CARBON CONTROL

By

Vikram Belthur

Mois Bourla

Tanmay Goyal

Final Report for ECE 445, Senior Design, Spring 2022

TA: Daniel Ahn

May 2022

Project No. 32

Abstract

The use of CO_2 as an air quality metric has increased in recent years. Indoor CO_2 levels above the atmospheric background concentration of above 400ppm are strongly caused by CO_2 exhalation from human occupants. This causes a measurable increase in the CO_2 levels. If these levels remain high over a large enough period, it is indicative that fresh outdoor air is not being ventilated into the space. Thus, the decay rates of CO_2 can be used as a measure of ventilation quality. Carbon Control is a scalable, multi-room, CO_2 monitor and alarm. The device will alarm when it detects high concentrations of CO_2. The alarm is both visual and auditory in nature. It exposes a web interface for remote monitoring and configuration.

Contents

1. Introduction

1.1 Problem and Solution Overview

Air quality for indoor spaces is critically important, and universally needed. Whether it is an office, a school, or a hospital, buildings where large amounts of people congregate need to be safe. A key component of this is indoor ventilation. If a space is under ventilated the safety of its occupants is compromised. The Wisconsin Department of Health reports that indoor CO2 concentrations between 1000-2000 parts per million (ppm) are associated with “complaints of drowsiness and poor air.”[1] While levels between 2,000–5,000 ppm indicate “stagnant, stale, stuffy air.”[2] Public health experts have recommended the use of CO2 monitoring in “assessing ventilation … in an effort to reduce the risk of disease transmission” (Allen et al., 2020). Allen et al. recommend measuring the decay of CO2 concentration to estimate how many times the air is replaced in a room [3]. This metric is known as the Air Changes Per Hour or ACH. These tests are conducted by artificially raising the indoor concentration and then removing the CO2 source to monitor the rate of decay.

Commercially available CO2 sensors can monitor CO2 concentrations and alarms to alert occupants of high concentrations. However, these solutions do not provide functionality that is needed for large scale deployments across a variety of indoor environments. Current devices are also not designed to monitor multiple zones within the same room. Lastly, commercially available devices cannot run automated ventilation tests to estimate the ACH of a space by taking advantage of changes in a room’s occupancy.

Carbon Control is a Wi-Fi enabled CO2 and occupancy sensing node. It can not only trigger alarms based on CO2 concentrations but communicate its readings to server for facilities personnel to track. Moreover, this device will support same room multi-zone deployments, where nodes in the same room will have synchronized alarms. Finally, the devices will leverage the occupancy sensor to detect recently emptied rooms. These rooms have high CO2 concentrations from their previous occupants and have decaying concentrations. This allows the device to automatically estimate the ACH.

1.2 Visual Aid

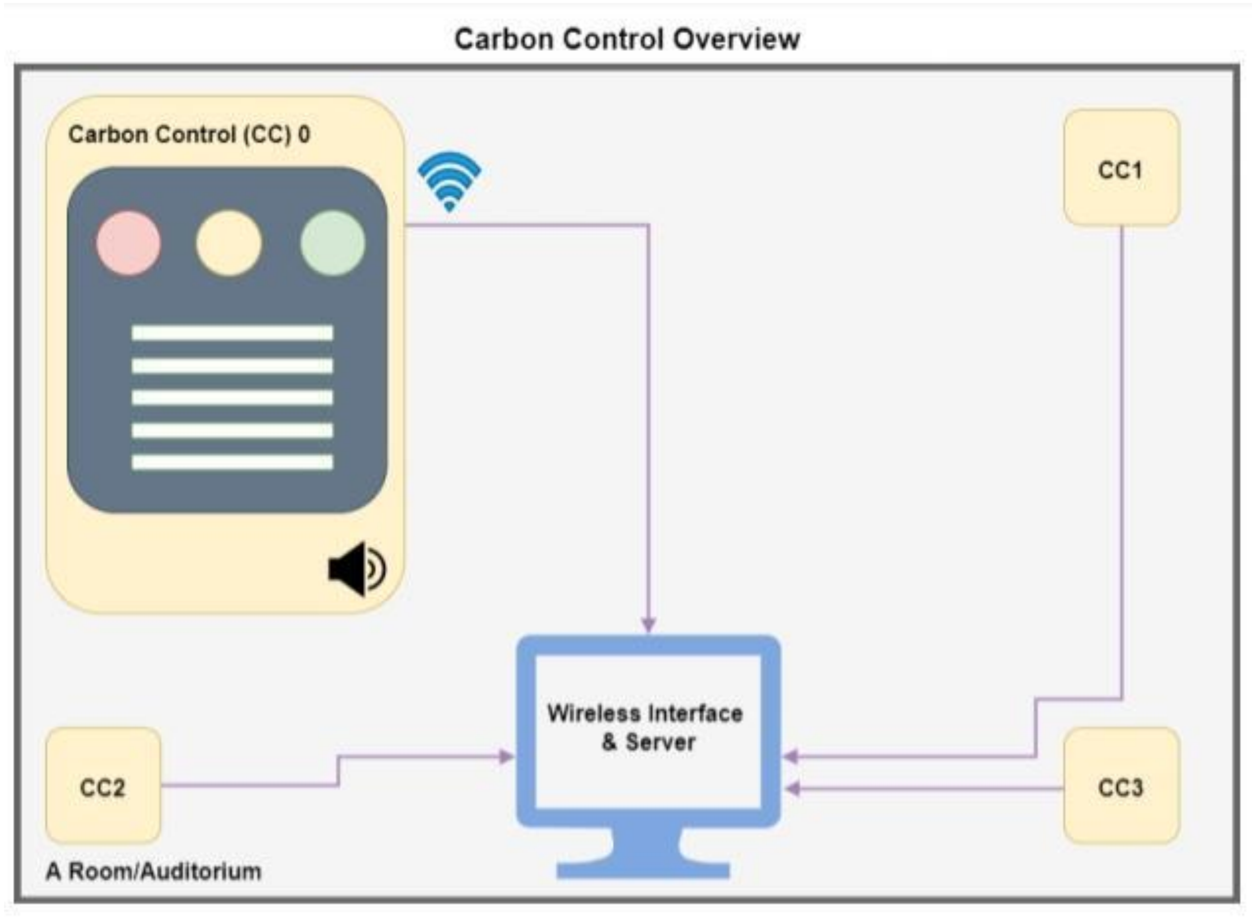


Figure 1 - Visual Aid

1.3 Physical Design

The physical design of the project is shown below, the project is a device that contains a PCB housed inside a square enclosure. The front side of the project depicts the LED (Light Emitting Diode) along with the mounted PIR sensor and CO2 sensor. The device also has an accessible USB port towards one side and has access holes for the internally placed speaker. The enclosure also has openings for the charging state indication LEDs.

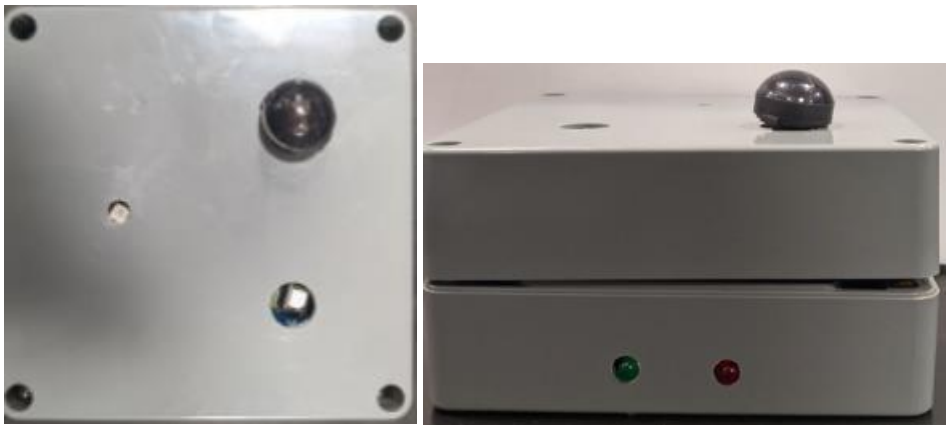


Figure 2 - CO_2 Sensor and PIR Sensor

2 Design

Our high-level design philosophy involved first defining various use cases for our product. This necessitating designating a set of actions our system had to make in response to various inputs or stimuli. We then derived high-level requirements from these use cases and ultimately individual subsystem requirements. This process ensures that all functionality central to the user experience is properly tested and verified.

2.1 Block Diagram

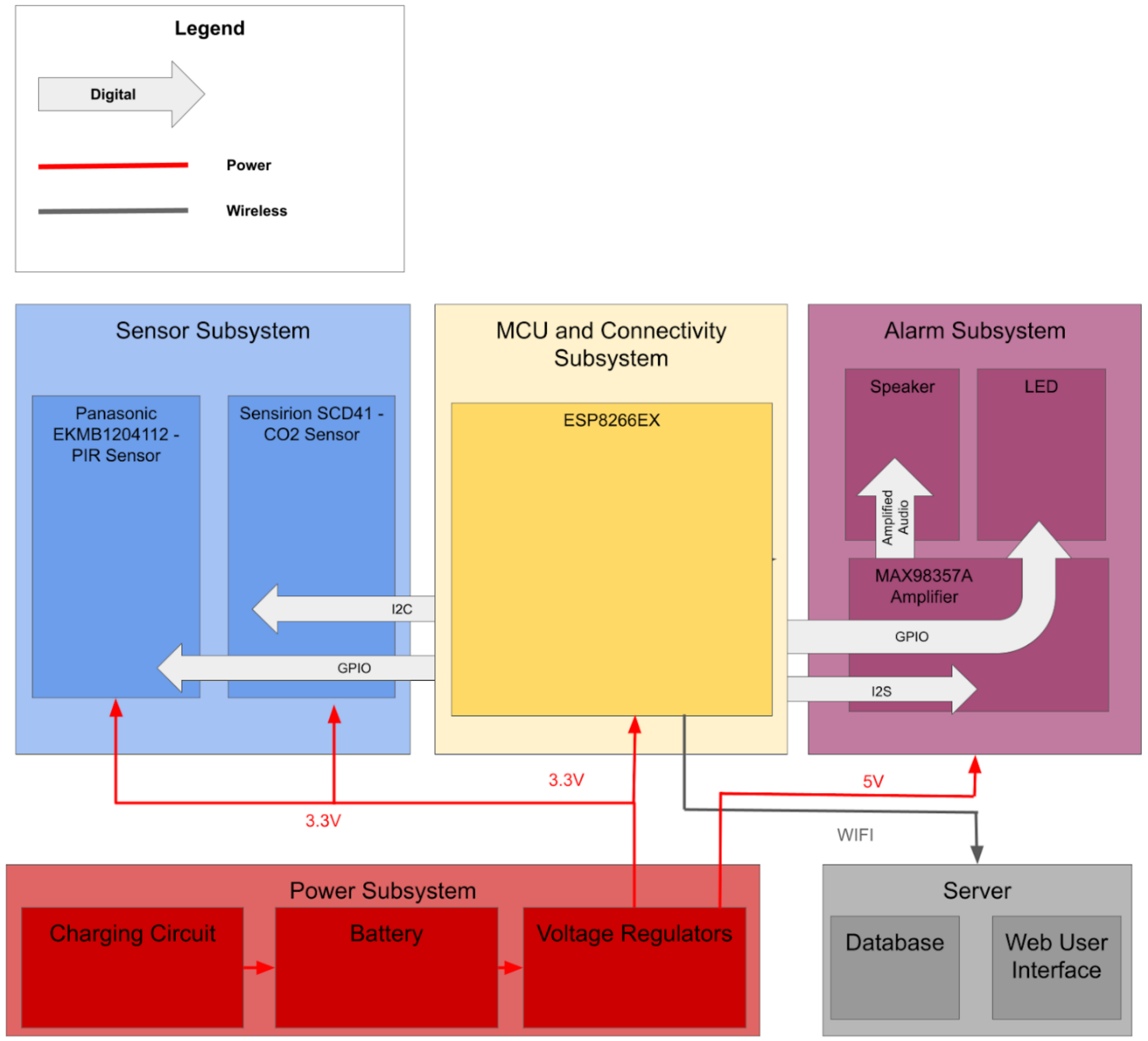


Figure 3 - Carbon Control Block Diagram

2.1.1 MCU Subsystem

A carbon control node is built around its microcontroller (MCU). The MCU serves as a connectivity hub for our device with all subsystems connected to it. As a result, the primary criteria for the MCU selection IO capabilities. We required sufficient IO to connect to one I2S bus, one I2C bus, and two GPIO pins for our PIR sensor and LED. Moreover, the MCU was selected to have wireless capabilities. Additional considerations included whether the system was easily programmable, power efficient, and computationally powerful. We ultimately selected the ESP32-WROOM-E as our MCU.

The ESP32-WROOM -E has sufficient IO to maintain an I2C bus for the CO_2 sensor, an I2S bus for the sound signal of the alarm system, and two GPIO pins, one for the PIR output and one to trigger the LED. Device libraries were used to interface with the CO_2 sensor as well as the speaker and LED. The MCU proved more than capable for our application. We did however run into computational difficulties. A large difficulty was the single threaded nature of our code. This meant that only one command could be executed at once. This problem manifested in the main loop of our program when the speaker output would block the execution of our wireless communications. This problem can be solved in future iterations by selecting a multi-core MCU with better support for multi-threading. The system's standby power draw was measured at an average of 40 mA (including inefficiencies of our power subsystem). WIFI performance was solid with no dropped packets observed during our testing, but it was limited to 2.4 GHz networks, and was unable to authenticate on to the IllinoisNet network.

2.1.2 Sensor Subsystem

Our device is to be capable of monitoring CO_2 levels in indoor space. To do this, we needed a sensor which is accurate, has a low response time, and can detect CO_2 at a high enough concentration. Our sensor is capable of detecting values from 400 to 5000 ppm. For this project we used the Sensirion SCD41-D-R2 as our CO_2 sensor. This part was chosen in part due to its sensing range, accuracy, and sufficient response time. This sensor communicated with the MCU through the I2C bus. It was also calibrated through I2C as well using manufacture provided programs. Calibration entailed exposing the sensor to concentrations of at least 400ppm once a week. Since the Carbon Control solution calculates the ACH of a particular room when it is empty, we used a PIR sensor to determine the occupancy of that room. We selected the EKMB1204112 PIR sensor manufactured by Panasonic since it had a maximum range of 12 meters, while also consuming very little power.

The performance of Carbon Control's sensor subsystem was satisfactory and there were no significant errors. The CO_2 sensor was able to measure the concentrations of CO2 quickly and accurately. When it was exhaled upon, to see its response to the diffusion of CO_2 gas, the sensor was able to detect both the local maxima and subsequent exponential decay in CO_2 concentration. The PIR sensor was able to detect motion in a room, to use it as a proxy for occupancy. When individuals were moving in the room the occupancy was set to true and the ACH for that room was not calculated. The CO_2 and PIR sensors were straightforward to program and were able to successfully integrate with one another. No major difficulties were faced in this area.

2.1.3 Alarm Subsystem

The alarm subsystem informed the occupants of a room if they are safe based on the level of CO2 in the air. The subsystem consists of an RGB LED which changes color-based CO2 concentration, and an alarm will sound upon excessive levels of the gas. To accomplish these objectives, we used an Adafruit Flora reprogrammable RGB LED, and a speaker used in conjunction in an amplifier (with an onboard DAC). The LED was connected to the MCU via a GPIO pin. The sound for the alarm was transmitted to the MCU via the I2S protocol, and the connection to the MCU was three GPIO pins. The alarm subsystem was powered by the 5V rather than the 3.3V rail because the

For the project, we opted to use the MAX98357 audio amplifier by Maxim. The amplifier integrates a digital audio interface, among other features, and a DAC on-chip. The alarm tone itself was a simple sine wave (A440) generated by the MCU. The thresholds we used for the alarm and LEDs were a green light if the gas concentration was beneath 1000ppm. If the concentration was between 1000 and 1300 ppm, the LED shone yellow. If the concentration of CO_2 exceeded 1300 ppm, the LED emitted a red color. Any CO_2 concentration more than 1400 ppm would trigger the alarm. This is called the local alarm setting. The server can also send an alarm signal to the MCU, thereby triggering it to sound on the speakers. This would typically be done in a multi-zone configuration, with multiple Carbon Control devices, when one of them has detected a spike in CO_2 levels, but the other has not.

The overall performance of the alarm subsystem was satisfactory. The LED consumed on average between 40 – 60 mA depending on the color being emitted. When touching the LED, it did not get excessively hot, eliminating a source of potential hazard for this part of the project. In terms of the sound emitted by the speaker, it was generated via a third-party library, where the parameters we could change were the frequency (i.e., the note) and the sampling rate. Perhaps in future implementations of this project, a different configuration and be used, to make the sound more agreeable to the audience. This could be done with harmonics. An idea of merit is to save a warning audio message on the MCU's flash storage and have that be outputted by the speaker. A more powerful speaker can also be considered if this project is to be deployed to a large hall or space.

2.1.4 Power Subsystem

The power subsystem has two objectives. First, it must charge our battery cell from power sourced from USB. The battery charging circuit is designed to charge our battery at a controlled maximum 500mA. The charging IC in our final design is the MCP73831. The chip was selected because it was cost effective compared to more specialized ICs that enable balance charging. The current limit was set to 500mA to remain in compliance with the USB specification by a feedback resistor. An additional feature that is used is the charge status signal that is reported to a GPIO pin on the MCU. The signal is converted into a visual cue by two LEDs that are wired such that one is reverse biased, and one is forward biased depending on the charge status.

A PMOS transistor is used to switch between battery power and USB power. This helps conserve battery life by disconnecting the battery when it is not needed. The battery voltage is boosted by a boost converter to 5V, as our alarm subsystem requires 5V for our amplifier and LED. We used the TPS61032 as our boost converted because of its high efficiency and its 5V output capability. We used a

5V rail as it would be more efficient than a simple 3.3V for these higher current draw applications. It is rated at 1A, which is sufficient for our applications. The maximum current drawn by our device during a standard operating regime is 560 mA. We also set our system to output a low battery signal at 3.3V. In practice this occurred at approximately 3.35V. The output of our 5V rail was set by two feedback resistors. We observed that our 5V rail was at approximately 4.98V under light load. This satisfied our requirements. These deviations are also expected as we had to compute the various resistors used for the feedback and low battery network. The relevant equations are included below. A specific challenge was that exact values could not be obtained so rounding was used, this contributed to our 5V rail not being 5V but our system was tolerant to this.

$$R7 = R9 * \left(\frac{V_O}{V_{FB}} - 1\right) = 180k\Omega * \left(\frac{V_O}{500mV} - 1\right) \tag{1}$$

$$R1 = R2 * \left(\frac{V_{BAT}}{V_{LBI-threshold}} - 1\right) = 390k\Omega * \left(\frac{V_{BAT}}{500mV} - 1\right) \tag{2}$$

A 3.3V rail is maintained by the NCP1117 3.3V low drop-out regulator. This device was chosen for lower power ICs. It maintained a rail at approximately 3.28V - 3.32V depending on the status of the 5V that feeds it. The board was designed on a split power plane system on the second inner copper layer to assist in routing.

2.1.5 Server Subsystem

The server subsystem was written in the Flask framework using python. It also utilized a MySQL database to hold past readings. The server received data from the device and uploaded an internal representation as well as graphs automatically. The server was able to hold more than eight hours of recorded data which was collected in one sitting. The server was able to display the data in a graphical format. The plots were generated using matplotlib in python. They were then exported to an image and displayed using HTML and CSS. The server also successfully triggered the alarm on our device which validated several of our requirements. While the requirements were met for our server subsystem, there are a few improvements that would be useful for our project. Firstly, we are using a static template instead of a dynamic webpage. This means that our website needs to be refreshed to update the content. This was mitigated by auto-refreshing every 59 seconds, but that is inelegant. Our server used approximately 100kB of storage space to record approximately 8 hours of data. This reflects the use of additional fields for our database versus proposal specifications.

2.1.6 Layout

We designed our board with a compact layout. Our board was built on a PCB, including a split power plane and a ground plane. The board dimensions were 85mm by 85mm. A central cut out was added to the second revision of our PCB to aid in WIFI signal integrity. Our board also feature a controlled impedance differential trace for the USB data lines which was carefully routed to assure signal integrity.

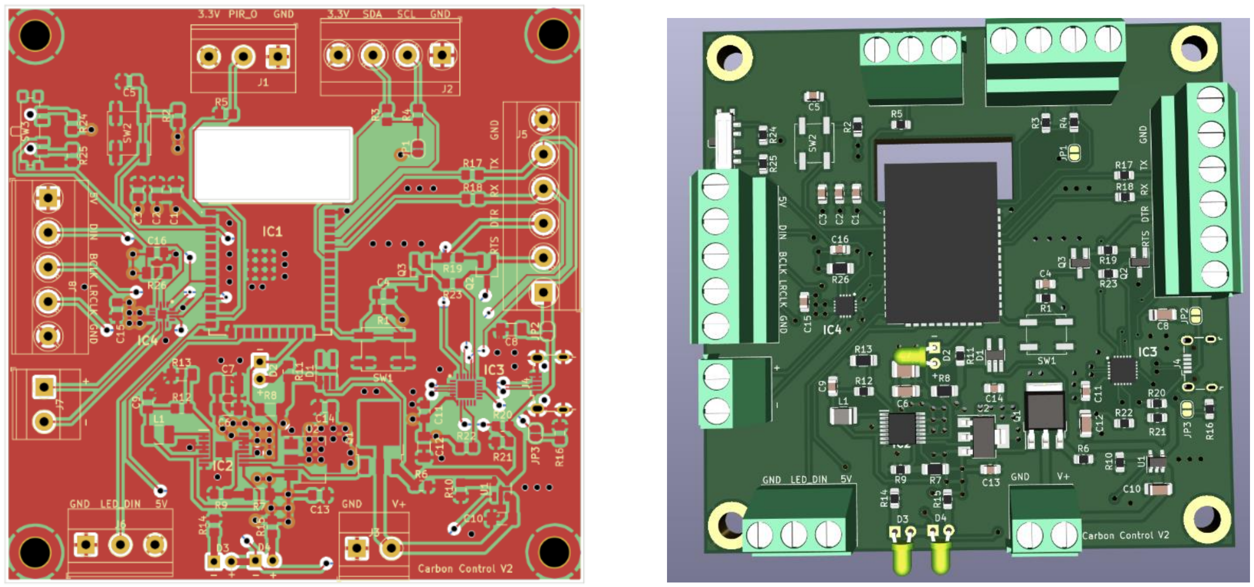
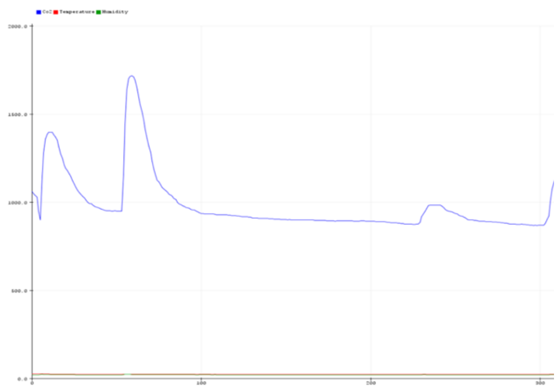


Figure 4 - Carbon Control PCB Layout and Render

3. Design Verification

3.1 Sensor Subsystem

The first order of testing used to verify the integrity of these subsystems involved using a digital multimeter (DMM) to measure that the sensor was receiving the proper voltage (3.3V) and drawing a reasonable amount of current. The DMM was also used to check for short circuits using the continuity setting. Once this was verified, we tested the performance of the sensors to make sure we could use them for our device. These tests were performed on both our prototype and final PCBs. For the CO_2 sensor we devised a simple test to ascertain whether the sensor was working or not. To create a local hotspot of gas concentration one of our team members would exhale on the sensor. Since exhalation is a source of CO_2, the gas's concentration spiked. After causing a local spike in CO_2 levels, the gas was then allowed to diffuse. This allowed us to then monitor whether the response time was sufficiently low. The figure below shows the CO_2 concentration over a five-minute period. We can estimate that the response time, the time it takes for concentration to change by more than 25%, will be vastly under two minutes. Based on this graph, we can estimate the response time to be around nine seconds.



To verify the PIR sensor was working properly, we used a DMM to probe the output of the sensor. When there was motion detected, the output pin was high, if there was no motion, the output pin was low.

3.2 MCU Subsystem

Many of the MCU requirements are verified through the overall function of our project. The MCU can send POST requests with our data to our server, the is verified by the graph displayed on our server, which is populated by our MCU through POST requests. We also verified that our MCU was reading the sensor properly by causing a local bolus of CO_2 to spike the sensor reading and verifying the increase was reported to our server. This verified that MCU was able to read the sensor. In response to the bolus, the alarm sounded with a tone of frequency 440Hz, above our required minimum of 80Hz. This verified the MCU's ability to produce a sound output. To test the timing abilities of our device, we programmed the device to make a transmission every 5000 milliseconds. The transmission included the current time of transmission and so we were able to validate the performance of our device by comparing the delta between them. The transmissions had a period of 5002 – 5010 ms which suggests that the average millisecond tick was within 1% of a millisecond, well in compliance with our 50% target. The final requirements involved the ability to classify CO_2 into three distinct zones. This was verified by testing in a low CO_2 environment such that the readings are classified in a "green" zone. We confirmed that the LED is green and then breathed onto the sensor causing the LED to show both "yellow" and "red" zone warnings.

3.3 Alarm Subsystem

Alarm verification was performed along with the MCU and sensor subsystem verifications. The team first checked for short circuits and the correct voltage supplied (5V) using the DMM. The alarm subsection was able to produce a 440 Hz tone, and this was successfully outputted to the speaker. We tested the speakers by playing pre-recorded sound from a .wav file, and by generating the sine wave to play the tone. The LEDs were tested to output various colors, and these tests were broadly successful. The LED can be seen working below.

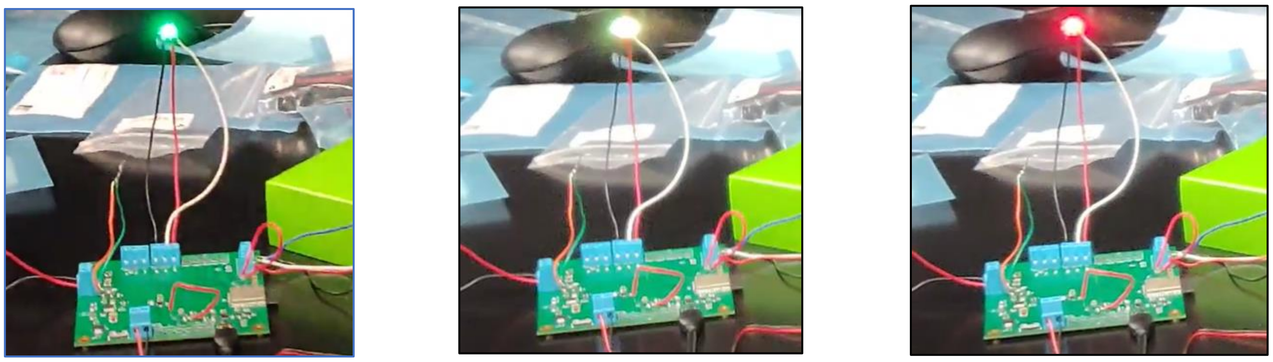
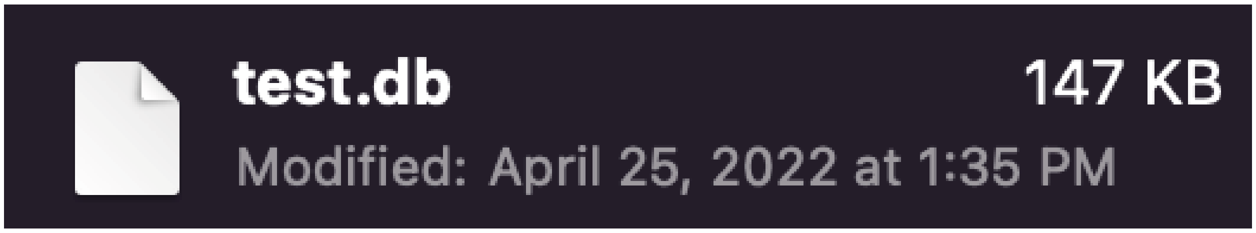


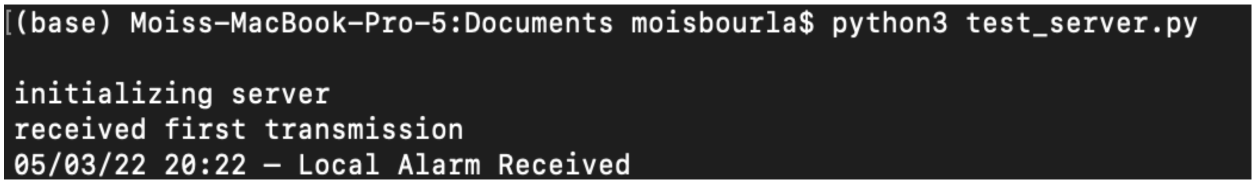
Figure 6 - Alarm Subsystem LEDs

3.4 Server Subsystem

The server subsystem requirements were also verified rigorously. Our device can post data to the server. This is verified not only during regular operation, when our device is uploading readings from the sensor. To test the server's ability to POST we had it trigger an alarm on the device via POST request, this was successful. Our server's ability to hold long term data was verified by continuously collecting 6 hours of data during the day of our demo. This is more than our 20-minute requirement. The total file size grew to over 147KB as we were recording additional data. During our demo, we also showed our server refreshing every 10 seconds. This time between refreshes is user configurable so any time under 60 seconds would satisfy our requirements. The time between refreshes was verified using console logs. Finally, to compute the ACH, we selected two data points on the curve that produced a valid ACH and compared it to our system's estimate. Since our system applies a formula directly, there was no error satisfying our requirement.



test.db 147 KB
Modified: April 25, 2022 at 1:35 PM



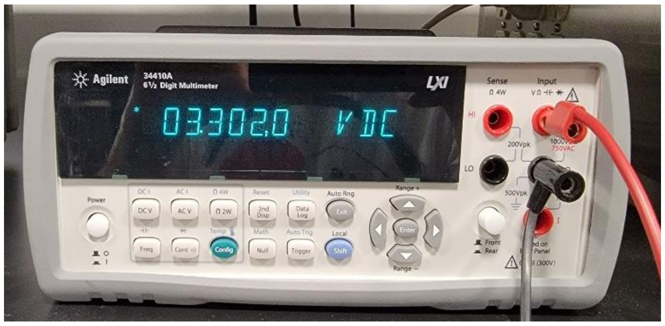
```
[(base) Moiss-MacBook-Pro-5:Documents moisbourla$ python3 test_server.py

initializing server
received first transmission
05/03/22 20:22 — Local Alarm Received
```

Figure 7 - Carbon Control Server Test

3.5 Power Subsystem

Since the power subsystem serves as the bedrock for this project, this section was tested and verified thoroughly. The board was connected to a power supply in the lab and the 3.3V and 5V outputs were probed with a DMM. Our boost converter and regulator consistently output a stable voltage of approximately 5V (4.98V) and 3.3V (3.32V) respectively.



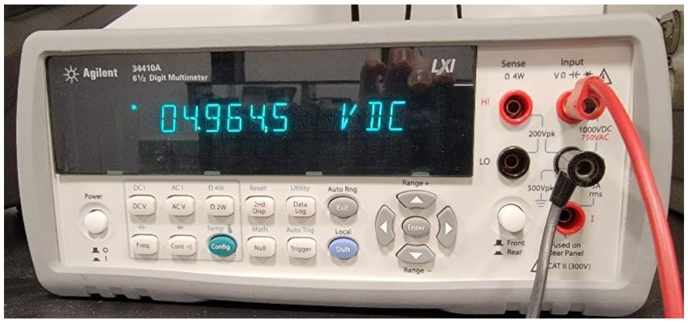
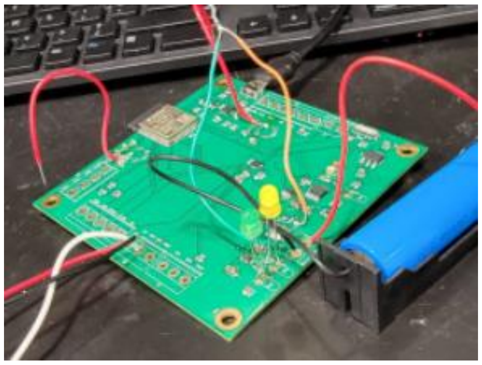


Figure 8 - Power Rail Voltages

A second set of tests was to verify that charge status LEDs were displaying yellow and green for charging and charged. We measured the battery voltage using the DMM in the lab, and when the battery was below 4.1V yellow LED was lit, when the battery was charged the green LED was lit. Yellow and Green LEDs can be seen respectively below.



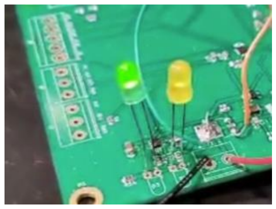
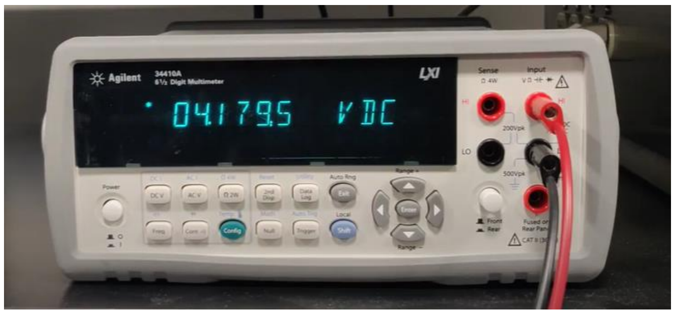


Figure 9 - Charge Status LED

A final test was to verify whether the battery life was indeed capable of lasting for eight hours. We set up Carbon Control in the lab (with a fully charged battery) and connected a DMM to it to monitor battery life. We then left the device for approximately eight hours. The battery voltages are measured below. At 3.75V, our battery voltage is still able to be boosted to 5V and stepped down to 3.3V allowing for normal operation. Both measurements are taken under load.



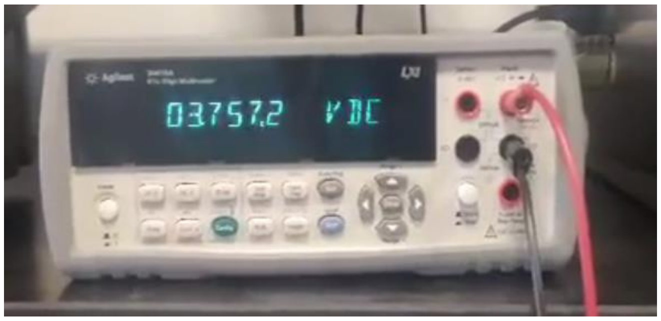


Figure 10 - Battery Discharging

4. Costs

To determine the cost of this project we need to add the total cost of all the parts as well as the cost of our labor.

4.1 Parts

Table 1: Parts Costs

Part	Manufacturer	Retail Cost ($)	Bulk Purchase Cost ($) /Unit	Actual Cost ($)
MCU	Expressif	3.9	3.9	3.9
CO2 Sensor	Sensirion	52.18	52.18	52.18
PIR Sensor	Panasonic	28.29	21.745	28.29
Amplifier	Adafruit	2.84	2.84	2.84
RGB LED	Adafruit	7.95	7.16	7.95
Boost Converter	Texas instruments	0.61	0.525	0.61
Voltage Regulator	STM Microelectronics	0.77	0.633	0.77
USB charging IC	Microchip	0.69	0.52	0.69
USB to Serial Bridge	Silicon Labs	2.66	0.88	2.66
Resistors [28 units]	Various	4.6	0.62	4.6
Capacitors [30 units]	Various	14.15	1.25	14.15
Total		**118.64**	**92.253**	**118.64**

Table 1 - Part Costs

4.2 Labor

All three of the members of this group are in electrical engineering, and according to the latest data available to us from the ECE department (2020), the average salary was $79,714 USD/year [13]. Assuming we work a standard 40 hours/week in a 52-week year, with no overtime pay, our hourly wage corresponds to 38.32 USD/hr. We worked on this project for an average of 10 hours each week for 10 weeks of the semester. That is a total of 100 hours. Using the formula given to us:

$$38.32\left[\frac{USD}{hr}\right] * 2.5 * 100[hrs] = 9580\left[\frac{USD}{person}\right] => 9580\left[\frac{USD}{person}\right] * 3 = \$28{,}740$$

To get the total cost we sum the parts and the labor to get a cost of $28,858.64.

5. Conclusion

5.1 Accomplishments

Some of the achievements unique to our project were firstly being able to demonstrate the utility of a product like our device in a real-life setting. We were able to notice how high the CO2 concentrations were inside the ECE building and this was a surprising find. Another achievement was to design a unique method to alarm individuals of the CO2 concentration that we designed. The alarm was designed to be unique to not be confused with other alarms like that for a fire. The alarm also issues warnings in stages to help mitigate the issue before reaching a critical point.

5.3 Ethical considerations

To ensure no electrical hazard was encountered we were careful when utilizing the electrical test equipment, as needed, and whenever we were present in the lab. During the drilling of our enclosure, we wore safety glasses and used a clamp to mount the device safely to a desk. During soldering we were careful to be in an attentive state of mind while using the solder iron and the reflow oven. A potential safety issue pertaining to individuals using our device could be in our device's alarm. We isolated its testing to prevent confusion and unnecessary panic that could ensue if it were to be sounded without warning. The sound was designed to be unique enough to not be confused with other potential alarms like fire safety. The device is electrically isolated to reduce electrical hazards. To conform to the IEEE code of ethics (1.1),[14] this project made improvements on the existing infrastructure, and results from the project have been handled fairly and openly to not cause harm during implementation. Data privacy regarding the room occupancy is important and has been safeguarded, to comply with the ACM's codes of privacy (1.6) and security (2.9) [15].

5.4 Future work

The future works include trying to make the device more compact. This would include designing a smaller PCB to be placed in a smaller enclosure. The device would also have to be more lightweight. We also plan to develop novel methods to make multiple devices communicate with each other and share data intelligently to make the background CO2 levels more accurate along with a CO2 density map in our web server for each room we place the CO2 devices in.

References

[1] Wisconsin department of health services, “Carbon dioxide guidelines” [Online] Available: https://www.dhs.wisconsin.gov/chemical/carbondioxide.htm

[2] Minnesota department of health, “Carbon dioxide guidelines” [Online] Available: https://www.health.state.mn.us/communities/environment/air/toxins/co2.html

[3] Joseph Allen, Jack Spengler, Emily Jones, Jose Cedeno- Laurent, “Guide to measuring ventilation rates in schools,” Harvard T.H Chan School of Public Health [Online] Available: **https://schools.forhealth.org/wp-content/uploads/sites/19/2020/08/Harvard-Healthy-Buildings-program-How-to-assess-classroom-ventilation-08-28-2020.pdf** [Accessed : August 2020]

[4] “Microcontroller unit ESP32-WROVER-E datasheet”, Adafruit [Online] Available: espressif.com/sites/default/files/documentation/esp32-wrover-e_esp32-wrover-ie_datasheet_en.pdf

[5] “CO2 Sensor unit SCD41-D-R2 datasheet”, Sensirion.com [Online] Available: **https://sensirion.com/media/documents/C4B87CE6/61652F80/Sensirion_CO2_Sensors_SCD4x_Datasheet.pdf**

[6]” PIR Sensor EKMB1204112 datasheet”, media.digikey.com [Online], Available: **https://media.digikey.com/pdf/Data%20Sheets/Panasonic%20Sensors%20PDFs/EKMB_MC_AMN2_3_Rev_Sep_2012.pdf**

[7] “Amplifier MAX98357A datasheet”, Adafruit.com [Online] Available: **https://cdn-shop.adafruit.com/product-files/3006/MAX98357A-MAX98357B.pdf**

[8] “RGB LED ADAFRUIT FLORA”, Adafruit.com [Online] Available : https://www.adafruit.com/product/1260#technical-details

[9] “Boost converter TPS613222ADBV datasheet”, Texas instruments [Online]Available:**https://www.ti.com/general/docs/suppproductinfo.tsp?distId=10&gotoUrl=http%253A%252F%252Fwww.ti.com%252Flit%252Fgpn%252Ftps61322**

[10] “Voltage regulator LD1117V33 datasheet”, STMicroelectronics. [Online]Available: **https://www.mouser.com/datasheet/2/389/cd00000544-1795431.pdf**

[11] “USB charging IC MCP73831T-2ACI/OT datasheet”, Microchip. [Online]Available: **https://ww1.microchip.com/downloads/en/DeviceDoc/MCP73831-Family-Data-Sheet-DS20001984H.pdf**

[12] “USB to Serial Bridge CP2104-F03-GMR datasheet”, SiliconLabs. [Online]Available:
https://www.silabs.com/documents/public/data-sheets/cp2104.pdf

[13] “Illini success survey 2020-21 ”, UIUC BOX. [Online] Available:
https://uofi.app.box.com/s/aoply09y5kf6i36es8v3bl758n2lfl08

[14] “IEEE Code of Ethics” IEEE [Online] Available:
https://www.ieee.org/about/corporate/governance/p7-8.html

[15] “ACM (Association for Computing Machinery) Code of Ethics” ACM [Online] Available:
https://www.acm.org/code-of-ethics

Appendix A Requirement and Verification Table

1. Power Subsystem

Requirements	Verification	Verification Status (Y/N)
1. The device will maintain a low voltage power rail (3.3v +/- 0.3v) and a high voltage power rail (5v +/- 0.3v), when powered by an input voltage between 3.5v to 4.2v.	1A. Connect the battery input terminals to a lab-based power supply. Connect a multimeter to the output of the 3.3v supply rail. 1B. Set the output voltage of the power supply to 6.90v +/- 0.05v and turn on the output. 1C. Verify that the regulator output voltage is between 3v and 3.6v. Increase the power supply voltage by 0.1v increments until output voltage is 7.50v +/- 0.05v, confirming that the rail voltage is between 3v and 3.6v at each increment. 1D. Repeat the procedure for the 5v supply rail, verifying that the rail voltage is between 4.7v and 5.3v.	Yes
2. The device will power on its power status LED red while the battery is charging, until the battery reaches its maximum voltage, 4.1v +/- 0.15v.	2A. Attach multimeter probes to the terminals of a battery with voltage <3.95v. Initiate charging by plugging in usb power. 2B. Verify that the device's status LED is red while the device's terminal voltage is below 4.1v +/- 0.15v. For this test to fail, the battery would need to exceed 4.25v and the LED continues to be red, alternatively, it would need to turn off the red LED while the battery voltage is below 3.95v.	Yes
3. The device's green power LED will be turned on after the	3A. Attach multimeter probes to the terminals	Yes

battery is finished charging, i.e., battery voltage is 4.1v +/- 0.15v.	of a battery with voltage <3.95v and green power status LED is off. Initiate charging by plugging in USB power. 3B. Monitor the device until the green power LED is on. Ensure that the battery voltage is between 3.95v and 4.25v when this occurs.	
4. The device can operate for 8 hours on a single battery charge.	4A. Power on a fully charged device and begin a stopwatch. 4B. Trigger a local alarm through breath 4C. After the alarm has been active for 2 minutes (either through speaker tone or LED indication), remove the alarm stimulant by ventilating the area with a fan. 4D. Monitor the device every 30 minutes to ensure that the device is still powered on until it is not powered on, or eight hours have elapsed, whichever occurs first.	Yes

2. Sensor Subsystem

Requirements	Verification	Verification Status (Y/N)
1. The device must be able to detect a human body in motion (0.8 - 1.5 m/s), within two meters of the device.	1A. Configure the MCU to print “motion identified” to the command line if the PIR sensor is activated. 1B. Walk in an arc around the device such that the distance to the device is approximately two meters. The distance will be measured by measuring tape. 1C. Verify that motion was detected by inspecting the command line.	Yes
2. The CO_2 sensor will have a response time of two minutes or less. The response time is defined as a	2A. The device will be configured to report the CO_2 concentration every five seconds to the command line. 2B. A baseline concentration will be	Yes

greater than 25% change in concentration.	taken as an average concentration 2C. A team member will breathe onto the sensor. 2D. The amount of deviation after breathing onto the sensor will be noted and the deviation should occur from the starting value by 25% or more. 2F. We will verify that the time taken for the deviation is less than or equal to two minutes.	

3. MCU

Requirements	Verification	Verification Status (Y/N)
1.The MCU shall send a POST request to a web server through WIFI.	1A. Configure the web server to have an endpoint that can accept POST requests, which displays incoming request payloads. 1B. Program the MCU to make a POST request with a known message up to 25 characters. Execute the program. 1C. Verify the received data is what was transmitted on the server.	Yes
2. The MCU shall receive a POST request from the web server through WIFI.	2A. Configure the web server to send data through a POST request payload. 2B. Program the MCU to receive a POST request and print the payload to the command line. Execute the program. 2C. Verify the received data is what was transmitted from the server.	Yes

3. The MCU shall read the CO_2 sensor readings over an I2C bus.	3. Configure the MCU to read the sensor value and print the value read. Verify that a value between 400 and 5000ppm is read.	Yes
4. The MCU shall be able to transmit a sine wave tone with a frequency above 80Hz over I2S for at least 15 seconds.	4A. The MCU will be programmed to play a sine wave tone with a frequency above 80Hz over its I2S bus. 4B. An oscilloscope will have its probes on the positive and negative terminals of the speaker output. 4C. Verify that a sinusoidal wave is visible on the oscilloscope for at least 15 seconds.	Yes
5. The MCU shall maintain a timer with millisecond ticks (+/- 0.5ms).	5A. The MCU will have a timer initiated with 1 millisecond increments and set to run for 10^5 iterations. As the program is triggered to run (within 2 seconds), a secondary timer will be initiated on a stopwatch. 5B. Verify that the elapsed time is within 50-150 seconds +/- 2 seconds. The additional increment accounts for synchronization errors.	Yes
6. The MCU shall classify the current CO_2 concentration into either low, medium, or alarm, based on user presets from the server.	6A. The MCU will be set to monitor CO_2 concentration. The MCU will print current concentration and current classification (low, medium, or alarm).	Yes

4. Alarm Subsystem

Requirements	Verification	Verification Status (Y/N)
1. The alarm system shall be able to emit Red, Yellow, and Green light from its LED.	1A. The MCU will be programmed to turn the LED to Red, Yellow, and Green in sequence. Each color should be held for 20 seconds. Verify that all colors are visible and solid in appearance.	Yes
2. The alarm system shall be able to emit a tone from its speaker for at least 15 seconds.	2A. The MCU will be programmed to send a tone to the amplifier. 2B. The speaker will be plugged into the amplifier output. Verify that a tone can be heard for at least 15 seconds.	Yes

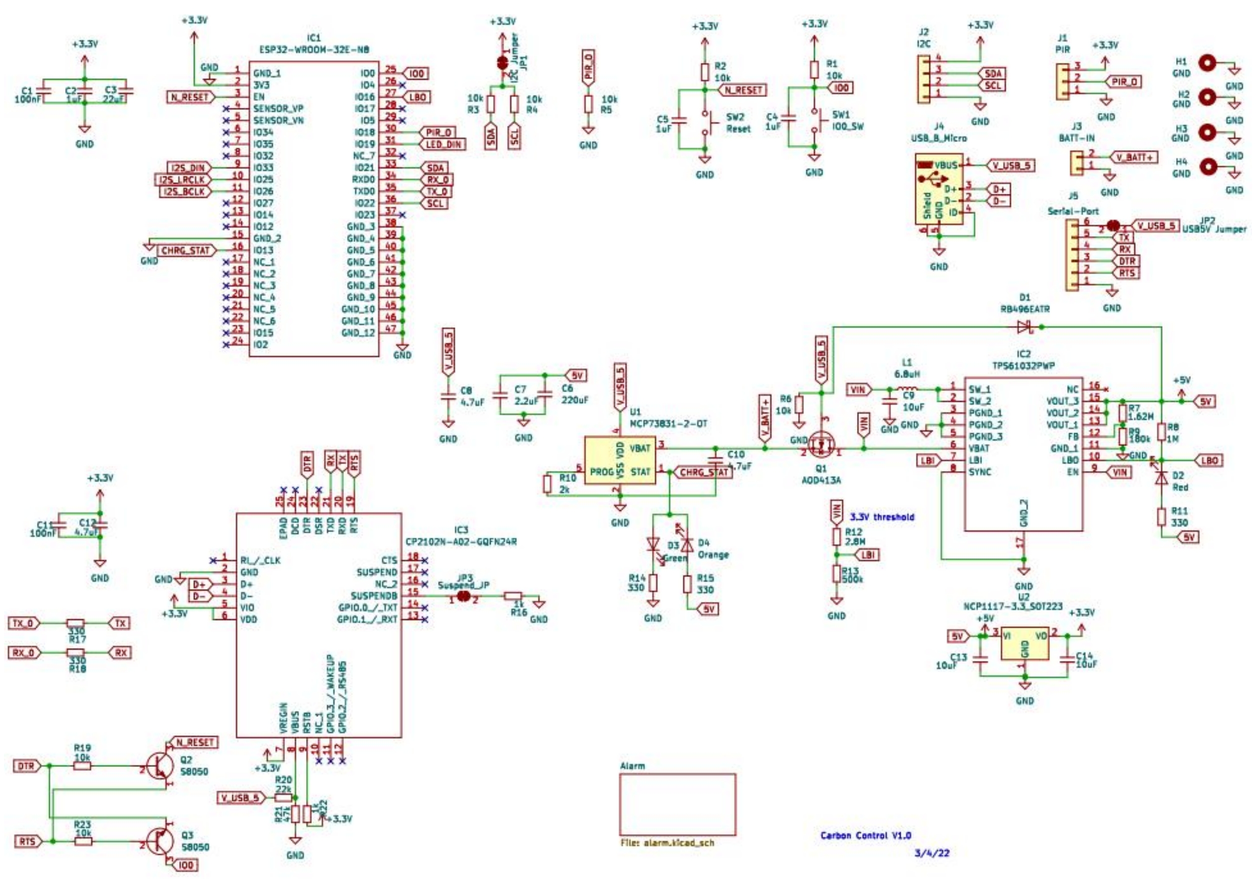
5. Server Subsystem

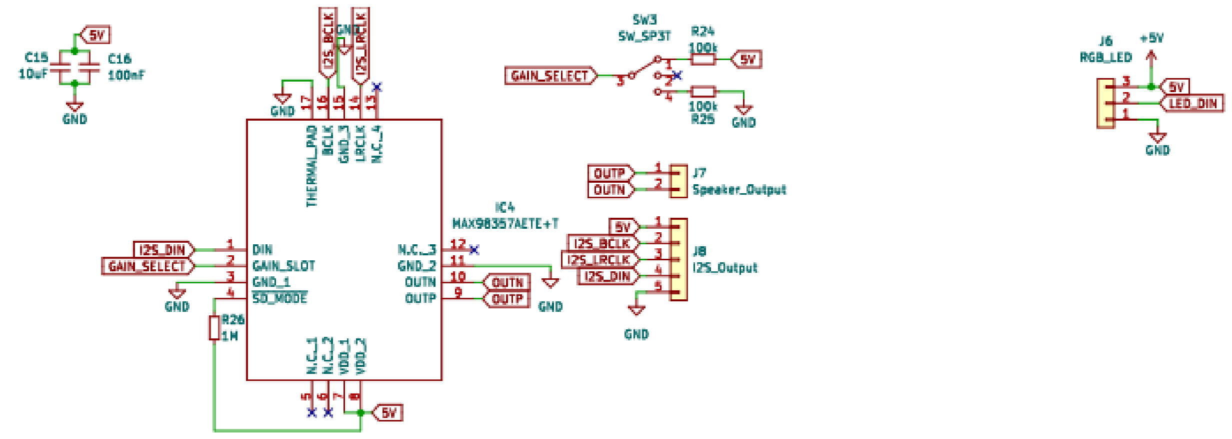
Requirements	Verification	Verification Status (Y/N)
1. The server must maintain an endpoint that the device can POST to that it is in a local alarm condition.	1A. Configure the server to receive POST requests and display the output. 1B. Transmit a local alarm condition from the MCU. 1C. Verify that the server displays the local alarm condition.	Yes
2. The server can make POST requests to the MCU with a data payload.	2A. Configure the MCU to have an endpoint that can receive POST requests. The request contents should be printed. 2B. Program the server to make a POST request with a data payload. 2C. Verify that the request is printed by the MCU to the serial port.	Yes

3. The server can store at least 30 KB of data in its database.	3A. Create a text file (.txt) with a 30 KB size or greater. 3B. Save the contents of the text file to an entry in the database. 3C. Retrieve the entry and verify that the entry is complete.	Yes
4. The server will be able to update the web interface within 60 seconds of receiving updated data.	4A. Configure the server to display the time of the last inbound request and current time difference. 4B. Run the Carbon Control device until it makes an outbound transmission. 4C. Verify that the difference is less than or equal to 60 seconds.	Yes
5. When one device has a local alarm, the server will activate a global alarm in all devices that are tagged to be in the same room.	5A. Configure the server such that two devices are tagged in the same room. 5B. Trigger an alarm in one device with breath. 5C. Ensure that the other device also triggers with only the speaker, but not the LED, active.	Yes
6. The server's web UI can display at least 20 minutes of historical CO_2 data from each device in a room.	6A. Configure two devices to be in the same room on the server. Power up the devices. 6B. Run the devices for 30 minutes to permit at least 20 minutes of data to be uploaded to the server. 6C. Verify that the server can display the recorded data on a plot.	Yes
7. The server will compute the ACH to within 30%.	7A. View the computed ACH on the server's web UI. 7B. The ACH will be manually computed using the historical data graph on the web UI. The ACH = -1 * ln (C1/C0)/(t1- t0). The points used will correspond to the first point of noticeable decay and an arbitrary final point	Yes

	during the decay. 7C. The manually computed ACH will be compared.	

Appendix B Circuit Schematic





Appendix C Carbon Control ESP32 Code

```
/*
  Carbon Control ESP32 Program to integrate CO2 sensor, PIR sensor, Speaker/LED Output
  and communication with webserver.

  4/25/2022

*/

#include <Arduino.h>
#include <SensirionI2CScd4x.h>
#include <Wire.h>
#include <Adafruit_NeoPixel.h>
#include <ArduinoJson.h>
#include <WiFi.h>
#include <HTTPClient.h>

#include "AudioTools.h"
#include "time.h"

// Constants
#define LED_PIN  12
#define LED_COUNT 1
#define PIR_PIN 14

// global variables
```

```
Adafruit_NeoPixel strip(LED_COUNT, LED_PIN, NEO_GRB + NEO_KHZ800);
SensirionI2CScd4x scd4x;
uint32_t red = strip.Color(255, 0, 0);
uint32_t green = strip.Color(0, 255, 0);
uint32_t yellow = strip.Color(255, 150, 0);
uint32_t cur_color = strip.Color(0, 0, 0);
float co2, temp, humid;

//audio consts
uint16_t sample_rate=88200;
uint8_t channels = 2;                                   // The stream will have 2 channels
SineWaveGenerator<int16_t> sineWave(7000);              // subclass of SoundGenerator with max
amplitude of 32000
GeneratedSoundStream<int16_t> sound(sineWave);          // Stream generated from sine wave

I2SStream out;
StreamCopy copier(out, sound );

//PIR globals
int pirState = LOW;
int pirVal = 0;

//WiFi network Information
const char* ssid = "M4";
const char* password = "12345678";
unsigned long lastTime = 0;
unsigned long timerDelay = 5000;

//date-time
const char* ntpServer = "pool.ntp.org";
const long  gmtOffset_sec = 0;
const int   daylightOffset_sec = 0;

bool globalAlarm;

//setup
void setup() {

  pinMode(PIR_PIN, INPUT);    // declare sensor as input
  Serial.begin(115200);
  while (!Serial) {
    delay(100);
  }

  //AudioLogger::instance().begin(Serial, AudioLogger::Info);
  Wire.begin();

  uint16_t error;
  char errorMessage[256];

  scd4x.begin(Wire);

  strip.begin();           // INITIALIZE NeoPixel strip object (REQUIRED)
  strip.show();            // Turn OFF all pixels ASAP
  strip.setBrightness(100); // Set BRIGHTNESS to about 1/5 (max = 255)

  // stop potentially previously started measurement
  error = scd4x.stopPeriodicMeasurement();
  if (error) {
    Serial.print("Failed to execute");
    errorToString(error, errorMessage, 256);
    Serial.println(errorMessage);
  }

  // Start Measurement
  error = scd4x.startPeriodicMeasurement();
  if (error) {
    Serial.print("Failed to execute");
    errorToString(error, errorMessage, 256);
    Serial.println(errorMessage);
  }

  Serial.println("First Measurement");
  cur_color = green;

  //start I2S
  Serial.println("Initializing I2S Bus");
  auto config = out.defaultConfig(TX_MODE);
  config.sample_rate = sample_rate;
  config.channels = channels;
  config.bits_per_sample = 16;
  config.pin_bck = 26;
  config.pin_ws = 25;
  config.pin_data = 33;
```

```
    out.begin(config);

    // Setup sine wave
    sineWave.begin(channels, sample_rate, 440.0);


    //setup WiFi Connectivity
    WiFi.begin(ssid, password);
    while(WiFi.status() != WL_CONNECTED) {
      delay(500);
      Serial.print(".");
    }

    Serial.println("");
    Serial.print("Connected to WiFi network with IP Address: ");
    Serial.println(WiFi.localIP());
    //Serial.println("Timer set to 5 seconds (timerDelay variable), it will take 5 seconds before
publishing the first reading.");

    configTime(gmtOffset_sec, daylightOffset_sec, ntpServer);
  }

  /* Function polls the sensor and if the sensor has new data,
   * it returns a pointer to that data. If the sensor is not ready,
   * or there is an error, it returns a nullptr.
   * @return length 3 float array with co2 value in ppm, temperature in celsius, and humidity in %.
   */


  int getData(float* ret){
    uint16_t error;
    char errorMessage[256];

    uint16_t co2 = 0;
    float temperature = 0.0f;
    float humidity = 0.0f;
    bool isDataReady = false;
          // check if the sensor is ready to be measured
    error = scd4x.getDataReadyFlag(isDataReady);
    // error handling
    if (error) {
      Serial.print("Error trying to execute 1 readMeasurement(): ");
```

```
        errorToString(error, errorMessage, 256);
        Serial.println(errorMessage);
        return -1;
    }
            // is sensor ready to be polled
    if (!isDataReady) {
        ret = NULL;
        return -1;
    }
            // get the data and store in three variables
    // handle appropriate errors
            //Serial.print("is data ready? ");
            //Serial.println(isDataReady);
    error = scd4x.readMeasurement(co2, temperature, humidity);
    Serial.println(co2);
    if (error) {
        Serial.print("Error trying to execute readMeasurement(): ");
        errorToString(error, errorMessage, 256);
        Serial.println(errorMessage);
    } else if (co2 == 0.0) {
        Serial.println("Invalid sample detected, skipping.");
    } else {
        ret[0] = (float) co2;
        ret[1] = (float) temperature;
        ret[2] = (float) humidity;
        return 0;
    }
}

bool process_data(float* data){
  //Serial.print("data is null? :");
  //Serial.println(data == NULL);
  bool sound;
  if(data != NULL){
    co2 = data[0];
    //Serial.println(co2);
    //Serial.println(co2);
    if(co2 < 1000){
      cur_color = green;
      sound = false;
    }
    else if(co2 >= 1000.0 && co2 < 1300.0){
      cur_color = yellow;
      sound = false;
    }
    else if(co2 >= 1300.0 && co2 < 1400){
      cur_color = red;
      sound = false;
    }
    else if(co2 >= 1400.0){
      cur_color = red;
      sound = true;
    }

  }
  setColor(cur_color, 100, 0);
  return sound;
}

void loop() {
  delay(100);
  float data[3];
  int retVal = getData(data);
  //pirSensor
  pirVal = digitalRead(PIR_PIN);
  if (pirVal == HIGH) {            // check if the input is HIGH
    if (pirState == LOW) {
      Serial.println("Motion detected!");
      pirState = HIGH;
    }
  } else {
    //digitalWrite(ledPin, LOW); // turn LED OFF
    if (pirState == HIGH){
      Serial.println("Motion ended!");
      pirState = LOW;
    }
  }

  if (retVal == 0){

    bool localSound = process_data(data);
    //want to now send data to the webserver
    if(WiFi.status() == WL_CONNECTED) {
      HTTPClient http;
```

```
http.begin("http://192.168.178.54:8090/update");
http.addHeader("Content-Type", "application/json");

StaticJsonDocument<200> doc;

//send readings to the sensor
//doc["sensor"] = "co2";
doc["time"] = getTime();
doc["co2"] = data[0];
doc["temp"] = data[1];
doc["humid"] = data[2];
doc["occupancy"] = pirState;
//doc["occupancy"] = false;
doc["sensor_id"] = 1;
doc["room_id"] = "ECEB 1002";

String requestBody;
serializeJson(doc, requestBody);
int httpResponseCode = http.POST(requestBody);

// //check httpResponseCode
// if(httpResponseCode>0){
//   Serial.println(httpResponseCode);
// } else {
//   Serial.println("Error on sending POST");
// }



HTTPClient http2;
http2.begin("http://192.168.178.54:8090/alarm_check");
http2.addHeader("Content-Type", "application/json");
int httpResponseCode2 = http2.GET();
String payload = http2.getString();
StaticJsonDocument<200> doc2;
const char* json {payload.c_str()};
DeserializationError error = deserializeJson(doc2, json);
if (error) {
  Serial.print(F("deserializeJson() failed: "));
  Serial.println(error.f_str());
  return;
}
```

```
      //Serial.println(payload);
      globalAlarm = doc2["alarm"];
      Serial.print("global alarm: ");
      Serial.println(globalAlarm);
      http2.end();
      http.end();  //Free resources

    } else {

      Serial.println("Error in WiFi connection");

    }

    //check to play audio
    if (localSound || globalAlarm) {
      for(int i = 0; i < 550; i++){
        copier.copy();
      }
    }
  }
}

void setColor(uint32_t color, int wait, int led) {
 // For each pixel in strip...
  strip.setPixelColor(led, color);         //  Set pixel's color (in RAM)
  strip.show();                          //  Update strip to match
  delay(wait);                           //  Pause for a moment
}

//function to generate the current date/time
unsigned long getTime() {
  time_t now;
  struct tm timeinfo;
  if (!getLocalTime(&timeinfo)) {
    //Serial.println("Failed to obtain time");
    return(0);
  }
  time(&now);
  //Serial.println(now);
  return now;
}
```

Appendix D Carbon Control Server Code

```
from distutils.util import strtobool
import requests
import base64
import io
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from matplotlib.dates import DateFormatter
from matplotlib.figure import Figure
from flask import Flask, render_template, url_for, request, redirect
from flask_sqlalchemy import SQLAlchemy
from datetime import datetime
from werkzeug.utils import redirect
import json
import matplotlib
import numpy as np
matplotlib.use('Agg')


def string_to_bool(string):
    return bool(strtobool(str(string)))


# launch app
app = Flask(__name__)
# associate database
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test.db'
# create database
db = SQLAlchemy(app)

global global_alarm
global_alarm = False


class data(db.Model):
    '''Database Schema'''
    # ID is integer, serves as primary key, unique id per reading
    id = db.Column(db.Integer, primary_key=True)
    # CO2 reading is an integer representing CO2 value
    co2 = db.Column(db.Integer, unique=False, nullable=False)
    # Occupancy (integer for now, maybe bool)
    occupancy = db.Column(db.Boolean, unique=False, nullable=False)
    # Time as a datetime python onject
    time = db.Column(db.DateTime, nullable=False)
    # Sensor ID
    sensor_id = db.Column(db.Integer, unique=False, nullable=False)
    # room_id
    room_id = db.Column(db.Integer, unique=False, nullable=False)
    # temp and humidity
    temp = db.Column(db.Float, unique=False, nullable=False)
    humid = db.Column(db.Float, unique=False, nullable=False)

    def __repr__(self):
        return 'CO2 was %d with %d occupancy' % (self.co2, self.occupancy)


@app.route('/update', methods=['POST'])
def update():
    if request.method == 'POST':
        # parse data
        jsonData = request.data.decode('utf-8')
        jsonData = json.loads(jsonData)

        # extract values from JSON
        co2 = jsonData["co2"]
        occupancy = string_to_bool(jsonData["occupancy"])
        time = jsonData["time"]
        sensor_id = jsonData["sensor_id"]
        room_id = jsonData["room_id"]
        temp = round(float(jsonData["temp"]), 2)
        humid = round(float(jsonData["humid"]), 2)

        # careful with the format string
        dt_object = datetime.fromtimestamp(time)
        # make new entry
        new_data = data(co2=co2, occupancy=occupancy, time=dt_object,
                        sensor_id=sensor_id, room_id=room_id, temp=temp, humid=humid)

        try:
            # try to add to database
            db.session.add(new_data)
            db.session.commit()
        return index()
    except BaseException as e:
        print(e)
        return "error!"


@app.route('/sendAlarm', methods=['POST', 'GET'])
def trigger_alarm():
    if request.method == 'POST':
        global global_alarm
        global_alarm = not global_alarm
        print(global_alarm)
    return redirect('/')


@app.route('/alarm_check', methods=['GET'])
def alarm_check():
    return {"alarm": global_alarm}


@app.route('/', methods=['GET', 'POST'])
def index():
    graph_data = data.query.order_by(data.time).with_entities(
        data.time, data.co2).all()
    graph_data = list(zip(*graph_data))
    dates = graph_data[0]
    co2 = graph_data[1]
    img = io.BytesIO()
    fig = Figure(figsize=(round(16 * 0.7), round(9 * 0.7)))
    ax = fig.subplots()
    h_fmt = DateFormatter("%H:%M:%S")
    ax.xaxis.set_major_formatter(h_fmt)
    ax.plot(dates, co2)
    ax.set_title("$CO_2$ Concentrations over time")
    ax.set_xlabel("Time")
    ax.set_ylabel("$CO_2$ Concentrations (PPM)")
    fig.tight_layout()
    fig.savefig(img, format="png")
    im = base64.b64encode(img.getbuffer()).decode("ascii")
    img_string = im

    ach_img = computeACH()
    tasks = data.query.order_by(data.time).all()
    return render_template("index.html", tasks=tasks, img_string=img_string, ach_im=ach_img)


# @app.route('/ach', methods=['GET'])
def computeACH():
    graph_data = data.query.order_by(data.time).with_entities(
        data.time, data.co2, data.occupancy).all()
    graph_data = list(zip(*graph_data))
    dates = np.array(graph_data[0])
    co2 = np.array(graph_data[1])
    occu = np.array(graph_data[2])

    t = 5
    N = len(dates)
    ach = [0.0]
    occ = [0.0]
    out_str = ""
    # ach calculation for all windows of length t
    for i in range(2, N):
        if i - t < 0:
            c_start = co2[0]
            c_end = co2[i]
            d_start = dates[0]
            d_end = dates[i]
            avg_occ = np.mean(occu[0: i])
        else:
            c_start = co2[i - t]
            c_end = co2[i]
            d_start = dates[i - t]
            d_end = dates[i]
            avg_occ = np.mean(occu[i - t: i])

        delta_t = d_end - d_start
        delta_t_seconds = delta_t.total_seconds()
        delta_t_hours = delta_t_seconds / 3600.0

        co2Ratio = c_end / c_start

        if co2Ratio >= 0.8 or avg_occ > 0.9:
            cur_ach = ach[-1]
        else:
            # ach = (-1 *  ln(c1/c0) / t1 - t0
            cur_ach = -1 * np.log(c_end / c_start) / delta_t_hours
            # out_str = out_str + "<p>" + str(cur_ach) + "</p>"

        ach.append(cur_ach)
        occ.append(avg_occ)

    img = io.BytesIO()
    fig = Figure(figsize=(round(16 * 0.7), round(9 * 0.7)))
    ax = fig.subplots()
    h_fmt = DateFormatter("%H:%M:%S")
    ax.xaxis.set_major_formatter(h_fmt)
    ax.plot(dates[1:], ach)
    ax.set_title("ACH over time")
    ax.set_xlabel("Time")
    ax.set_ylabel("ACH (1/H)")
    fig.tight_layout()
    fig.savefig(img, format="png")
    im = base64.b64encode(img.getbuffer()).decode("ascii")
    img_string = im
    return img_string
    # tasks = data.query.order_by(data.time).all()
    # return render_template("index.html", tasks=tasks, img_string=img_string)


if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8090, debug=True)

    # print("recieved data from ESP32", jsonData)

    # # # get the JSON
    # print(json)
    # tasks = data.query.order_by(data.time).all()
    # return render_template("index.html", tasks=tasks)
```