Water Weather Station Board Setup Guide

Ian Donovan, Dylan Gappa

Table of Contents

1 Introduction

1.1 Purpose

This is an internal development document designed to describe the current state of the prototype board and to assist with the initial setup of the device. It covers the assembly, environment, integration, and physical information of the development prototype and is intended for internal project use only. Please note that this document only instructs about the board prototype, and does not describe the procedure for assembling the buoy itself or for developing project backend.

Before reading: Please read the “README” file on the documentation repository! This document only covers project aspects directly related to the prototype board. For more information on the database and android application, docker, git, and many other project areas, refer to the README and other documentation resources in the project gitlab.

1.2 Acknowledgements

This document contains information from the original Water Weather Station Documentation file. This original document was compiled in parts by Grant Barton, Jacob DeBoer, Karina Sandlin, Ryan Lingg, Sara Morimoto, Daniel Shtunyuk, Elana Cueto, Erik Fretheim, Harry Saliba, Morgan Stimpson, Paul Haithcock, Vipul Kumar, Alex Wilson, Jay Hechter, Douglas Woods, and Jessica Avery.

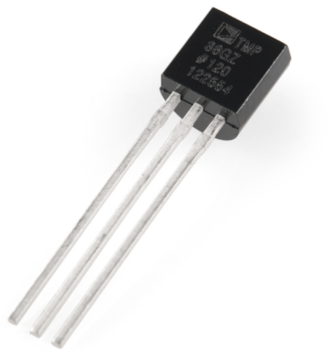
2 Components

2.1 Components List

Component - Model	Quantity
Microcontroller - SparkFun ESP32 Thing	1
Photodiode - SFH 2505-Z	1
Temperature Sensor - TMP36GT9Z	3
Turbidity Sensor - TSD-10	1
Potentiometer - 100 Kohm	1
M-to-F Jumper Wires	3
Prototyping Wire	As much as you need.

2.2 Sensor Information

Temperature:



Sensor model: TMP36GT9Z

Manufacturer: Analog Devices Inc.

Input channel: TEMP1_CHANNEL, TEMP2_CHANNEL, TEMP3_CHANNEL

Attenuation value: ADC_ATTEN_DB_6

Unit conversion formula: $Degrees\ Celsius\ =\ (Voltage\ (mV)\ -\ 500)\ /\ 10$

Implementation state: Implemented and tested.

Light:

Sensor model: GM5539 Photoresistor

Manufacturer: Wodeyijia

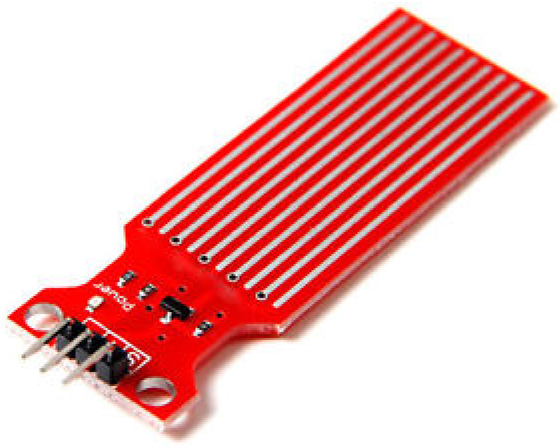
Input channel: LIGHT_CHANNEL

Attenuation value: ADC_ATTEN_DB_11

Unit conversion formula: $Watts/M^2 = 0.005167 * Voltage\,(mV) + 0.7315$

Implementation state: This sensor is **not** calibrated against sunlight, so this conversion formula only works for LED light. It represents an example of the final equation and is not implemented into the board. For the sensor to work with sunlight, the reference resistance must be increased and the sensor must be calibrated against sunlight.

Salinity (Water):



Sensor model: Water Level Sensor

Manufacturer: ARCELI

Input channel: SALINITY_CHANNEL

Attenuation value: ADC_ATTEN_DB_6

Unit conversion formulas: *none*

Implementation state: Unimplemented.

Turbidity:

Sensor model: TSD-10

Manufacturer: Amphenol Advanced Sensors

Input channel: TURBD_CHANNEL

Attenuation value: ADC_ATTEN_DB_6

Unit conversion formula:

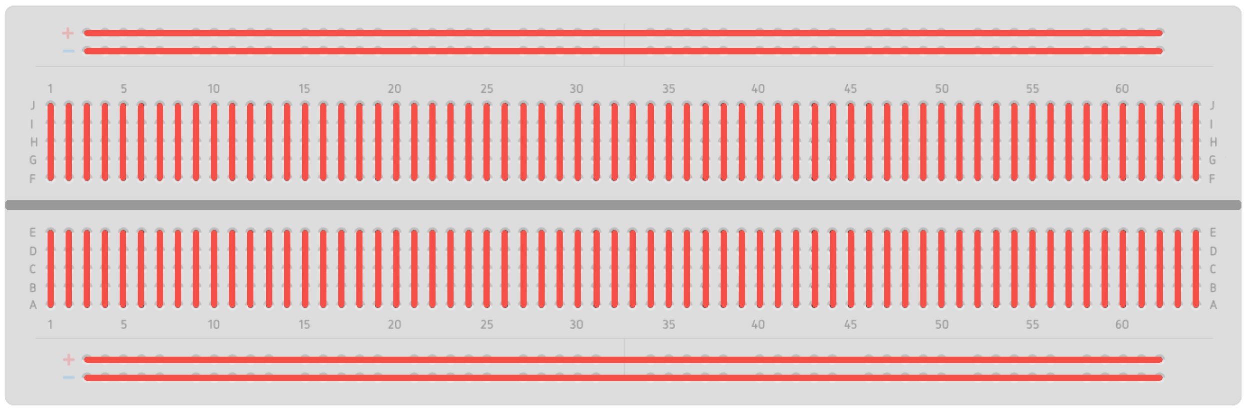
$$Turbidity\ (NTU)\ =\ -\ 1120.4(Voltage\ (mV)^{2})\ +\ 5742.3(Voltage\ (mV))\ -\ 4352.9$$

Implementation state: Implemented. Note that this formula was not empirically derived, and is a temporary implementation. In the future, it would be best to determine our own formula based on our own sensors. This formula was retrieved from documentation on our sensor, and was derived using a TSD-10.

3 Board Assembly

3.1 Using a Breadboard

Breadboards operate in sections, with strips of conductive metal running vertically through the numbered columns and horizontally through the positive and negative rails. You can think of a breadboard like a collection of wires organized like so:



Any charge flowing through a wire connected to one of these ports will be carried across the entire column, and can be picked up by connecting a wire to another port.

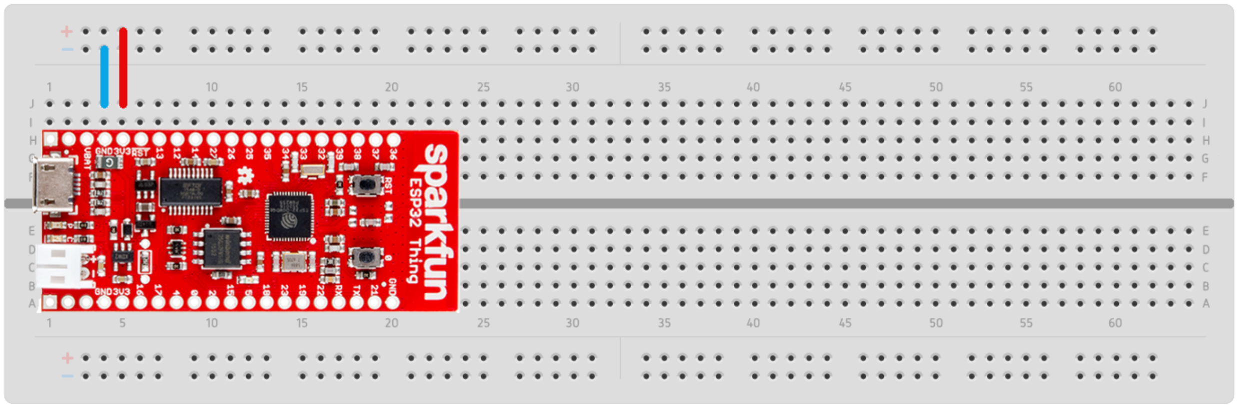
All breadboards are layed out a little differently, but ours is indexed from the bottom-left port, meaning the bottom-left port is port 1A. If your breadboard is labeled differently, don't worry. As long as the components are connected to power, ground, and their respective data channels correctly, the specific port doesn't matter.

Be careful to keep track of which rail is positive, and which rail is negative. Our breadboards are vertically aligned, but are rotated to be horizontal for these instructions. This means our positive rail is on the top, while our negative rail is on the bottom.

3.2 Building the Prototype

For these instructions, colored wires correspond to their purpose: **red** wires carry power from the positive rail, **blue** wires connect back to ground, and **green** wires transmit data as voltage back to our ESP32. These instructions will give a step-by-step method of assembling the board, but as long as the right pins are connected to the right components, you can organize your development board however you like.

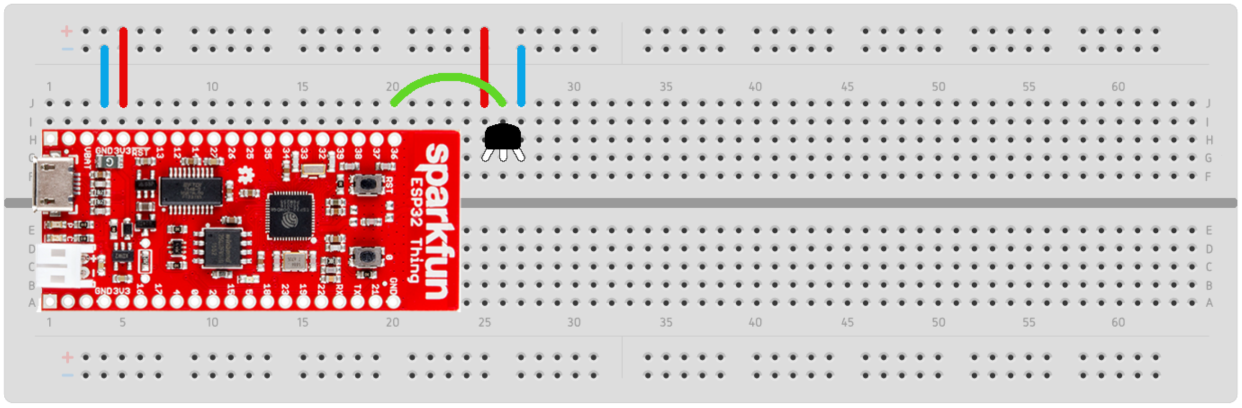
Step 1 - Power:



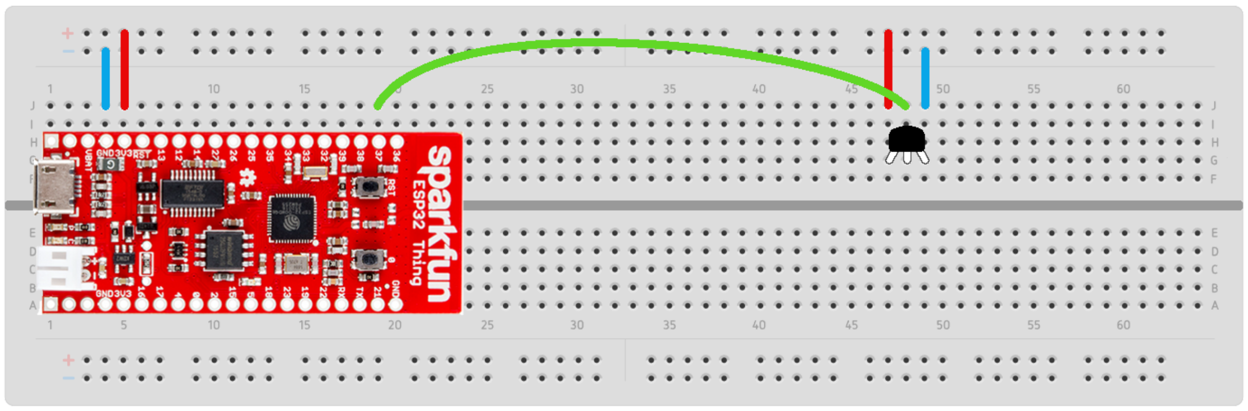
First, make sure your ESP32 board is situated properly. The bottom-left pin of the board should be connected to the 1st column in row A. The board should take up columns 1 through 20, and should cover rows A through H.

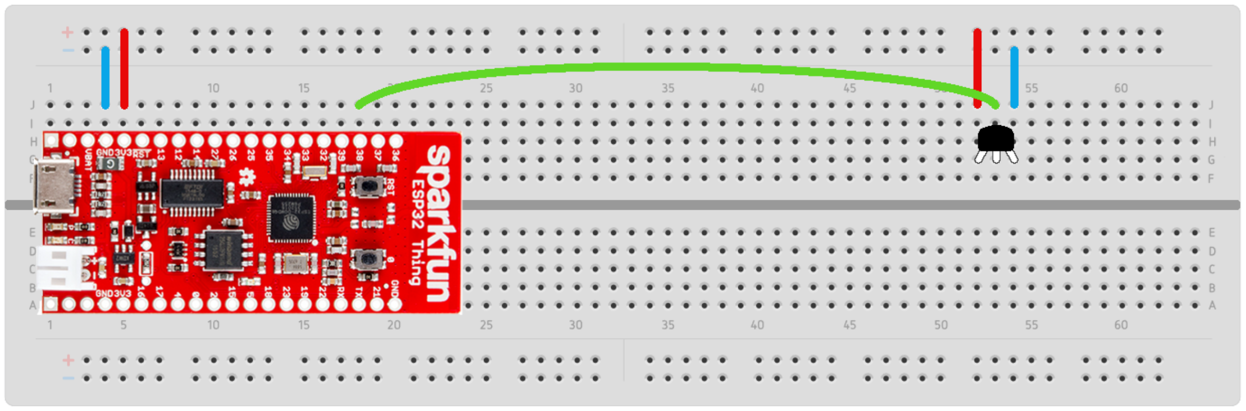
Next, connect column 4 to the negative rail and column 5 to the positive rail. This ensures all of the components connected to these rails can receive power. The ESP32 pin labeled GND, which is our ground, should be connected to the negative rails, and the ESP32 pin labeled 3v3, which is our power, should be connected to the positive rail.

Step 2 - Temperature:



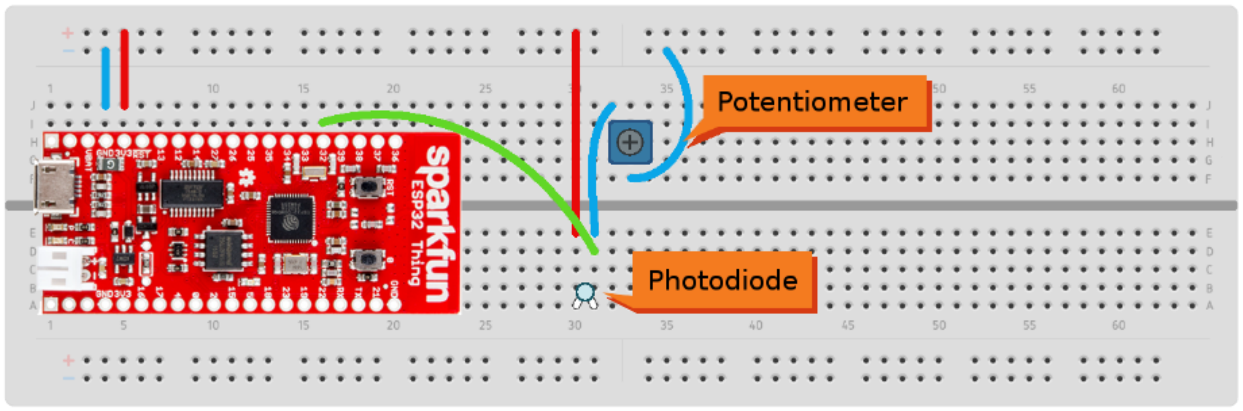
Next, we will assemble our first temperature sensor. Insert the temperature sensor into columns 25, 26, and 27 so the flat side of the sensor is facing the middle of the board. From this orientation, the left pin of the sensor is power, the right pin is ground, and the middle pin is data, or output. Connect the output of the temperature sensor to **pin 36** of the ESP32, which should be connected to column 20 of the breadboard.



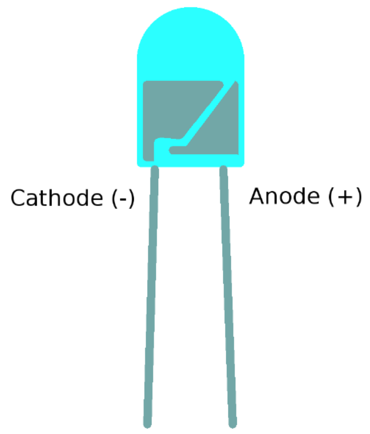


Now, we'll add the second and third temperature sensors elsewhere on the board. It's good to keep them a little bit apart so we can test localized heat sources without heating up all of the sensors at once. Insert the second sensor into columns 47, 48, and 49, and the third sensor into columns 52, 53, and 54. The data pin of the second sensor should be connected to **pin 37** on the ESP32, and the data pin of the third sensor should be connected to **pin 38**.

Step 3 - Light:



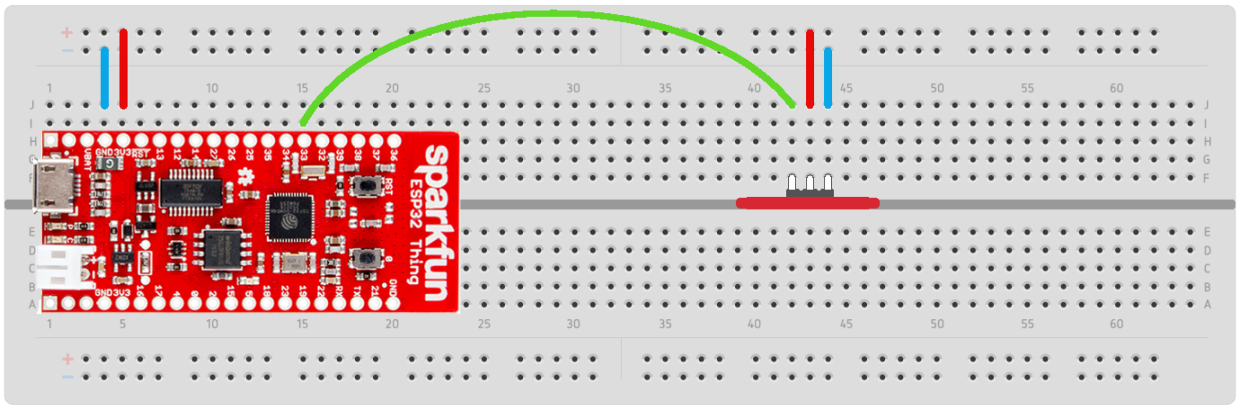
Now we'll assemble our light sensor. This sensor has two components: a photodiode and a variable resistor (a potentiometer). The photodiode should be far away from other components as to not be obstructed by wires and to allow for a diffuser to be placed over it. Insert the photodiode into columns 30 and 31, and pay close attention to the orientation of the diode.



The photodiode is running in **reverse bias**, which means the cathode (the negative pin) is connected to the positive power rail, and the anode (the positive pin) is connected to the rest of the circuit. If you look closely at the photodiode, you will see two small metal components on the inside of the cell; the larger of the two is on the negative side of the device.

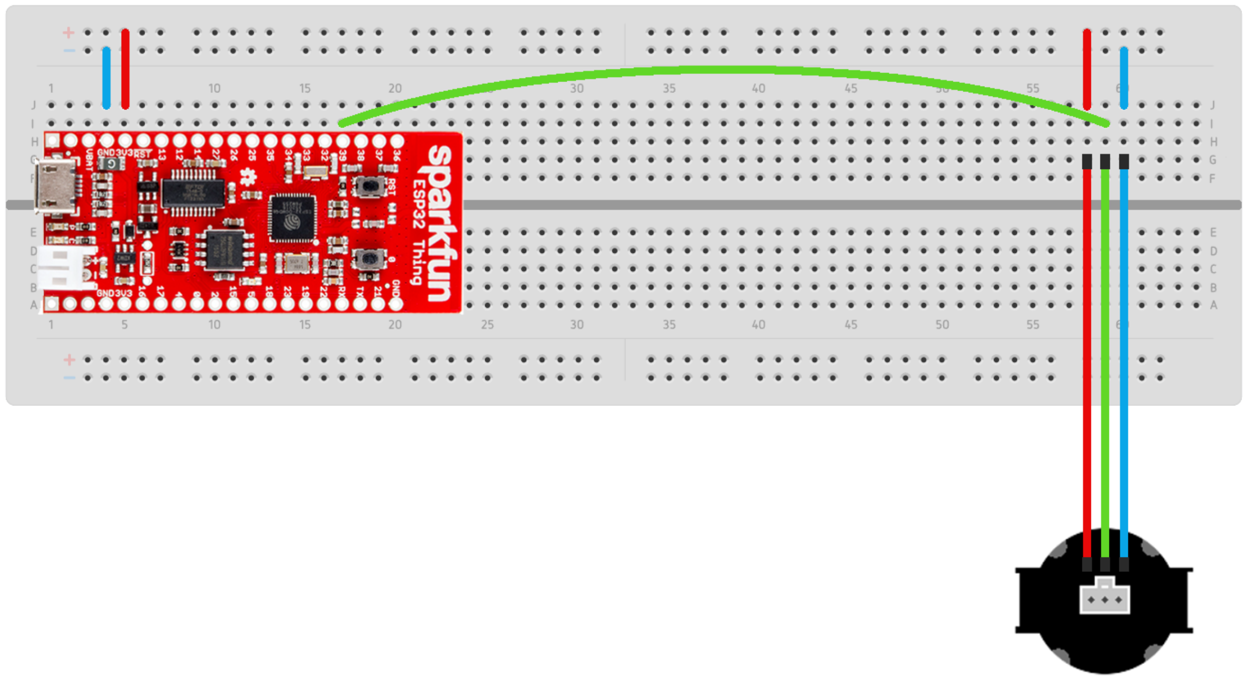
The positive pin of the photodiode (on column 31) connects first to our output, which is read by **pin 32** of the ESP32 (column 16). After that (as shown in the above diagram), another connection is made from column 31 to our potentiometer. In the future, we will have a fixed resistor, but for prototyping having a variable one is helpful. Put the potentiometer in columns 37, 36, and 37, and connect column 36 (the middle pin of the potentiometer) to the negative rail.

Step 4 - Salinity



The board's water sensor slots into the center groove of the breadboard. Insert the sensor into columns 42, 43, and 44, with the pins facing the top of the board. In this orientation, the middle pin is power, the right pin is ground, and the left pin is data. Connect the data pin (in column 41) to **pin 33** of the ESP32 in column 15, the middle pin to the power rail, and the right pin to the negative rail.

Step 5 - Turbidity



The turbidity sensor is connected to the board by a set of M-to-F jumper wires. Orienting the sensor as shown, the power pin is on the left, the data pin is in the middle, and the ground pin is on the right. Insert the jumper wires into columns 58, 59, and 60, and connect each to the appropriate sensor pin. Then, connect column 59 to **pin 39** of the ESP32, in column 17.

4 Development Environment

4.1 ESP-IDF

The Sparkfun ESP32 uses the ESP-IDF development environment by Espressif. Their comprehensive documentation page, which includes a full getting started guide, can be found here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/

4.2 Visual Studio Code

ESP-IDF can also be installed as an extension for Visual Studio Code. It can be found by searching “Espressif IDF” on the marketplace. This allows you to build, flash, and monitor from the Visual Studio Code application, and may be a good alternative to the standard installation.

4.2 Building, Flashing, and Monitoring

So the board is built and the environment is set up. What do I do once I've made my changes?

Once the code you've added is finished, the first step is to build it. In order to do so, you need to be in your local board directory, and to have run the "export.bat" file. Remember, unless you are using the ESP-IDF Command Prompt shortcut or the VSCode extension, you will have to run this bat file every time you open your terminal. In your local board directory, run

```
idf.py build
```

This command compiles and builds your code, and will fail if you have any compiler errors. Once it succeeds, you will need to flash the newly built instructions to the board. The command for this is

```
idf.py -p (PORT) flash
```

The (PORT) part of this command is the usb port ID that your board is plugged in to. The method of finding this port differs depending on your operating system, but the Espressif documentation site has instructions for Windows, Mac, and Linux. Once your code is flashed, you can test it with

```
idf.py monitor
```

Which will cause your board to start taking readings of its surroundings.

4.3 Common Issues

ESP-IDF tends to need a good deal of configuration to get working, and a long-standing project like this one can build up maintenance issues. Here are a few common issues you might run into:

My IDE can't find any of the included files.

This may be a problem with your path variable. Unless you use the ESP-IDF Command Prompt or edit your terminal's configuration file, the path variable is not saved between system restarts. This error seems to be less common in the Visual Studio Code extension method of installation.

I get the error “assertion ‘partition != NULL’ failed.”

We define our storage partition in a CSV file called partitions.csv. If you’re getting this error, chances are ESP-IDF isn’t configured to recognize this file. Run the command “**idf.py menuconfig**” and navigate to Partition Table -> Partition Table, and make sure it is set to “Custom partition table CSV.”

My board builds, but it just outputs nonsense characters when I monitor.

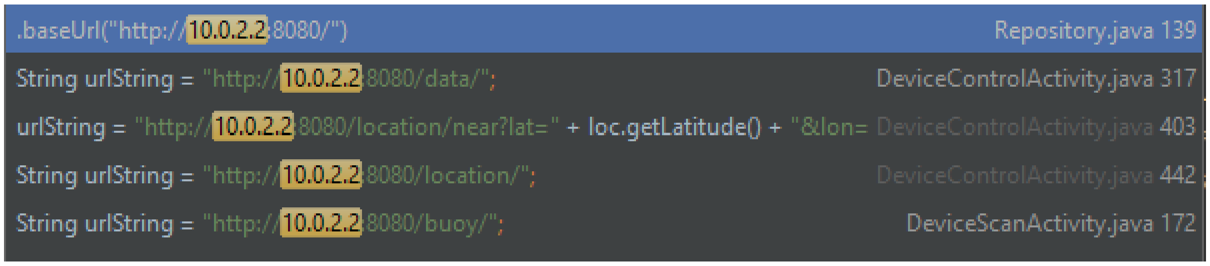
This is another configuration issue. The ESP32 has an internal clock that runs at a specific frequency, which we call the XTAL frequency. Our board’s frequency is 26 MHz, but it may be set to 40 by default. Run the command “**idf.py menuconfig**” and navigate to Component config -> ESP32-specific -> Main XTAL frequency, and ensure the value is set to 26 MHz.

5 App Integration

5.1 Android App Operation

Logging In:

To log into the Android Application, users can use either an Android emulator or a physical Android Device. Depending on which method is being used, the url in the source code needs to be modified.



```
.baseUrl("http://10.0.2.2:8080/")                                              Repository.java 139
String urlString = "http://10.0.2.2:8080/data/";                               DeviceControlActivity.java 317
urlString = "http://10.0.2.2:8080/location/near?lat=" + loc.getLatitude() + "&lon= DeviceControlActivity.java 403
String urlString = "http://10.0.2.2:8080/location/";                           DeviceControlActivity.java 442
String urlString = "http://10.0.2.2:8080/buoy/";                               DeviceScanActivity.java 172
```

10.0.2.2 is what should be used when connecting locally through an emulator. Otherwise, the address of the system hosting the docker environment should be used. If this isn’t set properly, requests to the database won’t work and the user will be unable to log in. **Upon release this address should be the address of the system on the Cyber Range.**

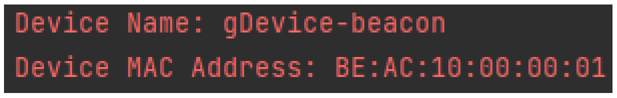
Once the address is set properly, users can log in by either creating a new account or using the Erik/John/Kris accounts if the database has been preseeded. **Note: As of writing this, there is a known issue where the Erik/John/Kris accounts require inputting the raw MD5 hash. This issue isn’t present with registered accounts, implying our MD5 hashing isn’t set up properly. This needs to be investigated and fixed.**

Buoy Selection:

Upon logging in, a list of bluetooth devices will be visible. What’s listed here also depends on the device being used. If in an emulator, two instances of “gDevice-beacon” will be displayed. These are virtual buoys that contain data for testing purposes. If using a physical device, your buoys name will instead be displayed. The default name is “SSDS Test”.

Registering Buoys:

From buoy selection, neither emulator nor physical devices will be able to connect to a buoy until the buoy is registered. In order to register a buoy, click the hamburger menu at the top right and open “Register Device”. At this point you’re prompted to enter a name and a MAC address. The name can be whatever description you want, however the MAC address needs to match the address associated with that buoy.This address can be found by selecting your buoy on the buoy selection page, causing it to be printed to the debug console. This is what that may look like:

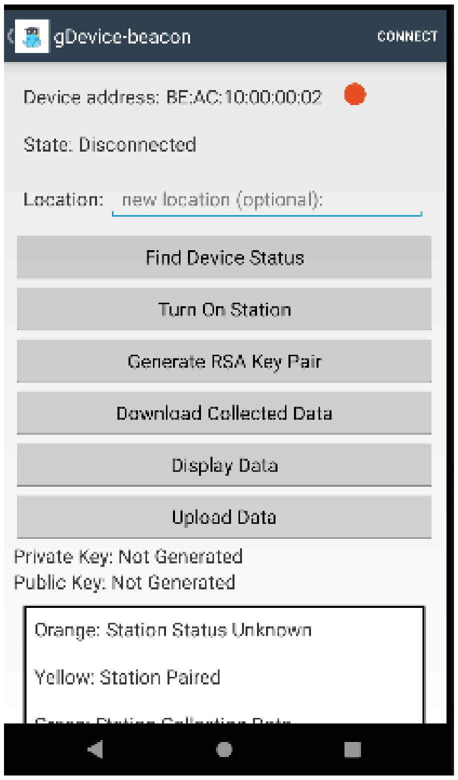


```
Device Name: gDevice-beacon
Device MAC Address: BE:AC:10:00:00:01
```

Upon release, this address should instead be included with the users’ buoys. After registering the buoy, it should now be possible to connect to it.

Connecting With the Buoy:

Once a buoy is registered, it can now be clicked on and opened. This is what the next screen looks like:



Device address: BE:AC:10:00:00:02

State: Disconnected

Location: new location (optional):

Private Key: Not Generated
Public Key: Not Generated

Orange: Station Status Unknown

Yellow: Station Paired

In order to collect the data for upload, turn on the station, generate an RSA key pair, then click download. At this point, all the data contained on the buoy should now also be on the mobile device. This data can now be uploaded by clicking the upload data button. Note: if data has not been previously sent from the phone's current GPS location before, a location label needs to be put in the location text entry box. After successfully uploading, the data will now be visible in the web-ui.

5.2 Web App Operation

To log in, users can either register a new account or use the Erik/John/Kris accounts if the database has been preseeded. From there the Data and Buoy tabs become available. The buoy tab serves t1o display the name of each buoy and the group it belongs to. This page may need to be updated or removed in the future as it's pretty barebones. Future groups should find what works and what doesn't, then assess whether or not it's worth keeping around. The display tab is where most of the web-ui's functionality lies. Here, users can view all collected data in both a raw table format as well as with the graphic control system. Something to improve upon in this page would be to add more controls for users to filter the data with.