Remote Data Logger Tutorial

Background

Overview

This tutorial explains how to run a basic demo of our Remote Data Logger. The Remote Data Logger stays in low-power "sleep" mode, and wakes up every few minutes or hours to collect data from its sensors. It can transmit its data via WiFi, or by sending them directly to a "collector" device.

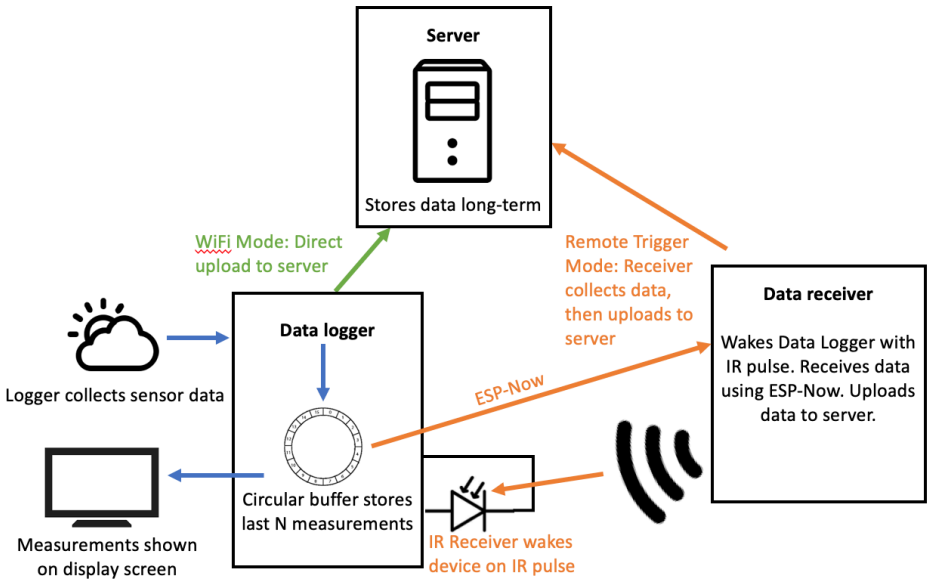

Figure 1: Notional diagram

The tutorial covers how to make this entire system: a Data Logger that has one sensor, one display screen, and an infrared-based triggering circuit; the Data Receiver; and the Server. However, the system is very modular: you can equip the Data Logger with your own sensors or use own triggering circuit on the Receiver, and modify the code accordingly.

The tutorial assumes you have the following background knowledge:

- How to use the command line to navigate to folders and run basic commands.
    - Read this for Mac: https://www.macworld.com/article/221277/master-the-command-line-navigating-files-and-folders.html
    - Read this for Windows: https://riptutorial.com/cmd/example/8646/navigating-in-cmd
- Basic understanding of circuits. You can just follow the wiring guide step-by-step and your device should work, but it helps to understand what's going on.
- If you want to create visualizations of the sensor data at the end, then you'll also need proficiency in Python or another programming language.

Detailed device specs and parts list

The device has two modes: an indoor mode where the sensor uses the WiFi network to upload its data to a server, and an outdoor mode where the sensor saves all its recordings. In outdoor mode, the device only wakes up when it is triggered by a signal from a collector device, just like a TV remote control powers on a TV. It sends data directly to the collector device, and the collector device uploads all the data to the server using WiFi.

The core of the device is an ESP 32 development board, which controls the device. There is a pressure, humidity, and temperature sensor, and an OLED display screen. The device uses a protocol called I2C to communicate with the sensor and the OLED. It has an infrared receiver module that allows the collector device to "trigger" the device, or wake it up from a distance. It has two buttons: one that allows the user to toggle between the two modes, and one that "triggers" the device in the same way the IR signal would.

Hardware: Parts required

You'll need the following parts to assemble both this board and the receiver board:

Item	Qty	Suggested link	Price
ESP 32 DevKit board	2	https://www.amazon.com/MELIFE-Development-Dual-Mode-Microcontroller-Integrated/dp/B07Q576VWZ/ref=sr_1_3?dchild=1&keywords=esp32&qid=1614190703&sr=8-3	$14.99
BME280 Sensor	1	https://www.amazon.com/dp/B07T2KFD4M/ref=cm_sw_r_cp_apa_fabc_6AV9CY91X9MPGW09A8SY	$22.99 (together)
SH1106 OLED screen	1		
HX-M21 IR Receiver	1	https://www.amazon.com/Digital-Receiver-Transmitter-Arduino-Compatible/dp/B01E20VQD8/ref=sr_1_5?dchild=1&keywords=arduino+ir+38khz&qid=1614191863&sr=8-5	$7.99
3-battery holder	1	https://www.amazon.com/LAMPVPATH-Battery-Holder-Switch-Leads/dp/B07JF3DD9Q/ref=sr_1_9?dchild=1&keywords=3+aaa+battery+holder&qid=1614191703&sr=8-9	$9.99
PCB	2		

If you're assembling on a breadboard instead of a PCB, you'll need the following additional components:

Item	Qty	Suggested link	Price
Solderless breadboard	3	https://www.amazon.com/dp/B01EV6LJ7G/ref=redir_mobile_desktop?_encoding=UTF8&aaxitk=IBR.z0Y.4NCvWJVp3bQxxQ&hsa_cr_id=1434913550101&pd_rd_plhdr=t&pd_rd_r=7f730906-3b69-4ff1-84da-695a7e8aba46&pd_rd_w=TmP9p&pd_rd_wg=xajaE&ref_=sbx_be_s_sparkle_mcd_asin_2_img	$9.99

Jumper wires	40	https://www.amazon.com/gp/product/ B07Z1BK7NG/ ref=ppx_yo_dt_b_asin_title_o06_s00?ie=UTF8&psc=1	$17.99
2.4KΩ resistor	2		
10KΩ resistor	5		
100KΩ resistor	1		
2N2222 transistor	2		
Tactile push buttons	4		
100 resistor	2		
IR LED	1	https://www.digikey.com/en/products/detail/osram-opto-semiconductors-inc/SFH-4545/2205955	$0.81

Data Logger Assembly Guide

If you're using a PCB:

1. **Mount the PCB**

Assembling the hardware is easy with our PCB. First, push the ESP board into the 15-hole connectors on the *back* (which has the drone logo).

2. **Mount the OLED, BME, and IR on the front**

Push these sensors in the labeled spots. Make sure the labels on the PCB match the labels on the devices.

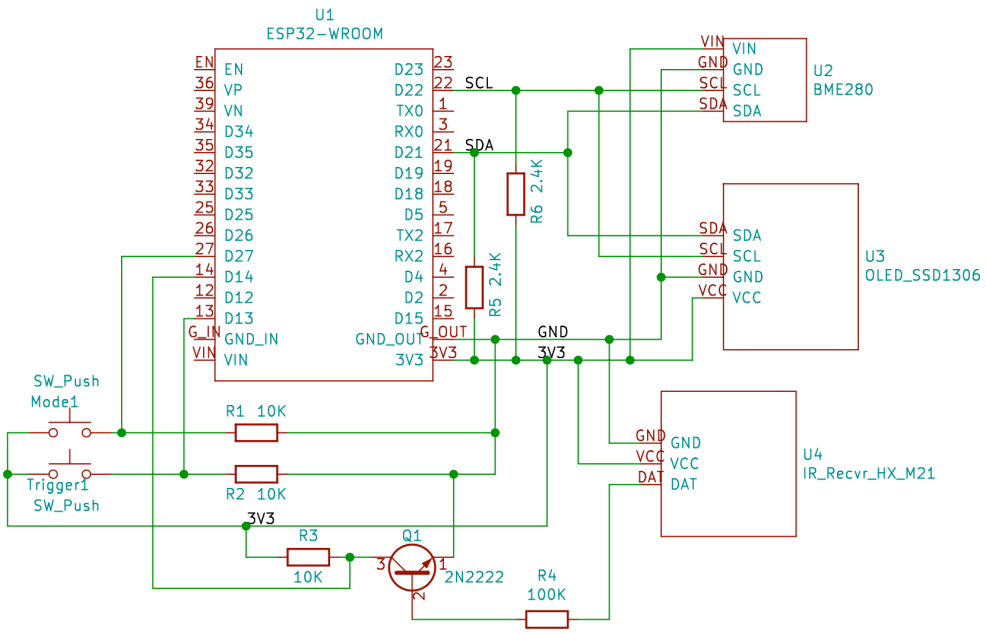

Figure 2: Data Logger schematic diagram

If you're using a breadboard, you'll be wiring up this diagram. If you've never used a breadboard before, take a look at the first few images in this article to understand how a breadboard works: https://computers.tutsplus.com/tutorials/how-to-use-a-breadboard-and-build-a-led-circuit--mac-54746

Now, we'll walk you through the connection guide step by step, with photos.

1. **Put the ESP32 into two breadboards:** the left pins on one breadboard, and the right pins in another. The pins are (annoyingly) spaced too far apart to fit into one breadboard while leaving room for jumper wires, so we need to use two.

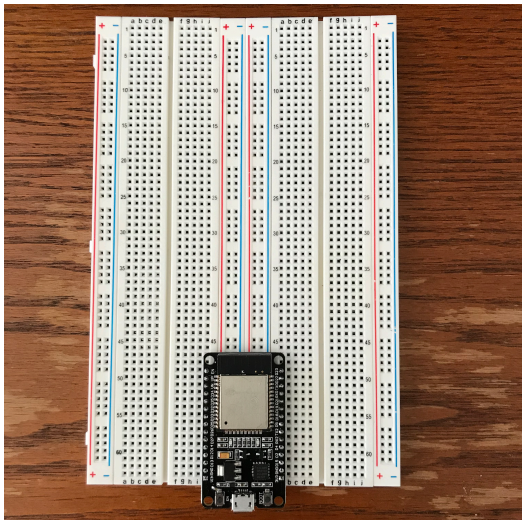

Figure 3(a): Place the ESP on the breadboard.

2. **Connect the power pins**. Connect the pin labeled "3.3V" to the long, vertical row of pins on the side of the breadboard, and the pin labeled "GND" to the second vertical row.

These are called the *ground and power rails*. The vertical rows might be colored or labeled red for power and blue for ground, but it depends on your breadboard. If your board has multiple power/ground rails, connect the power rails together, and ground rails together.

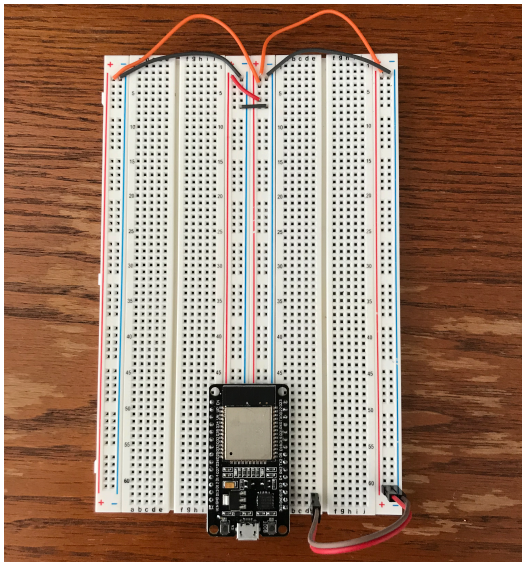

Figure 3(b): Connect the power rails.

3. **Plug in the BME sensor.** Put the BME into a set of four adjacent pins. Remember that pins on the breadboard are connected to each other horizontally, so place the BME vertically, so its four pins are *not* connected to each other, like so:

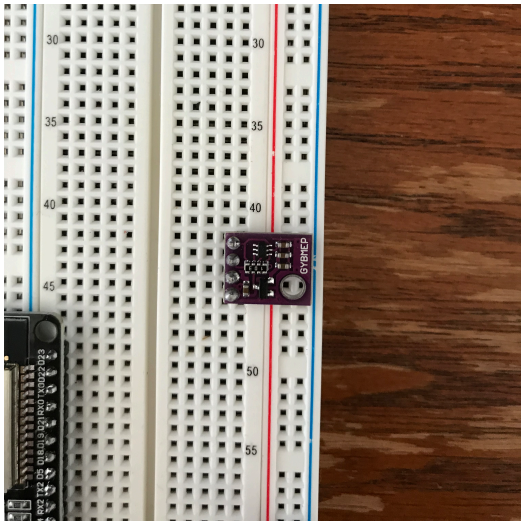

Figure 3(c): Place the BME280 sensor on the breadboard.

Also make sure not to overlap the BME pins with the ESP's pins!

4. **Wire the BME sensor**. Connect VIN to the power rail and GND to the ground rail. Connect SDA to the ESP's Pin 21, and connect SCL to the ESP's Pin 22.

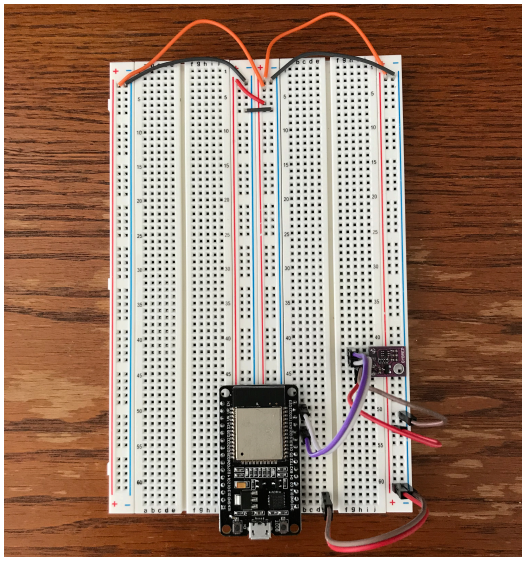

Figure 3(d): Wire the BME280 sensor.

These are used for a digital communication protocol called I^2C. The sensor sends data to the microcontroller using the SDA (data) and SCL (clock) wires. Read this article to learn more about I^2C: https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/

5. **Plug in and wire the OLED**. Plug in the OLED horizontally, just like you did for the BME. Just like the BME, connect VIN to power, GND to ground, SDA to Pin 21, and SCL to Pin 22.

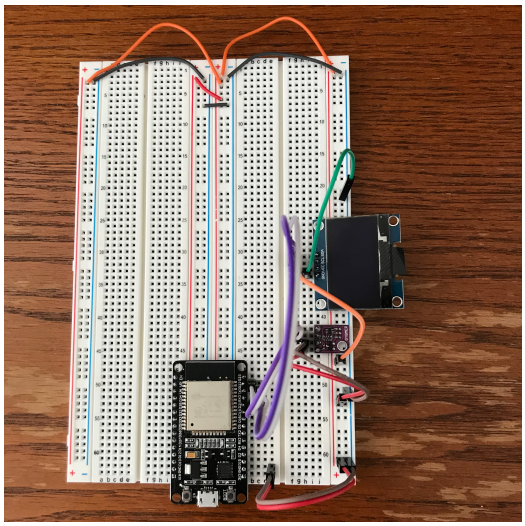

Figure 3(e): Place and wire the OLED screen.

6. **Pull up the I2C line.** Those SDA and SCL connections are used to send data to the ESP device using a protocol known as “I2C.” When no data is being sent, the lines are “pulled high,” which means their voltage is equal to 3.3V volts instead of ground. To do this, we’re going to add “pull-up resistors” on the SDA and SCL lines. Connect one end

of your 2.4K Ohm resistor to Pin 21, and connect the other end to Power. Do the same for Pin 22.

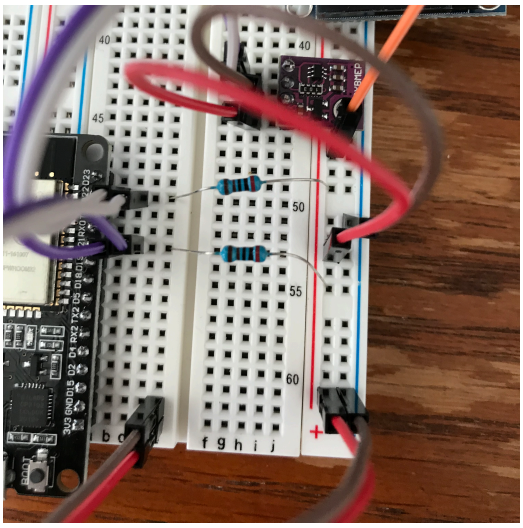

Figure 3(f): Wire pull-up resistors on the SDA and SCL pins.

7. **Place the button.** Put the button in between two halves of one breadboard, like this:

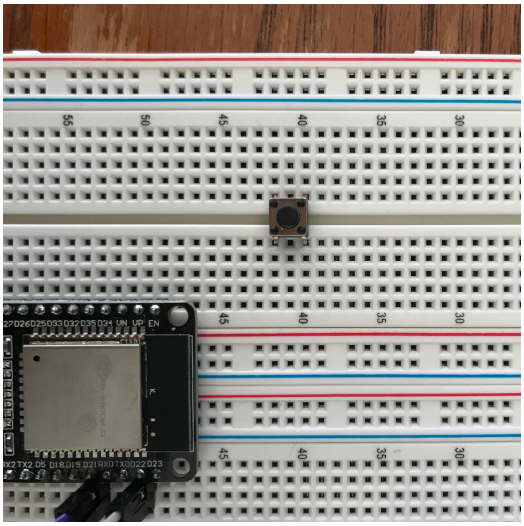

Figure 3(g)(i): Place the push button on the breadboard.

Two sides of the button are already connected to each other, as shown in the following diagram: When you click the button, it connects the two sides together.

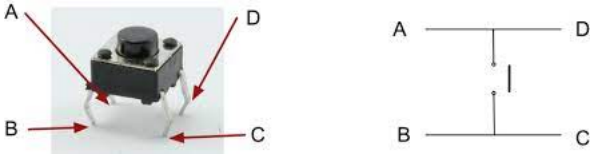

Figure 3(g)(ii): Wiring diagram of the tactile buttons. Source: https://learn.adafruit.com/adafruit-arduino-lesson-6-digital-inputs/push-switches

8. **Wire the button**. Connect one side of the button to power. Connect one side of the 10K resistor to the other side of the button, and connect the other side of the resistor to ground. Finally, connect the second side of the button (the one that's connected to the resistor) to Pin 27.

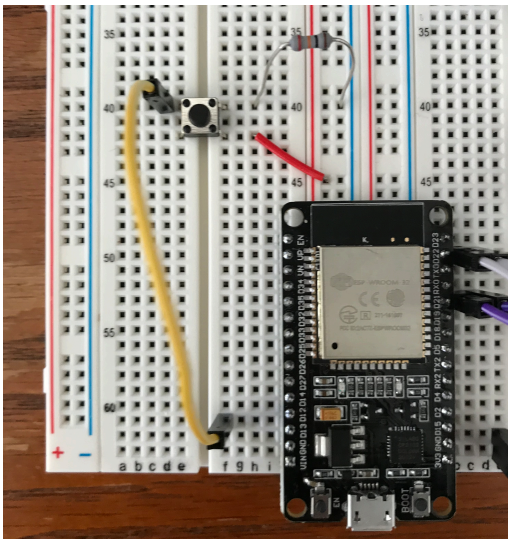

Figure 3(h): Wire the button.

9. **Wire another button**. Repeat steps 7 and 8 above, but connect to Pin 13 instead.

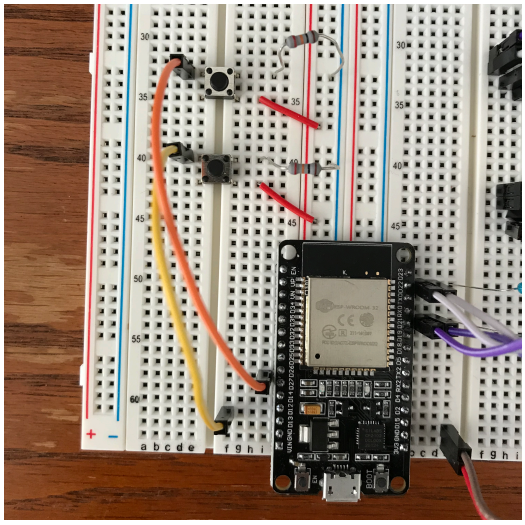

Figure 3(i): Wire a second button.

10. **Place a transistor.** Bend the pins of the transistor so that they can all fit into the breadboard. The "emitter" is the pin on the left if you're facing the flat side of the transistor, and the "collector" is the pin on right.

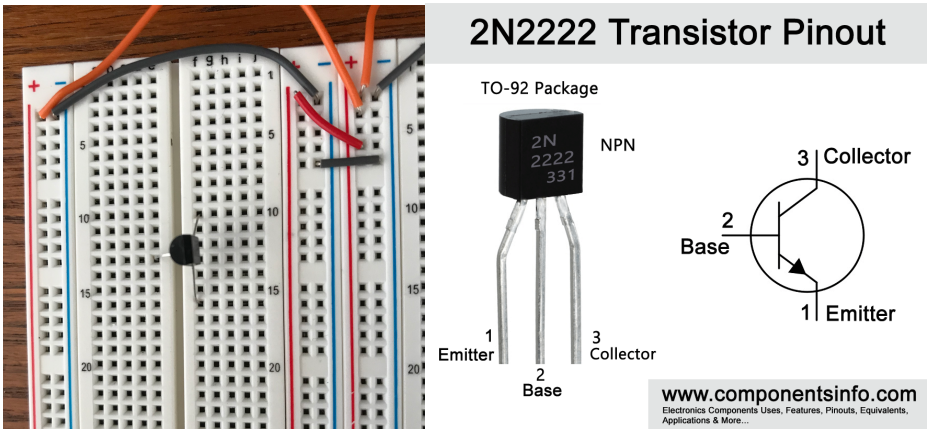

Figure 3(j): Wire a transistor and transistor pin diagram.

11. **Wire the transistor.** Connect the emitter to ground. Connect one end of a 10K Ohm resistor to power, and the other end to the collector. Connect the collector to Pin 14.

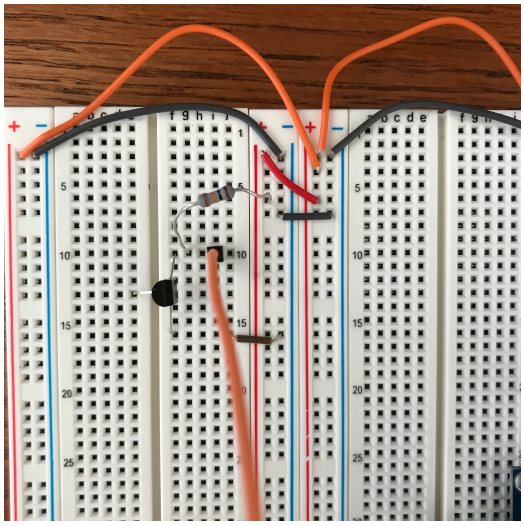

Figure 3(k): Wire the transistor.

12. **Place the IR Receiver.** Just like the OLED and BME sensor, place it horizontally.

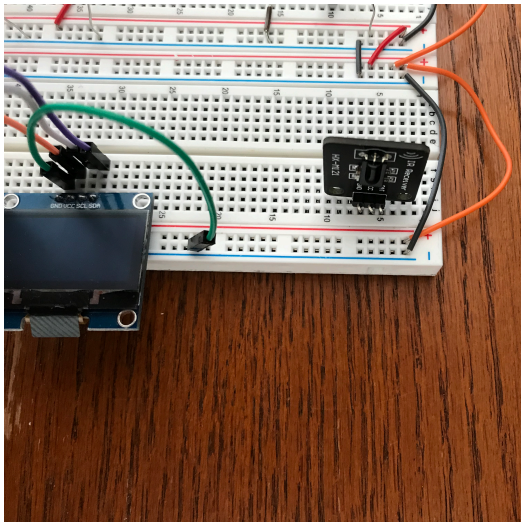

Figure 3(l): Place the IR receiver on the breadboard.

13. **Wire the IR receiver.** Connect VIN to power and GND to ground. Connect one side of a 100K Ohm resistor to the DAT pin, and the other side to the base of the transistor (the middle pin).

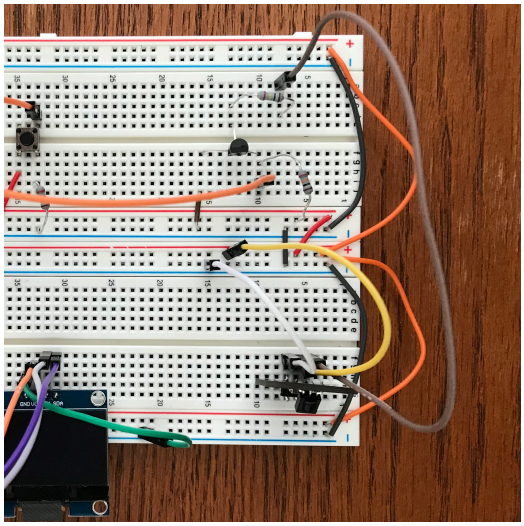

Figure 3(m): Wire the IR receiver.

14. **Check the connections**. Double-check that all the connections are correct. Your circuit should look like this, and also match the schematic:

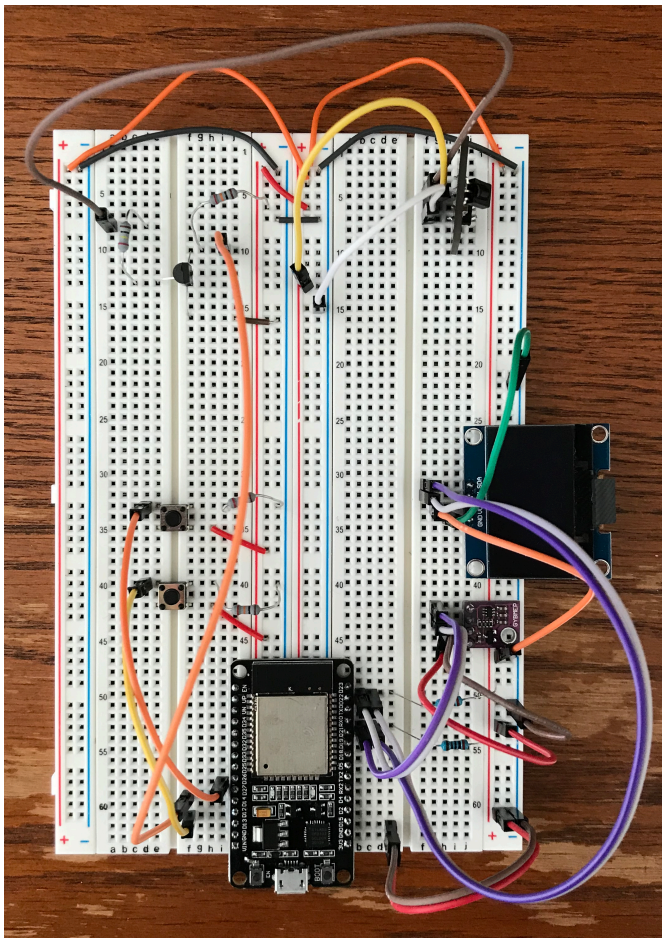

Figure 3(n): Final breadboard.

Data Receiver Assembly Guide

Now, we'll go through assembling the Data Receiver device. With the PCB, you'll simply have to place the ESP-32 onto the PCB and you're done.

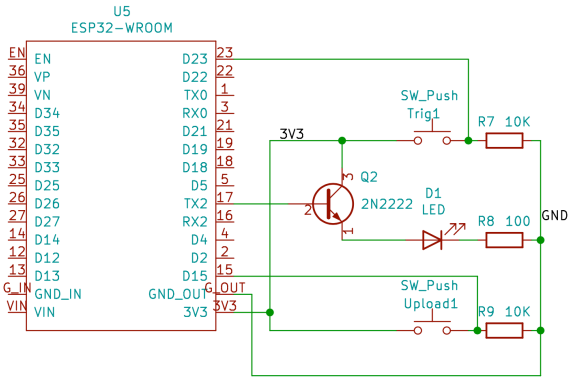

Figure 4: Receiver schematic.

Without the PCB, you'll want to follow this schematic above. Follow these steps:

1. **Put the ESP32 into one breadboard.** Only the pins on its right side (the side that includes the pins for 3V3, GND, and D23) should be plugged into the breadboard, like so. You can leave the other pins disconnected.

2. **Connect the power rails.** Connect 3V3 to the power rail and GND to the ground rail, just like before.
3. **Wire up the buttons**. Follow exactly the same steps to place and connect the buttons: connect one side to power, connect the other side to ground through a 10KΩ resistor. Connect one of the button's jumpers to pin 23, and other one to pin 16 (labeled RX2).
4. **Place a transistor**. Place a transistor on the breadboard so its three pins are connected to different pins of the breadboard. We will use this transistor to control the IR LED (unlike the last transistor).
5. **Place the LED.** The "cathode" is the side of the LED with the shorter leg, and a flat spot on the LED bulb. Place the LED so that the cathode is on the ground rail and the "anode" (the other side) is on an empty row of pins.
6. **Wire the transistor**. Connect the collector to the power rail. Connect the emitter to one side of a 100Ω resistor, and connect the other side of the resistor to the LED anode.

Software: setup

1. **Set up your Arduino IDE to use ESP-32**

Follow the instructions here to tell your Arduino application how to program the ESP-32: https://github.com/espressif/arduino-esp32/blob/master/docs/arduino-ide/boards_manager.md

2. **Install the required libraries**

In the ribbon at the top, click "Tools," then "Manage Libraries." Enter the following libraries in the search bar, and hit install:

"WiFi" (this may already be built in)
"HTTP Client" (also may already be built in)
"Adafruit BusIO"
"Adafruit Unified Sensor"
"ESP8266 and ESP32 OLED driver for SSD1306 displays"
"Adafruit BME280"

3. **Load our generic demo code**

Download the code from our GitHub repository here: https://github.com/oyetkin/droneiotclient

Open the terminal, and navigate to a directory where you want to keep the repository. Then simply type:

```
git clone https://github.com/oyetkin/droneiotserver
```

If you don't have git commands installed, you can follow these instructions to install them: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git.

Alternatively, you can just download the code manually by clicking the green Code button in the top right corner, and then clicking "Download ZIP". Unzip it after it downloads to access the code on your computer.

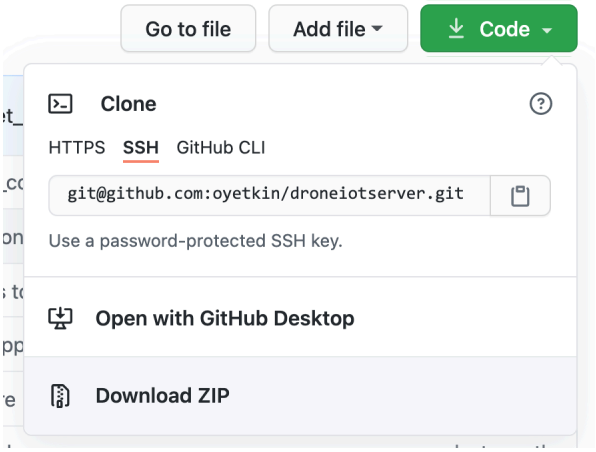

Figure 5(a): Downloading code from GitHub

4. Quick start: edit the demo code

You need to fill a few fields in the demo code based on your own device. You'll need to edit the WiFi settings, latitude and longitude, and sensor ID. Put in the name and password for your WiFi network. Give your sensor any ID you want – like your own name. You can find your latitude and longitude on Google Maps.

```
//WIFI SETTINGS
const char* ssid = "YOUR_WIFI_NETWORK_NAME";
const char* password = "YOUR_WIFI_PASSWORD";
const String sensor_name = "SENSOR_ID";
float lat = LATITUDE;
float lon = LONGITUDE;
```

Figure 5(b): Code to edit

If you want to change how often your device sleeps and wakes up, you can do it here. It governs how many seconds the device sleeps between wake-ups.

```
#define TIME_TO_SLEEP  30
```

5. Upload your code

Once you've made those edits, connect the ESP 32 to your computer using a USB cable. Then, in the Arduino IDE, go to Tools > Ports > select the USB port your device is connected to. Make sure all the settings match the ones in the image below (Board, Upload Speed, CPU Frequency, etc).

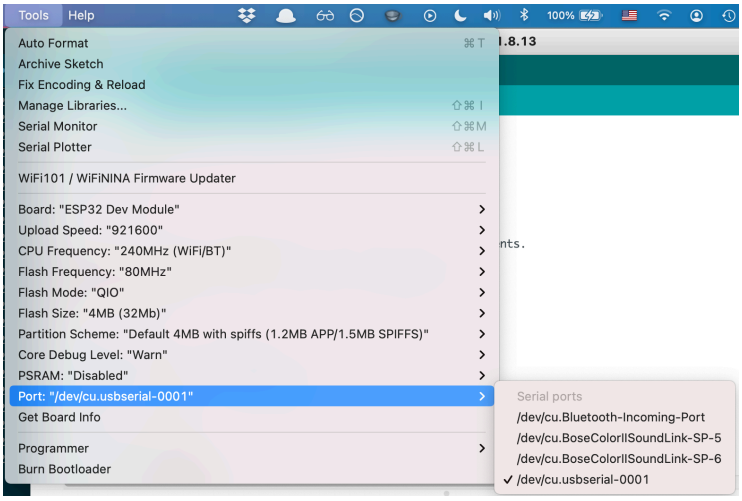

Figure 5(c): Selecting a port

Then, click the right-facing arrow icon in the top left corner. You should see a message at the bottom that your code is compiling, and then orange messages that it's writing to the ESP-32.

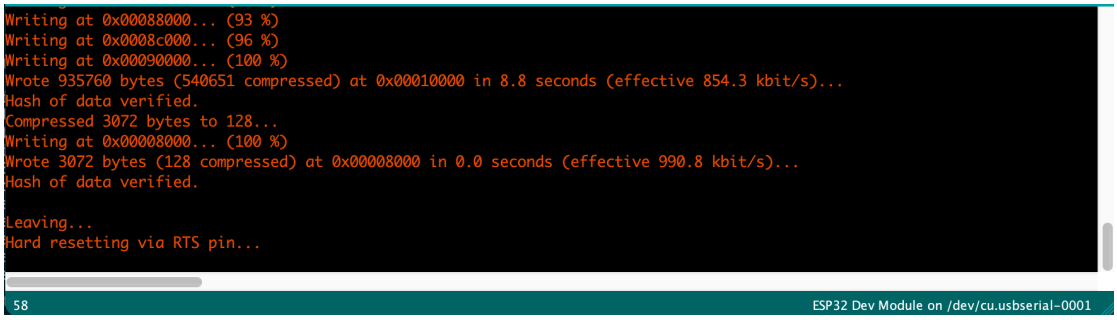

Figure 5(d): Uploading your code

6. Test the device!

Keep your device connected to power (i.e. keep the USB cable plugged in), and open the Serial Monitor by hitting the spyglass button in the top right corner. Make sure the baud rate is set to 115,200, as stated in the code. You can edit it in the bottom right corner of the Serial Monitor, as shown above.

Figure 5(e): Opening the Serial Monitor (left) and setting the baud rate (right)

Then, wait a few minutes as the device collects a few sensor measurements. You'll see what the device is doing, as well as the measurement values it has sensed, on the Serial Monitor every time it wakes up. You should see something like this:

```
17:22:43.955 -> Starting up in Remote Trigger Mode
17:22:46.076 -> Read sensors
17:22:48.069 -> Humidity (RH%): 46.11
17:22:48.069 -> Temperature (C): 21.90
17:22:48.069 -> Pressure (Pa): 97450.42
17:22:48.069 -> Going to sleep now
```

Figure 5(f): Serial Monitor as the sensor collects data

After a few minutes, press the Trigger button (the tactile push button connected to pin 13 on the ESP 32). The device should display the last few measurements as a graph on the screen. If it looks right, you officially have a working Data Logger!

We'll dive deeper into the software later. Before that, we're going to cover how to get your server up and running.

Server

In order to upload data, we need to get our web server up and running. The server accepts POST requests, in which other devices can send data to be stored on the server. The server also accommodates GET requests, in which other devices can fetch data stored on the server.

1. **Clone the server repository**

We're simply going to copy the existing open-source server code from this Github repository: https://github.com/oyetkin/droneiotserver.

2. **Install all requirements**

From the command line or Terminal, navigate to the droneiotserver/ repository, where you downloaded it. Then, type this command to install all the libraries you need to run the server:

```
pip install -r requirements.txt
```

3. **Run the server**

From the command line, simply run

```
python server.py
```

and your server will be up and running! It should be available at the URL http://localhost:8080/, since it's a local server.

4. **Check that your server is working, and let it collect some data.**

Open your web browser. Type in the URL http://localhost:8080/api/v0p1/debug/get_dallas, just like you would if you were navigating to a website. If your server is working, you should see something like this:

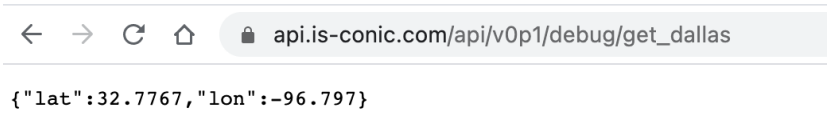

Figure 6(a): Checking that your server is working

This is a simple URL used for debugging to check that the server is online. If it is, then it's time to gather some data. You can either set your Data Logger to WiFi Mode, or use your Data Collector to collect and send measurements to the server. Wait a few minutes as your sensor collects data (how long you wait will depend on how long your sensor is sleeping in between measurements – configurable using the TIME_TO_SLEEP parameter in the Data Logger code).

You can see what data your server has managed to obtain at this URL: http://localhost:8080/api/v0p1/debug/get_data. You should see an unformatted list of sensor data, like this:


```
[{"key":"otto_stations","measurement_name":"otticity","unit":"milli_otto","value":5.0,"timestamp":1616800223,"receipt_time":1616800312.58
94532,"lat":32.7767,"lon":-96.797,"hardware":"otto"},
{"key":"Arjun_weather_kit","measurement_name":"temperature","unit":"Celsius","value":21.28,"timestamp":1617034860,"receipt_time":16170348
61.7855837,"lat":32.64,"lon":-117.1,"hardware":"BME280"},{"key":"Arjun_weather_kit","measurement_name":"humidity","unit":"Relative
%","value":54.93,"timestamp":1617034862,"receipt_time":1617034862.0948553,"lat":32.64,"lon":-117.1,"hardware":"BME280"},
{"key":"Arjun_weather_kit","measurement_name":"pressure","unit":"Pa","value":101228.0,"timestamp":1617034862,"receipt_time":1617034862.40
1102,"lat":32.64,"lon":-117.1,"hardware":"BME280"},
{"key":"Arjun_weather_kit","measurement_name":"temperature","unit":"Celsius","value":22.75,"timestamp":1617035788,"receipt_time":16170357
89.858053,"lat":32.64,"lon":-117.1,"hardware":"BME280"},{"key":"Arjun_weather_kit","measurement_name":"humidity","unit":"Relative
%","value":46.25,"timestamp":1617035790,"receipt_time":1617035790.1327689,"lat":32.64,"lon":-117.1,"hardware":"BME280"},
{"key":"Arjun_weather_kit","measurement_name":"pressure","unit":"Pa","value":101215.0,"timestamp":1617035790,"receipt_time":1617035790.44
32092,"lat":32.64,"lon":-117.1,"hardware":"BME280"},
{"key":"Arjun_weather_kit","measurement_name":"temperature","unit":"Celsius","value":22.55,"timestamp":1617035861,"receipt_time":16170358
63.2503161,"lat":32.64,"lon":-117.1,"hardware":"BME280"},{"key":"Arjun_weather_kit","measurement_name":"humidity","unit":"Relative
%","value":47.64,"timestamp":1617035863,"receipt_time":1617035863.5804358,"lat":32.64,"lon":-117.1,"hardware":"BME280"},
{"key":"Arjun_weather_kit","measurement_name":"pressure","unit":"Pa","value":101215.0,"timestamp":1617035863,"receipt_time":1617035863.88
12482,"lat":32.64,"lon":-117.1,"hardware":"BME280"},
{"key":"Arjun_weather_kit","measurement_name":"temperature","unit":"Celsius","value":22.41,"timestamp":1617035928,"receipt_time":16170359
30.5443733,"lat":32.64,"lon":-117.1,"hardware":"BME280"},{"key":"Arjun_weather_kit","measurement_name":"humidity","unit":"Relative
%","value":46.71,"timestamp":1617035930,"receipt_time":1617035930.6500914,"lat":32.64,"lon":-117.1,"hardware":"BME280"},
{"key":"Arjun_weather_kit","measurement_name":"pressure","unit":"Pa","value":101217.0,"timestamp":1617035930,"receipt_time":1617035930.83
78356,"lat":32.64,"lon":-117.1,"hardware":"BME280"},
```


Figure 6(b): Checking that your server is working

5. Set up the data visualization notebook

To quickly run our existing demonstration, navigate to the folder data_vis_demo. We're going to launch the visualization code in a Jupyter notebook. If you've never used Jupyter before, follow these instructions to install it: https://jupyterlab.readthedocs.io/en/1.2.x/getting_started/installation.html

Then launch the notebook server with:

```
jupyter lab --no-browser
```

The following messages should show up in your terminal. Click the link to launch the notebook in your browser.

```
(base) MBP-2971:data_vis_demo ArjunTambe$ jupyter lab --no-browser
[I 13:33:06.139 LabApp] JupyterLab extension loaded from /Users/ArjunTambe/opt/anaconda3/lib/python3.7/site-packages/jupy
terlab
[I 13:33:06.139 LabApp] JupyterLab application directory is /Users/ArjunTambe/opt/anaconda3/share/jupyter/lab
[I 13:33:06.142 LabApp] Serving notebooks from local directory: /Users/ArjunTambe/Dropbox/5_AMA/droneiotclient/data_vis_d
emo
[I 13:33:06.142 LabApp] The Jupyter Notebook is running at:
[I 13:33:06.143 LabApp] http://localhost:8888/?token=2efb1212788aee4aba1e05505d9e99f06c488f655bd5a478
[I 13:33:06.143 LabApp]  or http://127.0.0.1:8888/?token=2efb1212788aee4aba1e05505d9e99f06c488f655bd5a478
[I 13:33:06.143 LabApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
```

Figure 6(c): Running a Jupyter lab notebook

6. **Run the notebook to see your visualization.** You should see a new tab pop up in your browser. Click Run > Run All Cells.

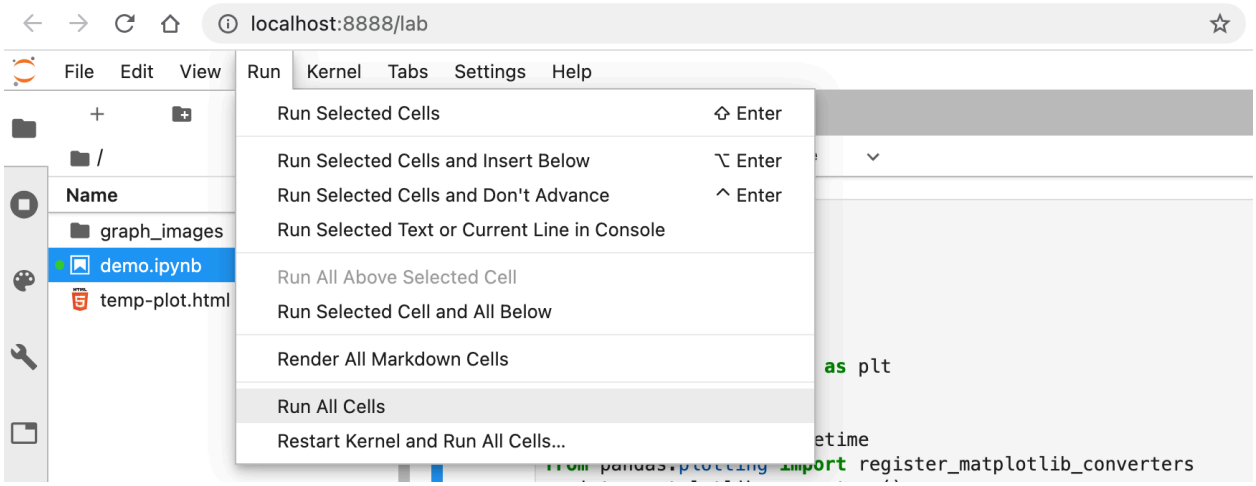

Figure 6(d): Running a Jupyter notebook

Give it some time to execute the code. You should see two visualizations pop up: one should look like a map, with blue indicators for each sensor that has sent data to the server.

Figure 6(e): Jupyter notebook sample

The second visualization should look like a contour plot, overlaid on some geography.

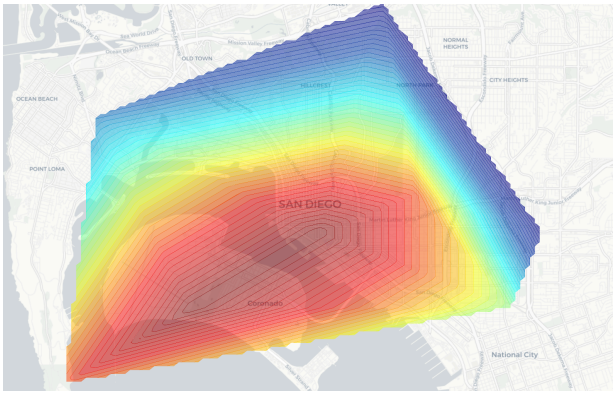

Figure 6(f): Another Jupyter notebook sample

7. Customizing your visualization

There are myriad ways to visualize your sensor data. We'll just cover how to retrieve your sensor data from the server here, and suggest some further readings on ways to visualize the data. One way is to use the requests library (https://docs.python-requests.org/en/master/) to make a GET request, and get all the sensor data.

```
import requests
header = {"Content-Type": "application/json"}
response = requests.request("GET",
  "https://localhost:8080/api/v0p1/debug/get_data",
  headers=header, verify=False)
```

Figure 6(g): Code for visualizing data

We'll check if the response was valid, and raise an exception if it wasn't. Otherwise, we'll parse it into a Pandas DataFrame—a super-convenient way of storing and manipulating data of varying types (https://pandas.pydata.org/).

```
import pandas as pd
if response.status_code != 200:
    raise Exception(response.status_code, response.text)
json_data = response.json()
df = pd.DataFrame.from_dict(json_data)
```

Figure 6(h): Code for visualizing data

The second way we can get sensor data is by using the sensor name/ID. The sensor name/ID was the parameter device_name in the Data Logger code. If we know the name of our sensor in advance, we can use another GET request to get the data from that sensor ID:

```
response = requests.request("GET", "
  https://localhost:8080/api/v0p1/sensor_by_id/{sensor_id}",
  headers=header, verify=False)
```

Figure 6(i): Code for visualizing data

You can also retrieve a list of all the sensor IDs saved on the server from the URL https://localhost:8080/api/v0p1/list_sensors.

Check out the documentation page of your server for more info.

Software: details

Let's dive into what the code is doing, step by step. We'll start with an overview of the key ideas in the code.

1. **Key design features of this code**
    a. **Remembering data when we sleep**. Our device is designed to go to sleep when it's not doing anything. When the device sleeps, any variables, arrays, or other information created in the code will be lost. However, the ESP32 has a special segment of RTC (Real-Time Clock) Memory, which stays alive even when the device is asleep. All you do to save a variable in RTC Memory is add RTC_DATA_ATTR when declaring the variable: RTC_DATA_ATTR int x; However, RTC memory is very limited (only about 4 kilobytes), and if we exceed our use of the RTC Memory buffer, our code won't compile. We have to be careful to limit our use of the RTC Memory when possible.
    b. **How we record measurement data.** To keep track of what kinds of data we're recording, we use a struct called a Measurement. We want to create an instance of a Measurement for each kind of measurement we'll be taking: for instance, one for temperature, one for humidity, and one for air pressure. The temperature Measurement object keeps track of the minimum value of the measurement (e.g. -40*C), the minimum increment the sensor is capable of recording (e.g. the sensor can detect a change of 0.1*C or greater), and the name of the measurement and unit (e.g. "temperature" and "Celsius").
    c. **Storing the data**. Because of how limited the RTC Memory is, we want to store data in the most efficient form possible. Most measurements will be given in the float data type, which costs 4 bytes. This code has a special format of representing floats as 2-byte integers. Given the minimum value and increment for a measurement (e.g. -40*C and 0.1*C), we convert any measurement into an integer number of "steps" away from the minimum. For example, 12.7*C with a min of -40*C and an increment of 0.1*C would be equal to 527 because (12.7*C – (-40*C))/0.1*C = 527.
    The code already has functions that convert between the different representations for you, so there's no need to interface with this special storage format unless you want to change the format yourself.
    One important note is that the data *is stored in the order it was recorded* with most recent measurements first. This will be relevant when the Collector figures out the timing of the measurements.
    d. **Transmitting data on ESP-Now.** The ESP-Now protocol requires us to send raw bytes. Each message, or "packet' is limited to 250 bytes, so this compact data representation is also helpful for reducing the number of packets we need to send to the receiving device. However, the receiver has to know how to convert *back*! We accomplish this by sending a "metadata" packet, which contains all the information necessary to transform the bytes into floats. As

mentioned above, we need the minimum value and resolution to do this, so the metadata packet just tells the receiver the minimum value and resolution. The metadata also contains other handy information about the sensor, which you can read more about in the technical notes at the end.

2. **The setup() function.** If you've used Arduino before, you'll notice something odd: the loop() function is empty! This is because when we put the device to sleep, the device will just run the setup() function, and sleep at the end of that function. It never reaches the loop() function at all! All the main code is inside of the loop() function.
    a. **Opening the Serial connection**. We start the Serial connection at a rate of 115200 baud. Serial is how the ESP sends data to display on our computer. We pause the code for 1 second (1000 milliseconds) to let the Serial initialize.

```
Serial.begin(115200);
delay(1000);
```

Figure 7(a): Setup function, 1

    b. **Get the reason the device woke up**, using esp_sleep_get_wakeup_cause; and get the pin number that woke us up using get_wakeup_pin_number (which will be 0 if we woke up using the timer instead of a pin).

```
esp_sleep_wakeup_cause_t wakeup_reason = esp_sleep_get_wakeup_cause();
print_wakeup_reason(wakeup_reason);
Serial.println("Mode: " + mode_to_str(device_mode_wifi));
int pin_triggered = get_wakeup_pin_number();
```

Figure 7(b): Setup function, 2

    c. **Decide what to do based on the reason we woke up.**

```
  if (pin_triggered == DEVICE_MODE_SELECT_PIN) { //The Mode Select button.
    mode_select();
  } else if (device_mode_wifi) {
    if (wakeup_reason == ESP_SLEEP_WAKEUP_EXT1) { //Either the IR Trigger,  or the Trigger Button
      if (!read_sensors()) {                      //If we fail to read the sensors, don't post and sleep instead.
        go_to_sleep();                            //Sleeping exits this code block.
      }
      post_sensors();  //If desired, you can use the bool this returns to check if posting was successful.
      display_graph();
    } else if (wakeup_reason == ESP_SLEEP_WAKEUP_TIMER) { //The timer
      if (!read_sensors()) { go_to_sleep(); }
      post_sensors();
    } else {                                      // Device booted up for the first time
      show_wakeup();
      if (!read_sensors()) { go_to_sleep(); }
      post_sensors();
    }
  } else {
    if (wakeup_reason == ESP_SLEEP_WAKEUP_EXT1) { //Either the IR Trigger,  or the Trigger Button
      ESPNowBroadcast();
      display_graph();
    } else if (wakeup_reason == ESP_SLEEP_WAKEUP_TIMER) { //The timer
      if (!read_sensors()) { go_to_sleep(); }
    } else {                                      // Device booted up for the first time
      show_wakeup();
      read_sensors();
    }
  }
  go_to_sleep();
}
```

Figure 7(c): Setup function, 3

If it was the mode select pin, allow the user to change the mode using mode_select.

Otherwise, what we do depends on the mode we're in.

In Wifi mode:

if we woke from the trigger button, read the sensors, post to the server, and show the graph

if we woke from the timer, read and post but don't show the graph

if we're waking up the first time, display that on the OLED

In Trigger mode:

if we woke from the trigger button, immediately send the data using ESP-Now, and show the graph

if we woke from the timer, just read the sensors

if we're waking up for the first time, display that on the OLED

d. **Go to sleep** at the very end, using go_to_sleep.

3. **Selecting the mode**.

```
void mode_select() {
  /*
   * Call when the device is woken by the mode select pin. Activates the OLED
   * and allows the user to select the mode. Exits after a few seconds idle.
   */
  Serial.println("Press to select mode, currently " + mode_to_str(device_mode_wifi));
  pinMode(DEVICE_MODE_SELECT_PIN, INPUT);

  init_display();
  display_mode();
  unsigned long start_time = millis();
  unsigned long debounce = millis();
  while (millis() - start_time <= MODE_SELECT_LEN) {
    //if button is pressed, change mode and restart timer
    if (digitalRead(DEVICE_MODE_SELECT_PIN) && millis() - debounce >= DEBOUNCE_TIME) {
      device_mode_wifi = 1 - device_mode_wifi;
      Serial.println("Button pressed. Mode is: " + mode_to_str(device_mode_wifi));
      display_mode();
      start_time = millis();
      debounce = start_time;
    }
  }
  //turn off the OLED display
  display.displayOff();
}
```

Figure 7(d): Mode selection

You likely won't need to edit this, so we'll keep it short. The code first starts up the display screen. It stays in the while loop, allowing the user to change the mode by pressing the button, and exits only when 4 seconds have passed since the last button press. Button presses are ignored if fewer than 400 milliseconds have passed since the last button press (otherwise, a single button press "bounces" and will count as hundreds of button presses!). If the button is pressed, we change the mode and show it on the display screen.

4. **Reading the sensors.**

a. **Take measurements.**

```
if (!bme.begin(BME_ADDR)) {
  Serial.println("Could not find a valid BME sensor!");
  return false;
}
Serial.println("Read sensors");
delay(2000); //The sensor takes 2 seconds to read after we turn it on
float curr_hum = bme.readHumidity();
float curr_temp = bme.readTemperature();
float curr_pres = bme.readPressure();
if (isnan(curr_hum) || isnan(curr_temp) || isnan(curr_pres)) {
  Serial.println("Failed to read!");
  return false;
}
```

Figure 7(e): Read sensors

The first half uses the sensor's built-in library to initialize the sensor using bme.begin(0x76), where 0x76 is the "address" of the sensor in hexadecimal. We check if the sensor successfully initialized; if it didn't, we go to sleep. Then, we read the measurements of the sensor using bme.readHumidity, readTemperature, and readPressure. We check if any of these measurements are invalid using isnan ("Is Not a Number"); if they are, we just go to sleep. Modify this part of the code if you're using another sensor.

b. Record them in our data buffer.

```
//Save all the measurements. The function handles the conversion between floats and bytes for us.
save_float_to_buffer(curr_temp, temperature, temp_data, MAX_RECORDS*RECORD_SIZE);
save_float_to_buffer(curr_hum, humidity, hum_data, MAX_RECORDS*RECORD_SIZE);
save_float_to_buffer(curr_pres, pressure, pres_data, MAX_RECORDS*RECORD_SIZE);

//Print readings to the serial
Serial.println("Humidity (RH%): " + String(curr_hum));
Serial.println("Temperature (C): " + String(curr_temp));
Serial.println("Pressure (Pa): " + String(curr_pres));

//update the last RTC reading so we'll know how long it's been since last measurement. used when sending ESP-Now data.
last_rtc_time = rtc_time_get();
//update number of measurements taken
n_cycles_recorded ++;
return true;
```

Figure 7(f): Read sensors

The code uses bytes to represent each measurement in the buffer, but you don't have to worry about that unless you're editing it. You can just use save_float_to_buffer, supplying the measurement value, the type of measurement, the array where you want to save the data, and the length of the array (which is MAX_RECORDS). At the end, increment the number of measurements we've taken, and update the time in the real-time clock (this will come in handy later!)

5. **Posting sensor data to the server**. This is fairly straightforward. First, we use connect_to_server to establish a server connection, and configTime to figure out what time it is, using Wifi. We retrieve the last measurement with last_float_from_data; the arguments are the array to draw from and the measurement type.

```
if (connect_to_server()) { //Attempt to connect to server, and proceed only if successful
  configTime(0, 0, ntpServer);

  //Now get the last measurements from the data buffers
  float temp = last_float_from_data(temp_data, temperature);
  float hum = last_float_from_data(hum_data, humidity);
  float pres = last_float_from_data(pres_data, pressure);
```

Figure 7(g): Upload data

We convert each float into a String formatted in a way that works for the server using create_post_string (see the function definition for what arguments to supply). We prepare a Batch Post by putting all the strings we need in an array. Then we combine that array into another string using multi_post_string. Finally, we call http.POST with the resulting string. We check for an error—which occurs if the response from POST is anything but 200—and close the connection using http.end().

```
String posts[3] = {create_post_string(temp, temperature, true, true, true),
                   create_post_string(hum, humidity, true, true, true),
                   create_post_string(pres, pressure, true, true, true)};
Serial.println(multi_post_string(posts, 3));
int response = http.POST(multi_post_string(posts, 3));
if (response != 200) {
  Serial.println("HTTP Post error");
}
http.end();
```

Figure 7(h): Upload data

6. **Sending data on ESP Now.** There are two methods for this, utilizing different ways of sending data. The first, which is the default, uses the broadcast address – this sends data to every device like a broadcast. The advantage is that we don't need to know the listener's MAC address in order to send the data.

```
void ESPNowBroadcast() {
  /*
   * Send ESP Now data as a broadcast. This has the advantage that we
   * don't need to know the receiving device's MAC address beforehand.
   */
  InitESPNow();
  esp_now_add_peer(&listener);
  //Send each of the 3 data buffers.
  sendData(temp_data, temperature, 1);
  sendData(hum_data, humidity, 2);
  sendData(pres_data, pressure, 3);
}
```

Figure 7(i): ESP Now, 1

All it does is initialize ESP-Now, add the broadcast address (saved in the variable listener; the broadcast address is just FF:FF:FF:FF:FF:FF), and send the data.

The second option is to send to a specific MAC address. You need to get your Collector's MAC address to do that. (see here: https://randomnerdtutorials.com/get-change-esp32-esp8266-mac-address-arduino/#:~:text=Change%20ESP32%20MAC%20Address%20(Arduino%20IDE)&text=You%20can%20set%20a%20custom,old%20and%20new%20MAC%20Address.)

Save the MAC address at the top of the code where it says esp_now_peer_info_t listener. Also make sure that the ESP_NOW_CHANNEL definition matches the CHANNEL definition in the Collector code.

```
void ESPNowToMac() {
  /*
   * Send ESP Now data to a specific, hard-coded MAC address.  This avoids the
   * time/power required to scan all wifi networks, but still checks that a device
   * is ready to listen, and can also listen for an ACK signal from the listener.
   *
   * Does not retry or handle pairing failure!
   */
  InitESPNow();
  esp_now_register_send_cb(OnDataSent); //optional, tells you the status of each packet sent.
  if (listener.channel == ESP_NOW_CHANNEL) { //check for correct channel; add peer
    Serial.print("Listener Status: ");
    esp_err_t addStatus = esp_now_add_peer(&listener);
    if (debug_ESP_error(addStatus)) { //Check if adding was successful
      //Send each of the 3 data buffers
      sendData(temp_data, temperature, 1);
      sendData(hum_data, humidity, 2);
      sendData(pres_data, pressure, 3);
    } else {
      Serial.println("Listener pair failed!");
    }
  } else {
    Serial.println("Non matching listener channel!");
  }
}
```

Figure 7(j): ESP Now, 2

The advantage of this code is that we can actually check if the data was sent successfully. The code checks that the listener's channel matches ours and that we added the listener successfully before sending the data. It also registered for a "Send Callback:" When we send the data, we'll automatically call the OnDataSent function, which checks whether sending was successful.

7. **Going to sleep.** This is an important function! First, we flush the serial and power off what we don't need.

```
Serial.println("Going to sleep now");
delay(1000);
Serial.flush();
adc_power_off(); //turn off ADC
WiFi.mode(WIFI_MODE_NULL); //turn off WiFi
esp_wifi_stop();
```

Figure 7(k): Going to sleep

Before we actually sleep, we have to tell the sensor how it will wake up next time. First, tell the device that it should wake up from the timer. The unit is microseconds; TIME_TO_SLEEP * uS_TO_S_FACTOR (which are both defined at the top of the code) tells us how many microseconds to sleep for. Next, we tell the device to wake up from a specific set of pins, or ext_1_wakeup. The set of pin is determined by WAKE_PIN_BITMASK (also defined at the top of the code). Its value is equal to $2^A + 2^B + 2^C$…, where A, B, C are the pin numbers of the pins that we want to wake the device up.

ESP_EXT_1_WAKEUP_ANY_HIGH tells the device to wake up if any of the pins are activated (the alternative is requiring all of them to be activated). Then we start deep sleep.

```
esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR); //timer wakeup
esp_sleep_enable_ext1_wakeup(WAKE_PIN_BITMASK,ESP_EXT1_WAKEUP_ANY_HIGH); //pin wakeup
esp_deep_sleep_start();
```

Figure 7(l): Going to sleep, 2

The Collector

Parts of the Collector code are fairly thorny, so read the technical notes at the bottom if you want to dig deeper. We're just going to cover the basics here.

1. **How the Collector stores data**. Recall that the Collector can't just read the Logger's data "raw." It needs extra information to decode the bytes into their actual values. The first thing the Logger sends is metadata, which tells the Collector how to decode the incoming packets.
   The Collector keeps track of an array of "DataSeries" structs. Each DataSeries stores the metadata, as well as an array of actual data. When we receive a metadata packet from a device, we store it into one of the DataSeries, and when we receive the data corresponding to that metadata, we store it in the same DataSeries. The Collector stores decoded data, not the raw bytes.
   The Collector always assumes that the first packet is metadata, and following packets are data. The packets have a special "preamble" that the Collector uses to know when we've stopped sending packets related to one type of data (e.g. temperature metadata and data) and we've started sending packets related to a new type of data (e.g. humidity metadata and data).
2. **The setup() function**. This is pretty simple.
   It just starts the Serial, and sets the relevant pin modes. Those 2 last lines are used to set up "Pulse Width Modulation," which the device uses to automatically blink the LED pins at a 38 kHz frequency.

```
void setup() {
  /*
   * Initialize the serial and configure the pinouts.
   */
  Serial.begin(115200);
  delay(1000);

  pinMode(TRIGGER_PIN, INPUT);
  pinMode(LED_PIN, OUTPUT);

  //Used for PWM; set LED_FREQ to the desired freuqency
  ledcSetup(PWM_CHANNEL, LED_FREQ, DUTY_CYCLE_RES);
  ledcAttachPin(LED_PIN, PWM_CHANNEL);
}
```

Figure 8(a): Collector setup function

3. **The loop() function**.

```
void loop() {
  /*
   * Waits for one of the two buttons to be pressed. The first button makes it activate the LED,
   * and listen for ESP-Now packets. The other makes it upload its data to the server.
   */
  bool trigger = false;
  bool upload = false;
  while(!trigger && !upload) {              //stay in this loop until one of the buttons is pressed
    delay(10);
    trigger = digitalRead(TRIGGER_PIN);
    upload = digitalRead(UPLOAD_PIN);
  }
  if (trigger) {       // if it was the trigger button, then prepare for ESP Now and fire the IR LED
    Serial.println("Trigger pressed!");
    //When data is received, it calls the callback function "OnDataRecv."
    setUpESPNow();
    sendIRTrigger();
  } else if (upload) {       //If it was the upload button, connect to the wifi and upload your data
    Serial.println("Upload pressed!");
    uploadData();
  }
}
```

Figure 8(b): Collector loop function

The control just sits inside of the “while” loop until either of the buttons are pressed. It then decides what to do based on which button was pressed. If it’s the trigger button, it prepares to receive ESP-Now transmissions, and activates the trigger LED. Otherwise, it uploads its data.

4. **Activating the trigger LED.**

```
void sendIRTrigger() {
  /*
   * Simply turn the LED on, and turn it back off after a few seconds
   * The PWM frequency of the LED is configured in setup(), and must match
   * the hardware of the data logger's IR receiver.
   */
  //Write a 50% duty cycle (a square wave that's HIGH half the time) by dividing the
  //maximum resolution value (2^DUTY_CYCLE_RES) by 2.
  ledcWrite(PWM_CHANNEL, pow(2, DUTY_CYCLE_RES-1));
  delay(LED_ON_TIME);
  ledcWrite(PWM_CHANNEL, 0); //write 0% duty cycle, aka off
}
```

Figure 8(c): Sending IR trigger

Activating the IR trigger just requires us to write to the same PWM channel as we used in the setup() function. The second argument is the “duty cycle,” or the fraction of time each pulse spends HIGH. A duty cycle of 0% means the device is fully off. We’re shooting for a 50% duty cycle (though the specific value isn’t actually that important). If DUTY_CYCLE_RES is 8, then the maximum value is 2^8, or 256, so a value of 128 is a 50% duty cycle. Therefore, $2^{\text{DUTY\_CYCLE\_RES-1}}$ gives us a 50% duty cycle.

5. **Uploading data.**

```
void uploadData() {
  /*
   * Upload all data to the server. Iterates through each of the series we have,
   * and posts each of the records for that series.
   */
  if (connect_to_server()) {
    for (int i=0; i<n_series_received; i++) {
      Serial.println("Posting records for " + String(all_data_series[i].series_id));
      post_all_records(&(all_data_series[i]));
    }
    http.end();
  }
}
```

Figure 8(d): Uploading data

All we do here is connect to the server, then start posting all the records. The records are each posted individually, one by one.

6. Receiving data.

```
void OnDataRecv(const uint8_t *mac_addr, const uint8_t *data, int data_len) {
  /*
   * Called as an interrupt whenever data is received on ESP Now.
   */
  ledcWrite(PWM_CHANNEL, 0); //stop firing the IR LED once the other ESP sends data
  String series_id = get_series_id(mac_addr, data); //convert mac address and the top of the data into the Series ID
  Serial.print("Packet Recv from: "); Serial.println(series_id);

  //Get the index in the all_series array of this series id. If it's not in the array, the index will be -1
  int idx = find_series_by_id(series_id);
  if (idx == -1) {                                        //if not found, it is the first packet for a new series, so it's metadata
    Serial.println("Unpack sensor metadata");
    decodeMetaData(data, series_id);
    n_series_received += 1; //increment counter of series received only if it's a new series; and must be after decode metadata
  } else if (all_data_series[idx].n_records_recd >= MAX_RECORDS) { //if we've seen this series id, but the number of records is full,
    Serial.println("Buffer full from this ID; overwriting");        // then treat it as a new metdata and overwrite the old data.
    decodeMetaData(data, series_id);
    all_data_series[idx].n_records_recd = 0;                        //reset the number of records to 0
  } else {                                                          //If it's found, we now have regular data, so save it in the array
    save_series_in_series_array(idx, data, data_len);
  }
}
```

Figure 8(e): Receiving data

The first thing we do is turn off the LED, since we don't need it anymore.

We get a Series ID—this is a string used to tell when two packets are part of the same set of measurements or different ones. We look up that Series ID using the find_series_by_id function.

If it's not found, we get a value of -1: this means we've never seen this Series ID before. In this case, we'll treat the packet as metadata, decode it, and save the Series ID, so we'll know where to look next time.

If it is found, then we know we have metadata for it, because we recorded the metadata the first time we saw this Series ID. In this case, we call save_series_in_series_array to store the data. This function uses the stored metadata to decode the packet.

The special case is if it is found, but the associated Series is full (i.e. the number of records received, stored in n_records_recd, exceeds the MAX_RECORDS for a Series). In this case, we're going to start fresh, and treat it as metadata again. We'll overwrite the old metadata, and reset the number of records received. Then, the next packet we'll get from the same Series will be treated as regular data and will overwrite the old data.

Troubleshooting and FAQs

Software

My code isn't uploading.

Make sure your USB cord is plugged in to your laptop, and to the ESP 32. A red LED on the ESP 32 should be turned on while it is plugged in. If it's off, your ESP 32 is not connected properly.

Make sure that you selected the correct port. Click Tools > Ports and try selecting a different port if your code isn't uploading.

I can't upload the sketch for the first time, or the device keeps crashing when I run it for the first time.

This is a common error relating to the device's flash memory. To fix it, you just have to erase the device's flash memory, and then you should be able to program it without any difficulty.

Navigate to your esptool.py file, which should be located in a path like this:

```
/Users/<YourName>/Library/Arduino15/packages/esp32/esptool.py
```

Then, open a Terminal window, and run this command:

```
python esptool.py --chip esp32 --port /dev/cu.usbserial-0001 --
  baud 115200 --before default_reset --after hard_reset
  erase_flash
```

Make sure to edit usbserial-0001 to the port your board is connected to (you can see the port names if you click Tools > Port in the Arduino IDE).

For more details, take a look at this link: https://github.com/espressif/esptool/issues/348

When I'm uploading code, I get an error that the device "timed out waiting for packet header."

This is a tricky error, but fortunately it's easy to fix. Just unplug your USB from your computer, and plug it back in again, then try uploading again.

The Serial Monitor is really wonky – it's printing slowly and lines get cut off.

This is not uncommon. Use the classic fix – unplug it, plug it back in again, and re-upload your code.

My Serial Monitor is posting a bunch of weird symbols like backwards question marks.

Make sure your baud rate is set to 115,200. The first line in the setup() function defines the required baud rate as 115,200. You can modify this if you want, but make sure your Serial Monitor is set to match the rate written in the code.

Technical notes, and advanced stuff

Hardware

1. **Connect batteries (optional)**

After you've uploaded code to the device, you can get the device to run on battery power. The top of the PCB has exposed gold-colored pads where power from a battery pack flows in. Put your batteries in the battery pack, and clip or solder the wires from the battery pack to the gold-colored corner pads.

Make sure the positive end of the battery pack connects to the pad labeled 5V IN, and the negative ends connects to 5V GD! If you switch these, you'll destroy the ESP 32!

2. **Wiring up different I2C devices**

Let's say you want to use your own sensor instead of the BME 280. You can just use an I2C device, and connect the device to the same I2C pins (Pins 21 and 22 on the ESP32, or the same set of header pins on our PCB). If you're using the PCB, make sure that the position of all the pins (SDA, SCL, VIN, GND) on your device matches the position of the pins on the PCB! If not, you can use jumper cables to rearrange the pins.

3. **Using analog or SPI devices or something else.** On our PCB, all of the pins of the ESP are available for your use. You can use any of the open through-hole slots to solder in a wire. Be mindful of which pins are already in use! Pins 21 and 22 are used for I2C and Pins 13, 14, and 27 are used as trigger pins; the rest are up for grabs. The same is true of the breadboard. Take a look at this diagram for help deciding which pins to use for your purposes:

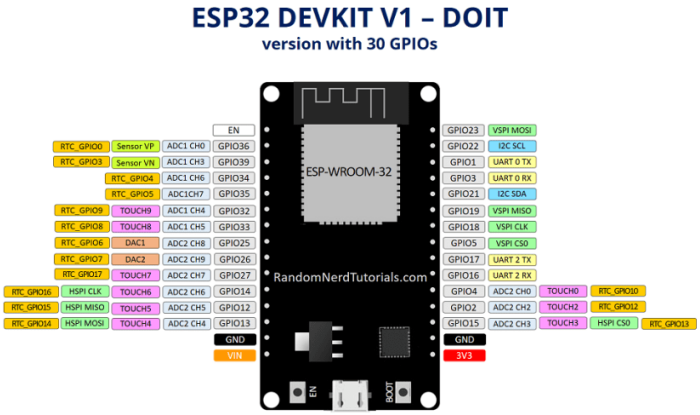

Figure 9: ESP 32 pinouts. Source: https://randomnerdtutorials.com/getting-started-with-esp32/

You'll have to read the device specifications/pinouts and connect them to the ESP's pins accordingly.

Software

1. **Detailed description of the transmission protocol**
    a. **Details of the metadata packet.** The metadata packet has 10 key elements: (1) a 2-byte preamble; (2) a 32-character or less sensor name; (3) a float for the longitude; (4) a float for the latitude; (5) a 32-character or less measurement name; (6) a float minimum value for the measurement; (7) the float resolution for the measurement; (8) the 32-character hardware name; (9) number of packets to be sent; (10) the time interval between measurements; and (11) the time since the last measurement was taken. All data of the same measurement type is sent

in a series of consecutive packets, and data of a new measurement type is sent with new metadata afterwards, like this:

<Packet 1: Temperature Metadata> <2: Bytes 0-248 of temperature><3: Bytes 249-496 of temperature> <4: Humidity Metadata><5: Bytes 0-248 of humidity><6: Bytes 249-496 of humidity>…

b. **How the Collector knows which packet is which**.
    i. The preamble is used to distinguish between packets from different sensors or of different measurements. The packet's MAC Address, plus the Preamble, is a "Series ID." All packets from the same Series will have the same Series ID. So, all packets relating to Temperature data from my sensor will have one Series ID, all the packets relating to Humidity data from my sensor will have a second Series ID, and so on.
    ii. The Collector first identifies the Series ID of a packet. If the Collector has never received data from this Series ID, it'll treat the packet as metadata, and record it in a new DataSeries slot. If the Collector has received data from this Series ID before, it'll treat the packet as data, use the associated metadata to decode it, and save it. The Series ID works like a "key" in a dictionary, telling the device where to find the metadata to decode the data, and where to store the data.
    iii. The DataSeries also track how much data the Collector has received so far. So, if we receive a third packet with an already-known Series ID, we can tell we've already received 124 records (the first packet was metadata, and the second packet contained 248 bytes of data, which is 124 records), so the data from this packet gets placed in the 125th record and onwards.
    iv. There's one special case. The data in a DataSeries can be full: for instance, suppose we expected at most 200 records, and we already received 200 records (across 2 packets). If we receive *another* packet from the same Series ID, we'll start fresh—we'll treat that packet as metadata, overwriting the old metadata, and treat subsequent packets as data, using the *new* metadata to decode it and overwriting the old data.
    v. Packet losses are not currently handled! This can result in misalignment: if we miss a packet, and then receive a new transmission of the same data again, the Collector will treat the second metadata packet as the last, missing packet, which could lead to problems.
c. **How the Collector deduces the times that data was read**. The metadata packet includes two pieces of timing information: the interval in between measurements, and the time since the last measurement was taken. If the Collector has access to an accurate clock, it uses this to infer the actual values of the measurements, and post those measurements directly:

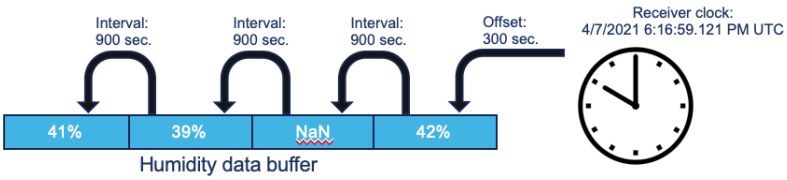

Figure 10: Timing

The poses some problems; we outline a method to correct them in the final section, "Development Ideas."

2. **Editing the data storage method.** We've already covered how to create different kinds of measurements. But the protocol we have in the code has its limitations: it can only capture a range of about 65,000 unique values. Let's say you're using a measurement with even more precision than that. Or, perhaps, your measurement is best represented as a base and an exponent, instead of a minimum value plus some number of increments. To capture this, you have a few different options.
    a. **Keep the TwoByte struct, and edit the conversion to and from floats**. Currently, the short_to_float and float_to_short functions assume we are using the TwoByte to represent the number of increments between the measurement's minimum value and the measured value. To change the conversion, edit these two functions. The key idea is that *TwoByte is just two bytes: one high byte, and one low byte.* You can convert between floats and a pair of two bytes in any way you want. Here's an example using scientific notation, which would represent 5,810 as 5.81×10^3 (you might have seen this on calculators as 5.81E3). We'll use one byte for the significand (5.81) and one byte for the exponent (3). (This is actually how floats are already represented, but our version manually converts down to 2 bytes instead of 4!)

```
float scientific_short_to_float(SplitShort s) {
  uint8_t factor = s.high;
  uint8_t exponent = s.low;
  return factor*pow(10, exponent);
}
```

To convert back:

```
SplitShort float_to_scientific_short(float v) {
   uint8_t exponent = 0;
   while (v_next >= 10) {
    v = v/10;
    exponent += 1;
   }
   uint8_t factor = (v - 1)/(10.0/256.0);
   return {factor, exponent};
}
```

Figure 11: Sample conversion between different formats

    b. **Abandon the TwoByte struct entirely**. If you don't want to use TwoByte, and you'd rather use original data types like floats and ints, you'll have to modify the code in a few places.

- Change the types of the arrays storing the data
- Change the read_sensors function to store in these news arrays directly, instead of converting types first
- Change post_sensors to take data from the new arrays instead of the TwoByte-compatible arrays
- The tricky part: edit sendDataMulti, which is used by the ESPNow functions to send data. The library function esp_now_send requires us to use a byte (uint8_t) array. If you're using a different type, you'll have to use memcpy to directly copy the bytes from your other data type into a uint8_t type array, and send that second array.

Future Development work

There are a lot of things that can be done to make this device work better. Here are some ideas:

Determining when measurements were taken

1. Currently, the Data Logger tells the Data Collector the time interval between its data, and the time since the last measurement. The Data Collector can use this to figure out what time each of the data points were recorded. However, this only works if you never miss a measurement. If the Logger fails to record a measurement today, it will record tomorrow's measurement in the spot meant for today, which messes up the timing. To fix this, you could record a "gap" or "blank" whenever you miss a measurement. To do this, you'd have to edit the data protocol to reserve a specific byte pattern as NaN. For instance, you could write 0xFFFF (which is currently the maximum value of a measurement) whenever you want to write a "blank," and make sure the Data Collector knows that 0xFFFF means "blank," not the max value.

Batch posting

1. The code to batch post is currently very clunky. You post a list of measurements that all contain a ton of redundant information, like this:

[{key: john's sensor, value: 12.4, measurement_name: temperature, unit: Celsius},
{key: john's sensor, value: 15.4, measurement_name: temperature, unit: Celsius}, …]

The majority of the fields stay the same across every member of the batch! A more efficient way to implement would be to use a single dictionary, containing key-value pairs for all of the fields except for "value." "value" would be replaced by a "data" key, and it would map to a list of floats.

2. Currently, the Collector deduces the times that the Logger made measurements using the metadata. This may not be accurate, so end users should have a way of knowing whether the time-stamps represent actual, accurate times, or times that we deduced from the metadata. Also, the server should do the reconstruction, rather than the Collector. To that end, and in line with (1) above, we suggest that the server accommodate batch posts that look like this:

```
{key="my_sensor_name" (String),
measurement_name="temperature" (String),
unit="Celsius" (String),
```

```
hardware="BME280" (String, optional),
time_interval=x (int, seconds),
time_offset=y (int, seconds),
collection_time=z (int, Unix time),
data=[12.4, 12.5, 13.8,…]}
```

Then, the server should "ingest" the batch post by:

- Separating the batch post into a list of individual dicts, where each dict contains all the information on the measurement.
- Reconstructing the time of each measurement. It assumes that the data are in sequential order. So if len(data) = N, then data[N - 1] has the time collection_time - time_offset; and data[N – i] has the time collection_time - time_offset - (i * time_interval).
- Adding a field that's True if the time was reconstructed (i.e. True for batch posts, False if the time was included in an individual post as usual)
- Also adding a field with the 3 reconstruction parameters, time_interval, time_offset, collection_time (in case we want to make corrections or something to it)

Data from various sources should all be placed in the same storage format, to minimize the burden for end-users. Ingesting batch-posted data in this way allows it to have the same “shape” as individually-posted data, so the two can be organized side-by-side in the same format, rather than having to separate them.

Deployability

If the device loses power, it'll lose all its memory. This is a weakness for applications where the sensor needs to log data in harsh conditions (such as low temperature or being moved around or dropped). You could write to PROGMEM instead of the RTC memory.

Power saving

1. The device currently works on an ESP-32 Dev Board. The board has a voltage regulator for 5V --> 3V3, and an LED that's always on. That takes a whole lot of power! To achieve the low-power modes this design is meant for, you can redesign the device to work on the bare ESP32 chip instead of the Dev module.
2. In this vein, you could use a transistor, controlled by the ESP32, to control power to other devices. Before putting the ESP32 to sleep, you'd switch the transistor off so that the other devices don't consume any power when not needed. You'd want to pull the base low and use a bigger resistor to limit the current drain associated with the transistor itself.
3. The OLED and the Serial Monitor are slow and power-inefficient. You could remove the print statements and delay()s associated with using these displays in order to spend less time and power awake.