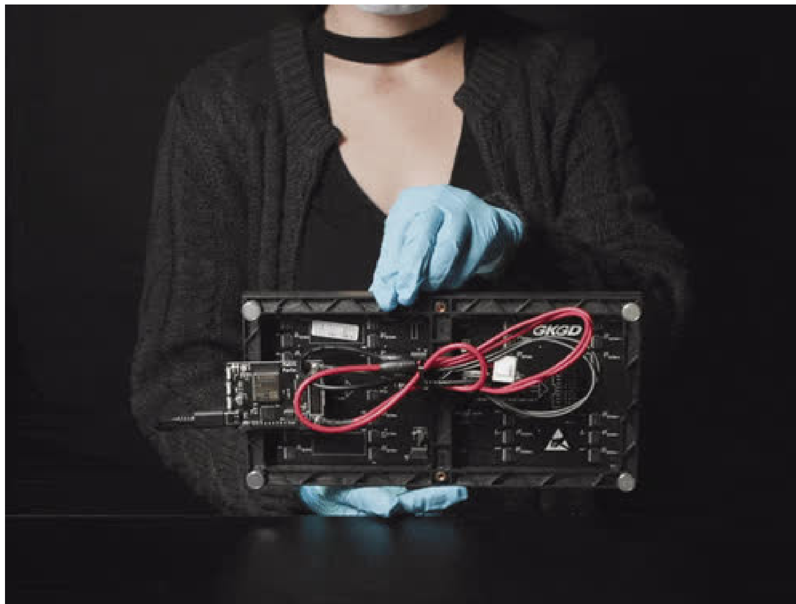
Adafruit MatrixPortal M4

Created by Melissa LeBlanc-Williams



Last updated on 2021-08-09 02:02:07 PM EDT

Guide Contents

Guide Contents 2
Overview 5
Pinouts 9
- Microcontroller and Flash 9
- WiFi 9
- HUB75 Connector 10
- RGB Matrix Power 10
- Sensors 11
- Stemma QT Connector 11
- Reset Pin 11
- Debugging Interface 12
- Serial UART Pins 12
- Analog Connector/Pins 12
- Power Pins 13
- Status LED and NeoPixel 13
- USB-C Connector 13
- Buttons 14
- Address E Line Jumper 14

Prep the MatrixPortal 16
- Power Prep 16
- Power Terminals 17
- Panel Power 17
- Board Connection 19

LED Matrix Diffuser 21
- LED Diffusion Acrylic 21
- Measure and Cut the Plastic 21
- Uglu Dashes 24
- Stand 25

Install CircuitPython 28
- Set up CircuitPython Quick Start! 28
- Further Information 28

What is CircuitPython? 30
CircuitPython is based on Python 30
- Why would I use CircuitPython? 30

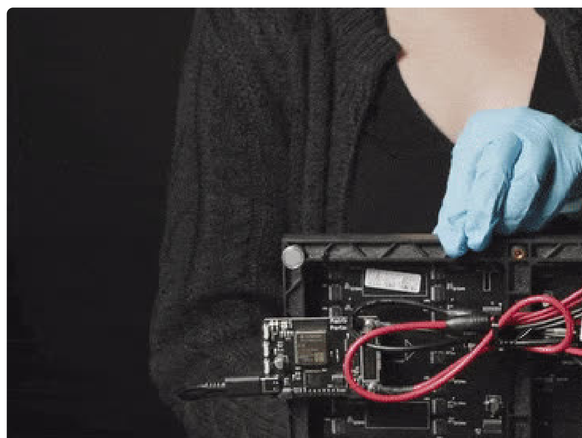
CircuitPython Setup 32
Adafruit CircuitPython Bundle 32
Internet Connect! 33
What's a secrets file? 33
Connect to WiFi 33
Requests 38
- HTTP GET with Requests 40
- HTTP POST with Requests 41
- Advanced Requests Usage 42

WiFi Manager 43
MatrixPortal Library Overview 47
- Network Branch 47
  - WiFi Module 48
  - Network Module 48
- Graphics Branch 48
  - Matrix Module 48
  - Graphics Module 48

MatrixPortal Module 48
Library Demos 49
CircuitPython Pins and Modules 50
CircuitPython Pins 50
import board 50
I2C, SPI, and UART 51
What Are All the Available Names? 52
Microcontroller Pin Names 53
CircuitPython Built-In Modules 54
MatrixPortal Library Docs 55
CircuitPython RGB Matrix Library 56
CircuitPython BLE 57
CircuitPython BLE UART Example 57
On-Board Airlift Co-Processor - No Wiring Needed 57
Update the AirLift Firmware 57
Install CircuitPython Libraries 57
Install the Adafruit Bluefruit LE Connect App 58
Copy and Adjust the Example Program 58
Talk to the AirLift via the Bluefruit LE Connect App 59
Arduino IDE Setup 63
https://adafruit.github.io/arduino-board-index/package_adafruit_index.json 64
Using with Arduino IDE 66
Install SAMD Support 66
Install Adafruit SAMD 67
Install Drivers (Windows 7 & 8 Only) 68
Blink 70
Successful Upload 71
Compilation Issues 72
Manually bootloading 72
Ubuntu & Linux Issue Fix 73
Arduino Libraries 74
Install Libraries 74
Adafruit NeoPixel 74
Adafruit SPIFlash 74
Adafruit Protomatter 75
Adafruit LIS3DH 75
Adafruit GFX 75
WiFiNINA 75
Adafruit ImageReader 76
Adafruit PixelDust 76
Using the Protomatter Library 77
Include Protomatter Library 78
Setting Up Matrix Pin Usage 78
Create the Protomatter Object 78
Begin Protomatter Driver 80
Draw Shapes & Text Using Adafruit GFX 80
Adafruit_GFX is the same library that drives many of our LCD and OLED displays…if you’ve done other graphics projects, you might already be familiar! And if not, we have a separate guide explaining all of the available drawing functions (https://adafru.it/DtY). Most folks can get a quick start by looking at the “simple” and

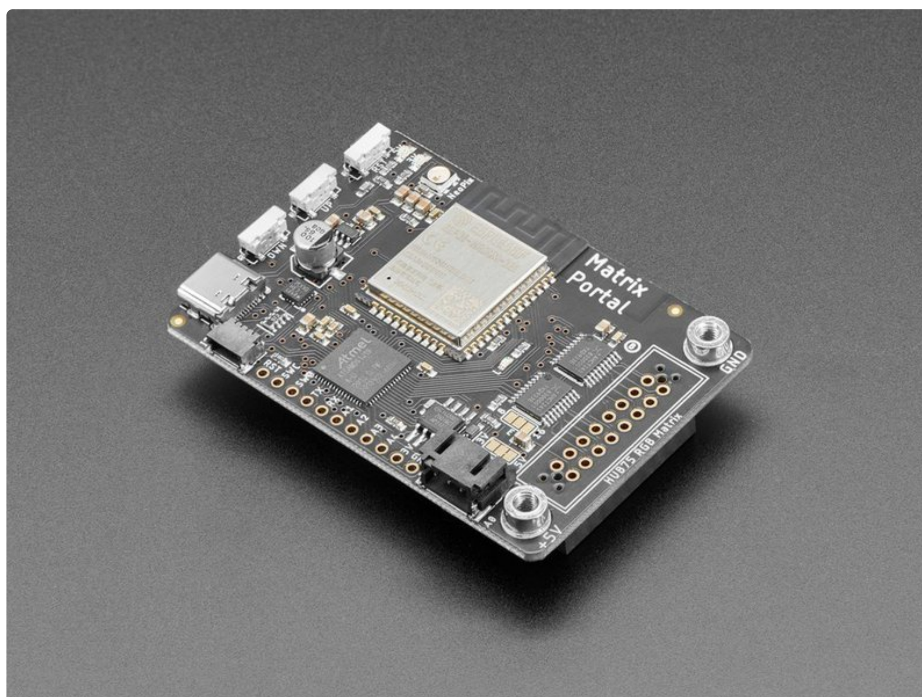
“doublebuffer_scrolltext” examples and tweaking these for their needs. 80
Check Refresh Rate 81
Arduino Sand Demo 83
Protomatter Library 87
Using the Accelerometer 88
Arduino Usage 88
CircuitPython Usage 90
Updating ESP32 Firmware 93
Downloads 94
Files 94
Schematic 94
Fab Print 94

Overview



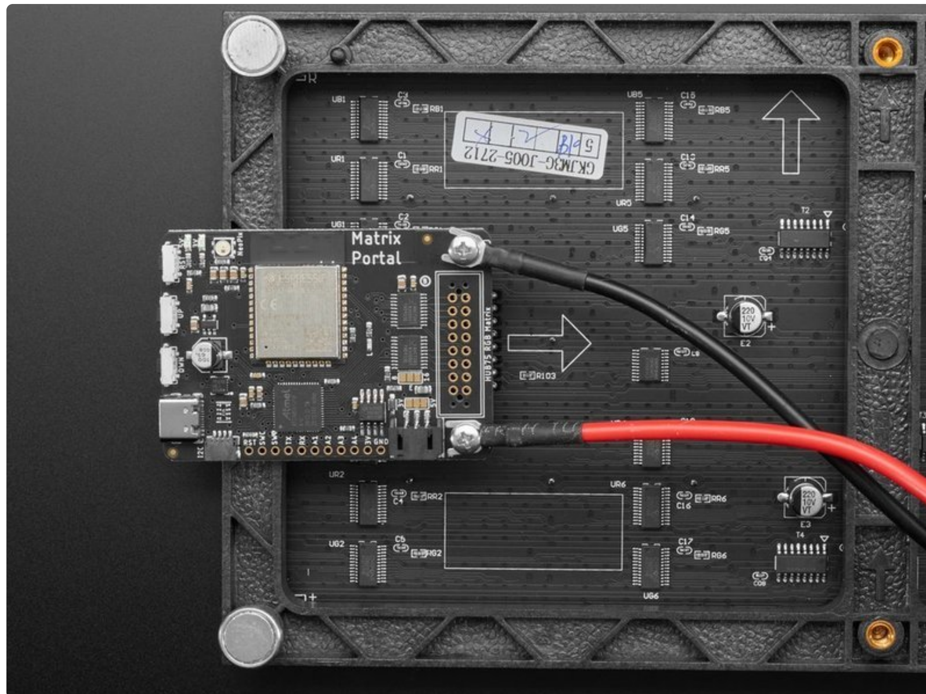
Folks love Adafruit's [wide selection of RGB matrices](https://adafru.it/NAX) (https://adafru.it/NAX) and accessories for making custom colorful LED displays... and Adafruit RGB Matrix Shields and FeatherWings can be quickly soldered together to make the wiring much easier.

But what if we made it *even easier* than that? **Like, no solder, no wiring, just instant plug-and-play?** Dream no more - with the **Adafruit Matrix Portal add-on for RGB Matrices**, there has never been an easier way to create powerful internet-connected LED displays.



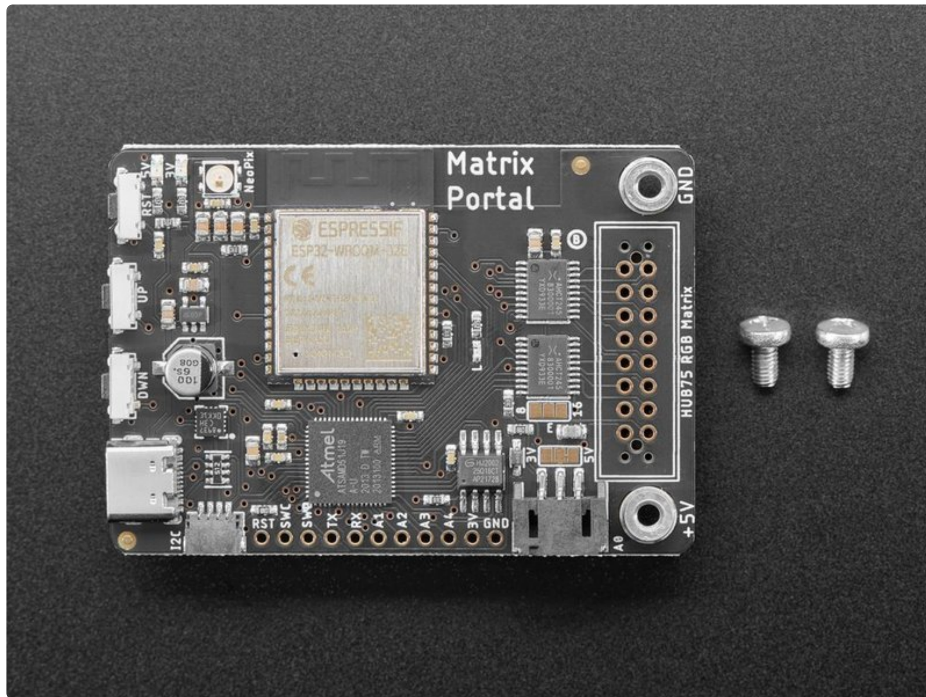
Plug The Matrix Portal directly into the back of [any HUB-75 compatible display (all the ones we stock will work) from 16x32 up to 64x64](https://adafru.it/NAX) (https://adafru.it/NAX)! Use the included screws to attach the power cable to the power plugs with a common screwdriver, then power it with any USB C power supply. (For larger

projects, power the matrices with a separate 5V power adapter)

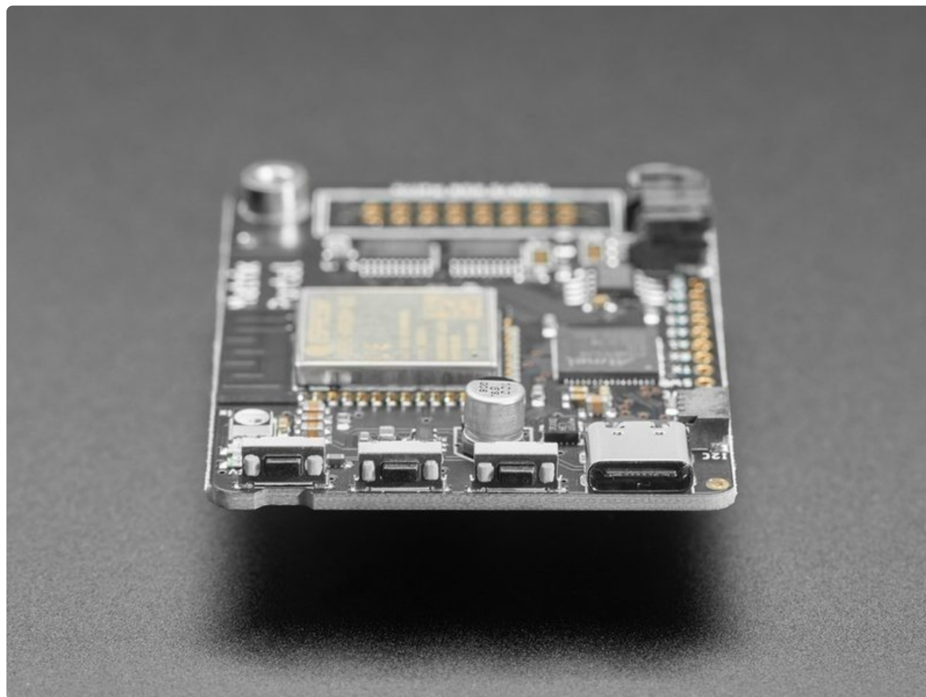


Then code up your project in CircuitPython (https://adafru.it/L7b) or Arduino (https://adafru.it/MNa), the Adafruit Protomatter matrix library works great on the SAMD51 chipset, knowing that you've got the wiring and level shifting all handled. Here's what you get:

- **ATSAMD51J19 Cortex M4 processor**, 512KB flash, 192K of SRAM, with full Arduino or CircuitPython support
- **ESP32 WiFi co-processor** with TLS support and SPI interface to the M4, with full Arduino or CircuitPython support
- **USB Type C** connector for data and power connectivity
- **I2C STEMMA QT connector** for plug-n-play use of any of our STEMMA QT devices or sensors (https://adafru.it/NmD) can also be used with any Grove I2C devices using this adapter cable (https://adafru.it/Ndk)
- **JST 3-pin connector** that also has analog input/output, say for adding audio playback to projects (https://adafru.it/Gpf)
- **LIS3DH accelerometer** for digital sand projects or detecting taps/orientation.
- **GPIO breakouts** including 4 analog outputs with PWM and SPI support for adding other hardware.
- **Address E line jumper** for use with 64x64 matrices (check your matrix to see which pin is used for address E!
- **Two user interface buttons** + one reset button
- **Indicator NeoPixel** and red LED
- **Green power indicator LEDs** for both 3V and 5V power
- **2x10 socket connector** fits snugly into 2x8 HUB75 ports without worrying about 'off by one' errors.

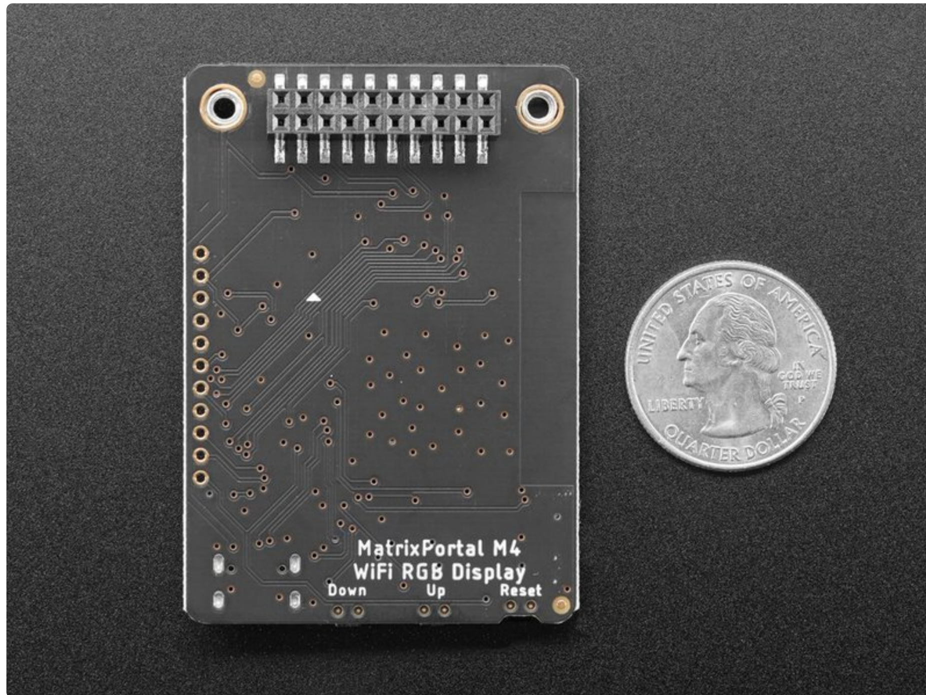


The Matrix Portal uses an ATMEL (Microchip) ATSAMD51J19, and an Espressif ESP32 Wi-Fi coprocessor with TLS/SSL support built-in. The M4 and ESP32 are a great couple - and each bring their own strengths to this board. The SAMD51 M4 has native USB, so it can show up like a disk drive, act as a MIDI or HID keyboard/mouse, and of course bootload and debug over a serial port. It also has DACs, ADC, PWM, and tons of GPIO, so it can handle the high speed updating of the RGB matrix.

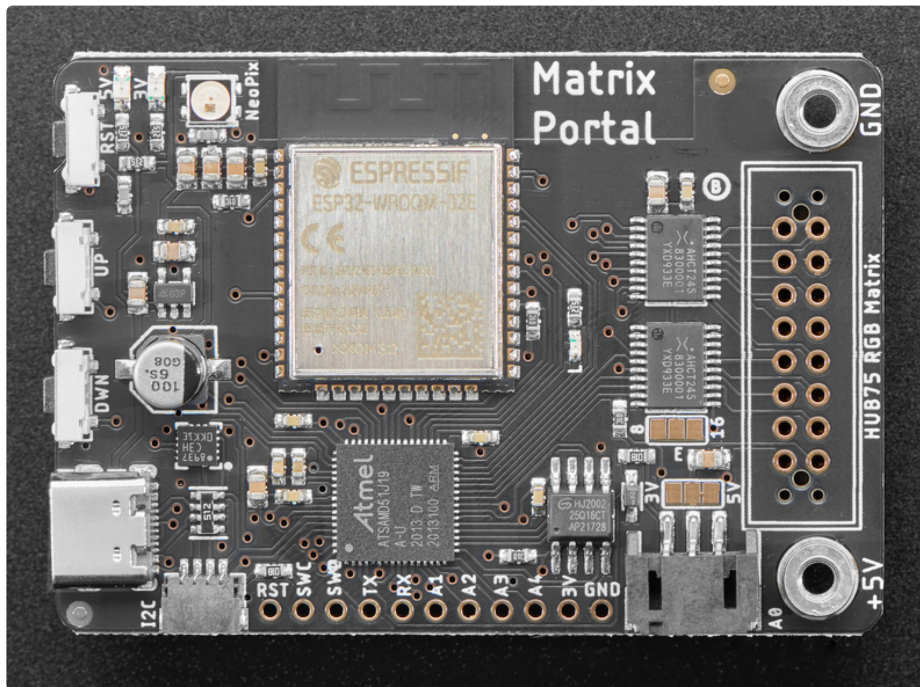


Meanwhile, the ESP32 has secure WiFi capabilities, and plenty of Flash and RAM to buffer sockets. By letting the ESP32 focus on the complex TLS/SSL computation and socket buffering, it frees up the SAMD51

to act as the user interface. You get a great programming experience thanks to the native USB with files available for drag-n-drop, and you don't have to spend a ton of processor time and memory to do SSL encryption/decryption and certificate management. It's the best of both worlds!

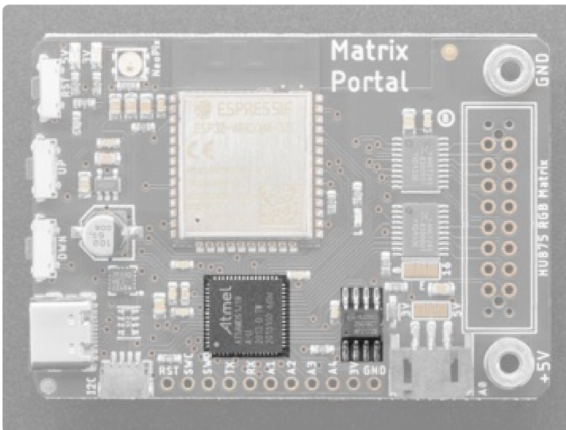


Pinouts



There are so many great features on the Adafruit MatrixPortal M4. Let's take a look at what's available!

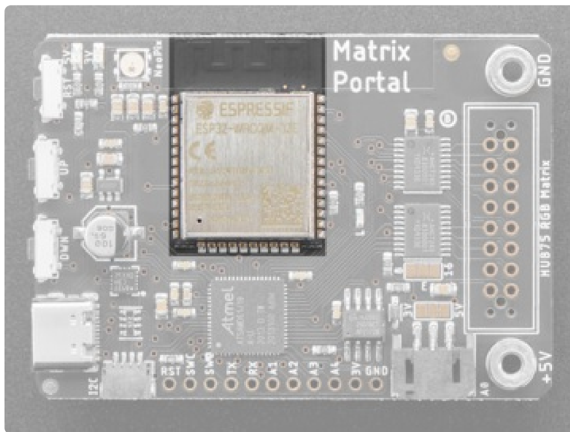
Microcontroller and Flash



The main processor chip is the **ATSAMD51J19 Cortex M4** running at 120MHz with 3.3v logic/power. It has 512KB of Flash and 192KB of RAM.

We also include **2 MB of QSPI Flash** for storing images, sounds, animations, whatever!

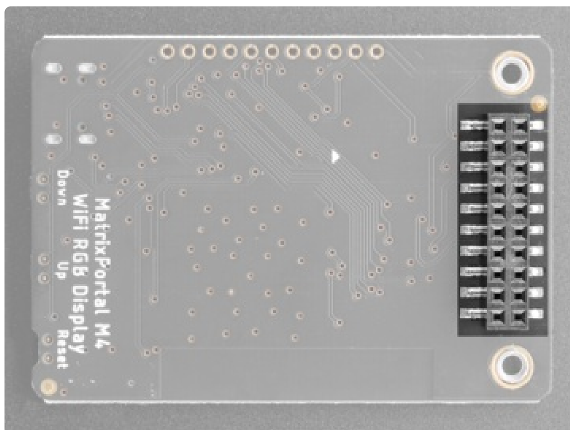
WiFi

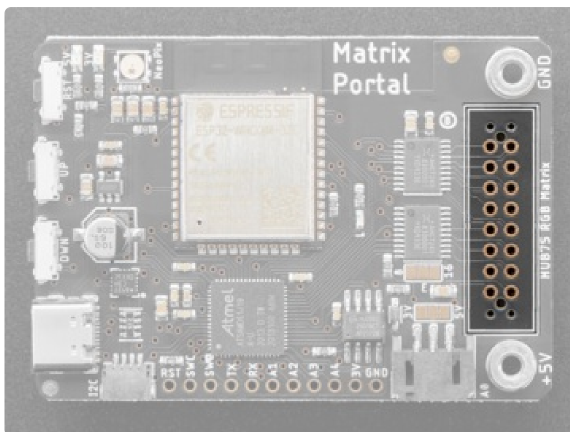


The WiFi capability uses an **Espressif ESP32 Wi-Fi coprocessor** with TLS/SSL support built-in.

The ESP32 uses the SPI port for data, and also uses a CS pin (`board.ESP_CS` or Arduino `33`), Ready/Busy pin (`board.ESP_BUSY` or Arduino `31`), and reset pin (`board.ESP_RESET` or Arduino `30`)

HUB75 Connector

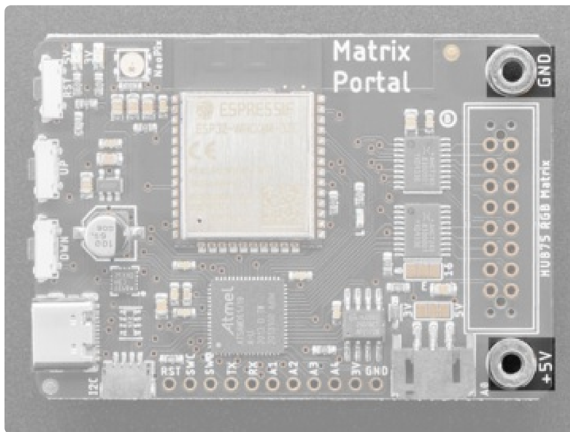




There is a **2x8 pin HUB75** connector on the reverse side that plugs directly into the HUB75 port on your RGB Matrix.

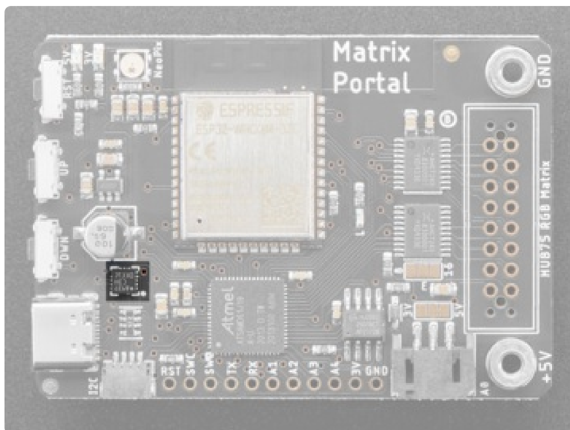
The socket itself is 2x10 so that it fits snug and lined up in a 2x8 IDC socket. Otherwise its easy to get it 'off by one'

RGB Matrix Power



There are **+5V** and **Ground** screw terminals on either side of the HUB75 connector. These provide power to the RGB Matrix.

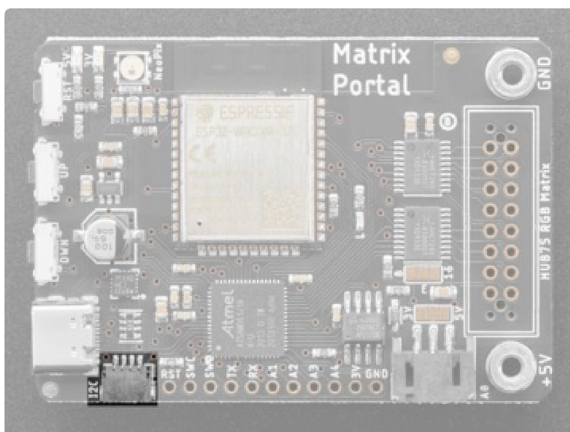
Sensors



The MatrixPortal M4 includes a **LIS3DH Triple-Axis Accelerometer**. The accelerometer is connected via the I2C bus.

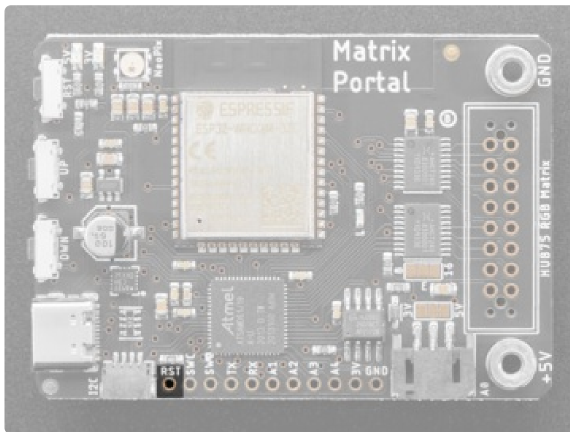
Please note the address of the accelerometer is 0x19 not 0x18 which is the default in our libraries.

Stemma QT Connector



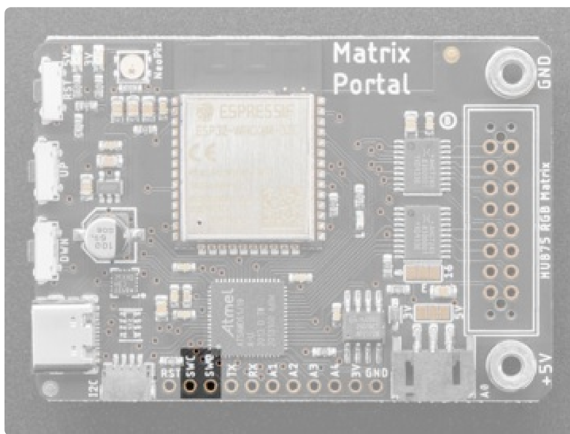
There is a 4-pin **Stemma QT connector** on the left. The I2C has pullups to 3.3V power and is connected to the LIS3DH already.

Reset Pin



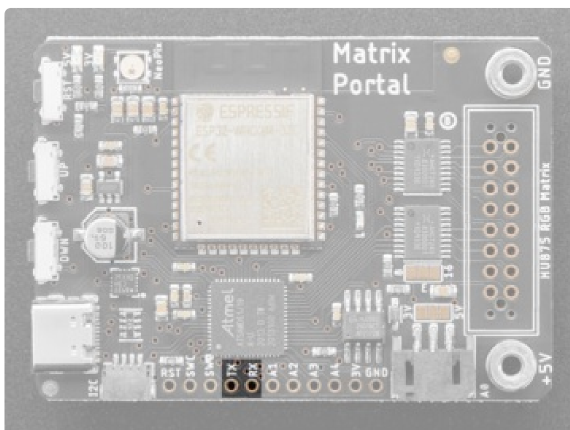
RST is the Reset pin. Tie to ground to manually reset the ATSAMD51, as well as launch the bootloader manually.

Debugging Interface



If you'd like to do more advanced development, trace-debugging, or not use the bootloader, we have the SWD interface exposed.

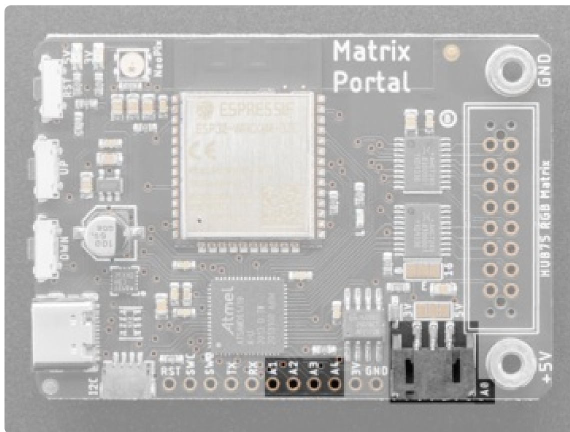
Serial UART Pins



The **TX pin** and **RX pin** are for serial communication with the SAMD51 microcontroller and can be used to connect various peripherals such as a GPS.

The RX pin is attached to `board.RX` and Arduino `0` and the TX pin is attached to `board.TX` and Arduino `1`.

Analog Connector/Pins

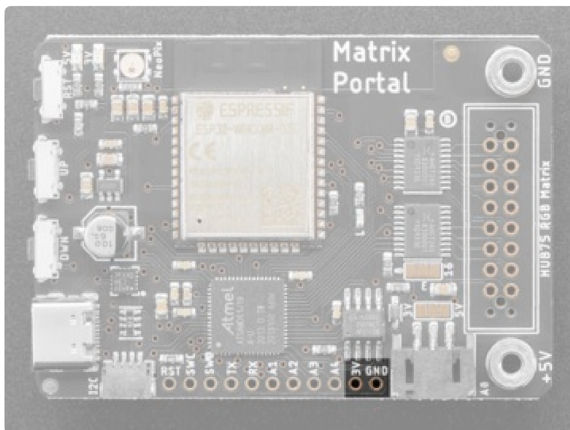


On the bottom side towards the right, there is a connector labeled **A0**. This is a **3-pin JST analog connector** for sensors or NeoPixels, analog output or input

Along the bottom there are also pins labeled `A1` through `A4`.

All of these pins can be used for analog inputs or digital I/O.

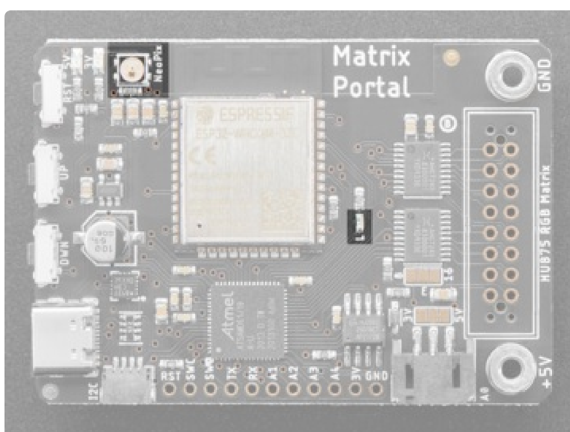
Power Pins



3V is the output from the 3.3V regulator, it can supply 500mA peak.

GND is the common ground for all power and logic.

Status LED and NeoPixel

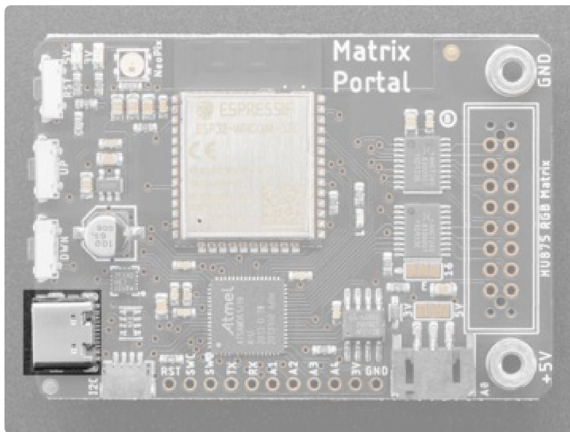


There are two LEDs on the board.

There is the RGB **status NeoPixel** labeled "STATUS". It is connected to `board.NEOPIXEL` or Arduino `4`

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

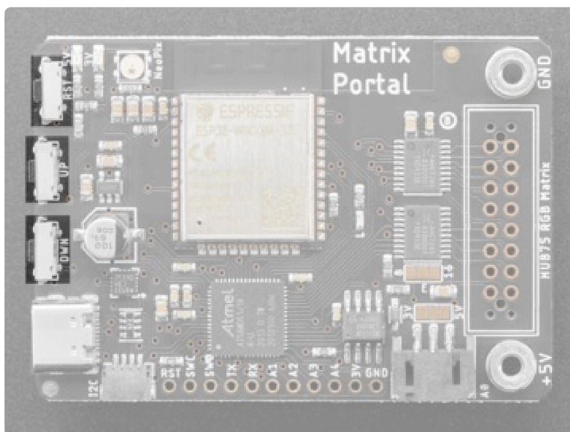
USB-C Connector



There is one USB port on the board.

On the left side, towards the bottom, is a **USB Type C** port, which is used for powering and programming both the board and RGB Matrix.

Buttons



There are three buttons along the left side of the MatrixPortal M4.

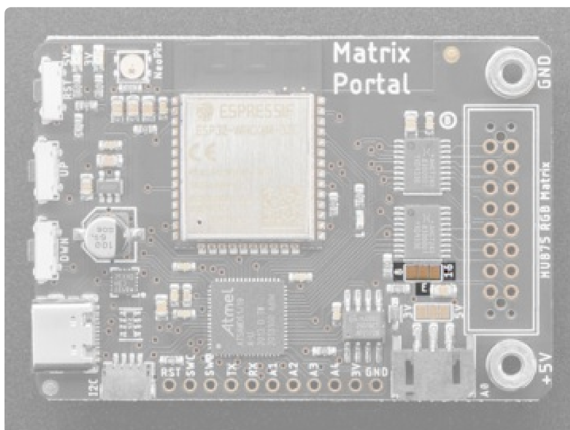
The **reset button** is located in the top position. Click it once to re-start your firmware. Click twice to enter bootloader mode.

The **up button** is located in the middle and is attached to `board.BUTTON_UP` and Arduino `2`.

The **down button** is located on the bottom and is attached to `board.BUTTON_DOWN` and Arduino `3`.

The up and down buttons do not have any pull-up resistors connected to them and pressing either of them pulls the input low.

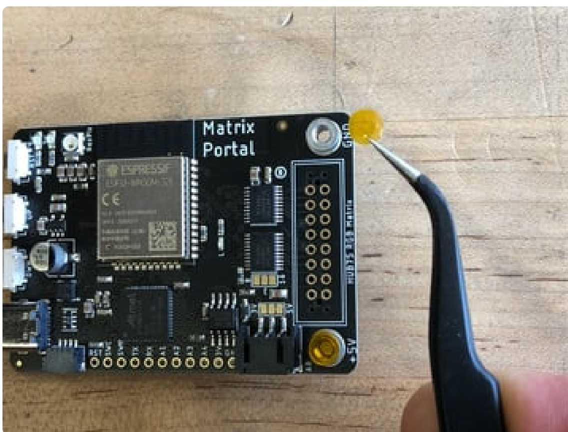
Address E Line Jumper



This jumper is used for use with **64x64 matrices** and is either connected to pin 8 or pin 16 of the HUB75 connector. Check your matrix to see which pin is used for address E.

You can close the jumper by using your soldering iron to melt a blob of solder on the bottom solder jumper so the middle pad is 'shorted' to 8. *(This is compatible with 64x64 matrices in the Adafruit store. For 64x64 matrices from other sources, you might need to use 16 instead, check the datasheet of your display.)*

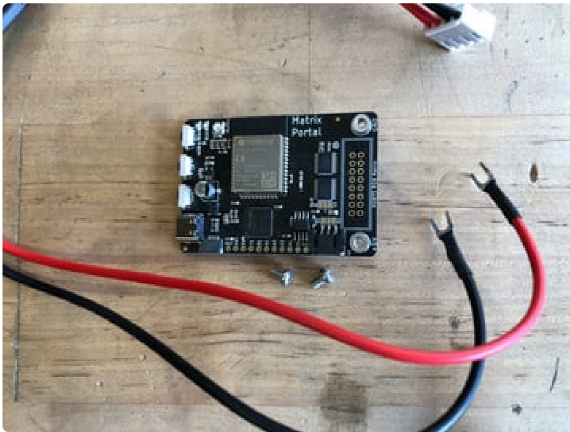
Prep the MatrixPortal

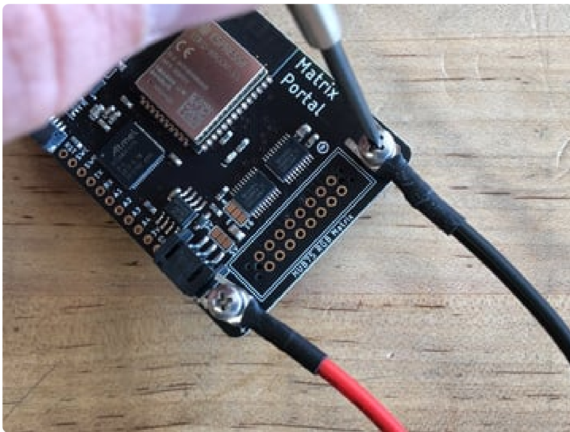


Power Prep

The MatrixPortal supplies power to the matrix display panel via two standoffs. These come with protective tape applied (part of our manufacturing process) which MUST BE REMOVED!

Use some tweezers or a fingernail to remove the two amber circles.

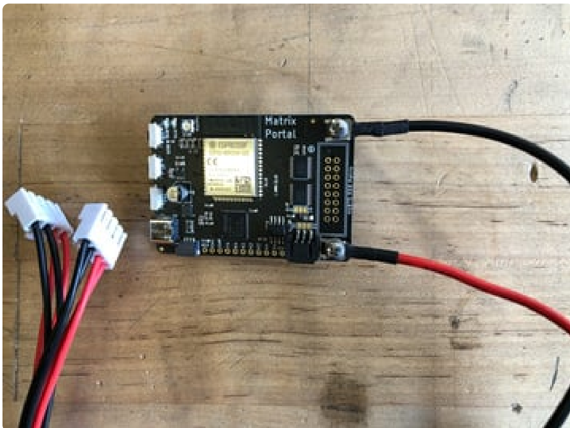


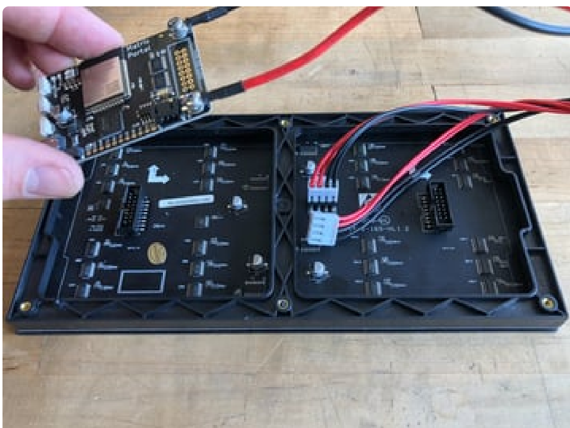


Power Terminals

Next, screw in the spade connectors to the corresponding standoff.

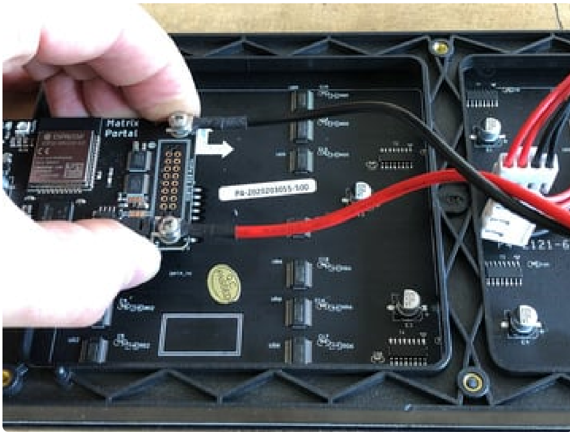
- **red** wire goes to **+5V**
- **black** wire goes to **GND**





Panel Power

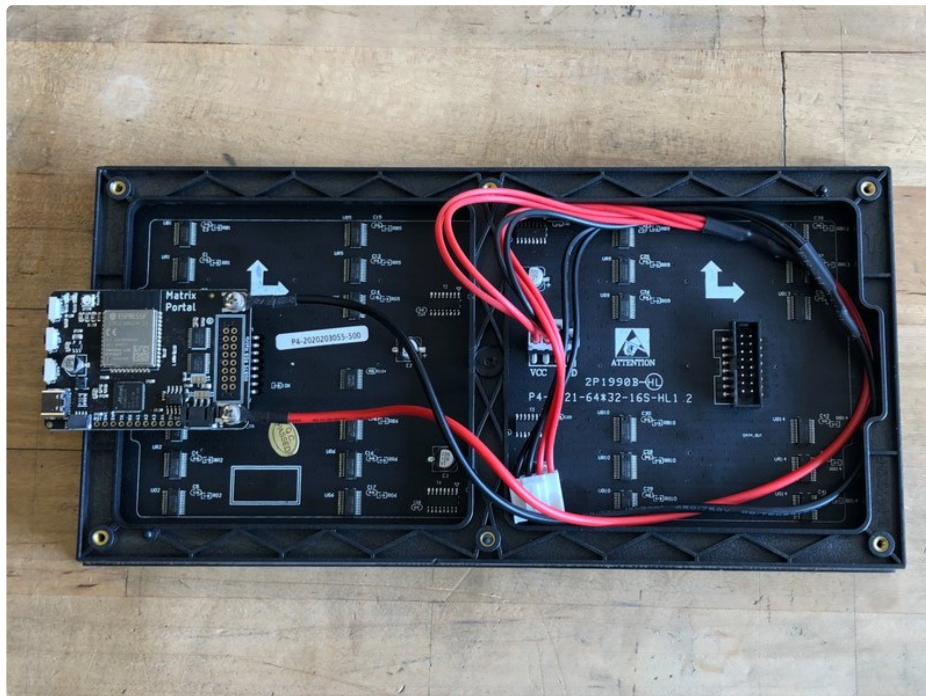
Plug either one of the four-conductor power plugs into the power connector pins on the panel. The plug can only go in one way, and that way is marked on the board's silkscreen.



Board Connection

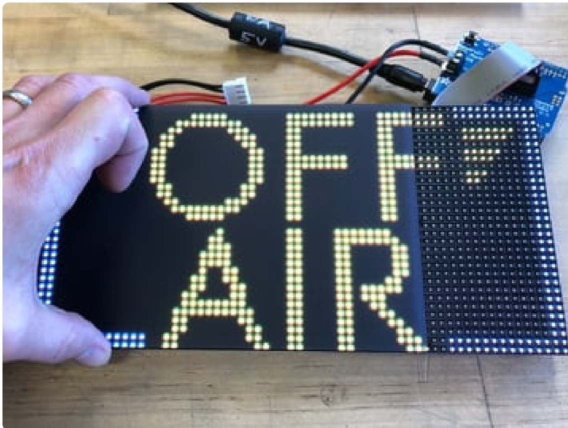
Now, plug the board into the left side shrouded 8x2 connector as shown. The orientation matters, so take a moment to confirm that the **white indicator arrow on the matrix panel is oriented pointing up and right** as seen here and the MatrixPortal overhangs the edge of the panel when connected. This allows you to use the edge buttons from the front side.

Check nothing is impeding the board from plugging in firmly. If there's a plastic nub on the matrix that's keeping the Portal from sitting flat, cut it off with diagonal cutters



For info on adding LED diffusion acrylic, see the page LED Matrix Diffuser.

LED Matrix Diffuser



LED Diffusion Acrylic

You can add an LED diffusion acrylic faceplate (https://adafru.it/MEF) to the your LED matrix display. (Pictured here with the ON AIR project (https://adafru.it/MPE))

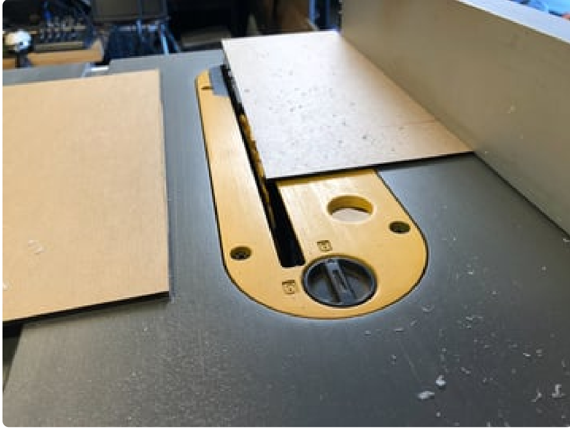
This can help protect the LEDs as well as enhance the look of the sign both indoors and out by reducing glare and specular highlights of the plastic matrix grid.

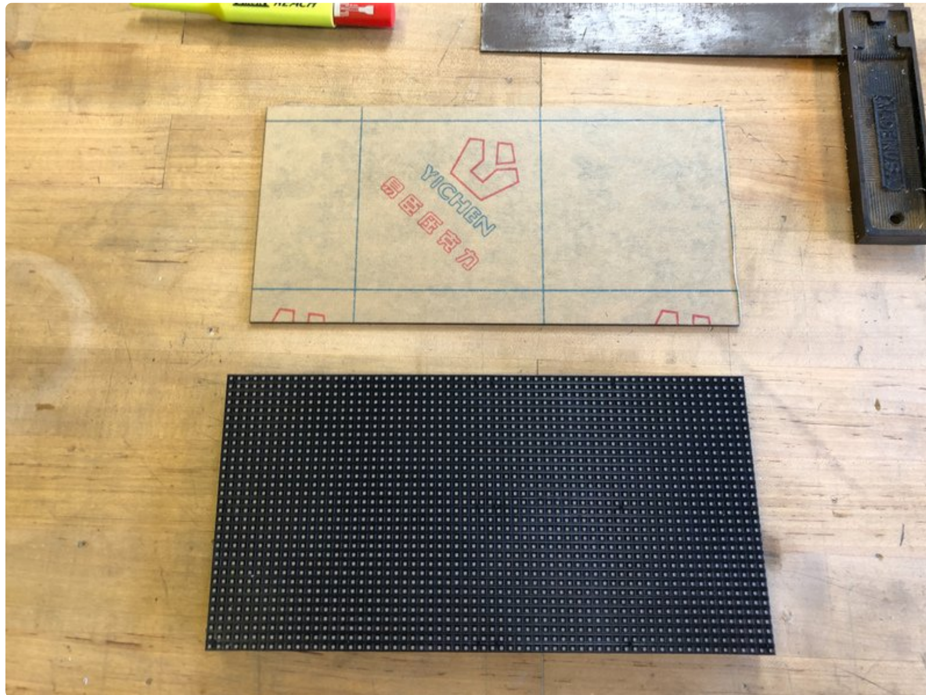
Measure and Cut the Plastic

You can use the sign to measure and mark cut lines on the paper backing of the acrylic sheet.

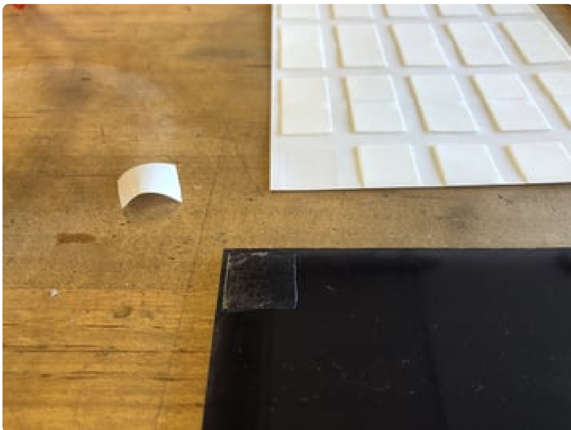
Then, use a tablesaw or bandsaw with a fine toothed blade and a guide or sled to make the cuts.

Note: it is possible to score and snap acrylic, but it can be very tricky to get an even snap without proper clamping.




YICHEN

Peel away the paper backing from both sides and set the acrylic onto your matrix display.

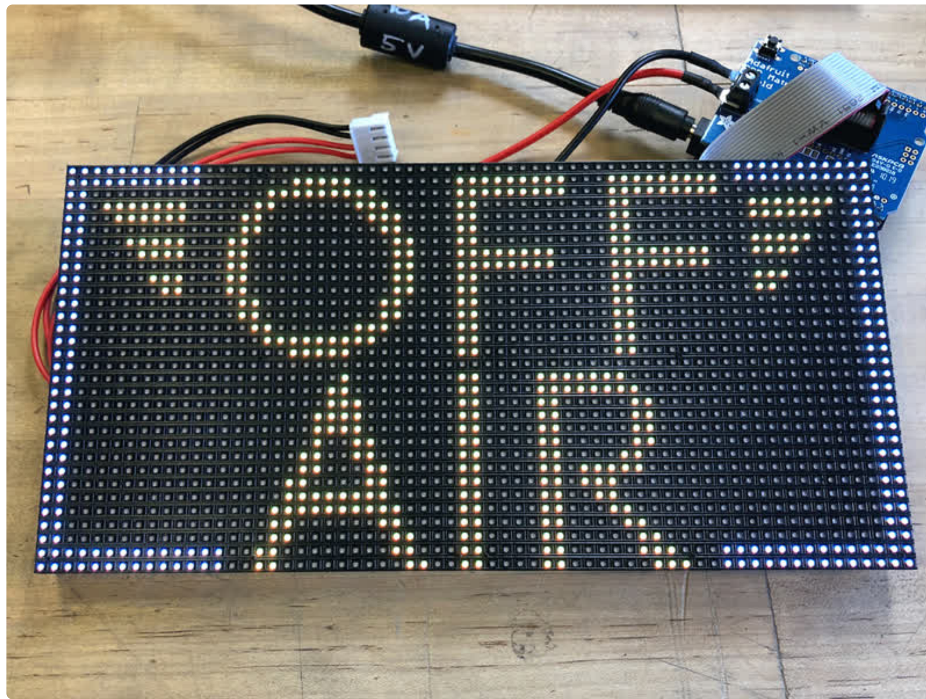


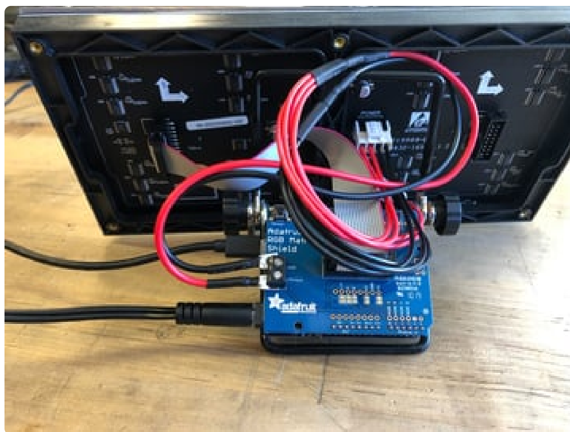
Uglu Dashes

The best method we've found for adhering acrylic to the matrix display is to use Uglu Dashes clear adhesive rectangles from Pro Tapes (https://adafru.it/NcP). They are incredibly strong (although can be removed if necessary), easy to apply, and are invisible once attached.

Use one at each corner and one each at the halfway point of the long edges, then press the acrylic and matrix panel together for about 20 seconds.

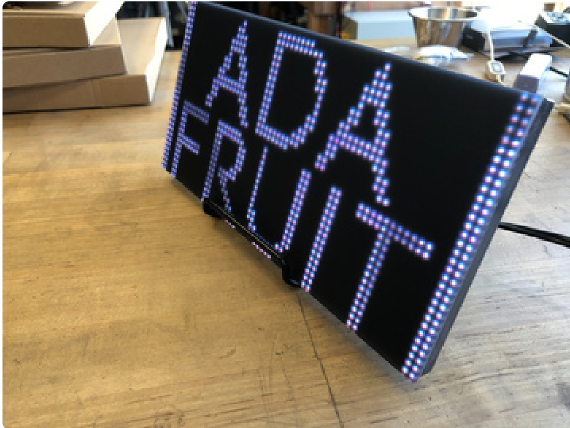
Here you can see the impact of using the diffusion acrylic. (Pictured here with the ON AIR sign project)





Stand

A very simple and attractive way to display your matrix is with the adjustable bent-wire stand (https://adafru.it/MPF).



Alternately, you can use a frame, [3D printed brackets](https://adafru.it/MZf) (https://adafru.it/MZf), tape, glue, or even large binder clips to secure the acrylic to the sign and then mount it on on a wall, shelf, or display cabinet.

[These mini-magnet feet](https://adafru.it/MZA) (https://adafru.it/MZA) can be used to stick the sign to a ferrous surface.

Install CircuitPython

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

Set up CircuitPython Quick Start!

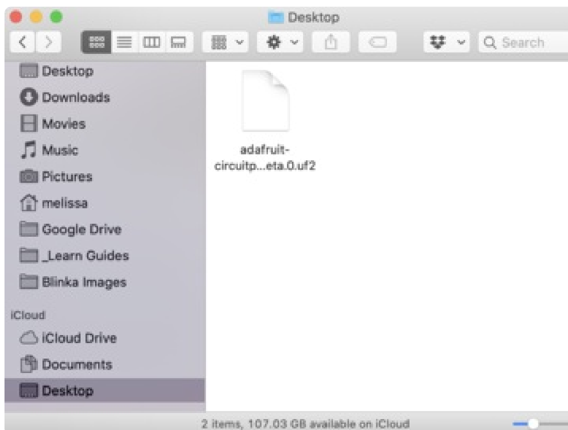
Follow this quick step-by-step for super-fast Python power :)

https://adafru.it/Nte

https://adafru.it/Nte

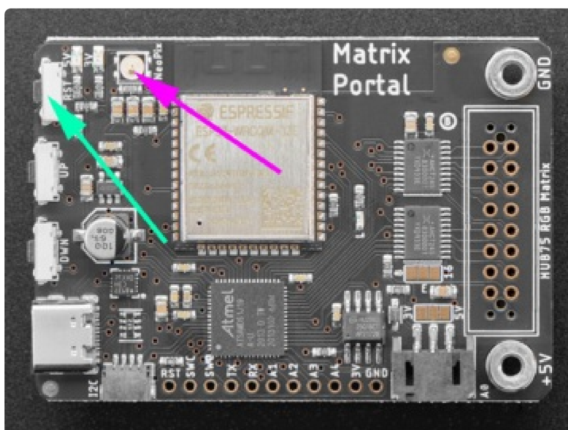
Further Information

For more detailed info on installing CircuitPython, check out [Installing CircuitPython](https://adafru.it/Amd) (https://adafru.it/Amd).



Click the link above and download the latest UF2 file.

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

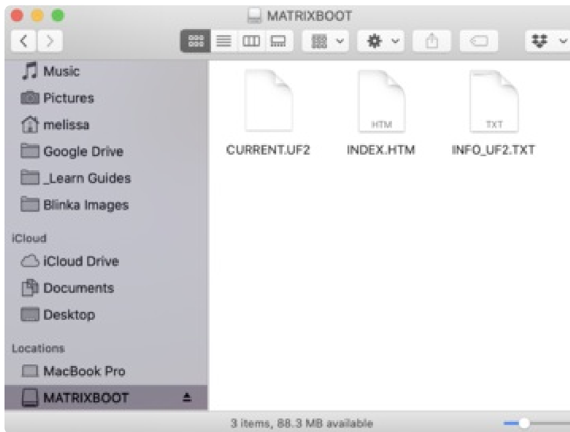


Plug your MatrixPortal M4 into your computer using a known-good USB cable.

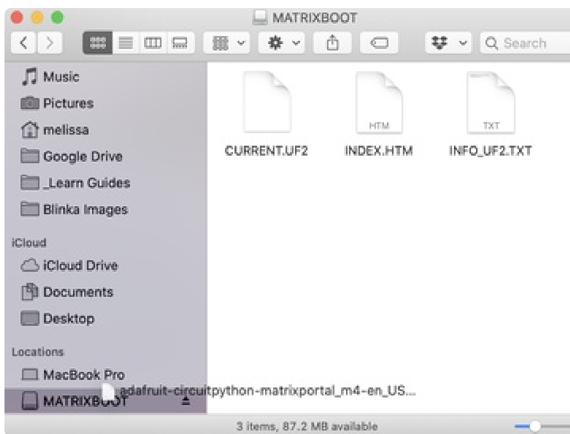
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Double-click the **Reset** button (indicated by the green arrow) on your board, and you will see the NeoPixel RGB LED (indicated by the magenta arrow) turn green. If it turns red, check the USB cable, try another USB port, etc.

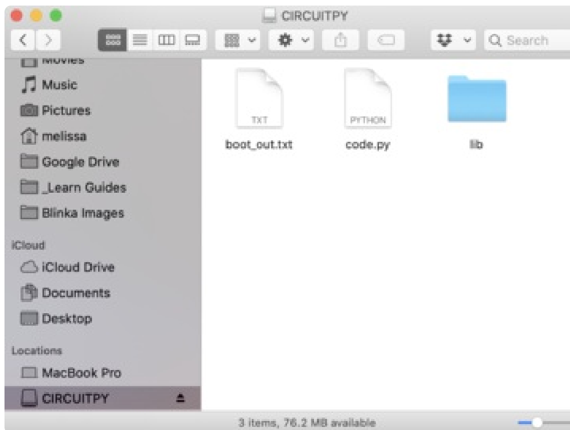
If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!



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



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



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

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

What is CircuitPython?

CircuitPython is a programming language designed to simplify experimenting and learning to program on low-cost microcontroller boards. It makes getting started easier than ever with no upfront desktop downloads needed. Once you get your board set up, open any text editor, and get started editing code. It's that simple.

CircuitPython is based on Python

Python is the fastest growing programming language. It's taught in schools and universities. It's a high-level programming language which means it's designed to be easier to read, write and maintain. It supports modules and packages which means it's easy to reuse your code for other projects. It has a built in interpreter which means there are no extra steps, like *compiling*, to get your code to work. And of course, Python is Open Source Software which means it's free for anyone to use, modify or improve upon.

CircuitPython adds hardware support to all of these amazing features. If you already have Python knowledge, you can easily apply that to using CircuitPython. If you have no previous experience, it's really simple to get started!

Why would I use CircuitPython?

CircuitPython is designed to run on microcontroller boards. A microcontroller board is a board with a

microcontroller chip that's essentially an itty-bitty all-in-one computer. The board you're holding is a microcontroller board! CircuitPython is easy to use because all you need is that little board, a USB cable, and a computer with a USB connection. But that's only the beginning.

Other reasons to use CircuitPython include:

- **You want to get up and running quickly.** Create a file, edit your code, save the file, and it runs immediately. There is no compiling, no downloading and no uploading needed.
- **You're new to programming.** CircuitPython is designed with education in mind. It's easy to start learning how to program and you get immediate feedback from the board.
- **Easily update your code.** Since your code lives on the disk drive, you can edit it whenever you like, you can also keep multiple files around for easy experimentation.
- **The serial console and REPL.** These allow for live feedback from your code and interactive programming.
- **File storage.** The internal storage for CircuitPython makes it great for data-logging, playing audio clips, and otherwise interacting with files.
- **Strong hardware support.** There are many libraries and drivers for sensors, breakout boards and other external components.
- **It's Python!** Python is the fastest-growing programming language. It's taught in schools and universities. CircuitPython is almost-completely compatible with Python. It simply adds hardware support.

This is just the beginning. CircuitPython continues to evolve, and is constantly being updated. We welcome and encourage feedback from the community, and we incorporate this into how we are developing CircuitPython. That's the core of the open source concept. This makes CircuitPython better for you and everyone who uses it!

CircuitPython Setup

To use all the amazing features of your MatrixPortal M4 with CircuitPython, you must first install a number of libraries. This page covers that process.

Adafruit CircuitPython Bundle

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

https://adafru.it/ENC

https://adafru.it/ENC

Download the **adafruit-circuitpython-bundle-version-mpy-*.zip** bundle zip file, and unzip a folder of the same name. Inside you'll find a **lib** folder. The entire collection of libraries is too large to fit on the **CIRCUITPY** drive. Instead, add each library as you need it, this will reduce the space usage but you'll need to put in a little more effort.

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

- **adafruit_matrixportal** - this library is the main library used with the MatrixPortal.
- **adafruit_portalbase** - This is the base library that adafruit_matrixportal is built on top of.
- **adafruit_esp32spi** - this is the library that gives you internet access via the ESP32 using (you guessed it!) SPI transport. You need this for anything Internet
- **neopixel** - for controlling the onboard neopixel
- **adafruit_bus_device** - low level support for I2C/SPI
- **adafruit_requests** - this library allows us to perform HTTP requests and get responses back from servers. GET/POST/PUT/PATCH - they're all in here!
- **adafruit_fakerequests.mpy** - This library allows you to create fake HTTP requests by using local files.
- **adafruit_io** - this library helps connect the PyPortal to our free data logging and viewing service
- **adafruit_bitmap_font** - we have fancy font support, and it's easy to make new fonts. This library reads and parses font files.
- **adafruit_display_text** - not surprisingly, it displays text on the screen
- **adafruit_lis3dh** - this library is used for the onboard accelerometer to detect the orientation of the MatrixPortal

Internet Connect!

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

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

What's a secrets file?

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

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

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

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

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

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

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

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

Connect to WiFi

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

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

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

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

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

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

print("ESP32 SPI webclient test")

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


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

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

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

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

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

requests.set_socket(socket, esp)

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

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

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

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

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

print("Done!")
```

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

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

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

COM61 - PuTTY

```
ESP32 SPI webclient test
ESP32 found and in idle mode
Firmware vers. bytearray(b'1.2.2\x00')
MAC addr: ['0x1', '0x5c', '0xd', '0x33', '0x4f', '0xc4']
        MicroPython-d45f8a              RSSI: -44
        adafruit_tw             RSSI: -63
        FiOS-QOG1B              RSSI: -63
        adafruit                RSSI: -71
        AP819           RSSI: -73
        FiOS-K57GI              RSSI: -74
        AP819           RSSI: -77
        linksys_SES_2868                RSSI: -79
        linksys_SES_2868                RSSI: -79
        FiOS-K57GI              RSSI: -83
Connecting to AP...
Connected to adafruit   RSSI: -65
My IP address is 10.0.1.54
IP lookup adafruit.com: 104.20.38.240
Ping google.com: 30 ms
Fetching text from http://wifitest.adafruit.com/testwifi/index.html
----------------------------------------
This is a test of the CC3000 module!
If you can read this, its working :)
----------------------------------------

Fetching json from http://api.coindesk.com/v1/bpi/currentprice/USD.json
----------------------------------------
{'time': {'updated': 'Feb 27, 2019 03:11:00 UTC', 'updatedISO': '2019-02-2
7T03:11:00+00:00', 'updateduk': 'Feb 27, 2019 at 03:11 GMT'}, 'disclaimer'
: 'This data was produced from the CoinDesk Bitcoin Price Index (USD). Non
-USD currency data converted using hourly conversion rate from openexchang
erates.org', 'bpi': {'USD': {'code': 'USD', 'description': 'United States
Dollar', 'rate_float': 3832.74, 'rate': '3,832.7417'}}}
----------------------------------------
Done!
```

In order, the example code...

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Requests

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

HTTP GET with Requests

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

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

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

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

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

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

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

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

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

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

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

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

HTTP POST with Requests

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

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

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

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

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

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

Advanced Requests Usage

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

WiFi Manager

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

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

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

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

```
from adafruit_esp32spi import adafruit_esp32spi
from adafruit_esp32spi import adafruit_esp32spi_wifimanager

print("ESP32 SPI webclient test")

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

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

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

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

counter = 0

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

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

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

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

- aio_username
- aio_key

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

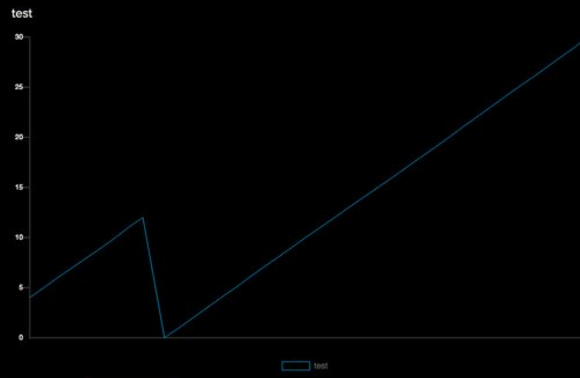
```
# This file is where you keep secret settings, passwords, and tokens!
# If you put them in the code you risk committing that info or sharing it

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

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

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

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

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


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

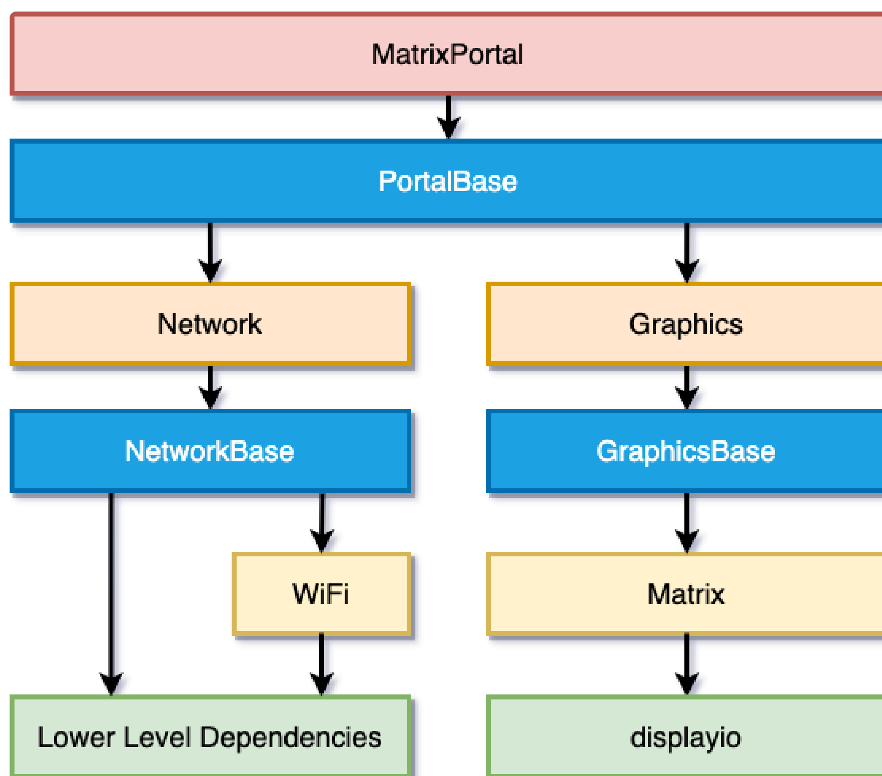
MatrixPortal Library Overview

The MatrixPortal library was inspired by the PyPortal library, but a slightly different approach was taken. Rather than having everything in a single module, it was divided into layers. The reason for having different layers is you can use lower layers if you want more control and better memory usage.

The main library now piggyback's on top of the base library. The base library was named PortalBase which is split up into 3 components. The main base, the GraphicsBase, and the NetworkBase. In the diagram, you can see these components represented in blue.

We also have a library for lower-level control of just the RGB Matrix (https://adafru.it/L7b), but it doesn't have integrated WiFi access so we recommend using the MatrixPortal library.

Here is the way it is logically laid out with dependencies. The MatrixPortal library is comprised of the top layer, the Network and Graphics layers, and the WiFi and Matrix layers in the diagram.



There are two main branches of dependencies related to Network Functionality and Graphics functionality. The **MatrixPortal** library ties them both together and allows easier coding, but at the cost of more memory usage and less control. We'll go through each of the classes starting from the bottom and working our way up the diagram starting with the Network branch.

Network Branch

The network branch contains all of the functionality related to connecting to the internet and retrieving data. You will want to use this branch if your project need to retrieve any data that is not stored on the device itself.

WiFi Module

The WiFi module is responsible for initializing the hardware libraries, controlling the status NeoPixel colors, and initializing the WiFi manager. You would want to use this library if you only wanted to handle the automatic initialization of hardware and connection to WiFi and didn't need any other functionality.

Network Module

The network module has many convenience functions for making network calls. It handles a lot of things from automatically establishing the connection to getting the time from the internet, to getting data at certain URLs. This is one of the largest of the modules as there is a lot of functionality packed into this.

Graphics Branch

This branch is a lot lighter than the Network Branch because so much of the functionality is built into CircuitPython and displayio.

Matrix Module

The matrix module is responsible for detecting and initializing the matrix through the CircuitPython rgbmatrix and framebufferio modules. It currently supports the MatrixPortal M4 and Metro M4 with RGB Matrix Shield. If you just wanted to initialize the matrix, you could use this module. If you would like to go lower level than this and use the rgbmatrix and framebufferio libraries directly, be sure to check out the guide RGB LED Matrices with CircuitPython (https://adafru.it/L7b).

Graphics Module

This module will initialize the Matrix through the matrix module. The main purpose of this module was to add any graphics convenience functions in such as displaying a background easily.

MatrixPortal Module

The MatrixPortal module is top level module and will handle initializing everything below it. Using this module is very similar to using the PyPortal library. The main differences are:

- Text labels are added after the module is initialized.
- Text labels can either be scrolling or static.
- There are more Adafruit IO functions

Library Demos

The MatrixPortal library has been used in a number of projects. Here are a few of them with guides available.

- RGB Matrix Automatic YouTube ON AIR Sign (https://adafru.it/MPE)
- Network Connected RGB Matrix Clock (https://adafru.it/NA-)
- Bitcoin Value RGB Matrix Display (https://adafru.it/NB0)
- Weather Display Matrix (https://adafru.it/NB1)
- Custom Scrolling Quote Board Matrix Display (https://adafru.it/NB2)
- Halloween Countdown Display Matrix (https://adafru.it/NB5)
- Moon Phase Clock for Adafruit Matrix Portal (https://adafru.it/NB7)

CircuitPython Pins and Modules

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

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

CircuitPython Pins

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

`import board`

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

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

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

Here is the output for the QT Py.

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

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

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

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

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

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

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

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

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

I2C, SPI, and UART

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

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

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

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

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

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

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

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

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

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

What Are All the Available Names?

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

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

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

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

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

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

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

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

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

Microcontroller Pin Names

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

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

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

CircuitPython Built-In Modules

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

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

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

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

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

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

MatrixPortal Library Docs

[MatrixPortal Library Docs](https://adafru.it/Oa5) (https://adafru.it/Oa5)

CircuitPython RGB Matrix Library

CircuitPython RGB Matrix Library (https://adafru.it/L7b)

CircuitPython BLE

CircuitPython BLE UART Example

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

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

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

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

On-Board Airlift Co-Processor - No Wiring Needed

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

Update the AirLift Firmware

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

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

https://adafru.it/FWs

https://adafru.it/FWs

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

Install CircuitPython Libraries

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

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

Install these libraries from the bundle:

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

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

Install the Adafruit Bluefruit LE Connect App

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

Copy and Adjust the Example Program

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

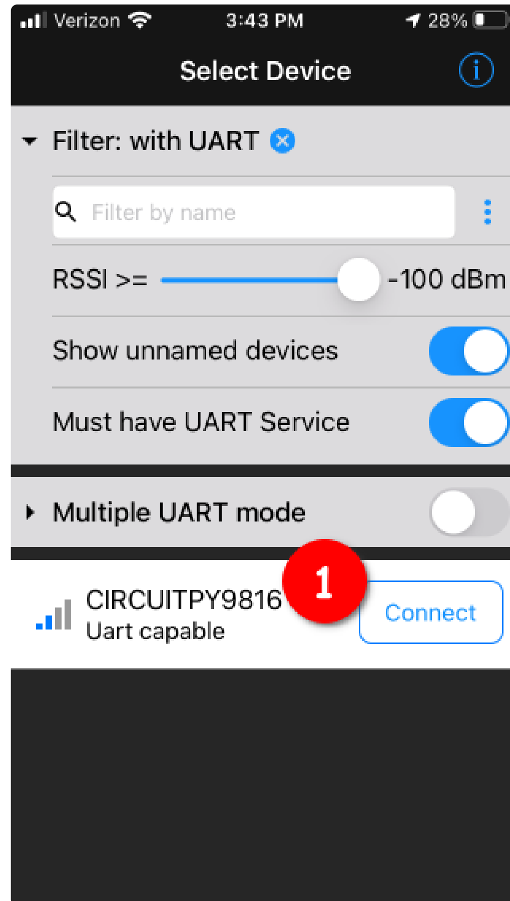
TAKE NOTE: Adjust the program as needed to suit the AirLift board you have. Comment and uncomment lines 12-39 below as necessary.

```
import board

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

from adafruit_airlift.esp32 import ESP32

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

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

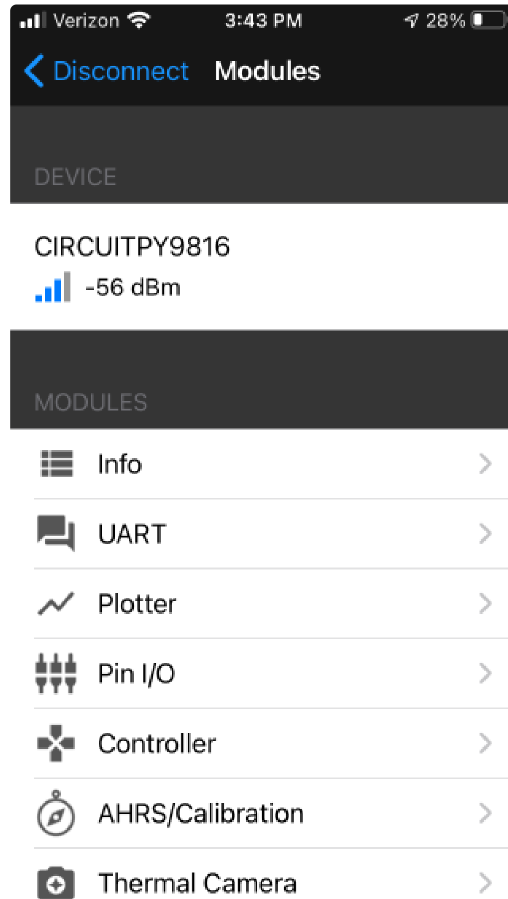
```
# esp32 = ESP32(tx=board.TX, rx=board.RX)

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

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

adapter = esp32.start_bluetooth()

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

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

Talk to the AirLift via the Bluefruit LE Connect App

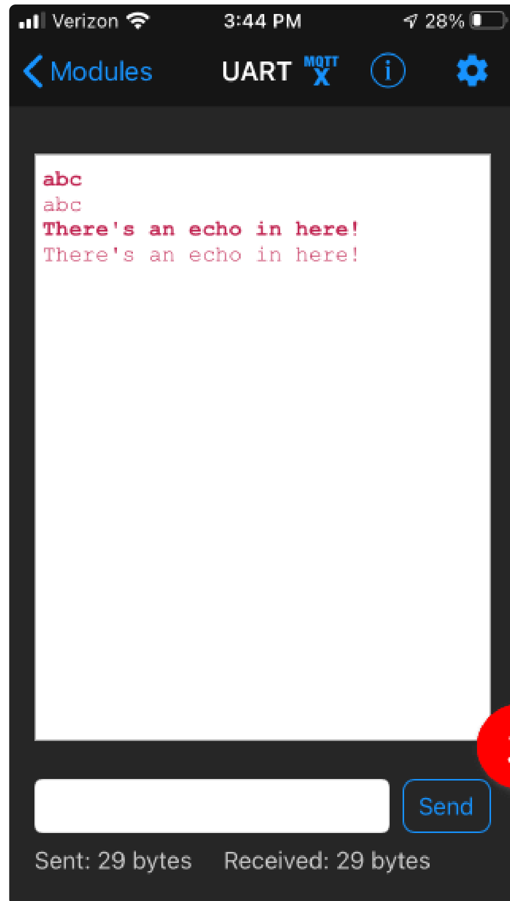
Start the Bluefruit LE Connect App on your phone or tablet. You should see a CIRCUITPY device available to connect to. Tap the Connect button (1):



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



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


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

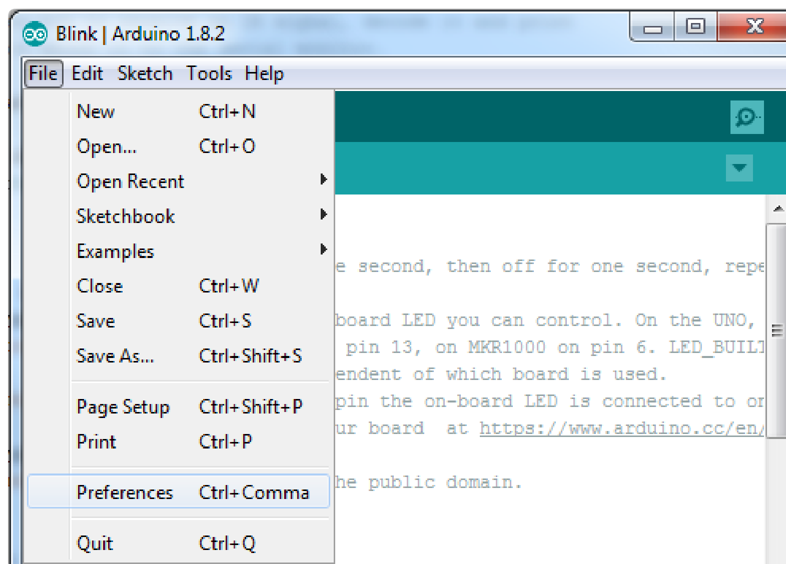
Arduino IDE Setup

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

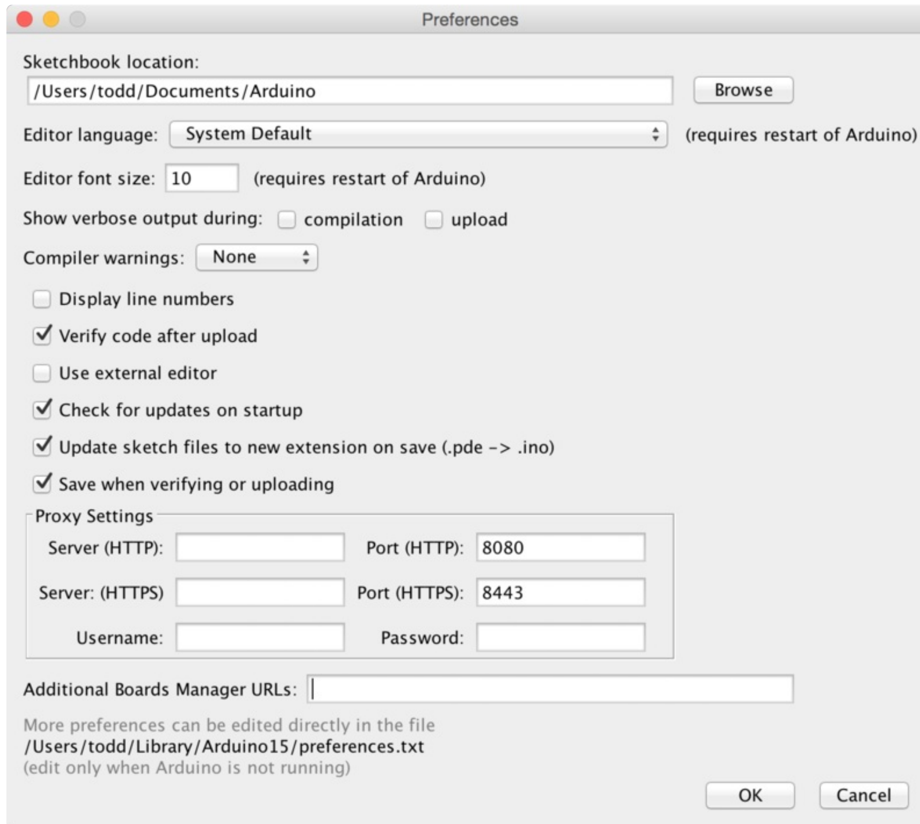
https://adafru.it/f1P

https://adafru.it/f1P

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



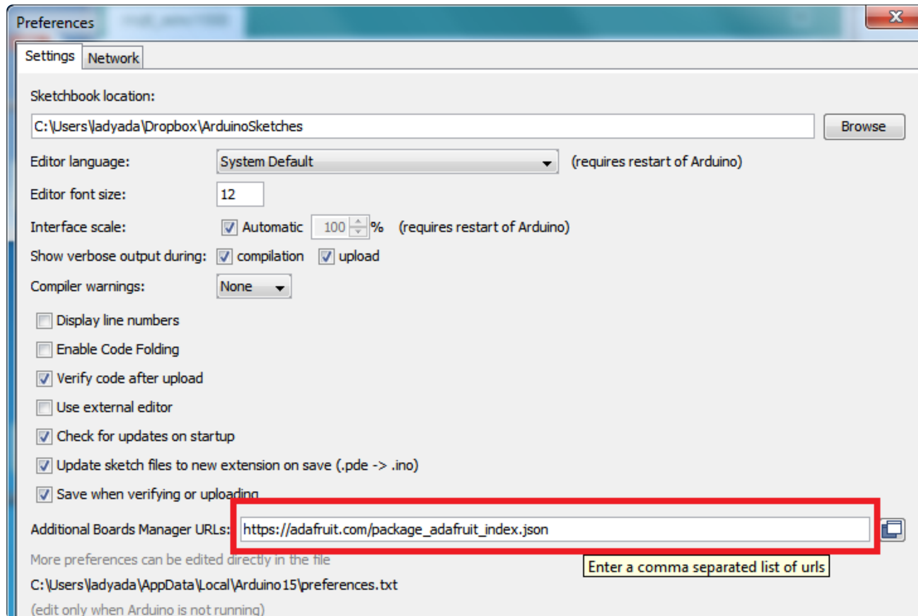
A dialog will pop up just like the one shown below.



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

To find the most up to date list of URLs you can add, you can visit the list of third party board URLs on the Arduino IDE wiki (https://adafru.it/f7U). We will only need to add one URL to the IDE in this example, but ***you can add multiple URLS by separating them with commas***. Copy and paste the link below into the **Additional Boards Manager URLs** option in the Arduino IDE preferences.

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



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

- **Adafruit AVR Boards** - Includes support for Flora, Gemma, Feather 32u4, ItsyBitsy 32u4, Trinket, & Trinket Pro.
- **Adafruit SAMD Boards** - Includes support for Feather M0 and M4, Metro M0 and M4, ItsyBitsy M0 and M4, Circuit Playground Express, Gemma M0 and Trinket M0
- **Arduino Leonardo & Micro MIDI-USB** - This adds MIDI over USB support for the Flora, Feather 32u4, Micro and Leonardo using the arcore project (https://adafru.it/eSI).

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

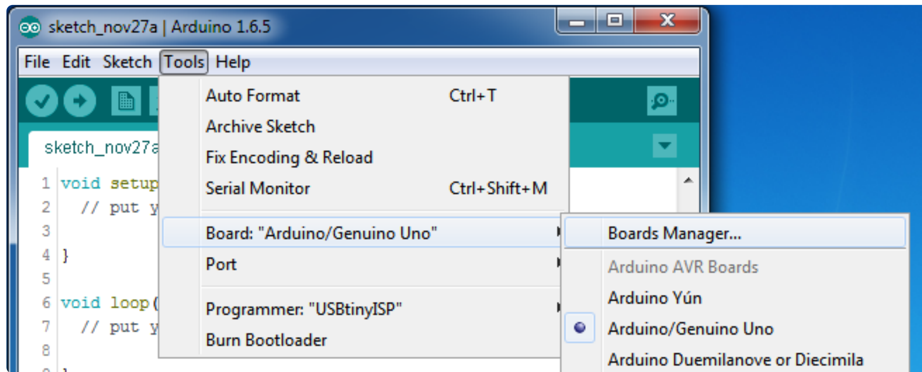
Once done click **OK** to save the new preference settings. Next we will look at installing boards with the Board Manager.

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

Using with Arduino IDE

The Feather/Metro/Gemma/QTPy/Trinket M0 and M4 use an ATSAMD21 or ATSAMD51 chip, and you can pretty easily get it working with the Arduino IDE. Most libraries (including the popular ones like NeoPixels and display) will work with the M0 and M4, especially devices & sensors that use I2C or SPI.

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



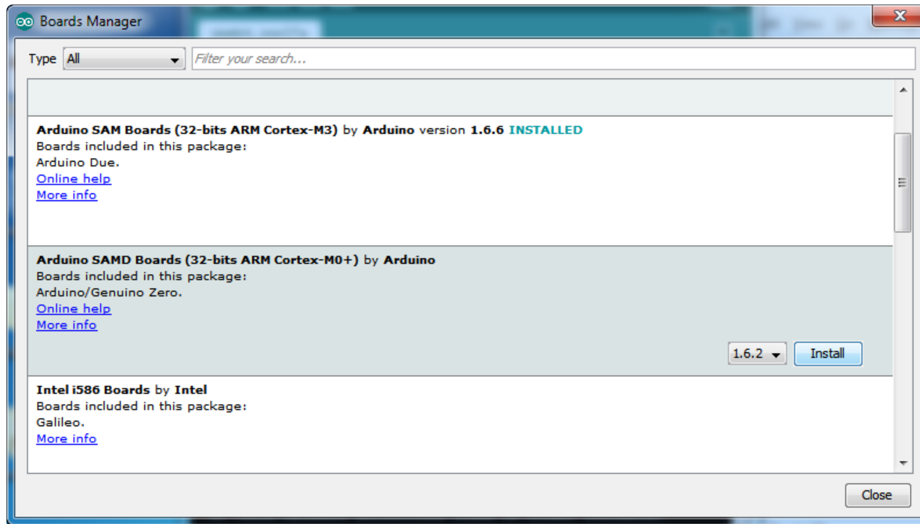
Once the Board Manager opens, click on the category drop down menu on the top left hand side of the window and select **All**. You will then be able to select and install the boards supplied by the URLs added to the preferences.

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

Install SAMD Support

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

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

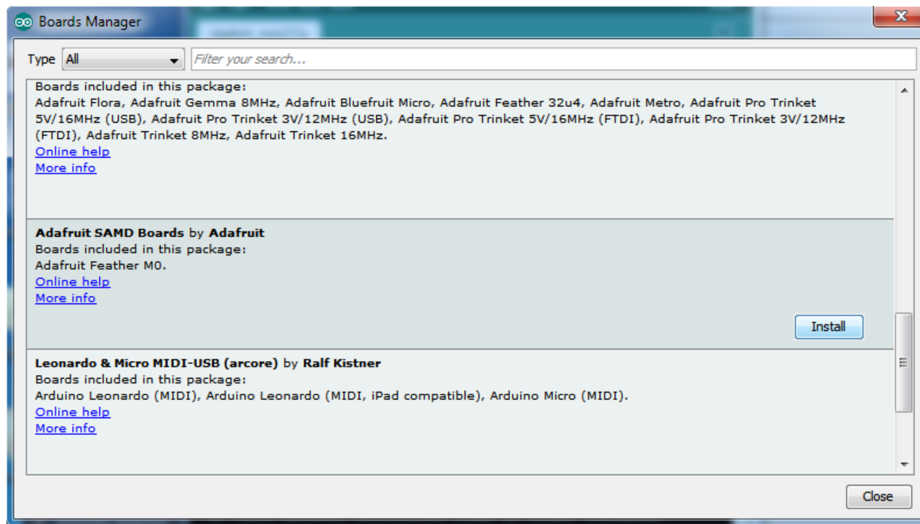


Install Adafruit SAMD

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

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

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



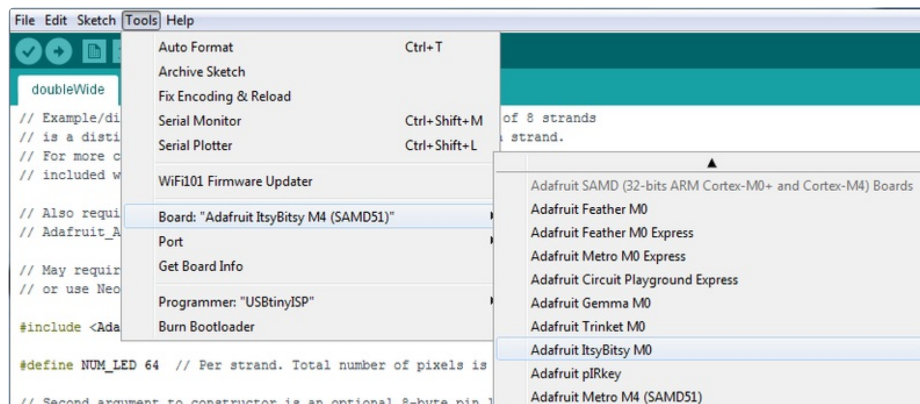
Even though in theory you don't need to - I recommend rebooting the IDE

Quit and reopen the Arduino IDE to ensure that all of the boards are properly installed. You should now be able to select and upload to the new boards listed in the **Tools->Board** menu.

Select the matching board, the current options are:

- **Feather M0** (for use with any Feather M0 other than the Express)

- **Feather M0 Express**
- **Metro M0 Express**
- **Circuit Playground Express**
- **Gemma M0**
- **Trinket M0**
- **QT Py M0**
- **ItsyBitsy M0**
- **Hallowing M0**
- **Crickit M0** (this is for direct programming of the Crickit, which is probably not what you want! For advanced hacking only)
- **Metro M4 Express**
- **Grand Central M4 Express**
- **ItsyBitsy M4 Express**
- **Feather M4 Express**
- **Trellis M4 Express**
- **PyPortal M4**
- **PyPortal M4 Titano**
- **PyBadge M4 Express**
- **Metro M4 Airlift Lite**
- **PyGamer M4 Express**
- **MONSTER M4SK**
- **Hallowing M4**
- **MatrixPortal M4**
- **BLM Badge**



Install Drivers (Windows 7 & 8 Only)

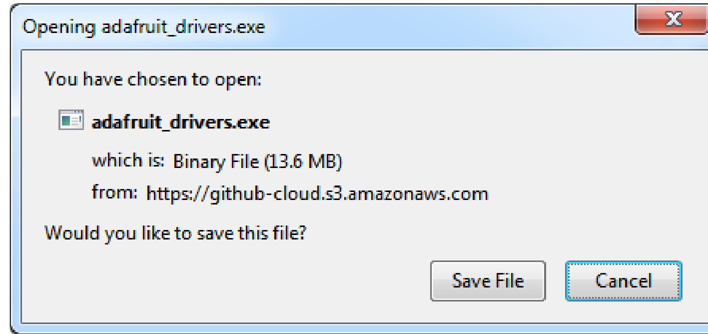
When you plug in the board, you'll need to possibly install a driver

Click below to download our Driver Installer

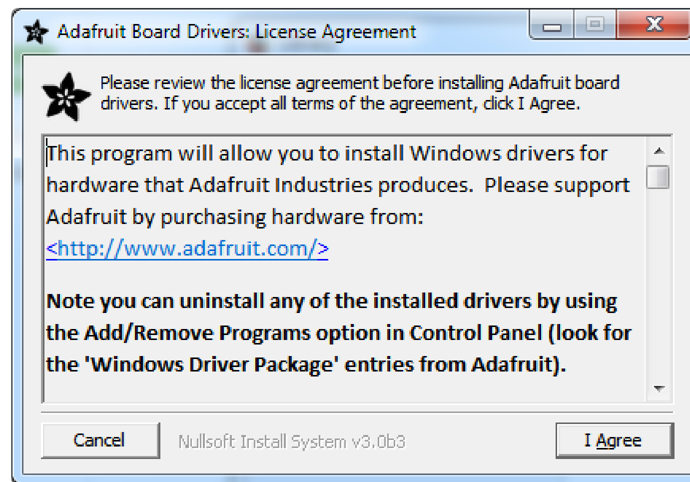
https://adafru.it/EC0

https://adafru.it/EC0

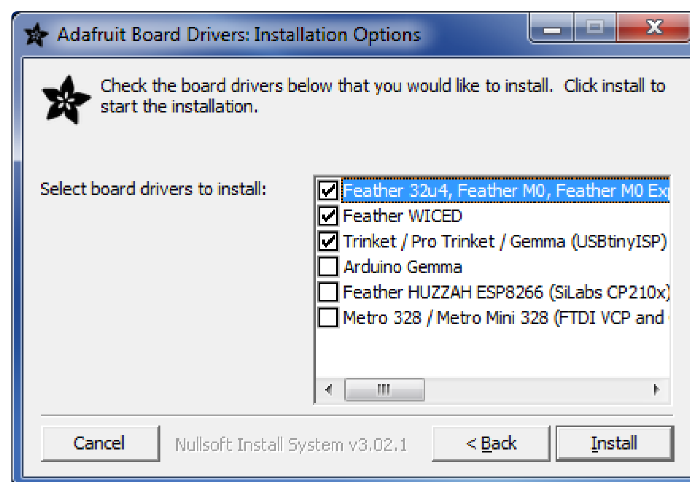
Download and run the installer



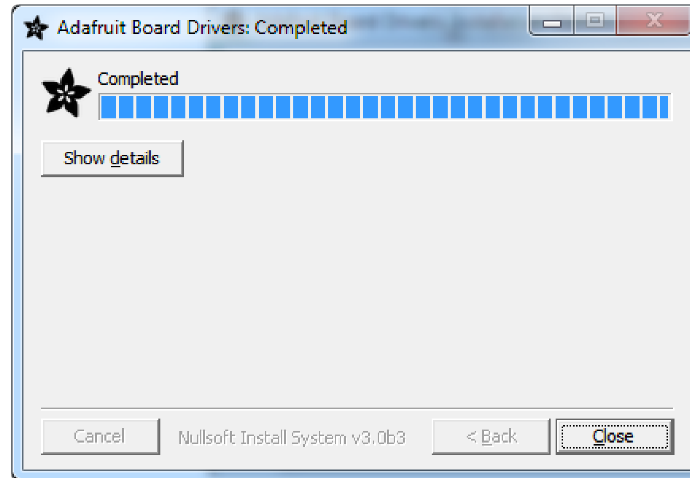
Run the installer! Since we bundle the SiLabs and FTDI drivers as well, you'll need to click through the license



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



Click **Install** to do the installin'

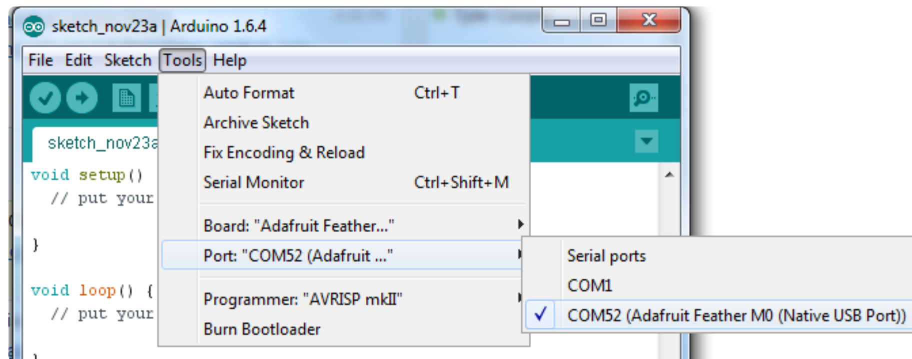


Blink

Now you can upload your first blink sketch!

Plug in the M0 or M4 board, and wait for it to be recognized by the OS (just takes a few seconds). It will create a serial/COM port, you can now select it from the drop-down, it'll even be 'indicated' as Trinket/Gemma/Metro/Feather/ItsyBitsy/Trellis!

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



Now load up the Blink example

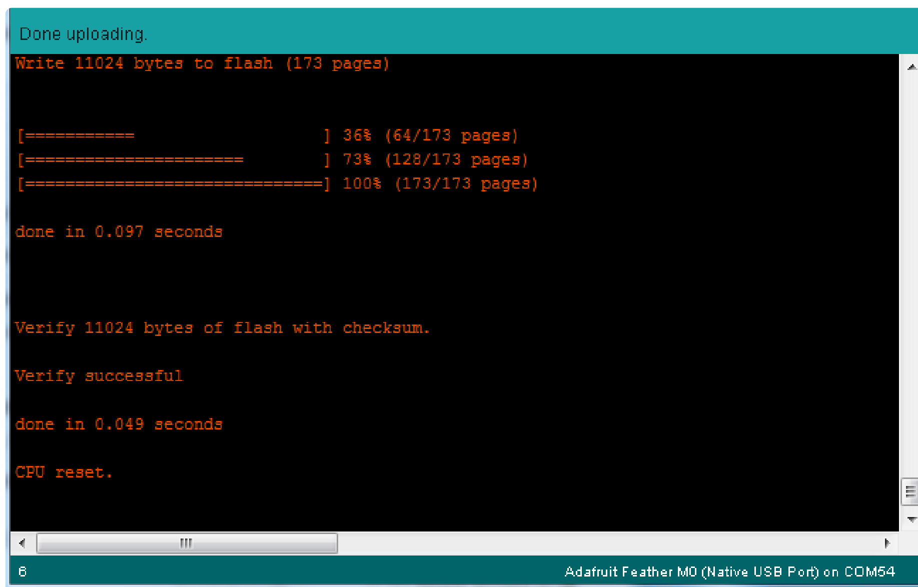
```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

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

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

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

Successful Upload

If you have a successful upload, you'll get a bunch of red text that tells you that the device was found and it was programmed, verified & reset



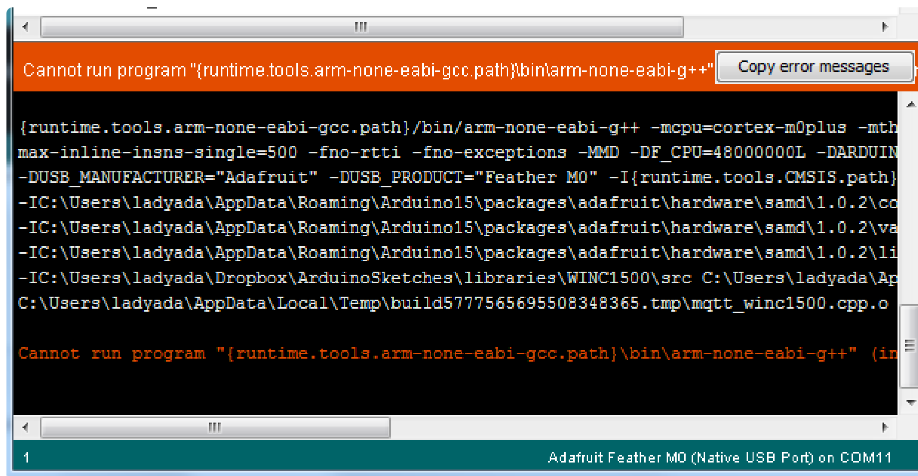
After uploading, you may see a message saying "Disk Not Ejected Properly" about the ...BOOT drive. You can ignore that message: it's an artifact of how the bootloader and uploading work.

Compilation Issues

If you get an alert that looks like

Cannot run program "{runtime.tools.arm-none-eabi-gcc.path}\bin\arm-non-eabi-g++"

Make sure you have installed the **Arduino SAMD** boards package, you need *both* Arduino & Adafruit SAMD board packages

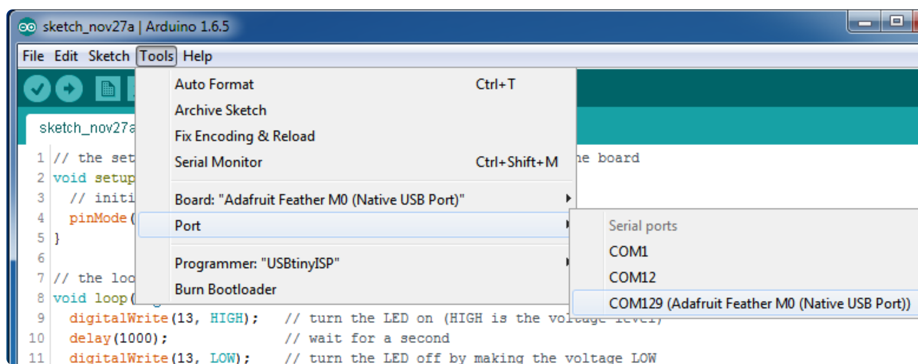


Manually bootloading

If you ever get in a 'weird' spot with the bootloader, or you have uploaded code that crashes and doesn't auto-reboot into the bootloader, click the **RST** button **twice** (like a double-click)to get back into the bootloader.

The red LED will pulse and/or RGB LED will be green, so you know that its in bootloader mode.

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



You may need to go back and reselect the 'normal' USB serial port next time you want to use the normal upload.

Ubuntu & Linux Issue Fix

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

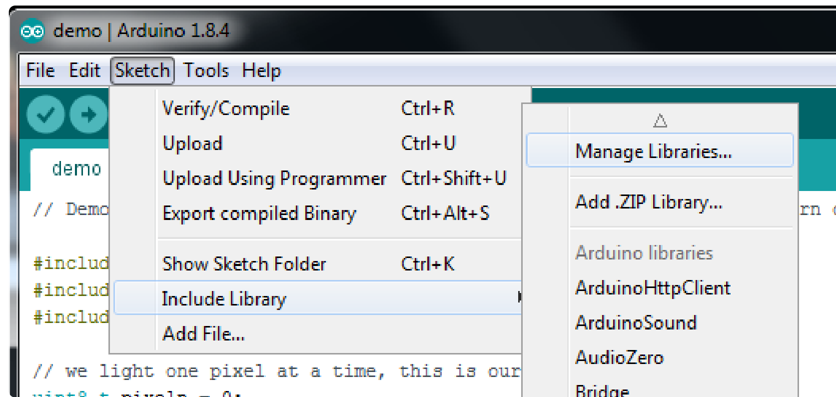
Arduino Libraries

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

There's a lot of libraries!

Install Libraries

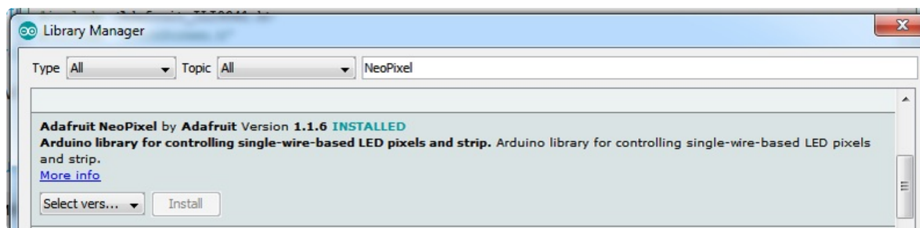
Open up the library manager...



And install the following libraries:

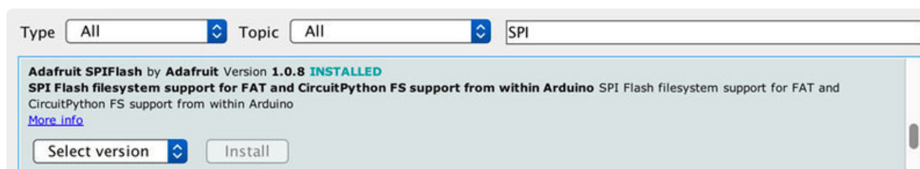
Adafruit NeoPixel

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



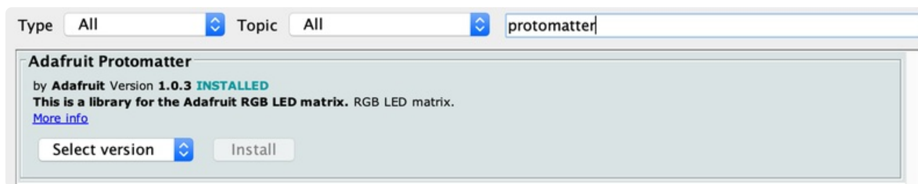
Adafruit SPIFlash

This is also needed to use the filesystem on QSPI



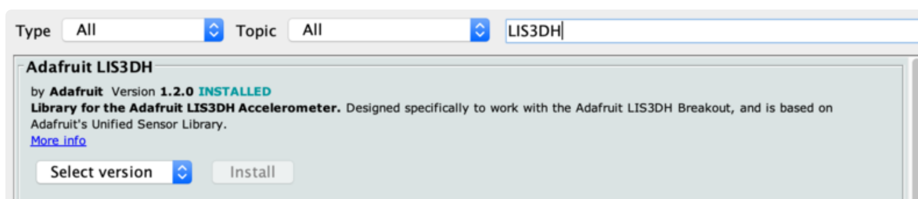
Adafruit Protomatter

This library is used for writing to the RGB Matrix.



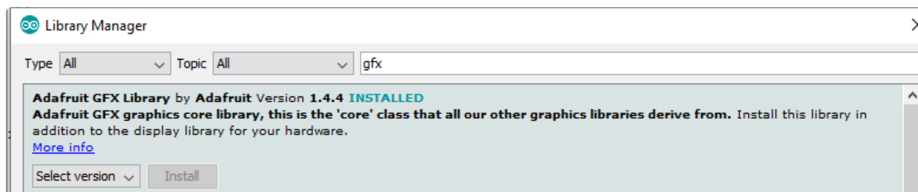
Adafruit LIS3DH

This will let you use the onboard accelerometer



Adafruit GFX

This is the graphics library used to draw to the screen



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

WiFiNINA

Will talk to the ESP32 WiFi co-processor to connect to the internet! We're using a variant of the Arduino WiFiNINA library, which is amazing and written by the Arduino team! **The official WiFi101 library won't work because it doesn't support the ability to change the pins**.

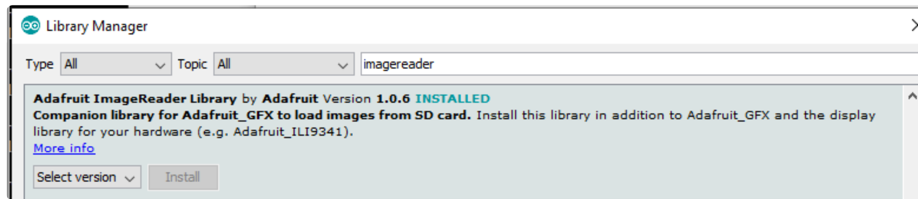
So! We made a fork that you can install. For more installation information see Arduino IO Library (https://adafru.it/OIC).

https://adafru.it/Evm

https://adafru.it/Evm

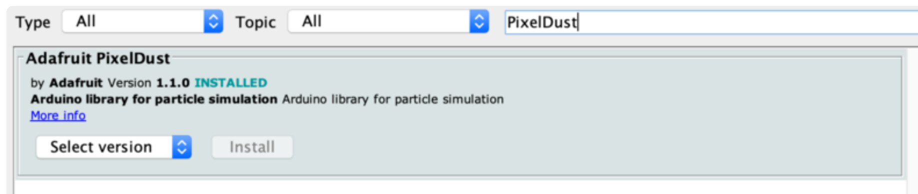
Adafruit ImageReader

For reading bitmaps from SD and displaying



Adafruit PixelDust

To compile and run the PixelDust demo, you will need this library. This library calculates where the particles should be.



Using the Protomatter Library

Let's look at a **minimal Arduino example** for the Adafruit_Protomatter library to illustrate how this works (this is pared down from the "**simple**" example sketch):

```
#include <Adafruit_Protomatter.h>

uint8_t rgbPins[]  = {7, 8, 9, 10, 11, 12};
uint8_t addrPins[] = {17, 18, 19, 20};
uint8_t clockPin   = 14;
uint8_t latchPin   = 15;
uint8_t oePin      = 16;

Adafruit_Protomatter matrix(
  64, 4, 1, rgbPins, 4, addrPins, clockPin, latchPin, oePin, false);

void setup(void) {
  Serial.begin(9600);

  // Initialize matrix...
  ProtomatterStatus status = matrix.begin();
  Serial.print("Protomatter begin() status: ");
  Serial.println((int)status);
  if(status != PROTOMATTER_OK) {
    for(;;);
  }

  // Make four color bars (red, green, blue, white) with brightness ramp:
  for(int x=0; x<matrix.width(); x++) {
    uint8_t level = x * 256 / matrix.width(); // 0-255 brightness
    matrix.drawPixel(x, matrix.height() - 4, matrix.color565(level, 0, 0));
    matrix.drawPixel(x, matrix.height() - 3, matrix.color565(0, level, 0));
    matrix.drawPixel(x, matrix.height() - 2, matrix.color565(0, 0, level));
    matrix.drawPixel(x, matrix.height() - 1, matrix.color565(level, level, level));
  }

  // Simple shapes and text, showing GFX library calls:
  matrix.drawCircle(12, 10, 9, matrix.color565(255, 0, 0));               // Red
  matrix.drawRect(14, 6, 17, 17, matrix.color565(0, 255, 0));             // Green
  matrix.drawTriangle(32, 9, 41, 27, 23, 27, matrix.color565(0, 0, 255)); // Blue
  matrix.println("ADAFRUIT"); // Default text color is white

  // AFTER DRAWING, A show() CALL IS REQUIRED TO UPDATE THE MATRIX!

  matrix.show(); // Copy data to matrix buffers
}

void loop(void) {
  Serial.print("Refresh FPS = ~");
  Serial.println(matrix.getFrameCount());
  delay(1000);
}
```

Breaking it down into steps…

Include Protomatter Library

First is to `#include` the library's header file. This in turn `#includes` **Adafruit_GFX.h**, so you don't have to.

```
#include <Adafruit_Protomatter.h>
```

Setting Up Matrix Pin Usage

The next few lines spell out the pin numbers being used. Using variables for this isn't entirely necessary… one *could* just pass the same numeric values directly to functions…but it makes the code a little more self-documenting (and easier to adapt the same sketch for multiple boards — the full example code has #ifdefs for each board with different pin assignments). These also could be `#defines` or `const` if one wants to be all Proper™ about it.

Technical stuff for developers, skip this if you just want to use the library:

> This is the one part of the Arduino code where **some knowledge of the underlying hardware is required**. `rgbPins[]` and `clockPin` **must all be on the same GPIO PORT peripheral** (e.g. all PORTA, all PORTB, etc.). The **other pins have no such restrictions**. Additionally, if the PORT has an atomic bit-toggle register, RAM requirements are minimized if `rgbPins[]` and `clockPin` are all **within the same byte** of that PORT*. They *do not need to be contiguous nor in any particular sequence* within that byte. If not within the same byte, next most efficient has them in the same upper or lower 16-bit word of the PORT. Scattered around a full 32-bit PORT still works but is the least RAM-efficient option.
>
> * For devices lacking an atomic bit-toggle register… `clockPin` *does not* need to be in the same byte, but still must be in the same PORT. Should still aim for `rgbPins[]` in a single byte or word though!

With those constraints in mind, here's what the code looks like for an Adafruit MatrixPortal M4 with a 64x32 pixel matrix:

```
uint8_t rgbPins[]  = {7, 8, 9, 10, 11, 12};
uint8_t addrPins[] = {17, 18, 19, 20};
uint8_t clockPin   = 14;
uint8_t latchPin   = 15;
uint8_t oePin      = 16;
```

The full "simple" example sketch has setups for a number of different boards and adapters.

Create the Protomatter Object

Next, still in the global area above `setup()`, we call the *constructor.* The Arduino library can only drive one matrix at a time (or one *chain* of matrices, where "out" from one is linked to "in" of the next), so we just have one instance of an `Adafruit_Protomatter` object here, which we'll call `matrix`:

```
Adafruit_Protomatter matrix(
  64, 4, 1, rgbPins, 4, addrPins, clockPin, latchPin, oePin, false);
```

The `Adafruit_Protomatter` constructor expects between 9 and 11 arguments depending on the situation. The vital ones here, in order, are:

- `64` — the total matrix chain width, in pixels. This will usually be `64` or `32`, the width of most common RGB LED matrices…but, if you have some other size or multiple matrices chained together, add up the total width here. For example, three chained 32-pixel-wide matrices would be `96`.
- `4` — the bit depth, in planes, from 1 to 6 (see below). More bitplanes provides greater color fidelity at the expense of more RAM. A value of 4 here (4 bits) provides 16 brightness levels each for red, green and blue — yielding 4,096 distinct colors possible.
- `1` — the number of matrix chains in parallel. This will almost always be 1, but the library could conceivably support up to 5, *if* the hardware driving it is set up *precisely just so.*
- `rgbPins` — a `uint8_t` array of pin numbers, which issue the red, green and blue data for the upper and lower half of the matrix (sometimes labeled R1, G1, B1, R2, G2, B2 on the matrix input).. The array should contain six times the prior argument…so, usually, six. If driving two chains in parallel, then 12 pin numbers and so forth. Obviously 12 pins won't fit in a single PORT byte, and you should aim for the upper or lower 16 bit word in that case, for best RAM utilization. Three or more chains, doesn't matter, but the pins all do still need to be in the same PORT.
- `4` — the number of row-select "address lines" used by the LED matrix (sometimes labeled A, B, C, etc. on the matrix input). 16-pixel-tall matrices will be three row-select lines, 32-pixel will have four, and 64-pixel will have five. Matrix height is always *inferred* from this value, not passed explicitly like width.
- `addrPins` — a `uint8_t` array of pin numbers, one for each row-select address line, starting from least-significant bit. These *do not* need to be on the same PORT as `rgbPins` or each other…they can be mixed about anywhere.
- `clockPin` — pin number which drives the RGB clock (CLK on matrix input). This *must* be on the same PORT register as `rgbPins`, and in most cases should also try to be in the same *byte.*
- `latchPin` — pin number for "latch" signal (LAT on matrix input), indicating end-of-data. Can be any output-capable pin, no special constraints.
- `oePin` — pin number for "!OE" signal (output-enable low, OE on matrix input). Can be any output-capable pin, no special constraints.
- `false` — this flag indicates if the display should be *double-buffered,* better for animation at the expense of double the RAM usage. Since the protomatter example isn't using animation, it passes `false` here…but if you look at the **doublebuffer_scrolltext** example, it uses `true`. A double-buffered display only modifies the matrix between refreshes, avoiding "tearing" artifacts. Optional. Default, if left unspecified, is `false`.
- Not used here, an optional 11th argument supports "tiling" of matrices vertically. Horizontal tiling is

already implicit in the first argument — if you had two 64x32 matrices side-by-side, you'd pass 128 there. But if you had *four* such matrices arranged 2x2, you'd still pass 128 for the first argument, but then add either 2 here (if cabling is in a "progressive" order) or -2 (if a "serpentine" order, where the second row of panels is rotated 180° relative to the first…the cabling is a little easier). The "tiled.ino" example demonstrates this. The concept is explained further in the [CircuitPython LED Matrix guide](https://adafru.it/Qey) (https://adafru.it/Qey)…the same principles apply to the Arduino library, the arguments are just a little different here. Default if unspecified is 1 (no vertical tiling).
- Also not used here, an optional 12th argument is a pointer to a hardware-specific timer structure…this is *super exceedingly esoteric* and not really used for now, but in principle would allow the library to work with other timer peripherals than the default.

Begin Protomatter Driver

Now, with the matrix object created, inside `setup()` we call its `begin()` function. It's pretty important to look at the value returned, which is a `ProtomatterStatus` type:

```
ProtomatterStatus status = matrix.begin();
```

Possible return status values include:

- `PROTOMATTER_OK` — everything is good and the program can proceed (otherwise it should *stop*…the example code is not a good neighbor in this regard).
- `PROTOMATTER_ERR_PINS` — the RGB data and clock pins are not all on the same PORT. Can't continue, the library *requires* these pins in this layout.
- `PROTOMATTER_ERR_MALLOC` — couldn't allocate enough memory for display. Can't continue. This is usually an error that happens in the `begin()` function, but in extreme cases even the constructor could hit an allocation problem, but you won't get this response until calling `begin()`.
- `PROTOMATTER_ERR_ARG` — some other bad input to function, distinct from PROTOMATTER_ERR_PINS. *Exceedingly* rare, might only happen if constructor failed.

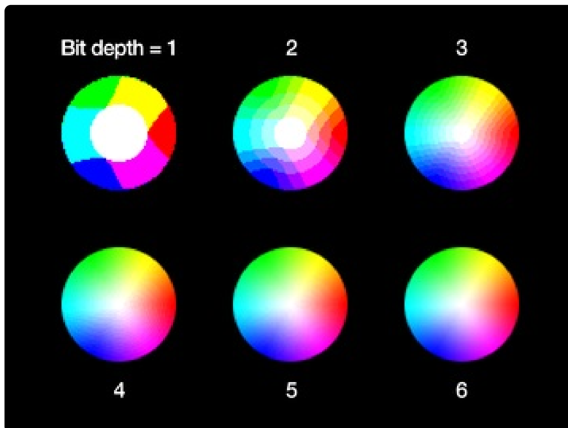
Draw Shapes & Text Using Adafruit GFX

Then we draw some stuff on the display. **Any graphics primitive supported by the [Adafruit_GFX library](https://adafru.it/doL) (https://adafru.it/doL) is available here.**

Adafruit_GFX is the same library that drives many of our LCD and OLED displays…if you've done other graphics projects, you might already be familiar! And if not, [we have a separate guide explaining all of the available drawing functions](https://adafru.it/DtY) (https://adafru.it/DtY). Most folks can get a quick start by looking at the "simple" and

“doublebuffer_scrolltext” examples and tweaking these for their needs.

Any color argument passed to a drawing function here is a 16-bit value, with the highest 5 bits representing red brightness (0 to 31), middle 6 bits for green (0 to 63), and least 5 bits for blue (0 to 31). It's just how Adafruit_GFX works and is a carryover from early PC graphics and most small LCD/OLED displays.



The effect of bit depth on image quality. Color values are always specified as full 16-bit “565” values, but will quantize to coarser representations at lower bit depths.

Sometimes you might want to avoid 6-bit depth even if RAM permits it. Only green handles the full 6 bits, while red and blue are quantized to 5 bits. This can result in some colors or gradients having slight green or magenta tints to them. 5-bit depth is slightly blockier but colors are more predictable.

```
matrix.drawCircle(12, 10, 9, matrix.color565(255, 0, 0));              // Red
matrix.drawRect(14, 6, 17, 17, matrix.color565(0, 255, 0));            // Green
...etc...
matrix.show(); // Copy data to matrix buffers
```

Notice though the call to `matrix.show()` at the end. Drawing operations have no immediate effect on the LED matrix, and instead are working on a buffer in RAM behind the scenes. Calling `show()` is **required** — it “pushes” the display data from that buffer to the matrix. You can call it after each drawing function, or group up a bunch of drawing commands with a single `show()` afterward to all appear at once. If you've worked with NeoPixel programming, it's a similar phenomenon.

Since this program isn't animating anything, it's finished at that point and `loop()` *could* be empty.

Check Refresh Rate

For the sake of curious information though, the example shows the matrix refresh rate using `getFrameCount()`. This returns the *number of frames* since the *last call to the same function,* not the refresh rate...but if spaced about one second apart (`delay(1000)`), you get a fair approximation of refresh rate:

```
Serial.println(matrix.getFrameCount());
delay(1000);
```

The matrix refresh rate is influenced by so many factors…processor speed, matrix chain length, bit depth… that it's difficult to accurately predict ahead of time, so this is a way to see what you get when changing different values in the constructor.

This is a subjective thing, but in broad terms 200 Hz or better should provide a solid image…any less and it starts to become flickery, so you might want a lower bit depth in that case. Conversely, refreshing too fast would waste CPU cycles that you probably want for other tasks like animation. The library does its best to throttle back and not refresh faster than practically needed.

Arduino Sand Demo

We have a Sand/Pixel Dust demo available for the MatrixPortal that runs in Arduino. Below is the code to run it. The demo is also available as an example in the Arduino Protomatter library.

```
/* ----------------------------------------------------------------------
"Pixel dust" Protomatter library example. As written, this is
SPECIFICALLY FOR THE ADAFRUIT MATRIXPORTAL M4 with 64x32 pixel matrix.
Change "HEIGHT" below for 64x64 matrix. Could also be adapted to other
Protomatter-capable boards with an attached LIS3DH accelerometer.

PLEASE SEE THE "simple" EXAMPLE FOR AN INTRODUCTORY SKETCH,
or "doublebuffer" for animation basics.
------------------------------------------------------------------------- */

#include <Wire.h>                 // For I2C communication
#include <Adafruit_LIS3DH.h>      // For accelerometer
#include <Adafruit_PixelDust.h>   // For sand simulation
#include <Adafruit_Protomatter.h> // For RGB matrix

#define HEIGHT  32 // Matrix height (pixels) - SET TO 64 FOR 64x64 MATRIX!
#define WIDTH   64 // Matrix width (pixels)
#define MAX_FPS 45 // Maximum redraw rate, frames/second

#if HEIGHT == 64 // 64-pixel tall matrices have 5 address lines:
uint8_t addrPins[] = {17, 18, 19, 20, 21};
#else            // 32-pixel tall matrices have 4 address lines:
uint8_t addrPins[] = {17, 18, 19, 20};
#endif

// Remaining pins are the same for all matrix sizes. These values
// are for MatrixPortal M4. See "simple" example for other boards.
uint8_t rgbPins[]  = {7, 8, 9, 10, 11, 12};
uint8_t clockPin   = 14;
uint8_t latchPin   = 15;
uint8_t oePin      = 16;

Adafruit_Protomatter matrix(
  WIDTH, 4, 1, rgbPins, sizeof(addrPins), addrPins,
  clockPin, latchPin, oePin, true);

Adafruit_LIS3DH accel = Adafruit_LIS3DH();

#define N_COLORS   8
#define BOX_HEIGHT 8
#define N_GRAINS (BOX_HEIGHT*N_COLORS*8)
uint16_t colors[N_COLORS];

Adafruit_PixelDust sand(WIDTH, HEIGHT, N_GRAINS, 1, 128, false);

uint32_t prevTime = 0; // Used for frames-per-second throttle

// SETUP - RUNS ONCE AT PROGRAM START --------------------------------------
```

```
void err(int x) {
  uint8_t i;
  pinMode(LED_BUILTIN, OUTPUT);       // Using onboard LED
  for(i=1;;i++) {                     // Loop forever...
    digitalWrite(LED_BUILTIN, i & 1); // LED on/off blink to alert user
    delay(x);
  }
}

void setup(void) {
  Serial.begin(115200);
  //while (!Serial) delay(10);

  ProtomatterStatus status = matrix.begin();
  Serial.printf("Protomatter begin() status: %d\n", status);

  if (!sand.begin()) {
    Serial.println("Couldn't start sand");
    err(1000); // Slow blink = malloc error
  }

  if (!accel.begin(0x19)) {
    Serial.println("Couldn't find accelerometer");
    err(250);  // Fast bink = I2C error
  }
  accel.setRange(LIS3DH_RANGE_4_G);   // 2, 4, 8 or 16 G!

  //sand.randomize(); // Initialize random sand positions

  // Set up initial sand coordinates, in 8x8 blocks
  int n = 0;
  for(int i=0; i<N_COLORS; i++) {
    int xx = i * WIDTH / N_COLORS;
    int yy =  HEIGHT - BOX_HEIGHT;
    for(int y=0; y<BOX_HEIGHT; y++) {
      for(int x=0; x < WIDTH / N_COLORS; x++) {
        //Serial.printf("#%d -> (%d, %d)\n", n,  xx + x, yy + y);
        sand.setPosition(n++, xx + x, yy + y);
      }
    }
  }
  Serial.printf("%d total pixels\n", n);

  colors[0] = matrix.color565(64, 64, 64);  // Dark Gray
  colors[1] = matrix.color565(120, 79, 23); // Brown
  colors[2] = matrix.color565(228,  3,  3); // Red
  colors[3] = matrix.color565(255,140,  0); // Orange
  colors[4] = matrix.color565(255,237,  0); // Yellow
  colors[5] = matrix.color565(  0,128, 38); // Green
  colors[6] = matrix.color565(  0, 77,255); // Blue
  colors[7] = matrix.color565(117,  7,135); // Purple
}

// MAIN LOOP - RUNS ONCE PER FRAME OF ANIMATION ----------------------------

void loop() {
  // Limit the animation frame rate to MAX_FPS.  Because the subsequent sand
```

```
  // Limit the animation frame rate to MAX_FPS.  Because the subsequent sand
  // calculations are non-deterministic (don't always take the same amount
  // of time, depending on their current states), this helps ensure that
  // things like gravity appear constant in the simulation.
  uint32_t t;
  while(((t = micros()) - prevTime) < (1000000L / MAX_FPS));
  prevTime = t;

  // Read accelerometer...
  sensors_event_t event;
  accel.getEvent(&event);
  //Serial.printf("(%0.1f, %0.1f, %0.1f)\n", event.acceleration.x, event.acceleration.y, 
event.acceleration.z);

  double xx, yy, zz;
  xx = event.acceleration.x * 1000;
  yy = event.acceleration.y * 1000;
  zz = event.acceleration.z * 1000;

  // Run one frame of the simulation
  sand.iterate(xx, yy, zz);

  //sand.iterate(-accel.y, accel.x, accel.z);

  // Update pixel data in LED driver
  dimension_t x, y;
  matrix.fillScreen(0x0);
  for(int i=0; i<N_GRAINS ; i++) {
    sand.getPosition(i, &x, &y);
    int n = i / ((WIDTH / N_COLORS) * BOX_HEIGHT); // Color index
    uint16_t flakeColor = colors[n];
    matrix.drawPixel(x, y, flakeColor);
    //Serial.printf("(%d, %d)\n", x, y);
  }
  matrix.show(); // Copy data to matrix buffers
}
```

This sketch was written for a **64x32** pixel matrix but is **easily modified** for a **64x64** matrix!

Look for this line in the code:

```
#define HEIGHT  32 // Matrix height (pixels) - SET TO 64 FOR 64x64 MATRIX!
```

and change it to:

```
#define HEIGHT  64 // Matrix height (pixels) - SET TO 64 FOR 64x64 MATRIX!
```

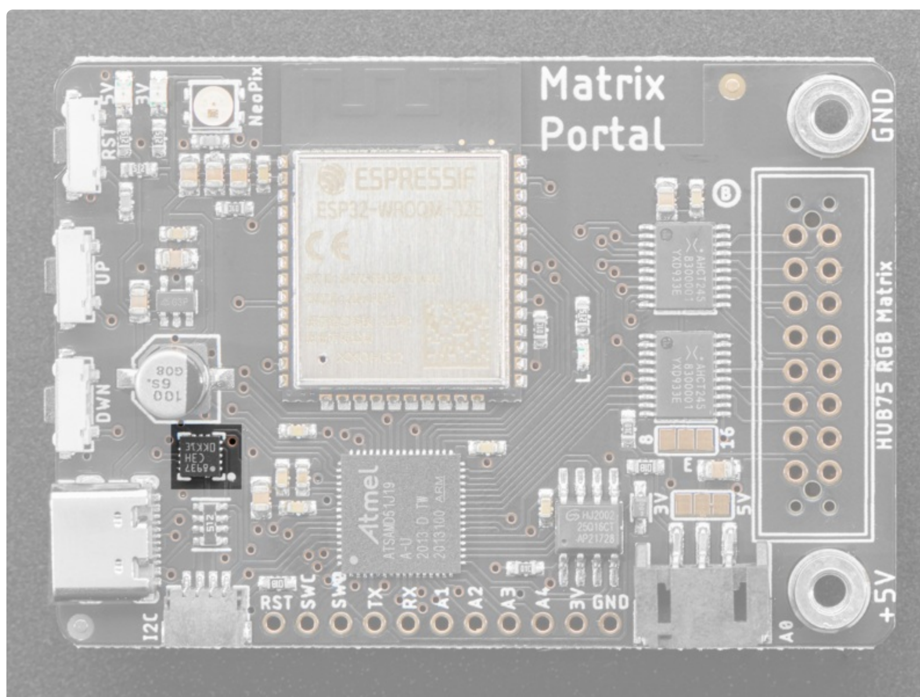
Now upload the sketch to your MatrixPortal M4. You may need to press the Reset button to reset the MatrixPortal. You should see a a series of colored rectangles along the bottom. Go ahead and start moving the matrix around!

If you have a 3D Printer, be sure to check out the Matrix Portal Sand Handles (https://adafru.it/NDg) guide.

Protomatter Library

Protomatter Library (https://adafru.it/MNa)

Using the Accelerometer



On the back of the MatrixPortal is a triple-axis accelerometer that you can use to detect motion. (That's how our cool digital sand demo works!)

You can use this sensor in both CircuitPython and Arduino. After the library support is installed, you can then use the example code provided to get X, Y and Z acceleration values.

For more details on the LIS3DH sensor, we have a full guide you should read once you get the basics working below (https://adafru.it/uBr)

Arduino Usage

In Arduino, make sure to install our LIS3DH library, our guide for the individual sensor covers all that here (https://adafru.it/OkE)

You can then load up this example

However, before you upload it - change this line:

`if (! lis.begin(0x18)) { // change this to 0x19 for alternative i2c address`

to:

`if (! lis.begin(0x19)) { // change this to 0x19 for alternative i2c address`

This will change the library to use the alternate address 0x19 of the accelerometer instead of the default 0x18!

```
// Basic demo for accelerometer readings from Adafruit LIS3DH

#include <Wire.h>
#include <SPI.h>
#include <Adafruit_LIS3DH.h>
#include <Adafruit_Sensor.h>

// Used for software SPI
#define LIS3DH_CLK 13
#define LIS3DH_MISO 12
#define LIS3DH_MOSI 11
// Used for hardware & software SPI
#define LIS3DH_CS 10

// software SPI
//Adafruit_LIS3DH lis = Adafruit_LIS3DH(LIS3DH_CS, LIS3DH_MOSI, LIS3DH_MISO, LIS3DH_CLK);
// hardware SPI
//Adafruit_LIS3DH lis = Adafruit_LIS3DH(LIS3DH_CS);
// I2C
Adafruit_LIS3DH lis = Adafruit_LIS3DH();

void setup(void) {
  Serial.begin(115200);
  while (!Serial) delay(10);     // will pause Zero, Leonardo, etc until serial console opens

  Serial.println("LIS3DH test!");

  if (! lis.begin(0x18)) {   // change this to 0x19 for alternative i2c address
    Serial.println("Couldnt start");
    while (1) yield();
  }
  Serial.println("LIS3DH found!");

  // lis.setRange(LIS3DH_RANGE_4_G);   // 2, 4, 8 or 16 G!

  Serial.print("Range = "); Serial.print(2 << lis.getRange());
  Serial.println("G");

  // lis.setDataRate(LIS3DH_DATARATE_50_HZ);
  Serial.print("Data rate set to: ");
  switch (lis.getDataRate()) {
    case LIS3DH_DATARATE_1_HZ: Serial.println("1 Hz"); break;
    case LIS3DH_DATARATE_10_HZ: Serial.println("10 Hz"); break;
    case LIS3DH_DATARATE_25_HZ: Serial.println("25 Hz"); break;
    case LIS3DH_DATARATE_50_HZ: Serial.println("50 Hz"); break;
    case LIS3DH_DATARATE_100_HZ: Serial.println("100 Hz"); break;
    case LIS3DH_DATARATE_200_HZ: Serial.println("200 Hz"); break;
    case LIS3DH_DATARATE_400_HZ: Serial.println("400 Hz"); break;

    case LIS3DH_DATARATE_POWERDOWN: Serial.println("Powered Down"); break;
    case LIS3DH_DATARATE_LOWPOWER_5KHZ: Serial.println("5 Khz Low Power"); break;
    case LIS3DH_DATARATE_LOWPOWER_1K6HZ: Serial.println("16 Khz Low Power"); break;
  }
```

```
  }
}

void loop() {
  lis.read();      // get X Y and Z data at once
  // Then print out the raw data
  Serial.print("X:  "); Serial.print(lis.x);
  Serial.print("  \tY:  "); Serial.print(lis.y);
  Serial.print("  \tZ:  "); Serial.print(lis.z);

  /* Or....get a new sensor event, normalized */
  sensors_event_t event;
  lis.getEvent(&event);

  /* Display the results (acceleration is measured in m/s^2) */
  Serial.print("\t\tX: "); Serial.print(event.acceleration.x);
  Serial.print(" \tY: "); Serial.print(event.acceleration.y);
  Serial.print(" \tZ: "); Serial.print(event.acceleration.z);
  Serial.println(" m/s^2 ");

  Serial.println();

  delay(200);
}
```

CircuitPython Usage

In CircuitPython, ditto - check the guide for how to install the library (https://adafru.it/OkF). Once you've dragged the files over, you can use this example code and paste it into your code.py file

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

import time
import board
import busio
import adafruit_lis3dh

# Hardware I2C setup. Use the CircuitPlayground built-in accelerometer if available;
# otherwise check I2C pins.
if hasattr(board, "ACCELEROMETER_SCL"):
    i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
    lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, address=0x19)
else:
    i2c = board.I2C()  # uses board.SCL and board.SDA
    lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c)

# Hardware SPI setup:
# spi = board.SPI()
# cs = digitalio.DigitalInOut(board.D5)  # Set to correct CS pin!
# lis3dh = adafruit_lis3dh.LIS3DH_SPI(spi, cs)

# PyGamer or MatrixPortal I2C Setup:
# i2c = board.I2C()  # uses board.SCL and board.SDA
# lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, address=0x19)


# Set range of accelerometer (can be RANGE_2_G, RANGE_4_G, RANGE_8_G or RANGE_16_G).
lis3dh.range = adafruit_lis3dh.RANGE_2_G

# Loop forever printing accelerometer values
while True:
    # Read accelerometer values (in m / s ^ 2).  Returns a 3-tuple of x, y,
    # z axis values.  Divide them by 9.806 to convert to Gs.
    x, y, z = [
        value / adafruit_lis3dh.STANDARD_GRAVITY for value in lis3dh.acceleration
    ]
    print("x = %0.3f G, y = %0.3f G, z = %0.3f G" % (x, y, z))
    # Small delay to keep things responsive but give time for interrupt processing.
    time.sleep(0.1)
```

Before you save, however, you must tell the example where to find the sensor!

Remove these lines:

```
# Hardware I2C setup. Use the CircuitPlayground built-in accelerometer if available;
# otherwise check I2C pins.
if hasattr(board, "ACCELEROMETER_SCL"):
    i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
    int1 = digitalio.DigitalInOut(board.ACCELEROMETER_INTERRUPT)
    lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, address=0x19, int1=int1)
else:
    i2c = busio.I2C(board.SCL, board.SDA)
    int1 = digitalio.DigitalInOut(board.D6)  # Set to correct pin for interrupt!
    lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, int1=int1)
```

And 'uncomment' these lines:

```
# PyGamer OR MatrixPortal I2C Setup:
# i2c = busio.I2C(board.SCL, board.SDA)
# int1 = digitalio.DigitalInOut(board.ACCELEROMETER_INTERRUPT)
# lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, address=0x19, int1=int1)
```

So they look like this:

```
# PyGamer OR MatrixPortal I2C Setup:
i2c = busio.I2C(board.SCL, board.SDA)
int1 = digitalio.DigitalInOut(board.ACCELEROMETER_INTERRUPT)
lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, address=0x19, int1=int1)
```

Now you can save, and check the REPL for acceleration data!

Updating ESP32 Firmware

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

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

Downloads

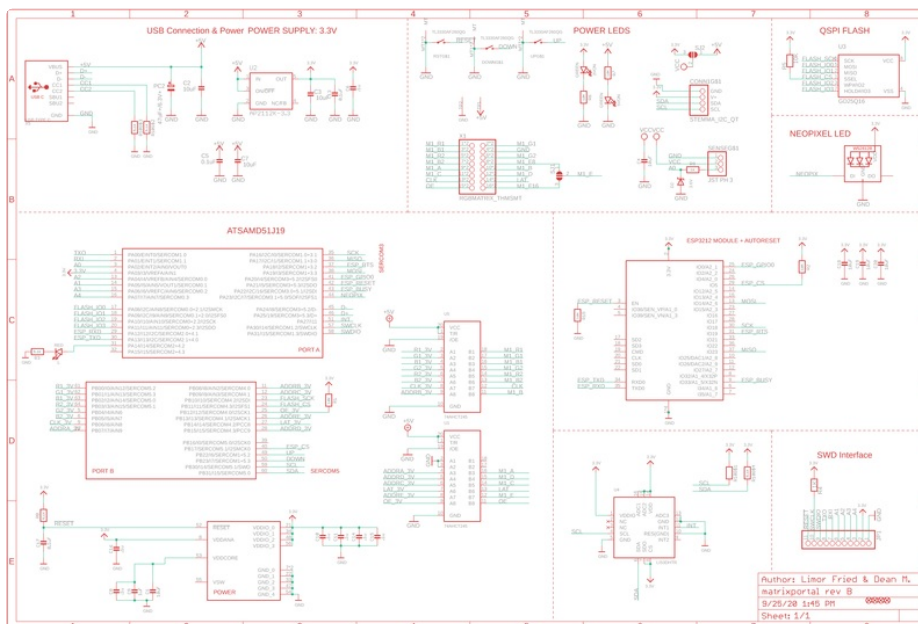
Files

- ATSAMD51J19 datasheet (https://adafru.it/NBJ)
- EagleCAD PCB files on GitHub (https://adafru.it/NBK)
- Fritzing object in Adafruit Fritzing Library (https://adafru.it/E9e)

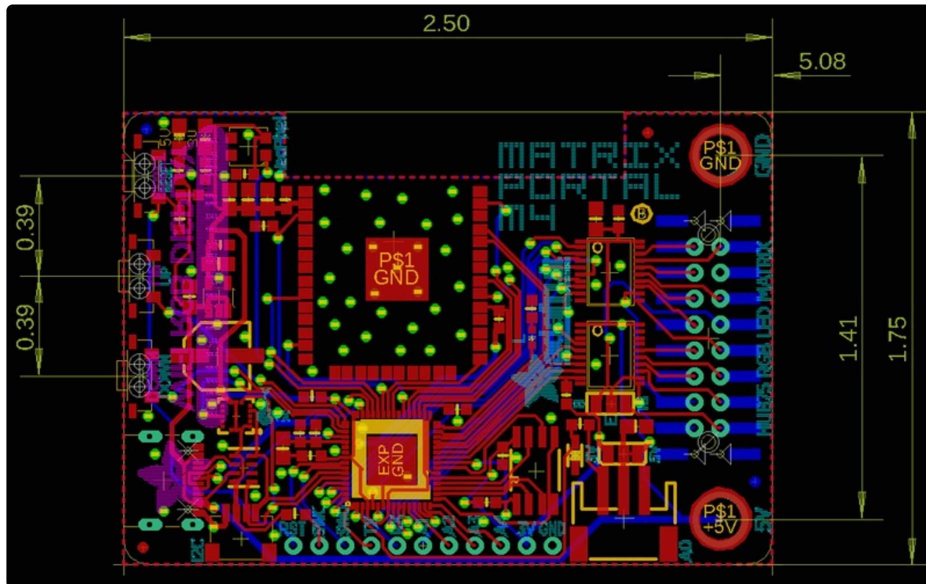
https://adafru.it/ScN

https://adafru.it/ScN

Schematic



Fab Print


2.50
5.08
0.39
0.39
1.41
1.75
MATRIX
PORTAL
M4
P$1
GND
EXP
GND
+5V
HUB75 RGB LED MATRIX