AIR POLLUTION MAPPING BANDS

By

Chirag Nanda

Vatsin Shah

Vedant Agrawal

Final Report for ECE 445, Senior Design, Spring 2022

TA: Amr O. Ghoname

May 2022

Project No. 33

Abstract

This report details the complete design process for engineering the "Air Pollution Mapping Band." The "Air Pollution Mapping Band" is a portable device accompanied by an android application. The band is designed to measure and send Carbon Dioxide, Carbon Monoxide, and Propane concentrations to the application. The application then plots these concentrations on a map to give a streetwise depiction of air pollution. Additionally, the application periodically sends these concentration and location data points to a webserver to maintain a standard map across users. While collecting data, the application can also warn the user should the pollutant concentration exceed safe thresholds.

Contents

1. Introduction 1
1.1 Problem Overview 1
1.2 Solution Overview 1
1.3 Visual Aid 2
1.4 High-Level Requirements 2
1.5 Subsystem Overview 3
2. Design 4
2.1 Power Subsystem 4
2.1.1 Battery and Switch 4
2.1.2 LDO Regulators 4
2.1.3 Final Circuit Schematic 5
2.2 Indicator Subsystem 5
2.2.1 Power LED 5
2.2.2 Connection LED 5
2.2.3 Final Circuit Schematic 5
2.3 Sensor Subsystem 6
2.3.1 Sensor array 6
2.3.2 Microcontroller and Bluetooth 6
2.3.4 Final Circuit Schematic 7
2.4 PCB Layout 7
2.5 App Subsystem 8
2.5.1 Bluetooth on Android 8
2.5.2 AWS Server 9
2.5.3 Google Maps 10
3. Design Verification 11
3.1 Power Subsystem 11
3.1.1 Voltage Regulation 11
3.1.2 Battery Life 11
3.1.3 Switch Check 12
3.2 Indicator Subsystem 12
3.2.1 Power check 12
3.2.2 Indicator check 12

3.3 Sensor Subsystem 12
3.3.1 Analog sensor data retrieval 12
3.3.2 Periodic Bluetooth data transfer 13
3.3.3 Threshold tests 13
3.4 Software Subsystem 14
3.4.1 Testing AWS Server Pipeline 14
3.4.2 Standard map maintenance 15
4. Costs and Schedule 16
4.1 Parts 16
4.2 Labor 16
4.3 Total Cost 17
4.3 Schedule 17
5. Conclusion 18
5.1 Accomplishments 18
5.2 Uncertainties 18
5.3 Ethical considerations 18
5.4 Future work 19
References 20
Appendix A Requirement and Verification Table 22
Appendix B Sensor plots 26
Appendix C Sensor Tolerance Analysis 27
Appendix D Initial PCB Layout 30

1. Introduction

1.1 Problem Overview

As air pollution has increased globally, the need for pollution tracking has grown in tandem. Today, most cities take readings using satellites and sensors scattered around the city to collect an aggregate reading of city-wide air quality [1]. While this may give a reasonable estimate of the air pollution over a city-wide area, the air quality of individual localities and streets may differ vastly.

Air pollution can change dramatically over a day. A variety of factors, including traffic, population density, the operation of office buildings, and factories, can influence the air quality. A more dynamic calculation of air quality can help people decide which routes to take and which places to avoid. Some cities like Barcelona and Chicago have tried implementing IOT-based air pollution trackers embedded into city-wide infrastructure to aid this effort. Google has even attempted to fit street view cars with sensors to track pollution levels [2]. Nonetheless, these devices are extremely expensive. For instance, the sensor nodes used in Chicago cost around five thousand dollars per node [3]. Additionally, the sensors are often spread far apart, preventing accurate locality-centric/streetwise data collection of pollution.

1.2 Solution Overview

Our solution to this problem was to create a cheap wearable band and an accompanying Android app that would continuously monitor the air quality around the user. The broader idea is to have thousands of users wear this band to help contribute to a city-wide map that everyone can access. Nevertheless, within the time constraints of the course, we planned to create a proof of concept of the band and a simple application that gave alerts to the user about their general vicinity. The app would keep a personal record of air pollutant levels of the places they visited on a map.

We aimed to measure carbon dioxide (CO_2) and carbon monoxide (CO) in parts per million (ppm). Additionally, since this band would be portable, it had the potential to be used as a warning device in indoor spaces. Hence, we also wished to sense dangerous flammable gasses like propane. The band could then help find poorly ventilated areas and warn users of potential gas leaks in warehouses and storage rooms. For our project, we decided to build one band. However, we planned to have multiple profiles on our app and a centralized server to test how numerous users could update the same map.

1.3 Visual Aid

Figure 1 highlights how our completed project looks, including the physical device and the android app:

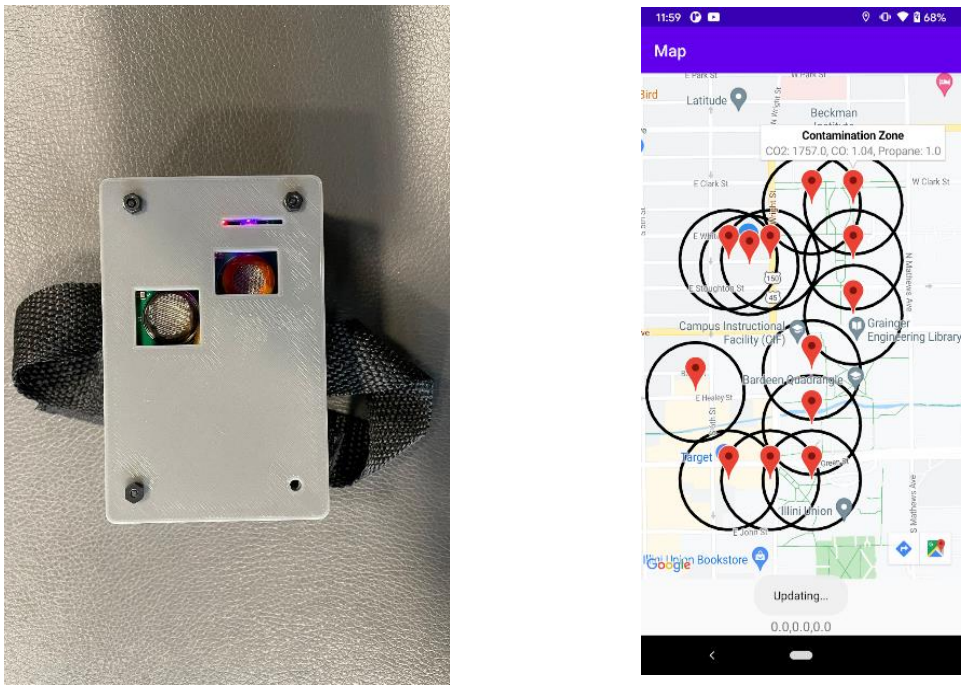

Figure 1: Final design of the project

1.4 High-Level Requirements

While designing our project, we adhered to the following requirements:

1. The system should warn the user if propane is detected (since it is flammable). It should detect carbon monoxide concentrations up to 200 ppm and warn the user if concentrations exceed 100 ppm. It should detect carbon dioxide concentrations up to 15000 ppm and warn the user if concentrations exceed 10000 ppm. We have picked these values based on USDA-determined values of dangerous exposure [4] [5].

2. The app will need to be able to take pollution data from the band and update the map with the detected concentrations at a period of 5 minutes since pollutant levels do not change rapidly.

3. Our band needs to be wearable and must have around 1-3 hours of battery life to be able to track pollution data when a person makes their commute

1.5 Subsystem Overview

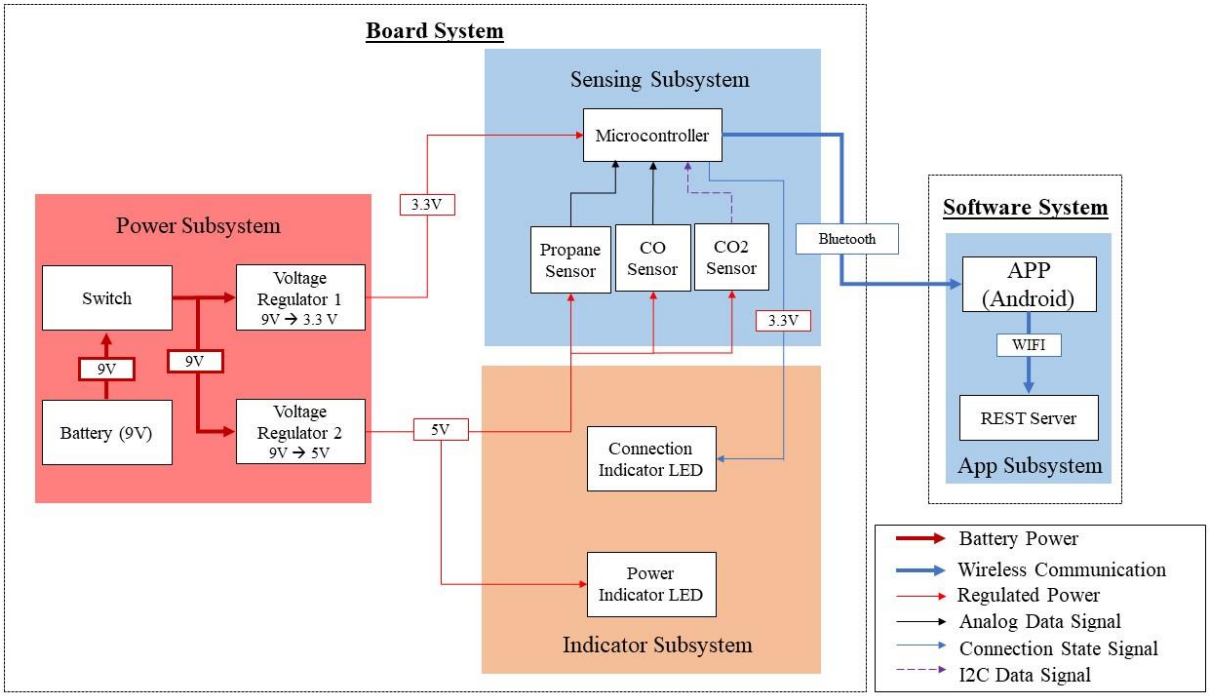

Figure 2: Block diagram of the project

Figure 2 shows the high-level block diagram for our project. Our design can be broken down into two broad categories: the board system and the software system. The board system includes the power, indicator, and sensing subsystem. The power subsystem is responsible for converting the 9V supply into 3.3V and 5V to be used by our microcontroller and sensor array, respectively. The indicator subsystem relays if the band is powered on and whether or not a device is connected to it. The sensing subsystem is responsible for collecting and transmitting air pollutant data to the connected device.

The board system met our first high-level requirement by being responsible for powering our sensors and microcontroller to monitor and send pollutant data to the app. The software system met our second high-level requirement and consisted of the app subsystem. The app subsystem's main goal was to visualize the pollutant data on a map and maintain a centralized map across different user profiles using a REST server. To fulfill our third high-level requirement, we kept the design minimal and ensured that our final printed circuit board (PCB) was as small as possible so that the band was compact and wearable.

2. Design

2.1 Power Subsystem

The Power subsystem consists of our battery, two Low-DropOut (LDO) regulators, and a switch to cut out the power to all other subsystems when the band is not in use.

2.1.1 Battery and Switch

The battery we originally planned to use was one made of three 3V coin cells. As our system needed a 5V rail and a 3.3V rail, we thought that a fully charged battery of 9V would not discharge below our minimum voltage requirements for our LDO regulators. But after we put our system together, we found that the current required by the system was between 200-300mA, which was not possible to supply using 3V coin cells. Thus, we instead shifted to using a Duracell 9V (MN1604) [6] battery capable of providing the required current for more than an hour, which was one of our high-level requirements. As we wanted to create a wearable device, we also wanted to provide an easy way to shut the device off when it was not in use. So, we added a switch to completely disconnect the battery from the rest of the system to prevent wastage of energy.

2.1.2 LDO Regulators

The LDO regulators we used for this project were the LD29150DT50R & LT1129IST-3.3#TRPBF. Our sensors require 5V, while our microcontroller requires 3.3V. Therefore, we needed to regulate the input voltage of 9V to two different levels. We decided to use these LDO regulators instead of buck converters for a couple of reasons. The first one was that we had an issue ordering buck converters that we could solder reliably even with a reflow oven. The available footprints were too small and required too much testing to ensure no false connections were made. Also, in case of a bad connection, there was no way to correct such a mistake due to their size. The second reason was pertaining to our high-level requirement for the size of the device. With the passive components required around the original buck converter, it would not have been possible to package everything within the dimensions we specified initially. With LDOs, we managed to reach the requirements easily.

2.1.3 Final Circuit Schematic

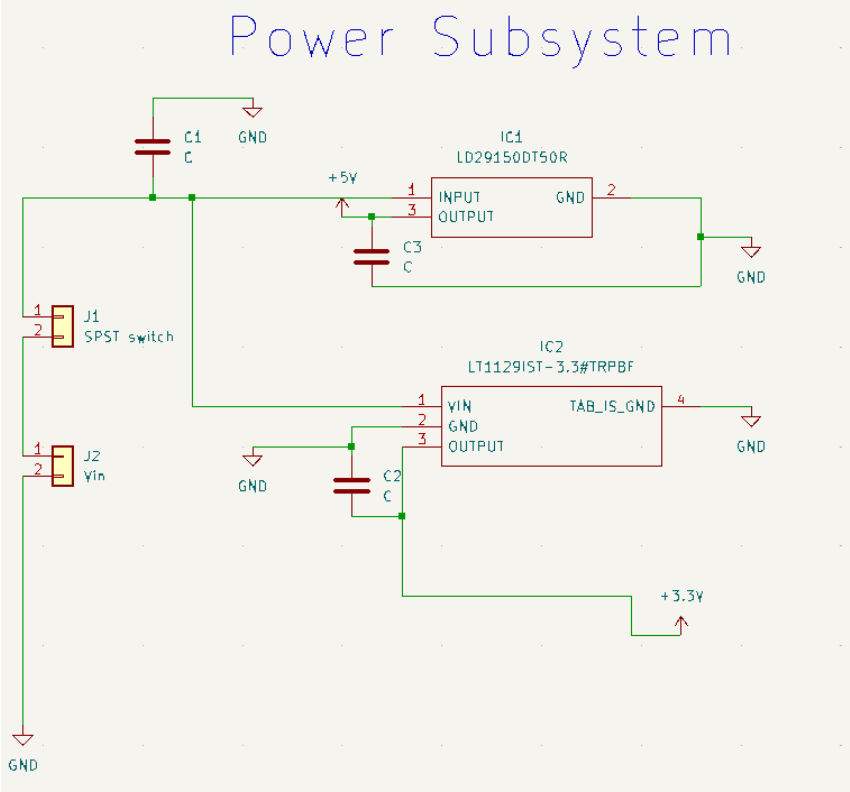

Figure 3: Circuit Schematic of Power Subsystem

2.2 Indicator Subsystem

We needed an easy-to-understand interface to indicate the state of the system. This helped with conveying to the user whether the device was powered on or not, if the device booted properly, and if a user with the app was connected to the device. The subsystem we came up with to help with this consists of a RED LED, that we call a Power LED and a RGB LED, that we call a Connection LED.

2.2.1 Power LED

This LED indicates that power is being supplied to the voltage regulators i.e., it indicates that the device is switched on. This also helps us see when the voltage from the battery drops low enough that the 5V regulator cannot supply 5V anymore, as the LED is directly connected to the output of the 5V regulator.

2.2.2 Connection LED

The color of this LED is controlled by the microcontroller. When the ESP32 boots up, it turns the Connection LED purple until a phone connects to the device. Once that happens, this LED is turned blue, thus indicating the connection. In case of a problem with boot, such as a failure to initialize Bluetooth or damage to the sensors, this light would not turn on at all.

2.2.3 Final Circuit Schematic

NOTE: RED_RGB, CON, and RDY are pins on the microcontroller that control the Connection LED. No such pins exist for the Power LED as it is not controlled by the microcontroller, and directly gets power from the 5V regulator.

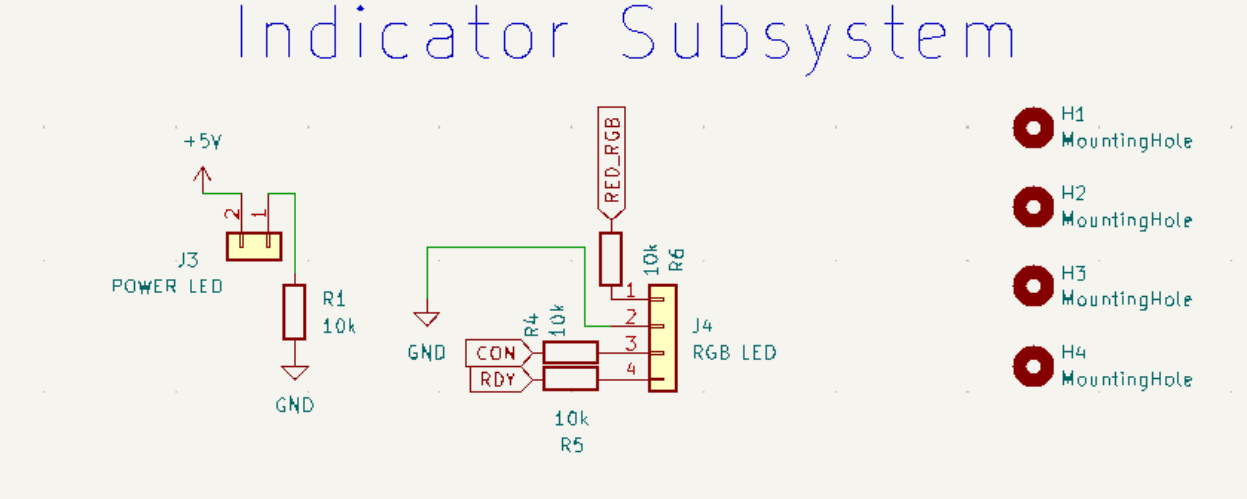

Figure 4: Circuit Schematic of Indicator Subsystem

2.3 Sensor Subsystem

The job of this subsystem is to measure any of the listed harmful gasses in the surroundings and periodically send this data to the app running on the user's phone. It consists of the MQ-2 sensor to detect flammable gasses like propane, an MQ-9B sensor to measure the concentration of Carbon Monoxide, an SGP-30 sensor which measures the concentration of Carbon Dioxide, and the ESP32 microcontroller that collects the data and sends it over Bluetooth to the app. The microcontroller also controls the Connection LED based on the current Bluetooth connectivity status.

2.3.1 Sensor array

This sensor array contains an MQ-2, MQ-9B, and an SGP30 sensor. MQ-2 and MQ-9B are variable resistance sensors. These have a heating coil and a sensing element that changes resistance based on the concentration of certain gasses around the sensor. The job of the heating coil is to bring the sensors to their respective operating temperatures, which are higher than room temperature. The potential drop across the sensing element changes as its resistance changes. Detecting this change in potential drop allows us to figure out the gas concentration. As we needed to detect this potential drop, we used analog pins on the ESP32, which connect to the internal 12-bit Analog to Digital Converter (ADC) pins. To use the SGP30, we connected it to the microcontroller's SDA and SCK pins and used the wire library on Arduino to communicate over the I2C bus.

2.3.2 Microcontroller and Bluetooth

The microcontroller we chose for our project was the ESP32. We selected the ESP32 because it had Bluetooth connectivity and an in-built timer. To save power and extend battery life, we transferred the collected sensor data to the application every two minutes. We used the ESP32's inbuilt timer to keep track of this period. We programmed the timer to create an interrupt signal every two minutes. Nevertheless, we programmed the ESP32 to break this periodic data transfer if any detected concentrations exceed safe thresholds. As soon as a high concentration value is detected, the ESP32 asynchronously sends the data to warn the user.

2.3.4 Final Circuit Schematic

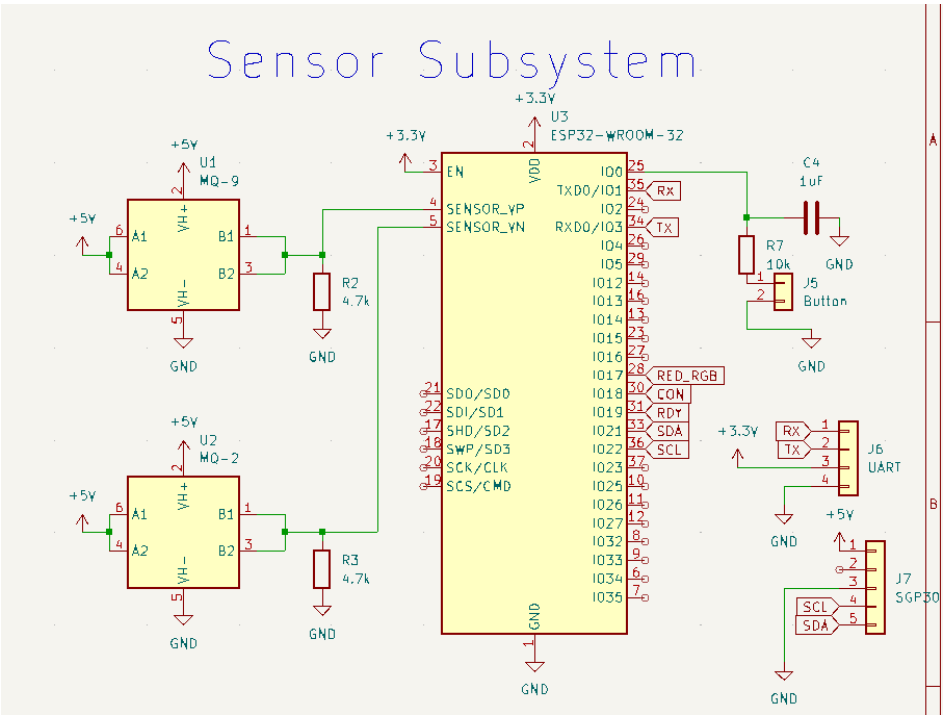

Figure 5: Circuit Schematic of Sensor Subsystem

2.4 PCB Layout

NOTE: Zones are turned off to show the routing of wires and arrangement of components better.

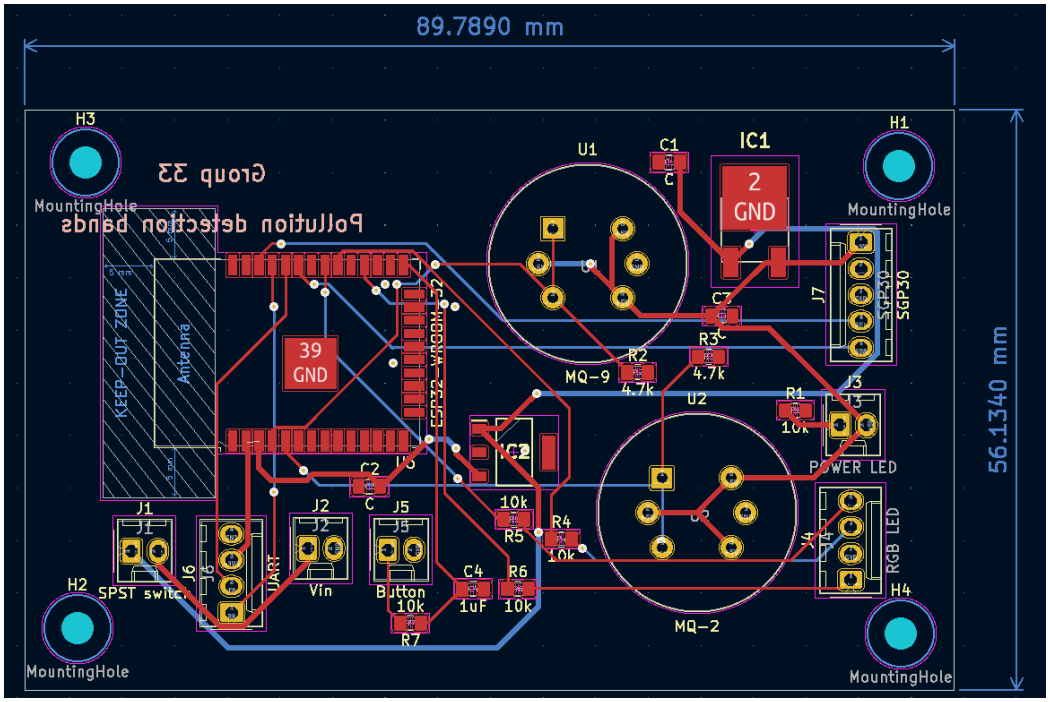

Figure 6: Final PCB layout

2.5 App Subsystem

To utilize the air contamination data obtained from the sensor subsystem, we designed an Android application that connects to the Bluetooth signal of ESP32. Using the application, we can visually show the gas values to the users in a human readable format. The application is also responsible for communicating with the server in the form of GET and POST requests. Finally, the application provides a way for the users to visually see the contamination data collected from the server on a Google Maps overlay to see the contamination data in areas around them and use the map to avoid highly polluted zones. The figure below shows all the wireless communication handled by the app subsystem

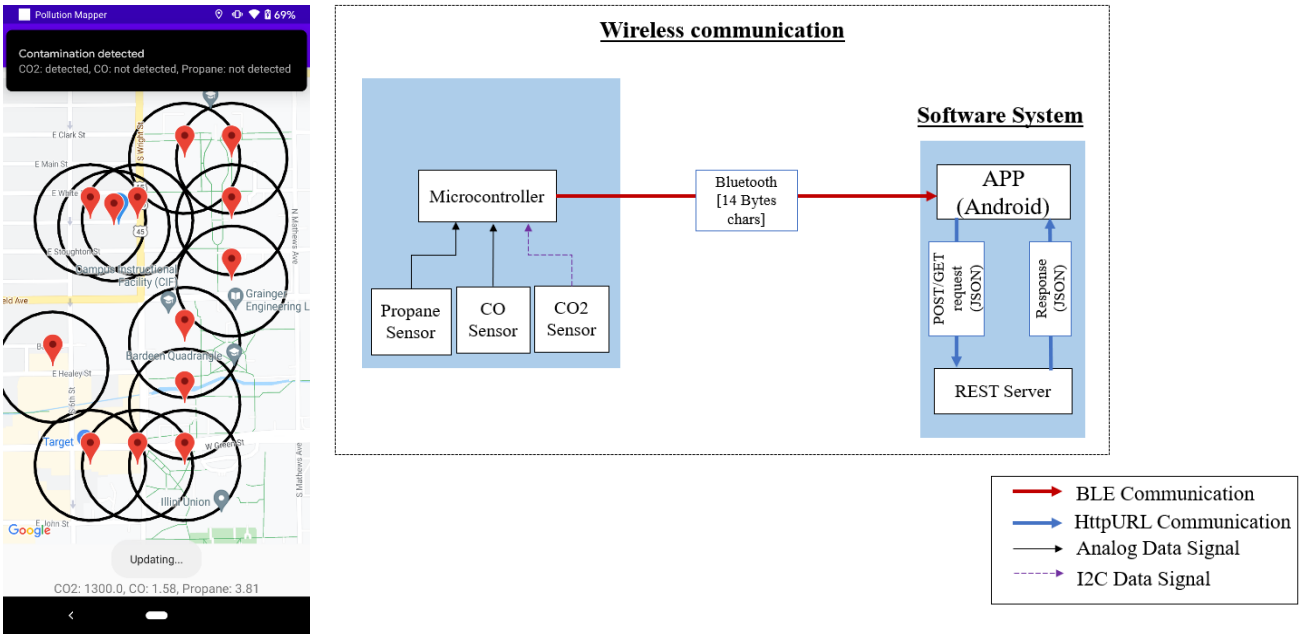

Figure 7: Block diagram for wireless communication (right) and finished application (left)

2.5.1 Bluetooth on Android

For handling the Bluetooth connection with the ESP32 microcontroller, we used the native Bluetooth library provided by Android. We created a Bluetooth client that runs on a separate thread to maintain a continuous Bluetooth connection without affecting any other functionality that the main thread handles. This involved establishing a Bluetooth socket with the Bluetooth server broadcasted by ESP32 using the function createRfcommSocketToServiceRecord(UUID). Then we could connect to this socket and establish input and output streams. We have designed the code in such a way that every time the app connects to a Bluetooth socket, it sends an output message of "connected" to the socket. This way, we could program the microcontroller to detect this message and change the color of the LED in the indicator subsystem. Then the code establishes an active input stream that collects the broadcasted packets. The Bluetooth packets are sent in the form of a string with the schema shown in figure 8, with the values being separated by commas. Each of the three gas concentration values takes up 4 characters and two commas to separate them, resulting in the total payload size being 14 bytes. We parse the string and get the values converted into Double type on the application side. We then immediately check for any value exceeding our set threshold. If any of them do, we alert the user using a notification banner and also send it to the central server. Regardless, the app updates the server every five minutes.

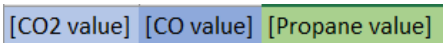

Figure 8: raw Bluetooth packet format

2.5.2 AWS Server

For creating the server, we decided to use AWS as it offers an intuitive interface to make endpoints that can invoke user-defined functions. This proves to be a helpful tool for creating highly available servers that can scale according to the number of incoming requests. We used AWS API Gateway to make POST and GET endpoints that invoke specific Lambda functions, which are in turn connected to a DynamoDB database also hosted on AWS to store and return the data. DynamoDB is a NoSQL database where we defined the primary key to be the GPS coordinates (Latitude, Longitude). For each of these keys, we store the Carbon-dioxide, Carbon Monoxide, and Propane gas concentration values for those particular coordinates.

For the POST request, we made a simple API endpoint with pollution-data which invokes a Lambda function called "AddData." This Lambda function is connected to a DynamoDB table called "pollution-data". The lambda function just inserts the event that is passed on to it in the body of the POST request. While sending the user location to the server, we decided to round down the latitude and longitude values to the third decimal place to group the GPS coordinates in a 100-meter radius together. An example of a successful POST request is shown in the figure below.

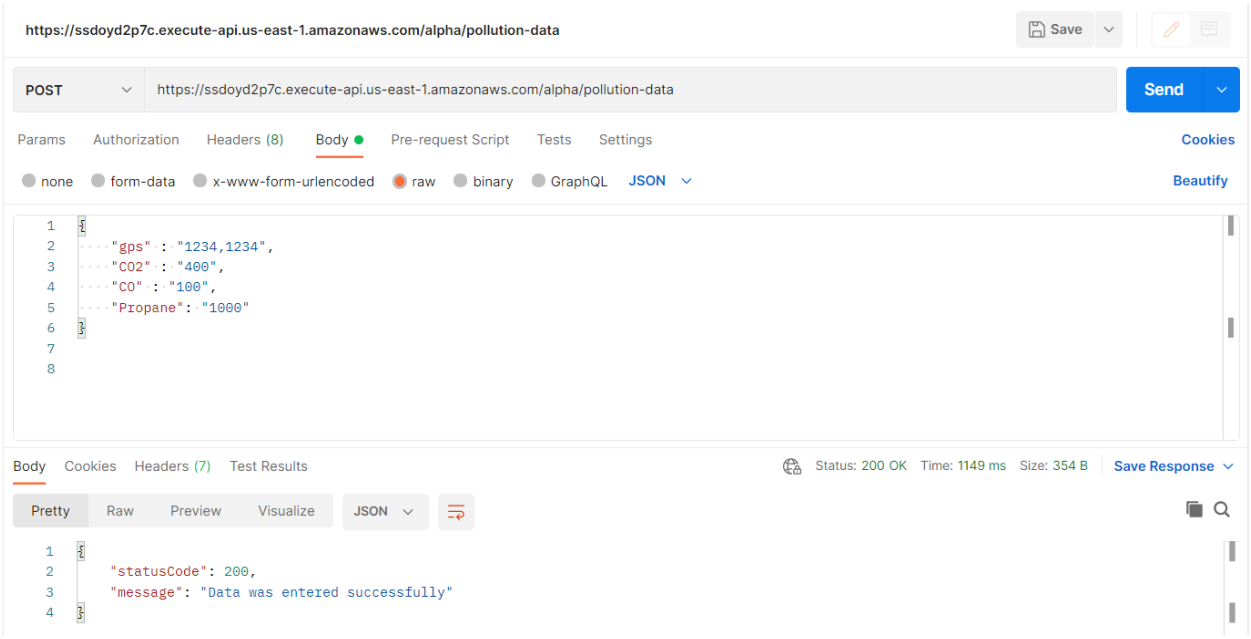

Figure 9: Example of a POST request body and response from the AWS server

For the GET request, we use the same API endpoint with /pollution-data. But this time, the GET request invokes a separate Lambda function called "ReturnData" that scans the DynamoDB table and returns all the items retrieved from the table as the body of the response object. Since the schema is uniform, we can loop through each item of the array and plot the GPS coordinates of the contamination zones on Google Maps along with the gas concentration values for each zone.

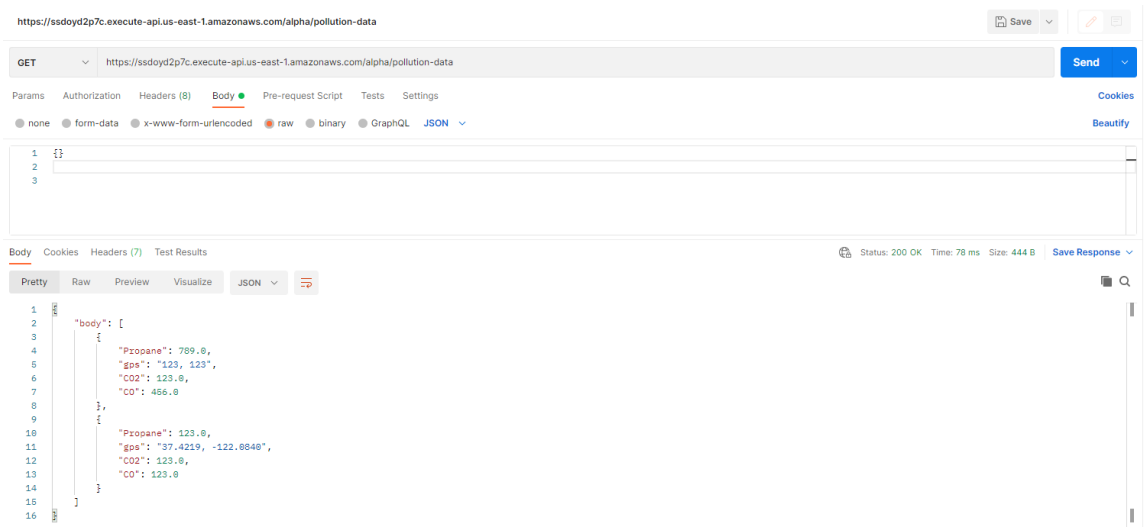

Figure 10: Example of a GET request response from the AWS server

2.5.3 Google Maps

For displaying the Google Maps overlay on our application and to plot all the contamination zones, we used the built-in Maps library provided by the Google Mobile Services (GMS) native to Android. To plot the contamination zones, we loop through all the entries returned by the GET request to the server and plot markers on the map using the GPS coordinates with the title "Contamination Zone". We could also add a circle of 100-meter radius on the map for each contamination zone using the built-in function "addCircle". Moreover, we decided to show the gas concentration values for each GPS coordinate returned in the info window of each marker. This way, the users can tap on the marker of the contamination zones to reveal the gas concentration values recorded for that zone.

We also decided to plot and update a blue colored marker to indicate the current location of the user. This is important as users can visually see if they are located in any contamination zone. We also use this to notify the user if they enter a contamination zone with higher than threshold concentration values.

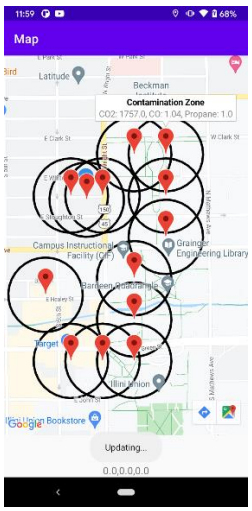

Figure 11: View of the Google Maps overlay with contamination zone marker and user location marker

3. Design Verification

This section details how we have met and verified each of our requirements. For a compiled list of subsystem-specific requirements, please refer to Appendix A.

3.1 Power Subsystem

3.1.1 Voltage Regulation

We verified the regulated voltage levels by running the board using constant 9V from a bench power supply. We wanted to confirm that our regulators of choice could provide stable voltages within acceptable ranges. We measured the voltage continuously for almost an hour, as shown in figure 12. Overall, we did not see any drops in voltage within the time span.

Figure 12: Voltage measurement across LDO

3.1.2 Battery Life

Battery life was verified by simply running the system until the battery ran out. We found that the battery lasted around 1 hour and 20 minutes, which aligns with the graph given in the battery's datasheet for our 200 - 300mA current draw (figure 13). The voltage across the battery was observed in 10-minute intervals, as shown in table 1. This battery life is within the range specified in our high-level requirements.

Time (minutes)	Voltage
10	8.9
20	8.6
30	7.8
40	6.7
50	6.1
60	5.8
70	5.2
80	4.7
90	4.3

Table 1: Time vs Voltage across the battery

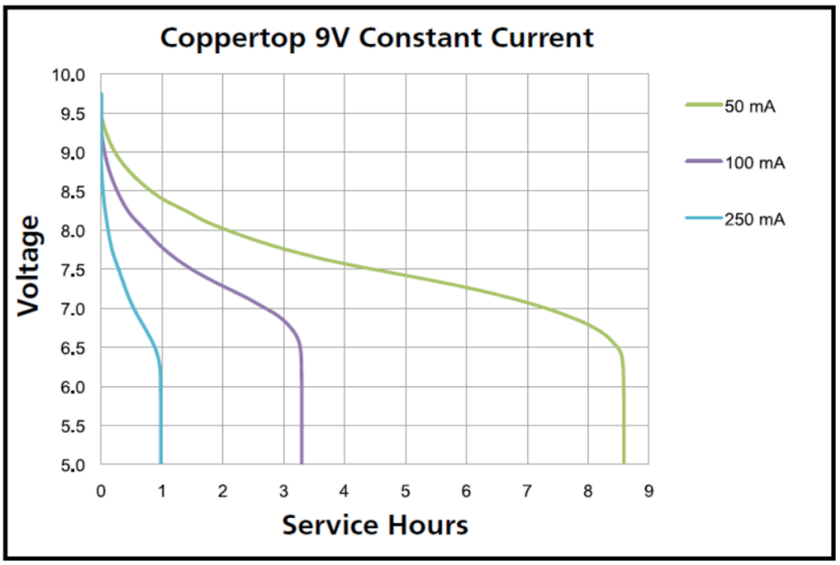

Figure 13: 9V Duracell battery life for different current draws [6]

3.1.3 Switch Check

Continuity across the switch was verified using a multimeter and the power LED also helps indicate the working of the switch as this LED turns on as soon as the 5V rail gets power.

3.2 Indicator Subsystem

3.2.1 Power check

Power LED was checked by verifying the voltage across the regulator using a multimeter and seeing the LED light up at the same time.

3.2.2 Indicator check

Indicator LED was checked by printing the connection state of Bluetooth over the serial terminal and by visually checking the color of the LED at the same time. We stress tested the status by repeatedly disconnecting and reconnecting our phone to the board.

3.3 Sensor Subsystem

3.3.1 Analog sensor data retrieval

Both the MQ-2 and MQ-9B sensors are analog sensors that measure concentrations of gases based on changing resistance. Upon being exposed to a gas, the resistance of their internal sensing materials change based on a gradient as shown in the graphs in Appendix B. Therefore, to verify the functionality of these sensors, we essentially had to measure the potential drop and resistance across them.

Before using the sensors, we needed to calibrate them by running 5V across them for around 24 hours. This calibration process is meant to burn off any impurities stuck to the heating coil. After calibrating the sensors, we used a breadboard to connect the sensors to our ESP 32 development board and measured the resistance and potential difference across them using a multimeter. We noticed that the measurements were very noisy. Nevertheless, both potential difference and resistance would drastically change if we blew on them or took them outside. This meant that the sensors were at least responsive to changes in

their environment. However, it was impossible to use these measurements to calculate actual values due to the erratic nature of the data collected.

We then converted the analog sensor data into ppm using an Arduino library called MQUnifiedSensor [7]. The library periodically compares the change in resistance values to the corresponding gradients of the resistance graphs (shown in Appendix B). Further explanation of the error tolerance of these measurements is provided in Appendix C.

3.3.2 Periodic Bluetooth data transfer

To verify the periodic data transfer, we used a Bluetooth debugger app on our phone. Once the app is connected to our board via Bluetooth, the two-minute periodic data transfer begins as shown in figure X. The transfer period is correct to the nearest minute, as shown in the timestamps. To simulate values exceeding our threshold, we used Arduino's Serial Input to enter a large measurement. As soon as these values were read by the ESP32, an asynchronous message was sent to the app.

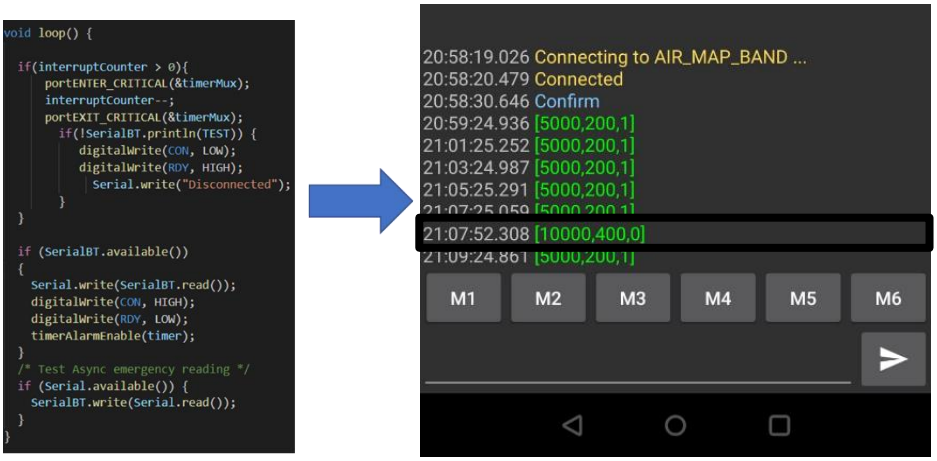

Figure 14: Bluetooth Connectivity Test

3.3.3 Threshold tests

To further test the sensor thresholds, we field tested the device around campus. We lowered the Carbon Dioxide and Carbon Monoxide thresholds for the sake of these tests. We observed that when vehicles passed, the concentration of Carbon Dioxide would spike and immediately a notification would appear on the phone. The route we took around campus, as well as the notification is shown in figure 4. Realistic tests of the threshold limits detailed in our high-level requirements were not possible as they would be too dangerous to perform and would require specialized lab equipment.

Lastly, we used a lighter to verify the Propane sensor. We performed these tests outside to ensure good ventilation. Upon switching on the lighter, we would immediately see spikes in propane concentrations. Since the measurements of the MQ-2 sensor were very noisy and would often spike randomly, through trial and error we figured out a threshold that would only be crossed on the detection of propane. This threshold was 500ppm.

3.4 Software Subsystem

3.4.1 Testing AWS Server Pipeline

The architecture of our server hosted on AWS consists of an API Gateway, Lambda functions and finally a DynamoDB table. The API Gateway exposes API endpoints for the GET and POST requests which when called on launch the specific Lambda functions. These Lambda functions are responsible for either entering the data or retrieving the data from the DynamoDB table. We tested this pipeline from end to end by making a test POST request to the API endpoint using Postman as shown on the left of the figure below. We verified that we get a status of 200 in the response of the POST request and that the entry was successfully made in the DynamoDB table as shown on the right in the figure below.

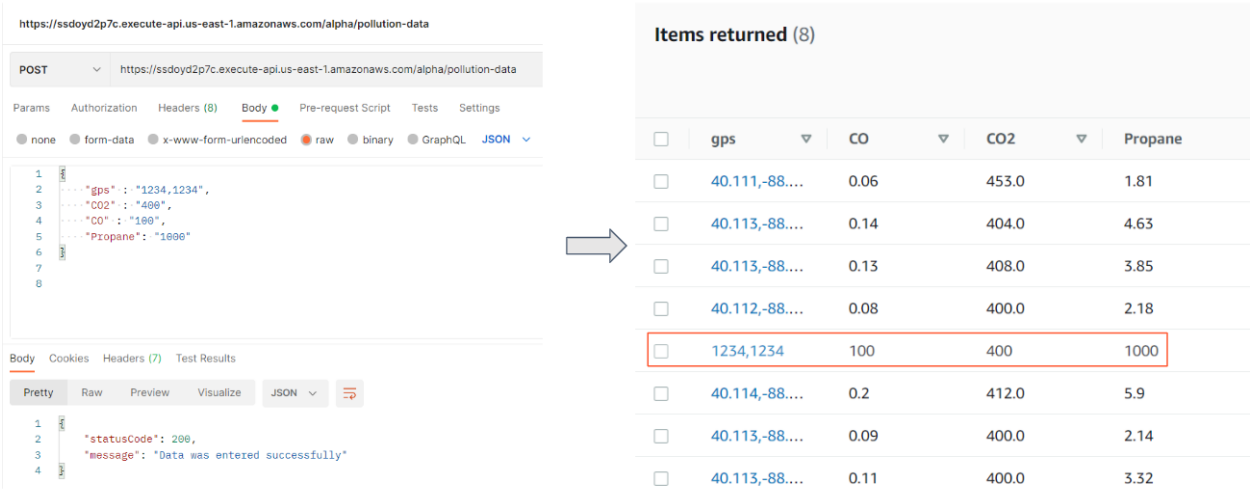

Figure 15: Testing the POST request pipeline (Postman on the left and DynamoDB table on the right)

Similarly, to test the GET request pipeline, we made a test GET request to the API endpoint using Postman and verified that the entire DynamoDB table was returned in the response object of the request.

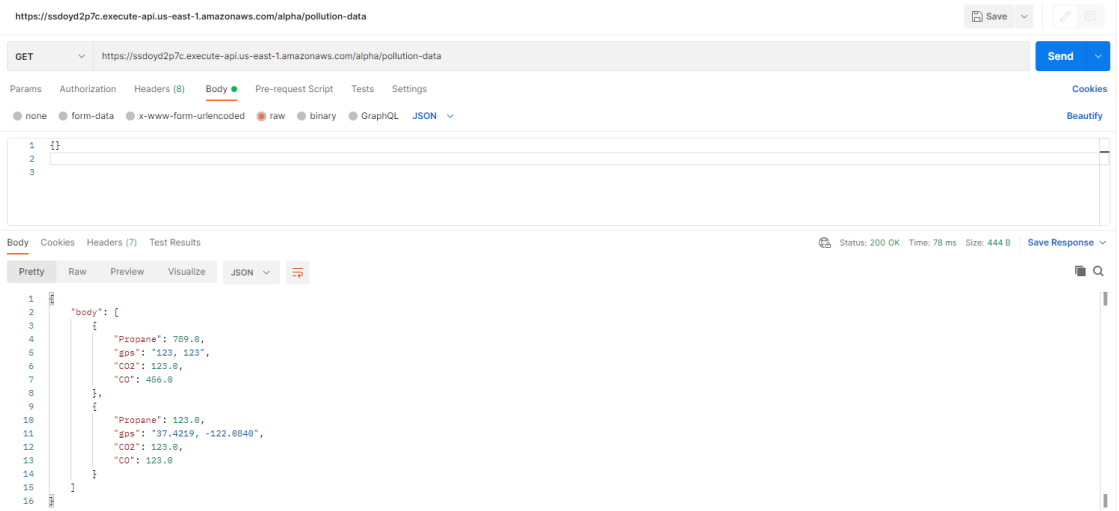

Figure 16: Response of a GET request made using Postman returning all the entries of the table

3.4.2 Standard map maintenance

To fully test our application all the way from testing the Bluetooth connection and server communication to plotting out the contamination zones on the Google Maps overlay, we cleared the DynamoDB table and conducted a walk around the campus while wearing the band on ourselves. Clearing the DynamoDB table was important to test if we were able to send and retrieve the data to the server in real time without the existing values interfering the test results.

We turned on the band and launched the application. The application could connect to the band right away as indicated by the indicator LEDs on the band and the gas values showing up on the application. This way we could test that the Bluetooth packets were being received and unpacked without any data loss. We verified that the application was making periodic POST requests to the server sending it the data that we were collecting by inspecting the DynamoDB table as we conducted our walk. We could also verify that the application was able to retrieve all the data from server as the map on our application would periodically be updated with the data we were collecting and sending to the server. We also verified that our application was able to alert the users if any gas concentration exceeded the threshold by seeing spikes in carbon-dioxide data when cars passed us on the walk and receiving a notification from the application accurately depicting the detection of carbon-dioxide gas.

We conducted this walk from the ECEB to Green Street through the Bardeen Quad. The figure below shows all the contamination zones we plotted on this walk along with a notification banner indicating that carbon-dioxide is detected since it exceeded our threshold.

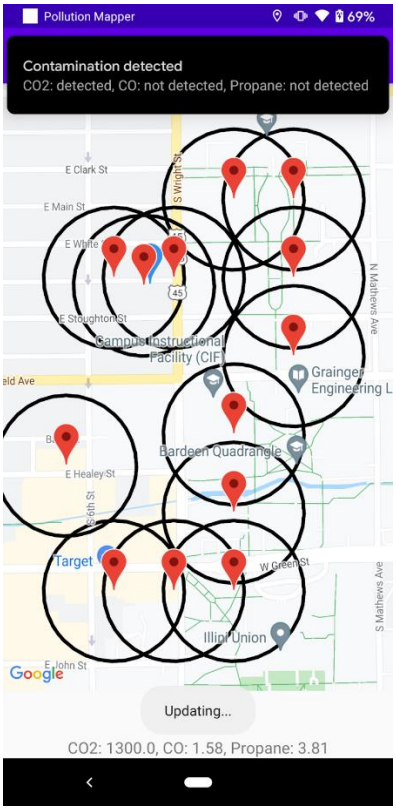

Figure 17: Test result of our walk around the campus

4. Costs and Schedule

The cost of the project can be divided into labor cost and parts cost.

4.1 Parts

Table 2 details the cost of all the parts we used to build our project.

Part	Manufacturer	Quantity	Retail Cost ($)	Bulk Purchase Cost ($)	Actual Cost ($)
Flammable gas sensor (MQ-2)	SparkFun	1	4.95	4.95	4.95
Carbon Monoxide sensor (MQ-9B)	SparkFun	1	5.95	5.95	5.95
Carbon Dioxide sensor (SGP30)	Adafruit	1	17.50	17.50	17.50
UART Programmer	SparkFun	1	14.95	14.95	14.95
ESP32 Development Board	Expressif	1	10.20	10.20	10.20
ESP32	Expressif	1	3.01	3.01	3.01
5V LDO (LD29150DT50R)	STMicroelectronics	1	1.44	1.12	1.44
3.3V LDO (LT1129IST-3.3#TRPBF)	Analog Devices	1	7.34	4.37	7.34
9V Battery	Duracell	1	3.25	1.95	3.25
10k Ohm Resistor	ROHM Semiconductor	4	0.16	0.02	0.64
4.7k Ohm Resistor	ROHM Semiconductor	2	0.16	0.02	0.32
Push Button	Gravitech	1	1.74	1.12	1.74
SPST Switch	Nidec Copal	1	0.75	0.44	0.75
RGB LED	SparkFun	2	0.10	0.04	0.20
1uF Capacitor	KEMET	1	1.13	0.22	1.13
0.33uF Capacitor	KEMET	1	0.34	0.05	0.34
10uF Capacitor	Kyocera AVX	1	1.04	0.35	1.04
3.3uF Capacitor	KEMET	1	0.58	0.14	0.58
Total				**66.40**	**75.33**

Table 2: Parts Costs

4.2 Labor

For this calculation we are taking $32 as our per hour wage because this is the average salary of ECE interns in Illinois [8]. Additionally, we spent an average of 10 hours a week for a total of 10 weeks of this semester working on our senior project. Lastly, we are using a 2.5 multiplier to reflect any miscellaneous costs like lab clean up, maintenance, etc. Using these numbers, the total labor cost became:

$$Labor\ Cost = \frac{\$\#}{hour} * 2.5 * \# \frac{hours}{week} * \#\ weeks * \#\ people$$

$$Labor\ Cost = \frac{\$32}{hour} * 2.5 * 10 \frac{hours}{week} * 10\ weeks * 3\ people$$

$$Labor\ Cost = \$24000$$

4.3 Total Cost

$$Total\ Cost = Labor\ Cost + Part\ Cost$$
$$Total\ Cost = \$24000 + \$75.33$$
$$Total\ Cost = \$24075.33$$

4.3 Schedule

Table 3 breaks down our contributions over the ten weeks we worked on the project

Week	Chirag Nanda	Vatsin Shah	Vedant Agrawal	Milestones
2/21	→ Sensing subsystem circuit schematic → Finalize sensors	→ Power subsystem circuit schematic → Find the appropriate buck converters for regulating voltage	→ Indicator subsystem circuit schematic	1. Circuit schematic 2. Final draft of design document 3. Finalize and order parts
2/28	→ PCB design for sensor subsystem	→ PCB design for power subsystem	→ PCB design for indication subsystem	1. PCB design
3/7	→ Figure out the capabilities of the google maps API and test possible UI designs.	→ Work on the power subsystem	→ Start building the android app	1. Test parts 2. First version of app
3/21	→ Figure out faults in the first PCB design and finalize the design for the PCB order.			1. Second PCB design
3/28	→ Test the sensors	→ Learn and practice using reflow oven	→ Work on building the REST API and server.	1. Test sensors 2. Individual progress report
4/4	→ Work on the indication subsystem and ensure band can connect with a device over Bluetooth	→ Start soldering parts on PCB	→ Finalize app UI design and multi-user functionality	1. Finalize app design
4/11	→ Test data transfer and threshold concentrations	→ Verify connections on PCB	→ Test server and google map's API	1. Finalize systems
4/18	→ Test band and application at various locations in Urbana-Champaign			1. Test project
4/25	→ Prepare for demo → Finalize and verify all components			1. Demo
5/2	→ Prepare for final presentation → Work on the final report			1. Final Presentation 2. Final Report

Table 3: Individual contribution over 10 weeks

5. Conclusion

5.1 Accomplishments

We were successfully able to fulfill all three of our high-level requirements. We met our first high level requirement by securing three sensors (SGP30, MQ-2, and MQ-9) that could measure Carbon Dioxide, Carbon Monoxide, and Propane up to our desired threshold. We were able to do this while keeping our cost under the given budget. Secondly, we fulfilled our second goal by completing our software system using Android, an ESP32 microcontroller, an AWS webserver, and the google maps API. Lastly, we satisfied our third requirement by ensuring that our final design was compact and wearable. Our device was 9cm in length, 5.8cm in width, and 4cm in height. We were even able to ensure a battery life of at least an hour.

5.2 Uncertainties

Even though we were able to achieve all our goals, we faced various challenges and uncertainties with our hardware. Firstly, right after the first PCB order, we realized that the footprints we used for certain components on our board were incorrect. As a result, our first PCB could not be used to even test our initial idea. Additionally, with our initial circuit design, we were unable to fit all our components within our initial specifications. This was mainly because our original design of the power subsystem was taking too much space. We were only able to significantly reduce the size our board after pivoting to LDO regulators.

Lastly, we also faced a lot of issues while soldering and booting our ESP32 chip on the board. Due to its pin layout, soldering the chip by hand was not possible without making numerous mistakes. We instead resorted to learning and using the reflow oven. After getting the chip soldered, we still were not able to boot a program. After hours of going over the design and checking every connection, we realized that we had made two mistakes. Firstly, we had forgotten to connect the EN pin of the ESP32 to power. Secondly, we had reversed the read and write connections between the programmer circuit and the ESP32. We fixed the first mistake by bridging the connection of the EN pin with the ESP32's power PIN. Lastly, we rectified the second issue by connecting wires to the correct pins of the ESP32 and using an external UART programmer to boot the chip with our code.

5.3 Ethical considerations

Since we were tracking the location data of the user, it was a possible violation of IEEE [9] and ACM [10] privacy standards. Additionally, we found that by using Bluetooth in our Android application, we would need to have "ACCESS_COARSE_LOCATION" permission enabled by the user [11]. Since there was no workaround for this permission and we needed the users' location for creating a heat map, we made sure that the user identity was not linked to the location data recorded by our application and sent to the server. Firstly, we only started using the user's GPS location after appropriate in-app permissions were given. To ensure user privacy, we only logged the location data and kept the user anonymous on our map. Our app only associates pollutant data to the user's location and no trace of the user's identity is recorded. We did this by only sending the rounded down GPS coordinates of the user in case of contamination detection. This way, even if an IP address is associated with a POST call to the server, the exact location of any IP would be impossible to determine.

5.4 Future work

This project has a massive scope for future development. On the hardware side, improvements can be made on the design to make the band more compact. This can be done by reducing the space between components and using a professional soldering service. Additionally, the MQ-9B and MQ-2 sensors take a lot of space. One way to further reduce the size is to use better, more expensive sensor boards that have in built voltage regulators. As a result, our power subsystem circuit would also become smaller. These digital sensors are also often more accurate and do not require pre-heating. We suspect that the heating coils in the MQ-2 and MQ-9B sensors are a major source of power drain in our circuit. As a result, by replacing these sensors, we can easily improve battery life. Lastly, the battery life could also be increased by using more expensive lithium-ion batteries that are also rechargeable.

On the software side, the data we collect from users can be used for further analysis. We could help users find the route with least air pollution between two locations. Additionally, this data can be further utilized to categorize living conditions in different areas based on the air pollution.

References

[1] "How is air quality measured?," NOAA SciJinks – All About Weather. [Online]. Available: https://scijinks.gov/air-quality. [Accessed: 04-May-2022].

[2] "Some google street view cars now track Pollution Levels," Colorado Public Radio, 30-Jul-2015. [Online]. Available: https://www.cpr.org/2015/07/30/some-google-street-view-cars-now-track-pollution-levels/. [Accessed: 04-May-2022].

[3] "How cities are using the internet of things to map Air Quality," How Cities Are Using the Internet of Things to Map Air Quality. [Online]. Available: https://datasmart.ash.harvard.edu/news/article/how-cities-are-using-the-internet-of-things-to-map-air-quality-1025. [Accessed: 04-May-2022].

[4] "Carbon Dioxide Health Hazard Information Sheet." [Online]. Available: https://www.fsis.usda.gov/sites/default/files/media_file/2020-08/Carbon-Dioxide.pdf. [Accessed: 05-May-2022].

[5] Kidde, "Carbon monoxide levels that sound the alarm," Carbon Monoxide Levels. [Online]. Available: https://www.kidde.com/home-safety/en/us/support/help-center/browse-articles/articles/what_are_the_carbon_monoxide_levels_that_will_sound_the_alarm_.html. [Accessed: 04-May-2022].

[6] "MN1604 size: 9V (6LR61) - Duracell." [Online]. Available: https://www.duracell.com/wp-content/uploads/2016/03/MN1604_US_CT1.pdf. [Accessed: 05-May-2022].

[7] "Mqunifiedsensor," MQUnifiedsensor - Arduino Reference. [Online]. Available: https://www.arduino.cc/reference/en/libraries/mqunifiedsensor/. [Accessed: 04-May-2022].

[8] "Electrical engineer internship salary (April 2022) - zippia | average ..." [Online]. Available: https://www.zippia.com/electrical-engineer-internship-jobs/salary/. [Accessed: 05-May-2022].

[9] "IEEE code of Ethics," IEEE. [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8.html. [Accessed: 04-May-2022].

[10] "The code affirms an obligation of computing professionals to use their skills for the benefit of society.," Code of Ethics. [Online]. Available: https://www.acm.org/code-of-ethics. [Accessed: 04-May-2022].

[11] "Android 6.0 changes: android developers," Android Developers. [Online]. Available: https://developer.android.com/about/versions/marshmallow/android-6.0-changes#behavior-hardware-id. [Accessed: 04-May-2022].

[12] "MQ9B datasheet," [Online]. Available: https://cdn.sparkfun.com/assets/d/f/5/e/2/MQ-9B_Ver1.4__-_Manual.pdf. [Accessed: 05-May-2022].

[13] "MQ2 datasheet." [Online]. Available: https://cdn.sparkfun.com/assets/3/b/0/6/d/MQ-2.pdf. [Accessed: 05-May-2022].

[14] “The best time to visit Champaign, IL, US for weather, Safety, & Tourism,” Champion Traveler. [Online]. Available: https://championtraveler.com/dates/best-time-to-visit-champaign-il-us. [Accessed: 04-May-2022].

[15] “Datasheet SGP30 sensirion gas platform - adafruit industries.” [Online]. Available: https://cdn-learn.adafruit.com/assets/assets/000/050/058/original/Sensirion_Gas_Sensors_SGP30_Datasheet_EN.pdf. [Accessed: 05-May-2022].

Appendix A Requirement and Verification Table

Requirements	Verification	Verification Status
The 9V input must come from three 3V button cells to keep the design compact. This voltage should not drop below 5V as we use a Buck converter to step the voltage down.	a) The 9V upper limit can be verified using a voltmeter and a fresh set of cells by measuring the voltage across them b) The 5V lower limit can be verified using a voltmeter and a discharged set of cells by measuring the voltage across them	Yes
When the band is not being used, there needs to be a switch between the battery and voltage regulators to turn off the device and conserve power.	a) Operation of the power switch can be verified using continuity tests in a multimeter.	Yes
The sensors expect a 5V input with a tolerance of ±0.1V from the voltage regulator.	a) Place the positive prong of the oscilloscope at the end of the 5V regulator and the negative prong at the common ground. b) Monitor the potential difference between the two prongs to make sure the voltage is 5±0.1V over time.	Yes
ESP32 expects an input of 3.3V but can accept between 2.2 to 3.6V from the voltage regulator.	a) Place the positive prong of the oscilloscope at the end of the 3.3V regulator and the negative prong at the common ground. b) Monitor the potential difference between the two prongs to make sure the voltage is between 2.2 and 3.6V over time.	Yes

Table 4: Power subsystem RV table

Requirements	Verification	Verification Status
The power indication LED needs to turn on when the device is switched on.	a) Measure the potential difference across the switch's end and ground using a voltmeter. b) Visually check if the power indicator LED turns on when the switch is turned on.	Yes
The connection indication LED needs to turn on when the microcontroller finishes the setup. By default, the LED needs to shine blue upon turning on the band.	a) Connection status of the microcontroller can be monitored using print statements and the corresponding LED can be visually monitored to verify that it lights up blue.	Yes
The connection indication LED needs to change to a different color depending on the state of the connection. If a mobile device is not paired with the band, it should shine blue. Once a device pairs with the band it should shine green.	a) Connection status of the microcontroller can be monitored using print statements and the corresponding LED can be visually monitored to verify that it lights up green instead of blue when we connect a device to the band.	Yes

Table 5: Indicator subsystem RV table

Requirements	Verification	Verification Status
Read analog data from MQ-2 and MQ-9B sensors into digital data using analog input pins on ESP 32.	a) Read voltage output across the load resistor using an oscilloscope. Call this value V_read. b) We know that we are using a 12-bit ADC and the maximum value shows up for 5 V. Thus, the 0-5V output is linearly mapped between 0 and 2^12-1. Hence calculate the digital value using the formula V_read/5 * (2^12-1) c) Compare this value to the value read by the microcontroller using print statements.	No, measurements were too erratic to calculate fixed values
Convert sensor data into PPM measurements.	a) Convert voltage readings to PPM readings using the conversion graphs for each sensor. This can be done by linearizing the graph in 100 ppm segments so that we can map voltage values to approximate PPM value using 4.7KΩ load resistors for which the datasheet has mapped ppm to voltage outputs as seen in figure 8 (for MQ-9B) and figure 9 (for MQ-2).	Yes
Establish a successful I2C connection between ESP 32 and SGP30 to read data from the sensor.	a) Send wake-up requests and receive a confirmation to make sure we can communicate with the sensor.	Yes
Establish a successful Bluetooth connection between ESP 32 and the app to reliably send data over.	a) Since we are sending our sensor data in one packet every five minutes, we need the reliability of one packet every 5 minutes.	Yes
Ensure that the sensor data is sent to the app every 5 minutes unless the concentration exceeds safe limits.	a) Use a timer on ESP 32 to interrupt the normal flow of code to send collected data over Bluetooth b) Verify on the app that the packets are received every five minutes using assertions and timeouts	Yes

Table 6: Sensing subsystem RV table

Requirements	Verification	Verification Status
Connect the phone to the Bluetooth module of ESP32 MCU and periodically receive Bluetooth packets and unpack them without losing any data in the app.	a) Compare the values sent by microcontroller (using print statements to the serial monitor) and the values obtained by unpacking the Bluetooth packets.	Yes
The app should alert the user if the carbon dioxide and carbon monoxide values go above a certain threshold. Additionally, any detection of propane gas should be notified to the user.	Test 1: a) Send dummy test values from ESP32 MCU for each gas above and below their respective threshold values b) Verify that the app only notifies if these values cross the threshold. Additionally, we can also artificially simulate conditions that would cause the sensors to detect a high concentration of gasses to verify the app alerts. Such conditions include: Test 2: a) Place a candle on a table and take readings at multiple distances from the candle b) Verify that CO2 and CO readings reduce the further away we are from the candle using print statements Test 3: a) In a well-ventilated area open a propane tank and place the band close to it b) Verify that a notification appears on the app that propane was detected	Yes
The app should be able to communicate with the central server to send the local gas values data (if they cross the threshold) and current GPS location.	a) Send dummy test values from ESP32 MCU for each gas above and below their respective threshold values b) Verify that the app only sends a POST request to the server if the values cross the threshold. c) Verify if the values received by the server and the GPS coordinates match the values sent by the app. In the case where gas values for the same rounded off GPS coordinate exist on the server, verify if the value is updated with an average of the new and existing values.	Yes

The server should maintain a ledger of all the values sent to it from the app when contamination is detected. It should also round off the GPS coordinates by a predetermined degree to group the values in 100m radius together on the map to adhere to the safety regulation.	a) Send multiple dummy test values from the app to the server using POST requests and verify that these values are accurately recorded by the server using print statements. b) Verify that the rounding off code works as intended and only rounded off GPS values are stored in the ledger. In the case where gas values for the same rounded off GPS coordinate exist on the server, verify that it updates it with the average of the existing and current value.	Yes
The app should periodically fetch the ledger data from the server to display the contaminations on a human-readable map.	a) Inject the server with some dummy values and verify that the app automatically executes GET requests periodically. b) Ensure that the ledger on the server matches exactly with the local ledger after the GET request is fulfilled. Since the Google Maps API is going to be used to display the heat map, ensure that the coordinates from the ledger are accurately displayed on the map.	Yes
The app should alert the user if they enter a zone that was marked as contaminated by other users.	a) Inject the server with dummy test values of gas concentration that exceed the threshold values. b) Verify that the app rounds off the current GPS coordinates (by the same degree as done while posting the values) and checks for it in the obtained ledger. If the coordinates match up, verify that the app sends a notification.	Yes

Table 7: App subsystem RV table

In addition to these tests, we individually tested every single connection on the PCB board using a multimeter.

Appendix B Sensor plots

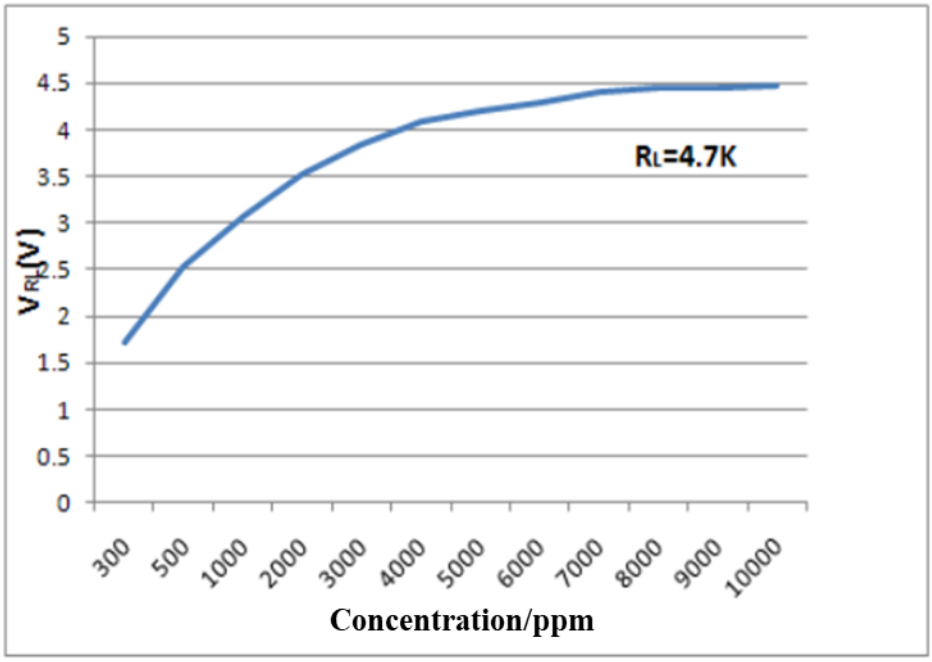

Figure 18: Voltage vs PPM from SparkFun's MQ9 datasheet [12]

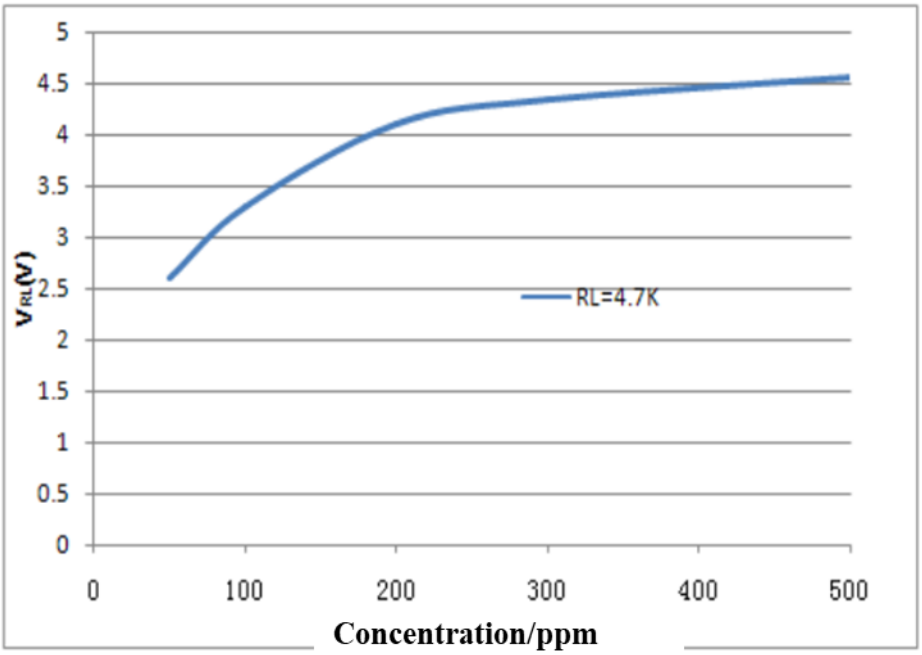

Figure 19: Voltage vs PPM from SparkFun's MQ2 datasheet [13]

Figures 8 and 9 show how the voltage across the sensor change with different gas concentrations

Appendix C Sensor Tolerance Analysis

For each sensor we have calculated the approximate error using graphs from their datasheet that catalog the relative readings based on various temperatures and humidity. Since we will be testing our bands in the spring in Champaign, we have assumed that the temperature will be between 15-25°C and the humidity will be around 55% [14]. To calculate the percentage error, we make use of the following formula:

$$\delta = \left|\frac{(measured\ value - absolute\ value)}{absolute\ value}\right| * 100$$

1. **MQ-2:**

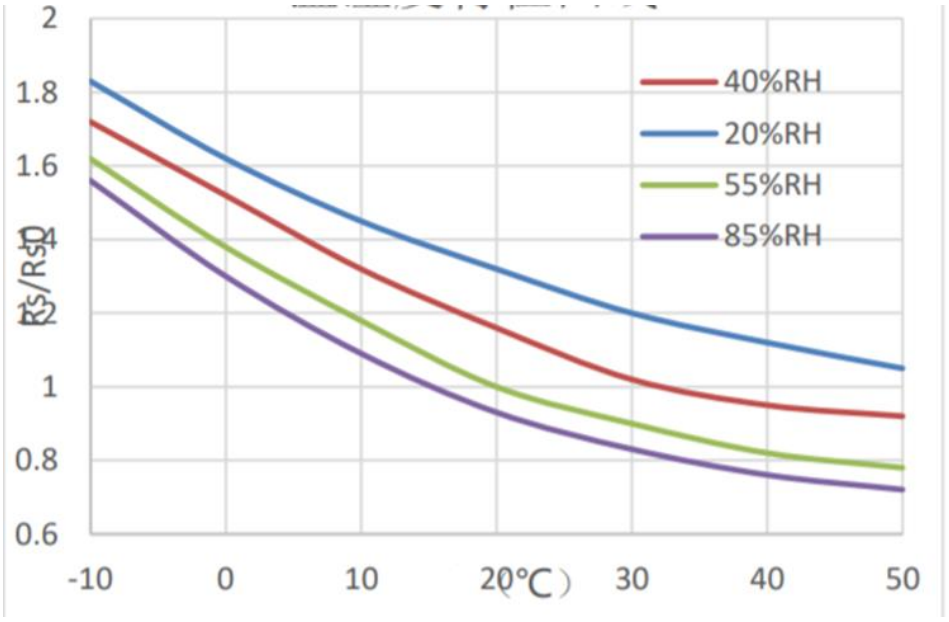

Figure 20: Graph of MQ-2's relative temperature/humidity characteristics from SparkFun's MQ2 datasheet [12]

The y-axis in figure 11 represents the ratio of Rs/Rso and the x-axis represents temperature. Rs (the measured value) is the resistance of the sensor in 2000ppm of propane in various temperatures and pressures. Rso (the absolute value) is the resistance of the sensor in 2000ppm propane under 20°C/55% relative humidity. Relative Humidity can be assumed to be 55% (the green curve) and operating temperature is 15-25°C.

Relative error at 15°C:

$$\left|\frac{Rs - Rso}{Rso}\right| * 100$$
$$= \left|\frac{Rs}{Rso} - 1\right| * 100$$
$$= |1.1 - 1| * 100$$
$$= \mathbf{10\%}$$

Relative error at 20°C:

$$\left|\frac{Rs - Rso}{Rso}\right| * 100$$
$$= \left|\frac{Rs}{Rso} - 1\right| * 100$$
$$= |1 - 1| * 100$$
$$= \mathbf{0\%}$$

Relative error at 25°C:

$$\left|\frac{Rs - Rso}{Rso}\right| * 100$$
$$= \left|\frac{Rs}{Rso} - 1\right| * 100$$
$$= |1.1 - 1| * 100$$
$$= \mathbf{10\%}$$

1. **MQ-9**:

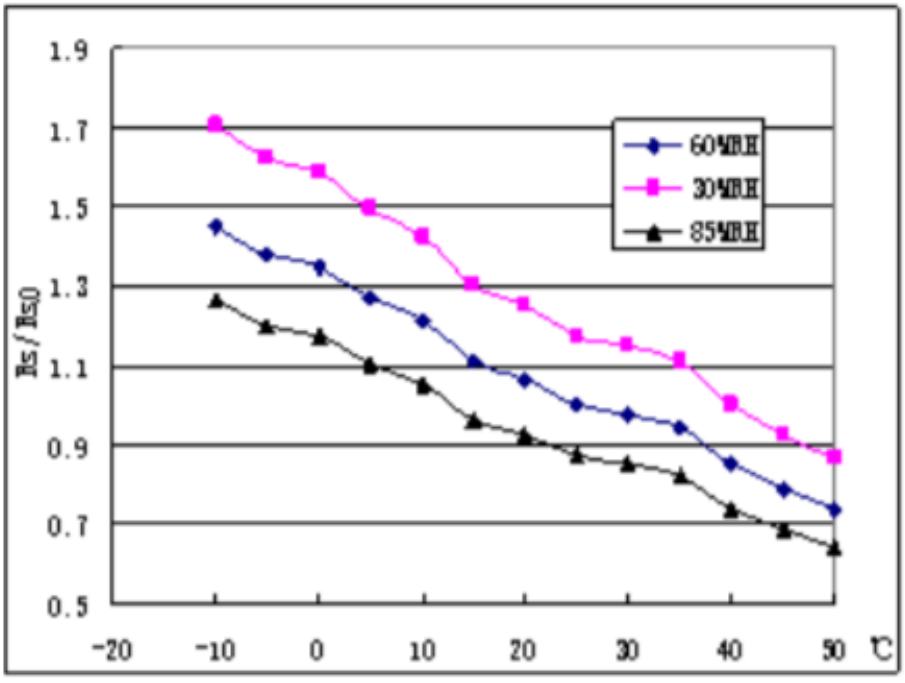

Figure 21: Graph of MQ-9's relative temperature/humidity characteristics from SparkFun's MQ9 datasheet [13]

The y-axis in figure 12 represents the ratio of Rs/Rso and the x-axis represents temperature. Rs (the measured value) is the resistance of the sensor in 150ppm of CO in various temperatures and pressures. Rso (the absolute value) is the resistance of the sensor in 150ppm CO under 20°C/55% relative humidity. Relative Humidity can be assumed to be 55% (the green curve) and operating temperature is 15-25°C. Relative Humidity can be assumed to be 60% (closest to 55%) (the blue curve) and operating temperature is 15-25°C.

Relative error at 15°C:

$$\left|\frac{Rs - Rso}{Rso}\right| * 100$$
$$= \left|\frac{Rs}{Rso} - 1\right| * 100$$
$$= |1.1 - 1| * 100$$
$$= \mathbf{10\%}$$

Relative error at 20°C:

$$\left|\frac{Rs - Rso}{Rso}\right| * 100$$
$$= \left|\frac{Rs}{Rso} - 1\right| * 100$$
$$= |1.07 - 1| * 100$$
$$= \mathbf{7\%}$$

Relative error at 25°C:

$$\left|\frac{Rs - Rso}{Rso}\right| * 100$$
$$= \left|\frac{Rs}{Rso} - 1\right| * 100$$
$$= |1 - 1| * 100$$
$$= \mathbf{0\%}$$

3. **SGP30:**

   The SGP30 assures an accuracy of 15% for carbon dioxide values in the measurable range of 0 - 60000ppm [15].

Appendix D Initial PCB Layout

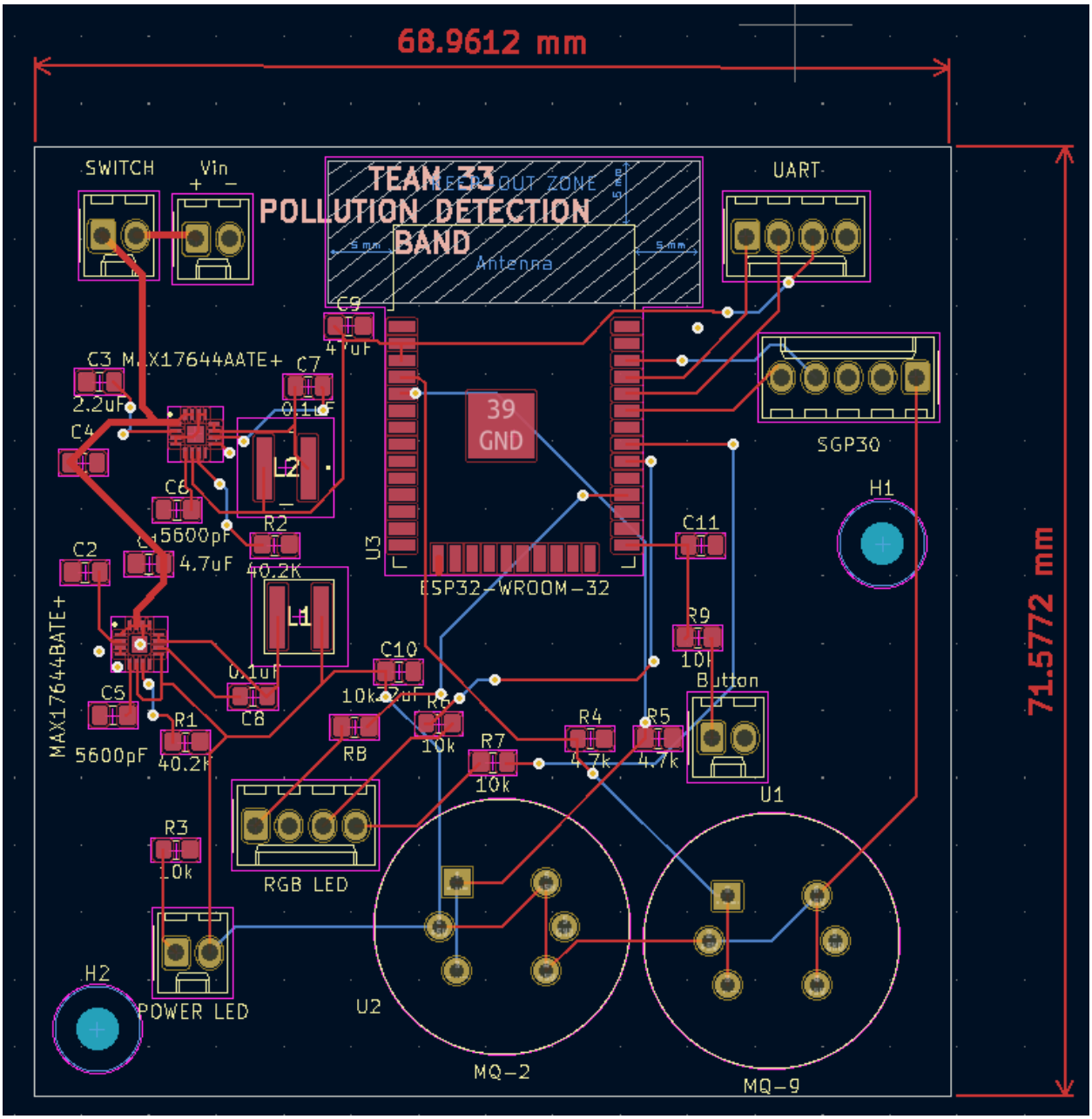

Figure 22: Original PCB design