WiFi Mailbox Notifier

Created by Kattni Rembor

https://learn.adafruit.com/wifi-mailbox-notifier

Last updated on 2024-06-03 03:40:58 PM EDT

Table of Contents

Overview

The US Postal Service delivers mail, etc., to your mailbox every week from Monday through Saturday. Amazon packages may be delivered to your mailbox on Sundays. That means up to seven days a week, you may be receiving mail and/or packages to your mailbox. In some cases, mail delivery is incredibly consistent, so you always know when it should arrive. However, in a lot of cases, delivery time varies. You could check multiple times per day, or only check late in the evening, or, instead, you could set up a way to notify you when something has been delivered to your mailbox!

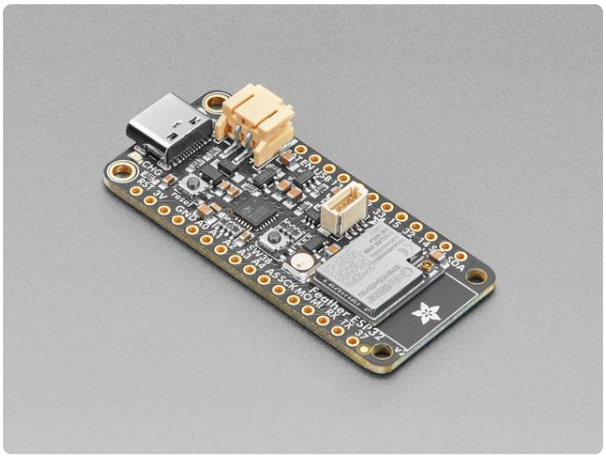
This guide will show you how to use the Feather ESP32 V2, CircuitPython, WiFi, and Adafruit IO to build a notifier that tells you when your mailbox has been opened. Once everything is set up, and the Feather is installed in your mailbox, you'll receive an email each time the door is opened!

This project requires that your mailbox be within range of a WiFi hotspot to which you have access, such as those found in your home. In the event that WiFi is not available, you should consider other radio options beyond the WiFi discussed in this guide.

Parts List

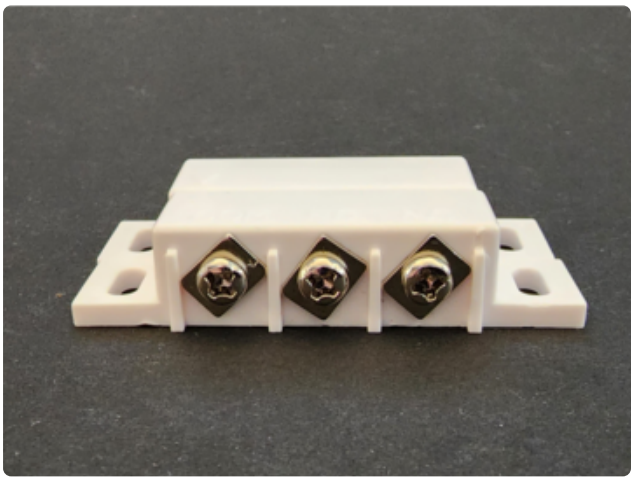
The following is required to build this project.

The Feather ESP32 V2 with the antenna connector will let you connect from further away.



Adafruit ESP32 Feather V2 w.FL Antenna - 8MB Flash + 2 MB PSRAM
One of our star Feathers is the Adafruit HUZZAH32 ESP32 Feather - with the fabulous ESP32 WROOM module on there, it makes quick work...
https://www.adafruit.com/product/5438

A reed switch detect when the mailbox door has been opened so the Feather can notify you.



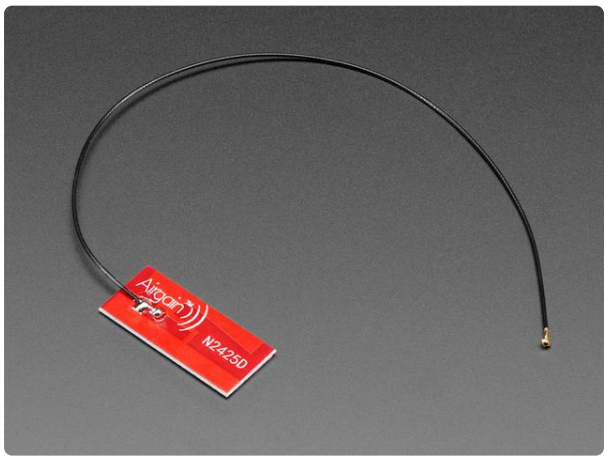
A **normally closed-capable magnetic reed switch**. Reed switches are available as normally closed (NC), normally open (NO), or both. **Normally closed is required for this project.** The guide build uses a switch that does both, but only the NC connection is used.

Here is an example (https://adafru.it/113d) available on Amazon. Magnetic reed switches are available from many sources, so feel free to order from wherever you prefer. Simply make sure you check the specs before purchasing to ensure you are getting a NC-capable switch.

A battery is required, the larger the better. The following battery is ideal, but if it's unavailable, you can downsize to 6600mAh or 4400mAh if needed.

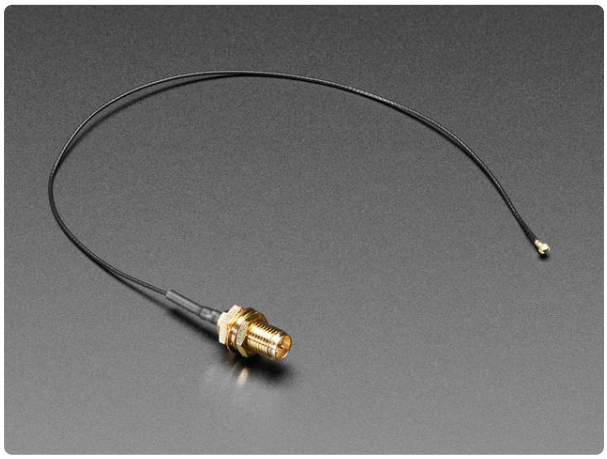
Lithium Ion Battery - 3.7V 10050mAh (10 Ah)
Lithium-ion (also known as 'liion' or 'li-ion') batteries are thin, light, and powerful. The output ranges from 4.2V when completely charged to 3.7V. This battery...
https://www.adafruit.com/product/5035

An antenna is also required. Here are three options to fit various needs.



WiFi Antenna with w.FL / MHF3 / IPEX3 Connector
That's one slim cellular antenna! At about 220mm long from tip to tip and with a cable thickness of just 0.8mm, this 2.4GHz WiFi or BLE antenna is slim, compact, and...
https://www.adafruit.com/product/5445

If you would prefer a larger antenna, you can purchase the adapter, and one of the antennae shown below. You do not need both antennae.



RP-SMA to w.FL / MHF3 / IPEX3 Adapter
Most antennas have SMA or RP-SMA connectors on them, how are you going to connect them to your little RF module? This little cable will bridge the two!
This adapter cable is...
https://www.adafruit.com/product/5444

2.4GHz Dipole Swivel Antenna with RP-SMA - 2dBi

This 4"/100mm long swivel dipole antenna has 2dBi of gain and 50Ω impedance so it will work fantastically with just about any 2.4-2.5GHz wireless receiver/transmitter such...

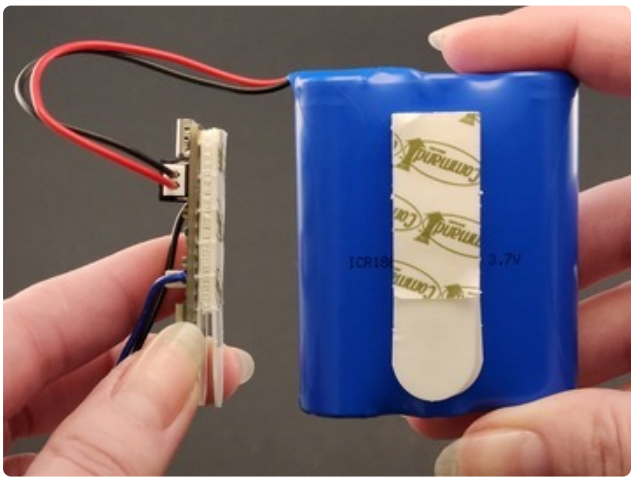
https://www.adafruit.com/product/944

2.4GHz Dipole Swivel Antenna with RP-SMA - 5dBi

This 8"/200mm long swivel dipole antenna has 5dBi of gain and 50Ω impedance so it will work fantastically with just about any 2.4-2.5GHz wireless receiver/transmitter such...

https://www.adafruit.com/product/945

Optional but Highly Suggested

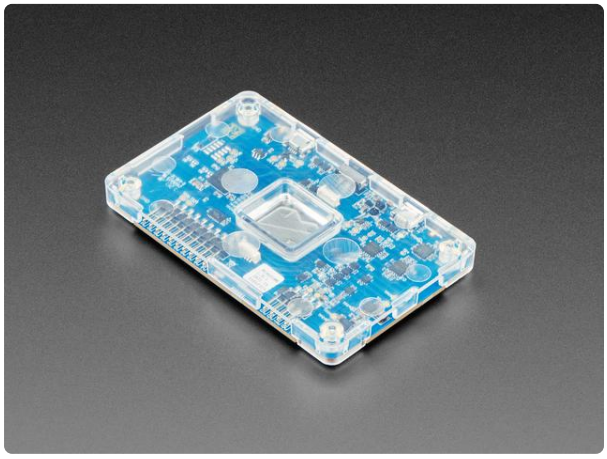


To make it easier to remove the hardware from the mailbox, this project uses Command Small Picture Hanging Strips (https://adafru.it/18fD), which are adhesive hook and loop fasteners used to mount things.

If permanent adhesive is used, it will make swapping or charging the battery, and updating CircuitPython on the Feather, much more difficult. Therefore, some way to remove things more easily is suggested. These strips are one option.

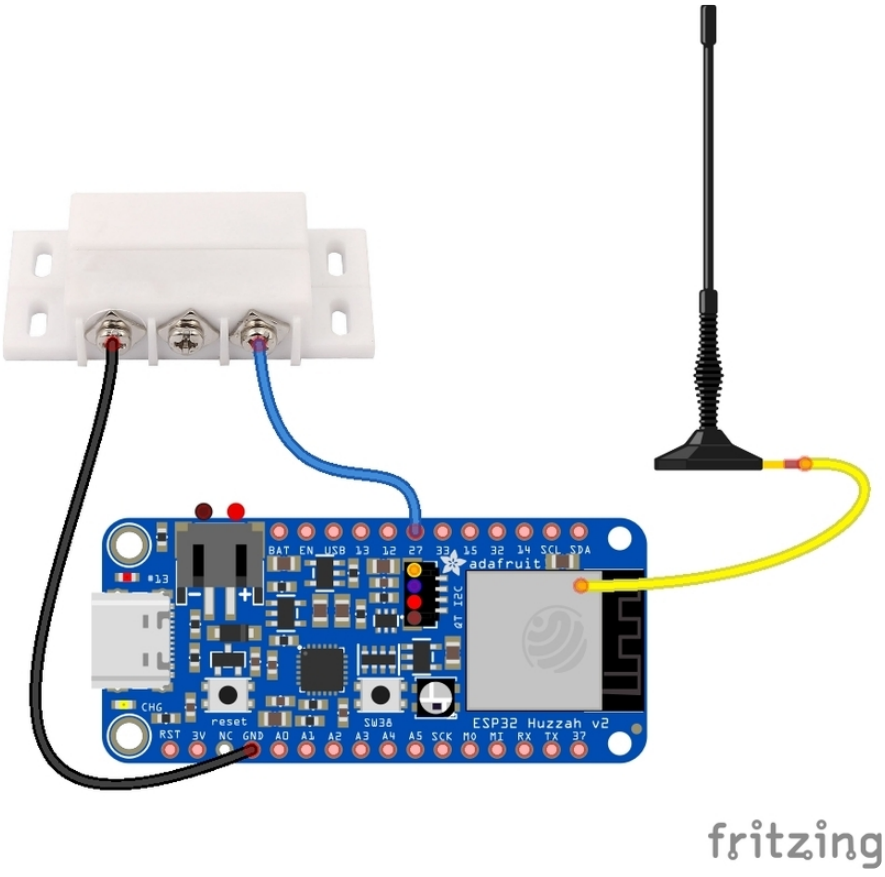
Optional

For measuring the current draw of the project, for example during deep sleep, a Nordic PPK2 is a lifesaver. This project has taken low power into account, but if you wish to verify your own build or another design, this is the instrument to have.



Nordic nRF-PPK2 - Power Profiler Kit II
The Power Profiler Kit II is a standalone unit, which can measure and optionally supply currents all the way from sub-uA and as high as 1A on all Nordic DKs, in...
https://www.adafruit.com/product/5048

Circuit Diagram



Reed Switch

- **Feather pin 27** to **reed switch NC terminal**
- **Feather GND** to **reed switch common terminal**

See the next page on ensuring you have the switch connections correct.

Antenna

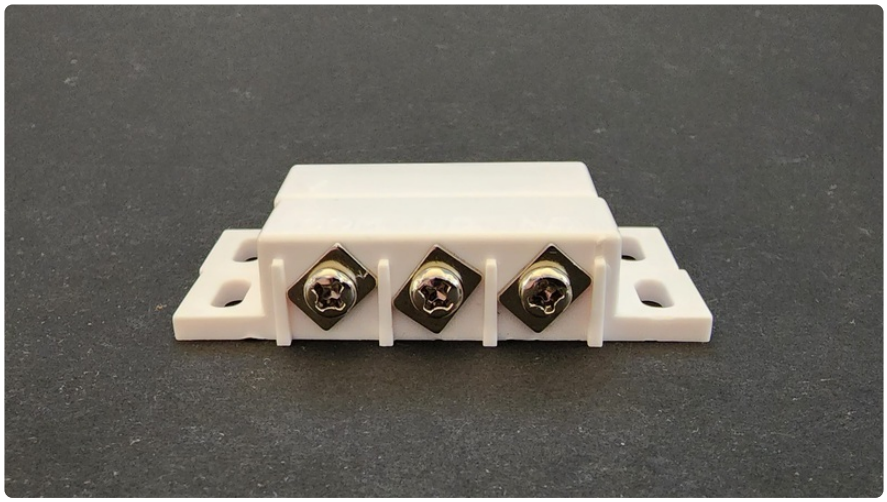
- **Plug the antenna w.FL connector** into the **Feather w.FL connector**

If you are using one of the bigger antennae with an adapter cable, use the following instructions.

- **Antenna SMA connector** into **SMA on the w.FL adapter cable**
- **Adapter cable w.FL connector** into **Feather w.FL connector**

Identify Your Reed Switch

This project uses a reed switch to determine when the mailbox door has been opened, and when it is closed again. Reed switches come in many form factors, including like the one shown below which is designed to be mounted onto something. In this case, the switch itself is contained within the half that has the three screw terminals on it. The other half is the matching magnet.



A reed switch is like any other switch, for example, a push button switch. However, instead of activating the switch with your finger, you activate it with a magnet.

The are two forms of reed switches that are most common. The first one is a normally open (NO) reed switch, which has two terminals. The second one is both NO and normally closed (NC), which has three terminals, NO, NC, and common. Standalone NC reed switches exist, but are highly uncommon and difficult to find.

For this project, you are required to use a switch that has a NC terminal. Since NC reed switches are rare, that typically means using a reed switch that has both NC and NO functionality.

Reed switches are in their initial state when the magnet is not present. As indicated by the names, a normally open (NO) switch's initial state is open, and a normally closed (NC) switch's initial state is closed.

For a NC switch, when there is no magnet present, the circuit between the normally closed terminal and the common terminal is completed, which means electricity is flowing through the circuit. When the magnet is introduced, the switch opens, which means no electricity is flowing through it. (NO switches are the opposite.)

When purchasing three-terminal reed switches, depending on where or what manufacturer you get your reed switch from, the terminals or wires may not come labeled, or, even worse, they may be labelled the opposite of what you expect (NC is marked as NO, and vice versa). This page is here to guide you through identifying the appropriate reed switch terminals to connect to your Feather. **Starting here can save you a lot of troubleshooting later.**

There are two options provided to identify the terminals on your reed switch: using a multimeter, or running a CircuitPython program. The multimeter method requires to you to have a multimeter. The program method is most easily done using your microcontroller on a breadboard.

Multimeter Continuity Mode

First, power your multimeter on and set it to **continuity mode**. If you're unsure how to do this, check the Multimeter guide (https://adafru.it/113a) for details.

The continuity setting on a multimeter is measuring resistance in ohms. When there is no continuity, multimeters will show various things on the display, including 1, OL, or -1. To sort out how your specific multimeter displays continuity and a lack thereof, try the following.

First, verify what your display shows when there is no continuity.

Make sure the probes are separated, and you will see what the initial display is for your specific multimeter. The multimeter used for this demonstration displays OL.

Then, verify how your multimeter indicates continuity.

Bring the probe tips together. You will see the display update with a number of ohms. As well, you may hear a beep, but only if the beep is a feature of your multimeter's continuity mode.

These are the two different results you will be looking for while identifying your reed switch terminals.

When checking for continuity, it doesn't matter which probe is held against which end of the circuit. Continuity is indicated when the circuit is complete, which is irrelevant to probe location.

Using a Multimeter to Identify your Reed Switch Terminals

The following steps will walk you through how to use a multimeter to determine which switch terminal is **common**, **NC**, and **NO**.

Begin with **the magnet separated from the reed switch**.

Choose **any two terminals**, and **hold each of the multimeter probes against one terminal**.

If the multimeter doesn't show continuity, there is no continuity between the two terminals.

Check the other terminal pairs, by moving the probes around.

When continuity is indicated, this means you have found the **common** and the **normally closed (NC)** terminals. However, you don't know which is which yet.

For the purposes of this project, **this is enough information**. This project does not care which terminal is wired to which pin, as long as they are the **common** and **NC** terminals. The Soldering and Assembly section will refer to connecting common to the ground pin, and normally closed to a specific digital pin. However, either terminal can be connected to either pin.

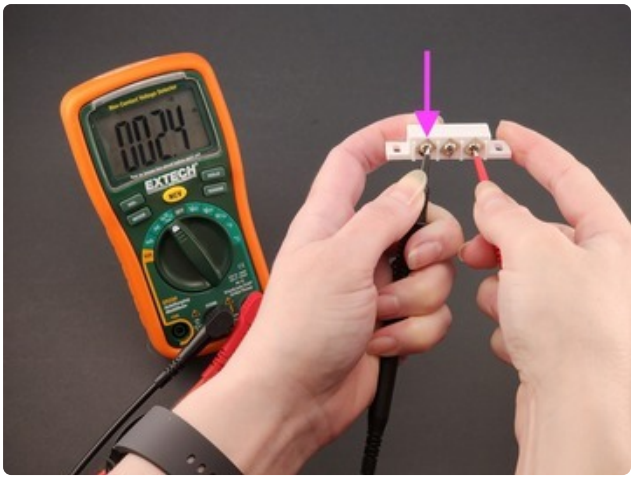
If you're interested in learning how to identify which terminal is **common** and which terminal is **NC**, feel free to continue through the rest of the steps.

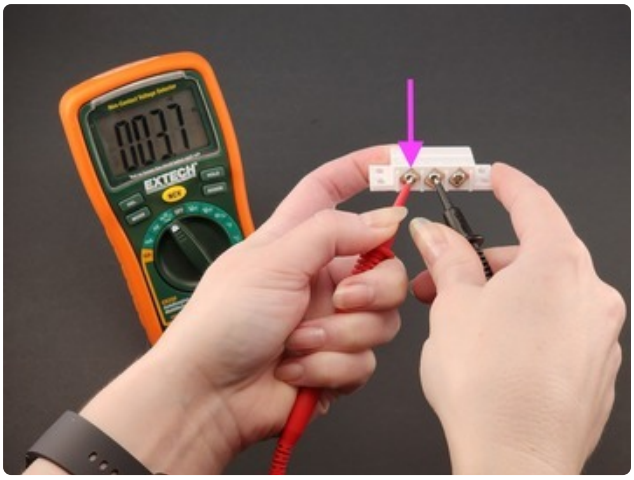
The next two steps **require the magnet to be present against the reed switch**.

Check each of the terminal pairs again for continuity.

When continuity is indicated, it means you have found the **common** and the **normally open (NO)** terminals. However, you still don't know which is which yet.

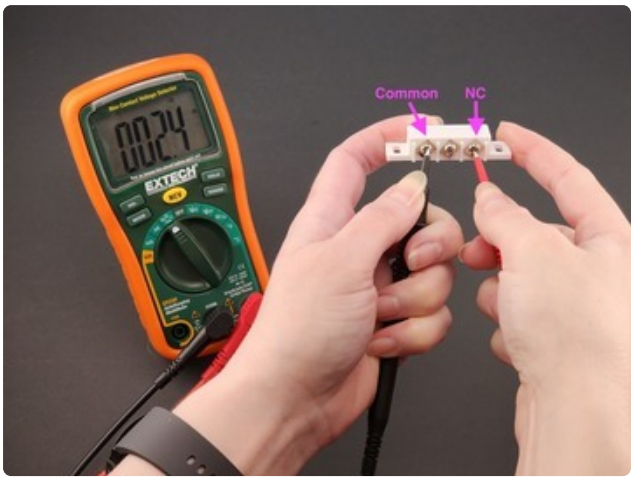
Now it's time to determine exactly which terminal is which.



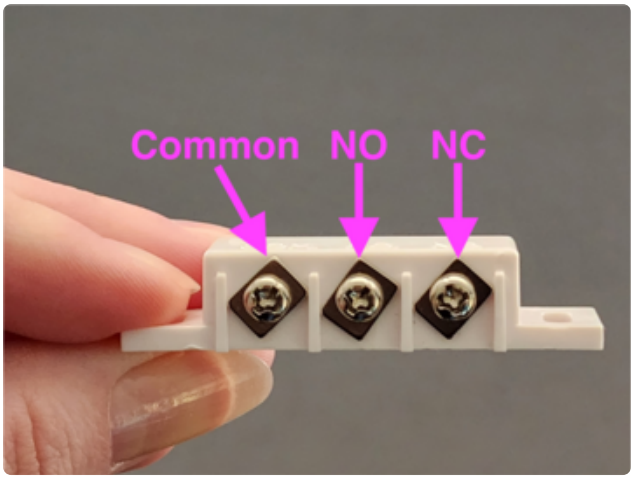


There will be one terminal that caused continuity in both cases, i.e. with and without the magnet present. This is your **common** terminal.

The first image shows continuity without the magnet. The second image shows continuity with the magnet present. The terminal that is common to both images is marked with an arrow. On this reed switch, this is the **common** terminal.



Now that you've identified your **common** terminal, separate the magnet from the reed switch. Hold one probe against the **common** terminal, and verify which of the other two terminals causes continuity. Continuity with the common terminal is indicated on the third terminal. That is your **NC** terminal.



You identified the common and NC terminals as the two outside terminals. Therefore, you now know the middle terminal is the **NO**. (You can verify this by reintroducing the magnet, and checking continuity on the **common** terminal and the **NO** terminal.)

Now you know the identification of all three terminals!

Project assembly will be easier if you label the two pins you intend to use.

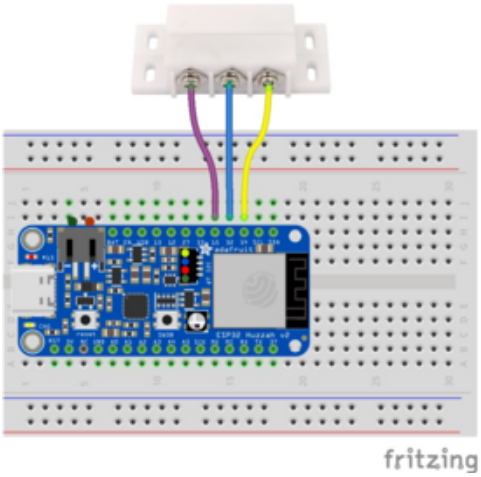
Using CircuitPython to Identify Your Reed Switch Terminals

Another option is to let CircuitPython tell you which terminal is which. This code verifies which terminals result in connectivity to which other terminals, and uses that information to identify which terminal is **normally closed (NC)**, which is **common**, and which is **normally open (NO)**.

Wiring

First, you'll need to wire the reed switch to your Feather so that all three terminals are connected.

The following diagram uses the Feather ESP32 V2, and the default pin assignments in the code match the diagram. You can substitute any CircuitPython-compatible microcontroller, and any pin combination, as long as the code is updated to match.



Attach **wires** to all **three terminals on your reed switch**.
Reading left to right, connect the following:

Reed switch terminal 1 to **Feather D15.**
Reed switch terminal 2 to **Feather D32.**
Reed switch terminal 3 to **Feather D14.**

CircuitPython Code

Download the following using the **Download Project Bundle** button, and load **code.py** onto to your microcontroller.

```
# SPDX-FileCopyrightText: 2022 Brian Rossman
# SPDX-FileCopyrightText: 2022 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT
"""
A CircuitPython program to identify reed switch terminals on a switch with three
terminals (normally closed, normally open, and common). This code is not designed
for two terminal reed switches.
"""
import time
import board
import digitalio
import supervisor

# Update these pins to match the pins to which you connected the reed switch.
TERMINAL_ONE = board.D14
TERMINAL_TWO = board.D32
TERMINAL_THREE = board.D15

# Create digital pin objects using the pins defined above.
pin_1 = digitalio.DigitalInOut(TERMINAL_ONE)
pin_2 = digitalio.DigitalInOut(TERMINAL_TWO)
pin_3 = digitalio.DigitalInOut(TERMINAL_THREE)

# Wait for the serial connection to be established.
while not supervisor.runtime.serial_connected:
    time.sleep(0.25)
time.sleep(1)

# Prompt and wait for confirmation the magnet is NOT near the reed switch.
print("Ensure no magnet is near the reed switch.")
print("Press ENTER to continue")
input()  # Waits for you to press enter to continue.

# Set Terminal 1 as the only output.
pin_1.switch_to_output()

# Set Terminal 2 & 3 as inputs to detect connectivity.
pin_2.switch_to_input(pull=digitalio.Pull.UP)
pin_3.switch_to_input(pull=digitalio.Pull.UP)
```

```
# Set the output pin to False.
pin_1.value = False

# Negate pin logic due to use of pull-up.
ab_common = not pin_2.value
ac_common = not pin_3.value

# Prompt and wait for confirmation the magnet IS near the reed switch.
print("Place the magnet against the reed switch.")
print("Press ENTER to continue")
input()  # Waits for you to press enter to continue.

# Negate pin logic due to use of pull-up.
b_when_closed = not pin_2.value
c_when_closed = not pin_3.value

# Print pin assignments for reference.
print(f"Terminal pin assignments:\nTerminal 1 = {TERMINAL_ONE}" +
      f"\nTerminal 2 = {TERMINAL_TWO}\nTerminal 3 = {TERMINAL_THREE}\n")

# Print which terminal is Normally Closed, Common, and Normally Open.
if ab_common and not ac_common and not b_when_closed and not c_when_closed:
    print("Normally Closed: Terminal 1, Common: Terminal 2, Normally Open: Terminal
3")
elif not ab_common and ac_common and not b_when_closed and not c_when_closed:
    print("Normally Closed: Terminal 1, Common: Terminal 3, Normally Open: Terminal
2")
elif ab_common and not ac_common and not b_when_closed and c_when_closed:
    print("Normally Closed: Terminal 2, Common: Terminal 1, Normally Open: Terminal
3")
elif not ab_common and not ac_common and not b_when_closed and c_when_closed:
    print("Normally Closed: Terminal 2, Common: Terminal 3, Normally Open: Terminal
1")
elif not ab_common and ac_common and b_when_closed and not c_when_closed:
    print("Normally Closed: Terminal 3, Common: Terminal 1, Normally Open: Terminal
2")
elif not ab_common and not ac_common and b_when_closed and not c_when_closed:
    print("Normally Closed: Terminal 3, Common: Terminal 2, Normally Open: Terminal
1")
else:
    # All options are covered above. If none are valid, there may be an issue with
your wiring.
    print("Something went wrong, check connections and try again.")
```

Now, run the code and connect to the serial console.

First, you will be prompted by the following. As it says, make sure there is no magnet near the reed switch, and press **enter** on your keyboard.

```
Ensure no magnet is near the reed switch.
Press ENTER to continue
```

Next, you'll be prompted by the following. Again, follow the instructions by holding a magnet against the reed switch, and while the magnet is held in place, press **enter** on your keyboard to continue.

```
Place the magnet against the reed switch.
Press ENTER to continue
```

Finally, you'll see there results. First, the serial console lists which terminals are assigned to which pins for reference, so you don't have to go back through the code to figure out how you wired it. Then, it prints out which terminal is **normally closed**, **common**, and **normally open**, in that order.

```
Terminal pin assignments:
Terminal 1 = board.D15
Terminal 2 = board.D32
Terminal 3 = board.D14

Normally Closed: Terminal 3, Common: Terminal 1, Normally Open: Terminal 2
```

Since you only need NC and common for this project, you'll want to label those two terminals to make project assembly smoother.

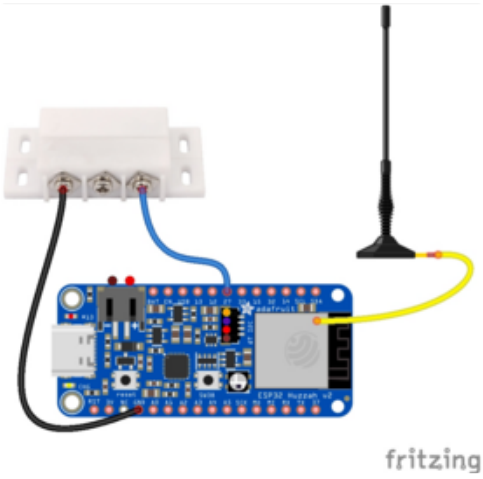
Between those two options, you should have successfully identified the terminals on your reed switch, regardless of what the product copy or terminal labels indicate. The next step is assembly!

Soldering and Assembly

This page covers how to solder up and assemble this project. It's only a few steps, as the build is quite simple.

If you need help with soldering, check out this guide (https://adafru.it/drl) for more details.

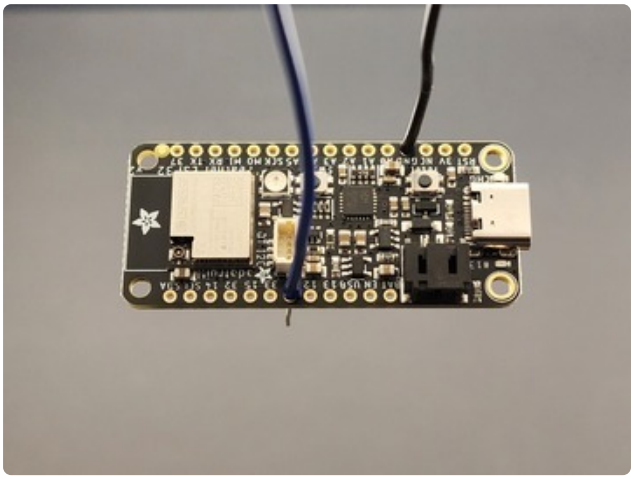
Wiring Diagram



Use this diagram for reference while soldering and assembling.

Soldering

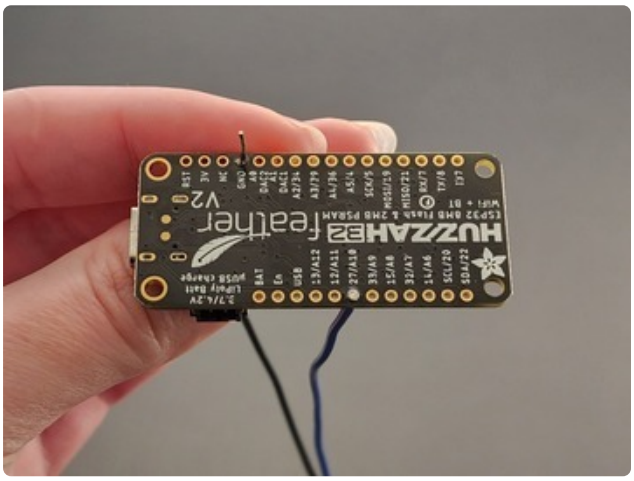
There is very little soldering needed!



Solder or otherwise solidly attach two wires to your Feather ESP32 V2. **The wires should be attached from the top through to the bottom.** This means soldering from the bottom of the Feather.

The **first wire attaches to pin D27**. The **second wire attaches to the GND pin**.

The colors of the wires don't matter, but using black on the ground pin can help avoid confusion, if that's something you're already used to.

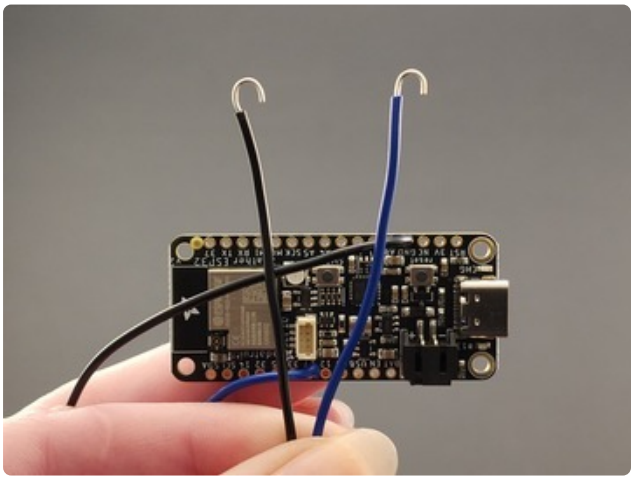


Trim any unnecessary excess wire from the bottom of the Feather. The image shows D27 trimmed, but not GND.

Assembly

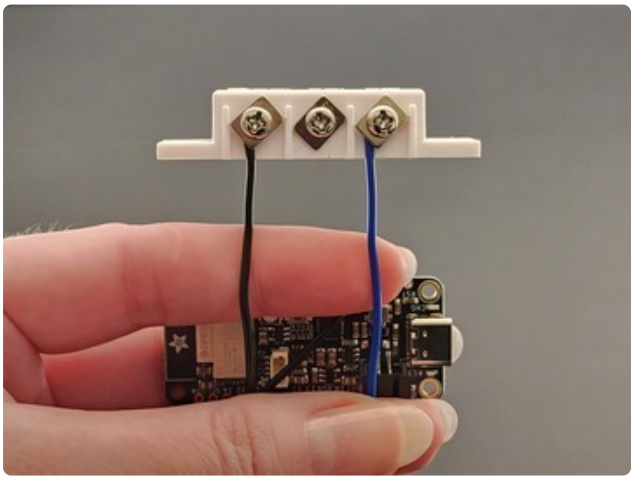
The next sections walk through the assembly process.

Reed Switch



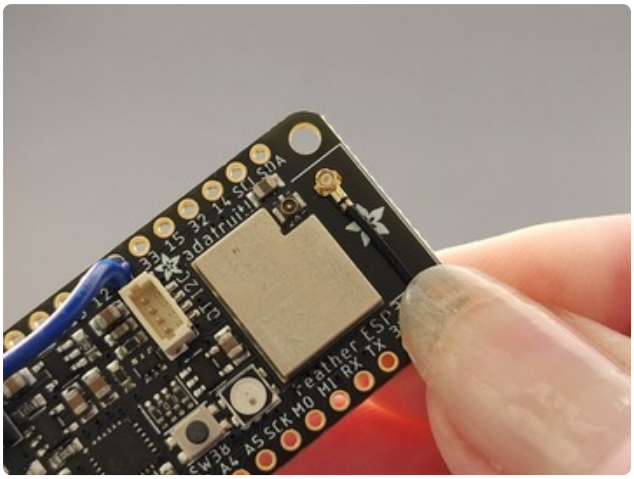
Strip the ends of the wire opposite the Feather end. **Prepare them to connect to your reed switch.**

In the case of the switch used to demonstrate this project, it means preparing them to attach to via screws. You would want to make hook shaped curls on the end of the wires to hold them steady on the screw as you tighten it.

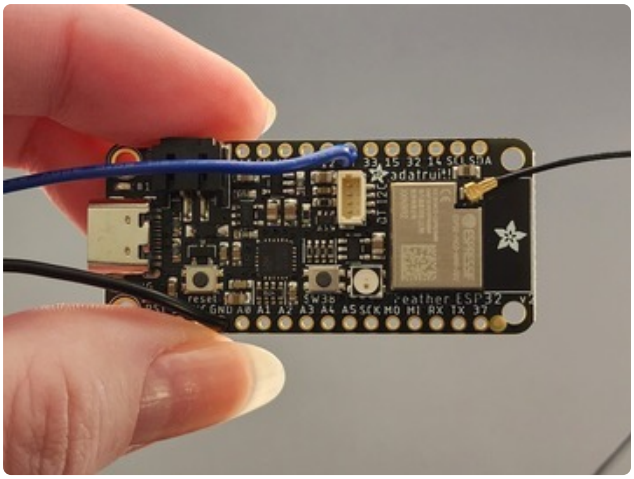


Connect the wire attached to D27 to the **normally closed** terminal on your reed switch.
Connect the wire attached to GND to the **common** terminal on your reed switch.

WiFi Antenna



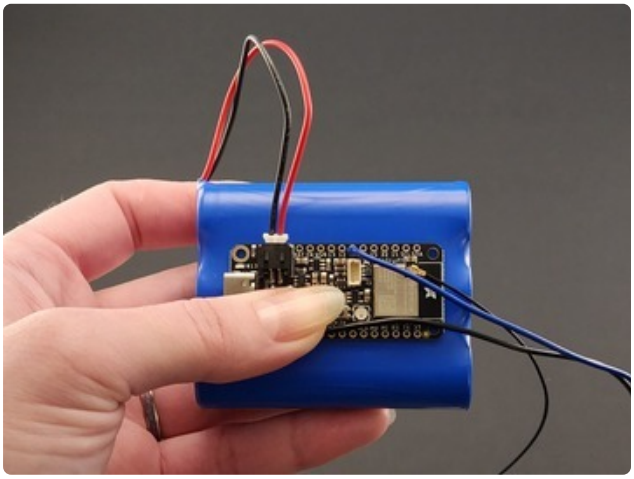
Grab your **WiFi antenna**, find the **w.FL connector end**. This is the end that matches the connector on your Feather.



Carefully press the antenna connector onto the Feather connector. Be patient - these connectors are tiny and it can sometimes take a few tries to get them fitted.

The w.FL connectors are super small! It might take a few tries to get it to connect. Be patient - this is normal!

Battery



Plug the **battery JST connector** into the **Feather JST connector**.

Code the WiFi Mailbox Notifier

Once you've finished setting up your Feather ESP32 V2 with CircuitPython, you're ready to load the code and libraries using the Project Bundle.

Click the blue **Download Project Bundle** button above the code shown below to download the Project Bundle zip.

```
# SPDX-FileCopyrightText: 2022 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""CircuitPython WiFi Mailbox Notifier"""

import time
import ssl
```

```
import alarm
import board
import digitalio
import analogio
import wifi
import socketpool
import supervisor
import microcontroller
import adafruit_requests
from adafruit_io.adafruit_io import IO_HTTP

# Get WiFi/Adafruit IO details from secrets.py
try:
    from secrets import secrets
except ImportError:
    print("Please create secrets.py and add your WiFi and AIO credentials there!")
    raise

# Update to True if you want metadata sent to Adafruit IO. Defaults to False.
METADATA = False

# If the reason the board started up is due to a supervisor.reload()...
if supervisor.runtime.run_reason is supervisor.RunReason.SUPERVISOR_RELOAD:
    alarm.sleep_memory[3] += 1  # Increment reload number by 1.
    print(f"Reload number: {alarm.sleep_memory[3]}")  # Print current supervisor
reload number.
    if alarm.sleep_memory[3] > 5:  # If supervisor reload number exceeds 5...
        # Print the following...
        print("Reload is not resolving the issue. \nBoard will hard reset in 20
seconds. ")
        time.sleep(20)  # ...wait 20 seconds...
        microcontroller.reset()  # ...and hard reset the board. This will clear
alarm.sleep_memory.

# Initialise metadata.
if alarm.wake_alarm:
    print("Awake! Alarm type:", alarm.wake_alarm)
    # Increment wake count by 1.
    alarm.sleep_memory[0] += 1
else:
    print("Wakeup not caused by alarm.")
    # Set wake count to 0.
    alarm.sleep_memory[0] = 0
    # Set error count to 0.
    alarm.sleep_memory[2] = 0

# Print wake count to serial console.
print("Alarm wake count:", alarm.sleep_memory[0])

# No data has been sent yet, so the send-count is 0.
alarm.sleep_memory[1] = 0

# Set up battery monitoring.
voltage_pin = analogio.AnalogIn(board.VOLTAGE_MONITOR)
# Take the raw voltage pin value, and convert it to voltage.
voltage = (voltage_pin.value / 65536) * 2 * 3.3

# Set up red LED.
led = digitalio.DigitalInOut(board.LED)
led.switch_to_output()

# Set up the alarm pin.
switch_pin = digitalio.DigitalInOut(board.D27)
switch_pin.pull = digitalio.Pull.UP


# Send the data. Requires a feed name and a value to send.
def send_io_data(feed_name, value):
    """
```

```
    Send data to Adafruit IO.
    Provide an Adafruit IO feed name, and the value you wish to send.
    """
    feed = io.create_and_get_feed(feed_name)
    return io.send_data(feed["key"], value)


# Connect to WiFi
try:
    wifi.radio.connect(secrets["ssid"], secrets["password"])
    print("Connected to {}!".format(secrets["ssid"]))
    print("IP:", wifi.radio.ipv4_address)

    pool = socketpool.SocketPool(wifi.radio)
    requests = adafruit_requests.Session(pool, ssl.create_default_context())
# WiFi connectivity fails with error messages, not specific errors, so this except
is broad.
except Exception as error:  # pylint: disable=broad-except
    print("Failed to connect to WiFi. Error:", error, "\nBoard will reload in 15
seconds.")
    alarm.sleep_memory[2] += 1  # Increment error count by one.
    time.sleep(15)
    supervisor.reload()

# Pull your Adafruit IO username and key from secrets.py
aio_username = secrets["aio_username"]
aio_key = secrets["aio_key"]
# Initialize an Adafruit IO HTTP API object
io = IO_HTTP(aio_username, aio_key, requests)

# Print battery voltage to the serial console and send it to Adafruit IO.
print(f"Current battery voltage: {voltage:.2f}V")
# Adafruit IO can run into issues if the network fails!
# This try/except ensures your code will continue to run.
try:
    led.value = True  # Turn on the LED to indicate data is being sent.
    send_io_data("battery-voltage", f"{voltage:.2f}V")
    led.value = False  # Turn off the LED to indicate data sending is complete.
# Adafruit IO can fail with multiple errors depending on the situation, so this
except is broad.
except Exception as error:  # pylint: disable=broad-except
    print("Failed to send to Adafruit IO. Error:", error, "\nBoard will reload in
15 seconds.")
    alarm.sleep_memory[2] += 1  # Increment error count by one.
    time.sleep(15)
    supervisor.reload()

# While the door is open...
while not switch_pin.value:
    # Adafruit IO sending can run into various issues which cause errors.
    # This try/except ensures the code will continue to run.
    try:
        led.value = True  # Turn on the LED to indicate data is being sent.
        # Send data to Adafruit IO
        print("Sending new mail alert to Adafruit IO.")
        send_io_data("new-mail", "New mail!")
        print("Data sent!")
        # If METADATA = True at the beginning of the code, send more data.
        if METADATA:
            print("Sending metadata to Adafruit IO.")
            # The number of times the board has awakened by an alarm since the last
reset.
            send_io_data("wake-count", alarm.sleep_memory[0])
            # The number of times the mailbox data has been sent.
            send_io_data("send-count", alarm.sleep_memory[1])
            # The number of WiFi or Adafruit IO errors that have occurred.
            send_io_data("error-count", alarm.sleep_memory[2])
            print("Metadata sent!")
        time.sleep(30)  # Delay included to avoid data limit throttling on Adafruit
```

```
IO.
        alarm.sleep_memory[1] += 1  # Increment data send count by 1.
        led.value = False  # Turn off the LED to indicate data sending is complete.

    # Adafruit IO can fail with multiple errors depending on the situation, so this
except is broad.
    except Exception as error:  # pylint: disable=broad-except
        print("Failed to send to Adafruit IO. Error:", error, "\nBoard will reload
in 15 seconds.")
        alarm.sleep_memory[2] += 1  # Increment error count by one.
        time.sleep(15)
        supervisor.reload()

# Deinitialise the alarm pin.
switch_pin.deinit()

# Turn off the NeoPixel/I2C power for deep sleep.
power_pin = digitalio.DigitalInOut(board.NEOPIXEL_I2C_POWER)
power_pin.switch_to_output(False)

# Turn off LED for deep sleep.
led.value = False

# Create a timer alarm to be triggered every 12 hours (43200 seconds).
time_alarm = alarm.time.TimeAlarm(monotonic_time=(time.monotonic() + 43200))

# Create a pin alarm on pin D27.
pin_alarm = alarm.pin.PinAlarm(pin=board.D27, value=False, pull=True)

print("Entering deep sleep.")

# Exit and set the alarm.
alarm.exit_and_deep_sleep_until_alarms(pin_alarm, time_alarm)
```

Upload the Code and Libraries to the Feather ESP32 V2

Unzip the folder and upload the following items to your Feather.

- **lib/** folder
- **code.py**

Create and/or Update secrets.py

CircuitPython uses information in a **secrets.py** file for WiFi and Adafruit IO credentials. You must create a **secrets.py** file that contains code resembling the following.

Once updated, save the file to your ESP32 Feather V2.

```
secrets = {
    "ssid": "wifi_ssid",
    "password": "wifi_password",
    "aio_username": "adafruit_io_username",
```

```
    "aio_key": "adafruit_io_key",
}
```

Do not share your secrets.py file once it has been updated with your credentials!

Code Walkthrough

Here's a walkthrough of the **code.py** file used with this project.

Choose Whether to Send Metadata to Adafruit IO

After the imports, you'll find a `METADATA` variable set to `False`. Update this to `True` if you would also like to send metadata to Adafruit IO. The metadata is generated by the code, and is regarding the Feather and the data being sent.

```
METADATA = False
```

When set to `True`, the code will also include the following feeds in your Adafruit IO account.

- **wake-count**: The number of times the board has been awakened by an alarm since the last reset.
- **send-count**: The number of times the mailbox data has been sent since the last time the board started up. The mailbox data is sent every 30 seconds as long as the mailbox door is open. This feed tracks that number.
- **error-count**: The number of WiFi or Adafruit IO errors that have occurred since the last reset.

Reset the Feather based on the Number of Error Reloads

The next section is verifying why the Feather started up. If it's due to anything other than a standard CircuitPython reset-type start up, e.g. caused by a command found later in the code (`supervisor.reload()`), then it runs the rest of the code in this block.

```
if supervisor.runtime.run_reason is supervisor.RunReason.SUPERVISOR_RELOAD:
    alarm.sleep_memory[3] += 1
    print(f"Reload number: {alarm.sleep_memory[3]}")
    if alarm.sleep_memory[3] &gt; 5:
        print("Reload is not resolving the issue. \nBoard will hard reset in 20
seconds. ")
```

```
        time.sleep(20)
        microcontroller.reset()
```

If the Feather restarted due to a reload, begin tracking the number of reloads. First, increment the reload number tracking by 1, then print the current number of reloads. If the number of reloads exceeds 5, then reloading is obviously not resolving the issue, and so, instead, hard reset the board.

Reload vs. Reset

You may have noticed that the actions "reload" and "reset" are both mentioned above. It's important to understand the difference.

A reload is what happens when you hit CTRL+D from the REPL, as well as when the board reloads and the serial console shows a "soft reboot" before beginning to run your code again. It is also what happens when you call `supervisor.reload()` in your code, as shown above. **Data stored in `alarm.sleep_memory` will persist through a reload.**

A reset is what happens when you press the reset button on your board. It can also happen automatically for a variety of reasons, including a low voltage brown out or a watchdog timer. It is what happens when you call `microcontroller.reset()` in your code, as shown above. **Data stored in `alarm.sleep_memory` will not persist through a reset.**

Initialise the Metadata and Various Pins

Here is where the metadata you opted in or out of above is initialised.

If the Feather start up was being awakened by an alarm (the mailbox door opening or the twelve hour battery voltage check-in), it will print to the serial console the type of alarm that caused it. It then increments the wake count by 1.

If the Feather startup was caused by something other than an alarm, it will print to the serial console, "Wakeup not caused by alarm.", and set the wake count and the error count to 0.

```
if alarm.wake_alarm:
    print("Awake! Alarm type:", alarm.wake_alarm)
    alarm.sleep_memory[0] += 1
else:
    print("Wakeup not caused by alarm.")
    alarm.sleep_memory[0] = 0
    alarm.sleep_memory[2] = 0
```

Then, the code prints the wake count to the serial console. This is the number of times the Feather has awaken due to an alarm since the last time the Feather was reset.

```
print("Alarm wake count:", alarm.sleep_memory[0])
```

Following that, as no data has been sent yet, the send count is set to 0.

```
alarm.sleep_memory[1] = 0
```

The next section of code sets up the various pins needed for this project.

First, you set up the voltage pin. The `VOLTAGE_MONITOR` pin returns a raw value.

To obtain a useful voltage value from that raw value, you must apply some math, which happens on the next line.

The raw pin value varies between 0-65535, a full-scale value for an analog input. Dividing the raw pin value by 65536 converts the raw value to a value between 0.0 and 1.0. Note that CircuitPython begins counting at 0, so while the values are between 0 and 65535, there are 65536 values, which is why the pin value is divided by 65536.

Next, it is multiplied by 2 to compensate for the hardware voltage divider on the Feather, which divides the value in half.

Finally, it is multiplied by 3.3 which is the reference voltage value.

This math results in a useful voltage value that you can use and track to determine when you need to charge or swap your battery!

```
voltage_pin = analogio.AnalogIn(board.VOLTAGE_MONITOR)
voltage = (voltage_pin.value / 65536) * 2 * 3.3
```

The little red LED is used to indicate that data is being sent to Adafruit IO. This code creates the LED object and sets it to an output.

```
led = digitalio.DigitalInOut(board.LED)
led.switch_to_output()
```

A majority project is based around using the reed switch to wake up the Feather and send data to Adafruit IO. The reed switch is connected physically to pin D27, but to use it in code, you are required tell CircuitPython where to look for it and how to manage it. This code creates the switch_pin on pin D27, and sets it to a pullup.

```
switch_pin = digitalio.DigitalInOut(board.D27)
switch_pin.pull = digitalio.Pull.UP
```

Adafruit IO Data Sending Helper Function

The next section is a helper function designed to send data to Adafruit IO.

```
def send_io_data(feed_name, value):
    feed = io.create_and_get_feed(feed_name)
    return io.send_data(feed["key"], value)
```

You might be wondering why a helper function is needed here. Basically, there are a number of places in the code where you send data to Adafruit IO. This function takes what would be longer and noisier code, and simplifies it. It keeps your code cleaner, and more importantly, easier to read.

Without this function, every time you wanted to send data to an Adafruit IO feed, you would include something like the following line. This example sends the string "New mail!" to the new-mail feed.

```
io.send_data(io.create_and_get_feed("new-mail")["key"], "New mail!")
```

The helper function simplifies the sending data code to the following.

```
send_io_data("new-mail", "New mail!")
```

Much simpler, right?

In general, you may find you're using a line or block of code repeatedly in your code. There are times when it makes sense to factor the repeated code into a helper function, and use that instead. It's not always the best option, but in many cases it can be super helpful.

Connect to WiFi

This section connects your Feather to WiFi. As WiFi is fraught with peril, the code is contained within a `try` and `except` block. This ensures your code will continue running if an error occurs.

The first part is inside the `try`. It pulls your WiFi credentials out of your **secrets.py** file, and uses them to connect to your WiFi network. It also prints that you are

connected to your SSID, and the IP address assigned to the Feather. The next two lines create the socketpool and begins the requests session.

```
try:
    wifi.radio.connect(secrets["ssid"], secrets["password"])
    print("Connected to {}!".format(secrets["ssid"]))
    print("IP:", wifi.radio.ipv4_address)

    pool = socketpool.SocketPool(wifi.radio)
    requests = adafruit_requests.Session(pool, ssl.create_default_context())
```

The next part handles any potential errors. Without the try and except included, any error thrown by the code would halt, and your program would stop running. There is no way to notify you of this, other than the serial console, so unless it's plugged into your computer and you're viewing the console, you might now know your code is no longer running.

The except block begins with `except Exception as error:`. You could simply begin with `except:` and the code would run, however it's good Python practice to specify the error you're attempting to except. In this case, WiFi can fail for such a wide variety of reasons, it's not reasonable to include them all. So, this code excepts `Exception`, which while considered not quite up to good Python practice, allows the code to catch every possible error.

When an error occurs, it prints to the serial console that there's been an error, what the exact error text is, and that the board will reload in 15 seconds. It increments the error count by 1. Then there is a 15 second delay before reloading the Feather, which will restart the code.

```
except Exception as error:
    print("Failed to connect to WiFi. Error:", error, "\nBoard will reload in 15
seconds.")
    alarm.sleep_memory[2] += 1
    time.sleep(15)
    supervisor.reload()
```

Adafruit IO Credentials and HTTP Initialization

This section gets your connection to Adafruit IO set up so you can send data.

Your Adafruit IO username and key are pulled from **secrets.py**.

Your Adafruit IO credentials are then used to initialize the Adafruit IO HTTP API object.

```
aio_username = secrets["aio_username"]
aio_key = secrets["aio_key"]

io = IO_HTTP(aio_username, aio_key, requests)
```

Battery Voltage Data

You created the voltage pin earlier in your code, and applied the math necessary to obtain a useful voltage value. Now it's time to do something with that data!

First the code prints the data to the serial console.

```
print(f"Current battery voltage: {voltage:.2f}V")
```

Next you'll find another `try` and `except` block, for the same basic reason as described above in the WiFi section.

First, the red LED is powered on to indicate data is being sent.

The code then tries to send the battery voltage data to Adafruit IO. This is the first time you get to use the helper function. You provide the helper function with the feed name, battery-voltage, and the content to send to the feed, the voltage value.

The content sent is presented in what's called an f-string, which is a way to format strings in CircuitPython. In this example, you want to limit the number of decimal places to two, and include a V after the value so it's clear that it is a voltage value. The entire string begins with `f`, followed by code in quotes. Inside the quotes, the formatted value is inside brackets, and the rest of the desired text is included after the closing bracket.

Finally, the LED is turned off to indicate data sending is complete.

```
try:
    led.value = True
    send_io_data("battery-voltage", f"{voltage:.2f}V")
    led.value = False
```

This section should look very familiar; it is nearly identical to the WiFi `except` block above. The only difference is the text printed to the serial console. This time it refers to an Adafruit IO failure instead of WiFi. The rest of the content is the same and functions in the same way.

```
except Exception as error:
    print("Failed to send to Adafruit IO. Error:", error, "\nBoard will reload in
15 seconds.")
```

```
    alarm.sleep_memory[2] += 1
    time.sleep(15)
    supervisor.reload()
```

The Mailbox Door is Opened

This next section is only run when the switch is separated from its magnet, which is to say, the mailbox door is opened. In CircuitPython, when the switch used in this project is separated from the magnet, it returns `False`. Therefore, this block runs as long as the `switch_pin.value` is `False`. It will continue to repeat until the mailbox door is closed, and the magnet is reintroduced to the switch.

```
while not switch_pin.value:
```

As you're going to be sending data to Adafruit IO, the following sections of code are contained within a `try`.

First, turn on the red LED to indicate data is about to be sent.

The code prints to the serial console that it is sending the new mail alert to Adafruit IO. It then sends the phrase New mail! to the new-mail IO feed. Finally, it prints to the serial console that the data has been sent.

```
[...]
        led.value = True
        print("Sending new mail alert to Adafruit IO.")
        send_io_data("new-mail", "New mail!")
        print("Data sent!")
```

The next section of the code only runs if you enabled metadata at the beginning of the program by setting `METADATA = True`. If metadata is enabled, the code prints to the serial console that it is sending metadata to Adafruit IO. It then sends the wake count, send count, and error count to their respective feeds. Finally, it prints to the console that metadata has been sent.

```
[...]
        if METADATA:
            print("Sending metadata to Adafruit IO.")
            send_io_data("wake-count", alarm.sleep_memory[0])
            send_io_data("send-count", alarm.sleep_memory[1])
            send_io_data("error-count", alarm.sleep_memory[2])
            print("Metadata sent!")
```

This section ends with the following. First, there is a 30 second delay. This creates a delay in how often the data can be sent to Adafruit IO, which is included to avoid the data limit throttling built into Adafruit IO.

As data has been sent, the code increments the data send count by 1.

And finally, since the data is done being sent for now, turn off the LED.

```
[...]
        time.sleep(30)
        alarm.sleep_memory[1] += 1
        led.value = False
```

This section should look completely familiar; it is identical to the previous Adafruit IO `except` block. It looks the same and functions in the same way.

```
[...]
    except Exception as error:
        print("Failed to send to Adafruit IO. Error:", error, "\nBoard will reload
in 15 seconds.")
        alarm.sleep_memory[2] += 1
        time.sleep(15)
        supervisor.reload()
```

Prepare for Deep Sleep and Set Up Alarms

This is the last section of the code. In it, you prepare for deep sleep, set up the wake alarms, and finally, enter deep sleep until one of the alarms awakens the Feather.

First, the code deinitialises the switch pin. This is so you can use it to create the pin alarm in a bit.

```
switch_pin.deinit()
```

Next, the code turns off the NeoPixel and I2C power pin, and the red LED, to conserve power draw while in deep sleep. Even if the NeoPixel is off and no I2C devices are plugged in, leaving the power pin on results in an increased power draw. The red LED should already be off, but on the off chance it's not, this is good to include.

```
power_pin = digitalio.DigitalInOut(board.NEOPIXEL_I2C_POWER)
power_pin.switch_to_output(False)

led.value = False
```

Then, you create the two alarms that can wake up the Feather from deep sleep.

First, you create a time alarm. A time alarm wakes the microcontroller at a specified time in the future. `time.monotonic()` is the number of seconds since the microcontroller was last reset. It is always increasing and will be different at any point in time in your code. You choose the length of time you would like to pass before the

alarm is triggered, convert it to seconds, and add it to `time.monotonic()`. In this case, you want to send the battery voltage every 12 hours, both to keep track of it and have a heartbeat from the Feather to keep an eye on. To convert 12 hours to seconds, you multiply 12 * 60 minutes per hour * 60 seconds per minute which is 43200. Therefore, you add 43200 seconds.

```
time_alarm = alarm.time.TimeAlarm(monotonic_time=(time.monotonic() + 43200))
```

Next you create the pin alarm. A pin alarm wakes up the microcontroller when a specified pin changes state. While the mailbox door is closed, the switch is open. When the mailbox door opens, the switch closes. This change in state triggers the pin alarm. As the switch is connected to pin D27, you'll use that pin.

```
pin_alarm = alarm.pin.PinAlarm(pin=board.D27, value=False, pull=True)
```

The code then prints to the serial console that the Feather is entering deep sleep.

```
print("Entering deep sleep.")
```

Finally, the code exits and deep sleeps until the either of the specified alarms are triggered, in this case, the pin alarm or the time alarm created above.

```
alarm.exit_and_deep_sleep_until_alarms(pin_alarm, time_alarm)
```

Adafruit IO Reactive Action

Adafruit IO and CircuitPython make it easy to send data to Adafruit IO feeds (https://adafru.it/ioA). They automatically show up in your Feeds tab, where you can check on the status at anytime, which is excellent! However, you probably want to do something with your data. Adafruit IO has a number of options for that as well!

One way to use your data is through **Adafruit IO Actions**. Actions allow you to do a variety of things when a situation specified by you occurs. There are three types of Actions. You'll be using the **Reactive** type Action for two of the alerts needed for this project.

Reactive Actions are more advanced than the other Action types. They can integrate basic logic. At a basic level, they check if a feed value is somehow comparable to a value or to even another feed. If so, you'll be able to send an email notification, post a webhook or even publish a message to another feed. An example used in this project is that you can set up a reactive trigger to notify you when the battery voltage falls below a certain value. Super useful!

You'll create two Reactive Actions: one for new mail, the other for reporting low battery.

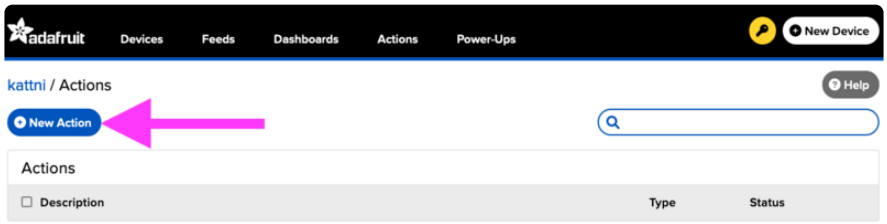
This page walks you through each step necessary for setting up the new mail Reactive Action.

In the last section, you'll find only the final few steps for setting up the low battery Reactive Action, as you can follow the same steps using different options that you followed while setting up the new mail Action.

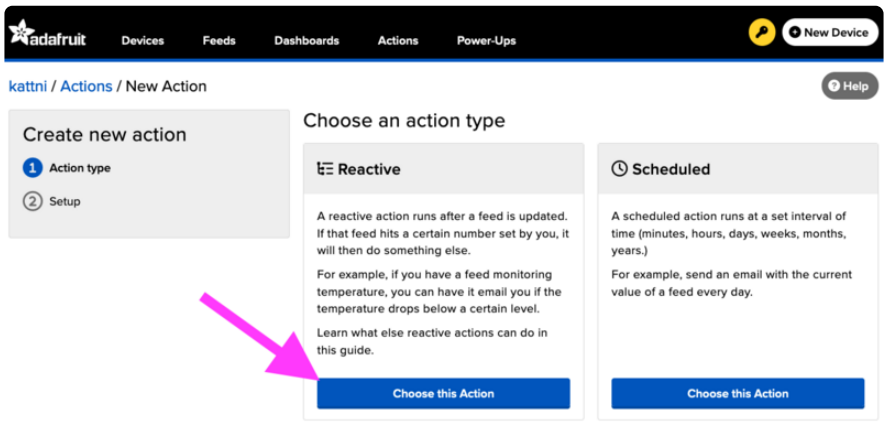
New Mail Action Set Up

In Adafruit IO, click **Actions** in the black menu bar.

On the Actions page, click **New Action**.



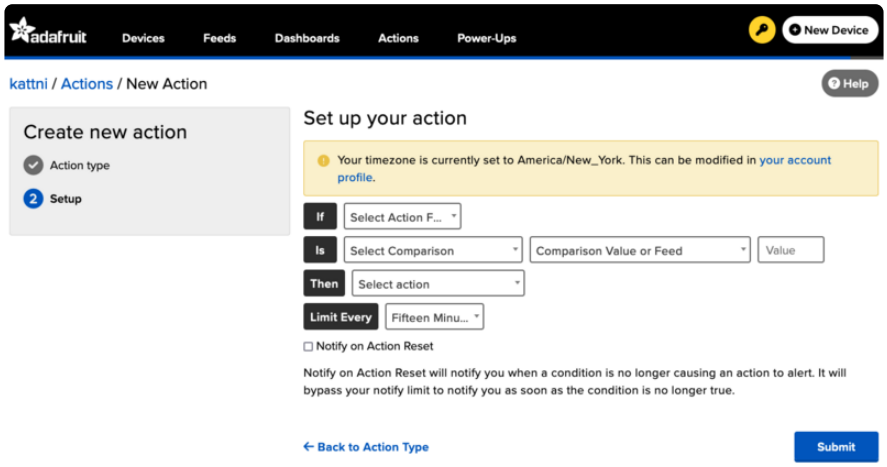
Choose the Reactive action type by clicking **Choose this Action** at the bottom of the Reactive box.



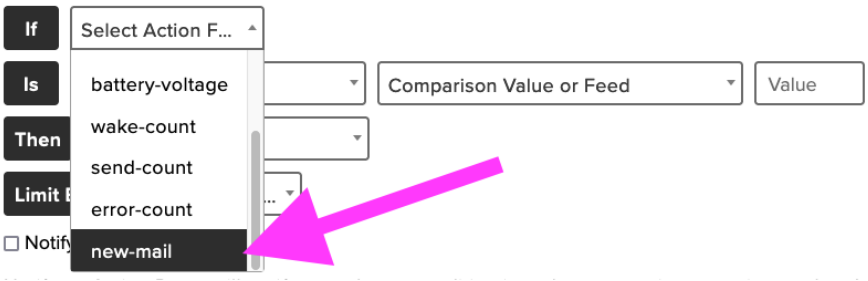
Once you've chosen the Reactive Action type, you'll see the following page. This is the set up page for the Reactive Action, which provides you with a series of dropdowns to choose from to create the Action you want.

This might seem a little confusing. Basically, the top three lines read as a sentence: "If the chosen feed, is compared in a specific way to another feed or a specific feed value, then complete the chosen action." You'll choose the options to update the sentence to read, "If the new-mail feed is equal to New mail! then email me."

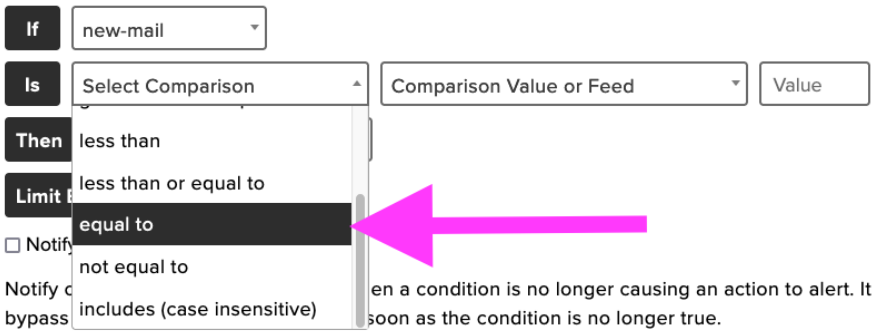
The rest of this page shows you how to choose the right options to have Adafruit IO send you an email when the new-mail feed indicates your mailbox door has been open.



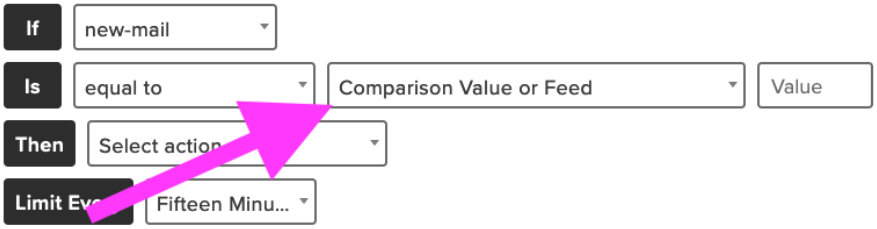
In the **If** section, for **Select Action Feed**, choose **new-mail**. This is the main feed that tells you when your mailbox door has been opened.



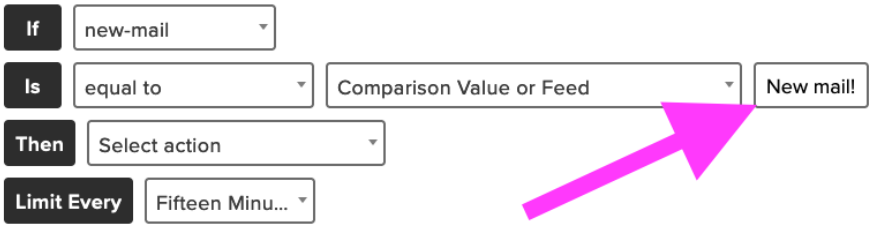
The **Is** section contains multiple things to update. For **Select Comparison**, choose **equal to**. We'll be comparing it to the value sent to Adafruit IO by the Feather, so we want it to match.



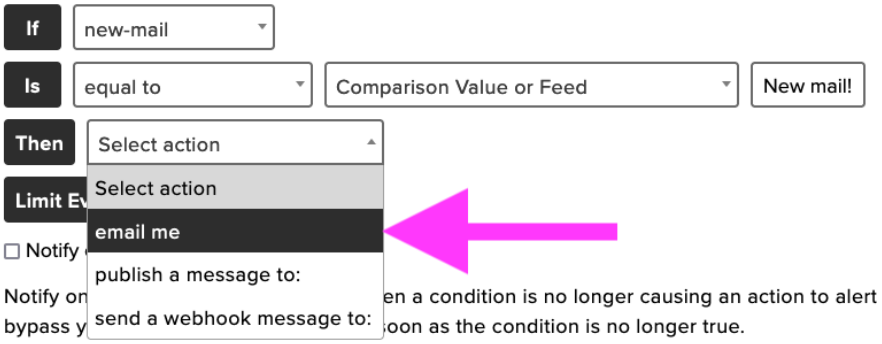
For **Comparison Value or Feed**, leave it as the default. You are not comparing it to another feed, you'll be comparing it to a value.



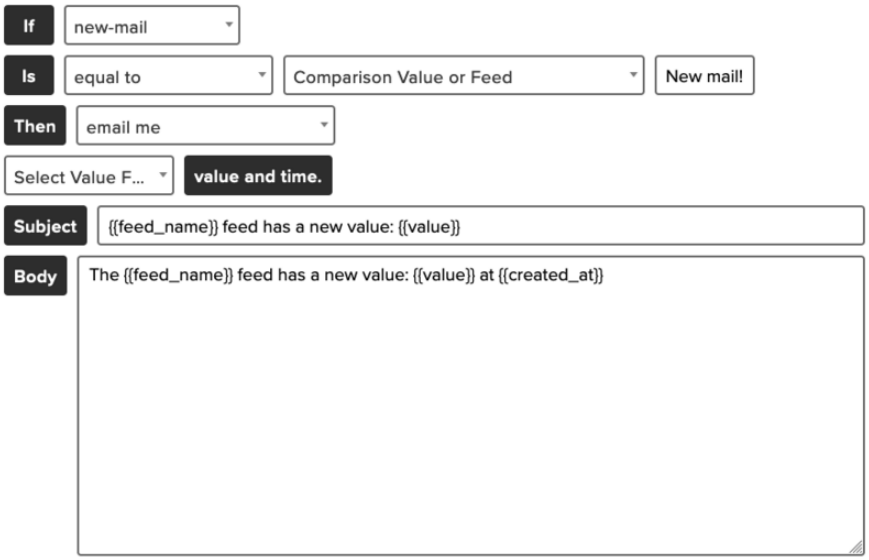
For the **Value** text box, type in, **New mail!**. This is the value sent to the new-mail feed when the mailbox door is opened.



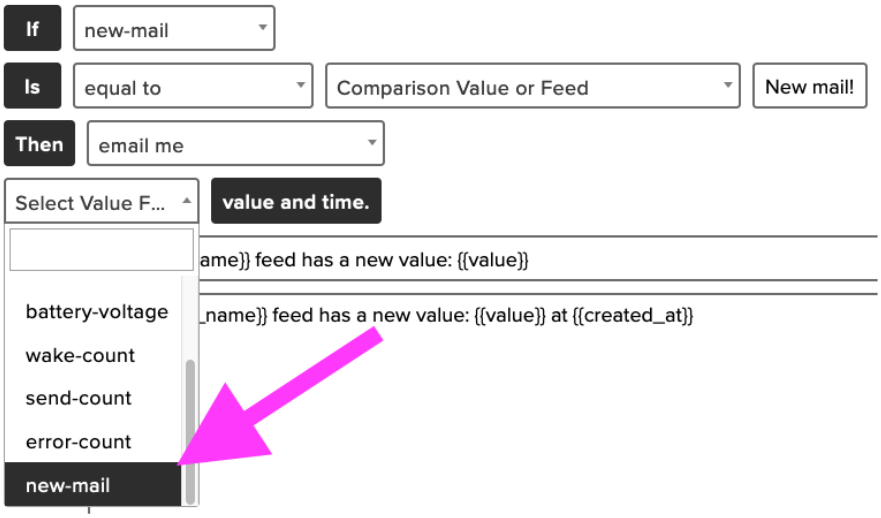
In the **Then** menu, choose **email me**.



Once you choose **email me**, you'll see the options expand to include more options, as shown here.

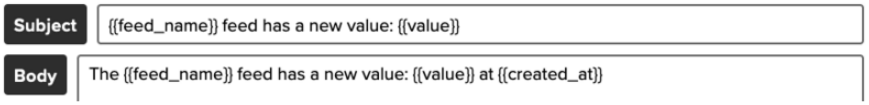


For **Select Value Feed** (the menu shown to the left of **value and time.**), choose **new-mail**.



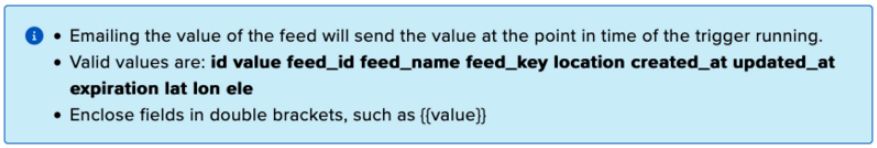
There is default content provided for the email generated by the Action.

- The subject of the email generated by the default would read, "new-mail feed has a new value: New mail!
- The body would read, "The new-mail feed has a new value: New mail! at (current date and time)"

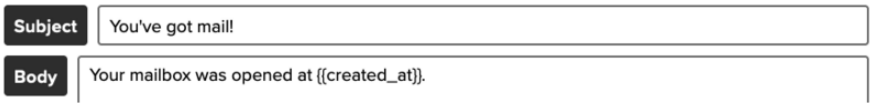


You'll notice some text surrounded by double brackets, e.g. {{ text }}. These are values that are provided by Adafruit IO with the value at the point of the Action being

triggered. There are multiple options for these values, including **feed_id**, **value**, **feed_name**, **created_at** and **updated_at**. See the blue box below for the full list.



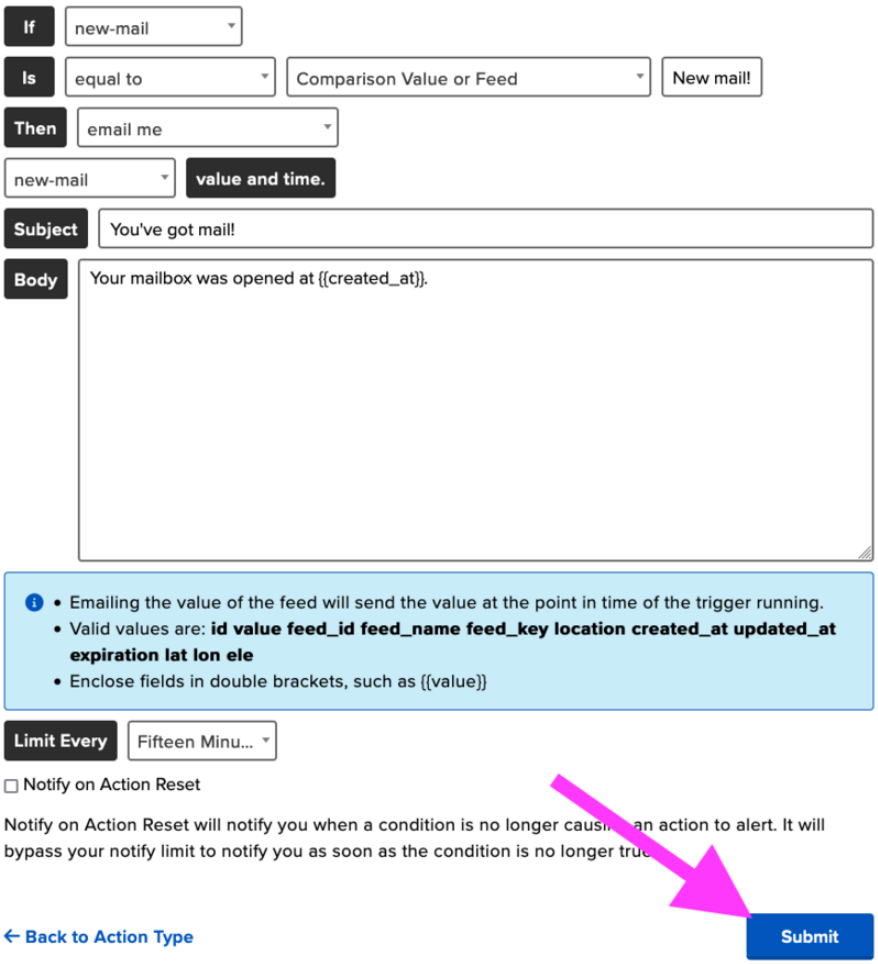
The default email content is a bit redundant with the feed name and the resulting value are essentially the same thing. Therefore, it's worth updating the content to something more applicable. The following image shows the email content updated. Feel free to customise this to fit your needs.



Leave the **Limit Every** at the default of **Fifteen Minutes**. The **Limit Every** value limits the amount of time needed between Actions triggering, before the Action can trigger again. The code sends a message every 30 seconds while the mailbox door is open. Someone might need a little time to place mail and packages in the box, so the door might be open for a few minutes. Rather than send you six emails, you can set **Limit Every** value to a higher amount of time, e.g. 15 minutes. This means you'll most likely only get one email each time the door is opened. In the event that the door is left open for an extended period of time, you will receive an email every 15 minutes until you close it. Feel free to increase or decrease this value as desired.

Leave **Notify on Action Reset** at the default of unchecked. Checking the box means Adafruit IO would notify you when the conditions that cause the email to be sent are no longer true. In this case, the feed value never changes, the same value is sent when the mailbox door is opened. Therefore, there is never a situation where the feed value would change, which is what would trigger the Reset Notification.

You're almost done! First verify the previous settings all match the options shown below. Once you're certain, click **Submit**.



Once submitted, you'll see your new Action listed in the Actions list! The listing includes a short description of the Action, the Action Type (Reactive), and the Status (Active).

The status column is important to be aware of. The Status may change from Active to Stopped. This can happen if you stop it (accidentally or otherwise), or if there's an issue on the Adafruit IO end of things. If the status is Stopped, the Action will not run and you will not receive notifications. If you find you haven't been notified in a while, and you see more recent data in your feed, be sure to check the Action Status as your first troubleshooting step.

Battery Voltage Action Set Up

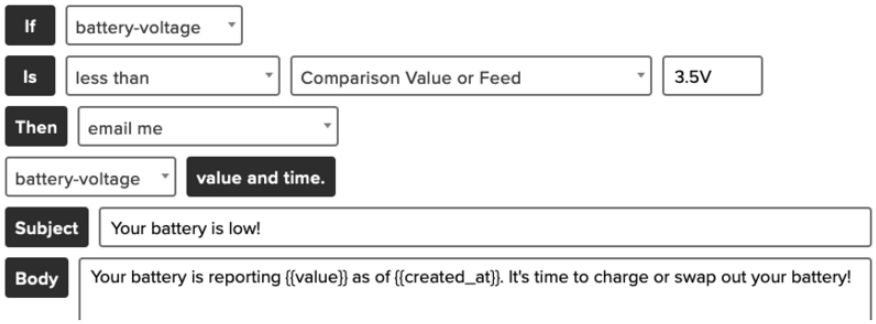
This project is ultimately battery-powered. As there is no charging included in the build, the battery will eventually drain and require either swapping or charging.

Therefore, you'll want to set up a notification when the battery is low. This section shows you what settings to chose to receive an email when your battery needs to be swapped or charged.

As the steps are basically outlined above, this section will not go into as much detail.

Update the conditions to match the following.

- Choose **battery-voltage** for the Action Feed.
- This time, you're tracking whether the feed value has dropped below a specific threshold, so you'll choose **less than** for Select Comparison.
- The voltage below which your battery is no longer a viable power source is 3.5V, so you'll update the Value to **3.5V**.
- Choose **email me**, as before.
- Choose **battery-voltage** as the Select Value Feed.
- Finally, update the email content to match what's shown below, or to whatever fits your needs.



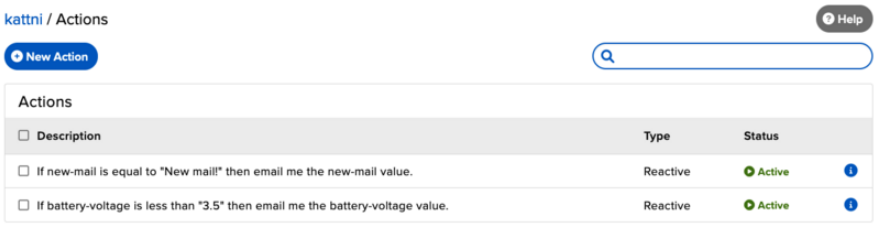
The major difference between the new mail set up and the battery voltage set up is found in the final two steps. Above you are instructed to leave the default options.

The battery voltage is sent every twelve hours, and when the mailbox door opens (though, it is not continually sent while the door is open). If the limit on emails was kept short, you would be receiving emails every time the battery voltage is reported to Adafruit IO (up to three times per day). As that's entirely unnecessary, you'll change **Limit Every** to **One Day**. This means you'll only receive an email once per day, regardless of how often the data is sent to Adafruit IO.

As before, the **Notify on Action Reset** should be left at its default of unchecked.

When you're ready, click **Submit**. You'll now see the new Action added to your list!

Verify both of your Actions are **Active** before continuing!



Adafruit IO Feed Notification

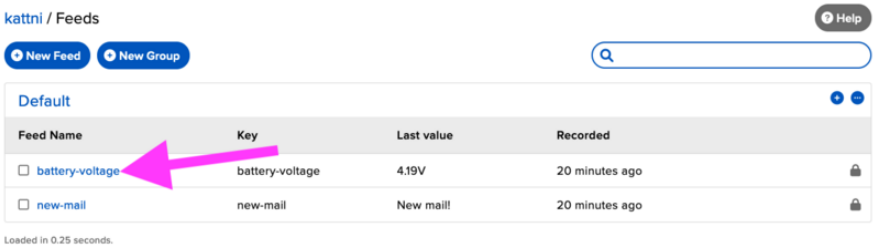
The reason the code for this project send battery voltage to Adafruit IO every twelve hours is to provide a sort of heartbeat for you to track. The idea is that, if the battery-voltage feed stops updating for a specified duration, you'll get a notification. At that point, it's probably worth checking on the hardware in your mailbox to verify that there aren't any issues (battery low/disconnected, antenna disconnected, etc).

Feed Notifications are a feature built into the Adafruit IO Feeds. For each feed, you can click to view it. You'll find all your data, and you'll also find the **Notifications** set up. This feature allows you to turn on the notification, and choose a duration for how long the feed must go without an update before you receive the notification. These notifications are sent via email.

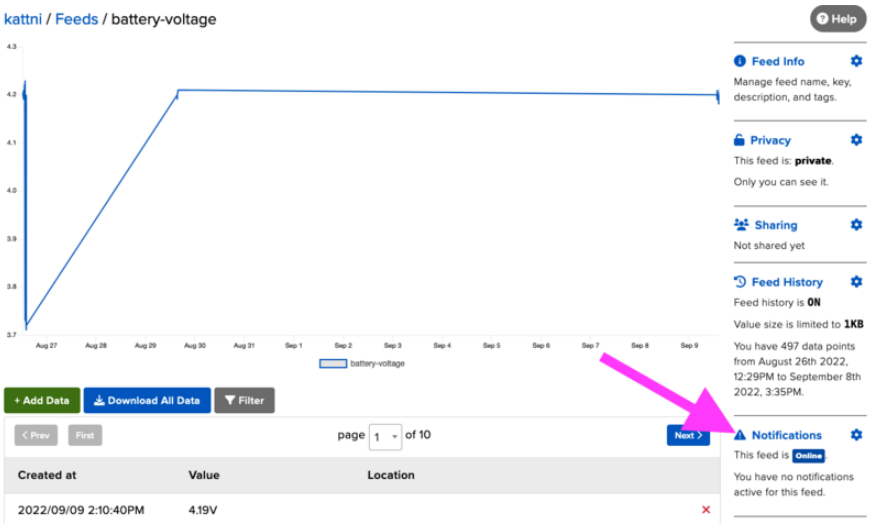
Battery Voltage Feed Notification Set Up

In Adafruit IO, click **Feeds** in the black menu bar.

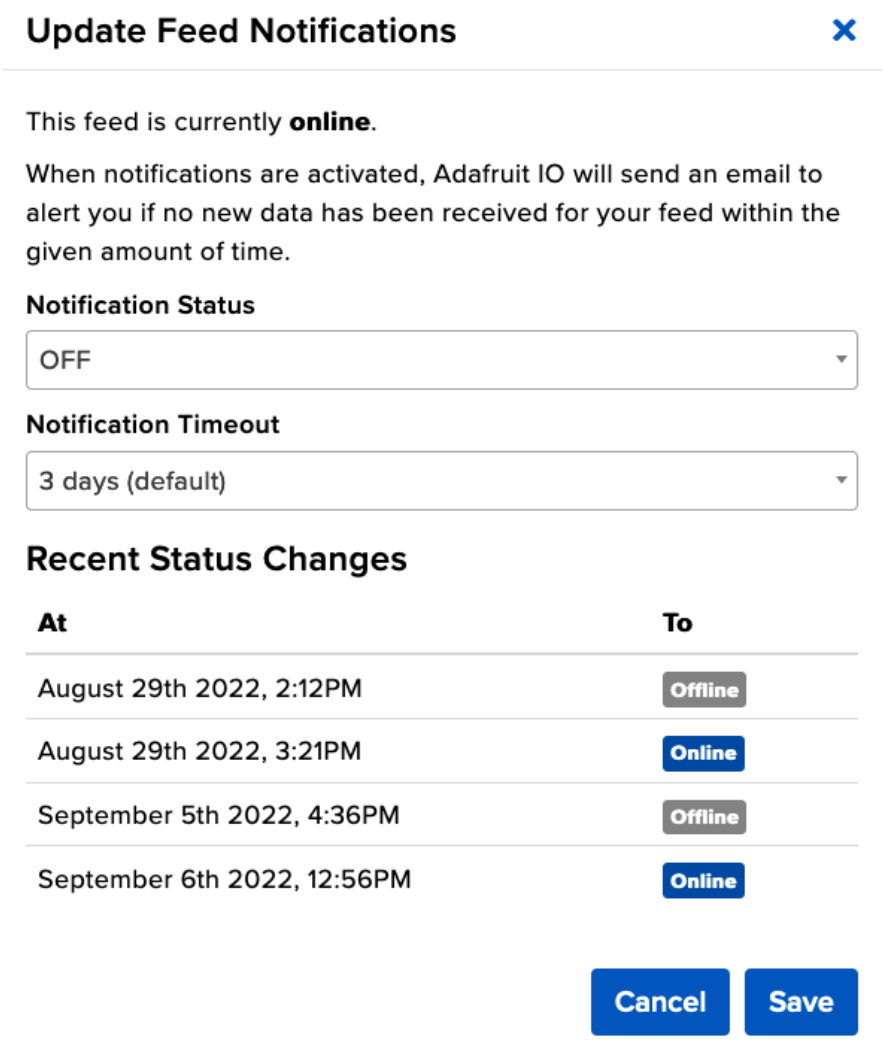
From the Feeds list, click the **battery-voltage feed**.



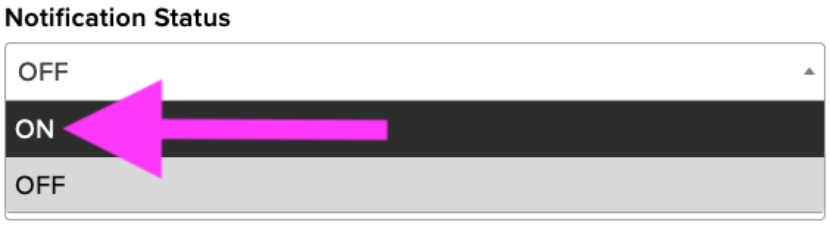
On the battery-voltage feed page, click **Notifications**.



The **Update Feed Notifications** window looks like this when opened.



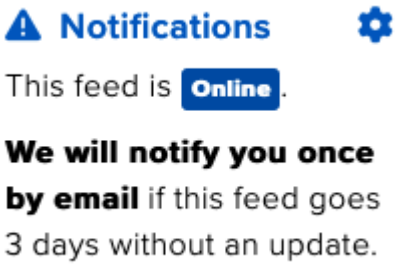
You'll want to update the **Notification Status** to **ON**.



You can leave the **Notification Timeout** as **3 days (default)**. Alternatively, you can update it to a different timeframe that works for you. Don't make it less than 12 hours, or you'll get the notification repeatedly.

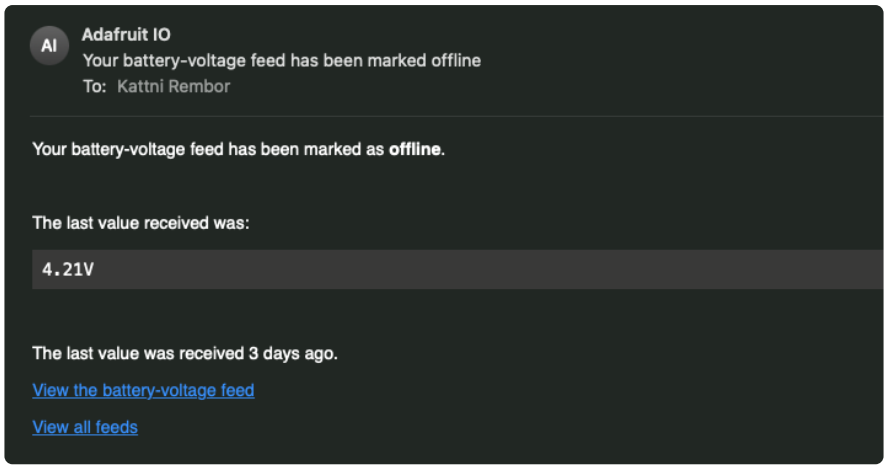
Once you've turned the notification on, and decided on a duration, click **Create**.

Now, when you look at the battery-voltage feed page, you'll see the following for Notifications. If it resembles the image shown here, you're ready to go!



Feed Notification Email

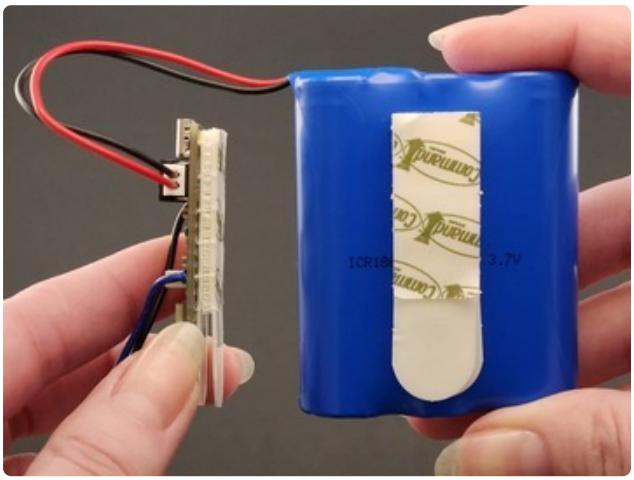
If your battery-voltage feed is quiet for the duration you specified above, you'll receive an email that looks something like the following. If that last value received was less than 3.5V, then almost certainly your battery needs to be swapped/charged. If it's above that, then something else may have gone wrong. Time to look into it!



Installation

Once you have the code on your Feather, and you've tested that everything is working successfully, it's time to install the hardware into your mailbox.

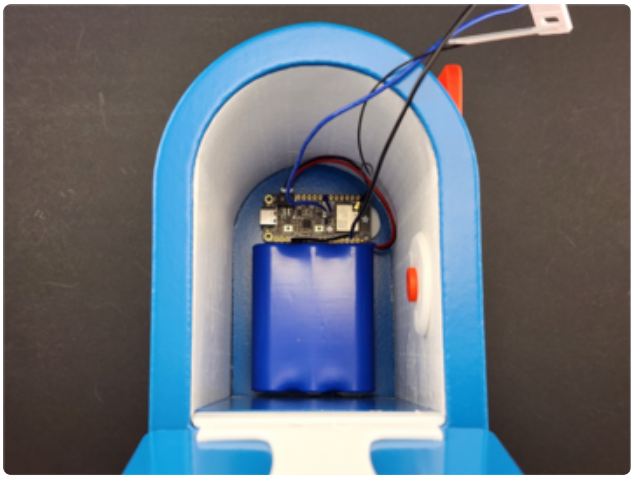
For a smooth installation experience, follow these steps in order. It's definitely possible to install things out of order, but the order below helps make things a bit simpler.



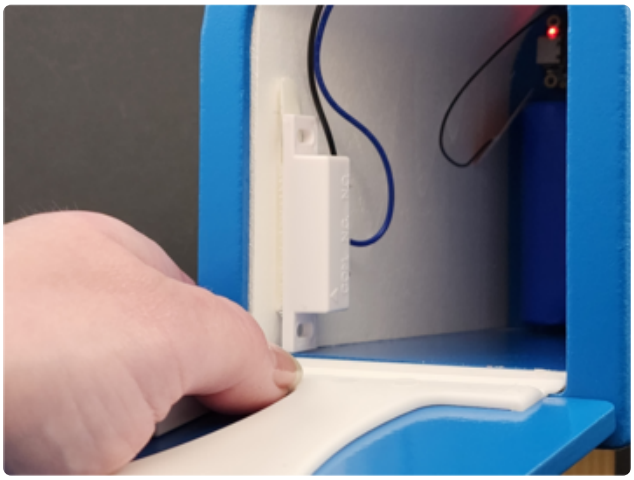
Attach your adhesive to the back of the Feather and the back of the battery.

As you will eventually need to swap the battery or update CircuitPython on the Feather, it's best to mount them in a way that they are secure, but easily removed.

This build uses Command Small Picture Hanging Strips (https://adafru.it/18fD) for mounting each piece of hardware that is attached to the Feather.

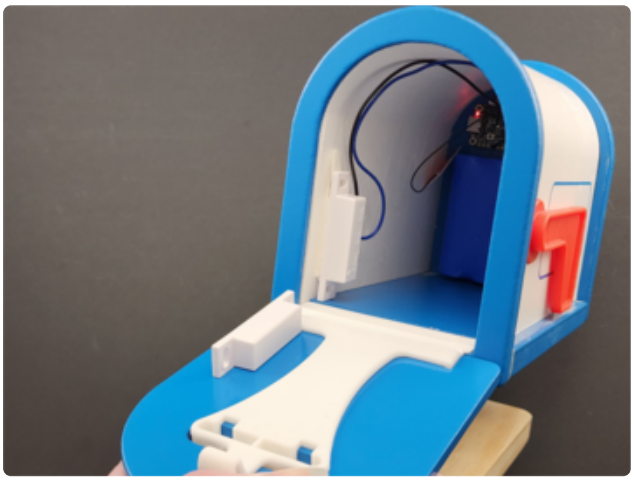


Mount the Feather and the battery to the back of the mailbox, to ensure they are out of the way of incoming mail.



Mount the reed switch just inside the mailbox door, on the side of the mailbox.

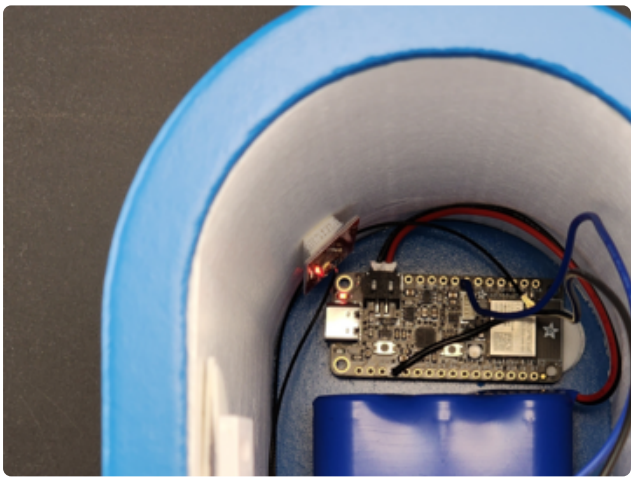
The reed switch and the magnet will need to align when the mailbox door is closed for the code to know when the door is closed. Keep in mind the magnet will be mounted on the mailbox door. It's best to use the magnet to help determine where best to mount the switch. You can temporarily tape the magnet to the switch for alignment purposes.



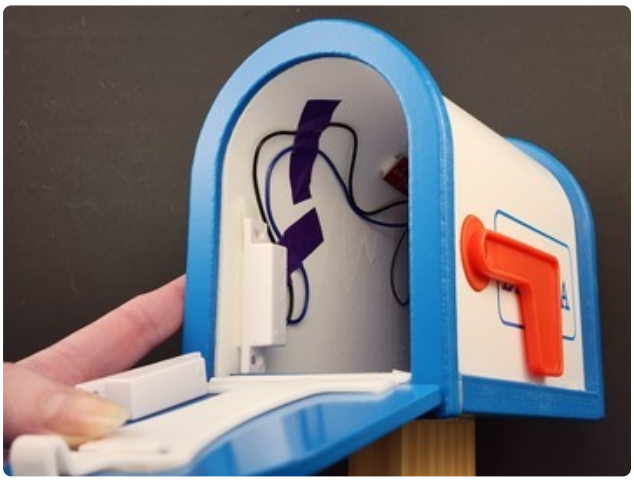
Mount the magnet on the mailbox door, ensuring alignment with the mounted switch. You can use something permanent to mount the magnet, as it is not attached to the Feather, and does not need to be removed when working with the rest of the hardware.

The magnet was mounted using a bit of a trick. The method used for this demonstration may not work in your situation due to a number of variables, for example, your mailbox door isn't completely flat. The trick is included here in case it will help you with your build.

The magnet here was mounted by applying adhesive to the side that would attach to the mailbox door, placing the magnet (adhesive side up) on the switch so they are aligned properly, and closing the mailbox door. The adhesive attached to the door, and lightly mounted the magnet. Pressure was applied to solidify the mount.



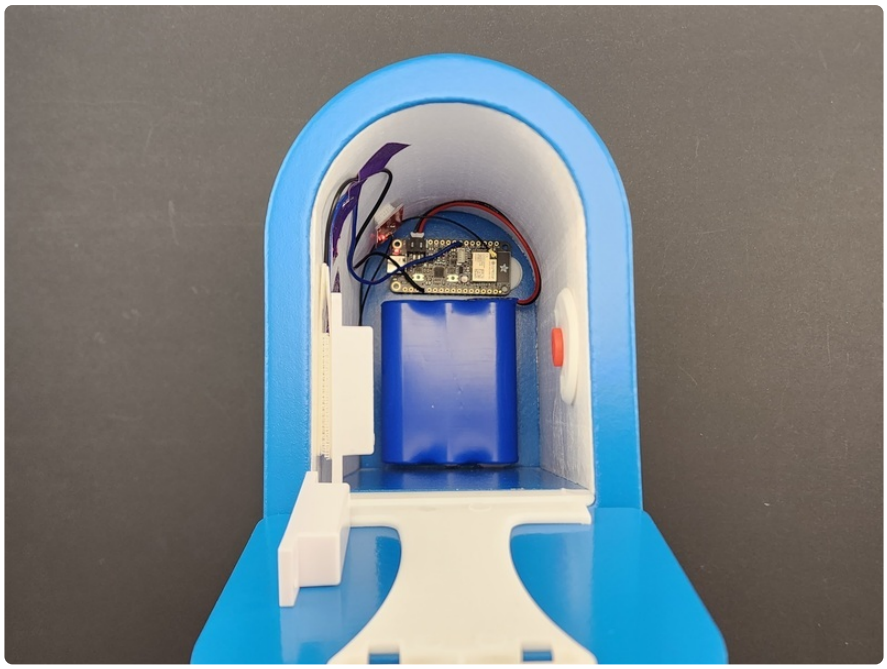
Mount the WiFi antenna towards the back, near the Feather. You can tuck the cable between the antenna, the Feather, and the battery, to keep it out of the way.



Secure the loose wires to ensure they remain out of the way of incoming mail. **Be sure to secure them to the same side of the mailbox as the switch is mounted.**

The wires here are secured using duct tape, but any tape that can handle the potential temperatures in your mailbox will work.

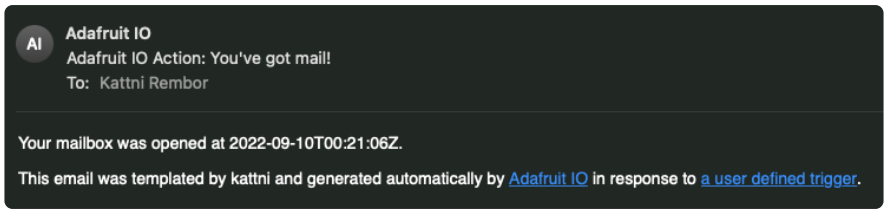
The install is complete!



Usage

You should verify your project build and Adafruit IO set up before calling it finished. To do so, simply open your mailbox door.

If you receive an email from Adafruit IO indicating you have mail, everything is good to go! If you don't receive an email, follow the steps in the previous pages to make sure everything was set up correctly.



The two interactions necessary in the future are to keep an eye out for the low battery and the silent feed notifications. When you receive the low battery notification, swap or charge your battery. If you receive the silent feed notification, go check out the project in your mailbox to make sure everything is still functioning.

Now it's time to sit back, and wait to be notified of new mail!

Power Analysis

One of the goals of this project is to use CircuitPython alarms and deep sleep to conserve power and extend battery life. The project code ends by setting up two alarms, and entering deep sleep until the alarms are triggered. However, you can't always trust the code.

There are a number of hardware variables that affect the current draw during sleep. The code may say it's entering deep sleep, but if those variables are not also accounted for, you may not actually be getting deep sleep current draw numbers. So, how do you know for sure that your microcontroller is drawing the lowest possible current? You use a **precision power monitor**, such as the Nordic Power Profiler Kit II (http://adafru.it/5048), to measure the power usage!

This page will show you how to wire the PPK II to a microcontroller using a JST connector, and what the current draw looks like on the Feather ESP32 V2 running this project code.

Wiring up the PPK

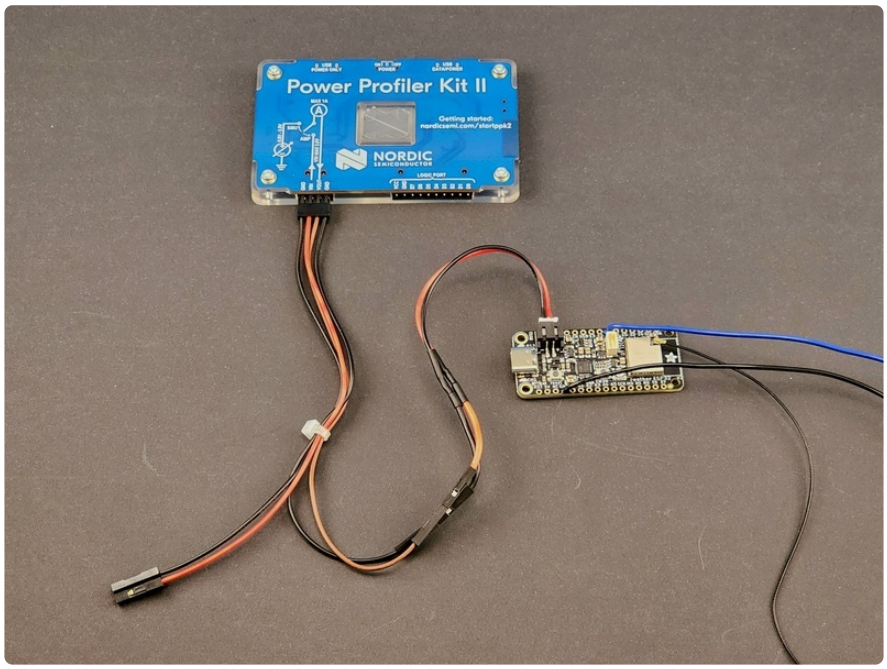
The first step is to connect your microcontroller to the PPK. This is most easily done using a microcontroller with a JST battery connector built in, such as the Feather ESP32 V2.

The PPK comes with two cables. For this, you'll need the 1x4 pin cable comprised of a black wire, red wire, brown wire and black wire, in that order.

Plug it into the 4-pin header on the PPK, so that GND is black, VIN is red, VOUT is brown, and GND is black.

Then, connect it to the Feather JST-PH battery connector as follows:

- **VOUT (brown wire)** to **positive on the Feather JST-PH connector**
- **VOUT-GND (black wire on the right)** to **negative on the Feather JST-PH connector**

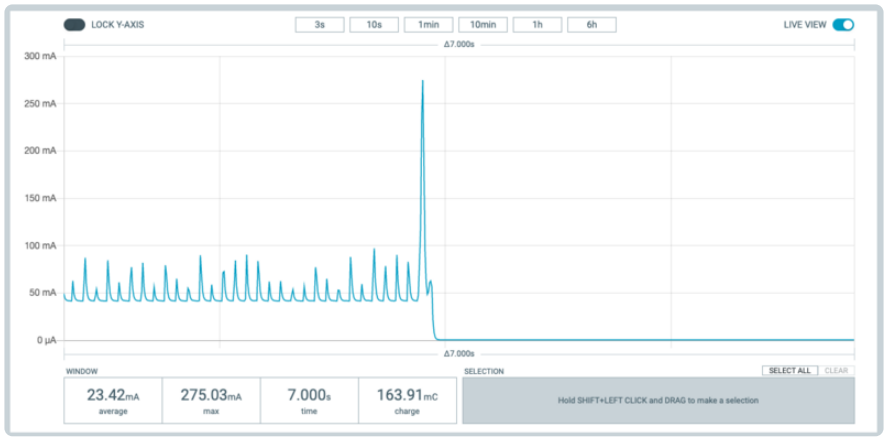


Measuring the Power Usage

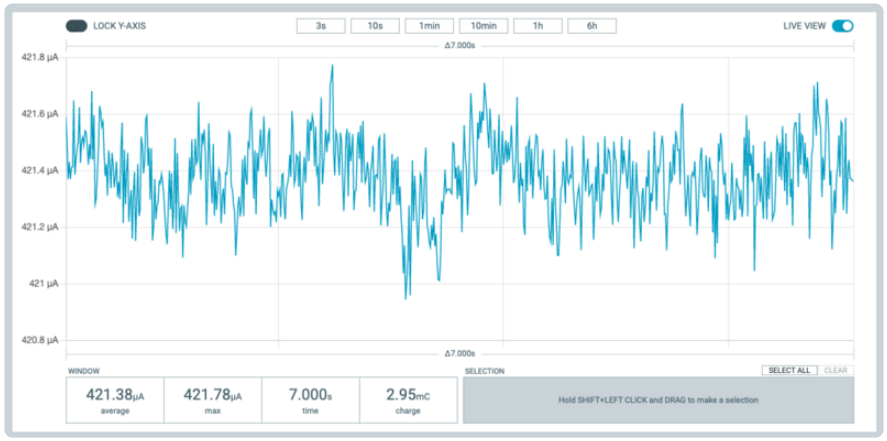
Now that everything is wired up, you're ready to measure the current draw of your Feather once it enters deep sleep.

Do not power up the PPK while connected to your microcontroller until you have disconnected ALL other power sources from the microcontroller! This includes USB.

This first graph shows the current draw while the code is running, and the dip as the Feather enters deep sleep. While the code is running, the average draw between data being sent is around 50mA, and usage spikes to around 280mA when data is sent.



This is the usage while in deep sleep. Current draw in deep sleep averages around 420uA.



Due to a known CircuitPython issue, the deep sleep current draw is presently higher on the Feather ESP32 V2. The issue is being actively investigated, and a fix is expected in the near future.