IoT Food Scale with Azure and CircuitPython

Created by Liz Clark

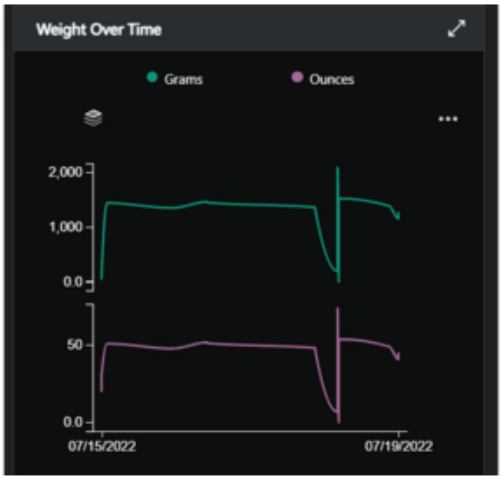
https://learn.adafruit.com/iot-food-scale-with-azure-and-circuitpython

Last updated on 2024-06-03 03:39:42 PM EDT

Table of Contents

Overview

In the NAU7802 Food Scale Learn Guide (https://adafru.it/10dX), you can build a food scale using CircuitPython, a NAU7802 STEMMA board and a strain gauge. In this guide, you'll take that project and connect it to Microsoft Azure (https://adafru.it/10dR) to log the weight measurements over time. Additionally, you'll setup a text alert when it looks like you need to refill the food container.



Tracking inventory usage over time is a great way to utilize IoT tools. It can help you see patterns and better plan for the future. In the context of pet food, it can also help you monitor your pet's diet.

Prerequisite Guides

- The Getting Started with Microsoft Azure and CircuitPython Learn Guide (https://adafru.it/10dY) goes through everything you need to get up and running with Microsoft Azure and CircuitPython. You'll need to follow along with the steps in that guide to make sure that your Azure account is setup properly to use for CircuitPython Azure IoT Central projects.

- This guide is considered part two to the NAU7802 Food Scale Learn Guide (https://adafru.it/10dX). You'll need to follow along with that guide to assemble the electronics and 3D printed parts for the food scale.

The following buttons take you to guides on the major components for reference:

Adafruit QT Py ESP32-S2

https://adafru.it/10bJ

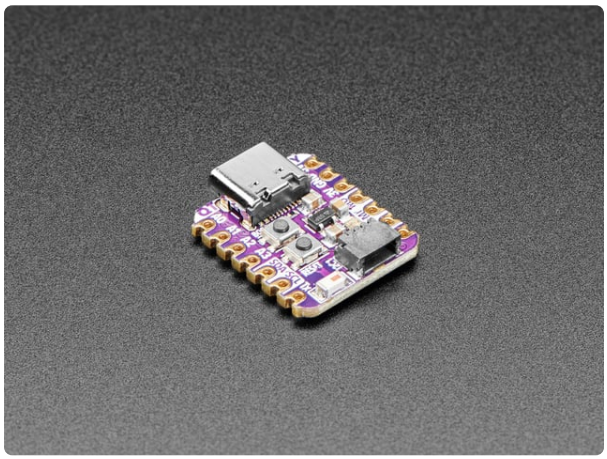
Adafruit NAU7802 24-Bit ADC

https://adafru.it/10bK

Adafruit LED Backpacks

https://adafru.it/dAo

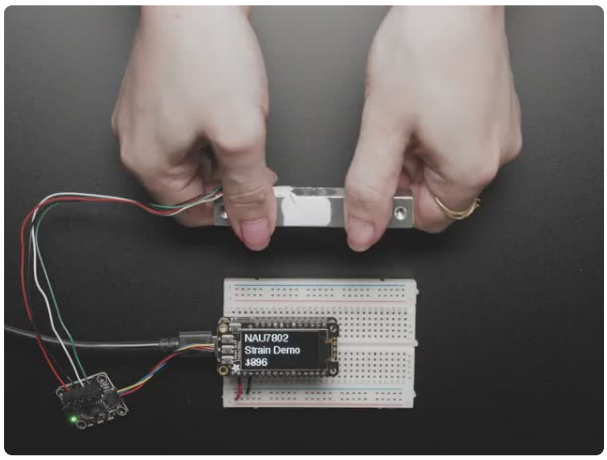
Parts



Adafruit QT Py ESP32-S2 WiFi Dev Board with STEMMA QT

What has your favorite Espressif WiFi microcontroller, comes with our favorite connector - the STEMMA QT, a chainable I2C port, and has...

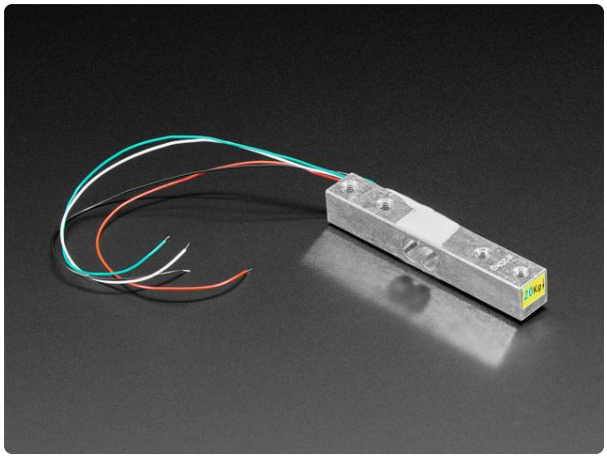
https://www.adafruit.com/product/5325



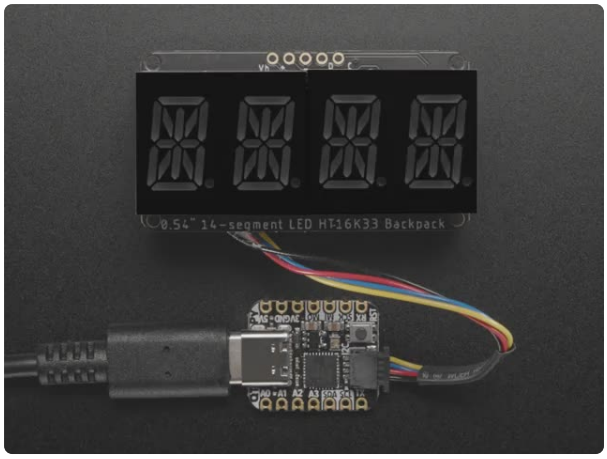
Adafruit NAU7802 24-Bit ADC - STEMMA QT / Qwiic

If you are feeling the stress and strain of modern life a Wheatstone bridge and you want to quantify it, this handy breakout will do the job, no sweat! The Adafruit...

https://www.adafruit.com/product/4538



[Strain Gauge Load Cell - 4 Wires - 20Kg]
A strain gauge is a type of electronic sensor used to measure force or strain (big surprise there). They are made of an insulating flexible backing with a metallic...
https://www.adafruit.com/product/4543



[Quad Alphanumeric Display - Yellow 0.54" Digits w/ I2C Backpack]
Display, elegantly, 012345678 or 9! Gaze, hypnotized, at ABCDEFGHIJKLM - well it can display the whole alphabet. You get the point. This is a nice, bright alphanumeric display that...
https://www.adafruit.com/product/2158

[16mm Illuminated Pushbutton - Green Momentary]
A button is a button, and an LED is a LED, but this LED illuminated button is a lovely combination of both! It's a medium sized button, large enough to press easily but not too big...
https://www.adafruit.com/product/1440

[16mm Illuminated Pushbutton - Blue Momentary]
A switch is a switch, and an LED is an LED, but this LED illuminated button is a lovely combination of both! It's a medium sized button, large enough to press easily but not too...
https://www.adafruit.com/product/1477

USB C extension

1 x USB C Round Panel Mount Extension Cable — https://www.adafruit.com/product/4218
USB C extension

---

1 x Pink and Purple Woven USB A to USB C Cable - 1 meter long — https://www.adafruit.com/product/5153
USB C to USB A

---

3 x STEMMA QT / Qwiic JST SH 4-pin Cable - 100mm Long — https://www.adafruit.com/product/4210
STEMMA QT cable

---

1 x Silicone Cover Stranded-Core Wire - 30AWG in Various Colors — https://www.adafruit.com/product/2051
Stranded-Core Wire

---

1 x M4 Screws — https://www.adafruit.com/product/1160
M4 screws

---

1 x Food storage container — https://www.target.com/p/oxo-pop-2-6qt-airtight-food-storage-container/-/A-15420440#lnk=sametab
2.6 quart storage container

---

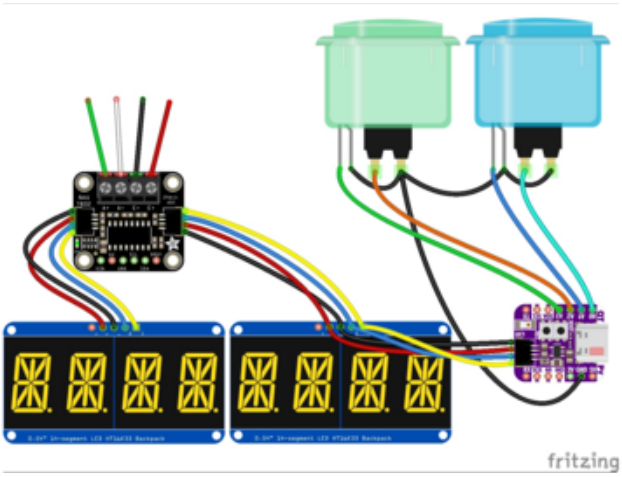
1 x Calibration Weight 1g 2g 5g 10g 20g — https://www.amazon.com/dp/B078Q3JZY7?psc=1&ref=ppx_yo2ov_dt_b_product_details
Calibration weights

Build the Food Scale

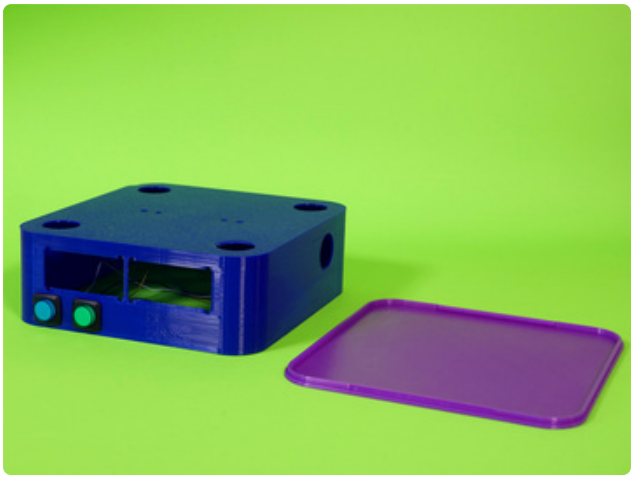
The NAU7802 Food Scale guide (https://adafru.it/10dX) has everything you need documented to build your own. Please follow along with that guide first before adding in the Azure IoT Central functionality.

Make sure to follow along with the NAU7802 Food Scale Learn Guide!

https://adafru.it/10dX



Follow the Fritzing diagram found on the Circuit Diagram page (https://adafru.it/10e0) to wire up the food scale. The Wiring page (https://adafru.it/10e1) goes into detail on soldering steps.

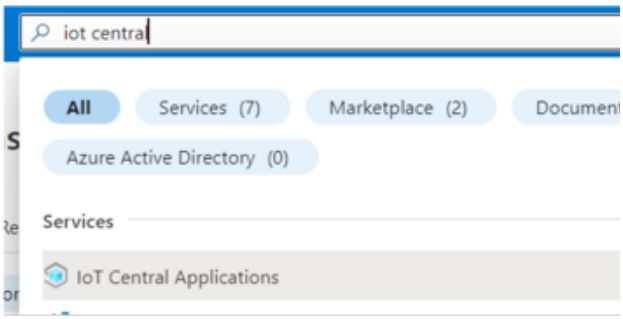


The 3D Printing page (https://adafru.it/10e2) details the 3D printed enclosure for the project. The Assembly page (https://adafru.it/10e3) goes through how to mount all of the components properly.

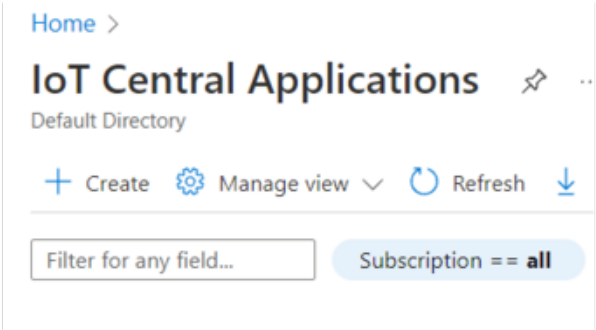
The Calibration page (https://adafru.it/10e4) shows you how to properly calibrate your NAU7802 and strain gauge to get accurate weight measurements.

Create an IoT Central Application

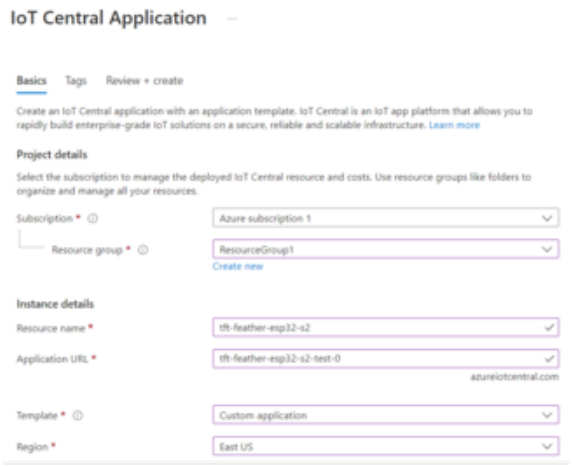
There are two Azure IoT options: IoT Hub (https://adafru.it/10dP) and IoT Central (https://adafru.it/10dQ). In this guide, you'll be using IoT Central, which is the simpler of the two options to get started with.



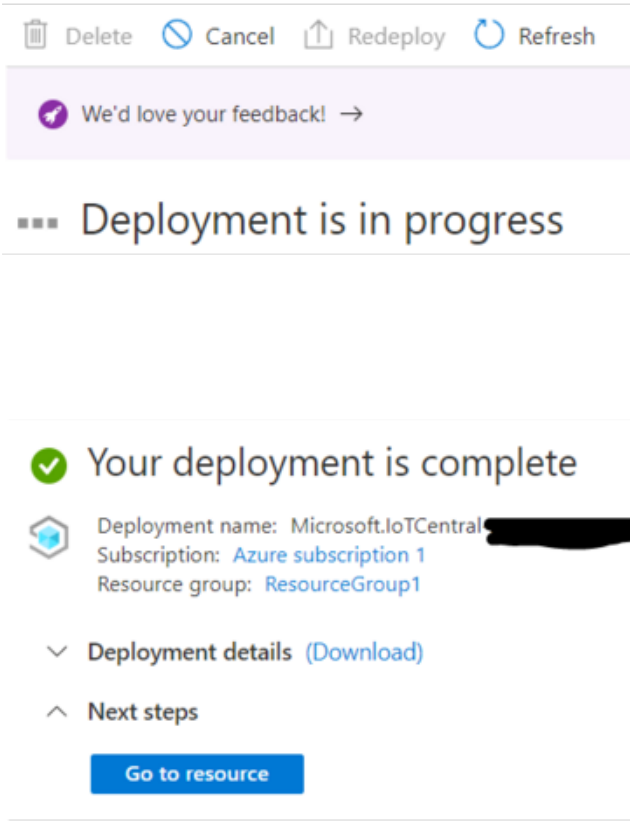
After logging into Microsoft Azure, in the Azure Portal homepage (https://adafru.it/10dR)search for "IoT Central" and navigate to IoT Central Applications (https://adafru.it/ETp)



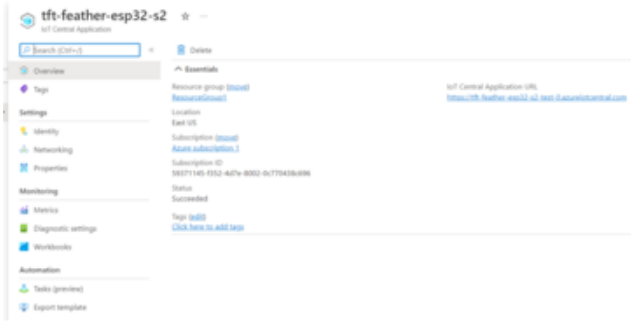
Create a new application by clicking on **Create**.



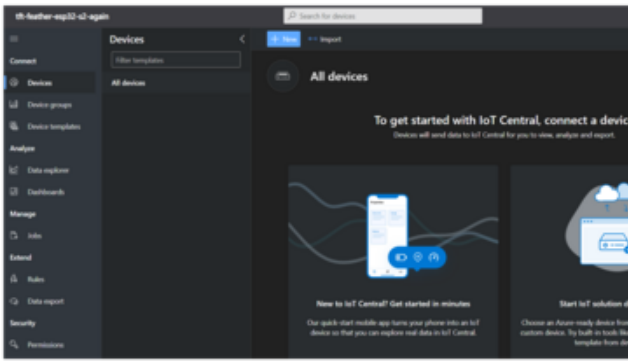
On the Basics page, you'll enter details about your application. The Application URL is how you will access your application's homepage.



After creating the application, the deployment of the app will begin. Once it finishes, you'll see a message that deployment is complete. Then, click on **Go To Resource**.

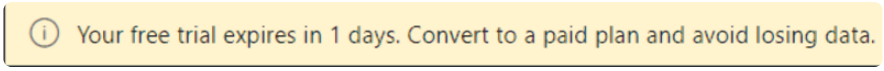


This brings you to your application's overview page. Click on the **link on the right-hand side of the screen** to view your app.



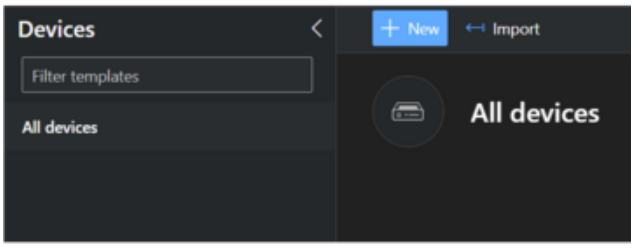
The link brings you to your app's homepage. Here you can connect devices, create dashboards, view data and control all aspects of the app.

Connect Your Device

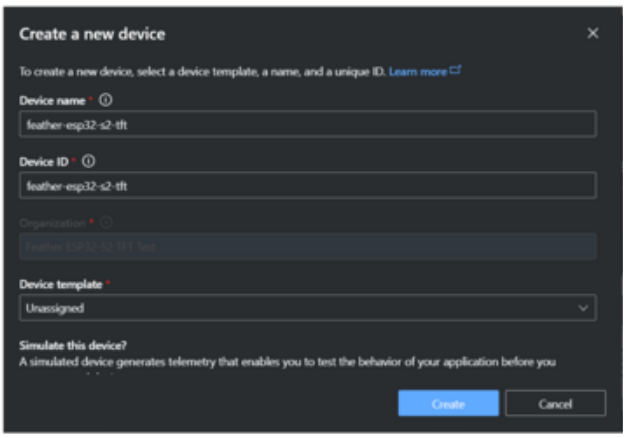


If you are on a free plan, then your data will only be accessible for 7 days. After 7 days, your data and application are deleted unless you convert to a paid plan.

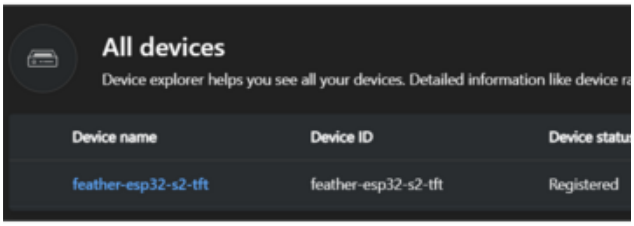
After creating your application, you'll connect a device instance for your development board that you'll be connecting to Azure with CircuitPython. This process will create the keys you need to include in your **secrets.py** file.



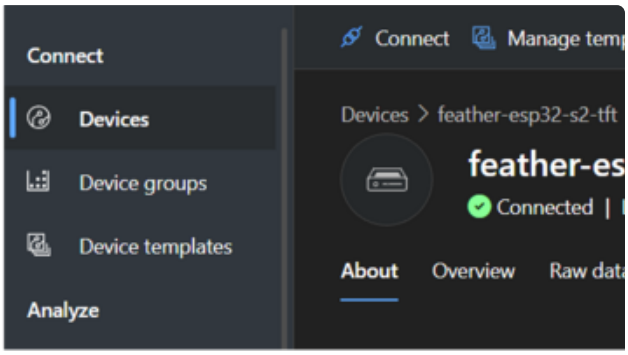
To connect your device, click on the **blue New button**.



This opens the new device screen. Name your device. The device ID will be identical to the name. Then click **Create**.



You'll see your device appear under the All devices list. Click on your **device**.



Click on **Connect**.

This opens the Device connection groups window that has all of your secret connection keys. You will need the ID scope, Device ID and Primary key for your **secrets.py** file.

Now you're ready to connect to Microsoft Azure with CircuitPython!

CircuitPython

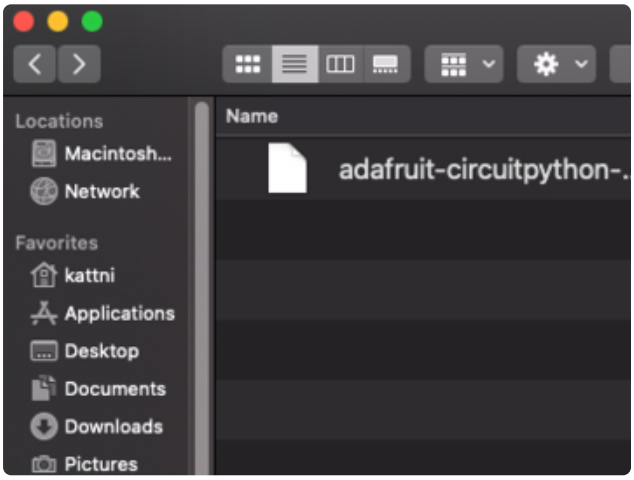
CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files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

CircuitPython Quickstart

Follow this step-by-step to quickly get CircuitPython running on your board.

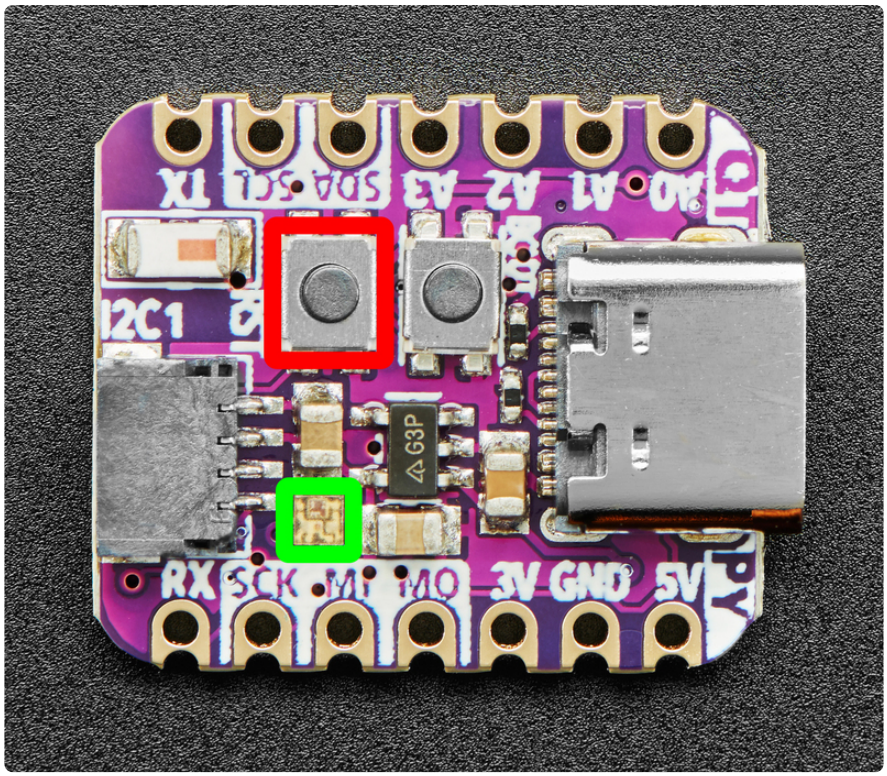
Download the latest version of CircuitPython for this board via circuitpython.org

https://adafru.it/XCk



Click the link above to download the latest CircuitPython UF2 file.

Save it wherever is convenient for you.



The board above has a chip antenna, not the u.FI connector, but the process is the same.

Plug your board into your computer, using a known-good data-sync cable, directly, or via an adapter if needed.

Click the **reset** button once (highlighted in red above), and then click it again when you see the **RGB status LED(s)** (highlighted in green above) turn purple (approximately half a second later). Sometimes it helps to think of it as a "slow double-click" of the reset button.

If you do not see the LED turning purple, you will need to reinstall the UF2 bootloader. See the **Factory Reset** page in this guide for details.

On some very old versions of the UF2 bootloader, the status LED turns red instead of purple.

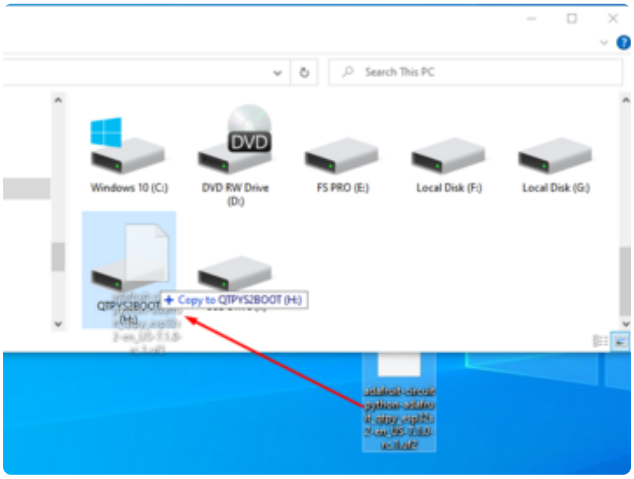
For this board, tap reset and wait for the LED to turn purple, and as soon as it turns purple, tap reset again. The second tap needs to happen while the LED is still purple.

Once successful, you will see the **RGB status LED(s)** turn green (highlighted in green above). If you see red, try another port, or if you're using an adapter or hub, try without the hub, or different adapter or hub.

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

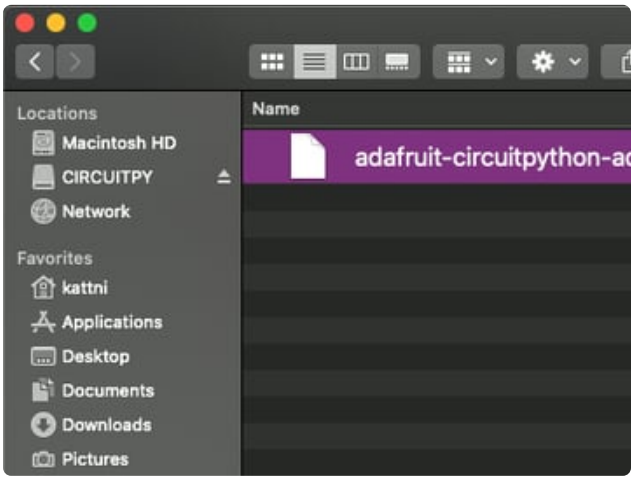
A lot of people end up using charge-only USB cables and it is very frustrating! **Make sure you have a USB cable you know is good for data sync.**

If after several tries, and verifying your USB cable is data-ready, you still cannot get to the bootloader, it is possible that the bootloader is missing or damaged. Check out the Factory Reset page for details on resolving this issue.



You will see a new disk drive appear called **QTPYS2BOOT**.

Drag the **adafruit_circuitpython_etc.uf2** file to **QTPYS2BOOT**.



The **BOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

That's it!

Code the IoT Food Scale

Once you've finished setting up your QT Py ESP32-S2 with CircuitPython, you can access the code and necessary libraries by downloading the Project Bundle.

To do this, click on the **Download Project Bundle** button in the window below. It will download as a zipped folder.

```
# SPDX-FileCopyrightText: 2022 Liz Clark for Adafruit Industries
# SPDX-License-Identifier: MIT

import time
import json
import board
from digitalio import DigitalInOut, Direction, Pull
from adafruit_ht16k33.segments import Seg14x4
from cedargrove_nau7802 import NAU7802
from calibration import calibration
import rtc
import socketpool
import wifi
import adafruit_ntp
from adafruit_azureiot import IoTCentralDevice

#  I2C setup with STEMMA port
i2c = board.STEMMA_I2C()
#  alphanumeric segment displpay setup
#  using two displays together
display = Seg14x4(i2c, address=(0x70, 0x71))
#  start-up text
display.print("*HELLO* ")
#  button LEDs
blue = DigitalInOut(board.A1)
blue.direction = Direction.OUTPUT
green = DigitalInOut(board.A3)
green.direction = Direction.OUTPUT
#  buttons setup
blue_btn = DigitalInOut(board.A0)
blue_btn.direction = Direction.INPUT
blue_btn.pull = Pull.UP
green_btn = DigitalInOut(board.A2)
green_btn.direction = Direction.INPUT
green_btn.pull = Pull.UP
# nau7802 setup
nau7802 = NAU7802(board.STEMMA_I2C(), address=0x2A, active_channels=2)
nau7802.gain = 128
enabled = nau7802.enable(True)

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

print("Connecting to WiFi...")
wifi.radio.connect(secrets["ssid"], secrets["password"])

print("Connected to WiFi!")
#  check system time
if time.localtime().tm_year < 2022:
    print("Setting System Time in UTC")
    pool = socketpool.SocketPool(wifi.radio)
    ntp = adafruit_ntp.NTP(pool, tz_offset=-4)
    # NOTE: This changes the system time so make sure you aren't assuming that time
    # doesn't jump.
    rtc.RTC().datetime = ntp.datetime

else:
    print("Year seems good, skipping set time.")
```

```
# Create an IoT Central device client and connect
esp = None
pool = socketpool.SocketPool(wifi.radio)
device = IoTCentralDevice(
    pool, esp, secrets['id_scope'], secrets['device_id'],
secrets['device_primary_key']
)
display.fill(0)
display.print("DIALING*")
print("Connecting to Azure IoT Central...")
device.connect()
display.print("CONNECTD")
print("Connected to Azure IoT Central!")
device.disconnect()
print("Disconnected")
#  zeroing function
def zero_channel():
    """Initiate internal calibration for current channel; return raw zero
    offset value. Use when scale is started, a new channel is green_btned, or to
    adjust for measurement drift. Remove weight and tare from load cell before
    executing."""
    blue.value = True
    print(
        "channel %1d calibrate.INTERNAL: %5s"
        % (nau7802.channel, nau7802.calibrate("INTERNAL"))
    )
    blue.value = False
    print(
        "channel %1d calibrate.OFFSET:   %5s"
        % (nau7802.channel, nau7802.calibrate("OFFSET"))
    )
    blue.value = True
    zero_offset = read_raw_value(100)  # Read 100 samples to establish zero offset
    print("...channel %1d zeroed" % nau7802.channel)
    blue.value = False
    return zero_offset
#  read raw value function
def read_raw_value(samples=100):
    """Read and average consecutive raw sample values. Return average raw value."""
    sample_sum = 0
    sample_count = samples
    while sample_count > 0:
        if nau7802.available:
            sample_sum = sample_sum + nau7802.read()
            sample_count -= 1
    return int(sample_sum / samples)
#  function for finding the average of an array
def find_average(num):
    count = 0
    for n in num:
        count = count + n
    average = count / len(num)
    return average
#  calibration function
def calculateCalibration(array):
    for _ in range(10):
        blue.value = True
        green.value = False
        nau7802.channel = 1
        print("channel %1.0f raw value: %7.0f" % (nau7802.channel,
abs(read_raw_value())))
        array.append(abs(read_raw_value()))
        blue.value = False
        green.value = True
        time.sleep(1)
    green.value = False
    avg = find_average(array)
    return avg
#  blink LED function
```

```
def blink(led, amount, count):
    for _ in range(count):
        led.value = True
        time.sleep(amount)
        led.value = False
        time.sleep(amount)
#  send data to azure with ounces and grams
def send_to_azure(current_oz, current_grams):
    #  turn on green LED
    green.value = True
    display.print("DIALING*")
    #  connect to azure
    device.reconnect()
    #  turn on blue LED
    blue.value = True
    display.print("CONNECTD")
    time.sleep(1)
    display.print("SENDING!")
    #  send JSON of ounces and grams
    message = {"Ounces": current_oz, "Grams": current_grams}
    device.send_telemetry(json.dumps(message))
    display.fill(0)
    display.print("SENT!")
    #  disconnect and turn off LEDs
    device.disconnect()
    green.value = False
    blue.value = False
#  zeroing on startup
display.fill(0)
display.marquee("CLEAR SCALE CLEAR", 0.3, False)
time.sleep(2)
display.fill(0)
display.print("ZEROING")
time.sleep(3)
#  zeroing each channel
nau7802.channel = 1
zero_channel()  # Calibrate and zero channel
display.fill(0)
display.print("STARTING")

#  variables and states
clock = time.monotonic() #  time.monotonic() device
reset_clock = time.monotonic()
long_clock = time.monotonic()
mode = "run"
mode_names = ["SHOW OZ?", "   GRAMS?", "   ZERO?", "CALIBRTE", " OFFSET?", "TO
AZURE"]
stage = 0
zero_stage = 0
weight_avg = 0
zero_avg = 0
show_oz = True
show_grams = False
zero_out = False
calibrate_mode = False
blue_btn_pressed = False
green_btn_pressed = False
run_mode = True
avg_read = []
avg_grams = []
avg_oz = []
values = []
val_offset = 0
avg_values = []

#  initial reading from the scale
for w in range(5):
    nau7802.channel = 1
    value = read_raw_value()
```

```
    #  takes value reading and divides with by the offset value
    #  to get the weight in grams
    grams = value / calibration['offset_val']
    avg_read.append(grams)
    if len(avg_read) > 4:
        the_avg = find_average(avg_read)
        oz = the_avg / 28.35
        display.print("   %0.1f oz" % oz)
        avg_read.clear()
    time.sleep(1)

while True:
    #  button debouncing
    if blue_btn.value and blue_btn_pressed:
        blue_btn_pressed = False
    if green_btn.value and green_btn_pressed:
        green_btn_pressed = False
        green.value = False
    #  default run mode
    #  checks NAU7802 every 2 seconds
    if run_mode is True and (time.monotonic() - clock) > 2:
        nau7802.channel = 1
        value = read_raw_value()
        value = abs(value) - val_offset
        values.append(value)
        #  takes value reading and divides with by the offset value
        #  to get the weight in grams
        grams = value / calibration['offset_val']
        oz = grams / 28.35
        avg_grams.append(grams)
        avg_oz.append(oz)
        if show_oz is True:
            #  append reading
            avg_read.append(oz)
            label = "oz"
        if show_grams is True:
            avg_read.append(grams)
            label = "g"
        if len(avg_read) > 10:
            the_avg = find_average(avg_read)
            the_grams = find_average(avg_grams)
            the_ounces = find_average(avg_oz)
            display.print("   %0.1f %s" % (the_avg, label))
            avg_read.clear()
            avg_grams.clear()
            avg_oz.clear()
            val_offset += 10
        clock = time.monotonic()
    if (time.monotonic() - reset_clock) > 43200:
        run_mode = False
        show_oz = False
        show_grams = False
        zero_out = True
        reset_clock = time.monotonic()
    #  if you press the change mode button
    if (not green_btn.value and not green_btn_pressed) and run_mode:
        green.value = True
        #  disables run mode (stops weighing)
        run_mode = False
        show_oz = False
        show_grams = False
        #  mode is set to 0
        mode = 0
        #  display shows the mode option
        display.print(mode_names[mode])
        blue.value = True
        green_btn_pressed = True
    #  advances through the modes menu
    if (not green_btn.value and not green_btn_pressed) and mode != "run":
```

```
        green.value = True
        #  counts up to 4 and loops back to 0
        mode = (mode+1) % 6
        #  updates display
        display.print(mode_names[mode])
        green_btn_pressed = True
    #  if you select show_oz
    if (not blue_btn.value and not blue_btn_pressed) and mode == 0:
        #  show_oz is set as the state
        show_oz = True
        label = "oz"
        blue.value = False
        #  goes back to weighing mode
        mode = "run"
        blue_btn_pressed = True
        display.print("   %0.1f %s" % (the_avg, label))
        run_mode = True
    #  if you select show_grams
    if (not blue_btn.value and not blue_btn_pressed) and mode == 1:
        #  show_grams is set as the state
        show_grams = True
        label = "g"
        blue.value = False
        #  goes back to weighing mode
        mode = "run"
        blue_btn_pressed = True
        display.print("   %0.1f %s" % (the_avg, label))
        run_mode = True
    #  if you select zero_out
    if (not blue_btn.value and not blue_btn_pressed) and mode == 2:
        #  zero_out is set as the state
        #  can zero out the scale without full recalibration
        zero_out = True
        blue.value = False
        mode = "run"
        blue_btn_pressed = True
    #  if you select calibrate_mode
    if (not blue_btn.value and not blue_btn_pressed) and mode == 3:
        #  calibrate_mode is set as the state
        #  starts up the calibration process
        calibrate_mode = True
        blue.value = False
        mode = "run"
        blue_btn_pressed = True
    #  if you select the offset
    if (not blue_btn.value and not blue_btn_pressed) and mode == 4:
        #  displays the curren offset value stored in the code
        blue.value = False
        display.fill(0)
        display.print("%0.4f" % calibration['offset_val'])
        time.sleep(5)
        mode = "run"
        #  goes back to weighing mode
        show_oz = True
        label = "oz"
        display.print("   %0.1f %s" % (the_avg, label))
        run_mode = True
        blue_btn_pressed = True
    if (not blue_btn.value and not blue_btn_pressed) and mode == 5:
        blue.value = False
        display.fill(0)
        #  sends data to azure
        send_to_azure(the_ounces, the_grams)
        time.sleep(1)
        mode = "run"
        #  goes back to weighing mode
        show_oz = True
        label = "oz"
        display.print("   %0.1f %s" % (the_avg, label))
```

```
        run_mode = True
        blue_btn_pressed = True
    #  if the zero_out state is true
    if zero_out and zero_stage == 0:
        blue_btn_pressed = True
        #  clear the scale for zeroing
        display.fill(0)
        display.print("REMOVE ")
        zero_stage = 1
        blue.value = True
        green.value = True
    if (not blue_btn.value and not blue_btn_pressed) and zero_stage == 1:
        green.value = False
        #  updates display
        display.fill(0)
        display.print("ZEROING")
        blue.value = False
        #  runs zero_channel() function on both channels
        nau7802.channel = 1
        zero_channel()
        display.fill(0)
        display.print("ZEROED ")
        zero_out = False
        zero_stage = 0
        #  goes into weighing mode
        val_offset = 0
        run_mode = True
        show_oz = True
        label = "oz"
        display.print("   %0.1f %s" % (the_avg, label))
    #  the calibration process
    #  each step is counted in stage
    #  blue button is pressed to advance to the next stage
    if calibrate_mode is True and stage == 0:
        blue_btn_pressed = True
        #  clear the scale for zeroing
        display.fill(0)
        display.print("REMOVE ")
        stage = 1
        blue.value = True
    #  stage 2
    if (not blue_btn.value and not blue_btn_pressed) and stage == 1:
        blue_btn_pressed = True
        #  runs the zero out function
        display.fill(0)
        display.print("ZEROING")
        blue.value = False
        nau7802.channel = 1
        zero_channel()
        display.fill(0)
        display.print("ZEROED ")
        stage = 2
        blue.value = True
    #  stage 3
    if (not blue_btn.value and not blue_btn_pressed) and stage == 2:
        blue_btn_pressed = True
        blue.value = False
        display.print("STARTING")
        blink(blue, 0.5, 3)
        zero_readings = []
        display.print("AVG ZERO")
        #  runs the calculateCallibration function
        #  takes 10 raw readings, stores them into an array and gets an average
        zero_avg = calculateCalibration(zero_readings)
        stage = 3
        display.fill(0)
        display.print("DONE")
        blue.value = True
    #  stage 4
```

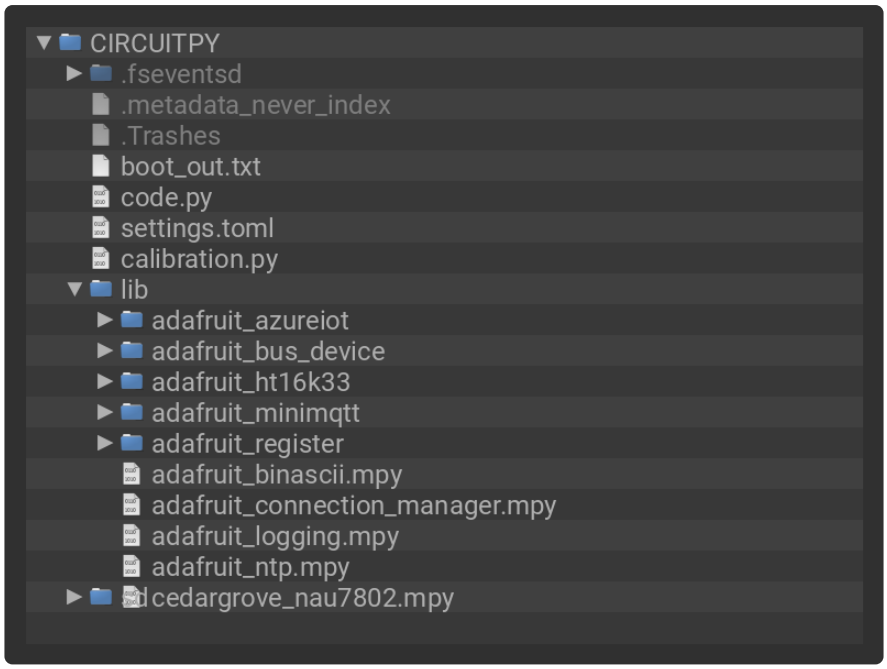
```
    if (not blue_btn.value and not blue_btn_pressed) and stage == 3:
        #  place the known weight item
        #  item's weight matches calibration['weight'] in grams
        blue_btn_pressed = True
        blue.value = False
        display.fill(0)
        display.print("PUT ITEM")
        stage = 4
        blue.value = True
    #  stage 5
    if (not blue_btn.value and not blue_btn_pressed) and stage == 4:
        blue_btn_pressed = True
        blue.value = False
        display.fill(0)
        display.print("WEIGHING")
        weight_readings = []
        #  weighs the item 10 times, stores the readings in an array & averages them
        weight_avg = calculateCalibration(weight_readings)
        #  calculates the new offset value
        calibration['offset_val'] = (weight_avg-zero_avg) / calibration['weight']
        display.marquee("%0.2f - CALIBRATED " % calibration['offset_val'], 0.3,
False)
        stage = 5
        display.fill(0)
        display.print("DONE")
        blue.value = True
    #  final stage
    if (not blue_btn.value and not blue_btn_pressed) and stage == 5:
        blue_btn_pressed = True
        zero_readings.clear()
        weight_readings.clear()
        calibrate_mode = False
        blue.value = False
        #  goes back into weighing mode
        show_oz = True
        label = "oz"
        display.print("   %0.1f %s" % (the_avg, label))
        val_offset = 0
        run_mode = True
        #  resets stage
        stage = 0
```

Upload the Code and Libraries to the QT Py ESP32-S2

After downloading the Project Bundle, plug your QT Py ESP32-S2 into the computer's USB port with a known good USB data+power cable. You should see a new flash drive appear in the computer's File Explorer or Finder (depending on your operating system) called **CIRCUITPY**. Unzip the folder and copy the following items to the QT Py ESP32-S2's **CIRCUITPY** drive.

- **lib** folder
- **calibration.py**
- **code.py**

Your QT Py ESP32-S2 **CIRCUITPY** drive should look like this after copying the **lib** folder, **calibration.py** file and the **code.py** file.



Install the **cedargrove_nau7802** CircuitPython Library

Follow along with the steps outlined in this guide (https://adafru.it/10bN) to download the **cedargrove_nau7802** CircuitPython library and upload it to your QT Py ESP32-S2 **CIRCUITPY** drive **lib** folder. The library is a part of the CircuitPython Community Bundle (https://adafru.it/ENC).

calibration.py File

The **calibration.py** file holds two important values for the main **code.py** file: `offset_val` and `weight`. `weight` is the known weight in grams that you use to calibrate the NAU7802. The `offset_val` is the calibration number used to divide against the raw value from the NAU7802 to find the actual weight on top of the scale.

Each strain gauge is slightly different, so you'll want to edit the **calibration.py** file with your known weight item and the `offset_val` you generate after running the calibration mode in the **code.py** file.

secrets.py

You will need to create and add a **secrets.py** file to your **CIRCUITPY** drive.
Your **secrets.py** file will need to include the following information:

```
secrets = {
    'ssid' : 'YOUR-SSID-HERE',
    'password' : 'YOUR-SSID-PASSWORD-HERE',
```

```
    'id_scope' : 'YOUR-AZURE-ID-SCOPE-HERE',
    'device_id' : 'YOUR-AZURE-DEVICE-ID-HERE',
    'device_primary_key' : 'YOUR-AZURE-DEVICE-PRIMARY-KEY-HERE',
     }
```

You'll gather your ID scope, device ID and device primary key from your device connection groups page in your Azure application. Make sure to refer to the Connect Your Device (https://adafru.it/-F4) page in this guide to see the process for accessing the keys.

How the CircuitPython Code Works

The CircuitPython code is identical to the original NAU7802 Food Scale CircuitPython code (https://adafru.it/10e5) as far as scale functionality. This version of the code adds functionality for connecting to, and sending data to, Azure.

Connect to WiFi and Azure

The code begins by connecting to WiFi and grabbing the date and time using the `adafruit_ntp` library.

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

print("Connecting to WiFi...")
wifi.radio.connect(secrets["ssid"], secrets["password"])

print("Connected to WiFi!")
#  check system time
if time.localtime().tm_year &lt; 2022:
    print("Setting System Time in UTC")
    pool = socketpool.SocketPool(wifi.radio)
    ntp = adafruit_ntp.NTP(pool, tz_offset=0)
    # NOTE: This changes the system time so make sure you aren't assuming that time
    # doesn't jump.
    rtc.RTC().datetime = ntp.datetime

else:
    print("Year seems good, skipping set time.")
```

Then, a connection is established with Microsoft Azure. The alphanumeric display updates its text to show what is going on with the connection process.

```
# Create an IoT Central device client and connect
esp = None
pool = socketpool.SocketPool(wifi.radio)
device = IoTCentralDevice(
    pool, esp, secrets['id_scope'], secrets['device_id'],
secrets['device_primary_key']
```

```
)
display.fill(0)
display.print("DIALING*")
print("Connecting to Azure IoT Central...")
device.connect()
display.print("CONNECTD")
print("Connected to Azure IoT Central!")
```

Send Data to Azure Function

The `send_to_azure()` function sends the current weight in ounces and grams to Azure with `device.send_telemetry(json.dumps(message))`. It also uses the buttons' LEDs and alphanumeric display to indicate the processes in the code.

```
#  send data to azure with ounces and grams
def send_to_azure(current_oz, current_grams):
    #  turn on green LED
    green.value = True
    display.print("DIALING*")
    #  connect to azure
    device.reconnect()
    #  turn on blue LED
    blue.value = True
    display.print("CONNECTD")
    time.sleep(1)
    display.print("SENDING!")
    #  send JSON of ounces and grams
    message = {"Ounces": current_oz, "Grams": current_grams}
    device.send_telemetry(json.dumps(message))
    display.fill(0)
    display.print("SENT!")
    #  disconnect and turn off LEDs
    device.disconnect()
    green.value = False
    blue.value = False
```

Logging Ounces and Grams

The alphanumeric display shows either ounces or grams depending on the mode selected. In the background though, both the ounces and grams are logged in `the_grams` and `the_ounces`.

```
grams = value / calibration['offset_val']
        oz = grams / 28.35
        avg_grams.append(grams)
        avg_oz.append(oz)
        if show_oz is True:
            #  append reading
            avg_read.append(oz)
            label = "oz"
        if show_grams is True:
            avg_read.append(grams)
            label = "g"
        if len(avg_read) &gt; 10:
            the_avg = find_average(avg_read)
            the_grams = find_average(avg_grams)
            the_ounces = find_average(avg_oz)
```

```
            display.print("   %0.1f %s" % (the_avg, label))
            avg_read.clear()
            avg_grams.clear()
            avg_oz.clear()
```

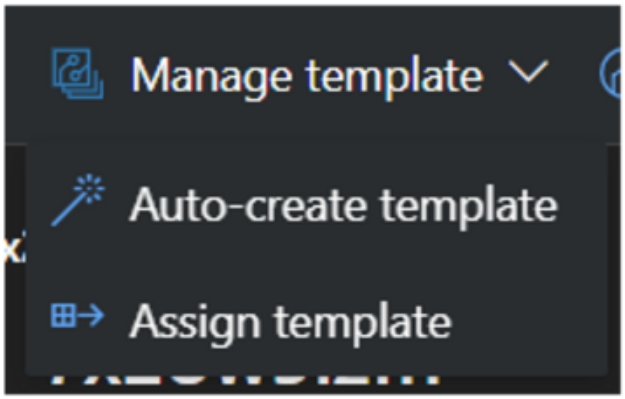
Send Weight to Azure

The food scale is coded to have a selection of mode functionality. One of the functions in the list is sending the data to Azure. If `mode` is `5` and the blue button is pressed, then `send_to_azure(the_ounces, the_grams)` is called and sends the current weigh in ounces and grams is sent to your Azure IoT Central application. Then, the scale goes back to weighing mode.

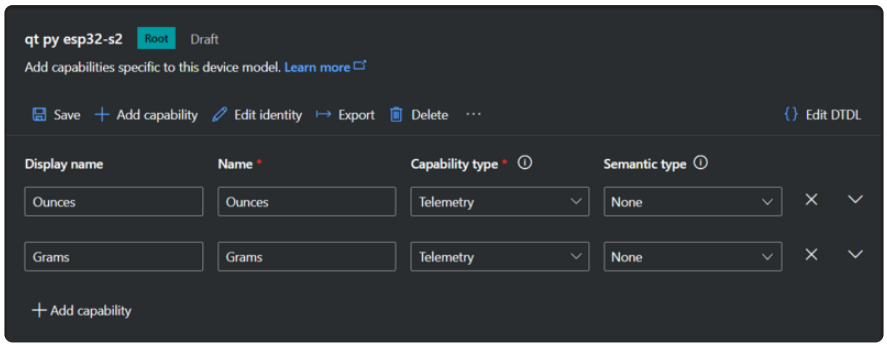
```
if (not blue_btn.value and not blue_btn_pressed) and mode == 5:
        blue.value = False
        display.fill(0)
        #  sends data to azure
        send_to_azure(the_ounces, the_grams)
        time.sleep(1)
        mode = "run"
        #  goes back to weighing mode
        show_oz = True
        label = "oz"
        display.print("   %0.1f %s" % (the_avg, label))
        run_mode = True
        blue_btn_pressed = True
```

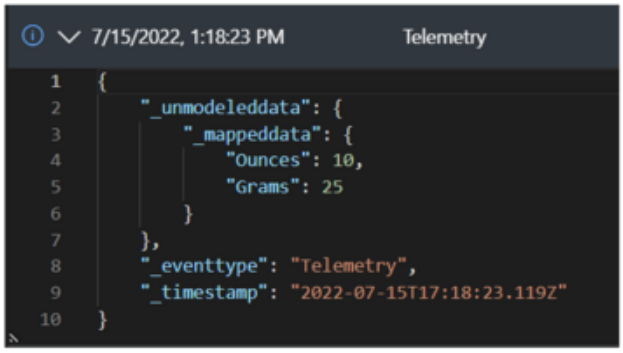
Edit the Data Template

The device's template organizes and categorizes the incoming data so that it can be logged properly.

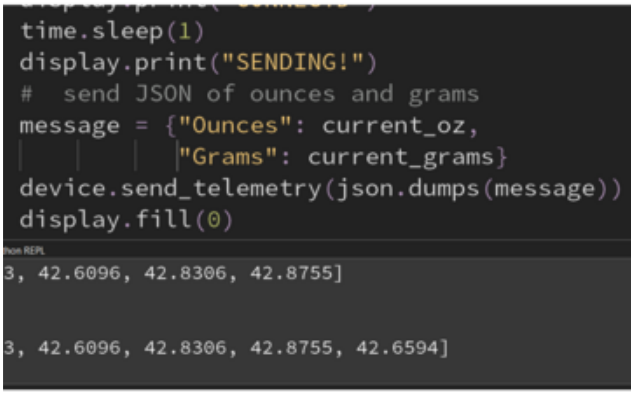


At the top of your device's page, click on **Manage template -> Auto-create template**.





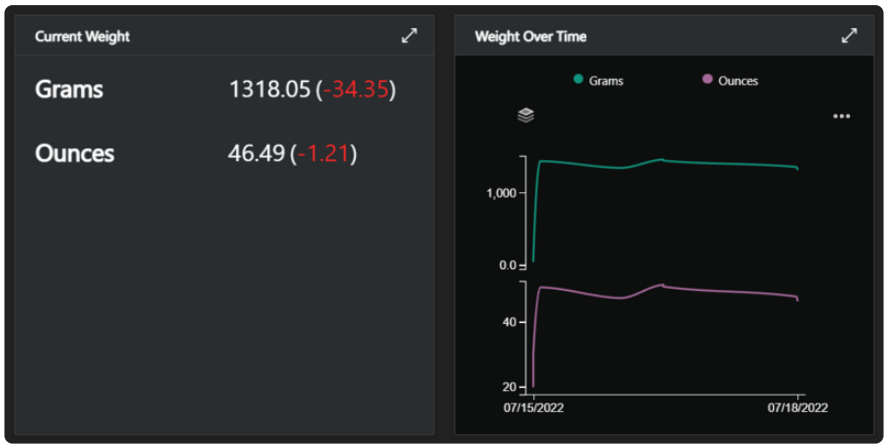
If you've already logged data from the QT Py ESP32-S2, then the template will be auto-filled with the data that you sent to Microsoft Azure which was previously categorized as unmodeled data.



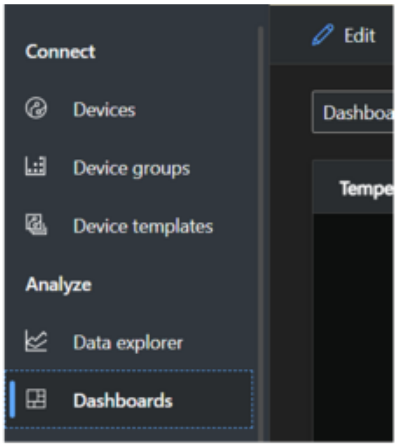
You can also add the data types that you want to collect by clicking Add capability. The display name is how the data will be categorized on Azure. The name has to match the string in the JSON message that is sent from the CircuitPython code. Capability type needs to match the type of data that is going to be received. If you are receiving sensor data, then you will select Telemetry.

Make sure to click **Save** when you're done editing your template.

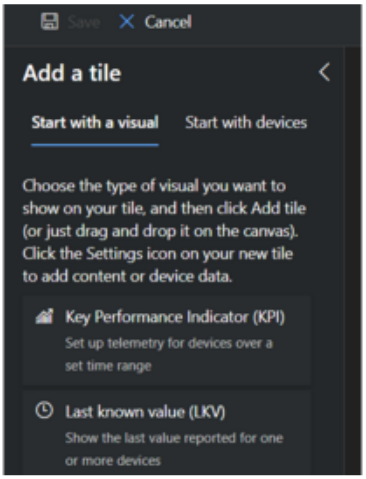
Create a Dashboard



You can create visual dashboards to display your data for easy viewing.

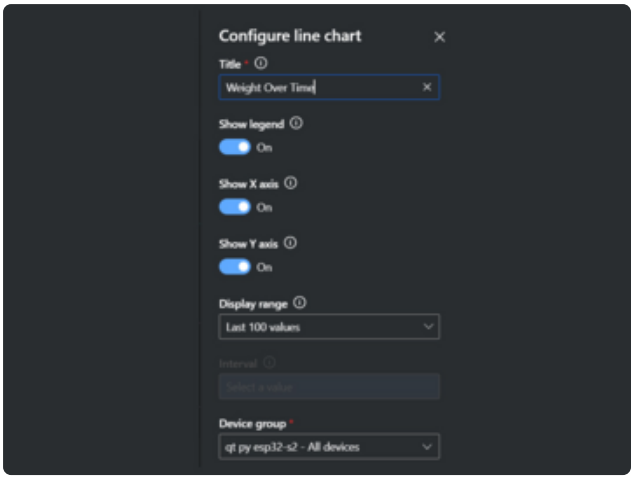


Navigate to the Dashboards page and click **Edit**.

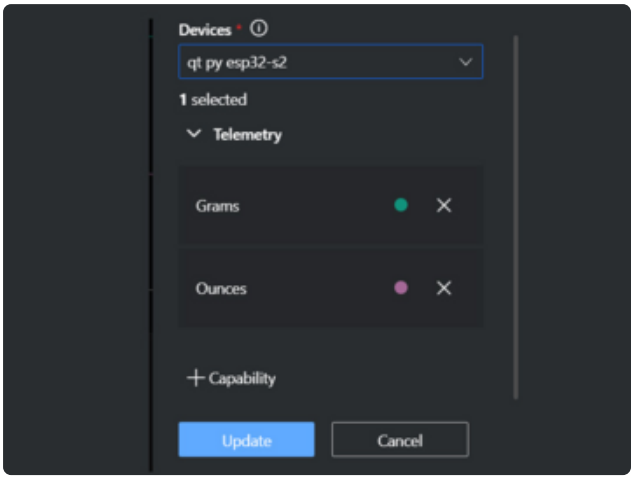


In edit mode, a tile menu opens on the left. You can drag and drop different tiles into the main dashboard space. There's a variety of graph and display types to choose from.

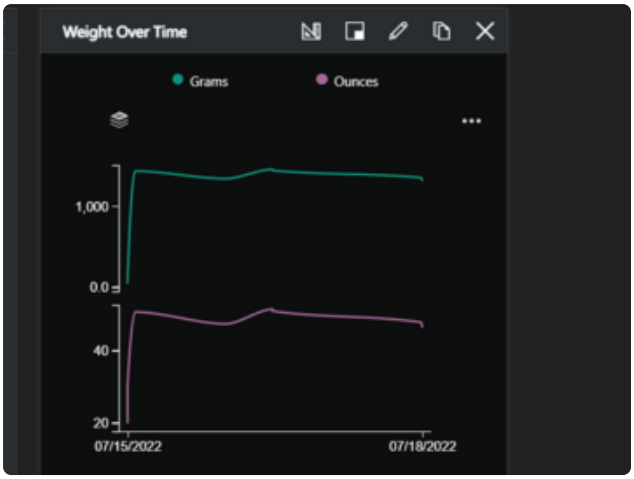
The tile will be empty when you first bring it to the dashboard. You can click on the **pencil icon** to edit the tile to display your data.



The title will be displayed above the tile. You can configure the chart's axes and legend.

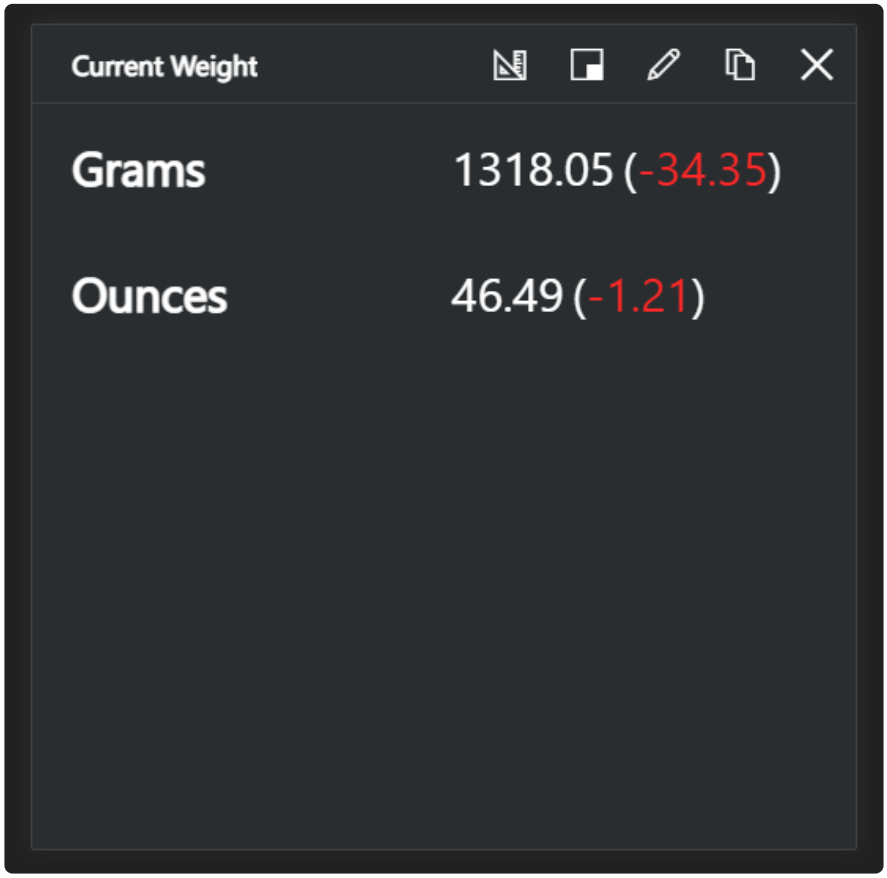


To bring the data into the tile, you'll **select your device from the dropdown**. Then, **add Capability** to bring in different data streams.

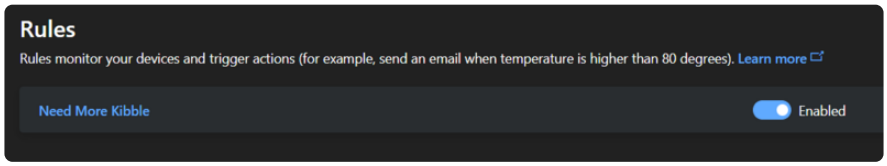


Click **Update** and you should see your data populate the tile.

You can add as many tiles as you want to fully customize your dashboard. The food scale dashboard utilizes line graph and last known value (LKV) tiles to show the weight over time, the current weight and the change in weight since the last reading.

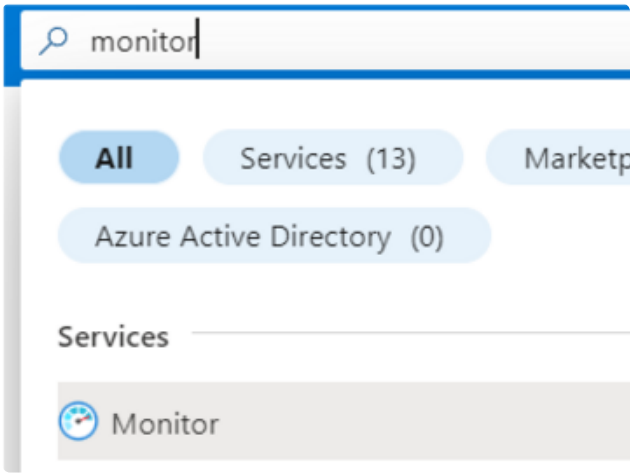


Create a Text Alert

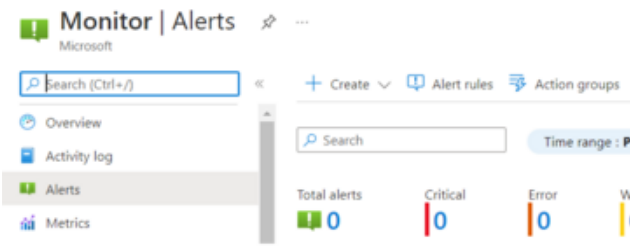


You can create Rules to alert you about certain data thresholds. For this project, you'll setup a text alert using an action group in the Azure Portal.

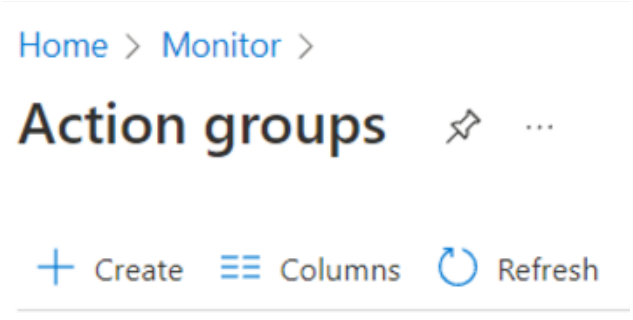
Create an Action Group



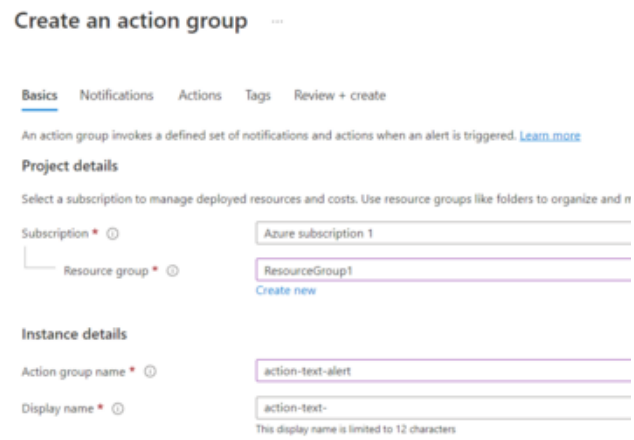
In the Azure Portal, search "monitor" and click on the **Monitor icon**.



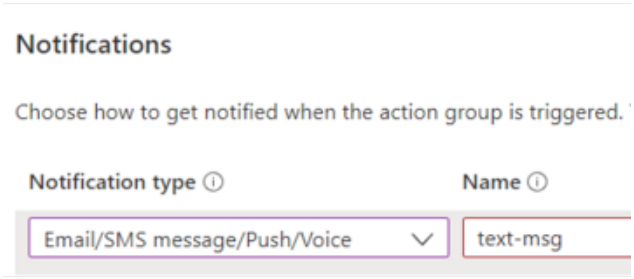
On the Monitor page, click on **Alerts** in the left column and then **Action Groups**.



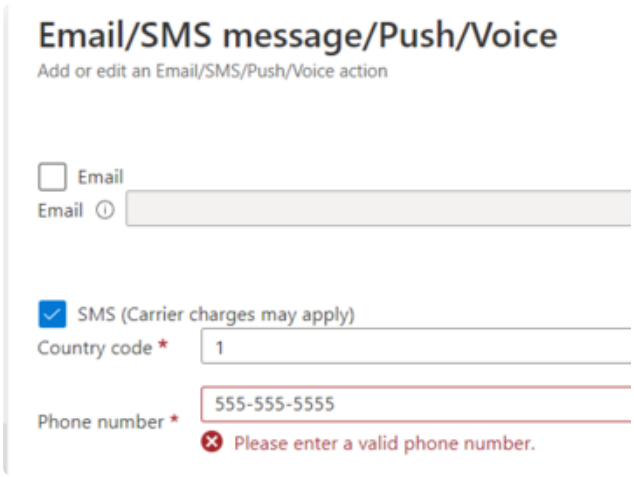
On the Action groups page, click on **Create** to create a new action group.



On the basics page, you'll link your subscription and resource group. Then, you'll name the action.



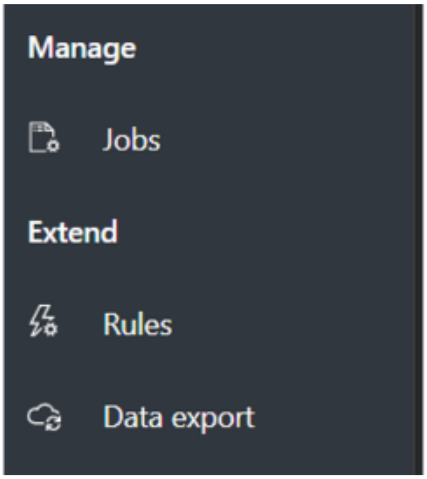
Under Notifications, you'll choose the notification type and name the notification. Choose Email/SMS message/Push/Voice for text message.



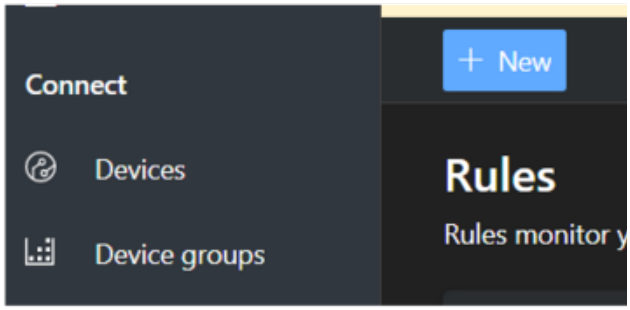
A menu for directing the notification will pop-up on the right of the screen. Check off **SMS** for a text message and enter the phone number that you want to receive the text messages.

Afterwards, you can click **"Review and create"** to complete the setup.

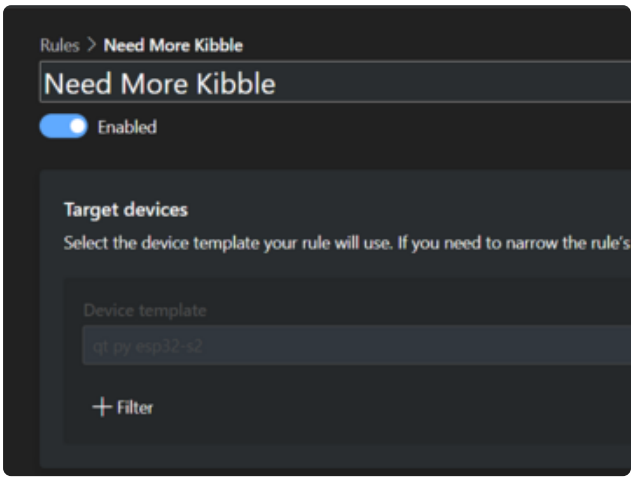
Create a Rule



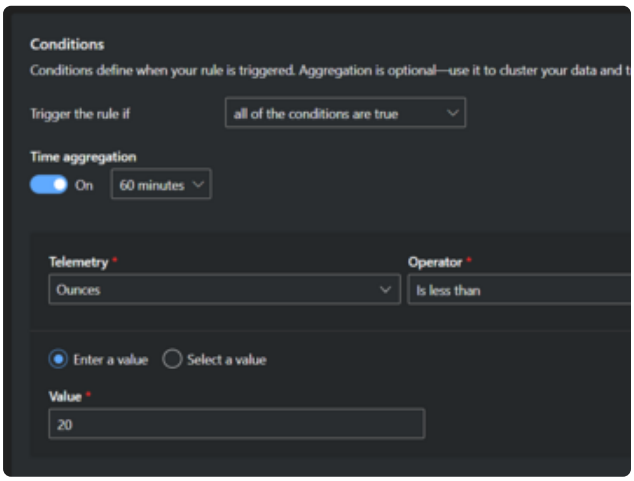
Navigate to the **Rules** page on the side menu bar.



Click on **New** to create a new rule.

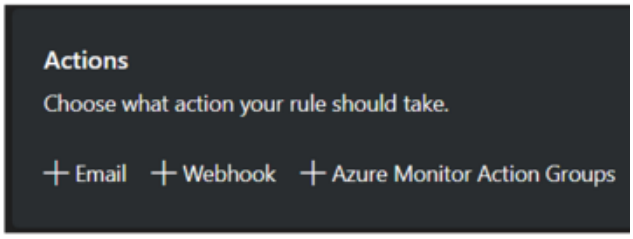


First, you'll need to name the alert and **toggle the Enable option** to turn it on. Then you can **select the device** to receive alerts for.

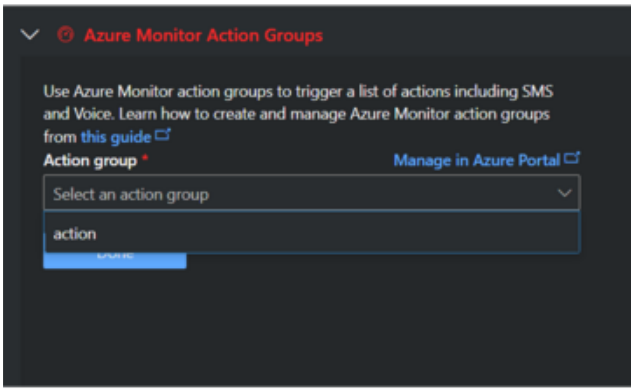


You'll setup conditions for the rule. You can have the rule be triggered if all or some of the conditions are true. The Time aggregation option can limit the number of alerts you receive. For example, if you select 60 minutes then you will only be alerted once per hour if a condition is true.

Under Telemetry, you'll select the data feed to monitor, as well as an operator; such as less than, equal to, etc. Under Value, you'll enter the value to watch for.



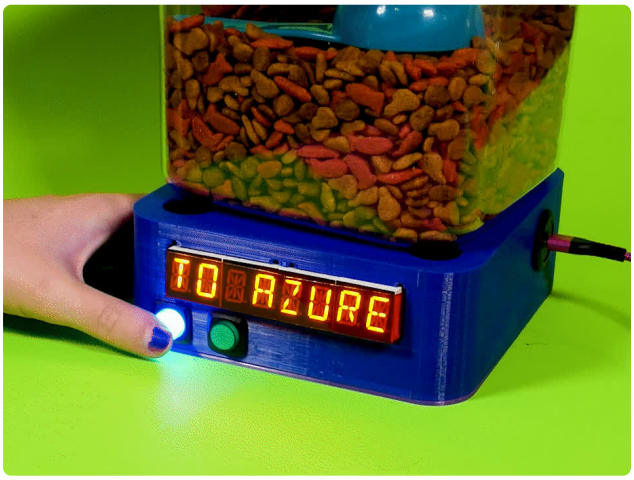
Under Actions, you'll select Azure Monitor Action Groups.



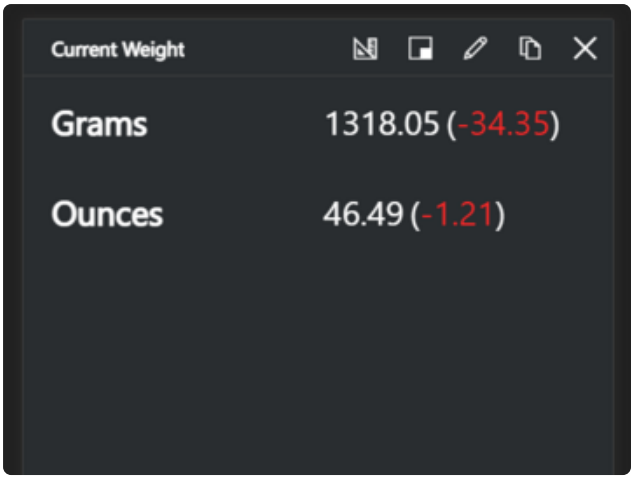
Select the Action Group that you setup earlier from the dropdown menu. When you're finished, **save your Rule** and you're ready to receive text alerts.

Usage

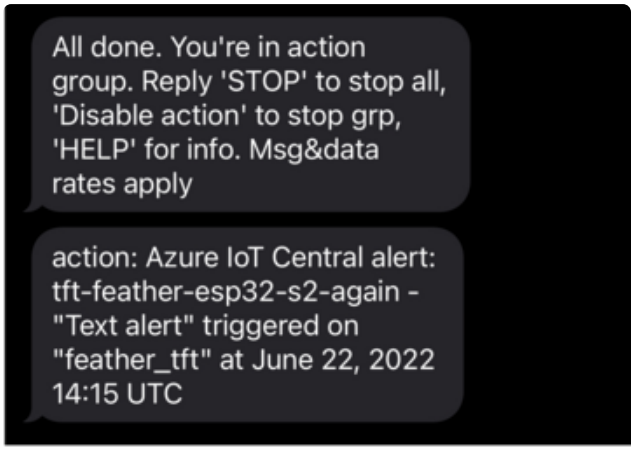
After calibrating your NAU7802 (https://adafru.it/10e4), place your food container on top and begin tracking your pet's food consumption over time.



Send data to Azure by selecting the mode on the alphanumeric displays.



View the data over time to estimate when you need to refill the container. You can also see on average how much is being eaten per meal with the last known value chart.



Receive text alerts when the scale drops below a threshold value.