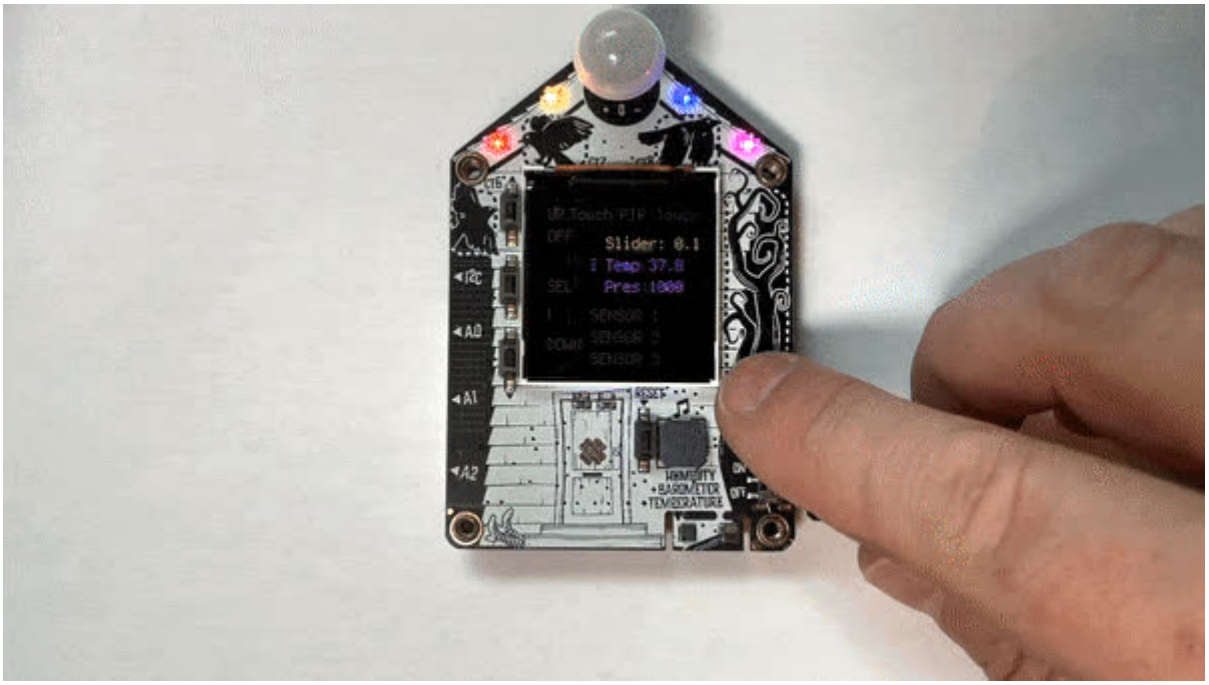
# Creating FunHouse Projects with CircuitPython

Created by Melissa LeBlanc-Williams

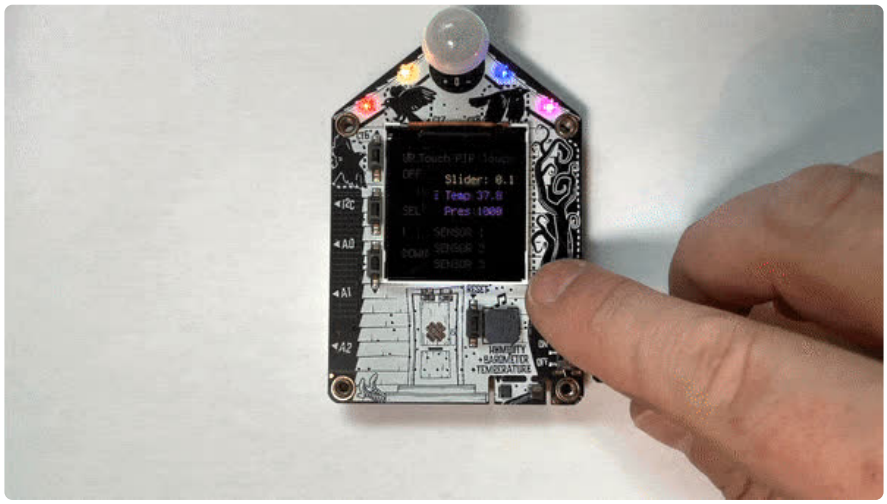



https://learn.adafruit.com/creating-funhouse-projects-with-circuitpython

Last updated on 2025-01-29 04:17:58 PM EST

# Table of Contents

# Overview

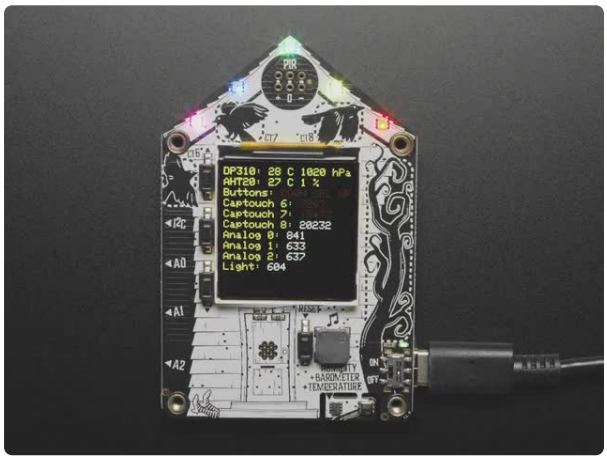


So you may have heard of the Adafruit FunHouse (http://adafru.it/4985) and you want to get started with building a project using CircuitPython. As the name implies, the FunHouse was designed to make **House** or Home Automation **fun**. The FunHouse library makes it really easy to get started with creating a new project, using this board, but it also supports a variety of other hardware pieces to make creating projects. It also makes it very easy to interface with Home Automation software or create a standalone project. The FunHouse doesn't need to be used as a Home Automation project either. Use your imagination and create something with the available features.

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 features that we added to this library to make it easy to interface with Home Automation software is the integration of the CircuitPython MiniMQTT library. This library makes it very easy to send and receive messages with a couple of the popular Home Automation Software packages. We will go over that plus more in this guide.

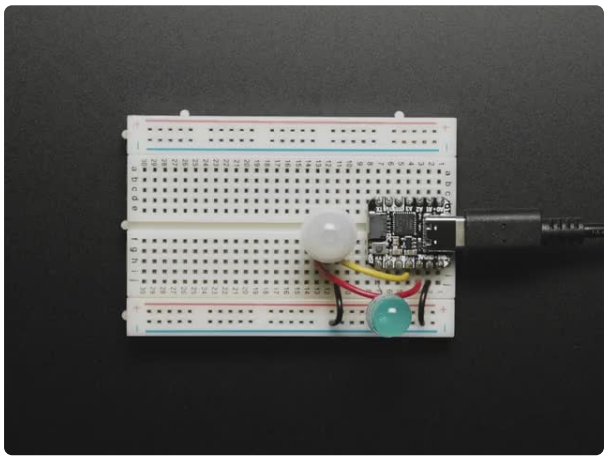
## Parts




[Adafruit FunHouse - WiFi Home Automation Development Board]

Home is where the heart is...it's also where we keep all our electronic bits. So why not wire it up with sensors and actuators to turn our house into an electronic wonderland....

https://www.adafruit.com/product/4985



[Breadboard-friendly Mini PIR Motion Sensor with 3 Pin Header]

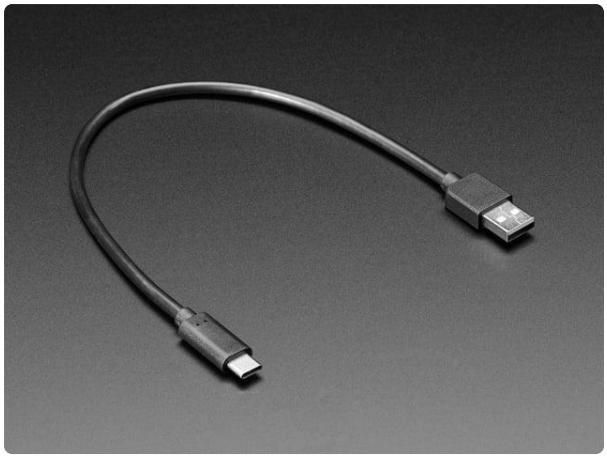
PIR sensors are used to detect motion from pets/humanoids from about 5 meters away (possibly works on zombies, not guaranteed). This sensor is much smaller than most PIR modules, which...

https://www.adafruit.com/product/4871

[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...

https://www.adafruit.com/product/4631



USB Type A to Type C Cable - 1ft - 0.3 meter

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...

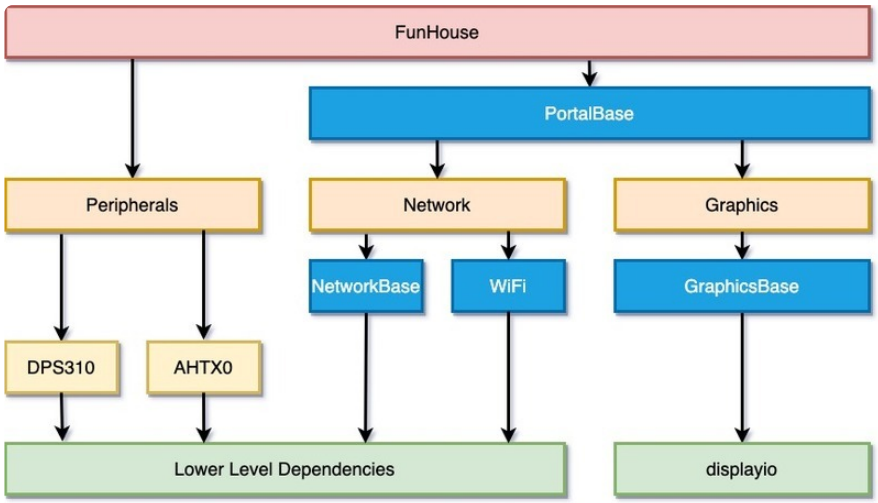
https://www.adafruit.com/product/4473

---

# FunHouse Library Overview

The FunHouse library was built upon the existing Portal-style libraries, taking a lot from the MagTag library due to the similar processor, the ESP32-S2. Like most of the other Portal libraries, it was designed with a layered approach to allow maximum use of code with minimum maintenance. This was achieved by 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.

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




There are three main branches of dependencies related to Network Functionality, Graphics functionality, and Peripherals. The **FunHouse** 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 now part of PortalBase and is responsible for initializing the hardware libraries and controlling the status DotStar 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 was never 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 FunHouse board.

## 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.

## Peripherals Branch

This branch is kind of a catch-all of specific board hardware. For the FunHouse library, this includes the onboard DotStar LEDs, buttons, Capacitive Touchpads, Slider, PIR Sensor, Environmental Sensors and audio. This library automatically initializes the hardware to make it easier to use it in your scripts.

This library also initializes the Temperature, Humidity, and Pressure sensors. To do this, I2C is also automatically initialized and to use I2C with other sensors, you can access it through the i2c object in peripherals branch.

### FunHouse Module

The FunHouse module is top level module and will handle initializing everything below it. Using this module is very similar to using the MagTag library. The main differences are the peripherals library is different and the display is doesn't need to be refreshed.

Another new feature is the MQTT functionality. This may eventually be moved to PortalBase if it is popular and it doesn't cause the PortalBase library to become too large.

---

## Choosing Your Layers

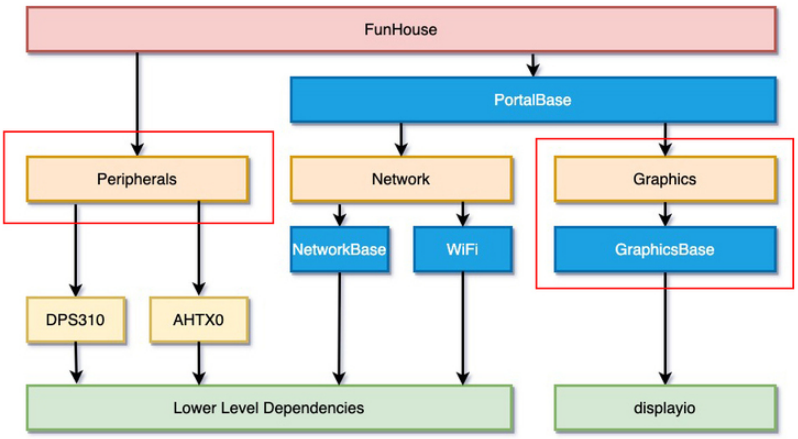
Choosing your layers is one of the more important parts of creating a project since it's easy to accidentally choose layers that end up duplicating some of the functions. This guide is intended to help clarify your understanding of the layout so you can make the best choices for your needs.

The FunHouse library, like the other Portal-based libraries is split up into layers. Layers allow you to use certain portions of the library to save on memory.

The PyPortal library, which is what inspired the other Portal-type libraries was originally written as a single layer. This had the advantage of making it really simple to write and use for specific types of projects. It has now been split into layers like the other Portal-style libraries.

We'll go over the layers that are specific to the FunHouse library.

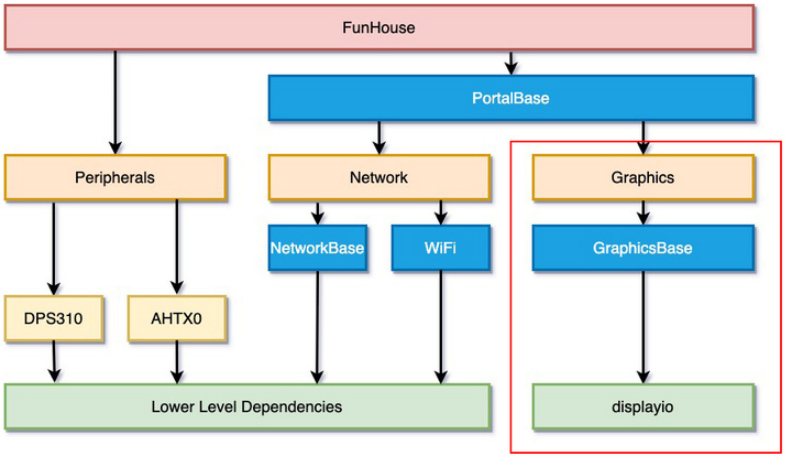
## Mixing and Matching Layers




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.

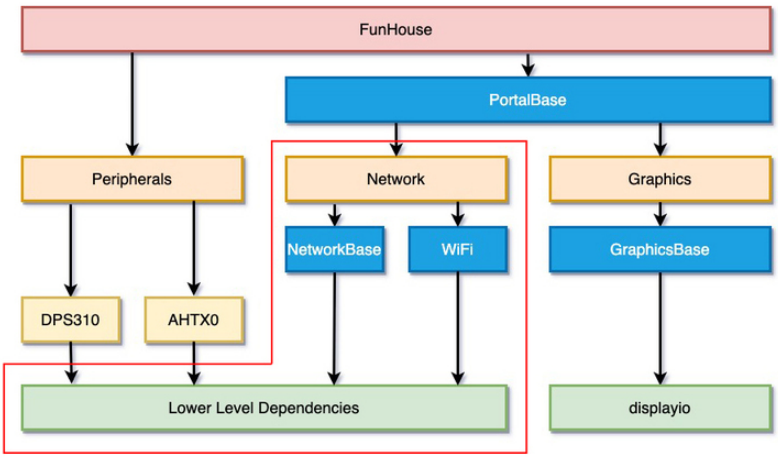
In general, at a minimum, you will likely want at least the Graphics layers and optionally either the Network or Peripheral layers. However, by using the top level layer, you will have access to everything.

## Graphics Layers



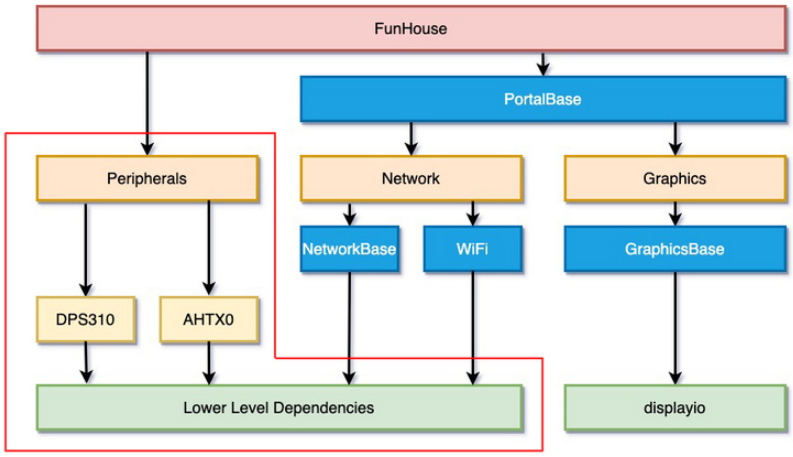

You can use the **graphics** layer if you wanted to have some convenient graphics functions, such as easily drawing a background or displaying a QR code.

## Network Layers



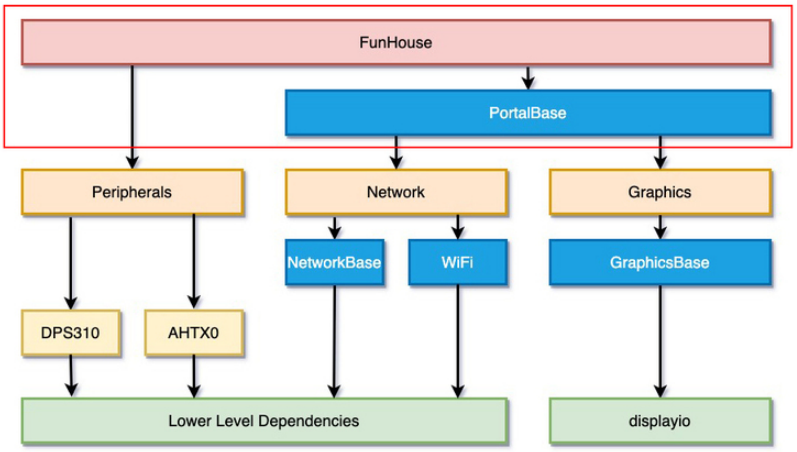

On the network functionality side of things, if you just wanted to initialize the network, you could use the **WiFi** layer, but if you wanted more of the network functions as well, you would use the **network** layer.

## Peripherals Layer




To use the on-board peripheral functionality, such as if you just wanted to initialize the buttons, DotStars, Capacitive Touch Pads, Slider, and audio, then you could use the **Peripherals** layer.

## Top Layer




If you wanted everything along with some great functionality that ties all the legs of the hierarchy together, then you would want the very top layer, which is the **FunHouse** layer. This layer was intended to be similar to the MagTag or MatrixPortal library'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_funhouse import FunHouse
```

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

```
funhouse = FunHouse()
network = funhouse.network
graphics = funhouse.graphics
peripherals = funhouse.peripherals
```

## Sub-Layers

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

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

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

# Basic Examples

Here is the code from a couple of the examples that are included with the library. To run the examples, you will need to have your FunHouse set up with CircuitPython and a **settings.toml** file created. You can find detailed instructions in the Adafruit FunHouse guide (https://adafru.it/RMB). You will also need all of the libraries included in this project bundle:

**Download Project Bundle**

https://adafru.it/XJC

## Simple Test

This example uses of the top level **FunHouse** layer and makes use of the graphics and peripherals. The focus of this example is the usage of the attached peripherals. The code starts out with a few imports.

```
import board
from digitalio import DigitalInOut, Direction, Pull
from adafruit_funhouse import FunHouse
```

After that, the FunHouse library is initialized with a couple of parameters. The DotStars could have been supplied for status, but then we wouldn't have the DotStar available for programmatic use. You could also provide an external NeoPixel, DotStar or RGB LED.

The parameters given were the **default background color** to use and the **scale** so everything can automatically scale up.

```
funhouse = FunHouse(
    default_bg=0x0F0F00,
    scale=2,
)
```

I2C is initialized inside the peripherals layer in order to make use of the environmental sensors. To use I2C for the STEMMA QT port, you can access the I2C object at funhouse.peripherals.i2c.

Next up is a convenience function to allow setting all the DotStar LED colors in a single line.

```
funhouse.peripherals.set_dotstars(0x800000, 0x808000, 0x008000, 0x000080, 0x800080)
```

After that is the external sensor setup for ports A0 through A2. This sets them to be digital inputs using **digitalio**, but **analogio** could have been used as well.

```
# sensor setup
sensors = []
for p in (board.A0, board.A1, board.A2):
    sensor = DigitalInOut(p)
    sensor.direction = Direction.INPUT
    sensor.pull = Pull.DOWN
    sensors.append(sensor)
```

Here's a function to set the color of a specific label index to an on or off color depending on the conditional value. This makes the code much shorter and easier to read.

```
def set_label_color(conditional, index, on_color):
    if conditional:
        funhouse.set_text_color(on_color, index)
    else:
        funhouse.set_text_color(0x606060, index)
```

The following section creates all the labels and stores the index of the labels in variables. This will make accessing specific labels very easy further down in the program. A new parameter that was recently added is the ability to set the initial text of the label, but that comes at a cost. The library will attempt to draw each label as the value is filled in and this has the appearance of if sluggishly initializing. To get around that, the `root_group` property of the `funhouse.display` object is set.

By setting `None` as the value, this informs displayio to not show any layers. At the end, setting this to `funhouse.splash`, which is the layer that everything is drawn to, causes displayio to draw all the newly created labels at the same time.

```
# Create the labels
funhouse.display.root_group = None
slider_label = funhouse.add_text(
    text="Slider:", text_position=(50, 30), text_color=0x606060
)
capright_label = funhouse.add_text(
    text="Touch", text_position=(85, 10), text_color=0x606060
)
pir_label = funhouse.add_text(text="PIR", text_position=(60, 10),
text_color=0x606060)
capleft_label = funhouse.add_text(
    text="Touch", text_position=(25, 10), text_color=0x606060
)
onoff_label = funhouse.add_text(text="OFF", text_position=(10, 25),
text_color=0x606060)
up_label = funhouse.add_text(text="UP", text_position=(10, 10), text_color=0x606060)
sel_label = funhouse.add_text(text="SEL", text_position=(10, 60),
text_color=0x606060)
down_label = funhouse.add_text(
    text="DOWN", text_position=(10, 100), text_color=0x606060
)
jst1_label = funhouse.add_text(
    text="SENSOR 1", text_position=(40, 80), text_color=0x606060
)
```

```
jst2_label = funhouse.add_text(
    text="SENSOR 2", text_position=(40, 95), text_color=0x606060
)
jst3_label = funhouse.add_text(
    text="SENSOR 3", text_position=(40, 110), text_color=0x606060
)
temp_label = funhouse.add_text(
    text="Temp:", text_position=(50, 45), text_color=0xFF00FF
)
pres_label = funhouse.add_text(
    text="Pres:", text_position=(50, 60), text_color=0xFF00FF
)
funhouse.display.root_group = funhouse.splash
```

Finally, is the main loop. We'll break this down into a couple of different sections.

First calling the `funhouse.set_text()` function will set the text. You can just pass in the indices that were stored a bit earlier to make it easy to read. This changes the labels for a couple of the environmental sensors and also prints out some values to the console output.

```
funhouse.set_text("Temp %0.1F" % funhouse.peripherals.temperature, temp_label)
funhouse.set_text("Pres %d" % funhouse.peripherals.pressure, pres_label)

print(funhouse.peripherals.temperature, funhouse.peripherals.relative_humidity)
```

In the next section, the `set_label_color` function that was defined above is used to highlight any sensors that are considered on or **True**.

There is also a section that reads the slider value, which is between `0-1.0` when touched and `None` if it is not touched. The slider value is used to set the brightness of the DotStar LEDs.

```
set_label_color(funhouse.peripherals.captouch6, onoff_label, 0x00FF00)
set_label_color(funhouse.peripherals.captouch7, capleft_label, 0x00FF00)
set_label_color(funhouse.peripherals.captouch8, capright_label, 0x00FF00)

slider = funhouse.peripherals.slider
if slider is not None:
    funhouse.peripherals.dotstars.brightness = slider
    funhouse.set_text("Slider: %1.1f" % slider, slider_label)
set_label_color(slider is not None, slider_label, 0xFFFF00)

set_label_color(funhouse.peripherals.button_up, up_label, 0xFF0000)
set_label_color(funhouse.peripherals.button_sel, sel_label, 0xFFFF00)
set_label_color(funhouse.peripherals.button_down, down_label, 0x00FF00)

set_label_color(funhouse.peripherals.pir_sensor, pir_label, 0xFF0000)
set_label_color(sensors[0].value, jst1_label, 0xFFFFFF)
set_label_color(sensors[1].value, jst2_label, 0xFFFFFF)
set_label_color(sensors[2].value, jst3_label, 0xFFFFFF)
```

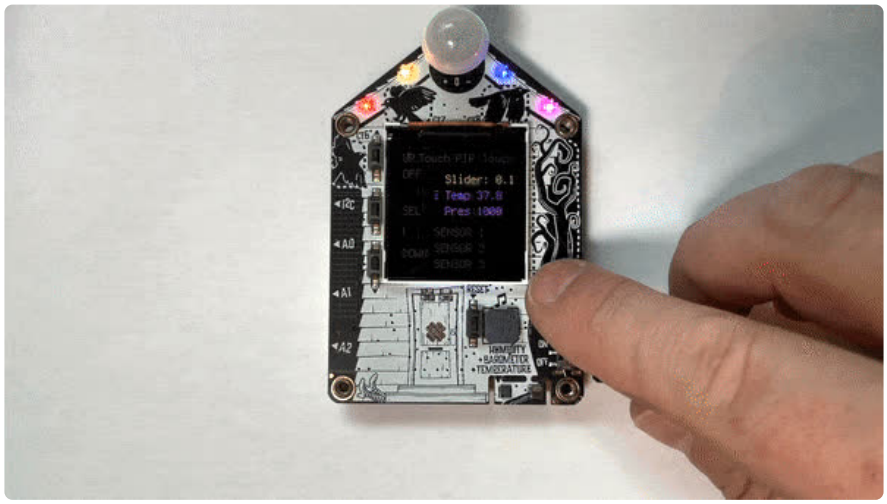
## Full Example Code

Go ahead and click **Download Project Bundle** to download the full example and the required libraries. Rename **funhouse_simpletest.py** to **code.py** and copy all the files over to your FunHouse to run the Simple Test example.

```
# SPDX-FileCopyrightText: 2017 Scott Shawcroft, written for Adafruit Industries
# SPDX-FileCopyrightText: Copyright (c) 2021 Melissa LeBlanc-Williams for Adafruit
Industries
#
# SPDX-License-Identifier: Unlicense
import board
from digitalio import DigitalInOut, Direction, Pull
from adafruit_funhouse import FunHouse

funhouse = FunHouse(
    default_bg=0x0F0F00,
    scale=2,
)

funhouse.peripherals.set_dotstars(0x800000, 0x808000, 0x008000, 0x000080, 0x800080)

# sensor setup
sensors = []
for p in (board.A0, board.A1, board.A2):
    sensor = DigitalInOut(p)
    sensor.direction = Direction.INPUT
    sensor.pull = Pull.DOWN
    sensors.append(sensor)


def set_label_color(conditional, index, on_color):
    if conditional:
        funhouse.set_text_color(on_color, index)
    else:
        funhouse.set_text_color(0x606060, index)


# Create the labels
funhouse.display.root_group = None
slider_label = funhouse.add_text(
    text="Slider:", text_position=(50, 30), text_color=0x606060
)
capright_label = funhouse.add_text(
    text="Touch", text_position=(85, 10), text_color=0x606060
)
pir_label = funhouse.add_text(text="PIR", text_position=(60, 10),
text_color=0x606060)
capleft_label = funhouse.add_text(
    text="Touch", text_position=(25, 10), text_color=0x606060
)
onoff_label = funhouse.add_text(text="OFF", text_position=(10, 25),
text_color=0x606060)
up_label = funhouse.add_text(text="UP", text_position=(10, 10), text_color=0x606060)
sel_label = funhouse.add_text(text="SEL", text_position=(10, 60),
text_color=0x606060)
down_label = funhouse.add_text(
    text="DOWN", text_position=(10, 100), text_color=0x606060
)
jst1_label = funhouse.add_text(
    text="SENSOR 1", text_position=(40, 80), text_color=0x606060
)
jst2_label = funhouse.add_text(
    text="SENSOR 2", text_position=(40, 95), text_color=0x606060
)
jst3_label = funhouse.add_text(
```

```
    text="SENSOR 3", text_position=(40, 110), text_color=0x606060
)
temp_label = funhouse.add_text(
    text="Temp:", text_position=(50, 45), text_color=0xFF00FF
)
pres_label = funhouse.add_text(
    text="Pres:", text_position=(50, 60), text_color=0xFF00FF
)
funhouse.display.root_group = funhouse.splash

while True:
    funhouse.set_text("Temp %0.1F" % funhouse.peripherals.temperature, temp_label)
    funhouse.set_text("Pres %d" % funhouse.peripherals.pressure, pres_label)

    print(funhouse.peripherals.temperature, funhouse.peripherals.relative_humidity)
    set_label_color(funhouse.peripherals.captouch6, onoff_label, 0x00FF00)
    set_label_color(funhouse.peripherals.captouch7, capleft_label, 0x00FF00)
    set_label_color(funhouse.peripherals.captouch8, capright_label, 0x00FF00)

    slider = funhouse.peripherals.slider
    if slider is not None:
        funhouse.peripherals.dotstars.brightness = slider
        funhouse.set_text("Slider: %1.1f" % slider, slider_label)
    set_label_color(slider is not None, slider_label, 0xFFFF00)

    set_label_color(funhouse.peripherals.button_up, up_label, 0xFF0000)
    set_label_color(funhouse.peripherals.button_sel, sel_label, 0xFFFF00)
    set_label_color(funhouse.peripherals.button_down, down_label, 0x00FF00)

    set_label_color(funhouse.peripherals.pir_sensor, pir_label, 0xFF0000)
    set_label_color(sensors[0].value, jst1_label, 0xFFFFFF)
    set_label_color(sensors[1].value, jst2_label, 0xFFFFFF)
    set_label_color(sensors[2].value, jst3_label, 0xFFFFFF)
```

Once it is running, start touching the capacitive touch pads, buttons, and whatever else you can think of to see the example in action.



## MQTT Example

The next example demonstrates how to use the new built-in MQTT functionality with Adafruit IO, though the MQTT functionality can be used with a standard MQTT server as well. To run this, make sure you have added your Adafruit IO information to your **settings.toml** file. If you're not sure how, you can check out this page in the Adafruit FunHouse guide (https://adafru.it/RXe).

For this example, you'll want to create 2 feeds named **buzzer** and **neopixels** on your Adafruit IO (https://adafru.it/fsU) Feeds page. You can also create a Dashboard and add a Color Picker block to associate with the **neopixels** feed and a Momentary Button to associate with the **buzzer** feed. To learn more about how to do this, check out the Getting Started with Adafruit IO (https://adafru.it/Ef8) guide.

This time for imports, there's just the top level `FunHouse` object and the time module.

```
import time
from adafruit_funhouse import FunHouse
```

For initializing the FunHouse library, by passing `None` as the default background, this causes displayio to keep the console up on the display.

```
funhouse = FunHouse(default_bg=None)
funhouse.peripherals.set_dotstars(0x800000, 0x808000, 0x008000, 0x000080, 0x800080)
```

Next up are the handler functions. These allow the script to tell the library what to do when certain events occur. The different events are the `connect`, `disconnect`, `subscribe`, `unsubscribe`, and `message` events. These are the events common to both the Mini MQTT library and the Adafruit IO MQTT library.

In the connect handler, it will subscribe to the `buzzer` and `neopixel` topics. You can change these to whatever suits you, but make sure the names match the feed IDs in the message handler.

Subscribing to a topic will cause the library to listen to the MQTT server for these topics and respond with the message handler whenever it hears something. The library will listen during the `loop()` function, which will be covered in more detail a little lower in the code. To read more about MQTT Topics, you can check out the MQTT Topics section of our All the Internet of Things Protocols (https://adafru.it/Fmb) guide.

```
def connected(client):
    print("Connected to Adafruit IO! Subscribing...")
    client.subscribe("buzzer")
    client.subscribe("neopixels")


def subscribe(client, userdata, topic, granted_qos):
    print("Subscribed to {0} with QOS level {1}".format(topic, granted_qos))


def disconnected(client):
    print("Disconnected from Adafruit IO!")


def message(client, feed_id, payload):
    print("Feed {0} received new value: {1}".format(feed_id, payload))
    if feed_id == "buzzer":
        if int(payload) == 1:
```

```
            funhouse.peripherals.play_tone(2000, 0.25)
    if feed_id == "neopixels":
        print(payload)
        color = int(payload[1:], 16)
        funhouse.peripherals.dotstars.fill(color)
```

The next section will initialize the MQTT library. If you are using Adafruit IO, you will want to call the `init_io_mqtt()`, otherwise you will want to call the `init_mqtt()` function. To work properly, **one** of the **two** functions must be called, which tells the library which underlying libraries to use.

The remaining lines assign the above handlers to their respective properties.

```
# Initialize a new MQTT Client object
funhouse.network.init_io_mqtt()
funhouse.network.on_mqtt_connect = connected
funhouse.network.on_mqtt_disconnect = disconnected
funhouse.network.on_mqtt_subscribe = subscribe
funhouse.network.on_mqtt_message = message
```

The next section will connect to the MQTT server. The library does not automatically connect, so you will need to do this. The reason it doesn't automatically connect is to give you the opportunity to set up whatever you want prior to this.

The next couple of lines set the initial state of the `sensorwrite_timestamp` and `last_pir` state variables. The reason the last_pir variable is set to None as opposed to the initial value of the PIR sensor is so that it will always publish it's state on connection regardless of the value.

```
print("Connecting to Adafruit IO...")
funhouse.network.mqtt_connect()
sensorwrite_timestamp = time.monotonic()
last_pir = None
```

The last block of code is the main loop. We'll break this up into sections as well. As part of the loop, the `mqtt_loop()` will get called during each iteration. This allows the MQTT library to respond appropriately to any messages. By default it has a 1 second timeout, but you can shorten that for your needs. This will make your script more responsive, but it may also fail to respond to all messages if the traffic is pretty busy.

Right after the loop, we read the temperature and barometric pressure and print that to the serial console.

```
funhouse.network.mqtt_loop()

print("Temp %0.1F" % funhouse.peripherals.temperature)
print("Pres %d" % funhouse.peripherals.pressure)
```

During the main loop, we check that 10 seconds has elapsed. If it has, then we publish the temperature, humidity, and pressure to their respective feeds on Adafruit IO.

We also check to see if the PIR sensor has either changed since the last time we checked or if we haven't checked at all. In either case, we publish the current value to its feed and update the `last_pir` variable.

```
# every 10 seconds, write temp/hum/press
if (time.monotonic() - sensorwrite_timestamp) &gt; 10:
    funhouse.peripherals.led = True
    print("Sending data to adafruit IO!")
    funhouse.network.mqtt_publish("temperature", funhouse.peripherals.temperature)
    funhouse.network.mqtt_publish(
        "humidity", int(funhouse.peripherals.relative_humidity)
    )
    funhouse.network.mqtt_publish("pressure", int(funhouse.peripherals.pressure))
    sensorwrite_timestamp = time.monotonic()
    # Send PIR only if changed!
    if last_pir is None or last_pir != funhouse.peripherals.pir_sensor:
        last_pir = funhouse.peripherals.pir_sensor
        funhouse.network.mqtt_publish("pir", "%d" % last_pir)
    funhouse.peripherals.led = False
```

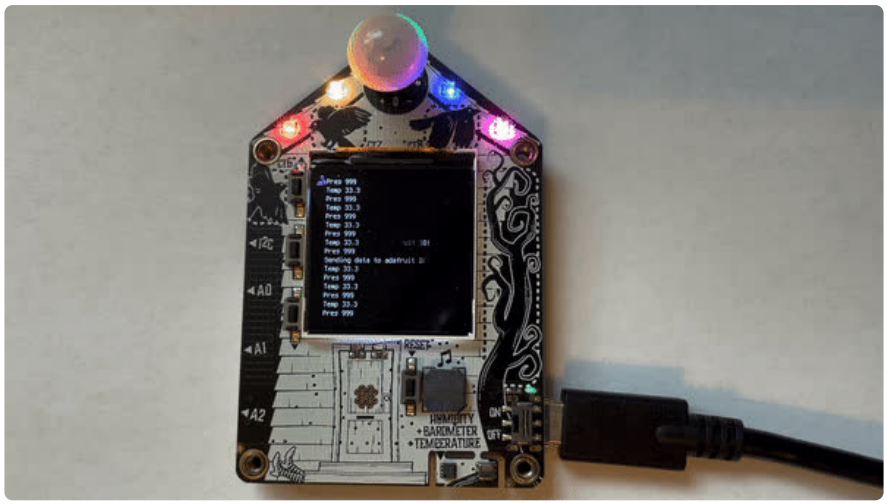
## Full Code Example

Go ahead and click **Download Project Bundle** to download the full example and the required libraries. Rename **funhouse_adafruit_io_mqtt.py** to **code.py** and copy all the files over to your FunHouse to run the Simple Test example. Make sure your **settings.toml** file includes your Adafruit IO information.

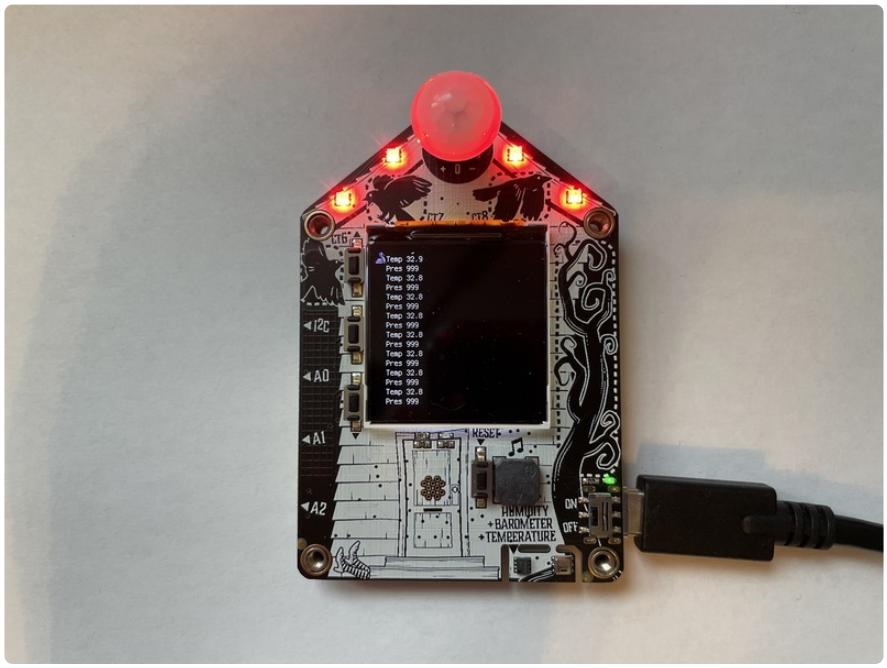
```
# SPDX-FileCopyrightText: 2017 Scott Shawcroft, written for Adafruit Industries
# SPDX-FileCopyrightText: Copyright (c) 2021 Melissa LeBlanc-Williams for Adafruit
Industries
#
# SPDX-License-Identifier: MIT
import time
from adafruit_funhouse import FunHouse

funhouse = FunHouse(default_bg=None)
funhouse.peripherals.set_dotstars(0x800000, 0x808000, 0x008000, 0x000080, 0x800080)


# pylint: disable=unused-argument
def connected(client):
    print("Connected to Adafruit IO! Subscribing...")
    client.subscribe("buzzer")
    client.subscribe("neopixels")


def subscribe(client, userdata, topic, granted_qos):
    print("Subscribed to {0} with QOS level {1}".format(topic, granted_qos))


def disconnected(client):
    print("Disconnected from Adafruit IO!")


def message(client, feed_id, payload):
    print("Feed {0} received new value: {1}".format(feed_id, payload))
    if feed_id == "buzzer":
```

```
        if int(payload) == 1:
            funhouse.peripherals.play_tone(2000, 0.25)
    if feed_id == "neopixels":
        print(payload)
        color = int(payload[1:], 16)
        funhouse.peripherals.dotstars.fill(color)


# pylint: enable=unused-argument

# Initialize a new MQTT Client object
funhouse.network.init_io_mqtt()
funhouse.network.on_mqtt_connect = connected
funhouse.network.on_mqtt_disconnect = disconnected
funhouse.network.on_mqtt_subscribe = subscribe
funhouse.network.on_mqtt_message = message

print("Connecting to Adafruit IO...")
funhouse.network.mqtt_connect()
sensorwrite_timestamp = time.monotonic()
last_pir = None

while True:
    funhouse.network.mqtt_loop()

    print("Temp %0.1F" % funhouse.peripherals.temperature)
    print("Pres %d" % funhouse.peripherals.pressure)

    # every 10 seconds, write temp/hum/press
    if (time.monotonic() - sensorwrite_timestamp) > 10:
        funhouse.peripherals.led = True
        print("Sending data to adafruit IO!")
        funhouse.network.mqtt_publish("temperature",
funhouse.peripherals.temperature)
        funhouse.network.mqtt_publish(
            "humidity", int(funhouse.peripherals.relative_humidity)
        )
        funhouse.network.mqtt_publish("pressure",
int(funhouse.peripherals.pressure))
        sensorwrite_timestamp = time.monotonic()
        # Send PIR only if changed!
        if last_pir is None or last_pir != funhouse.peripherals.pir_sensor:
            last_pir = funhouse.peripherals.pir_sensor
            funhouse.network.mqtt_publish("pir", "%d" % last_pir)
        funhouse.peripherals.led = False
```

Once it is running, you will notice it printing the temperature and pressure to the display. Every 10 seconds it should say **Sending data to adafruit IO!** If you log into Adafruit IO and create a couple of feeds called **buzzer** and **neopixels**, you can try out the subscribe functionality.




To get it to react, you add a `1` to the **buzzer** feed or add a color value such as `#FF0000` for Red to the **neopixels** feed. You should hear a tone played for a brief period for the buzzer feed and you should see the DotStars all light up to the color value you provided.




---

# Temperature Logger Example

If you would like to accurately log the temperature on the FunHouse using the onboard environmental sensors, one potential issue you may notice is that the sensors are near the power supply. The sensors are on an area of the board that is cut out to physically separate them from the power supply components and help mitigate the heating. This helps to an extent, but it is still affected. We wanted to keep the sensors low on the PCB (since heat rises) and also far away from the TFT backlight since it is also very warm.

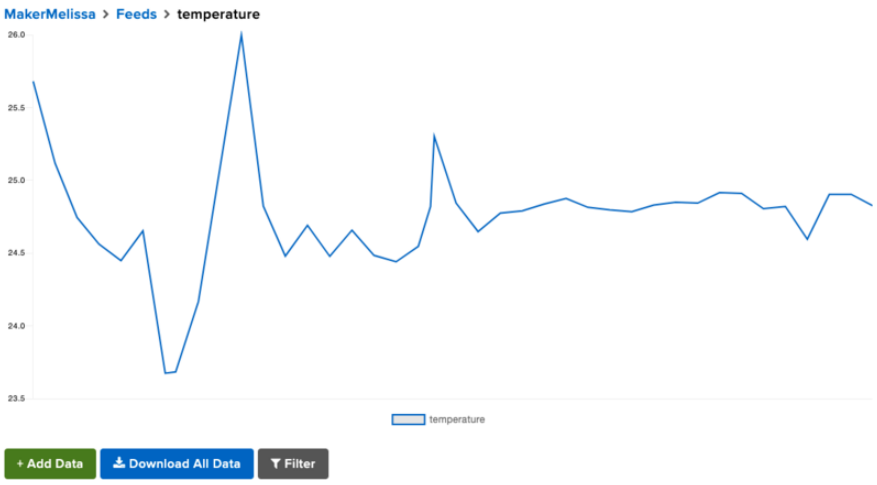
There are a few things you can do to make the temperature more accurate, which we'll cover in this example. Mostly we will keep the board asleep as much as possible and keep the backlight off (or dim) to conserve power.

> If you need precision temperature sensing, an externally connected sensor like an MCP9808 or TMP117, connected over I2C, will give better results as it is physically separable!

## Full Example

Go ahead and click **Download Project Bundle** to download the full example and the required libraries. Rename **funhouse_temperature_logger.py** to **code.py** and copy all the files over to your FunHouse to run the Temperature Logger example. Make sure your **secrets.py** file includes your Adafruit IO information.

Adjust the variables at the top to your liking. You may see the display initialize at first, but once the script runs, it should turn off the backlight. Connect to the serial console (https://adafru.it/S7e) to see the output and when it is logging the data. You can see the data being logged by connecting to Adafruit IO. To learn more, see our Welcome to Adafruit IO (https://adafru.it/BRB) guide.




```
# SPDX-FileCopyrightText: 2017 Scott Shawcroft, written for Adafruit Industries
# SPDX-FileCopyrightText: Copyright (c) 2021 Melissa LeBlanc-Williams for Adafruit
Industries
#
# SPDX-License-Identifier: MIT
"""
This example demonstrates how to log temperature on the FunHouse. Due to the
sensors being near the
power supply, usage of peripherals generates extra heat. By turning off unused
peripherals and back
on only during usage, it can lower the heat. Using light sleep in between readings
will also help.
By using an offset, we can improve the accuracy even more. Improving airflow near
```

```
the FunHouse will
also help.
"""

from adafruit_funhouse import FunHouse

funhouse = FunHouse(default_bg=None)

DELAY = 180
FEED = "temperature"
TEMPERATURE_OFFSET = (
    3  # Degrees C to adjust the temperature to compensate for board produced heat
)

# Turn things off
funhouse.peripherals.dotstars.fill(0)
funhouse.display.brightness = 0
funhouse.network.enabled = False


def log_data():
    print("Logging Temperature")
    print("Temperature %0.1F" % (funhouse.peripherals.temperature -
TEMPERATURE_OFFSET))
    # Turn on WiFi
    funhouse.network.enabled = True
    # Connect to WiFi
    funhouse.network.connect()
    # Push to IO using REST
    funhouse.push_to_io(FEED, funhouse.peripherals.temperature - TEMPERATURE_OFFSET)
    # Turn off WiFi
    funhouse.network.enabled = False


while True:
    log_data()
    print("Sleeping for {} seconds...".format(DELAY))
    funhouse.enter_light_sleep(DELAY)
```

## Code Walkthrough

This example uses of the top level **FunHouse** layer and makes use of the network and peripherals. The code starts out with just the FunHouse import and instantiating the `funhouse` object. Setting `default_bg` to **None** causes the console output to be written to the display, but it doesn't really matter because the backlight will be off all the time.

```
from adafruit_funhouse import FunHouse

funhouse = FunHouse(default_bg=None)
```

Next there are a few variables that you can change to match your needs. `DELAY` is the amount of time in between each time the script checks the temperature and logs it. Decreasing the number will update more often, but because WiFi is turned on to log, it may increase the temperature slightly.

`FEED` is the Adafruit IO feed that you would like to publish to. `TEMPERATURE_OFFSET` is in degrees Celsius and will be subtracted from the measured temperature. The

number will really depend on your specific setup to give more accuracy, so you will likely need to change this number. We will cover this more further down.

```
DELAY = 180
FEED = "temperature"
TEMPERATURE_OFFSET = (
    3  # Degrees C to adjust the temperature to compensate for board produced heat
)
```

We start off by immediately turning off the DotStar LEDs, display backlight, and WiFi.

```
# Turn things off
funhouse.peripherals.dotstars.fill(0)
funhouse.display.brightness = 0
funhouse.network.enabled = False
```

Next we create a function that will briefly turn on the WiFi, connect, and log the data. Then it will immediately shut it back off to reduce the heat. Since we are not maintaining a connection, it makes more sense to use the `push_to_io` function and publish via REST.

```
def log_data():
    print("Logging Temperature")
    print("Temperature %0.1F" % (funhouse.peripherals.temperature -
TEMPERATURE_OFFSET))
    # Turn on WiFi
    funhouse.network.enabled = True
    # Connect to WiFi
    funhouse.network.connect()
    # Push to IO using REST
    funhouse.push_to_io(FEED, funhouse.peripherals.temperature - TEMPERATURE_OFFSET)
    # Turn off WiFi
    funhouse.network.enabled = False
```

Finally there is the main loop. This loop is pretty basic and involves running the `log_data()` function and using the light sleep functionality of the ESP32-S2 processor to conserve power and heat.

```
while True:
    log_data()
    print("Sleeping for {} seconds...".format(DELAY))
    funhouse.enter_light_sleep(DELAY)
```

## Adjusting the Offset

Some factors that can affect the temperature include the airflow around the FunHouse, positioning, and whether it is near insulating materials. For instance, if there is good airflow, this will naturally reduce the temperature. If the FunHouse is standing up as opposed to laying flat against a desk, this will also affect the temperature and is related to airflow. If the FunHouse is inside of a box that insulates the sensor, it will also be higher than if it's not near anything insulating.

This is why we wanted the `OFFSET` variable to be easily settable. We went with about 3 degrees, which is more on the ideal side. One way you can tell the actual room temperature is by leaving the FunHouse unplugged for a while and then turning it on and seeing what the temperature is. It will rise to a certain amount and then you can use the difference to set the offset for your setup.

---

# Adafruit FunHouse Guide

Adafruit FunHouse Guide (https://adafru.it/RQf)

---

# FunHouse Library Documentation

FunHouse Library Documentation (https://adafru.it/RPE)

---

# PortalBase Library Documentation

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