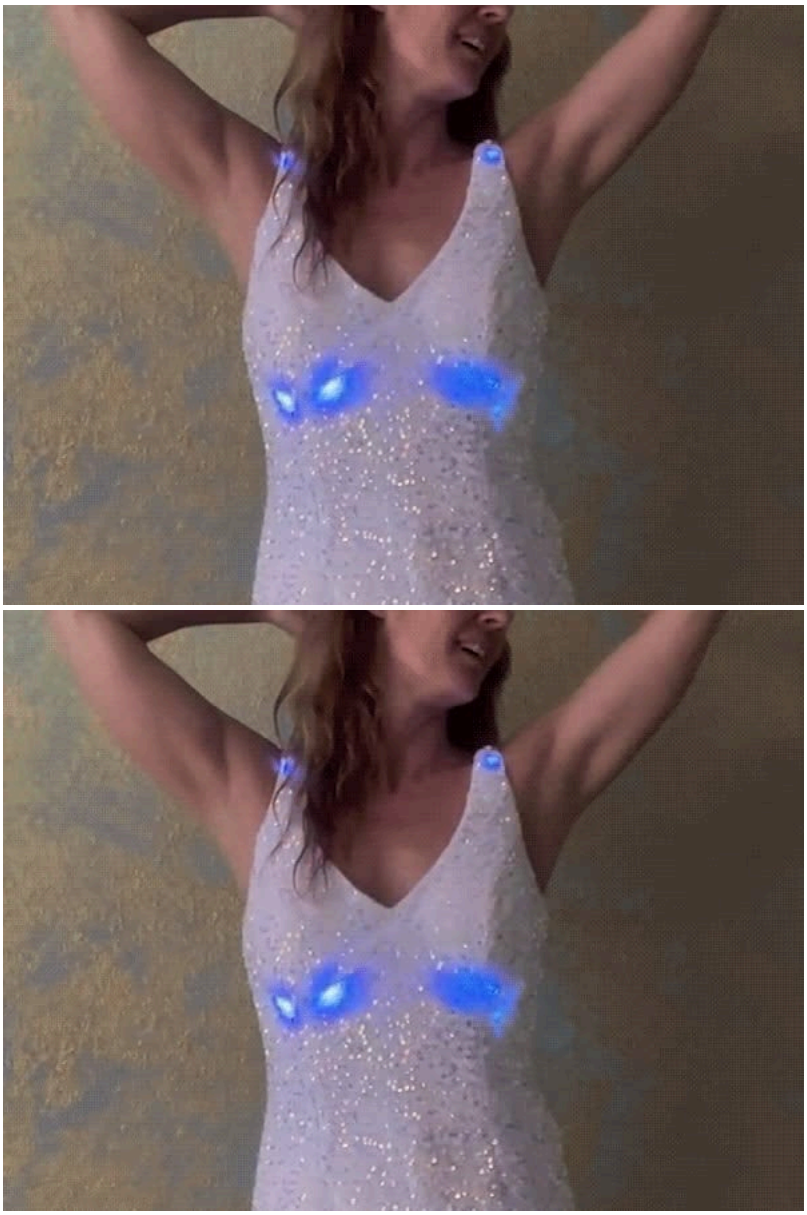


OOZE MASTER 3000: NeoPixel Simulated Liquid Physics
By Phillip Burgess
intermediate





Holiday Tree with Feather RP2040 Scorpio
By Ruiz Brothers
intermediate

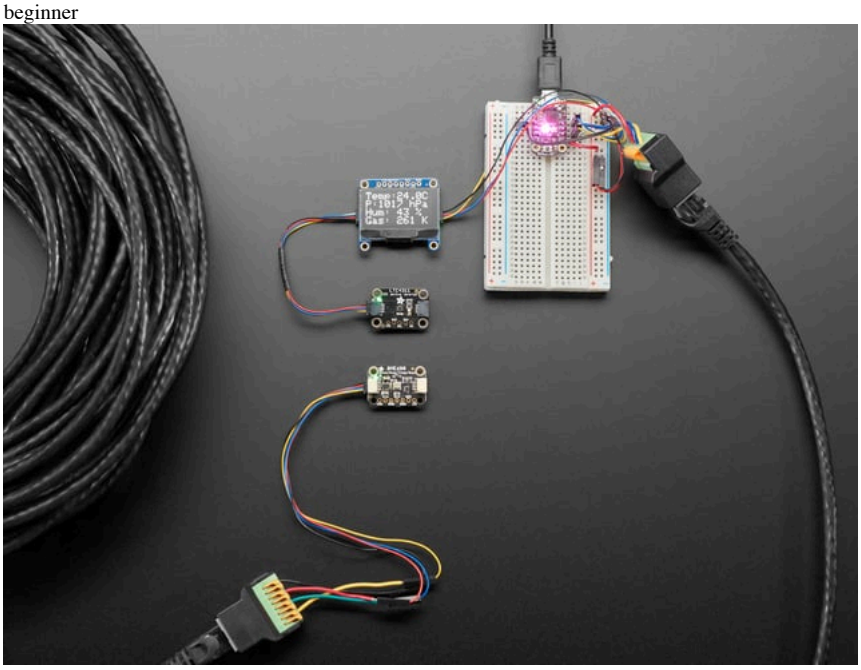


[Water Drip Dress with Oozemaster 3000]
By [Erin St Blaine]
advanced

[Feather Scorpio Snap Fit Case]

By Ruiz Brothers
beginner

Heart Rate Badge
By Becky Stern
beginner

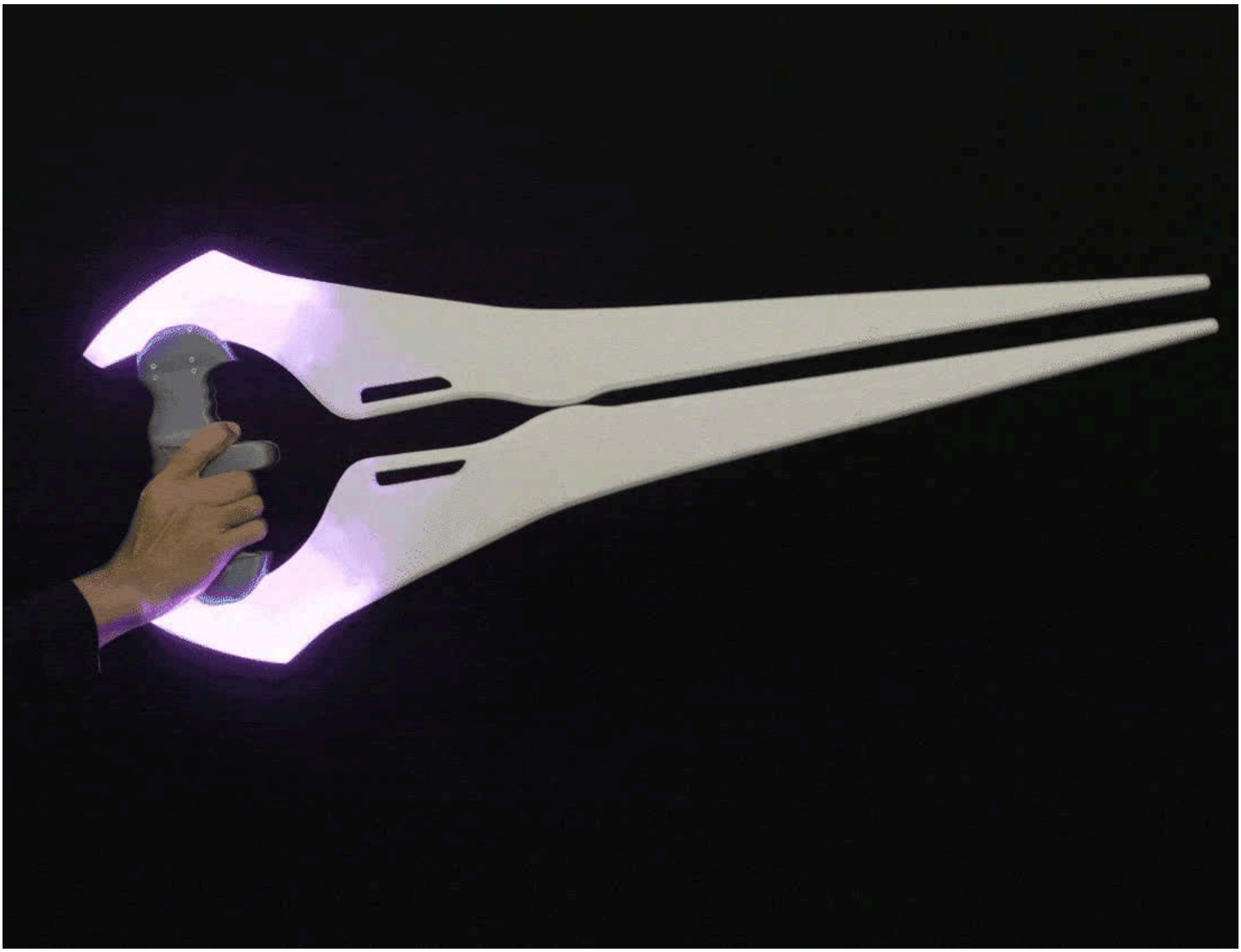


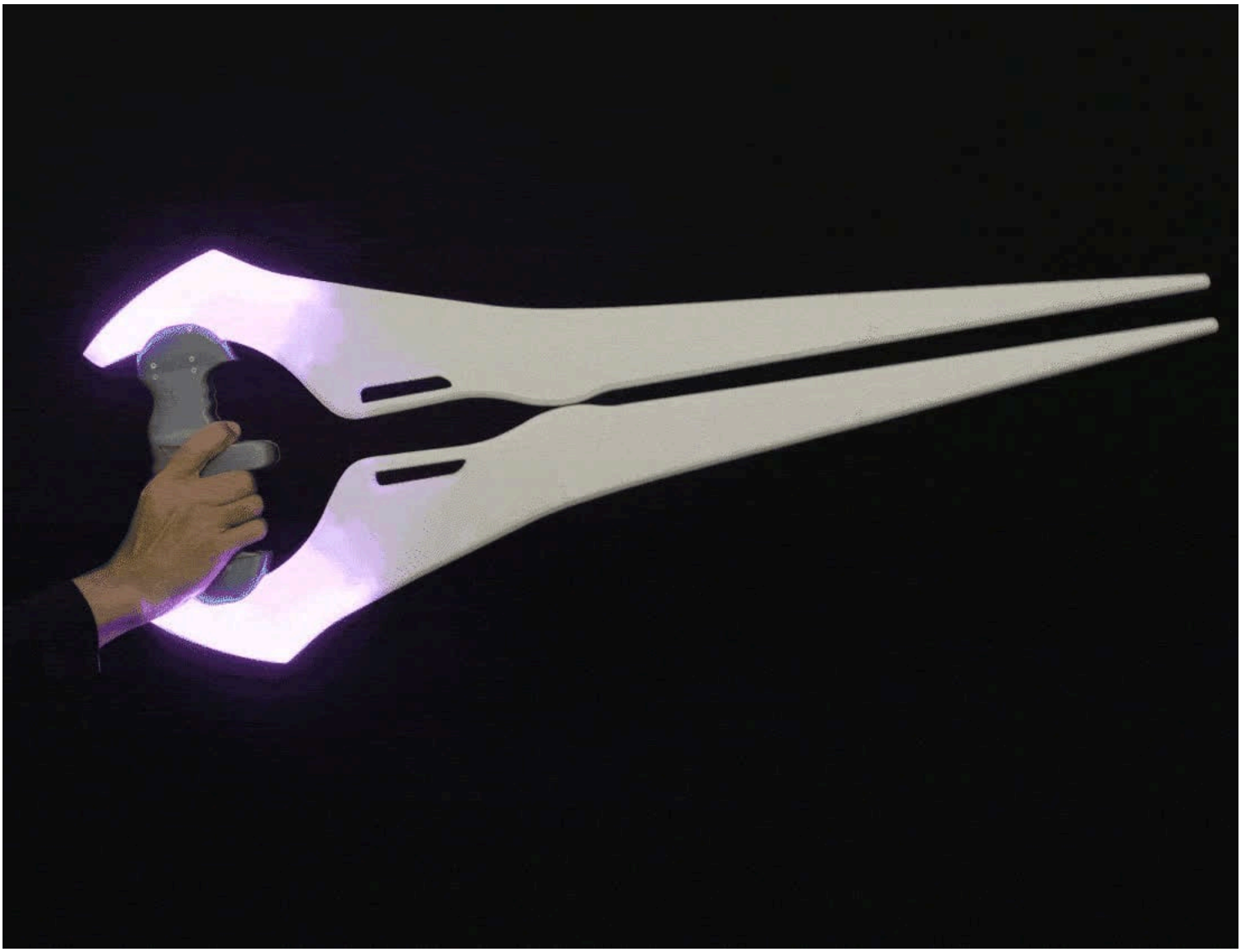
Adafruit LTC4311 I2C Extender / Active Terminator
By Kattni Rembor
beginner

Adafruit LIS3DH Triple-Axis Accelerometer Breakout
By lady ada
intermediate

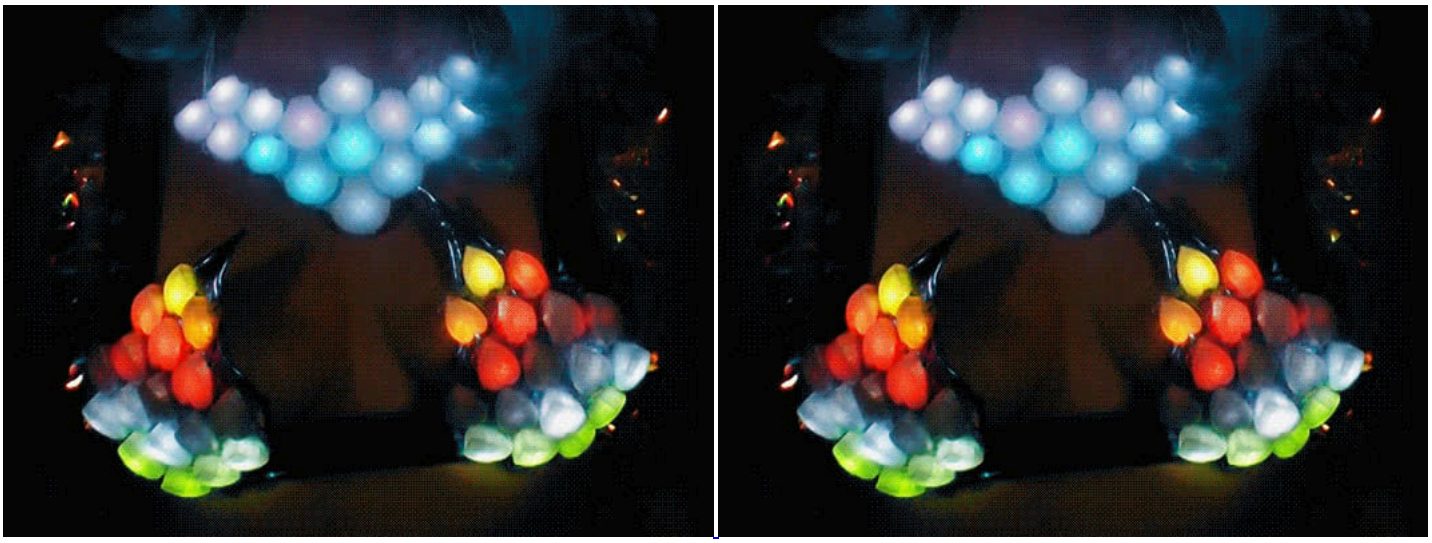
PyRuler Video Conference Panic Buttons
By John Thurmond
beginner

Adafruit ESP32-S2 TFT Feather
By Kattni Rembor
beginner

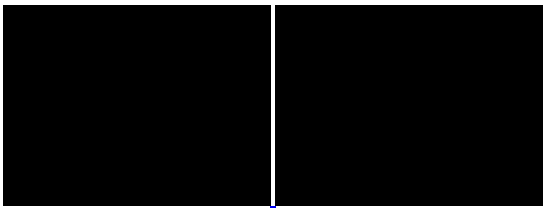




Halo Energy Sword RP2040
By Ruiz Brothers
intermediate



Glowing Scale Armor
By Erin St Blaine
intermediate



ReBoots Animated LED Boot Laces
By Erin St Blaine
beginner

RGB Matrix Portal Room CO2 Monitor
By Carter Nelson
beginner

MagTag Google Calendar Event Display
By Brent Rubell
intermediate

Create Wishlist

×
Title

Description

Create Wishlist Close

Search

Search

Categories

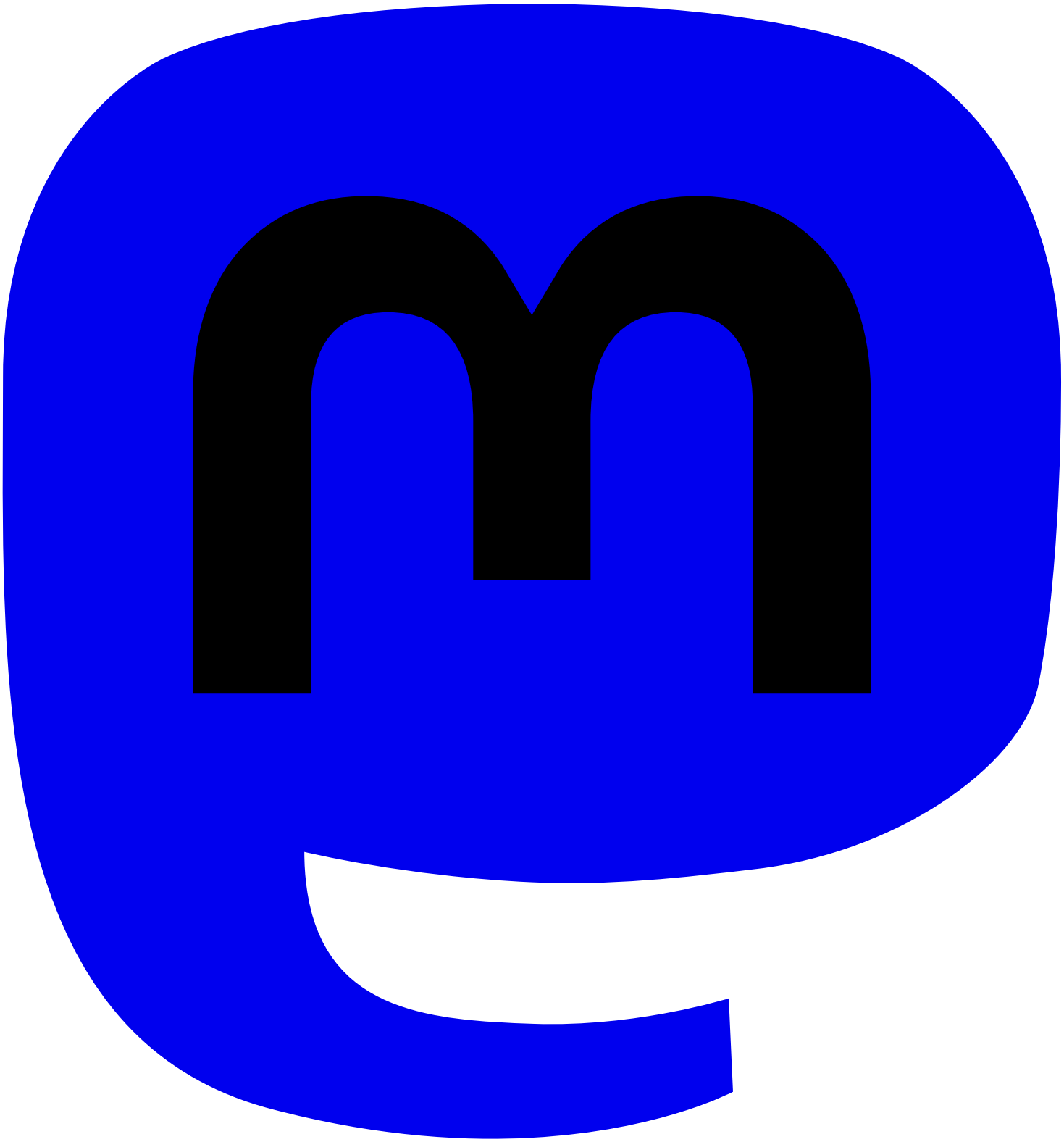
No results for query

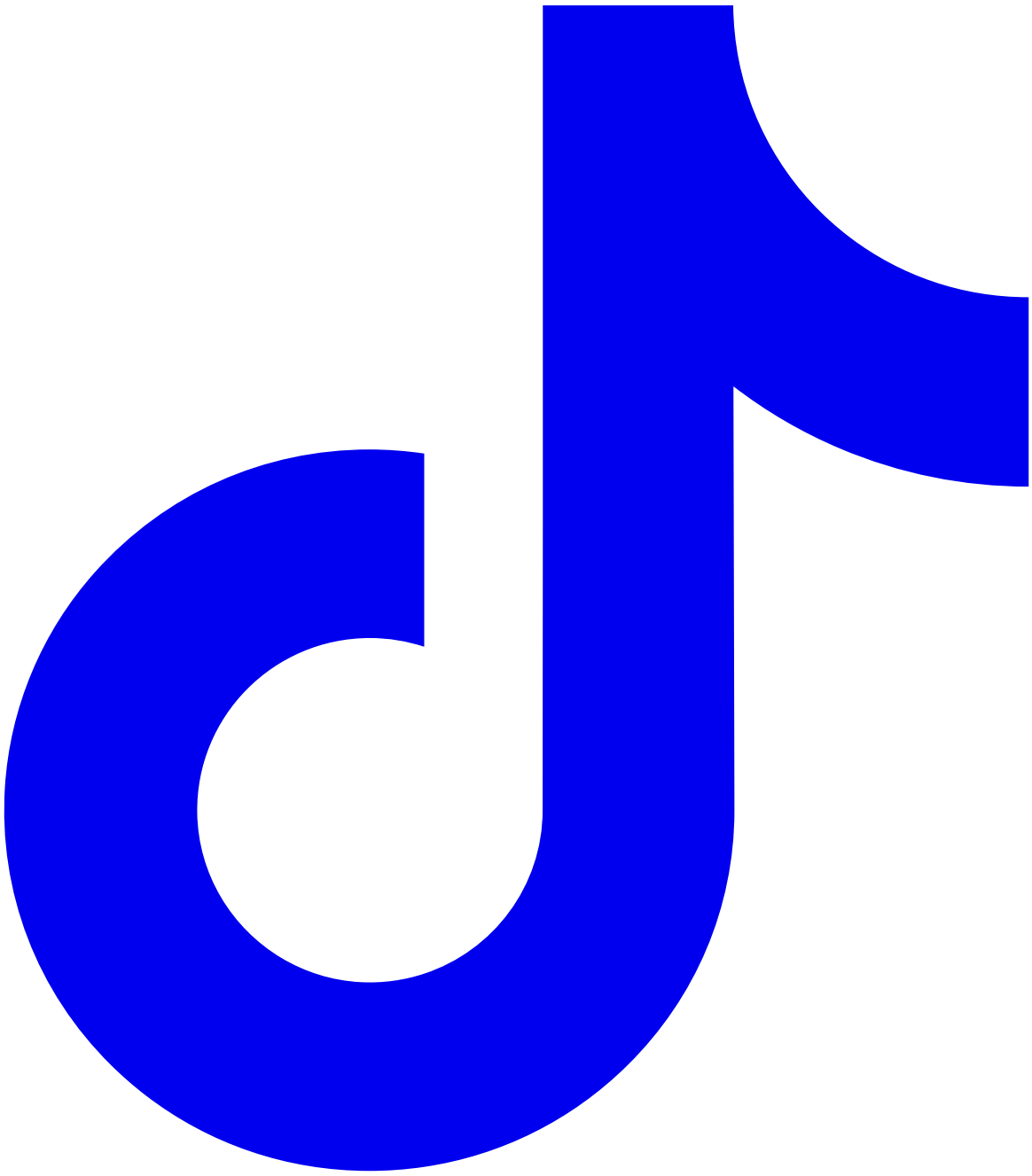
- «
- ‹
- 1
- ›
- »

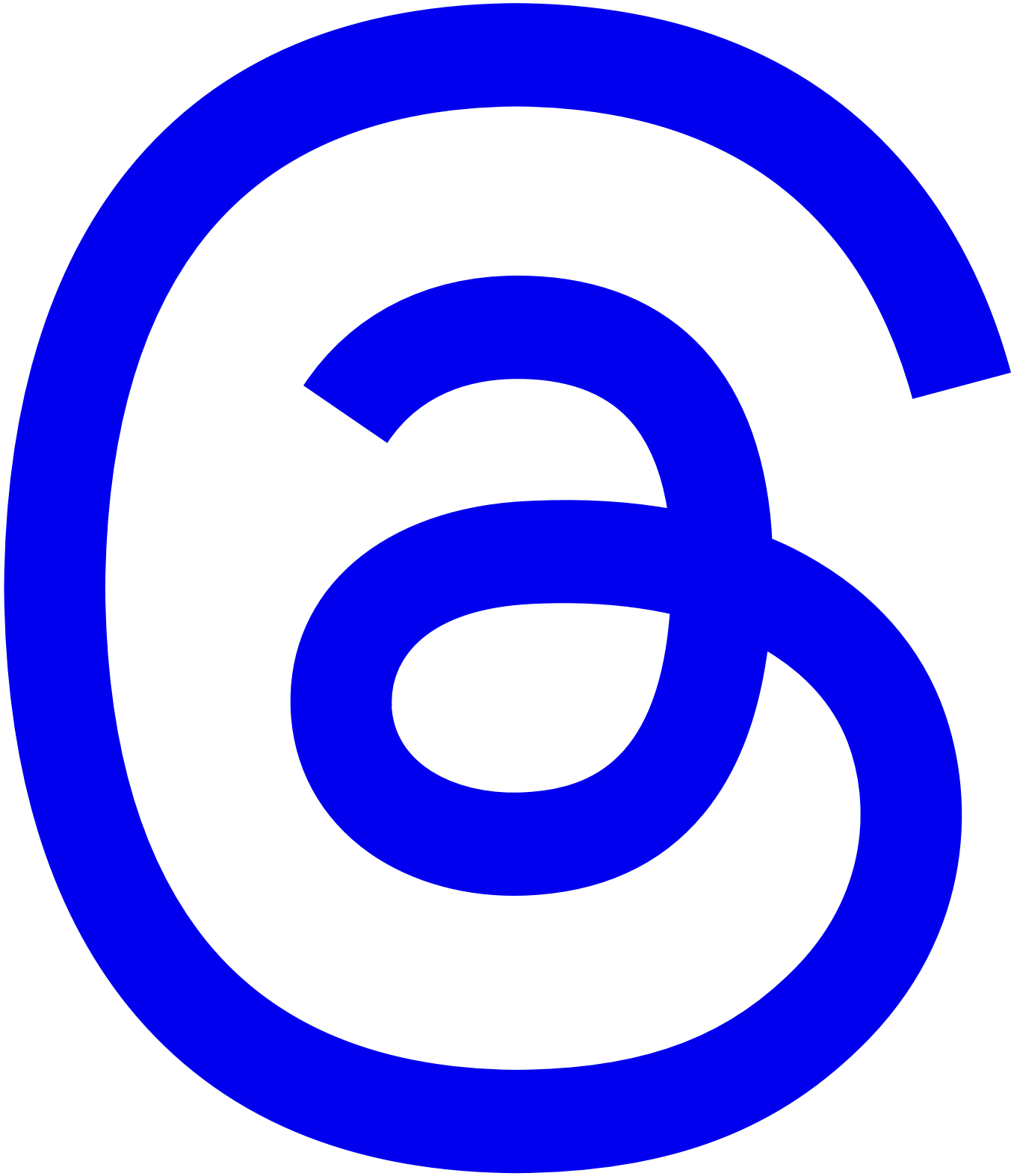
- Contact Us
- Tech Support Forums
- FAQs
- Shipping & Returns
- Freebies
- Terms of Service
- Privacy & Legal
- Website Accessibility
- LLMs.txt

- About Us
- Press
- Educators
- Distributors
- Jobs
- Gift Cards

"Premature optimization is the root of all evil"
Donald Knuth







A Minority and Woman-owned Business Enterprise (M/WBE)