Guide Contents

Guide Contents .............................................. 2
Overview ....................................................... 3
Parts .............................................................. 4
The Circuit ...................................................... 6
  Power .......................................................... 6
  BMP280 ....................................................... 6
  GPS ............................................................. 6
  Air Quality Sensor .......................................... 6
  AirLift .......................................................... 7
Construction ................................................... 8
  Preparing the Case ......................................... 8
  Wiring .......................................................... 9
  Connecting the Breakouts ................................. 9
  The Final Step ............................................... 10
Code ............................................................ 12
  Libraries ...................................................... 12
  The Secrets File ............................................. 13
  Download the Code ......................................... 14
code.py ......................................................... 18
code.py ......................................................... 18
aio.py .......................................................... 22
air_quality.py ............................................... 26
gps.py .......................................................... 29
Adafruit IO ..................................................... 31
An Alternative Build ......................................... 33
  Code Changes ............................................... 35
  Alternative Parts for this Version ....................... 36
Overview

This guide will walk you through making an environmental monitoring device with **GPS, Temperature, Humidity, Barometric Pressure,** and **Air Quality** sensors, all updating to the cloud. Once built, this project takes readings occasionally and sends them to Adafruit IO, stamped with time and location. All that is required is an AC outlet and a WiFi network.

**All done in CircuitPython with AirLift**, our wireless co-processor that makes secure IoT projects a breeze. And we stick it in a cute weatherproof case so it can be deployed anywhere.

While one of these devices is useful to monitor some environment (say your room, a greenhouse, or a garden area) many of them could be placed around a house or city to monitor conditions in various areas.
This collection of time and location stamped data could be very useful for determining pollution patterns and so forth.

The units are small, lightweight, and have minimal requirements (power and WiFi) so they can be placed almost anywhere for extended periods of time, or moved around.

When the unit starts up, it checks where it is. Every hour, it refreshes the time from AdafruitIO. Periodically it reads all its sensors and sends the readings to AdafruitIO. This is set to happen every 5 minutes in the code, but can be changed as required. The only limitation is that there has to be enough time between readings to let them be sent and AdafruitIO has some limitations on frequency of updates for the free and plus versions.

Parts

Adafruit Feather M4 Express - Featuring ATSAMD51
It's what you've been waiting for, the Feather M4 Express featuring ATSAMD51. This Feather is fast like a swift, smart like an owl, strong like a ox-bird (it's half ox,...
$22.95
In Stock
Add to Cart

Adafruit AirLift – ESP32 WiFi Co-Processor Breakout Board
Give your plain ol' microcontroller project a lift with the Adafruit AirLift - a breakout board that lets you use the powerful ESP32 as a WiFi co-processor. You probably...
$9.95
In Stock
Add to Cart

Adafruit Ultimate GPS FeatherWing
Give your Feather a sense of place, with an Ultimate GPS FeatherWing. In 2013 we designed the Ultimate GPS...
$24.95
In Stock
Add to Cart

Adafruit BME280 I2C or SPI Temperature Humidity Pressure Sensor
Bosch has stepped up their game with their new BME280 sensor, an environmental sensor with temperature, barometric pressure and humidity! This sensor is great for all sorts...
$14.95
In Stock
Add to Cart

Your browser does not support the video tag.
PM2.5 Air Quality Sensor and Breadboard Adapter Kit
Breathe easy, knowing that you can track and sense the quality of the air around you with the PM2.5 Air Quality Sensor with Breadboard Adapter particulate sensor....
$39.95
In Stock

1 x Pycom Universal IP67 Case
Small, configurable case with knockouts for various connectors. Well sealed if you need it to be.

1 x SMA to uFL/u.FL/IPX/IPEX RF Adapter Cable
To connect the GPS board to an external GPS antenna

1 x GPS Antenna - External Active Antenna - 3-5V 28dB 5 Meter SMA
External GPS antenna

1 x Short Headers Kit for Feather
12-pin + 16-pin short female headers for the Feather

1 x Short Feather Male Headers
12-pin and 16-pin short male headers for the GPS wing

1 x Silicone Cover Stranded-Core Ribbon Cable
10 wire 1 meter long - 28AWG black ribbon cable for wiring up the boards
The Circuit

Due to the extensive use of breakout boards, constructing this circuit is mostly a matter of wiring the boards together.

Power

Because of the power requirement of the WiFi transmitter, a 5v supply of at least 1A is advised. It connects to the USB pin of the Feather, Vin of the BMP280 and AirLift, and Vcc of the Air Quality Sensor. The grounds of all boards connect together and to the power supply’s ground.

BMP280

This connects to the Feather’s I2C interface as described in the BMP280 guide (https://adafru.it/EDU).

GPS

This projects uses the Ultimate GPS Featherwing (https://adafru.it/weQ) so, while the connections are shown above using wires, it simply plugs onto the Feather, using TX and RX.

Air Quality Sensor

This also requires a serial interface, but thankfully the SAMD51 has plenty of those. We can use A2 and A3 as TX and RX, respectively. The Sensor connects via the supplied cable and breakout. In the wiring diagram above it’s shows as the header that the breakout can plug into. See the product guide (https://adafru.it/EDV) for more information.
AirLift

The [AirLift breakout](https://adafru.it/EDW) connects to the Feather's SPI lines as well as D6, D9, and D10 for RST, BUSY, and CS respectively.
Construction

The PYCOM case is a nice one for this project. It's a convenient size which turns out to be just big enough to hold everything. It also has some easy to remove (carefully with an craft knife) cutouts for connectors. But those cutouts aren't always what's needed. Fortunately, the plastic the case is made from is very easy to drill and cut to make custom holes and cutouts.

Preparing the Case

Start by trimming out some of the molded-in mounting points and wings to make room for our components. The photo shows a case as it comes on the right, and trimmed on the left. The parts that were trimmed out are marked with red. These are easy to remove using diagonal cutters.

Below, you can see that all the cut outs have been removed on one side of the case. From left to right, these are use for the GPS antenna connector, the power barrel jack, and an opening for the BME280.

On the opposite side of the case, we need to cut some custom openings for the air quality sensor. Note the large round opening as well as the smaller rectangular opening. These correspond to the air intake and exhaust of the sensor.
Please be careful cutting or drilling plastic, eye protection as well as safe cutting practices should be used. An adult should help a younger maker with the cutting.

Wiring

First add the power and antenna connectors. Start by soldering a short length of wire to the power and ground connections of the barrel connector. Slip the wires through the appropriate hole in the case, slide the nut over the wires and secure the connector in place. The antenna connector goes through the smaller round hole with a washer (included with the connector) on the inside and outside of the case, and add the nut. You can use one pair of pliers to hold the connector on the inside and another to tighten the nut on the outside. Just make it snug, be careful not to overtighten it.

Short lengths of ribbon cable are connected to the BME280, AirLift, and air sensor breakouts. Solder wires directly into the header strip holes of the breakouts: no headers required. You will have to carefully desolder and remove the header strip on the air sensor breakout.

You’ll need 8 wires for the AirLift and 4 wires for the other two. The AirLift needs wires for: VIN, GND, SCK, MISO, MOSI, CS, BUSY, and RST. The BME280 needs wires for VIN, GND, SCK, and SDI (SDI serves as SDA when using I2C which we will be doing). Finally, the air sensor breakout needs VCC, GND, RXD, and TXD connected.

Connecting the Breakouts
Now, wire the breakouts directly to the pins under the feather board. You can use the strip of GND connections along the prototyping section of the feather for the breakouts' ground connections. One trick you can do is carefully cut the trace to the 3v strip on the other side of the prototype area and connect it to the USB pin of the feather with a bit of wire, taking care not to short against any other pins. Then you can connect the Vcc and Vin connections of the breakouts to that. Here you can see the breakouts connected to the feather.

The next step is to connect the wires from the power jack to the Feather. And the final step is to connect the GPS antenna and pack everything in.

When everything is happily working, you can use a bit of hot glue to hold the BME280 board, the air sensor breakout, and the air sensor in place inside the box. Ensure the air sensor is positioned over the hole you made for it.
The final step is to carefully tuck in all the cables, taking care not to disconnect the GPS wire, and then attach the cover.
Code

CircuitPython is a programming language based on Python, one of the fastest growing programming languages in the world. It is specifically designed to simplify experimenting and learning to code on low-cost microcontroller boards. Here is a guide which covers the basics:

- Welcome to CircuitPython! (https://adafru.it/cpy-welcome)

Be sure you have the latest CircuitPython for the Feather M4 Express loaded onto your board, as described here (https://adafru.it/CJN).

CircuitPython is easiest to use within the Mu Editor. If you haven't previously used Mu, this guide will get you started (https://adafru.it/ANO).

Libraries

Plug your Feather M4 Express board into your computer via a USB cable. Please be sure the cable is a good power+data cable so the computer can talk to the Feather board.

A new disk should appear in your computer's file explorer/finder called CIRCUITPY. This is the place we'll copy the code and code library. If you can only get a drive named FEATHERBOOT, load CircuitPython per the Feather guide above.

Create a new directory on the CIRCUITPY drive named lib.

Download the latest CircuitPython driver package to your computer using the green button below. Match the library you get to the version of CircuitPython you are using. Save to your computer's hard drive where you can find it.

https://adafru.it/zB-
With your file explorer/finder, browse to the bundle and open it up. Copy the following folders and files from the library bundle to your CIRCUITPY lib directory you made earlier:

- adafruit_bme280
- adafruit_logging
- adafruit_esp32spi
- adafruit_gps

All of the other necessary libraries are baked into CircuitPython!

Your CIRCUITPY/lib directory should look like the snapshot below.

The Secrets File

The secrets.py file in the root/main directory of the CIRCUITPY drive contains bits of information that should never be put in source code files. That includes things like usernames, passwords, and location specific information. In the case of this projects that's:

- SSID (name) and password for connecting to the local WiFi network,
- timezone for fetching the local time, and
- AdafruitIO account credentials.

The file should look like the following, except that it contains your information. Note that the WiFi credentials are encoded as bytearrays (the b prefix on the strings)
secrets = {
    'ssid' : b'My_SSID',
    'password' : b'My_WIFI_Password',
    'timezone' : 'Area/City',
    'aio_username' : 'my_username',
    'aio_key' : 'my_key',
}

Download the Code

You'll need to download the zip of the project to get all the required files. You can do this here.

```python
import time
import board
import busio
import air_quality
import gps
import adafruit_bme280
import aio
import adafruit_logging as logging

logger = logging.getLogger('main')
logger.setLevel(logging.INFO)

gps_uart = busio.UART(board.TX, board.RX, baudrate=9600, timeout=3.000)
gps_interface = gps.Gps(gps_uart)
gps_interface.begin()

logger.debug('GPS started')
aio_interface = aio.AIO()
if aio_interface.onboard_esp:
    air_uart = busio.UART(board.D5, board.D7, baudrate=9600)
else:
    air_uart = busio.UART(board.A2, board.A3, baudrate=9600)
air = air_quality.AirQualitySensor(air_uart)

logger.debug('Air quality sensor started')
```
```python
i2c = busio.I2C(board.SCL, board.SDA)
bme280 = adafruit_bme280.Adafruit_BME280_I2C(i2c)

reading_interval = 300.0
reading_time = time.monotonic()

time_update_interval = 3600.0
time_update_time = time.monotonic()

logger.info('Getting data from GPS')

while True:
    if gps_interface.get_fix():
        break
    logger.error('Failed getting fix... retrying')

gps_interface.read()

logger.info('Starting reading loop')

payload = {'value': 0,
            'lat': gps_interface.latitude,
            'lon': gps_interface.longitude,
            'created_at': ''}

while True:
    now = time.monotonic()

    if now >= time_update_time:
        time_update_time = now + time_update_interval
        logger.info('refreshing time')
        try:
            aio_interface.refresh_local_time()
        except RuntimeError as e:
            logger.debug('Time refresh failed with: %s', str(e))

    if now >= reading_time:
        reading_time = now + reading_interval
        logger.info('Taking a reading')

        st = time.localtime()
        timestamp = '{0}/{1:02}/{2:02} {3:2}:{4:02}:{5:02}'.format(st.tm_year,
                                                               st.tm_mon,
                                                               st.tm_mday,
                                                               st.tm_hour,
                                                               st.tm_min,
                                                               st.tm_sec)

        payload['created_at'] = timestamp

        if air.read():
            logger.info('Air Quality pm10 standard: %d', air.pm10_standard)
            payload['value'] = air.pm10_standard
            if not aio_interface.post('environmental-sensor.pm10-std', payload):
                logger.critical('post of pm10 standard failed')
                continue
```

© Adafruit Industries 2023
https://learn.adafruit.com/remote-iot-environmental-sensor
logger.info('Air Quality pm25 standard: %d', air.pm25_standard)
payload['value'] = air.pm25_standard
if not aio_interface.post('environmental-sensor.pm25-std', payload):
    logger.critical('post of pm25 standard failed')
    continue

logger.info('Air Quality pm100 standard: %d', air.pm100_standard)
payload['value'] = air.pm100_standard
if not aio_interface.post('environmental-sensor.pm100-std', payload):
    logger.critical('post of pm100 standard failed')
    continue

logger.info('Air Quality pm10 env: %d', air.pm10_env)
payload['value'] = air.pm10_env
if not aio_interface.post('environmental-sensor.pm10-env', payload):
    logger.critical('post of pm10 env failed')
    continue

logger.info('Air Quality pm25 env: %d', air.pm25_env)
payload['value'] = air.pm25_env
if not aio_interface.post('environmental-sensor.pm25-env', payload):
    logger.critical('post of pm25 env failed')
    continue

logger.info('Air Quality pm100 env: %d', air.pm100_env)
payload['value'] = air.pm100_env
if not aio_interface.post('environmental-sensor.pm100-env', payload):
    logger.critical('post of pm100 env failed')
    continue

logger.info('Air Quality particles 03um: %d', air.particles_03um)
payload['value'] = air.particles_03um
if not aio_interface.post('environmental-sensor.03um', payload):
    logger.critical('post of particles 03um failed')
    continue

logger.info('Air Quality particles 05um: %d', air.particles_05um)
payload['value'] = air.particles_05um
if not aio_interface.post('environmental-sensor.05um', payload):
    logger.critical('post of particles 05um failed')
    continue

logger.info('Air Quality particles 10um: %d', air.particles_10um)
payload['value'] = air.particles_10um
if not aio_interface.post('environmental-sensor.10um', payload):
    logger.critical('post of particles 10um failed')
    continue

logger.info('Air Quality particles 25um: %d', air.particles_25um)
payload['value'] = air.particles_25um
if not aio_interface.post('environmental-sensor.25um', payload):
    logger.critical('post of particles 25um failed')
    continue

logger.info('Air Quality particles 50um: %d', air.particles_50um)
payload['value'] = air.particles_50um
if not aio_interface.post('environmental-sensor.50um', payload):
    logger.critical('post of particles 50um failed')
    continue
logger.critical('post of particles 100um failed')
continue

logger.info('Air Quality particles 100um: %d', air.particles_100um)
payload['value'] = air.particles_100um
if not aio_interface.post('environmental-sensor.100um', payload):
    logger.critical('post of particles 100um failed')
continue

logger.info('Temperature: %f', bme280.temperature)
payload['value'] = bme280.temperature
if not aio_interface.post('environmental-sensor.temperature', payload):
    logger.critical('post of temperature failed')
continue

logger.info('Humidity: %f', bme280.humidity)
payload['value'] = bme280.humidity
if not aio_interface.post('environmental-sensor.humidity', payload):
    logger.critical('post of humidity failed')
continue

logger.info('Pressure: %f', bme280.pressure)
payload['value'] = bme280.pressure
if not aio_interface.post('environmental-sensor.pressure', payload):
    logger.critical('post of pressure failed')
continue

logger.info('Waiting for next reading')
This starts off by setting up a logger that can be used throughout the system.

```python
logger = logging.getLogger('main')
logger.setLevel(logging.INFO)
```

It then sets up the various pieces of hardware and software it will be using to make readings and publish them to AdafruitIO.

```python
gps_uart = busio.UART(board.TX, board.RX, baudrate=9600, timeout=3.000)
gps_interface = gps.Gps(gps_uart)
gps_interface.begin()

air_uart = busio.UART(board.A2, board.A3, baudrate=9600)
air = air_quality.AirQualitySensor(air_uart)

i2c = busio.I2C(board.SCL, board.SDA)
bme280 = adafruit_bme280.Adafruit_BME280_I2C(i2c)

aio_interface = aio.AI0()
```

Readings will be taken every 5 minutes (300 seconds) and the time will be refreshed hourly (3600 seconds) from AdafruitIO. To accomplish this, we'll set up an interval and scheduling time for each.

```python
reading_interval = 300.0
reading_time = time.monotonic()

time_update_interval = 3600.0
time_update_time = time.monotonic()
```

Once everything is set up, the GPS is read to discover where the device is. Note that, since the purpose of the project is to be stationary and collect data, the GPS location is only read once when the system starts up.
while True:
    if gps_interface.get_fix():
        break
    logger.error('Failed getting fix... retrying')

gps_interface.read()

Since only the data value and time will change for different posts to AdafruitIO, a template is made that can then be updated for each post.

```
payload = {'value' : 0, 'lat' : gps_interface.latitude, 'lon' : gps_interface.longitude,
'created_at' : ''}
```

Now that everything is initialized and configured, it's time for the main loop.

The first thing that happens is to grab the current `time`. It's not the time so much as a time related value. `time.monotonic()` returns the number of seconds since the system started up. But it's an always increasing number of seconds, and that's exactly what we need for scheduling.

```
now = time.monotonic()
```

That time value can be used to check against the scheduled times to see whether something needs to be done. First, is it time to refresh the system time? Since we can fetch this from a time server (AdafruitIO conveniently provides one), we don't need to worry about having a hardware RTC: we can just refresh the software RTC occasionally. For this use that's accurate enough.

This is a common approach to scheduling in a single threaded environment: set the time something should happen (the scheduled time), when the current time equals or is later than the scheduled time, set the next scheduled time and do whatever needs doing.

```
if now >= time_update_time:
    time_update_time = now + time_update_interval
    logger.info('refreshing time')
    try:
        aio_interface.refresh_local_time()
    except RuntimeError as e:
        logger.debug('Time refresh failed with: %s', str(e))
```

The next step is to make and report a set of readings if it's time to do so. It starts by formatting a timestamp and putting it in the payload.
if now >= reading_time:
    reading_time = now + reading_interval
    logger.info('Taking a reading')

    st = time.localtime()
    timestamp = '{0}/{1:02}/{2:02} {3:2}:{4:02}:{5:02}'.format(st.tm_year,
    st.tm_mon,
    st.tm_mday,
    st.tm_hour,
    st.tm_min,
    st.tm_sec)

    payload['created_at'] = timestamp
...  

Next, to read and report air quality measurements. To handle times when the sensor might not read successfully, this is conditional. If a fresh reading can't be taken for some reason, the air quality readings are skipped. If a reading was taken, each value in turn is placed in the payload and it is sent to AdafruitIO. If sending fails, this reading cycle is stopped and we hope it works the next time.

Note that the `continue` statement will cause the program to jump back to the start of the `while True:` loop.

if air.read():
    logger.info('Air Quality pm10 standard: %d', air.pm10_standard)
    payload['value'] = air.pm10_standard
    if not aio_interface.post('environmental-sensor.pm10-std', payload):
        logger.critical('post of pm10 standard failed')
        continue

    logger.info('Air Quality pm25 standard: %d', air.pm25_standard)
    payload['value'] = air.pm25_standard
    if not aio_interface.post('environmental-sensor.pm25-std', payload):
        logger.critical('post of pm25 standard failed')
        continue

...
logger.info('Temperature: %f', bme280.temperature)
payload['value'] = bme280.temperature
if not aio_interface.post('environmental-sensor.temperature', payload):
    logger.critical('post of temperature failed')
    continue

logger.info('Humidity: %f', bme280.humidity)
payload['value'] = bme280.humidity
if not aio_interface.post('environmental-sensor.humidity', payload):
    logger.critical('post of humidity failed')
    continue

logger.info('Pressure: %f', bme280.pressure)
payload['value'] = bme280.pressure
if not aio_interface.post('environmental-sensor.pressure', payload):
    logger.critical('post of pressure failed')
    continue

logger.info('Waiting for next reading')
Here we have a class that manages all interaction with AdafruitIO via the ESP32 AirLift.

The file starts with imports, logger fetching, some string constants for the time service and the import of the secrets file.

```python
import busio
import gc
from digitalio import DigitalInOut
from adafruit_esp32spi import adafruit_esp32spi
import adafruit_esp32spi_requests as requests
import adafruit_logging as logging
import rtc

logger = logging.getLogger('main')

TIME_SERVICE = "https://io.adafruit.com/api/v2/%s/integrations/time/strftime?x-aio-key=%s"
# our strftime is %Y-%m-%d %H:%M:%S,%L %j %u %z %Z see http://strftime.net/ for decoding details
# See https://apidock.com/ruby/DateTime/strftime for full options
TIME_SERVICE_STRFTIME = '&fmt=%25Y-%25m-%25d+%25H%3A%25M%3A%25S%3A%25L%3A%25j%3A%25u%3A%25z%3A%25Z'

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

Constructing the class sets up the SPI interface to the AirLift and initializes the web interface.
def __init__(self):
    esp32_cs = DigitalInOut(board.D10)
    esp32_ready = DigitalInOut(board.D9)
    esp32_reset = DigitalInOut(board.D6)

    spi = busio.SPI(board.SCK, board.MOSI, board.MISO)
    self._esp = adafruit_esp32spi.ESP_SPIcontrol(spi, esp32_cs, esp32_ready, esp32_reset)

    if self._esp.status == adafruit_esp32spi.WL_IDLE_STATUS:
        logger.debug('ESP32 found and in idle mode')
        logger.info('Firmware vers. %s', self._esp.firmware_version)
        logger.info('MAC addr: %s', ':'.join([hex(i)[2:4] for i in self._esp.MAC_address]))

        requests.set_interface(self._esp)

There is a convenience method for connecting to WiFi.

def connect(self):
    logger.debug("Connecting...")
    while not self._esp.is_connected:
        try:
            self._esp.connect_AP(secrets['ssid'], secrets['password'])
        except RuntimeError as e:
            logger.error("could not connect to AP, retrying: %s", e)
            continue

Now we have the main methods. First one to post a payload dictionary to AdafruitIO.

It ensures that there is a live connection to WiFi. Then it tries to post to AdafruitIO. It will make 5 attempts
before giving up. If the post gets processed ok, it closes the response and returns whether the post
resulted in a 200 status (i.e. OK). If a `RuntimeError` was raised by the post, the ESP32 is reset, it
reconnects to WiFi, and tries again. In practice an exception is occasionally raised, but a reset and retry
succeeds.
def post(self, feed, payload):
    api_url = 'https://io.adafruit.com/api/v2/{0}/feeds/{1}/data'.format(secrets['aio_username'], feed)
    logger.info('POSTing to %s', api_url)
    logger.info('payload: %s', str(payload))
    auth_header = {'X-AIO-KEY':secrets['aio_key']}
    self.connect()
    r = None
    tries = 0
    while True:
        if tries == 5:
            return False
        tries += 1
        try:
            r = requests.post(api_url, headers=auth_header, json=payload)
            logger.info('Status: %d', r.status_code)
            if r.status_code == 200:
                logger.debug('Headers: %s', str(r.headers))
                logger.debug('Text: %s', str(r.json()))
            else:
                logger.debug('Text: %s', str(r.json()))
            break
        except RuntimeError as err:
            logger.error('Error posting: %s', str(err))
            logger.info('Resetting and reconnecting')
            self._esp.reset()
            self.connect()
    r.close()
    return r.status_code == 200

Finally there is the method to update the time from AdafruitIO. This is copied from the PyPortal library.

First it makes sure an AdafruitIO username and password as well a timezone specification have been provided in the secrets file. If so, the time fetch URL is constructed and a GET is used to fetch the time information. If that returns successfully, the result text is torn apart to extract the date and time. That’s used to set the system’s software RTC as well as being returned.
def refresh_local_time(self):
    # pylint: disable=line-too-long
    """Fetch and "set" the local time of this microcontroller to the local time at the
    location, using an internet time API.
    Copied from adafruit_pyportal
    :param str location: Your city and country, e.g. "New York, US".
    """
    # pylint: enable=line-too-long
    api_url = None
    try:
        aio_username = secrets['aio_username']
        aio_key = secrets['aio_key']
    except KeyError:
        raise KeyError("Our time service requires a login/password to rate-limit. Please
register for a free adafruit.io account and place the user/key in your secrets file under
'aio_username' and 'aio_key'")
    # pylint: disable=line-too-long
    if location:
        logger.debug('Getting time for timezone %s', location)
        api_url = (TIME_SERVICE + '&tz=%s') % (aio_username, aio_key, location)
    else:  # we'll try to figure it out from the IP address
        logger.debug("Getting time from IP address")
        api_url = TIME_SERVICE % (aio_username, aio_key)
    api_url += TIME_SERVICE_STRFTIME
    logger.debug('Requesting time from %s', api_url)
    try:
        self.connect()
        response = requests.get(api_url)
        logger.debug('Time reply: %s', response.text)
        times = response.text.split(' ')
        the_date = times[0]
        the_time = times[1]
        year_day = int(times[2])
        week_day = int(times[3])
        is_dst = None  # no way to know yet
    except KeyError:
        raise KeyError("Was unable to lookup the time, try setting secrets["timezone"]
according to http://worldtimeapi.org/timezones")
    year, month, mday = [int(x) for x in the_date.split('-')]
    the_time = the_time.split(':')[0]
    hours, minutes, seconds = [int(x) for x in the_time.split(':')]
    now = time.struct_time((year, month, mday, hours, minutes, seconds, week_day, year_day,
                             is_dst))
    rtc.RTC().datetime = now
    logger.debug('Fetched time: %s', str(now))
    # now clean up
    response.close()
    response = None
    gc.collect()
    return now
This file contains a class that provides an interface to the air quality sensor. The constructor saves the passed in UART interface and initializes instance variables for caching the measurements.

```python
def __init__(self, uart):
    self._uart = uart
    self._buffer = []
    self._pm10_standard = 0
    self._pm25_standard = 0
    self._pm100_standard = 0
    self._pm10_env = 0
    self._pm25_env = 0
    self._pm100_env = 0
    self._particles_03um = 0
    self._particles_05um = 0
    self._particles_10um = 0
    self._particles_25um = 0
    self._particles_50um = 0
    self._particles_100um = 0
```

The other significant method reads the measurement data from the sensor and caches it in the instance variables that were initialized in the constructor.

Up to 32 bytes are read from the sensor through the UART, and appended to the buffer.

Then bytes are popped off the front of the buffer (index 0) until one with value 0x42 is found. Now the
buffer size is checked. If it's too big, there's a problem and we throw out the buffer and return False. If the buffer doesn't have enough in it, it simply returns False. More data will be read into the buffer next time a read is requested.

If everything is good, readings are extracted from the raw data in the first 32 bytes of the buffer. These readings are placed into corresponding instance variables.

The checksum is checked. If it doesn't match, the entire buffer is discarded and False is returned. Otherwise everything is good and True is returned.

```python
def read(self):
    data = self._uart.read(32)  # read up to 32 bytes
    data = list(data)
    self._buffer += data
    while self._buffer and self._buffer[0] != 0x42:
        self._buffer.pop(0)
    if len(self._buffer) > 200:
        self._buffer = []  # avoid an overrun if all bad data
    if len(self._buffer) < 32:
        return False
    if self._buffer[1] != 0x4d:
        self._buffer.pop(0)
        return False
    frame_len = struct.unpack(">H", bytes(self._buffer[2:4]))[0]
    if frame_len != 28:
        return False
    logger.debug('buffer length: %d', len(self._buffer) - 4)
    frame = struct.unpack(">HHHHHHHHHHHHHHH", bytes(self._buffer[4:32]))
    self._pm10_standard, self._pm25_standard, self._pm100_standard, self._pm10_env, 
    self._pm25_env, self._pm100_env, self._particles_03um, self._particles_05um, 
    self._particles_10um, 
    self._particles_25um, self._particles_50um, self._particles_100um, skip, checksum = frame
    check = sum(self._buffer[0:30])
    if check != checksum:
        return False
    return True
```

Once the measurements have been cached, the core code can then read them values through a set of properties.
@property
def pm10_standard(self):
    return self._pm10_standard

@property
def pm25_standard(self):
    return self._pm25_standard

@property
def pm100_standard(self):
    return self._pm100_standard

@property
def pm10_env(self):
    return self._pm10_env

@property
def pm25_env(self):
    return self._pm25_env

@property
def pm100_env(self):
    return self._pm100_env

@property
def particles_03um(self):
    return self._particles_03um

@property
def particles_05um(self):
    return self._particles_05um

@property
def particles_10um(self):
    return self._particles_10um

@property
def particles_25um(self):
    return self._particles_25um

@property
def particles_50um(self):
    return self._particles_50um

@property
def particles_100um(self):
    return self._particles_100um
The only thing left is an interface to the GPS library to bundle up it's use. The constructor takes a UART object and creates a GPS object with it. It also initializes instance variable in which to cache the GPS location data.

```python
def __init__(self, uart):
    self._gps = adafruit_gps.GPS(uart, debug=False)
    self._latitude = 0
    self._longitude = 0
```

Once an instance has been created, it is initialized using the `begin` method which configures the GPS object.

```python
def begin(self):
    self._gps.send_command(b'PMTK314,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0')
    self._gps.send_command(b'PMTK220,1000')
```

A big part of using GPS is getting a fix. I.e. locking onto signals from the satellites in view/range. No data can be fetched until that is done. The `get_fix` method takes care of waiting for a fix to be acquired. Once it has, `True` is returned. If an exception is raised, `False` is returned.
def get_fix(self):
    try:
        logger.debug('Calling gps update')
        self._gps.update()
        logger.debug('Back from gps update')
        while not self._gps.has_fix:
            # Try again if we don't have a fix yet.
            logger.debug('Waiting for fix...')
            time.sleep(0.1)
            self._gps.update()
        return True
    except UnicodeError:
        return False

The `read` method pulls data from the GPS and caches it. Corresponding properties return the cached data.

```python
def read(self):
    logger.debug('Reading GPS')
    self._latitude = self._gps.latitude
    self._longitude = self._gps.longitude

@property
def latitude(self):
    return self._latitude

@property
def longitude(self):
    return self._longitude
```

Notice that there's no attempt to re-establish a fix since we're only reading from the GPS once, when the system starts up.
The code, as it is, uses several feeds in a single feed group that you'll need to set up first.

If you don't have an Adafruit IO account or are unfamiliar with setting up feed/groups, see the Welcome to Adafruit IO (https://adafruit.io/BRB) guide.

You will need to set up a group named Environmental_Sensor containing the following feeds:

- 03um
- 05um
- 10um
- 25um
- 50um
- 100um
- pm10_env
- pm25_env
- pm100_env
- pm10_std
- pm25_std
- pressure
- temperature
- humidity

Add your Adafruit IO account credentials to the `secrets.py` file and it should be ready to go.

You can look at the collected data directly in the feeds, but making a dashboard gives a better (sometimes) view of it.
An Alternative Build

The Metro M4 AirLift Lite provides another way to build this project. This is a Metro M4 Express with a built-in ESP32 coprocessor. One advantage is that we don't have to wire it or mount it. Another is that it's not using the board's GPIO for support connections (CS, BUSY, etc). It uses a different, internal set.
For this build we can use the ultimate GPS shield. It provides the GPS connected to the Metro’s Tx/Rx, battery backup, an SD card interface (which we don’t use here) and, importantly, a sizable prototyping area.

That prototyping area is used to mount the BME280 and air sensor breakout. There is plenty of room for both. The air sensor breakout can go toward the end of the board to ease the connection of its cable.

Wiring is minimal:

- **5v** and **Gnd** to each breakout
- **SCL** and **SDA** to the BME280 breakout
- **Tx/D5** and **Rx/D7** to the air sensor breakout

You can solder the breakouts directly onto the shield or use strips of female header to match the make header on the breakouts.
Code Changes

There are a couple small changes to the code to accommodate running on a Metro M4 AirLift Lite as well as a Feather M4 Express:

- The Metro has dedicated SPI control pins for the ESP32 instead of using general purpose GPIO pins as we did on the Feather.
- The Metro's routing of physical pins on the SAMD51 to interface pins on the board are, unsurprisingly, done differently than on the Feather. The impact of that is that we need to use different pins for Rx and Tx. This build uses D5 and D7, respectively. How do we find pairs of pins that can be used for Tx/Rx? See the Where's my UART? section on this page in the CircuitPython Essentials guide (https://adafru.it/C7A)

The constructor of the AIO class (in aio.py) changes to accommodate the possibility of a built-in ESP. It saves whether it found one and makes that information available with a property.

```python
def __init__(self):
    try:
        esp32_cs = DigitalInOut(board.ESP_CS)
        esp32_busy = DigitalInOut(board.ESP_BUSY)
        esp32_reset = DigitalInOut(board.ESP_RESET)
        self._onboard_esp = True
    except AttributeError:
        esp32_cs = DigitalInOut(board.D10)
        esp32_busy = DigitalInOut(board.D9)
        esp32_reset = DigitalInOut(board.D6)
        self._onboard_esp = False

    spi = busio.SPI(board.SCK, board.MOSI, board.MISO)
    self._esp = adafruit_esp32spi.ESP_SPIcontrol(spi, esp32_cs, esp32_busy, esp32_reset)

    if self._esp.status == adafruit_esp32spi.WL_IDLE_STATUS:
        logger.debug('ESP32 found and in idle mode')
        logger.info('Firmware vers. %s', self._esp.firmware_version)
        logger.info('MAC addr: %s', ':'.join([hex(i)[2:4] for i in self._esp.MAC_address]))

    requests.set_interface(self._esp)

@property
def onboard_esp(self):
    return self._onboard_esp
```

The main hardware setup code changes a bit to select the appropriate Tx/Rx pins based on whether a built-in ESP was found (we're running on a Metro Air Lift) or not (we're running a Feather M4).
aio_interface = aio.AI0()

if aio_interface.onboard_esp:
    logger.info('Using onboard Airlift and air sensor on D5/D7')
    air_uart = busio.UART(board.D5, board.D7, baudrate=9600)
else:
    logger.info('Using external Airlift and air sensor on A2/A3')
    air_uart = busio.UART(board.A2, board.A3, baudrate=9600)
air = air_quality.AirQualitySensor(air_uart)

Other than those two changes, the code is identical.

Alternative Parts for this Version

**Adafruit Metro M4 Express AirLift (WiFi) - Lite**

Give your next project a lift with AirLift - our witty name for the ESP32 co-processor that graces this Metro M4. You already know about the Adafruit Metro...

$34.95

*In Stock*

**Add to Cart**

---

**Adafruit Ultimate GPS Logger Shield - Includes GPS Module**

Brand new and better than ever, we've replaced our Adafruit GPS shield kit with this assembled shield that comes with an Ultimate GPS module. This GPS shield works great with...

$29.95

*In Stock*

**Add to Cart**