Creating MatrixPortal Projects with CircuitPython

Created by Melissa LeBlanc-Williams

https://learn.adafruit.com/creating-projects-with-the-circuitpython-matrixportal-library

Last updated on 2023-10-11 06:22:45 PM EDT

Table of Contents

Overview

So you may have heard of the Adafruit MatrixPortal M4 () and you want to get started with building a project using CircuitPython. The MatrixPortal library makes it really easy to get started with creating a new project, using this board, but it also supports a variety of other hardware to make creating project for an RGB Matrix very easy.

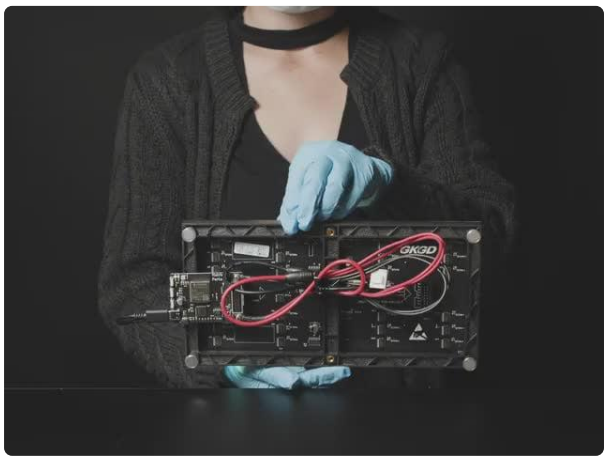
This library is built on top of the rgbmatrix and framebufferio modules which are included as part of CircuitPython on many of our boards. It also makes use of the ESP32SPI library, along with some lower-level dependencies, to communicate with server over the internet.

One of the big benefits of using this library is that it handles a lot of detection and initialization of RGB Matrices and drivers. For instance, if you had a Metro M4 Airlift Lite () and an RGB Matrix Shield (), you could still get the full benefit of the library. If you had a Feather M4 Express () with an RGB Matrix FeatherWing for M0/M4 feathers ( ), you could still get the benefit of RGB Matrix automatically initializing. It also supports the RGB Matrix FeatherWing for nRF52840 feathers () when used in conjunction with an nRF52840 feather ().

This library also supports a variety of different RGB Matrix Panel sizes. The number of address line pins used are based on the height of the display. It was originally written for the 64x32 matrix, but has also been tested on the 64x64 matrix (), 16x32 matrix () and 32x32 matrix () as well.

For this guide we're going to cover using a MatrixPortal M4 along with a 64x32 matrix as those are both very popular options and are feature complete. If your hardware setup varies, you may need to make some adjustments.

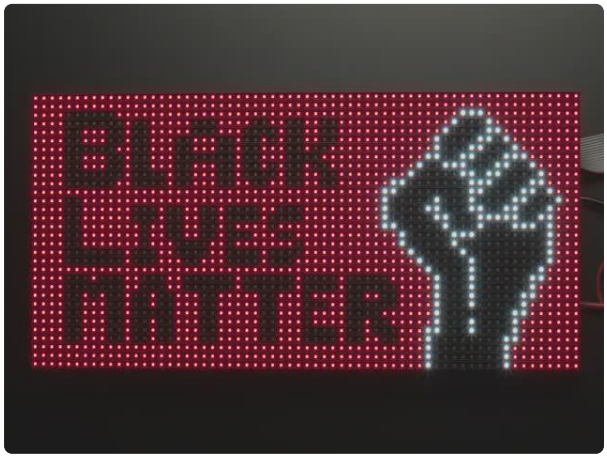
Parts



Adafruit Matrix Portal - CircuitPython Powered Internet Display

Folks love our wide selection of RGB matrices and accessories, for making custom colorful LED displays... and our RGB Matrix Shields...

https://www.adafruit.com/product/4745



64x32 RGB LED Matrix - 4mm pitch

Bring a little bit of Times Square into your home with this sweet 64 x 32 square RGB LED matrix panel. These panels are normally used to make video walls, here in New York we see them...

https://www.adafruit.com/product/2278

64x32 RGB LED Matrix - 3mm pitch

Bring a little bit of Times Square into your home with this sweet 64 x 32 square RGB LED matrix panel. These panels are normally used to make video walls, here in New York we see them...

https://www.adafruit.com/product/2279

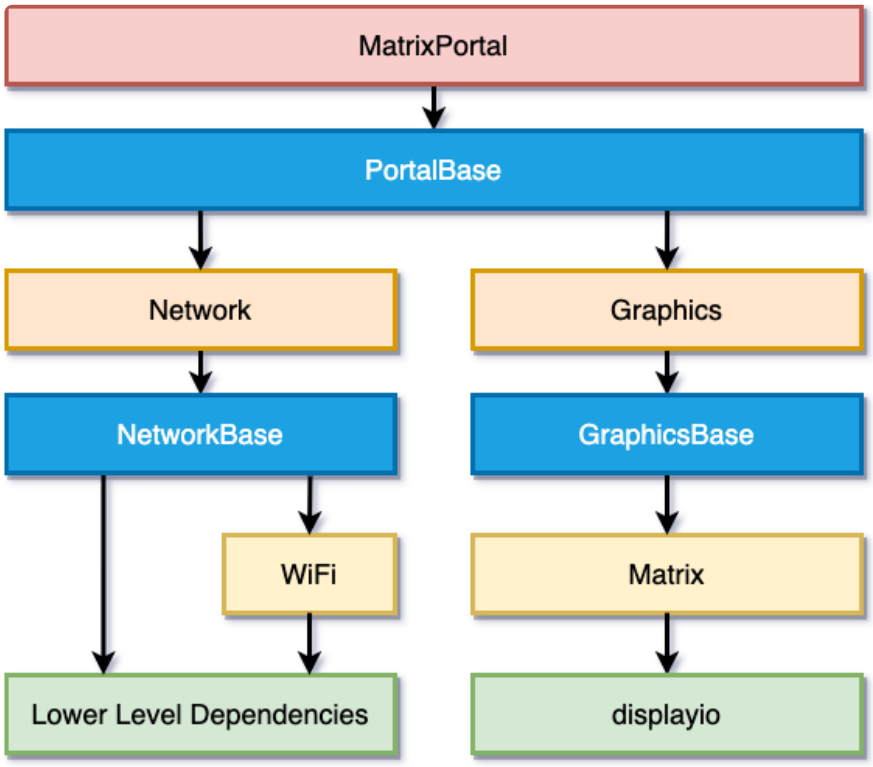
MatrixPortal Library Overview

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 (), 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 ().

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:

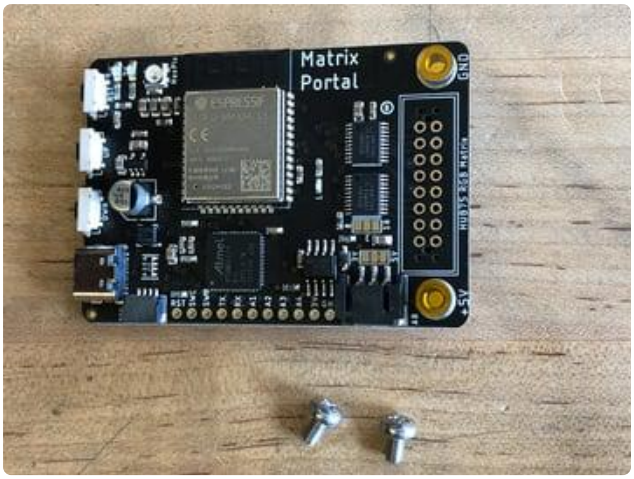
- Text labels are added after the module is initialized.
- Text labels can either be scrolling or static.
- There are more Adafruit IO functions

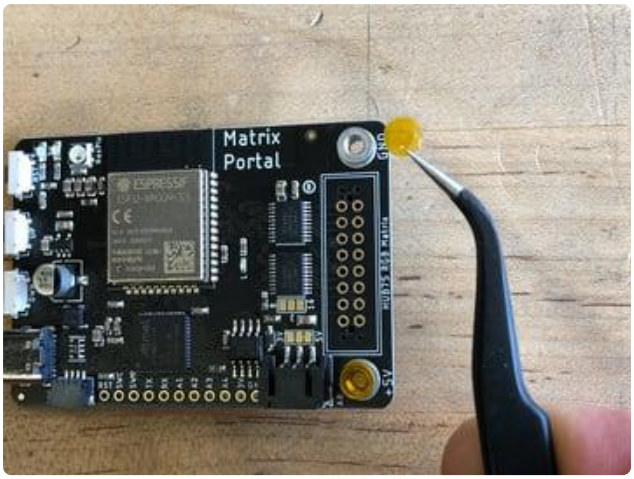
Library Demos

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 ()
- Network Connected RGB Matrix Clock ()
- Weather Display Matrix ()
- Custom Scrolling Quote Board Matrix Display ()
- Halloween Countdown Display Matrix ()
- Moon Phase Clock for Adafruit Matrix Portal ()

Prep the MatrixPortal

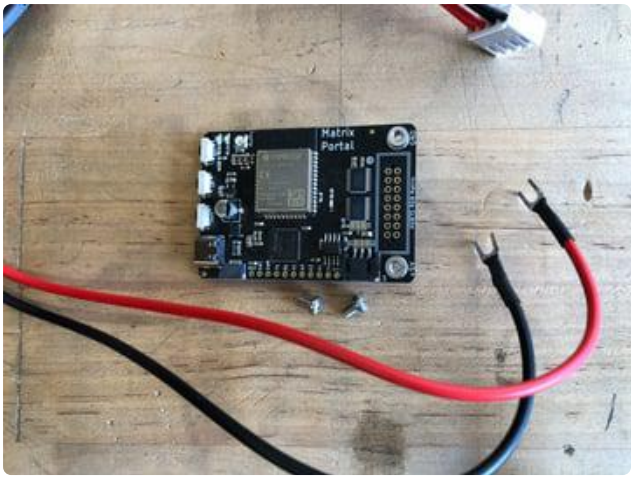


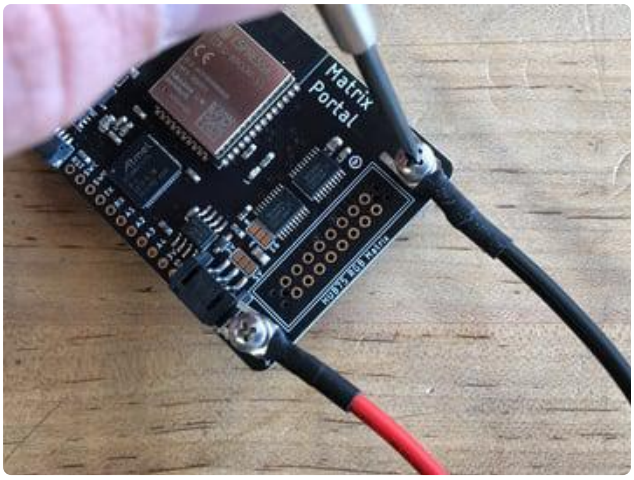


Power Prep

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.

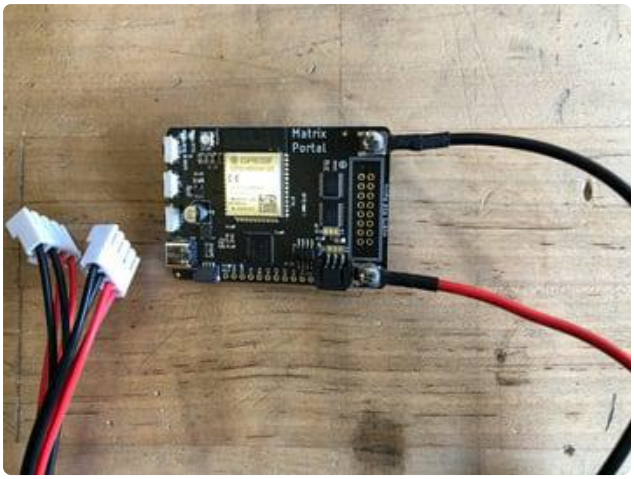


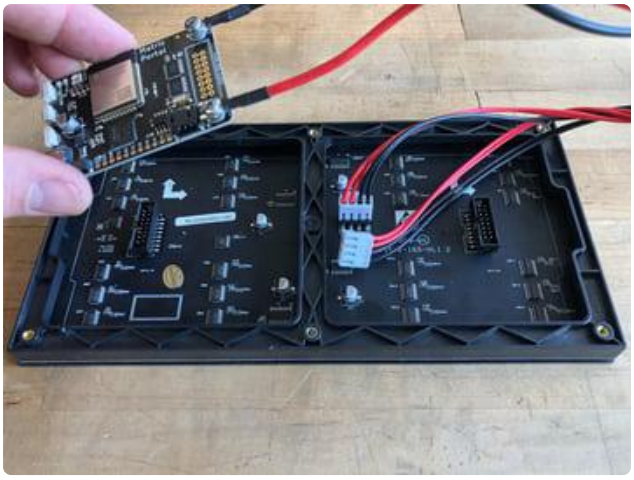


Power Terminals

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

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



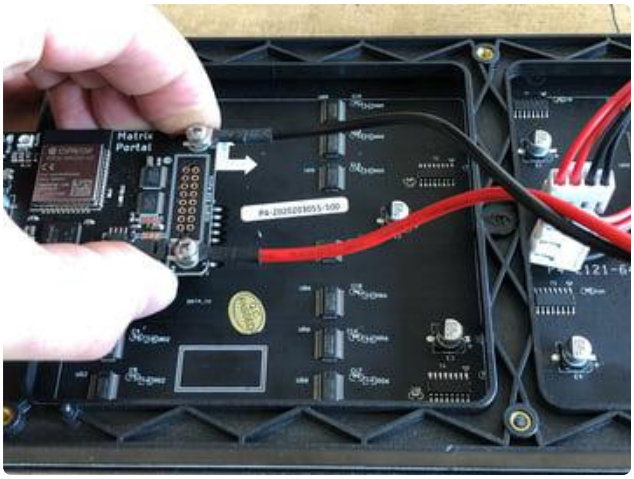


Panel Power

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.

Dual Matrix Setup

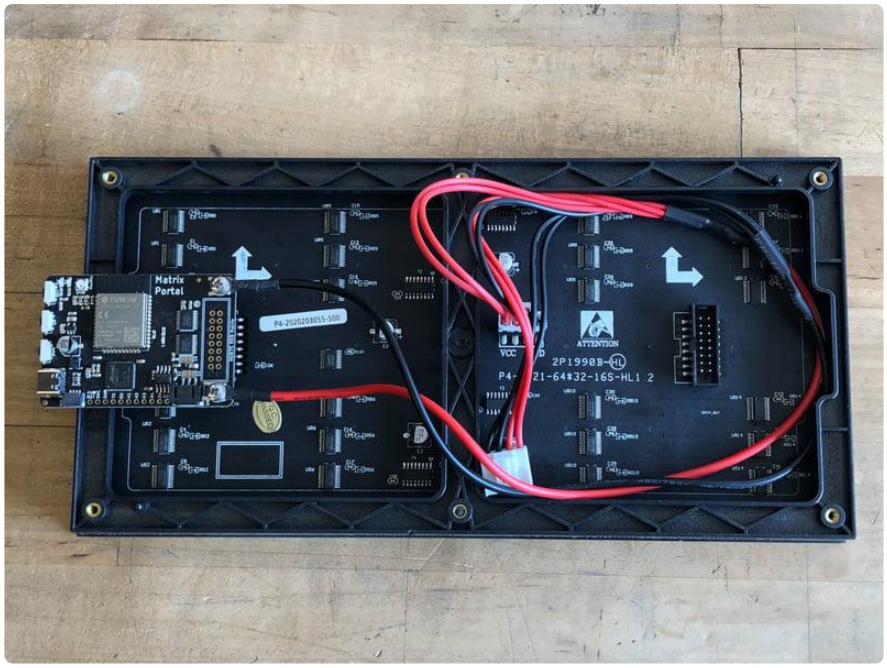
If you're planning to use a 64x64 matrix, follow these instructions on soldering the Address E Line jumper ().



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.

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:

Download latest Library Bundle

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.

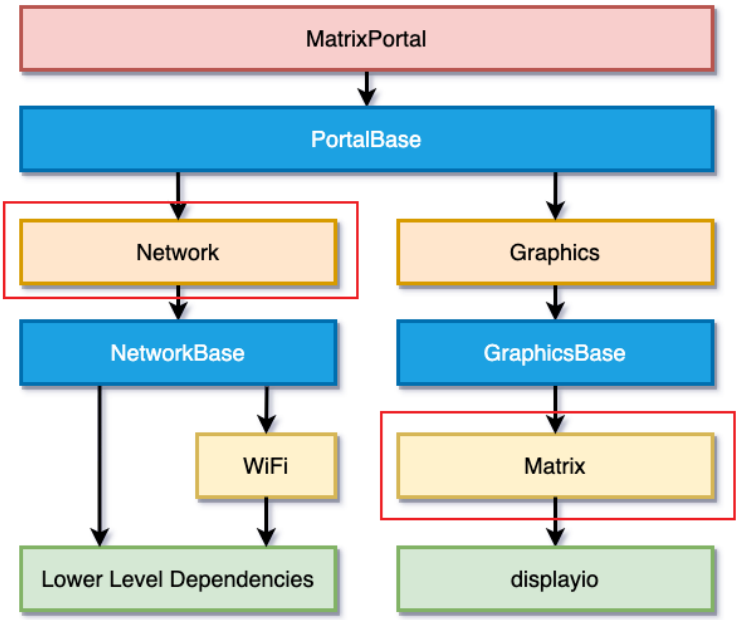
- adafruit_portalbase - This is the base library that adafruit_matrixportal is built on top of.
- adafruit_esp32spi - this is the library that gives you internet access via the ESP32 using (you guessed it!) SPI transport. You need this for anything Internet
- neopixel.mpy - for controlling the onboard neopixel
- adafruit_bus_device - low level support for I2C/SPI
- adafruit_requests.mpy - this library allows us to perform HTTP requests and get responses back from servers. GET/POST/PUT/PATCH - they're all in here!
- adafruit_fakerequests.mpy - This library allows you to create fake HTTP requests by using local files.
- adafruit_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.mpy - this library is used for the onboard accelerometer to detect the orientation of the MatrixPortal
- adafruit_minimqtt - this is used for communicating with MQTT servers.

Choosing Your Layers

Choosing your layers is one of the more important parts of creating a project since it's easy to accidentally choose layers that end up duplicating some of the functions. This guide is intended to help clarify your understanding of the layout so you can make the best choices for your needs.

The PyPortal library, which is what inspired this library was written as a single layer which had the advantage of making it really simple to use for a certain type of project and it worked well for the PyPortal because the hardware setup varies very little between the different models. For the MatrixPortal with having a variety of different panel sizes and other possible configurations, it made more sense to break everything up into layers.

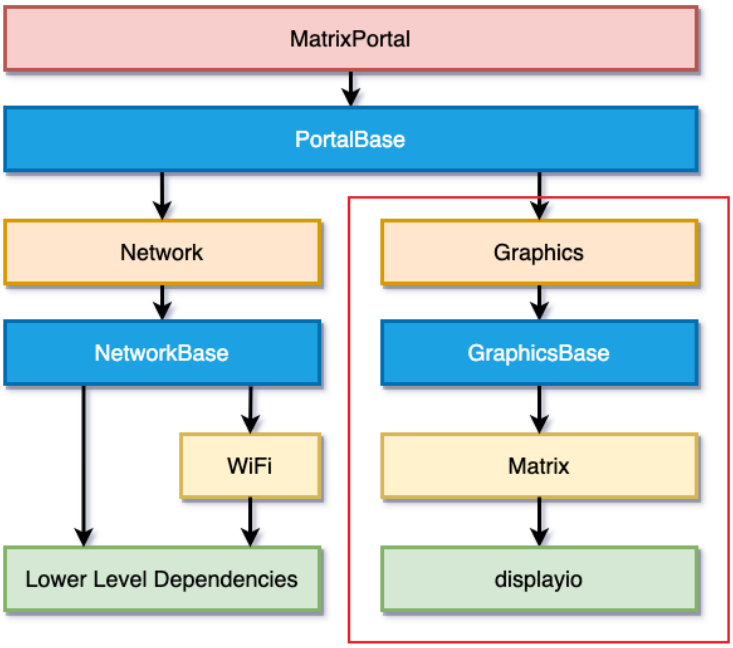
Mixing and Matching Layers



Which of the layers you choose to use for your project depends on the amount of customization and memory management you would like in your project. The higher level up you go in the library layer hierarchy, the more automatic functions you will have available to you, but it also takes away your ability to customize things and uses more memory.

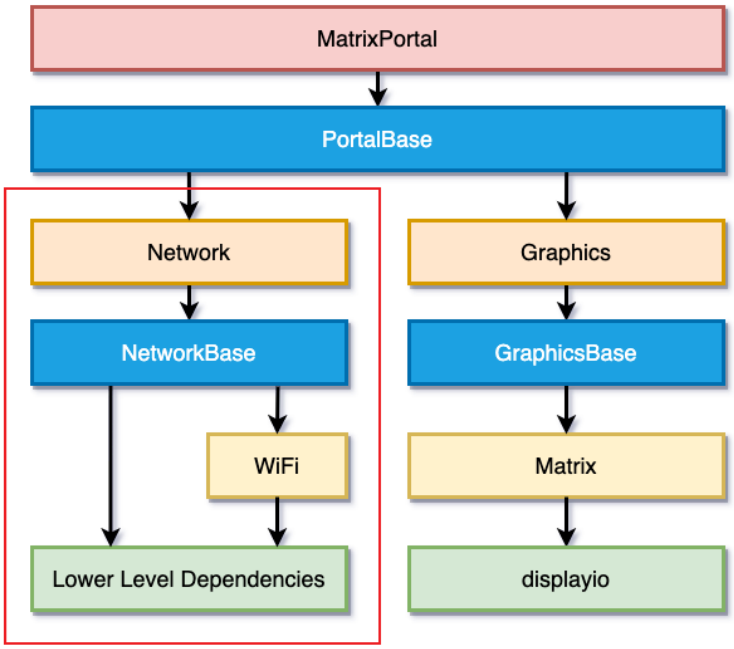
In general, you will likely want at least one of the Graphics layers and optionally one of the Network layers.

Graphics Layers



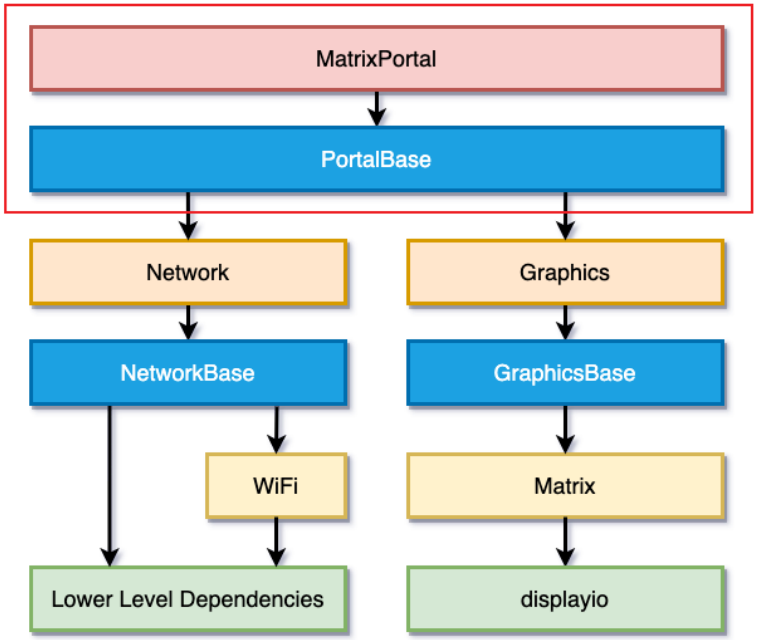
For instance, if you wanted to only have the RGB Matrix automatically initialize, you could just use the matrix layer of the library. If you wanted additional graphics functions too such as easily drawing a background, you could use the graphics layer instead.

Network Layers



On the network functionality side of things, if you just wanted to initialize then you could use the WiFi layer, but if you wanted a lot of the network functions as well, you would use the network layer.

Top Layer



If you wanted everything along with some great functionality that ties both legs of the hierarchy together or you were porting a project from the PyPortal library, then you would want the very top layer, which is the matrixportal layer. This layer was intended to be similar to the PyPortal's single library, but with some notable differences, which we'll cover in this guide.

Remember that if you go with this layer, you should not need to also import any of the lower layers.

Importing your layers

Top Layer

To import the top level layer only, you would simply just import it like this:

```
from adafruit_matrixportal.matrixportal import MatrixPortal
```

If you would like access to the Network and Graphics layers, they are available as objects named `network` and `graphics`. For instance, if you instantiated the top layer as `matrixportal`, then you would access the Network layer with `matrixportal.network` and the Graphics layer with `matrixportal.graphics`.

```
matrixportal = MatrixPortal()
network = matrixportal.network
graphics = matrixportal.graphics
```

Sub-Layers

To only import sub-layers such as the Graphics and Network layers, you would import it like this:

```
from adafruit_matrixportal.graphics import Graphics
from adafruit_matrixportal.network import Network
```

After they're imported, you would just instantiate each of the classes separately.

Code Example

In this page we'll go over the code from one of the many existing projects and take a deeper look at how the projects are making use of the library. The example that we want to take a look at comes from the Custom Scrolling Quote Board Matrix Display () guide. This makes use of the top level matrixportal layer and makes use of several of the key features including mixing static text with scrolling text as well as use of Adafruit IO.

The script starts with importing the standard modules time and random because those are needed for this specific project. It imports board because the onboard NeoPixel is passed into the library to indicate the network connection status. Then the script imports terminalio because the built-in terminal font will be used for displaying the quotes. Finally, we import the `MatrixPortal` class from the matrixportal layer.

```
import time
import random
import board
```

```
import terminalio
from adafruit_matrixportal.matrixportal import MatrixPortal
```

Next it instantiates the `MatrixPortal` class as an object which is stored in the `matrixportal` variable. The `board.NEOPIXEL` is passed in as the `status_neopixel` parameter and debug is set to `True` in order to see extra messages, but this can also be set to `False`.

```
# --- Display setup ---
matrixportal = MatrixPortal(status_neopixel=board.NEOPIXEL, debug=True)
```

Next, the text labels are created. There are two different types that are created here. The first one is a `scrolling` text label and this will scroll in from the right side. This is used for the quotes themselves and is an ideal type when the text is too long to display all at once. One thing to note is that even though the `text_position` is given as an X and Y tuple, the X value is ignored for scrolling text labels and the Y is the position of the center of the label.

An important thing to consider with scrolling labels is that they cannot scroll while network data is being fetched because retrieving data is a blocking operation.

The second label is a static type label, which is used to display the Connecting message. The main reason we are displaying the Connecting message with a static type is because of the blocking issue. We are displaying it at a position of 2 for the X value and it is automatically being centered to the height of the display for the Y value. Just like the first type of label, this label uses the `terminal.FONT` type of font.

The first label that is created has an ID of 0 and each subsequent label is incremented. In the case of these 2 labels, the quotes label is label 0 and the connecting label is label 1. This will become important further down.

```
# Create a new label with the color and text selected
matrixportal.add_text(
    text_font=terminalio.FONT,
    text_position=(0, (matrixportal.graphics.display.height // 2) - 1),
    scrolling=True,
)

# Static 'Connecting' Text
matrixportal.add_text(
    text_font=terminalio.FONT,
    text_position=(2, (matrixportal.graphics.display.height // 2) - 1),
)
```

The next set of variables are a few variables that are intended to be changed to the your preferences. In this case, `QUOTES_FEED` and `COLORS_FEED` are the names of a couple of feeds that are designed to changed to match whatever you set up in

Adafruit IO. The `SCROLL_DELAY` refers to the amount of time to wait between each scrolling animation frame. The smaller the number, the faster the text will scroll. Finally, the `UPDATE_DELAY` is the number of seconds to wait between checking for any changes from Adafruit IO.

```
QUOTES_FEED = "sign-quotes.signtext"
COLORS_FEED = "sign-quotes.signcolor"
SCROLL_DELAY = 0.02
UPDATE_DELAY = 600
```

Next the script sets up some initial values that will be changed once data is downloaded from Adafruit IO.

```
quotes = []
colors = []
last_color = None
last_quote = None
```

Next is the `update_data()` function. It starts out by defining the function, printing to the console that it is updating. Then it displays the Connecting message by setting the text of the label with an ID of 1 to "Connecting".

```
def update_data():
    print("Updating data from Adafruit IO")
    matrixportal.set_text("Connecting", 1)
```

Here we continue with the next part of the `update_data()` function. The code in both of these try blocks is nearly identical except for the fact that they grab the data from two different feeds. It makes use of the `matrixportal.get_io_data()` function to retrieve ALL values from a feed. This is useful if you have a small amount of data in your feed, but each piece of data counts against your data points that your IO account is allowed per minute, so if you are downloading a lot of data at once, you may want to consider this in your design.

The list that the data is to be stored in is cleared. After that, the data which was grabbed in the form of JSON data for each value, is iterated through a for loop as `json_data`. JSON data is laid out in a hierarchical tree very similar to the way that Python lists () and dictionaries () can be laid out. The `matrixportal.network.json_traverse()` function is used to easily traverse down this hierarchical structure and grab the value from the JSON data by providing a path in the form of a list or tuple. That value is appended to the list.

You may have noticed that this last function call was accessing the `network` object inside of `matrixportal`. You can access ALL of the network layer functions through this object.

The reason that these are in a type/except block is because if there are any issues, the script can let you know what the issue is and continue on its way.

```
try:
    quotes_data = matrixportal.get_io_data(QUOTES_FEED)
    quotes.clear()
    for json_data in quotes_data:
        quotes.append(matrixportal.network.json_traverse(json_data, ["value"]))
    print(quotes)
# pylint: disable=broad-except
except Exception as error:
    print(error)

try:
    color_data = matrixportal.get_io_data(COLORS_FEED)
    colors.clear()
    for json_data in color_data:
        colors.append(matrixportal.network.json_traverse(json_data, ["value"]))
    print(colors)
# pylint: disable=broad-except
except Exception as error:
    print(error)
```

In the final part of the `update_data()` function it verifies that it was able to retrieve at least 1 value from each feed. If it was not able to, there's no point in continuing so an error is raised. The connecting message is hidden by setting the label with an ID of 1 to " ", though it could also be set to a completely empty string.

```
if not quotes or not colors:
        raise "Please add at least one quote and color to your feeds"
    matrixportal.set_text(" ", 1)
```

The next part is the final setup right before the main loop. It calls the `update_data()` function, which we went over above to retrieve the data first. It sets the last_update to the current timer value and makes sure "Connecting" isn't being displayed. The `quote_index` and `color_index` variables are set to ensure that there is no possibility that the variables could be read before they are assigned new values.

```
update_data()
last_update = time.monotonic()
matrixportal.set_text(" ", 1)
quote_index = None
color_index = None
```

Next we get to the main loop.

```
while True:
```

The remainder of the code is inside the main loop. The length of quotes is checked and only if we have more than one quote and have already set the quote, we run inside a loop to randomly choose a new index that is different from the last one we

grabbed. If there is only one quote or we don't have an old value, there's no reason to run this while loop and we just grab one at random. We then set the last_quote to the new value we just grabbed which will be used next time.

```
# Choose a random quote from quotes
if len(quotes) &gt; 1 and last_quote is not None:
    while quote_index == last_quote:
        quote_index = random.randrange(0, len(quotes))
else:
    quote_index = random.randrange(0, len(quotes))
last_quote = quote_index
```

In the next block, we do the exact same thing except with the colors.

```
# Choose a random color from colors
if len(colors) &gt; 1 and last_color is not None:
    while color_index == last_color:
        color_index = random.randrange(0, len(colors))
else:
    color_index = random.randrange(0, len(colors))
last_color = color_index
```

After the random indices are chosen, the script sets the text and the color of the quote label. Since the label has an ID of 0, the index doesn't need to be specified.

```
# Set the quote text
matrixportal.set_text(quotes[quote_index])

# Set the text color
matrixportal.set_text_color(colors[color_index])
```

The `matrixportal.scroll_text()` function is called with the scroll delay parameter. This function will scroll the label from offscreen on the right all the way until it is offscreen on the left.

```
# Scroll it
matrixportal.scroll_text(SCROLL_DELAY)
```

Lastly, the amount of elapsed time on the built-in timer is checked against the `UPDATE_DELAY` value that was set near the top. If that amount of time has passed, which in this case is 600 seconds or 10 minutes, the data is updated and the `last_update` variable is set to the current value of the timer.

That's it. It keeps on looping from there.

```
if time.monotonic() &gt; last_update + UPDATE_DELAY:
    update_data()
    last_update = time.monotonic()
```

Full Example Code

As reference, here's the full example code.

```
# SPDX-FileCopyrightText: 2020 John Park for Adafruit Industries
#
# SPDX-License-Identifier: MIT

# Quote board matrix display
# uses AdafruitIO to serve up a quote text feed and color feed
# random quotes are displayed, updates periodically to look for new quotes
# avoids repeating the same quote twice in a row

import time
import random
import board
import terminalio
from adafruit_matrixportal.matrixportal import MatrixPortal

# --- Display setup ---
matrixportal = MatrixPortal(status_neopixel=board.NEOPIXEL, debug=True)

# Create a new label with the color and text selected
matrixportal.add_text(
    text_font=terminalio.FONT,
    text_position=(0, (matrixportal.graphics.display.height // 2) - 1),
    scrolling=True,
)

# Static 'Connecting' Text
matrixportal.add_text(
    text_font=terminalio.FONT,
    text_position=(2, (matrixportal.graphics.display.height // 2) - 1),
)

QUOTES_FEED = "sign-quotes.signtext"
COLORS_FEED = "sign-quotes.signcolor"
SCROLL_DELAY = 0.02
UPDATE_DELAY = 600

quotes = []
colors = []
last_color = None
last_quote = None


def update_data():
    print("Updating data from Adafruit IO")
    matrixportal.set_text("Connecting", 1)

    try:
        quotes_data = matrixportal.get_io_data(QUOTES_FEED)
        quotes.clear()
        for json_data in quotes_data:
            quotes.append(matrixportal.network.json_traverse(json_data, ["value"]))
        print(quotes)
    # pylint: disable=broad-except
    except Exception as error:
        print(error)

    try:
        color_data = matrixportal.get_io_data(COLORS_FEED)
        colors.clear()
        for json_data in color_data:
            colors.append(matrixportal.network.json_traverse(json_data, ["value"]))
```

```
        print(colors)
    # pylint: disable=broad-except
    except Exception as error:
        print(error)

    if not quotes or not colors:
        raise RuntimeError("Please add at least one quote and color to your feeds")
    matrixportal.set_text(" ", 1)


update_data()
last_update = time.monotonic()
matrixportal.set_text(" ", 1)
quote_index = None
color_index = None

while True:
    # Choose a random quote from quotes
    if len(quotes) > 1 and last_quote is not None:
        while quote_index == last_quote:
            quote_index = random.randrange(0, len(quotes))
    else:
        quote_index = random.randrange(0, len(quotes))
    last_quote = quote_index

    # Choose a random color from colors
    if len(colors) > 1 and last_color is not None:
        while color_index == last_color:
            color_index = random.randrange(0, len(colors))
    else:
        color_index = random.randrange(0, len(colors))
    last_color = color_index

    # Set the quote text
    matrixportal.set_text(quotes[quote_index])

    # Set the text color
    matrixportal.set_text_color(colors[color_index])

    # Scroll it
    matrixportal.scroll_text(SCROLL_DELAY)

    if time.monotonic() > last_update + UPDATE_DELAY:
        update_data()
        last_update = time.monotonic()
```

MatrixPortal Library Documentation

MatrixPortal Library Documentation ()

PortalBase Library Documentation

PortalBase Library Documentation ()