adafruit learning system

Creating MagTag Projects with CircuitPython

Created by Melissa LeBlanc-Williams

Last updated on 2021-10-22 11:44:44 AM EDT

Guide Contents

Overview

So you may have heard of the Adafruit MagTag (https://adafru.it/OMb) and you want to get started with building a project using CircuitPython. As the name implies, the MagTag was designed to be a low-powered "tag" that you could attach to your refrigerator with magnetic feet. The MagTag 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 pieces to make creating projects to display on the ePaper display very easy.

This library is built on top of the PortalBase library, which in turn is built on top of **displayio** which is included as part of CircuitPython. It also makes use of the ESP32-S2 **wifi** module, along with some lower-level dependencies, to communicate with server over the internet.

One of the latest features of CircuitPython, which is made easy with this library, is the ability to put the MagTag into deep sleep, which is a very low power mode. We will go over that plus more in this guide.

Parts

Adafruit MagTag Starter Kit - ADABOX017 Essentials

The Adafruit MagTag combines the new ESP32-S2 wireless module and a 2.9" grayscale E-Ink display to make a low-power IoT display that can show data on its screen...

$44.95

In Stock

Add to Cart

Adafruit MagTag - 2.9" Grayscale E-Ink WiFi Display

The Adafruit MagTag combines the new ESP32-S2 wireless module and a 2.9" grayscale E-Ink display to make a low-power IoT display that can show data on its screen even when power...

$34.95

In Stock

Add to Cart

Mini Magnet Feet for RGB LED Matrices (Pack of 4)

Got a glorious RGB Matrix project you want to mount and display in your workspace or home? If you have one of the matrix panels listed below, you'll need a pack of these...

$2.50

In Stock

Add to Cart

Lithium Ion Polymer Battery with Short Cable - 3.7V 420mAh

Lithium-ion polymer (also known as 'lipo' or 'lipoly') batteries are thin, light, and powerful. The output ranges from 4.2V when completely charged to 3.7V. This...

$6.95

In Stock

Add to Cart

Depending on which cables you already have on hand, you may also need one of the following items for coding and powering the MagTag.

Micro B USB to USB C Adapter

As technology changes and adapts, so does Adafruit, and speaking of adapting, this adapter has a Micro B USB jack and a USB C...

$1.25

In Stock

Add to Cart

USB Type A to Type C Cable - approx 1 meter / 3 ft long

As technology changes and adapts, so does Adafruit. This USB Type A to Type C cable will help you with the transition to USB C, even if you're still...

$4.95

In Stock

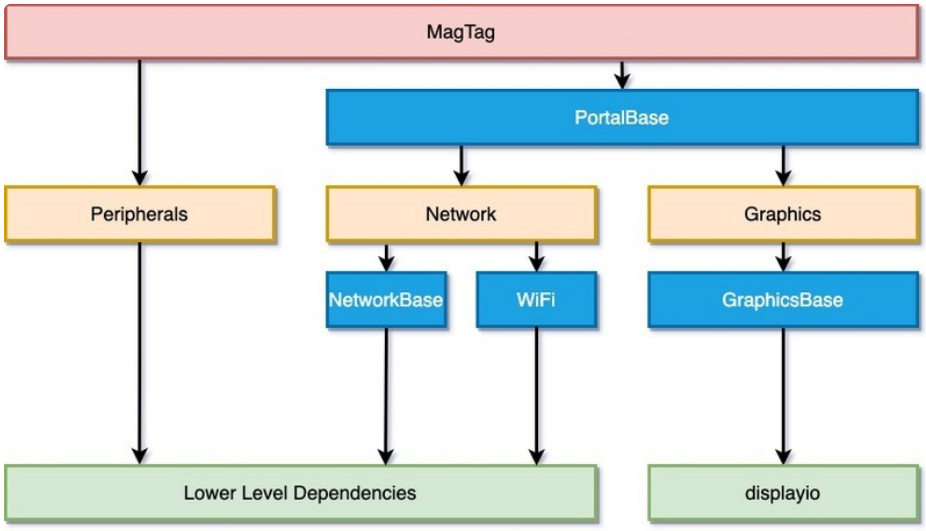
Add to Cart

MagTag Library Overview

The MagTag library was built upon the MatrixPortal library, but was designed around the ePaper display, low power usage, and the new ESP32-S2. While it started as a standalone library, it was designed with the intention of splitting the library up into a base library and the main library which 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.

FakeRequests became its own library, so that is now part of the Lower Level Dependencies in the hierarchy diagram.

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



There are three main branches of dependencies related to Network Functionality, Graphics functionality, and Peripherals. The **MagTag** 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 and controlling the status NeoPixel

colors. The main purpose of this library is to abstract away from the specific WiFi implementation so that PortalBase can be used both with boards with an external WiFi controller and boards with a built-in controller. This layer really wasn't intended to be used directly, but 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. This is built on top of NetworkBase and initializes the WiFi module and anything else specific to the MagTag device.

Graphics Branch

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

Graphics Module

Most of the functionality is part of GraphicsBase. This module contains some convenience functions such as setting the background to a color or image and displaying a QR code. The portion of the Graphics that is part of the MagTag library includes some graphics functionality specific to the eInk display.

Peripherals Branch

This branch is kind of a catch-all of specific board hardware. For the MagTag library, this includes the onboard NeoPixels, buttons, and audio. This library automatically initializes the hardware to make it easier to use it in your scripts. For audio, it will enable and disable the speaker automatically when you play tones. We decided not to automatically disable the NeoPixels so that you could have all the functionality of the NeoPixel library and be able to set the colors differently or even have some of them off.

MagTag Module

The MagTag module is top level module and will handle initializing everything below it. Using this module is very similar to using the MatrixPortal library. The main differences are the peripherals library is new and the display should be refreshed as needed.

Another new feature is the use of the low power modes. By placing the board into deep sleep, the script will exit and a timer will be set so that it restarts the script when it wakes up. For light sleep, it uses a bit more power, but it will resume in the middle of the script. This allows the board to go a long time between charges.

Install CircuitPython

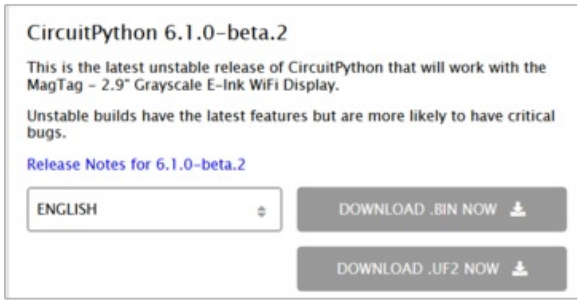
CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files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

Set Up CircuitPython

Follow the steps to get CircuitPython installed on your MagTag.

https://adafru.it/OBd

https://adafru.it/OBd



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

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

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

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

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

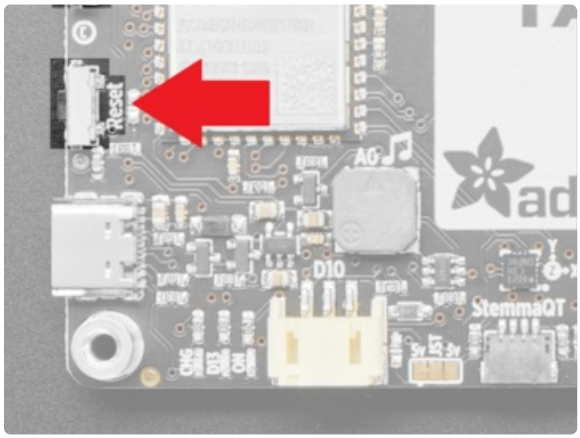
Option 1 - Load with UF2 Bootloader

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

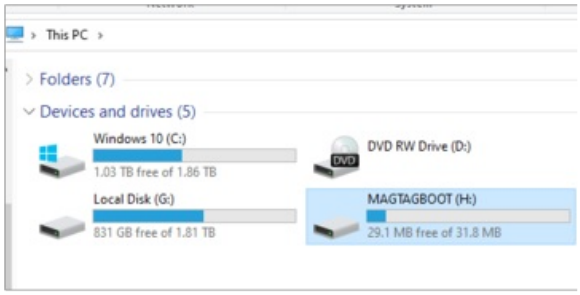
Still, try this first!

Try Launching UF2 Bootloader

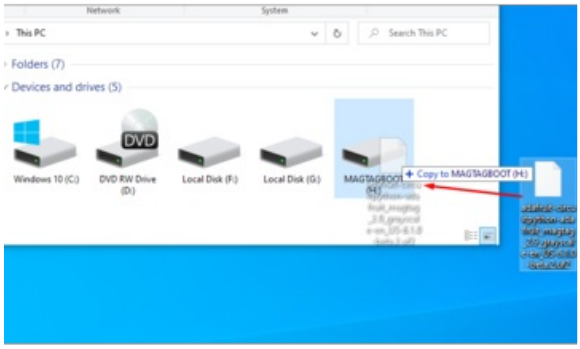
Loading CircuitPython by drag-n-drop UF2 bootloader is the easier way and we recommend it. If you have a MagTag where the front of the board is black, your MagTag came with UF2 already on it.



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



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



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

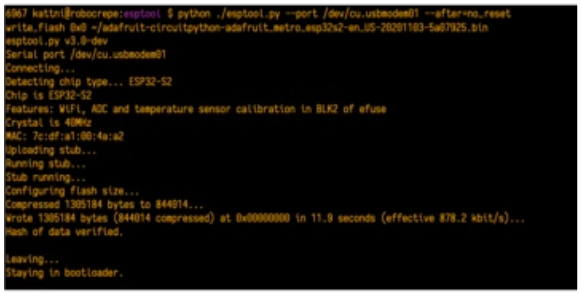
If you're using Windows and you get an error at the end of the file copy that says **Error from the file copy, Error 0x800701B1: A device which does not exist was specified.** You can ignore this error, the bootloader sometimes disconnects without telling Windows, the install completed just fine and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

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

Option 2 - Use esptool to load BIN file

If you have an original MagTag with while soldermask on the front, we didn't have UF2 written for the ESP32S2 yet so it will not come with the UF2 bootloader.

You can upload with **esptool** to the ROM (hardware) bootloader instead!



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

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

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

Press reset to exit the bootloader.

Your **CIRCUITPY** drive should appear!

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

Option 3 - Use Chrome Browser To Upload BIN file

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

MagTag-Specific CircuitPython Libraries

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

Get Latest Adafruit CircuitPython Bundle

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

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. Therefore, you'll need to copy the necessary libraries to your board individually.

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

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

- **adafruit_magtag** - This is a helper library designed for using all of the features of the MagTag, including networking, buttons, NeoPixels, etc.
- **adafruit_portalbase** - This library is the base library that adafruit_magtag is built on top of.
- **adafruit_bitmap_font** - There is fancy font support, and it's easy to make new fonts. This library reads and parses font files.
- **adafruit_display_text** - This library displays text on the screen.
- **adafruit_io** - This library helps connect the MagTag to our free data logging and viewing service

Library files:

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

Secrets

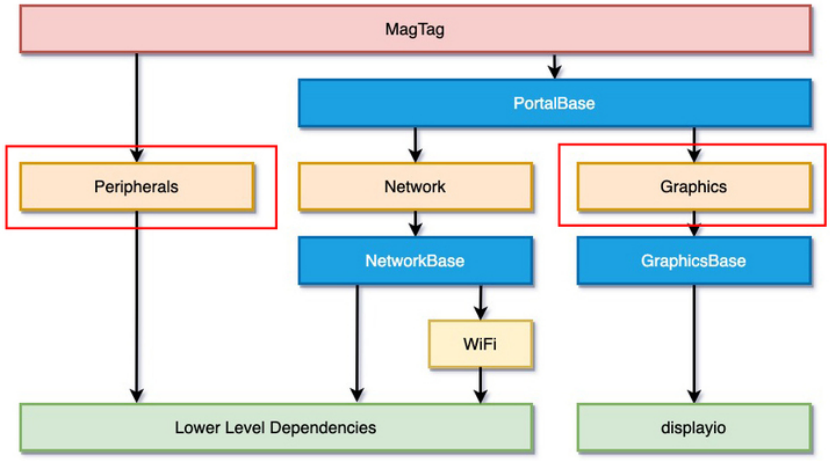
Even if you aren't planning to go online with your MagTag, you'll need to have a **secrets.py** file in the root directory (top level) of your **CIRCUITPY** drive. If you do not intend to connect to wireless, it does not need to have valid data in it. Here's more info on the secrets.py file (https://adafru.it/P3b).

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 MagTag Library, like the similar Matrix Portal library, we decided to break everything up into layers.

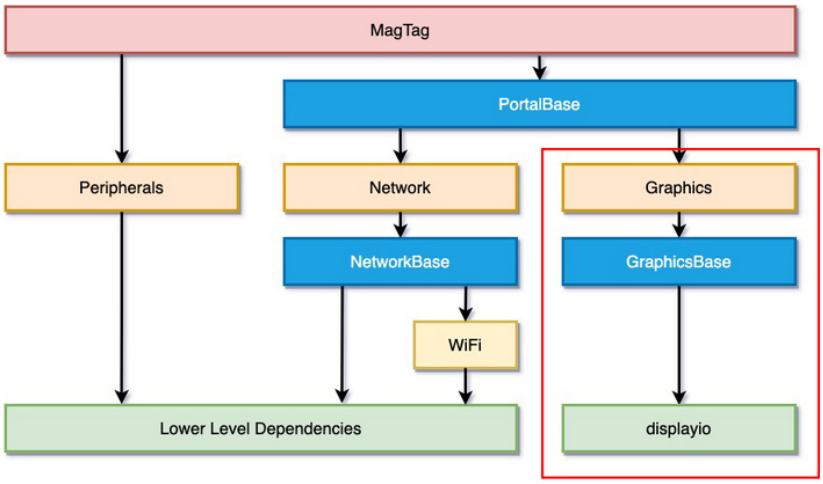
Mixing and Matching Layers



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.

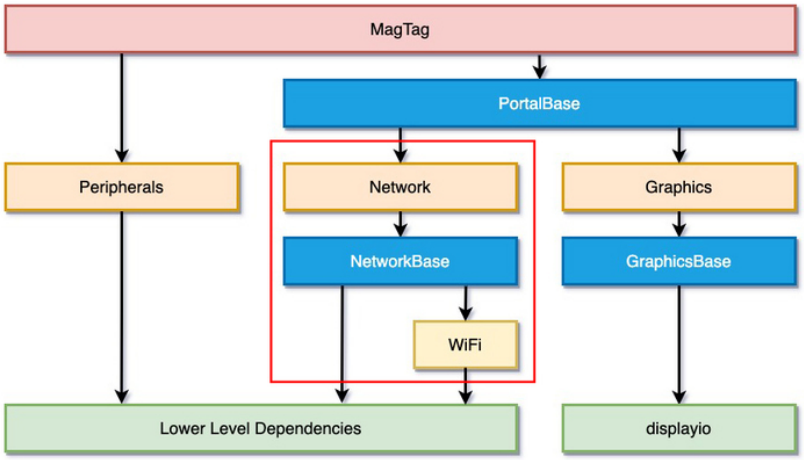
In general, at a minimum, you will likely want at least the Graphics layers and optionally either the Network or Peripheral layers. However, by using the top level layer, you will have access to everything.

Graphics Layers



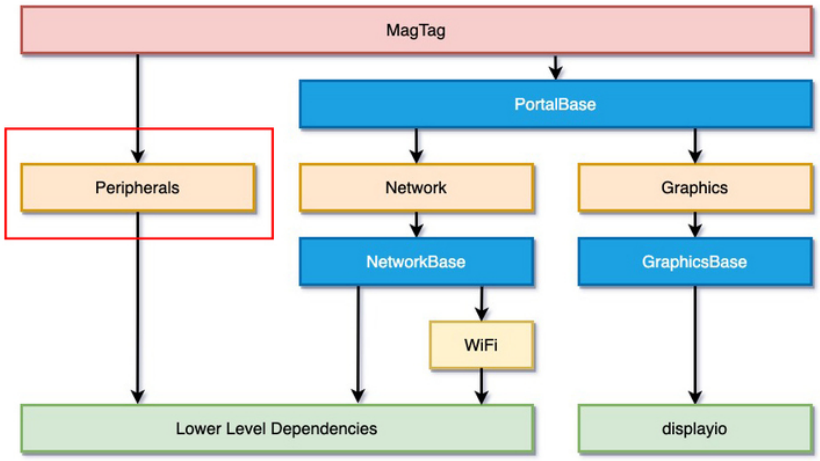
You can use the **graphics** layer if you wanted to have some convenient graphics functions, such as easily drawing a background or displaying a QR code.

Network Layers



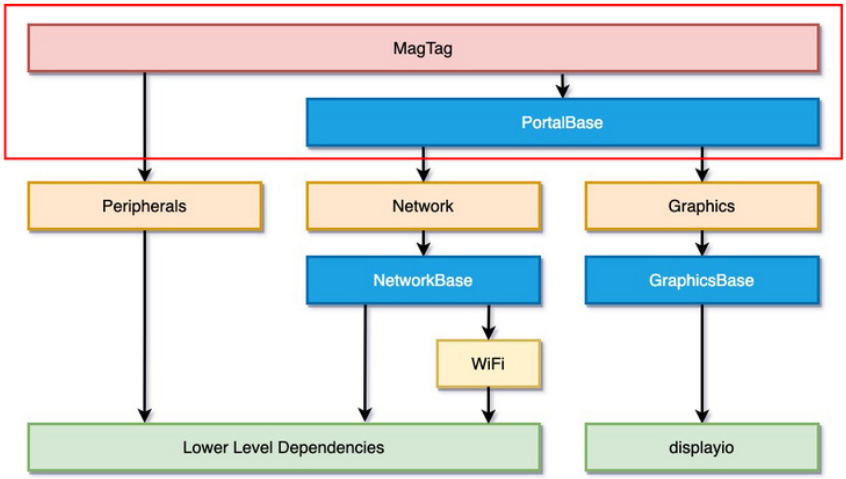
On the network functionality side of things, if you just wanted to initialize the network, you could use the **WiFi** layer, but if you wanted more of the network functions as well, you would use the **network** layer.

Peripherals Layer



To use the peripheral functionality, if you just wanted to initialize the buttons, NeoPixels, and audio, then you could use the **Peripherals** 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 **MagTag** layer. This layer was intended to be similar to the PyPortal's single library or the MatrixPortal's top layer, 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_magtag.magtag import MagTag
```

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 `magtag`, then you would access the Network layer with `magtag.network`, the Graphics layer with `magtag.graphics`, and the Peripherals layer with `magtag.peripherals`.

```
magtag = MagTag()
network = magtag.network
graphics = magtag.graphics
peripherals = magtag.peripherals
```

Sub-Layers

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

```
from adafruit_magtag.graphics import Graphics
from adafruit_magtag.network import Network
from adafruit_magtag.peripherals import Peripherals
```

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

Code Examples

Here is the code from a couple of the examples that are included with the library.

Simple Test

This example uses of the top level **MagTag** layer and makes use of the graphics and peripherals. The code starts out with a couple imports.

```
import time
from adafruit_magtag.magtag import MagTag
```

After that, the MagTag library is initialized with no parameters. A NeoPixel could have been supplied for status, but then there would be one less NeoPixel available for programmatic use.

```
magtag = MagTag()
```

Next a text label is created with a scale of 3, with the left edge 50 pixels in and centered vertically and it's text is set to `Hello World`.

```
magtag.add_text(
    text_position=(
        50,
        (magtag.graphics.display.height // 2) - 1,
    ),
    text_scale=3,
)

magtag.set_text("Hello World")
```

The button colors and tones are set here and are easy to change.

```
button_colors = ((255, 0, 0), (255, 150, 0), (0, 255, 255), (180, 0, 255))
button_tones = (1047, 1318, 1568, 2093)
```

Finally this leads to the main loop. Inside the main loop it will check if each of the buttons has been pressed. If it has, it will print a message about which button, light up the corresponding NeoPixel and then `break`, which causes it to skip the code inside the `else` portion of the `for` loop. After that it will `sleep` for 0.01 seconds and then repeat.

```
while True:
    for i, b in enumerate(magtag.peripherals.buttons):
        if not b.value:
            print("Button %c pressed" % chr((ord("A") + i)))
            magtag.peripherals.neopixel_disable = False
            magtag.peripherals.neopixels.fill(button_colors[i])
            magtag.peripherals.play_tone(button_tones[i], 0.25)
            break
    else:
        magtag.peripherals.neopixel_disable = True
    time.sleep(0.01)
```

After running the code, when you press the buttons, the NeoPixels should light up and each play a different tone.

Full Example Code

```
# SPDX-FileCopyrightText: 2017 Scott Shawcroft, written for Adafruit Industries
#
# SPDX-License-Identifier: Unlicense
import time
from adafruit_magtag.magtag import MagTag

magtag = MagTag()

magtag.add_text(
    text_position=(
        50,
        (magtag.graphics.display.height // 2) - 1,
    ),
    text_scale=3,
)

magtag.set_text("Hello World")

button_colors = ((255, 0, 0), (255, 150, 0), (0, 255, 255), (180, 0, 255))
button_tones = (1047, 1318, 1568, 2093)

while True:
    for i, b in enumerate(magtag.peripherals.buttons):
        if not b.value:
            print("Button %c pressed" % chr((ord("A") + i)))
            magtag.peripherals.neopixel_disable = False
            magtag.peripherals.neopixels.fill(button_colors[i])
            magtag.peripherals.play_tone(button_tones[i], 0.25)
            break
    else:
        magtag.peripherals.neopixel_disable = True
    time.sleep(0.01)
```

Bitcoin Example

This example, which available in many iterations across many pieces of hardware, is one of Adafruit's classic demos. The version for the MagTag is a barebones, stripped down version that just gets the current value, displays it, and then makes use of the deep sleep feature. This example also uses the **MagTag** top layer, but makes use of the graphics and network, but not the peripherals.

The code starts out with a single import.

```
from adafruit_magtag.magtag import MagTag
```

Then it defines where to grab the value from and the path to find the result in the JSON response.

```
# Set up where we'll be fetching data from
DATA_SOURCE = "https://api.coindesk.com/v1/bpi/currentprice.json"
DATA_LOCATION = ["bpi", "USD", "rate_float"]
```

Next, a text transform is defined, which is just a function that accepts and returns a single value. This function just formats the bitcoin value, adds a label, and returns it.

```
def text_transform(val):
    return "Bitcoin: $%d" % val
```

The MagTag object is created here. This time, the `url` and `json_path` parameters are passed in, which are used in the `fetch` function further down.

```
magtag = MagTag(
    url=DATA_SOURCE,
    json_path=DATA_LOCATION,
)
```

The network access point is then connected.

```
magtag.network.connect()
```

The text label is created, but it is being done a little differently here. The center of the display is given as the `text_position`, a scale of 3 is used like in the previous example, and the `text_transform` is passed in here.

The `text_anchor_point` is set here and this tells the library where you want the `text_position` to be used relative to the text label and expects a tuple with values between 0-1. In this case by passing in 0.5, it tells that we want to use the very center of the label. This makes it very easy to center text.

```
magtag.add_text(
    text_position=(
        (magtag.graphics.display.width // 2) - 1,
        (magtag.graphics.display.height // 2) - 1,
    ),
    text_scale=3,
    text_transform=text_transform,
    text_anchor_point=(0.5, 0.5),
)
```

Finally, you may notice that this script does not have a main loop like many other Python scripts. The reason for that is because of the last statement, which we'll explain in a bit.

First a fetch is performed which grabs the bitcoin value, automatically transforms it, and sets it to the label. Second, the `exit_and_deep_sleep` function is called which tells the script to exit, places the device in a very low power mode, basically just enough for it to keep track of the time to wait, and then it restarts from the beginning. In this case, the device is waiting 60 seconds.

```
try:
    value = magtag.fetch()
    print("Response is", value)
except (ValueError, RuntimeError) as e:
    print("Some error occured, retrying! -", e)
magtag.exit_and_deep_sleep(60)
```

When you run the code, it will connect to your access point, display the Bitcoin value, sleep for 60 seconds and repeat.

Full Example Code

```
# SPDX-FileCopyrightText: 2017 Scott Shawcroft, written for Adafruit Industries
#
# SPDX-License-Identifier: Unlicense
from adafruit_magtag.magtag import MagTag

# Set up where we'll be fetching data from
DATA_SOURCE = "https://api.coindesk.com/v1/bpi/currentprice.json"
DATA_LOCATION = ["bpi", "USD", "rate_float"]


def text_transform(val):
    return "Bitcoin: $%d" % val


magtag = MagTag(
    url=DATA_SOURCE,
    json_path=DATA_LOCATION,
)

magtag.network.connect()

magtag.add_text(
    text_position=(
        (magtag.graphics.display.width // 2) - 1,
        (magtag.graphics.display.height // 2) - 1,
    ),
    text_scale=3,
    text_transform=text_transform,
    text_anchor_point=(0.5, 0.5),
)

try:
    value = magtag.fetch()
    print("Response is", value)
except (ValueError, RuntimeError) as e:
    print("Some error occured, retrying! -", e)
magtag.exit_and_deep_sleep(60)
```

MagTag Library Documentation

MagTag Library Documentation (https://adafru.it/Pet)

PortalBase Library Documentation

PortalBase Library Documentation (https://adafru.it/Qen)