Raspberry Pi Azure IoT Hub Dashboard with CircuitPython

Created by Liz Clark


Last updated on 2023-08-29 04:49:32 PM EDT
# Table of Contents

**Overview** 3
  - Prerequisite Guides
  - Parts

**Create a Microsoft Azure Account** 7

**Create an IoT Hub** 9

**Add IoT Hub Devices** 12

**Setup the IoT Hub Devices with CircuitPython** 14

**ESP32-S2 TFT Feather Environment Monitor Setup** 15
  - Wiring the ESP32-S2 TFT Feather Environment Monitor
  - 3D Printed Mount
  - Coding the ESP32-S2 TFT Feather Environment Monitor
  - Upload the Code and Libraries to the ESP32-S2 TFT Feather
  - secrets.py

**QT Py ESP32-S2 CO2 Monitor Setup** 20
  - Wiring the QT Py ESP32-S2 CO2 Monitor
  - 3D Printed Case
  - Coding the QT Py ESP32-S2 CO2 Monitor
  - Upload the Code and Libraries to the QT Py ESP32-S2
  - secrets.py

**Prepare Your Raspberry Pi** 24
  - Install Azure CLI
  - Hide the Taskbar

**Code the Raspberry Pi IoT Dashboard** 26
  - Install the Required Libraries
  - Download the Code, Image and Font Files
  - Add Your IoT Hub Information
  - How the Code Works

**Using the IoT Hub's Built-in Endpoint** 34

**Using the Azure REST API** 35
  - Bearer Token

**Using an Azure Device Twin** 36

**Setup the Code to Run on Boot** 37

**Usage** 38
  - Going Further
Overview

In this project, you'll visualize all of your Azure IoT Hub devices' telemetry data on a Raspberry Pi-powered display. The Raspberry Pi runs Python code combining Azure's Python SDK and Blinka, the CircuitPython compatibility library for single-board computers running Linux. The code shows three different ways to retrieve data from IoT Hub with Python: a built-in endpoint, the Azure REST API and an Azure Device Twin.

IoT Hub is one of the two IoT options from Azure. It allows for two-way communication, which makes this project possible. It can be more complicated to setup and configure than IoT Central, but this guide will walk you through everything you need to get up and running.

The Raspberry Pi dashboard uses the Blinka_Displayio_PyGameDisplay library to utilize displayio elements over HDMI rather than a discrete display over I2C or SPI.
The dashboard updates every time a device sends telemetry data to IoT Hub. It also logs the last device to check-in with a timestamp so you always know the connection status of your devices.

By using a dashboard, you have a centralized place where you can easily visualize your various devices' status at a glance. That way you can place your devices exactly where you want them around your space.

Prerequisite Guides

- Getting Started with Microsoft Azure and CircuitPython
- CircuitPython Libraries on Linux and Raspberry Pi
- Adafruit ESP32-S2 TFT Feather
- Adafruit QT Py ESP32-S2
- Adafruit SCD-40 and SCD-41
- Adafruit BME680
Parts

**Raspberry Pi 4 Model B - 4 GB RAM**
The Raspberry Pi 4 Model B is the newest Raspberry Pi computer made, and the Pi Foundation knows you can always make a good thing better! And what could make the Pi 4 better...
https://www.adafruit.com/product/4296

**Adafruit ESP32-S2 TFT Feather - 4MB Flash, 2MB PSRAM, STEMMA QT**
We've got a new machine here at Adafruit, it can uncover your deepest desires. Don't believe me? I'll turn it on right now to prove it to you! What, you want unlimited...
https://www.adafruit.com/product/5300

**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
1 x Lithium Ion Polymer Battery - 3.7v 2500mAh
2500mAh battery
https://www.adafruit.com/product/328

1 x Official Raspberry Pi Power Supply 5.1V 3A with USB C
USB C power supply
https://www.adafruit.com/product/4298

1 x Official Raspberry Pi Micro HDMI to HDMI Cable - 1 meter long
Micro HDMI to HDMI Cable
https://www.adafruit.com/product/4302

1 x Pink and Purple Woven USB A to USB C Cable -

Adafruit SCD-40 - True CO2, Temperature and Humidity Sensor
Take a deep breath in...now slowly breathe out. Mmm isn't it wonderful? All that air around us, which we bring into our lungs, extracts oxygen from and then breathes out carbon...
https://www.adafruit.com/product/5187

Adafruit BME688 - Temperature, Humidity, Pressure and Gas Sensor
The long-awaited BME688 from Bosch gives you all the environmental sensing you want in one small package. This little sensor...
https://www.adafruit.com/product/5046

STEMMA QT / Qwiic JST SH 4-Pin Cable - 50mm Long
This 4-wire cable is 50mm / 1.9” long and fitted with JST SH female 4-pin connectors on both ends. Compared with the chunkier JST PH these are 1mm pitch instead of 2mm, but...
https://www.adafruit.com/product/4399
Create a Microsoft Azure Account

Before you can get started with Microsoft Azure, you need to create an account. There are multiple tiers of accounts, including a free version.

Go to the Microsoft Azure website (https://azure.microsoft.com/) and click on Try Azure for Free.

Click on Start Free to proceed in creating a free account.
You can create a new account, sign-in with your Microsoft account or sign-in with your GitHub account.

After signing in, you'll be prompted to enter your information.
Even with the free account, you do need to enter a payment method. Nothing is charged unless you change settings later.

After entering all of your information, your Azure account setup will be complete. You can navigate to the Azure portal to get started with creating an application.

Create an IoT Hub
Navigate to your Azure portal at portal.azure.com(). In the search bar, search for "IoT Hub". On the IoT Hub page, click + Create.

On the IoT Hub setup page, select your subscription and resource group from the dropdown menu. Then, name your new IoT Hub in the IoT Hub name box.

Under Management, select your pricing tier. There are a variety of tiers with various features. There is also a free tier (F1) with basic features.
Then, click the Review + create tab. This will validate your new IoT Hub’s settings. Once your IoT Hub is validated, you can click the blue Create button to finish setup.

After creating your IoT Hub, it will appear on the IoT Hub default directory homepage. You can click your IoT Hub's name to go to its overview page.
On the overview page, you can view basic IoT Hub information, like your Subscription ID, and general usage data, such as the number of messages that have been sent.

Add IoT Hub Devices

On your IoT Hub's page, select Devices under Device management. Then click + Add Device to begin adding a new device to your IoT Hub.
Enter your device's name into the Device ID field. Then, click Save to create your device. Your device will show-up in the Device ID list on the Devices page.

If you click on your device's name, you'll be brought to that device's page. There you can access its security keys and manage settings. The Primary Connection String is used by the CircuitPython IoT Hub library to connect to Azure.
Setup the IoT Hub Devices with CircuitPython

For this project, there are two example devices that you can build to send data to your IoT hub. The first is based on the project for the Getting Started with Microsoft Azure and CircuitPython guide. It uses a ESP32-S2 TFT Feather with a STEMMA BME688 sensor to monitor ambient temperature, humidity and air pressure data. The only difference in the code is that instead of it being an IoT Central device, it is setup as an IoT Hub device.

The second device is a QT Py ESP32-S2 with a STEMMA SCD40 sensor for monitoring CO2 data.

The Raspberry Pi display is setup to show two instances of the environment monitor and one instance of the CO2 monitor.
ESP32-S2 TFT Feather Environment Monitor Setup

Wiring the ESP32-S2 TFT Feather Environment Monitor

Use a STEMMA QT cable to connect a STEMMA BME688 board to the ESP32-S2 TFT Feather.

3D Printed Mount

You can refer to the 3D Printing page in the Getting Started with Microsoft Azure and CircuitPython guide to print a mount for the Feather and BME688.

Coding the ESP32-S2 TFT Feather Environment Monitor

First, setup your Feather ESP32-S2 TFT with CircuitPython. Then, 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.

```python
import json
import time
import digitalio
import supervisor
import simpleio
import vectorio
import board
import terminalio
import rtc
import socketpool
import wifi
import displayio
import adafruit_ntp
```

©Adafruit Industries
from adafruit_display_text import bitmap_label, wrap_text_to_lines
from adafruit_bitmap_font import bitmap_font
from adafruit_azureiot import IoTHubDevice
import adafruit_bme680
from adafruit_lc709203f import LC709203F, PackSize

# 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("Connecting to WiFi!")

# ntp clock - update tz offset to your timezone
pool = socketpool.SocketPool(wifi.radio)
ntp = adafruit_ntp.NTP(pool, tz_offset=0)
rtc.RTC().datetime = ntp.datetime

if time.localtime().tm_year < 2022:
    print("Setting System Time in UTC")
    rtc.RTC().datetime = ntp.datetime
else:
    print("Year seems good, skipping set time.")

esp = None
pool = socketpool.SocketPool(wifi.radio)
# Create an IoT Hub device client and connect
device = IoTHubDevice(pool, esp, secrets["device_connection_string")]

print("Connecting to Azure IoT Hub...")

# Connect to IoT Central
device.connect()

print("Connected to Azure IoT Hub!")

cal = ntp.datetime
year = cal[0]
mon = cal[1]
day = cal[2]
hour = cal[3]
minute = cal[4]

i2c = board.I2C()  # uses board.SCL and board.SDA
# i2c = board.STEMMA_I2C()  # For using the built-in STEMMA QT connector on a microcontroller
bme680 = adafruit_bme680.Adafruit_BME680_I2C(i2c, debug=False)

# change this to match the location's pressure (hPa) at sea level
bme680.sea_level_pressure = 1013.25

temperature_offset = -5

# Create sensor object, using the board's default I2C bus.
i2c = board.I2C()  # uses board.SCL and board.SDA
# i2c = board.STEMMA_I2C()  # For using the built-in STEMMA QT connector on a microcontroller
battery_monitor = LC709203F(i2c)

# Update to match the mAh of your battery for more accurate readings.
# Can be MAH100, MAH200, MAH400, MAH500, MAH1000, MAH2000, MAH3000.
# Choose the closest match. Include "PackSize." before it, as shown.
battery_monitor.pack_size = PackSize.MAH2000
temp = int((bme680.temperature * 9/5) + (32 + temperature_offset))
humidity = int(bme680.relative_humidity)
pressure = int(bme680.pressure)
battery = battery_monitor.cell_percent

# setup boot button as input
button = digitalio.DigitalInOut(board.BUTTON)
button.switch_to_input(pull=digitalio.Pull.UP)

# display setup
display = board.DISPLAY

palette0 = displayio.Palette(2)
palette0[0] = 0x00FF00
palette0[1] = 0xFF0000

# load bitmap
bitmap = displayio.OnDiskBitmap("/bmeTFT.bmp")
tile_grid = displayio.TileGrid(bitmap, pixel_shader=bitmap.pixel_shader)
group = displayio.Group()
group.append(tile_grid)

# rectangle for battery life monitor
# vectorio allows for resizing in the loop
rect = vectorio.Rectangle(pixel_shader=palette0, width=22, height=10, x=12, y=116,
color_index = 0)
group.append(rect)

# bitmap font
font_file = "/roundedHeavy-26.bdf"
font = bitmap_font.load_font(font_file)

# text elements
temp_text = bitmap_label.Label(font, text="%.1f° F" % temp, x=20, y=80,
color=0xFFFFFF)
humid_text = bitmap_label.Label(font, text="%.1f %" % humidity, x=95, y=80,
color=0xFFFFFF)
press_text = bitmap_label.Label(font, text="%.2f" % pressure, x=170, y=80,
color=0xFFFFFF)
time_text = bitmap_label.Label(terminalio.FONT,
text="\n".join(wrap_text_to_lines
("Data sent on %s/%s/%s at %s:%s" % (mon,day,year,hour,minute), 20)),
x=125, y=105, color=0xFFFFFF)
group.append(temp_text)
group.append(humid_text)
group.append(press_text)
group.append(time_text)
display.show(group)

# clock to count down to sending data to Azure
azure_clock = 500
# clock to count down to updating TFT
feather_clock = 30
# button debounce state
button_pressed = False

while True:
    try:
        if button.value and button_pressed:
            button_pressed = False
        if not button.value and not button_pressed:
            print("getting msg")
            # pack message
            message = {"Temperature": temp,
                        "Humidity": humidity,
                        "Pressure": pressure,
                        "BatteryPercent": battery,
                        "FeatherConnected": 1}
            print("sending json")
            device.send_device_to_cloud_message(json.dumps(message))
            print("data sent")
# read BME sensor

temp = int((bme680.temperature * 9/5) + (32 + temperature_offset))
humidity = int(bme680.relative_humidity)
pressure = int(bme680.pressure)

# log battery %
battery = battery_monitor.cell_percent

# map range of battery charge to rectangle size on screen
battery_display = round(simpleio.map_range(battery, 0, 100, 0, 22))

# update rectangle to reflect battery charge
rect.width = int(battery_display)

# if below 20%, change rectangle color to red
if battery_monitor.cell_percent < 20:
    rect.color_index = 1

# when the azure clock runs out
if azure_clock > 500:
    print("getting ntp date/time")
    cal = ntp.datetime
    year = cal[0]
    mon = cal[1]
    day = cal[2]
    hour = cal[3]
    minute = cal[4]
    time.sleep(2)
    print("getting msg")
    # pack message
    message = {"Temperature": temp,
               "Humidity": humidity,
               "Pressure": pressure,
               "BatteryPercent": battery}
    print("sending json")
    device.send_device_to_cloud_message(json.dumps(message))
    print("data sent")
    clock_view = "%s:%s" % (hour, minute)
    if minute < 10:
        clock_view = "%s:0%s" % (hour, minute)
    print("updating time text")
    time_text.text="\n".join(wrap_text_to_lines
    ("Data sent on %s/%s/%s at %s" % (mon,day,year,clock_view), 20))
    # reset azure clock
    azure_clock = 0

# when the feather clock runs out
if feather_clock > 30:
    print("updating screen")
    temp_text.text = "%0.1f° F" % temp
    humid_text.text = "%0.1f %" % humidity
    press_text.text = "%0.2f" % pressure
    # reset feather clock
    feather_clock = 0

# if no clocks are running out
# increase counts by 1
else:
    feather_clock += 1
    azure_clock += 1

    # ping azure
    device.loop()

# if something disrupts the loop, reconnect
# pylint: disable=broad-except
except (ValueError, RuntimeError, OSError, ConnectionError) as e:
    print("Network error, reconnecting\n", str(e))
    time.sleep(60)
    supervisor.reload()
    continue

# delay
time.sleep(1)
Upload the Code and Libraries to the ESP32-S2 TFT Feather

After downloading the Project Bundle, plug your ESP32-S2 TFT Feather 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 ESP32-S2 TFT Feather's CIRCUITPY drive.

- lib folder
- roundedHeavy-26.bdf
- bmeTFT.bmp
- code.py

Your ESP32-S2 TFT Feather CIRCUITPY drive should look like this after copying the lib folder, the bmeTFT.bmp image file, the roundedHeavy-26.bdf bitmap font file, and 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:
You'll locate your Primary Connection String from your devices page in your IoT Hub. Make sure to refer to the Add IoT Hub Devices page in this guide to see the process for accessing the Primary Connection String.

QT Py ESP32-S2 CO2 Monitor Setup

Wiring the QT Py ESP32-S2 CO2 Monitor

Use a STEMMA QT cable to connect a STEMMA SCD40 board to the QT Py ESP32-S2.

3D Printed Case

You can refer to the 3D Printing page in theDisconnected CO2 Data Logger guide to print a case with mounting holes for the SCD40.

Coding the QT Py ESP32-S2 CO2 Monitor

First, setup your QT Py ESP32-S2 with CircuitPython. Then, 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.

```python
import time
import json
```
import digitalio
import supervisor
import board
import rtc
import socketpool
import wifi
import adafruit_ntp
from adafruit_azureiot import IoTHubDevice
import adafruit_scd4x

# 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!")

# ntp clock - update tz offset to your timezone
pool = socketpool.SocketPool(wifi.radio)
ntp = adafruit_ntp.NTP(pool, tz_offset=0)
rtc.RTC().datetime = ntp.datetime

if time.localtime().tm_year < 2022:
    print("Setting System Time in UTC")
    rtc.RTC().datetime = ntp.datetime
else:
    print("Year seems good, skipping set time.")

esp = None
pool = socketpool.SocketPool(wifi.radio)
# Create an IoT Hub device client and connect
device = IoTHubDevice(pool, esp, secrets["device_connection_string"])

print("Connecting to Azure IoT Hub...")

# Connect to IoT Central
device.connect()

print("Connected to Azure IoT Hub!")

# setup for I2C
i2c = board.STEMMA_I2C()
# setup for SCD40
scd4x = adafruit_scd4x.SCD4X(i2c)

# start measuring co2 with SCD40
scd4x.start_periodic_measurement()
co2 = scd4x.CO2

# setup boot button as input
button = digitalio.DigitalInOut(board.BUTTON)
button.switch_to_input(pull=digitalio.Pull.UP)

# clock to count down to sending data to Azure
azure_clock = 500

# button debounce state
button_pressed = False

while True:
    try:
        # button debouncing
        if button.value and button_pressed:
button_pressed = False
# if you press boot
if not button.value and not button_pressed:
    pack message
    message = "CO2": co2,
    "QT Connected": 1
    # send co2 measurement
    device.send_device_to_cloud_message(json.dumps(message))
    # measure co2
    co2 = scd4x.CO2
    # when the azure clock runs out
    if azure_clock > 500:
        print("getting msg")
        # pack message
        message = "{"CO2": co2,
        "QT Connected": 1}
        print("sending json")
        device.send_device_to_cloud_message(json.dumps(message))
        print(\"data sent\")
        # reset azure clock
        azure_clock = 0
        # if no clocks are running out
        # increase counts by 1
        else:
            azure_clock += 1
        # ping azure
        device.loop()
        # if something disrupts the loop, reconnect
        pylint: disable=broad-except
        except (ValueError, RuntimeError, OSError, ConnectionError) as e:
            print("Network error, reconnecting\n", str(e))
            time.sleep(10)
            supervisor.reload()
            continue
        # delay
        time.sleep(1)

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
- code.py

Your QT Py ESP32-S2 CIRCUITPY drive should look like this after copying the lib folder and 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:

```python
secrets = {
    'ssid': 'YOUR-SSID-HERE',
    'password': 'YOUR-SSID-PASSWORD-HERE',
    'device_connection_string': 'YOUR-DEVICE-CONNECTION-STRING-HERE'
}
```

You'll locate your Primary Connection String from your devices page in your IoT Hub. Make sure to refer to the [Add IoT Hub Devices](#) page in this guide to see the process for accessing the Primary Connection String.
Prepare Your Raspberry Pi

First, install the latest version of Raspberry Pi OS onto an SD card. You can refer to the [CircuitPython Libraries on Linux and Raspberry Pi](#) guide for more help setting it up.

Once you have everything set up, you will need to open a terminal and install Blinka. Refer to the [Installing CircuitPython Libraries on Raspberry Pi](#) page to quickly get up and running.

Install Azure CLI

Azure CLI is a command line tool that you can use on Linux computers, like the Raspberry Pi, to access various Azure functions and information. In this instance, you'll use Azure CLI to generate a Bearer token to authenticate while making an HTTP request in the code.py file.

To install Azure CLI, open a terminal window and enter the following command:

```
sudo apt-get install azure-cli
```

This will run through the Azure CLI installation process. After installation finishes, log into your Azure account with:

```
az login
```
This opens a browser and prompts you to log into your Microsoft Azure account.

After logging in, the browser will let you know that the login was successful and then navigate to the Azure CLI documentation page.
In the terminal, you'll see a printout of your Azure subscription information. This information is sensitive so you may want to close out of the terminal after viewing.

Hide the Taskbar

To allow the dashboard display window to take up the full Raspberry Pi display, you'll want to hide the top taskbar when it isn't in use.

Right-click on the taskbar and select Panel Preferences. In the Panel Preferences window, go to the Advanced tab and check off "Minimize panel when not in use". Then, click Close.

Code the Raspberry Pi IoT Dashboard

Install the Required Libraries

You will need to have a few libraries installed before the script will run on your Raspberry Pi.

Install the required CircuitPython libraries with the terminal:

```
pip3 install adafruit-circuitpython-bitmap-font
pip3 install adafruit-circuitpython-display-shapes
pip3 install adafruit-circuitpython-display-text
pip3 install adafruit-circuitpython-imageload
pip3 install adafruit-blinka-displayio
pip3 install blinka-displayio-pygamemdisplay
```

Then, install the required Python libraries with the terminal:
pip3 install subprocess
pip3 install datetime
pip3 install pygame
pip3 install recordclass
pip3 install requests

Finally, install the required Azure IoT Python libraries with the terminal:

```
pip3 install azure-iot-hub
pip3 install azure-event-hub
```

### Download the Code, Image and Font Files

Once you've finished setting up your Raspberry Pi with Blinka and the library dependencies, you can access the Python code, bitmap image file and two font files 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.

```python
#!/usr/bin/env python3
# pylint: disable=unused-import
import json
from subprocess import Popen, PIPE
from datetime import datetime, timezone
import requests
import displayio
import adafruit_imageload
from adafruit_display_text import label
from adafruit_bitmap_font import bitmap_font
from adafruit_display_shapes.circle import Circle
from azure.iot.hub import IoTHubRegistryManager
from azure.iot.hub.models import Twin, TwinProperties, QuerySpecification, QueryResult
from azure.eventhub import EventHubConsumerClient, TransportType
from blinka_displayio_pygamedisplay import PyGameDisplay

# ***
# UPDATE THESE VARIABLES WITH YOUR IOT HUB INFORMATION

# event hub compatible end point from the built-in endpoints page
event_connection = "YOUR-EVENT-HUB-CONNECTION-STRING-HERE"
# Primary Connection String with registry read and service connect rights
# managed on shared access polices page
status_connection = "YOUR-STATUS-CONNECTION-STRING-HERE"

# iot hub subscription ID
# found on the overview of your iot hub
subscription_id = "YOUR-SUBSCRIPTION-ID-HERE"

# device id's (device names in iot hub)
```
qt_py = "YOUR-QT-PY-DEVICE-HERE"
tft_feather = "YOUR-TFT-FEATHER-DEVICE-HERE"
s3_feather = "YOUR-S3-FEATHER-DEVICE-HERE"

# ***
# array of devices
devices = [qt_py, tft_feather, s3_feather]

# create display
display = PyGameDisplay(width=1920, height=1080)

# open bitmap background with imageload
bitmap, palette = adafruit_imageload.load("/home/pi/azure_pi/piBG_0.bmp", bitmap=displayio.Bitmap, palette=displayio.Palette)
background = displayio.TileGrid(bitmap, pixel_shader=palette)

# load font files
font_file = "/home/pi/azure_pi/OstrichSans-Heavy-88.bdf"
smallFont_file = "/home/pi/azure_pi/OstrichSans-Heavy-60.bdf"
font = bitmap_font.load_font(font_file)
small_font = bitmap_font.load_font(smallFont_file)

# text objects
# co2 data
co2_text = label.Label(font, text = "500", color = 0xFFFFFF, x = 611, y = 800)

# living room data
lr_temp = label.Label(font, text = "72°F", color = 0xFFFFFF, x = 90, y = 424)
lr_humid = label.Label(font, text = "52%", color = 0xFFFFFF, x = 405, y = 424)
lr_press = label.Label(font, text = "1005", color = 0xFFFFFF, x = 715, y = 424)

# bedroom data
bd_temp = label.Label(font, text = "72°F", color = 0xFFFFFF, x = 1050, y = 424)
bd_humid = label.Label(font, text = "52%", color = 0xFFFFFF, x = 1365, y = 424)
bd_press = label.Label(font, text = "1005", color = 0xFFFFFF, x = 1675, y = 424)

# cost, timestamp and last device check-in text
timestamp_text = label.Label(small_font, text = "00:00AM 00/00/00", color = 0xFFFFFF, x = 1216, y = 886)
lastDevice_text = label.Label(small_font, text = "device", color = 0xFFFFFF, x = 1450, y = 805)
cost_text = label.Label(small_font, text = "$0.00", color = 0xFFFFFF, x = 1511, y = 734)

# status circles, defaults to white
qt_status = Circle(908, 695, 22, fill=0xFFFFFF)
lr_status = Circle(908, 52, 22, fill=0xFFFFFF)
bd_status = Circle(1869, 52, 22, fill=0xFFFFFF)

# array of status circles
status_circles = [qt_status, lr_status, bd_status]

# array of display objects
display_objects = [background, co2_text, lr_temp, lr_humid, lr_press, bd_temp, bd_humid, bd_press, timestamp_text, lastDevice_text, cost_text, qt_status, lr_status, bd_status]

# display group
main_group = displayio.Group()

# add all display objects from array to the display group
for x in display_objects:
    main_group.append(x)
display.show(main_group)
# convert UTC time to local timezone
def utc_to_local(utc_dt):
    return utc_dt.replace(tzinfo=timezone.utc).astimezone(tz=None)

# function for an incoming event from one of the devices
# pylint: disable=too-many-locals
def on_event_batch(partition_context, events):
    # gets bearer token from azure cli in terminal
    cost_json_index = 0
    output = Popen(['az', 'account', 'get-access-token'], stdout=PIPE)
    # parses the output into a json and grabs the bearer token
    bear = json.loads(output.communicate()[0].decode('utf-8').strip())
    # creates headers to include with GET HTTP request for cost data
    headers = {'Authorization': 'Bearer ' + bear['accessToken'], 'Content-Type': 'application/json'}
    # updates request URL with your subscription ID
    url = "https://management.azure.com/subscriptions/{}/providers/Microsoft.Consumption/usageDetails?api-version=2021-10-01".format(subscription_id)
    # makes HTTP request
    response = requests.get(url, headers=headers)
    # packs the HTTP response into a JSON
    feed = response.json()
    # grabs the cost per day from the JSON feed
    cost = feed['value'][cost_json_index]['properties']['costInBillingCurrency']
    # the JSON index for cost can move depending on the data recieved
    # adjusts index until a value is recieved
    while cost == 0:
        cost_json_index += 1
        cost = feed['value'][cost_json_index]['properties']['costInBillingCurrency']
    # iterates through incoming events
    for event in events:
        # gets timestamp for event
        clock = utc_to_local(event.enqueued_time)
        # calculates the month to date cost based on the date
        month_cost = cost*((float(clock.strftime('%d'))) - 1))
        # updates cost text
        cost_text.text = "$%.2f" % month_cost
        # updates timestamp text
        timestamp_text.text = clock.strftime("%I:%M%p %m/%d/%y")
        # gets the incoming event as a JSON feed
        telemetry = event.body_as_json()
        # grabs the device ID from the JSON
        device = event.system_properties[b'iothub-connection-device-id']
        # converts the device ID to a string
        string_device = device.decode("utf-8")
        # updates last device text
        lastDevice_text.text = string_device
        # if the device is the qt_py
        if string_device == qt_py:
            # update co2 text
            co2_text.text = str(telemetry['CO2'])
        # if the device is the tft feather
        if string_device == tft_feather:
            # update living room sensor text
            lr_temp.text = "%s" % str(telemetry['Temperature'])
            lr_humid.text = "%s" % str(telemetry['Humidity'])
            lr_press.text = str(telemetry['Pressure'])
        # if the device is the s3 feather
        if string_device == s3_feather:
            # update bedroom sensor text
            bd_temp.text = "%s" % str(telemetry['Temperature'])
            bd_humid.text = "%s" % str(telemetry['Humidity'])
            bd_press.text = str(telemetry['Pressure'])
        # iterate through the status circles
        for d in range(0, 3):
            # get connection status of all 3 devices using device twin
twin = status_client.get_twin(devices[d])
# if a device is connected, make the status circle green
if twin.connection_state == 'Connected':
    status_circles[d].fill = 0x00FF00
# if a device is disconnected, make the status circle red
if twin.connection_state == 'Disconnected':
    status_circles[d].fill = 0xFF0000
# update the read partition from event hub
partition_context.update_checkpoint()
# in case of error with an incoming event
def on_error(partition_context, error):
    if partition_context:
        print("An exception: {} occurred during receiving from Partition:
        {}.".format(
            partition_context.partition_id,
            error
        ))
    else:
        print("An exception: {} occurred during the load balance
        process.".format(error))
# connect to event hub
client = EventHubConsumerClient.from_connection_string(
    conn_str = event_connection,
    consumer_group = "$default",
)
# connect to device twin
status_client = IoTHubRegistryManager(status_connection)
# while the display is active
while display.running:
    try:
        # receive incoming events
        client.receive_batch(
            on_event_batch = on_event_batch,
            on_error = on_error
        )
    except KeyboardInterrupt:
        print("Receiving has ")

After downloading the Project Bundle, move the folder to your /home/pi directory. Then, unzip the folder. Your Raspberry Pi should have the following files in the /home/pi/azure_pi folder:

code.py
piBG_0.bmp
OstrichSans-Heavy-88.bdf
OstrichSans-Heavy-60.bdf
Add Your IoT Hub Information

At the top of the code.py file, you'll see six variables that need to be updated with your IoT hub information.

```
# UPDATE THESE VARIABLES WITH YOUR IOT HUB INFORMATION

# event hub compatible end point from the built-in endpoints page
event_connection = "YOUR-EVENT-HUB-CONNECTION-STRING-HERE"

# Primary Connection String with registry read and service connect rights
# managed on shared access polices page
status_connection = "YOUR-STATUS-CONNECTION-STRING-HERE"

# iot hub subscription ID
# found on the overview of your iot hub
# format: #####-#####-############
subscription_id = "YOUR-SUBSCRIPTION-ID-HERE"

# device id's (device names in iot hub)
qt_py = "YOUR-QT-PY-DEVICE-HERE"
tft_feather = "YOUR-TFT-FEATHER-DEVICE-HERE"
s3_feather = "YOUR-S3-FEATHER-DEVICE-HERE"
```

`event_connection` is your Event Hub compatible endpoint. It is located on the Built-in endpoints page of your IoT Hub.
status_connection is the Primary connection string from a Shared Access Policy (SAP) that has Registry Read and Service Connect permissions.

You can setup this SAP by going to the Shared access policies page of your IoT Hub and then clicking on + Add shared access policy. Then, name the SAP in the Access policy name box and check off Registry Read and Service Connect. Click Add to create the SAP.

You access the Primary connection string by clicking on the SAP's name to view the security keys. Then, copy and paste the Primary connection string into the code.py file as `status_connection`.
subscription_id is located at the top of your IoT Hub’s Overview page.

Finally, your Device IDs are the plain text names of your devices setup in your IoT hub. These can be found on the Devices page in your IoT hub.

How the Code Works

The Blinka_Displayio_PyGameDisplay() library lets you use CircuitPython displayio elements over HDMI on the Raspberry Pi’s main display. There are text elements that update with the incoming device telemetry from IoT Hub.

```python
# living room data
lr_temp = label.Label(font, text = "72°F", color = 0xFFFFFF, x = 90, y = 424)
lr_humid = label.Label(font, text = "52%", color = 0xFFFFFF, x = 405, y = 424)
lr_press = label.Label(font, text = "1005", color = 0xFFFFFF, x = 715, y = 424)
...

# if the device is the tft feather
if string_device == tft_feather:
    # update living room sensor text
    lr_temp.text = "%s°F" % str(telemetry['Temperature'])
lr_humid.text = "%s%" % str(telemetry['Humidity'])
lr_press.text = str(telemetry['Pressure'])
```

The code is accessing your IoT Hub information with three different methods: built-in endpoints, Azure REST API and device twin. The following pages will detail how each of these methods are being used.
Using the IoT Hub's Built-in Endpoint

Each IoT Hub has a built-in endpoint to receive incoming data. By default, all device telemetry is received via this endpoint and it can be accessed by Azure's Event Hub library. This is the method being used in code.py to get the devices' incoming telemetry from their sensors.

```python
client = EventHubConsumerClient.from_connection_string(
    conn_str = event_connection,
    consumer_group = "$default",
)
```

In the loop, `receive_batch()` is called, which listens to incoming events.

```python
try:
    # recieve incoming events
    client.receive_batch(
        on_event_batch = on_event_batch,
        on_error = on_error
    )
```

The `on_event_batch()` function iterates through the incoming events. From this incoming feed, the code gets the timestamp with `event.enqueued_time` and encodes the feed as a JSON to grab the device name from the `event`.

A series of `if` statements checks to see which device is sending telemetry data in the `event` and updates the text objects with the telemetry data.

```python
# iterates through incoming events
for event in events:
    # gets timestamp for event
    clock = utc_to_local(event.enqueued_time)
    ...
    # gets the incoming event as a JSON feed
    telemetry = event.body_as_json()
    # grabs the device ID from the JSON
    device = event.system_properties[b'iothub-connection-device-id']
    # converts the device ID to a string
    string_device = device.decode("utf-8")
    # updates last device text
    lastDevice_text.text = string_device
    # if the device is the qt_py
    if string_device == qt_py:
        # update co2 text
        co2_text.text = str(telemetry['CO2'])
    # if the device is the tft feather
    if string_device == tft_feather:
        ...```
Using the Azure REST API

The Azure REST API allows you to use the Python requests library to get information from IoT Hub in the form of an HTTP request. An HTTP request is the source of the total cost information for the dashboard.

Bearer Token

To make a GET HTTP request using the Azure REST API, you need a Bearer token to be included in the header for the request. The Bearer token is only viable for one hour. In order to always be sure you have a valid token, the Azure CLI is run using the Popen() function from the subprocess Python library.

The subprocess library allows for terminal commands to be run in the background. The output from the terminal can be brought into the Python script. In this case, az account get-access-token is run and its output is converted to a JSON feed. The 'accessToken' JSON item contains the Bearer token and is added to headers.

```
output = Popen(['az', 'account', 'get-access-token'], stdout=PIPE)
# parses the output into a json and grabs the bearer token
bear = json.loads(output.communicate()[0].decode('utf-8').strip())
# creates headers to include with GET HTTP request for cost data
headers = {'Authorization': 'Bearer ' + bear['accessToken'], 'Content-Type': 'application/json'}
```

The URL request needs your subscription ID added to it. This is accomplished using .format() and inserting {} in the required location of the URL for the subscription ID. The url and headers are packed into a requests.get() function that is returned as a JSON feed with response.json(). cost holds the 'costInBillingCurrency' JSON item that has the daily cost for your cloud services.

```
# updates request URL with your subscription ID
url = "https://management.azure.com/subscriptions/{}/providers/Microsoft.Consumption/usageDetails?api-version=2021-10-01".format(subscription_id)
# makes HTTP request
response = requests.get(url, headers=headers)
# packs the HTTP response into a JSON
```
# grabs the cost per day from the JSON feed
```python
cost = feed['value'][cost_json_index]['properties']['costInBillingCurrency']
```

## Using an Azure Device Twin

An **Azure device twin** is a JSON document that stores all of the information about a device. Each device in your IoT Hub has a device twin that you can access. In code.py, device twins are used to check the connection status of each device. The connection status is represented by circles in the top right-hand corners of each device's telemetry box on the Raspberry Pi's display.

IoTHubRegistryManager() is used to authenticate access to the IoT Hub's device twins.

```python
# connect to device twin
status_client = IoTHubRegistryManager(status_connection)
```

A device twin is accessed for each of the three devices using a **for** statement. If the device is connected, then its circle is updated to be green. If the device is disconnected, then its circle is updated to be red.

```python
# iterate through the status circles
for d in range(0, 3):
    # get connection status of all 3 devices using device twin
twin = status_client.get_twin(devices[d])
    # if a device is connected, make the status circle green
    if twin.connection_state == 'Connected':
        status_circles[d].fill = 0x00FF00
    # if a device is disconnected, make the status circle red
```
Setup the Code to Run on Boot

To run the IoT Hub Dashboard when the Raspberry Pi boots up, you need to make the code.py file an executable. To do this, add a shebang to the first line of code.py, above the SPDX license.

```bash
#!/usr/bin/env python3
# SPDX-FileCopyrightText: 2022 Liz Clark for Adafruit Industries
# # SPDX-License-Identifier: MIT
```

Then, in a terminal, change directories to the /azure_pi folder and mark the code.py file as an executable using `chmod`.

```bash
cd /home/pi/azure_pi
chmod a+x code.py
```

The code.py file needs both the graphical interface and a network connection to properly run. As a result, that needs to be considered when deciding on a method to have it run on boot. To make sure the GUI and network connections are ready before trying to run code.py, a .desktop file is going to be used in the `autostart` system.

To create this file, in a terminal create a directory called /autostart in the /.config folder.

```bash
mkdir /home/pi/.config/autostart
```
Then, create the .desktop file using the `nano` text editor. This file will be called `azure-pi.desktop`.

```
nano /home/pi/.config/autostart/azure-pi.desktop
```

In the text editor, enter the following text to run code.py.

```
[Desktop Entry]
Type=Application
Name=Azure-Pi
Exec=/usr/bin/python /home/pi/azure_pi/code.py
```

Exit and save the new `azure-pi.desktop` file. Then, try rebooting your Raspberry Pi. After loading the desktop, you should see the code.py PyGameDisplay window begin running.

**Usage**

Boot up your Raspberry Pi to run the IoT Hub Dashboard code. Then, power-up your IoT Hub devices. They will begin sending their telemetry to IoT Hub and that telemetry will populate the Raspberry Pi's dashboard.

Because the bitmap background is sized for a 1080p screen, it does take a few seconds longer than you may expect to load when the code starts up.
The dashboard will display all of the received telemetry from the IoT Hub devices, as well as connection status, the name of the last device to send telemetry, the timestamp that telemetry was received and the current month's cloud cost.

The current month's cost is displayed on the dashboard to easily track usage.
A device's connection status is represented with a green or red circle.

The last device to send telemetry data, along with the timestamp, is logged on the dashboard.

Going Further

You can create a custom bitmap background to better suite your devices and sensors that you want to monitor. You could also add hardware to the Raspberry Pi, like buttons, to make the dashboard have menus or modes.