# Sensor System for Self-Driven In-Home Climate Control

Team 20

Blake Wiker, Jadon Roth, James Beans, Yousif Albaker

Table of Contents

# 1. Overview

## 1.1. Executive Summary

This IOT sensor system will be designed to help homeowners make optimal use of their windows. As the cost of electricity and natural gas increases, the effective use of windows to regulate interior temperature can save users money. We will use hardware and software solutions to build two monitoring modules for indoor and outdoor use. These sensors will take weather data, such as temperature and humidity, and wirelessly transmit data to a mobile application. The sensors will be battery powered, IPX4 water resistant, and fit within a 3x3x1 inch enclosure. In our application, an algorithm will use temperature and humidity data as well as local weather to determine whether windows should be opened or closed before sending the user a push notification. The application will also allow users to pair/unpair devices and visualize sensor data.The project is currently in the building stage. We are now working towards putting together our final design. Our next step will be to complete our final pcb, enclose the system, and test to make sure pairing functionality, data transfer and notifications work as expected with our application.

## 1.2. Team Contacts and Protocols

**Table 1.1. Group Member Information and Roles**

| Name | Phone | Email | Role |
|---|---|---|---|
| Jadon Roth | 541-570-8932 | dirksjad@oregonstate.edu | Microcontroller and Sensor Hardware |
| Blake Wiker | 360-912-3488 | wikerb@oregonstate.edu | Power electronics and enclosure |
| Yousif Albaker | 541-908-6126 | albakery@oregonstate.edu | Microcontroller and system backend programming |
| James Beans | 541-340-5453 | beansja@oregonstate.edu | Application programming |

**Table 1.2. Group Protocols**

| Topic | Protocol | Standard |
|---|---|---|
| Documentation Structure | All information will be stored on the Google Shared Drive. | All information will be organized and accessible via our central cloud service. |
| Information Distribution | There will be an open information flow. | Whenever documentation or designs of any kind are generated, they will be posted to the central information hub. |
| Communication | We will primarily communicate via Discord, email, and phone. | Priority will take place in that order. |
| On-Time Deliverables | We will be having weekly meetings as a team at the beginning of the week where we will discuss timelines and tasks for the week. | Being that we have meetings at the beginning of the week, it is the perfect time to discuss what will be done during the week! We will generate timelines and to-dos for the following six days. |
| Task Management | We will use a spreadsheet to create a pseudo-Gantt chart. | We will generate an initial project spreadsheet that includes tasks and time to complete. This spreadsheet will be a living document, to be edited as more tasks arrive. |
| Work Quality | Work will be completed with complete effort and display sufficient quality. | If work quality is not up to group standards, members will be asked to update previous work. |
| Missing Meetings | Group members will be expected to come to every scheduled meeting unless notified in advance. | Members will let the group know if they will miss a meeting at least 24 hours in advance. |
| Deadlines | Deadlines will be created during group meetings and are expected to be met unless the group is notified. | Deadlines are noted in the timeline, members must notify the group if deadline will not be met multiple days in advance. |

**Mentor Collaboration**

We will be meeting weekly with our mentors Faiiq Waqar, Lyubo Gankov and Kiernan Canavan. We will be meeting via zoom. Our communication with them will involve these meetings and our group discord server. They can collaborate with us via this server, as well as through our Shared Google Drive of which they are also a part of. All documentation and updates will be uploaded to our Google Drive.

### 1.3. Gap Analysis

The reason for our project existence is that there will be no need to pay money for A/C when there is a climate control sensor that provides the customer with the temperature in the house and outside and that’s when the customer will have an option to either cool the inside of the house by opening a window or warm it up incase outside temperature was warmer than inside.

We assume connecting the hardware would be an important part of the project and how smooth it will run as well as the coding and software included and has to work simultaneously and in sync in order to provide the best results possible on the GUI.The project will be usable by everyone in the community and it should be affordable rather than expensive so all people can use it by downloading an application from their mobile application store and creating an account on the platform and connecting it to the probe and by that would get them up and running.

There are a few products already out there that are fairly similar to what we’re building here. The first and most glaring example is your standard smart sensor/system. In essence, they take in information from a series of nodes and talk with a central brain to make things happen. This is almost exactly what we’re doing with the exception of actually making the things happen. We’ll be able to draw ideas from the networking happening between these devices. Another product that could lend ideas are rechargeable battery-powered electronics. Here we can draw on more of the hardware side of things to give us a better idea of what a battery charging and power regulating circuit looks like. Finally, things like Garmin’s smart watch applet may help to give us a good example of a functional user interface built around hardware input.

## 1.4. Timeline/Proposed Timeline

### Table 1.3. Timeline

| Task | Name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Documentation | Group | Section 1 | | | Section 2 | | | Section 1+2 | | | | | | | | | | | | | | | | | | | | | | | |
| Partner Meetings | Group | Update 1 | | | | Update 2 | | | | Update 3 | | | | | | | | | | | | | | | | | | | | | |
| Design Impact Assessment | Group | Initial Assessment | | | | | | Final Assessment | | | | | | | | | | | | | | | | | | | | | | | |
| Communication Evaluation | Group | Sign Up for Meeting | | | | Meeting (20 min) | | | | | | | | | | | | | | | | | | | | | | | | | |
| Hardware Research | Jadon + James | | | Microcotrollers, sensors, power supply, circuit design | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Software Research | Yousif + Blake | | | Wireless connection, app GUI, data algorithm | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Ordering Components | Jadon | | | | | | | | Order Preliminary components | | | | | | | Order PCBS/Components | | | | | | | | | | | | | | | |
| Block Diagram and Interfaces | Group | | | | | | | | Block allocation and submission | | Finalize interface for block check offs | | | | | | | | | | | | | | | | | | | | |
| Microcontroller Circuit Design | Jadon | | | | | | | | | Begin Circuit Design | | Draft Circuit Design | | | | Finish Circuit Design | | | | Final Tweaks if necessary | | Redesign, Rebuild and Retest | | | | | | | | | |
| Power Electronics Circuit Design | James | | | | | | | | | Begin Circuit Design | | Draft Circuit Design | | | | Finish Circuit Design | | | | Final Tweaks if necessary | | Redesign, Rebuild and Retest | | | | | | | | | |
| Microcontroller Block Prototyping | Jadon | | | | | | | | | | | Initial Prototyping | | Finish Prototype | | | | | | | | | | | | | | | | | |
| Power Electronics Block Prototyping | James | | | | | | | | | | | Initial Prototyping | | Finish Prototype | | | | | | | | | | | | | | | | | |
| Microcontroller Building and Testing | Jadon | | | | | | | | | | | | | | | | | PCB Assembly | | PCB Testing | | | | | | | | | | | |
| Power Electronics Building and Testing | James | | | | | | | | | | | | | | | | | PCB Assembly | | PCB Testing | | | | | | | | | | | |

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Node connection, computation, data storage development | Blake | | | | | | | | | | | Initial prototyping | | Cloud storage and Debugging | | Block level testing | | | | | | | | | | | | | | | |
| GUI and notification development | Yousif | | | | | | | | | | | Initial prototyping | | Data visuals and debugging | | Block level testing | | | | | | | | | | | | | | | |
| Software system level testing | Blake + Yousif | | | | | | | | | | | | | | | Combine software experiences, implement into app | | | | | | | | | | | | | | | |
| Project System Level testing and assembly | Group | | | | | | | | | | | | | | | | | | Combine hardware and software experiences | | | | | | | | | | | | |
| Final project fixes | Group | | | | | | | | | | | | | | | | | | | | | Make sure project works | | | | | | | | | |
| Documentation and presentation preparation | Group | | | | | | | | | | | | | | | | | | | | | Organize and prepare documents and presentation | | | | | | | | | |
| Finalize project poster and documentation | Group | | | | | | | | | | | | | | | | | | | | | | | | | | Make poster and finalize presentation/documents | | | | |
| Knowledge Transfer | Group | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Transfer knowledge | |

### 1.5. References and File Links

#### 1.5.1. References (IEEE)

[1] F. Waqar, "EECS Project Portal," *Sensor System for Self-Driven In-Home Climate Control (ECE)*. [Online]. Available: https://eecs.oregonstate.edu/capstone/submission/pages/viewSingleProject.php?id=9vBa7B1brA8Osgxc. [Accessed: 14-Oct-2022].

#### 1.5.2. File Links

Block Diagram (Draft): Here

### 1.6. Revision Table

| Date Revised | Who? | Reason Revised/Revisions Made |
|---|---|---|
| 03/08/23 | Blake | Updated Summary |
| 01/20/23 | Blake | Updated Summary |
| 11/15/22 | Blake | Updated executive summary and added protocols |
| 11/15/22 | Jadon | Added extra examples to GAP assessment |
| 11/8/22 | Group | Updated timeline |
| 11/3/22 | Blake | Added Title page, table of contents, group info table, timeline |
| 10/14/22 | Jadon | Added File and Link |
| 10/11/22 | Group | Initial Revision |

# 2. Impacts and Risk

### 2.1. Design Impact Statement

This section will discuss the impact of our product and design choices in four major categories: Public Health, Safety, and Welfare impacts, Cultural and Social impacts, Environmental Impacts, and Economic Impacts. Within these categories, we will address the harm of lithium-ion batteries in our design, the negative impacts of a potential malfunction, and how the price can be a detriment to future users. We will also talk about potential solutions to these issues. Some, like calculation malfunctions, we can potentially fix, while others, like battery use, we can mitigate the issues but never fully remove due to the constraints of our design (small, lightweight, mobile sensor).

In terms of public health impacts, our sensors will use outdoor air quality as one of the numbers to determine if windows should be opened or closed. If our sensor malfunctions, for example not receiving air quality data from the internet or forgetting to tell users to close their windows when the outdoor air quality worsens, poor air could potentially enter a users home. This can be dangerous for people inside the home, where a concentration of polluted air can lead to shortness of breath, coughing, chest pain, nausea, fatigue and asthma attacks for those at risk [1]. A possible solution to this problem could be to remind people to check air conditions if the sensors are not receiving air quality data, or tell users to close their windows when calculations cannot be made.

When looking at cultural and social impacts, accessibility of products plays a key role. Our sensors need to be accessible to our users because homeowners come from all different backgrounds and experiences. People who are disabled, or older people, are a few groups we have to consider when designing our sensors. There are many smart home products that are overly confusing to work and set up, especially for people that are not familiar with technology in these groups. In order to create a more accessible product, we must work to cut costs and to make sure it falls within the price range for everyday users [2]. We also must make sure that the UI and information displayed is useful and easy to understand, especially for people that are visually impaired [2].

While our sensor system will save users money and lessen the environmental impact of air conditioning, the batteries we use can offset those effects. The batteries we use are lithium-ion and we have to take into account where that lithium comes from and how that affects the environment. The process of mining lithium has extreme side effects on the environment due to the difficult nature and small yield received after refinement. One way that lithium is mined and gathered is the use of evaporation ion pools of water which consumes over 2.2 million liters of water for just 1 ton of lithium [3]. For our project as we do use lithium-ion batteries, we do contribute to that environmental impact. However, at most, our battery will be 1 amp hour in size which over a small quantity of production of our project, is small compared to other sources of energy. We can minimize our energy usage by checking data sheets and finding modules with low energy consumption.

Finally, economic impacts cannot be ignored. Our sensors use many chips and modules to allow our team to fulfill all our requirements. Due to the global chip shortage, these components are rapidly increasing in price and are limited in availability [4]. With the supply not matching the demand for these parts, our sensors could potentially be 2 or 3 times more expensive than just a year ago [4]. A sensor that costs between $50 and $100 to manufacture (waterproof temperature and humidity sensors being the most expensive) could potentially offset any cost savings from a user's heat or A/C bill.

## 2.2. Risk

**Table 2.1. Risk Assessment and Action Plans**

| Risk ID | Risk Description | Risk category | Risk probability | Risk impact | Performance indicator | Action Plan |
|---|---|---|---|---|---|---|
| R1 | Incompatible interface | Technical | H | H | Sensor Data is not displayed in GUI application | Research compatibility between GUI and storage system, check code, break down into code sections and debug |
| R2 | Vendor delay | Timeline | M | M | Behind Schedule | Find local replacements or another vendor with faster shipping |
| R3 | Parts beyond budget | Cost | H | H | Parts prices are increasing due to the demand and low supply globally. | Find another vendor that provides similar parts with cheaper prices. Look into datasheets for cheaper parts to find similar specifications. |
| R4 | LiPo Battery catches fire | Safety | L | H | Fire | Locate fire extinguisher and call 911, mitigate fire spread |
| R5 | Team members need to take | Organizational | M | M | Team members sends notice of absence | Gather time sensitive material team members |

| | | | | | | |
|---|---|---|---|---|---|---|
| | time for personal matters | | | | | were working on, list the work that had to be completed, redistribute work among the team. |
| R6 | Code Errors | Technical | H | M | Gui bugs and errors, showing incorrect data | Use debug tools, find problem, update code in sections, debug the problem |
| R7 | Sensor error | Safety | L | M | Probe malfunctions and sends notification in the wrong time | Test the sensors and probe multiple times in order to get accurate data outputs, if issues persist, order new sensors. |
| R8 | Enclosure Failure | Technical | L | H | Moisture touches our electronics | Test before putting electronics in the elements, find failpoint, redesign enclosure |
| R9 | Electronics Failure | Organizational | L | H | Electronic components fail or malfunction | Order an extra part of the component that might malfunction easily. |

## 2.3. References and File Links

### 2.3.1. References (IEEE)

[1] "Air Quality and Health," *Minnesota Pollution Control Agency*. [Online]. Available: https://www.pca.state.mn.us/air-water-land-climate/air-quality-and-health.

[2] M. Cannistra, "Fully accessible guide to smart home tech for the disabled and elderly," *ZDNET*, 07-Sep-2021. [Online]. Available: https://www.zdnet.com/home-and-office/smart-home/fully-accessible-guide-to-smart-home-tech-for-the-disabled-and-elderly/.

[3] M. Campbell, "South America's 'lithium fields' reveal the dark side of Electric Cars," *euronews*, 01-November-2022. [Online]. Available: https://www.euronews.com/green/2022/02/01/south-america-s-lithium-fields-reveal-the-dark-side-of-our-electric-future.

[4] G. Scott, "Why the chips are down: Explaining the global chip shortage: Jabil," *Jabil.com*. [Online]. Available: https://www.jabil.com/blog/global-chip-shortages.html. [Accessed: 01-Nov-2022].

### 2.3.2. File Links

Risk workshop in Class: 20221021_114106.jpg

## 2.4. Revision Table

| Date Revised | Who? | Reason Revised/Revisions Made |
|---|---|---|
| 04/25/23 | Blake | Added impact statement |
| 11/15/22 | Blake | Updated action plans |
| 11/4/22 | Yousif and Blake | Initial table creation |

# 3. Top-Level Architecture

## 3.1. Block Diagram

This section details the black box diagram and the system block diagram for the sensor system. Figure 3.1 shows the black box diagram with four system input and two system outputs. Figure 3.2 is the main block diagram with block and interface names.

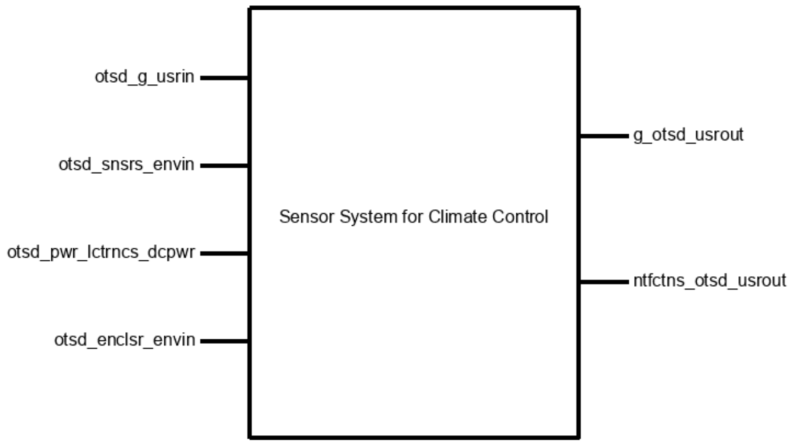


**Figure 3.1: System Black Box Diagram**

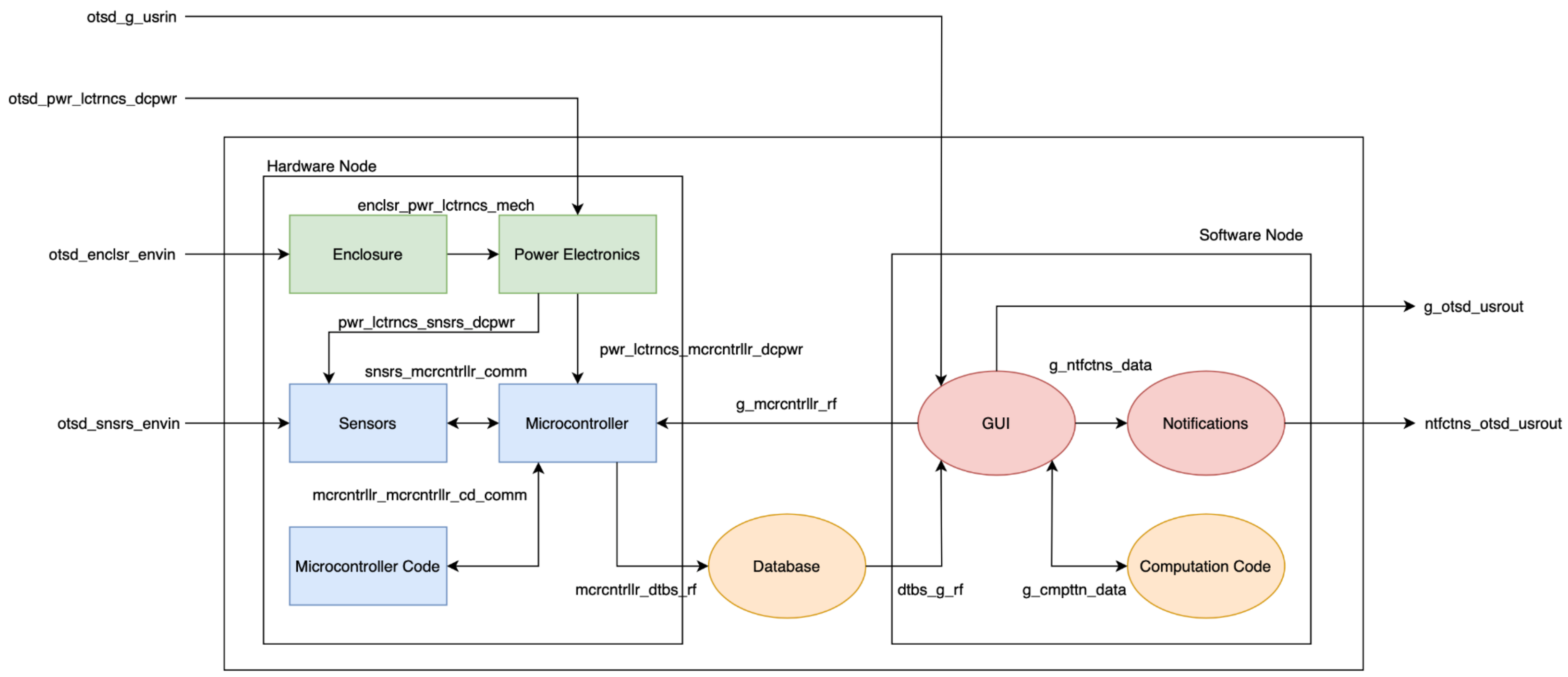


**Figure 3.2: System Block Diagram**

## 3.2. Block Descriptions

This section contains a table with the block descriptions for each block in our system. Block descriptions are a quick overview of what the block contains and how it functions within the whole system.

**Table 3.1: Block Descriptions**

| Name | Description |
|---|---|
| GUI<br>Champion: Yousif Albaker | The GUI block is the main application code block for the sensor system and allows our user add sensors and visualize sensor data. The GUI block uses android studio to build the climate control application in the Java programming language. The GUI consists of three main pages, also known as activities, in android studio. The main activity consist of a text view, which contains text relating to window status, a recycler view, which contains each sensor node connected to the application, and a button to add new sensors. Each sensor node on the main activity also displays the sensor's name and updated temperature and humidity data. This data is updated each time the activity is opened by querying the cloud database in a background data thread and putting the most current values into a local database called android room. Each element on the main activity can also trigger new windows for the user to navigate. When the user taps on the window status text view, a popup window is opened which allows the user to input their desired interior temperature for their house. Tapping on a sensor node opens the node info activity where 24 hr graphs for temperature and humidity are displayed for the specific node. Clicking the add node button opens the add node activity where sensors can be provisioned to the local wifi network and paired to the application. Information like wifi credentials and sensor name are sent to the ESP32 microcontroller from over bluetooth from this activity. The GUI gathers all of its input sensor data and information from the cloud database. Querying this database in the background allows the GUI to be continually updated and gather all the necessary data (like time, name and sensor data) for graphs and calculations. Two major blocks are attached to the GUI, computation block and the notification block. The computation block has two way communication with the GUI, where sensor and desired temperature data is send to the block and the calculation results and received. The GUI also outputs notification information to the notification block to make sure the user receives the proper notification when the windows need to be opened. |

| | |
|---|---|
| Notifications Champion: Yousif Albaker | The notification block allows the user to be notified when their windows need to be opened or closed. The notification block will take in String and integer data values and output push notification to the user using Android system notification. The input integer will correspond to the type notification sent and the string will be the content of the notification. For example, when the block is passed notification integer 1, the open window notification is built using the string "Open Window". The notification is then sent to the notification manager where the application notification channel displays the message for the use. There will be three output notification states, no change, open window, and close window. When the user taps on a notification, they are brought to the application home page. |
| Power Electronics Champion: James Beans | This block consists of the design of a battery charging circuit and the power managing design for the microcontroller and the power for the charging circuit. This would mean an output of 5V for a microcontroller, and output of 3.3V for a chip, a charging circuit for a 3.7V battery, and the ability to receive 5V charging power. |
| Enclosure Champion: James Beans | This block is for the enclosure for the device. The enclosure will house all the components and will be IPX4 weatherproof. It will have to be within a 3" x 3" x 1" size constraint. This enclosure will be a custom 3D printed enclosure in PLA material. |
| Database Champion: Blake Wiker | The database block allows our system to collect, store, and display sensor data over a longer time period. Temperature and humidity data from our sensor comes into the block, where it is packaged by the microcontroller using relevant tags and identifying information, such as username and node name (indoor/outdoor). This packaged data is then sent to our time series cloud database, Influx DB, over WIFI where data is stored with a timestamp the moment it arrives. I chose Influx DB due to its wide range of applications and code examples for both Arduino (ESP32) and Java (Android application) as well as automatic timestamp collection component, which is important for our project. In the database, different accounts (usernames) will be tagged within a single bucket of data and the sensor name will be tagged underneath the user for further organization. Once the data is stored, our Android application will query the database using these tags to collect temperature and humidity data as well as the timestamp from specific sensors. This data will be used in our GUI to create time graphs for all the sensors attached to a user. One of our project partner requirements was the ability to visualize sensor data in our application, and the database block allow the creation of this time-based graph with time on the x-axis and temp/hmd data on the y-axis. We will also use the temperature and humidity values in the computation block to determine when the windows need to be opened or closed. This was another major requirement of the system (window notifications) and the database block helps accomplish this by gathering and sending critical data for the calculations. Finally, one of our system requirements is the ability to show current temperature data, and the database block is needed to update the user interface with the most updated and last time stamped data point. |

| | |
|---|---|
| Computation Champion: Blake Wiker | The computation block will take in data from the indoor and outdoor temperature sensors to determine if windows need to be opened or closed. The user will be able to set a desired temperature using the application. If the indoor temperature rises above the desired temp and the outdoor temperature is below the indoor temp, the block will tell the GUI that windows need to be opened. Humidity will also be used in the computation block by integrating into a dew point calculation. This will be used to determine if the outdoor air is too moisture heavy compared to the indoor moisture. If the outdoor dew point is close to the indoor dew point, the windows can be opened if the temperatures are correct. The calculation block also determines if a notification needs to be sent using the previous window state. If the window state changes, the block will output the corresponding notification integer which is stored and then used by the notification block. The window status string is based on the window calculation and is used within the UI to clearly indicate the window status to the user. |
| Microcontroller Champion: Jadon Roth | This block is the central node for the hardware section on this system. It takes in inputs from the sensors and outputs to the communication block based off the code running on the microcontroller. This device is powered by the power electronics. THis block will consist of a homebrewed PCB, including a daughtered ESP. |
| Microcontroller Code Champion: Jadon Roth | The microcontroller code block lives on the ESP32 microcontroller. The code handles bluetooth and wifi connections, gets and stores temperature, humidity and time data, and write datapoints to the cloud database. Wifi information is collected using a bluetooth provisioning library and once the microcontroller is connected to the local network, time data is synced to the device. In our loop, the temperature and humidity sensor data as well as current time data is collected and stored into a FIFO queue every minute. Every half minute, the loop attempts to upload data to the database. If the queue is empty, the write is not executed. |
| Sensors Champion: Jadon Roth | The sensor block takes in an external data stream from the outside world and outputs this data to the microcontroller. These sensors will communicate with the microcontroller via some protocol, most likely I2C. They will receive power from the battery via the microcontroller. |

## 3.3. Interface Definitions

This section contains a table with all the interfaces in our design. Each interface has a name and properties contained in the interface with the type of property listed before the explanation.

**Table 3.2: Interface Definitions**

| Name | Properties |
|---|---|
| otsd_g_usrin | • **Other:** wifi information (struct) ex. username: mywifi, pw: 1234<br>• **Other:** desired temp (float) ex. 71<br>• **Other:** sensor name (string) ex. Indoor |
| otsd_snsrs_envin | • **Humidity:** Space Dependent<br>• **Temperature (Absolute):** Space Dependent |
| otsd_pwr_lctrncs_dcpwr | • **Inominal:** 150mA<br>• **Ipeak:** 500mA<br>• **Other:** 20W<br>• **Vnominal:** 5V |
| otsd_enclsr_envin | • **Other:** Size: 3" x 3" x 1"<br>• **Water:** IPX4 (Light splash and light spray) |

| | |
|---|---|
| g_otsd_usrout | • **Other:** Sensor Data (Graph) ex. 24 hr line graphs for temp and hmd<br>• **Other:** Window Status (Text Box) ex. Closed<br>• **Other:** Sensor Data (Node) ex. Indoor, temp: 72, hmd: 45.6 |
| g_ntfctns_data | • **Messages:** closed notification (string) ex. Close Window<br>• **Messages:** notification type (int) ex. 0, 1, 2<br>• **Messages:** open notification (string) ex. Open Window |
| g_cmpttn_data | • **Messages:** outdoor hmd (double) ex. 75.8<br>• **Messages:** indoor temp (double) ex. 75.2<br>• **Messages:** desired temp (float) ex. 71<br>• **Messages:** outdoor temp (double) ex. 68.9<br>• **Messages:** indoor hmd (double) ex. 57.3 |
| g_mcrcntrllr_rf | • **Messages:** wifi information (struct) ex. username: mywifi, pw: 1234<br>• **Messages:** sensor name (string) ex. Indoor<br>• **Messages:** reset (bool) ex. false |
| ntfctns_otsd_usrout | • **Other:** closed notification (android system notification)<br>• **Other:** no change (android system notification)<br>• **Other:** open notification (android system notification) |

| | |
|---|---|
| snsrs_mcrcntrllr_comm | • **Messages:** Humidity<br>• **Messages:** Temperature<br>• **Protocol:** I2C |
| pwr_lctrncs_snsrs_dcpwr | • **Inominal:** 3mA<br>• **Ipeak:** 5A<br>• **Vmax:** 5V<br>• **Vmin:** 3V<br>• **Vnominal:** 3.3V |
| pwr_lctrncs_mcrcntrllr_dcpwr | • **Inominal:** 30mA<br>• **Ipeak:** 1.5A<br>• **Vmax:** 5.1V<br>• **Vmin:** 4.6V<br>• **Vnominal:** 5V |
| enclsr_pwr_lctrncs_mech | • **Fasteners:** 3mm Mounting Holes (at least 2)<br>• **Other:** USB C |
| dtbs_g_rf | • **Messages:** tag (string) ex. Indoor<br>• **Messages:** humidity (double) ex. 80.5<br>• **Messages:** time stamp ex. 2023-1- 12T20:36:45<br>• **Messages:** temperature (double) ex. 72.1 |
| cmpttn_g_data | • **Messages:** notification type (int) ex. 0 (0 is none, 1 is open window, 2 is close window)<br>• **Messages:** window status (boolean) ex. false (true is open, false is closed) |

| | |
|---|---|
| | • **Messages:** window string (String) ex. "Closed" |
| mcrcntrllr_dtbs_rf | • **Messages:** humidity (double) ex 80.5<br>• **Messages:** temperature (double) ex. 72.1<br>• **Messages:** tag (string) ex. Indoor |
| mcrcntrllr_mcrcntrllr_cd_comm | • **Messages:** Time<br>• **Messages:** Humidity<br>• **Messages:** Wifi information ex. username: mywifi, pw: 1234<br>• **Messages:** Temperature<br>• **Messages:** Node name ex. Indoor |
| mcrcntrllr_cd_mcrcntrllr_comm | • **Messages:** Time<br>• **Messages:** Temperature<br>• **Messages:** Node name<br>• **Messages:** Humidity |

## 3.4. References and File Links

### 3.4.1. References (IEEE)

### 3.4.2. File Links

## 3.5. Revision Table

| **Date Revised** | **Who?** | **Reason Revised/Revisions Made** |
|---|---|---|
| | | |
| 03/10/23 | Blake | Initial Creation |

# 4. Block Validations

## 4.1. GUI

### 4.1.1. Description

The GUI block is the main application code block for the sensor system and allows our user add sensors and visualize sensor data. The GUI block uses android studio to build the climate control application in the Java programming language. The GUI consists of three main pages, also known as activities, in android studio. The main activity consists of a text view, which contains text relating to window status, a recycler view, which contains each sensor node connected to the application, and a button to add new sensors. Each sensor node on the main activity also displays the sensor's name and updated temperature and humidity data. This data is updated each time the activity is opened by querying the cloud database in a background data thread and putting the most current values into a local database called android room.

Each element on the main activity can also trigger new windows for the user to navigate. When the user taps on the window status text view, a popup window is opened which allows the user to input their desired interior temperature for their house. Tapping on a sensor node opens the node info activity where 24 hr graphs for temperature and humidity are displayed for the specific node. Clicking the add node button opens the add node activity where sensors can be provisioned to the local wifi network and paired to the application. Information like wifi credentials and sensor name are sent to the ESP32 microcontroller from over bluetooth from this activity.

The GUI gathers all of its input sensor data and information from the cloud database. Querying this database in the background allows the GUI to be continually updated and gather all the necessary data (like time, name and sensor data) for graphs and calculations. Two major blocks are attached to the GUI, computation block and the notification block. The computation block has two way communication with the GUI, where sensor and desired temperature data is sent to the block and the calculation results and received. The GUI also outputs notification information to the notification block to make sure the user receives the proper notification when the windows need to be opened.

### 4.1.2. Design

This section includes the black box diagram of the GUI block as well as a flow chart describing the code progression within the block. Figure 4.1, the black box diagram, shows that the block has three inputs (otsd_g_usrin, dtbs_g_rf, cmpttn_g_data), and four outputs (g_otsd_usrout, g_ntfctns_data, g_cmpttn_data, g_mcrcntrllr_rf). Input consists of user input to the GUI, data input from the cloud database, and computation data from the computation data block. The GUI block outputs to the user in the form of graphical sensor data, gives data to the notification block to send notifications, gives data to the computation block from the database to make calculations, and sends node data from the user to the microcontroller.

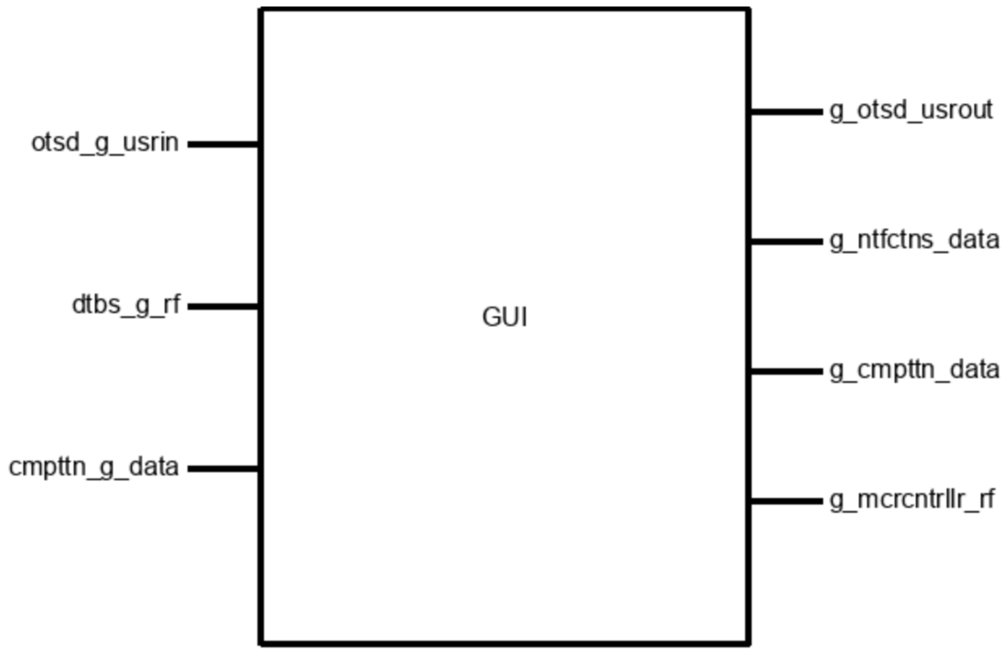
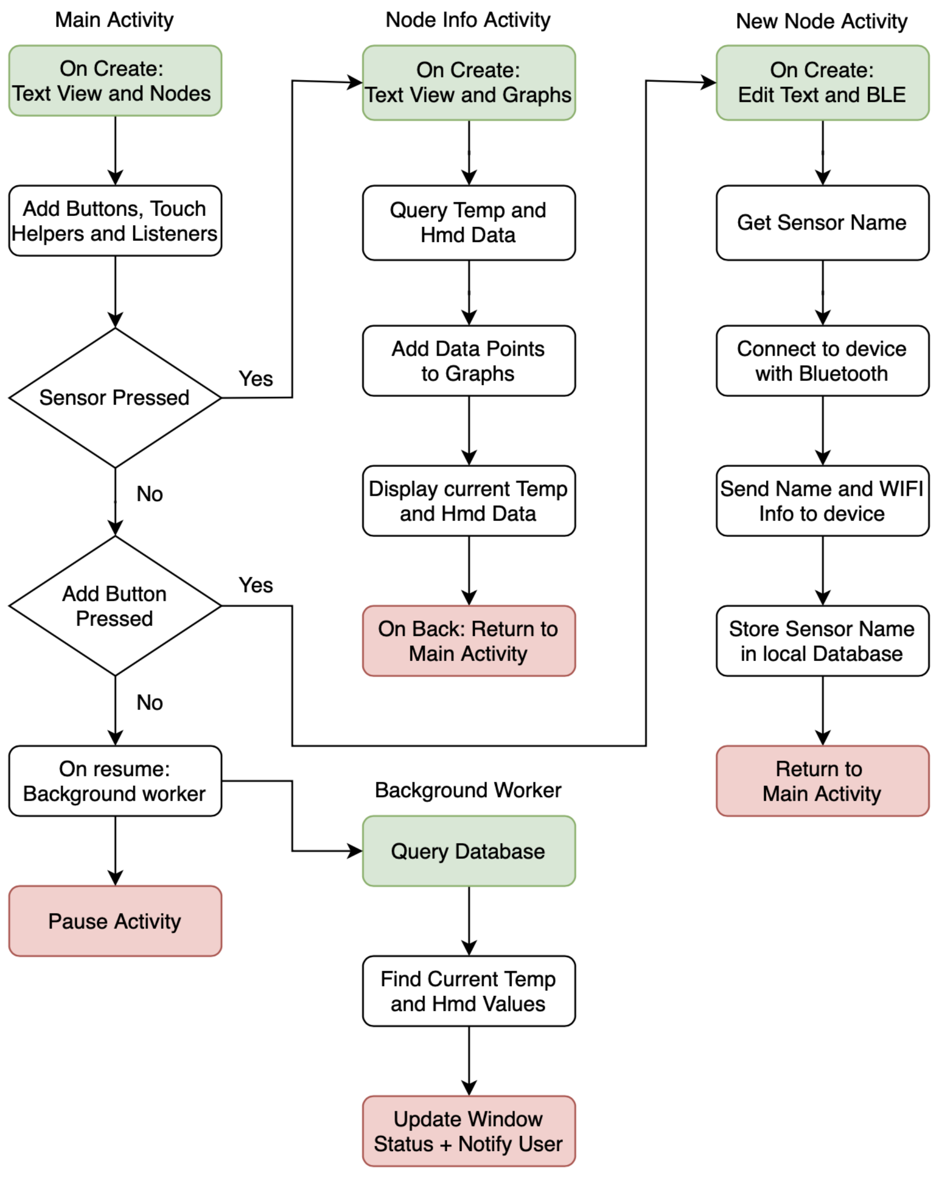


**Figure 4.1: GUI Black Box Diagram**

The GUI block is made of three main pages or activities: the main activity, the new node activity, and the node info activity. Each activity consists of actions that are executed on creation, actions that happen on resume, and varying functions that run different data transfers and connections between activities and databases within the application. On the main activity, users will find the window status on the top of the page and the connected nodes underneath in list form with the node name, current temperature data and current humidity data. On the bottom right of the main page is a plus button which opens the new node activity when pressed. In the new node activity, users will be asked to provide a node name and connect the sensor device to their local wifi network using bluetooth network provisioning. A connected sensor will be displayed in the main activity and its data will be continuously updated using a background worker activity.

The application also consists of a node info activity. When the user clicks on a sensor in the main activity, the node info activity page is opened. There the user will see name, updated temperature and humidity data as well as additional 24 hour graphical temperature and humidity data. This data is stored in the cloud database and queried when the page opens to populate the graphs. The last major part of the GUI is the background worker activity, which is where all the data queries, local database updates, and calculations are handled for the GUI. Figure 4.2 shown below is a basic overview of the GUI code and how activities and its background worker are interconnected.



Main Activity
On Create: Text View and Nodes
Add Buttons, Touch Helpers and Listeners
Sensor Pressed
Yes
No
Add Button Pressed
Yes
No
On resume: Background worker
Pause Activity
Node Info Activity
On Create: Text View and Graphs
Query Temp and Hmd Data
Add Data Points to Graphs
Display current Temp and Hmd Data
On Back: Return to Main Activity
Background Worker
Query Database
Find Current Temp and Hmd Values
Update Window Status + Notify User
New Node Activity
On Create: Edit Text and BLE
Get Sensor Name
Connect to device with Bluetooth
Send Name and WIFI Info to device
Store Sensor Name in local Database
Return to Main Activity

**Figure 4.2: GUI Flow Chart**

### 4.1.3. General Validation

The reason why this block is important to the system is that it is the main part of the software of the entire level system. It pulls cloud data from the database block and uses it to update important UI elements for the user. It allows users to get data visualization for each sensor and each data point created by the sensor. It allows pairing and unpairing of sensors in the application. These are very important requirements of our project partners [1].

The GUI interacts with almost all blocks of the project, for example the notification block is another separate block but is interacting with the GUI constantly in order to notify users about their window status. It is the GUI that is in charge of organizing and providing the block with the necessary information. The GUI is also key to the computation block and provides indoor and outdoor temperature and humidity data as well as collects the desired temperature from the user for the calculation. It also feeds the calculation results to the notification block seamlessly to provide a great user experience. In the end, the GUI block is necessary to display information to the user and transfer data between code blocks in the system [2].

The GUI also interacts with the microcontroller by sending node name and wifi information over bluetooth. When the sensor needs to be unpaired, the GUI will send the sensor a data packet telling the system to clear its flash and restart. This is another project partner requirement which makes it very important to the system [1].

This android application will work for the system due to the multitude of android java libraries and documentation available. This being our first app, we felt it was important to start with a platform that was accessible unlike iOS where many libraries are closed off and unusable [3].

### 4.1.4. Interface Validation

This section contains the two interfaces and their properties. The tables below talk about each property value and why the design details meet each property.

**Table 4.1: otsd_g_usrin : Input**

| **Interface Property** | **Why is this interface this value?** | **Why do you know that your design details <u>for this block</u> above meet or exceed each property?** |
|---|---|---|
| Other: wifi information (struct) ex. username: mywifi, pw: 1234 | Wifi credentials are presented to the microcontroller in the form of an ssid and password | The GUI will allow the user to enter this information before sending to the microcontroller |
| Other: desired temp (float) ex. 71 | The desired temp is stored in local data storage as a float, easy to read for the user | The block pulls up a popup window to allow the user to enter the temperature |
| Other: sensor name (string) ex. Indoor | The sensor name is stored in local data storage as a string, | The block allows the user to enter the node name when connecting |

| | | |
|---|---|---|
| | user need to know the name | the sensor to the network |

**Table 4.2: g_otsd_usrout : Output**

| | | |
|---|---|---|
| Other: Sensor Data (Graph) ex. 24 hr line graphs for temp and hmd | Graphical data is easily represented and is easy to view by the user | The GUI will query all datapoint and use an android graphing library to display the points to the user |
| Other: Window Status (Text Box) ex. Closed | The window status text box is present to let the user know the current window status at all times | The GUI will query data to update window status regularly, along with using a string to update the UI element for the user |
| Other: Sensor Data (Node) ex. Indoor, temp: 72, hmd: 45.6 | Sensor data needs to be easy to read for the user, string and floats with one decimal provide an easy to read interface | The block will displayed updated data for the user on the main activity every 15 min or on resumption of the activity |

**Table 4.3: g_ntfctns_data : Output**

| | | |
|---|---|---|
| Messages: closed notification (string) ex. Close Window | The closed notification message is this value to make sure notification are clear for the user | The notification builder take in string data to set the text of the notification |
| Messages: notification type (int) ex. 0, 1, 2 | The notification integer needs three states (no change, open, closed) to pick the proper notification type | The notification int is used to build the proper notification, the design details will meet the property |
| Messages: open notification (string) ex. Open Window | The open notification message is this value to make sure the user clearly understands the window status | The notification builder take in string data to set the text of the notification |

**Table 4.4: g_cmpttn_data : Output**

| | | |
|---|---|---|
| Messages: outdoor hmd (double) ex. 75.8 | Outdoor humidity is a double in order to calculate the most accurate information for the user | When performing calculations in Java, the numbers involved must be double or float values or else the final result may be truncated, giving inaccurate readings. |
| Messages: indoor temp (double) ex. 75.2 | Indoor temperature is a double in order to calculate the most accurate information for the user | When performing calculations in Java, the numbers involved must be double or float values or else the final result may be truncated, giving inaccurate readings. |
| Messages: desired temp (float) ex. 71 | Desired temperature is this value to make it easy for the user to enter the value. The value is a float to aid in future calculations. | The block sends the temperature to check if the interior temperature gets too high. Calculations utilizes the float to perform accurate results. |
| Messages: outdoor temp (double) ex. 68.9 | Outdoor temperature is a double in order to calculate the most accurate information for the user | When performing calculations in Java, the numbers involved must be double or float values or else the final result may be truncated, giving inaccurate readings. |
| Messages: indoor hmd (double) ex. 57.3 | Indoor humidity is a double in order to calculate the most accurate information for the user | When performing calculations in Java, the numbers involved must be double or float values or else the final result may be truncated, giving inaccurate readings. |

**Table 4.5: g_mcrcntrllr_rf : Output**

| | | |
|---|---|---|
| Messages: wifi information (struct) ex. username: mywifi, pw: 1234 | Wifi credentials are presented to the microcontroller in the form of an ssid and password | The GUI will allow the user to enter this information before sending to the microcontroller |

| | | |
|---|---|---|
| Messages: sensor name (string) ex. Indoor | The sensor name is stored in local data storage as a string, user need to know the name | The block allows the user to enter the node name when connecting the sensor to the network |
| Messages: reset (bool) ex. false | A boolean is used for resetting the sensor to limit the amount of data sent | The GUI will send the boolean when the user deletes the sensor from the application |

**Table 4.6: dtbs_g_rf : Input**

| | | |
|---|---|---|
| Messages: temperature (float) ex. 72.1 | The temperature and humidity data were chosen to be float based for our sensor accuracy. Rounding to the tenths place helps users to visualize the data easily within our user interface. We also want the data to be easily to graph and visualize, and this form helps us achieve that goal. | This data format is easiest to write and query from the database. While the database can handle all forms of data, numbers with only a few decimal places are best. |
| Messages: tag (string) ex. user1, indoor | Tags for the data were chosen because we needed to separate data by sensor and by account. For Influx DB, tags are normally strings for ease of use when querying the output. | This meets the standards for Influx DB, tags must be string data. I also know it is easiest to use tags to display cleanly to the user. |
| Messages: time stamp ex. 2023-1-12T20:36:45 | The time stamp was chosen for the output due to the requirement to visualize data in our application. This format is what comes standard with Influx DB and can be easily put into a Date object for graph creation. | Our design specifies time as a key data value for our GUI. Using this interface value from the time series database is best for our purposes of data visualization and using time as a component in our window open and close calculations. |
| Messages: humidity (float) ex. 80.5 | The humidity data were chosen to be float based for our sensor accuracy. Rounding to the tenths | This data format is easiest to write and query from the database. While the database can handle all forms |

| | | |
|---|---|---|
| | place helps users to visualize the data easily within our user interface. We also want the data to be easily to graph and visualize, and this form helps us achieve that goal. | of data, numbers with only a few decimal places are best. |

**Table 4.7: cmpttn_g_data : Input**

| | | |
|---|---|---|
| Messages: notification type (int) ex. 0 (0 is none, 1 is open window, 2 is close window) | The notification type integer is used to determine the notification sent to the user. An int is used since three states are needed for window state: open, close, or no change. | The design of the block sets the integer to 0 unless a state is changed, therefore it meets the expectations for this property. |
| Messages: window status (boolean) ex. false (true is open, false is closed) | The window status boolean is used since the current window status just needs two states, opened or closed. | The design of this block sets the boolean true if all the sensor specifications are met or defaults to false. The design meets expectations for the window status of the system. |
| Messages: window string (String) ex. "Closed" | The window string changes along with the window status boolean, it works the same way but allows the UI to easily update. | The design of this block for this property is the same as the boolean, therefore it continues to meet expectations. |

### 4.1.5. Verification Plan

This section talks about the verification plan for the GUI block. We can break down the verification into activities to make the input and output comparisons easier.

Main Activity
1. Open main activity
2. Check that sensor data and window status updates
3. Check tapping plus opens new node activity
4. Check tapping sensor opens node info activity

New Node activity
1. Enter sensor name
2. Connect and send wifi + name to sensor
3. Confirm that data is being received on the Main activity from the sensor
4. Check that deleting node sends boolean to sensor

Node info Activity
1. Check that updated name, temp, and hmd data is properly displayed
2. Check that graphs are updated and display correct values and times

Background Worker
1. Check that database values are being queried and stored correctly
2. Check that proper indoor and outdoor values are being sent to the computation block
3. Check that notification and status information is being revived from the computation clock
4. Check that the notification information is being sent to the notification block

### 4.1.6. References and File Links

[1] F. Waqar, "Sensor System for Self-Driven In-Home Climate Control (ECE)," *EECS Project Portal*, Sep-2022. [Online]. Available: https://eecs.engineering.oregonstate.edu/capstone/submission/pages/viewSingleProject.php?id=9vBa7B1brA8Osgxc.

[2] K. Johnson, "Gui – what is it and why is it important?," *GUI – What Is It and Why Is It Important?*, 05-Jan-2023. [Online]. Available: https://www.wikijob.co.uk/industry/it-technology/what-is-a-gui.

[3] "How To Write Your First Program in Java | DigitalOcean," *www.digitalocean.com*. https://www.digitalocean.com/community/tutorials/how-to-write-your-first-program-in-java

### 4.1.7. Revision Table

| Date Revised | Who? | Reason Revised/Revisions Made |
|---|---|---|
| 04/12/23 | Blake Wiker | Updated Description and Design |
| 2/10/2023 | Yousif Albaker | Black box and flow chart. |

|  |  |  |
|---|---|---|
| 02/08/2023 | Yousif Albaker | Grammar and interfaces. |
| 01/20/2023 | Yousif Albaker | first draft submission |

## 4.2. Notifications

### 4.2.1. Description

The notification block allows the user to be notified when their windows need to be opened or closed. The notification block will take in String and integer data values and output push notification to the user using Android system notification. The input integer will correspond to the type notification sent and the string will be the content of the notification. For example, when the block is passed notification integer 1, the open window notification is built using the string "Open Window". The notification is then sent to the notification manager where the application notification channel displays the message for the user. There will be three output notification states, no change, open window, and close window. When the user taps on a notification, they are brought to the application home page.

### 4.2.2. Design

This section includes the black box diagram of the notification block as well as a flow chart describing the code progression within the block. Figure 4.3, the black box diagram, shows that the block has one input, g_ntfctns_data, and one output, ntfctns_otsd_usrout. Input consists of an int value which determines the notification type, and two strings which fill the open and closed notifications. The notification block outputs three notification states: no change, open notification, and closed notification.

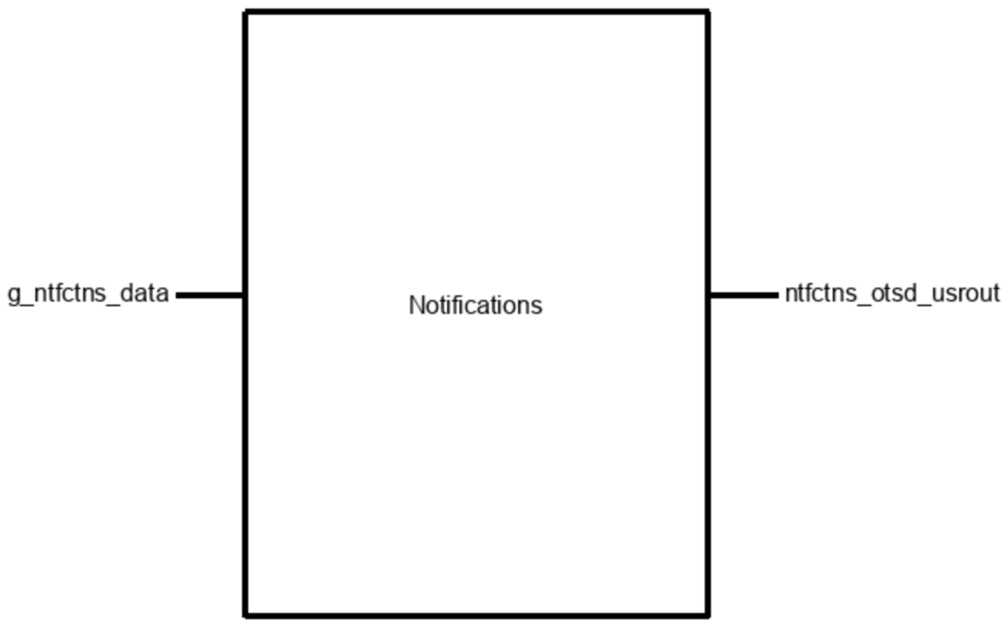


**Figure 4.3: Notification Black Box Diagram**

Figure 4.4 below showcases the flow chart describing the layout of the notification block. At the top, the notification int is called from storage. A notification channel is set up for the application. This channel is given an ID and specified “High Importance” in order for the user to see the notification in a timely manner. After channel creation, a notification manager is opened, which is where specific notifications are called to be built and sent. If the notification integer is a 1, then the open notification string is put in the notification and it is sent to the user. If the notification integer is a 2, then the closed notification string is put in the notification and it is sent to the user. If the notification integer is a 0, then no notification needs to be sent since the window status has not changed.

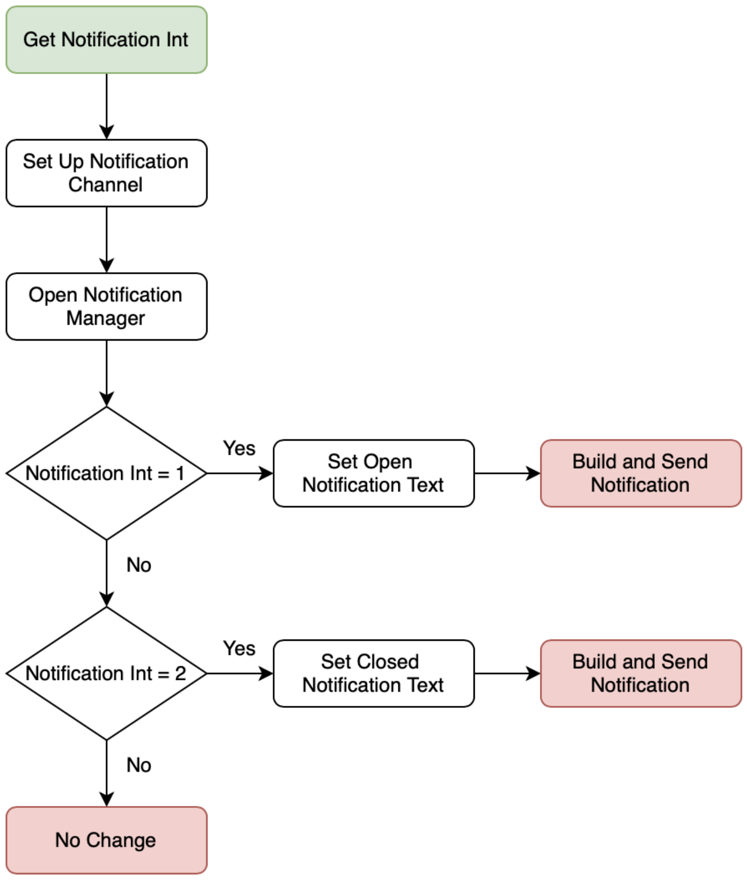

Get Notification Int
Set Up Notification Channel
Open Notification Manager
Notification Int = 1
Yes
Set Open Notification Text
Build and Send Notification
No
Notification Int = 2
Yes
Set Closed Notification Text
Build and Send Notification
No
No Change

**Figure 4.4: Notification Flow Chart**

### 4.2.3. General Validation

The notification block is necessary for our project due to our project partner requirements and the nature of our project. The main goal of our system is to notify a user when their windows need to be opened or closed [1]. In order to accomplish this, we must have a code block that sends and displays the window status information to the user based on the data provided by our sensors. The notification block accomplishes this by creating a notification channel for the application and consistently sending notification to that channel whenever the window status changes. The notification block is an integral and necessary part of the system to accomplish our goals.

This block will work for our system because it has been proven using android studio examples from the android development website. In the example, basic notifications are created using android notification channels, builders, and managers [2]. Different function calls like set content title or set content text are used to change the type of notification using the strings provided to the block. Managers use the function “notify” to send the notification to the android system where the user can see it.

We could also have implemented this using text notification from the application. A user's phone number could have been entered to receive the notifications. This process was set aside in favor of the process above since it could prove more cumbersome than implementing a system push notification to the testing device and provide less information than the android system can provide if needed.

#### 4.2.4. Interface Validation

This section contains the two interfaces and their properties. The tables below talk about each property value and why the design details meet each property.

**Table 4.8: g_ntfctns_data : Input**

| **Interface Property** | **Why is this interface this value?** | **Why do you know that your design details for this block above meet or exceed each property?** |
|---|---|---|
| Messages: closed notification (string) ex. Close Window | The closed notification message is this value to make sure the user clearly understands the window status | The notification builder take in string data to set the text of the notification |
| Messages: notification type (int) ex. 0, 1, 2 | The notification integer needs three states (no change, open, closed) to pick the proper notification type | The notification int is used to build the proper notification, the design details will meet the property |
| Messages: open notification (string) ex. Open Window | The open notification message is this value to make sure the user clearly understands the window status | The notification builder take in string data to set the text of the notification |

**Table 4.9: ntfctns_otsd_usrout : Output**

|  |  |  |
|---|---|---|
| Other: closed notification (android system notification) | Android system notifications are the output of the notification manager notify function | The notification block packages notifications using notification builder for the notification manager, which meets the property expectations. |
| Other: no change (android system notification) | Android system notifications are the output of the notification manager notify function | The notification block packages notifications using notification builder for the notification manager, which meets the property expectations. |
| Other: open notification (android system notification) | Android system notifications are the output of the notification manager notify function | The notification block packages notifications using notification builder for the notification manager, which meets the property expectations. |

#### 4.2.5. Verification Plan

This section details the verification plan for the notifications block. The plan will go through the process of verifying that the input interfaces produce the output interfaces.

1. Open Android studio and build the project. Once the project has been built, open the background worker class.
2. Find the notification integer and set the value to 0.
3. Open application on device and show that no notification is sent when the main activity is refreshed.
4. Open the background worker class and set the integer value to 1.
5. Check to make sure the open string is updated in the notification builder function.
6. Open the application and show that the open notification is displayed.
7. Open the background worker class and set the integer value to 2.
8. Check to make sure the closed string is updated in the notification builder function.
9. Open the application and show that the closed notification is displayed.

#### 4.2.6. References and File Links

[1] F. Waqar, “Sensor System for Self-Driven In-Home Climate Control (ECE),” *EECS Project Portal*, Sep-2022. [Online]. Available: https://eecs.engineering.oregonstate.edu/capstone/submission/pages/viewSingleProject.php?id=9vBa7B1brA8Osgxc.

[2] "Create a notification: Android developers," *Android Developers*. [Online]. Available: https://developer.android.com/develop/ui/views/notifications/build-notification

#### 4.2.7. Revision Table

| Date Revised | Who? | Reason Revised/Revisions Made |
|---|---|---|
| 03/12/2023 | Blake | Updated Design and General Validation |
| 03/08/2023 | Yousif | Initial Creation |

## 4.3. Enclosure

### 4.3.1. Description

The enclosure block in our project consists of a water resistant container that houses our device which consists of a battery, PCB, and sensor.

### 4.3.2. Design

In this section, we will explain the overall design of the block consisting of the black box diagram for the input and output, and our preliminary design for the enclosure.

If we take a look at the picture of the black box diagram in Figure 4.5, we can see that our block consists of 1 input and 1 output. The input would be "otsd_enclsr_envin" which is from the environment to the enclosure such as water splashes and mist. This is our input for environment protection which would be to withstand IPX4. Our output would be "enclsr_pwr_lctmcs_mech" which describes our case mounting inside for our PCB as well as the ability to install a USB C connector that goes through the case (outside to inside) which will allow the ability to provide power to the device via a USB C charging cable. In Figure 4.6, is our design developed on AutoCAD which shows a 2 piece enclosure which would be a top and bottom. The 2 pieces are almost identical  besides one piece having a port hole for the USB C connection. This enclosure has a hole for the sensor to be mounted externally beside the enclosure. Each piece also has a groove on the touching surfaces to provide a space for an o-ring to make the enclosure IPX4.

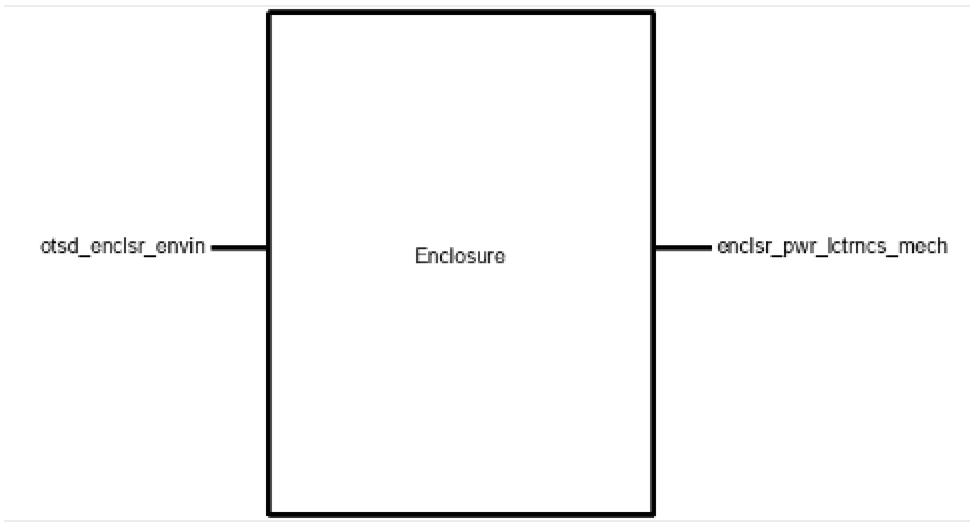


**Figure 4.5: Black box diagram displaying our inputs and outputs**

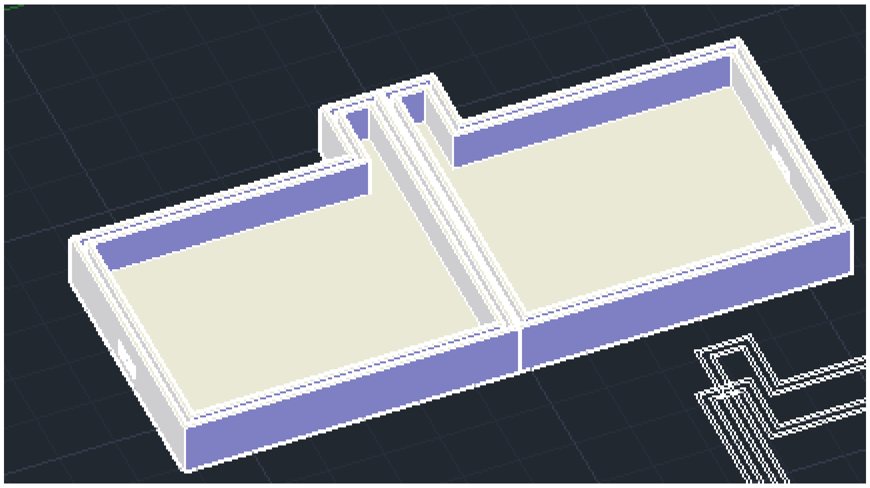


**Figure 4.6: Design of the enclosure block**

### 4.3.3. General Validation

The main purpose of this block is to house all components of our device in an enclosure that is of limited size and of IPX4 water resistant. This enclosure is there to provide protection for all components of our system. The items that mount inside the enclosure would be a LI-PO rechargeable battery, a custom PCB for the power electronics and microcontroller, as well as serve as a mounting apparatus for our external temperature and humidity sensor which mounts alongside our enclosure. The enclosure was designed to to perfectly fit within our size constraint of 3” X 3” X 1” with a cut out to allow room for our external sensor to fit while still being within our size constraint.

As our device also has a rechargeable battery, in order to be able to charge that battery , a USB C connection hole was made in the side of the enclosure where A USB C receptacle will be located to allow the drive to be charged without having to take apart the enclosure.

Another major part of the enclosure design would be its ability to withstand water with an IPX4 rating. This is to allow the device to be mounted outdoors in lightly wet condition and provide protection to the water sensitive components inside. Our device has a groove on both side of the enclosure to allow a round rubber o-ring to be placed inside the groove that way when both sides of the enclosure are connected, a tight seal is formed which will prevent and sprays of water or light splashes of water to get inside the enclosure and prevent damage to the internal components..

One cause of concern for our enclosure would be the ability to dissipate any heat formed inside the enclosure. We do not anticipate any major heat to be created but, with a LI-PO battery being charged and several voltage regulators inside, the potential for an increase in temperature is a possibility. And with an almost sealed enclosure, there are very few areas for air and heat to escape outside the enclosure.

### 4.3.4. Interface Validation

**Table 4.10: otsd_enclsr_envin: Input**

| **Interface Property** | **Why is this interface property this value?** | **Why do you know that your design details <u>for this block</u> above meet or exceed each property?** |
|---|---|---|
| Size 3” X 3” X 1” | This is the size that our block can have as its maximum. Meaning all parts including the enclosure must fit within this size. | Our design will fit inside this size constraint as the enclosure itself is designed just under the max size and all of our components fit within the enclosure. |
| IPX4 (light splash and light spray) | This interface property explained that the enclosure must provide a water resistant protection of IPX4. | Our design meets the interface as the enclosure was designed to have water repelling factors such as a rubber gasket to prevent water seepage. |

**Table 4.11: enclsr_pwr_lctmcs_mech: output**

| | | |
|---|---|---|
| USB C | This interface explained that the enclosure must have the ability to accept a cable of USB C size. | Our design meets this property as on the side of our enclosure is a USB C connector hole design to snuggly fit a USB C cable. |
| 3mm mounting Holes: at least 2) | This interface property explains that within the enclosure, there must be at least two 3 mm mounting holes to mount the PCb and or enclosure parts together. | Our design meets this property as our PCB was designed to have 2 mounting holes for proper securance. Our enclosure also has 2 standoffs inside to provide a mounting spot for those components with a size hole of approximately 3mm. |

### 4.3.5. Verification Plan

**To test the IPX4 water resistance (light splash and light spray).**
1. Assemble the enclosure with necessary parts to ensure water resistance such as o-ring.
2. Insert a dry piece of paper into the enclosure.
3. Ensure the enclosure is properly sealed and fastened.
4. Hold the enclosure in hand and use a spray bottle to lightly spray water on the enclosure from several angles.
5. Open enclosure and ensure paper is not wet.
6. Re-close the enclosure and use a bottle to light splash the enclosure from several angles.
7. Open enclosure and ensure paper is not wet.

**To test size constraints.**
1. Assembly enclosure with all parts.
2. Use a tape measure to measure the length, width and height of the enclosure.
3. Enclosure should not be over 3”x3”x1”

**To test the USB C cable hole.**
1. Assembly enclosure with all parts.
2. Use a standard USB C cable and put the cable into the USB C hole located on the side of the enclosure.
3. Ensure the cable is able to fit inside the hole.

**To test the PCB fasteners.**
1. Assembly enclosure with all parts.
2. Prior to closing the enclosure ensure there are at least two 3mm mounting screws.

#### 4.3.6. References and File Links

[1] “IP ratings,” *IEC*. [Online]. Available: https://www.iec.ch/ip-ratings.

File Link: https://drive.google.com/drive/u/1/folders/0AOJNSLM5IKJIUk9PVA

#### 4.3.7. Revision Table

| Date Revised | Who? | Reason Revised/Revisions Made |
|---|---|---|
| 03/12/2023 | James | Added General Validation |
| 03/09/2023 | James | Initial Creation |

### 4.4. Power Electronics

#### 4.4.1. Description

The power electronics block in our project consists of the design of a battery charging circuit and the power managing design for the microcontroller and the power for the charging circuit. This would mean an output of 5V for a microcontroller, and output of 3.3V for a chip, a charging circuit for a 3.7V battery, and the ability to receive 5V charging power.

#### 4.4.2. Design

In this section, we will explain the overall design of the block consisting of the black box diagram for the input and outputs, PCB schematics, Parts list, wiring diagram, and other design documents.

If we take a look at the black box of our diagram in Figure 4.7 we can see that the block consists of 2 inputs and 2 outputs. Of the 2 inputs we have, one input labeled “otsd_pwr_lctrncs_dcpwr” would be the outside power coming into the block. This input is 5v from external power such as a wall outlet with a standardized USB converter. This is used to charge the battery of the device. However, this input does not directly connect to the battery as the 5V voltage would overpower the 3.7-volt battery. From the external power, goes into a circuit to reduce the voltage such as a buck converter. That power would then go into a lipo battery charging circuit to appropriately regulate the voltage so as to not overcharge and damage the battery. Our other input for this block is “enclsr_pwr_lctrncs_mech” which is how the enclosure attaches to the pcb, namely a usb-c connector and mourning holes. Next is our two outputs which is regulated voltage coming from the the battery to the sensors labeled “pwr_lctrncs_snsrs_dcpwr” and our power from our boost converter that goes directly to the microcontroller labeled “pwr_lctrncs_mcrcntrllr_dcpwr”. All of these circuits will be on one PCB that also has a daughterless microcontroller on it which is a custom designed microcontroller. That way due to a constraint, we can save the most amount of space for our entire device to be small.

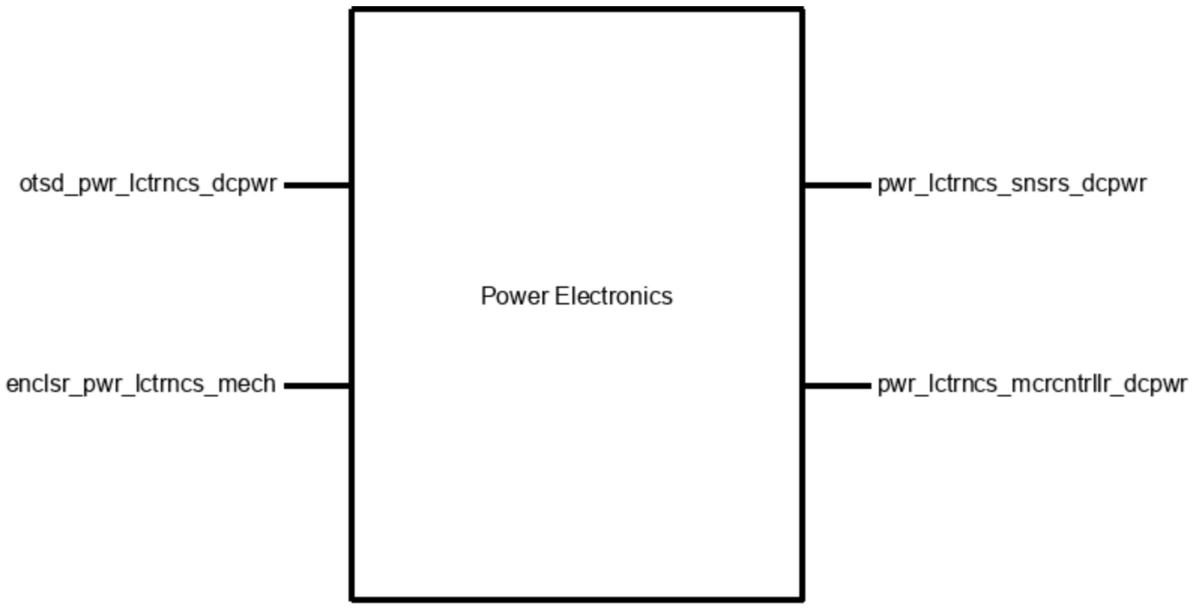


**Figure 4.7: Black box diagram displaying our inputs and outputs**

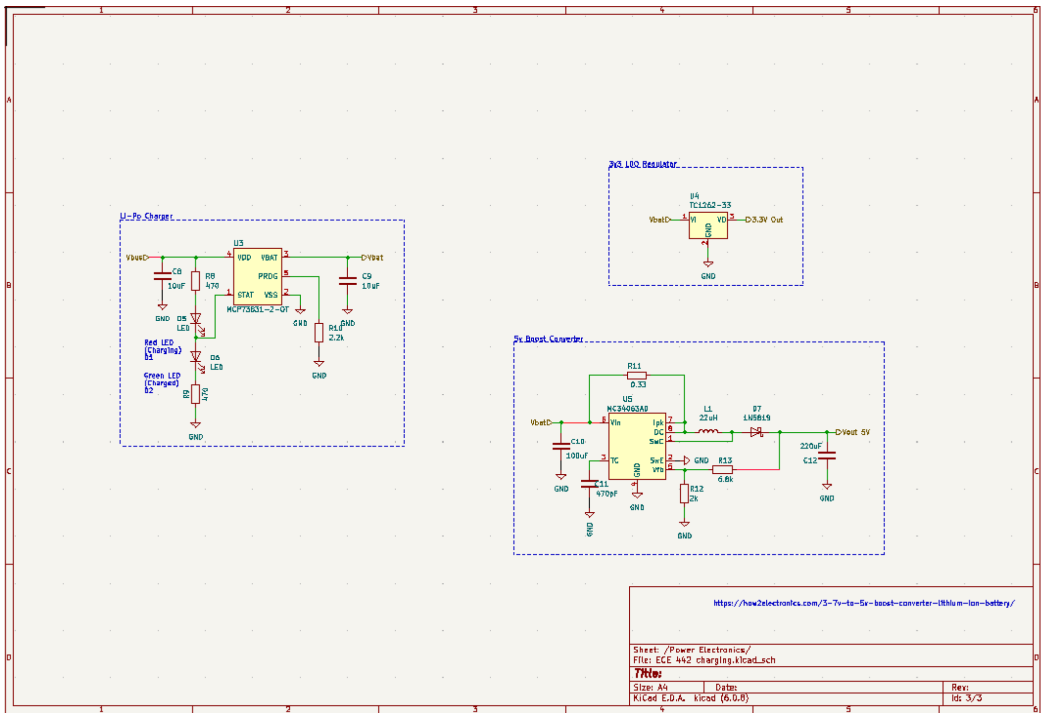


**Figure 4.8: Design of the power electronics block.**

### 4.4.3. General Validation

The main purpose of this block is to manage the device's needs in terms of power consumption. Since we are working with several hardware components, the voltages needed vary. We are needing to power our microcontroller which operates off 5 volts. We are also working with our humidity and temperature sensor that operates at 3.3 volts. Normally, the microcontroller would have a built-in voltage regulator that reduces the microcontroller’s operating voltage of 5 volts to 3.3 volts, but since this is a custom daughter microcontroller, that part of the hardware design has been placed into this block. We also have our 3.7-volt lipo battery that must have a sustained voltage of 4 volts to charge. This is why we need an additional buck converter to reduce the 120-volt wall outlet to 5-volt from a pre-made circuit.

The PCB that these circuit designs will be mounted on will be a 2-layer PCB custom-made by a local company such as JLC PCB, 4PCB, and or other distributors depending on the cost of the board and the shipping time. The components on the PCB will prioritize surface mount components with a size around 0805 to keep the PCB as small as possible to both fit in our 3”x3”x1” enclosure and also conserve costs as the bigger the PCB layout is, the cost goes up exponentially. As we are working with relatively low-power components that do not draw over 500mA, the trace width on the PCB can be relatively small. An additional option we have explored to reduce the size of the PCB would be on our daughterless microcontroller would be to only install the pinouts needed for our device instead of all connection on the microcontroller.

One major cause for concern that we have anticipated is the availability of parts and current shipping delays. While prototyping our blocks, we have ensured that we order at least 3 extra of each part that way we have 2 extra parts to build 2 other devices as required by our engineering requirements, and 1 extra as a backup in case we lose components due to their extreme size or a mistake happens while testing. That way, we do not have to wait an additional week for parts to ship, or for the rare chance the part we order is no longer in stock. The testing process for this block will consist of breadboard-mounted components with breakout boards for SMD components that you can not get for through-hole parts.

### 4.4.4. Interface Validation

| Interface Property | Why is this interface property this value? | Why do you know that your design details <u>for this block</u> above meet or exceed each property? |
|---|---|---|

**Table 4.12: enclsr_pwr_lctrncs_mech : Input**

| | | |
|---|---|---|
| USB C | This interface explained that the enclosure must have the ability to accept a cable of USB C size. | Our design meets this property as the power electronics block has a usb-c receptacle on the PCB |
| 3mm mounting Holes: at least 2) | This interface property explains that within the enclosure, there must be at least two 3 mm mounting holes to mount the PCb and or enclosure parts together. | Our design meets this property as our PCB was designed to have 2 mounting holes for proper securance. Our enclosure also has 2 standoffs inside to provide a mounting spot for those components with a size hole of approximately 3mm. |

**Table 4.13: otsd_pwr_lctrncs_dcpwr: Input**

| | | |
|---|---|---|
| Vnominal: 5V | This value is the standard voltage of a buck boost convert output when connected to a normal 120 volt wall outlet. | For USB Wall Charger:<br>As our device will only need this component for charger the LIPO battery, the 5 volts is sufficient enough as our battery will require around 4 volts to have efficient charger capabilities.It will meet this value because our design is configured to accept this value. |
| Inominal: 150mA | This Is the current that the wall converter can output. | For USB Wall Charger:<br>This is the charger current the 120 volt to 5 volt converter can output which is plenty of current for the battery as our entire battery capacity is 2 amps. Within our charging circuit we will have to put a limiter as to not over charge or charge the battery too quickly with the 2 amps.It will meet this value because this is the standard range AC-DC converters come in. |

|  |  |  |
|---|---|---|
| Ipeak: 500mA | This max current supported by our system | For USB Wall Charger: Within our charging circuit we will have to put a limiter as to not over charge or charge the battery too quickly with the 500mA.It will meet this value because this is the standard range AC-DC converters come in. |
| Pnominal: 20W | This is the normal power that the wall outlet converter can output. | For USB Wall Charger: This power value allows for quick charging capabilities and a wattage this high can change our 2 amp capacity battery fairly quickly. This can be found on the description of the power converter enclosure.It will meet this value because this is the standard range AC-DC converters come in. |

**Table 4.14: pwr_lctrncs_mcrcntrllr_dcpwr: Output**

|  |  |  |
|---|---|---|
| Inominal: 30mA | This will be the average current our device could see | For the Boost Converter: This is relatively an estimate on what our overall device could draw. For this interface, we are looking at the power that goes directly to the microcontroller. |
| Ipeak: 1.5A | This is the largest current that our microcontroller can support | For the Boost Converter: The converter has the capability to output this current as a max value according to the datasheet |
| Vmax: 5.1V | The maximum voltage our microcontroller can use to function reliably | For the Boost Converter: The converter can supply 3V to 40V according to the datasheet |
| Vmin: 4.6V | The minimum voltage our microcontroller can use to function reliably | For the Boost Converter: The converter can supply 3V to 40V according to the datasheet |

| | | |
|---|---|---|
| Vnominal: 5V | This is the necessary voltage that our microcontroller requires. | For the Boost Converter:<br>We will be boosting our 3.7 volts that come from the battery to the 5 volts our microcontroller needs. |

**Table 4.15: pwr_lctrncs_snsrs_dcpwr: Output**

| | | |
|---|---|---|
| Inominal: 3mA | This will be the average current our sensor needs to operate | For the Linear Voltage Regulator:<br>This is relatively an estimate on what our overall device could draw. For this interface, we are looking at the power that goes directly to the microcontroller. |
| Ipeak: 5A | This is the maximum possible current drawn by a pin on the sensor | For the Linear Voltage Regulator:<br>Has the ability to supply this peak current according to the datasheet |
| Vmax: 5V | Maximum rating according to the sensor datasheet | For the Linear Voltage Regulator:<br>Has the ability to supply this max voltage according to the datasheet |
| Vmin: 3V | Minimum rating according to the sensor datasheet | For the Linear Voltage Regulator:<br>Has the ability to supply this min voltage according to the datasheet |
| Vnominal: 3.3V | This is the necessary voltage that our sensor requires. | For the Linear Voltage Regulator:<br>We will be reducing our 3.7 volts that come from the battery to the 3.3 volts our other chips need. |

#### 4.4.5. Verification Plan

To test the voltage that comes from the external power supply:

1. Plug the standard, premade 120 volts to 5v plug into the wall.
2. Attach a USB to USB C cable to the converter to the USB C female plug on the test board.
3. Attach the two leads of a multimeter to the positive and negative test terminals on the board

4. View the multimeter screen and the voltage should be 5v give or take 0.1 volts

To test the voltage coming from the battery:

1. Plug the Lipo battery into the test board to the 2 head pins that are labeled + and -.
2. Attach the multimeter leads to the positive and negative test points on the board.
3. View the multimeter which should be 3.7 volts give or take 0.1 volts.

To test voltage going into and out of the buck converter:

1. First, we will test the power going into the buck converter by attacking a multimeter to the test connections before it.
2. After reading the value on the multimeter it should be around 3.7 volts.
3. Next attach the multimeter to the output of the buck converter.
4. After reading the value of the multimeter the voltage should be 3.3 volts

To test the voltage going into and out of the boost converter:

1. First we will test the power going into the boost converter by attacking a multimeter to the test connections before it.
2. After reading the value on the multimeter it should be around 3.7 volts.
3. Next attach the multimeter to the output of the boost converter.
4. After reading the value of the multimeter the voltage should be 5 volts

To test that the LIPO battery is charging:

1. First, discharge the battery through resistors until the battery measures 3 volts.
2. Plug in the 5v external power supply.
3. Attach a multimeter to the output end of the charging circuit.
4. Read the value at which it should be around 4 volts.
5. After, switch the mode in the multimeter to read amperage.
6. After reading the value on the multimeter, the amperage should be below 500mA but about 100mA.

### 4.4.6. References and File Links

[1] “Advanced circuits,” *Printed Circuit Board Manufacturer - PCB Manufacturing and Assembly*. [Online]. Available: https://www.4pcb.com/. [Accessed: 18-Jan-2023].

[2] “EECS Project Portal,” *Single Project*. [Online]. Available: https://eecs.engineering.oregonstate.edu/capstone/submission/pages/viewSingleProject.php?id=9vBa7B1brA8Osgxc. [Accessed: 18-Jan-2023].

[3] H. L. McKeefry, “Global chip shortages cast a long shadow with no end in sight,” *Digi*, 23-Nov-2021. [Online]. Available: https://www.digikey.com/en/blog/global-chip-shortages-cast-a-long-shadow. [Accessed: 18-Jan-2023].

[4] “PCB prototype & PCB fabrication manufacturer,” *JLCPCB*. [Online]. Available: https://jlcpcb.com/?from=VGB&gclid=CjwKCAiAzp6eBhByEiwA_gGq5JuAip5ZtNFF8Jrws3EMcVwZ9XrrlYo6_Qzap6F2ZwWGXhwHkI1vxhoCs20QAvD_BwE. [Accessed: 18-Jan-2023].

Boost Converter Datasheet:
https://www.ti.com/lit/ds/symlink/mc34063a.pdf?ts=1678765528326

Sensor Datasheet File:
https://cdn-shop.adafruit.com/product-files/4099/C13024-002+datasheet.pdf

Linear Regulator Datasheet:
https://ww1.microchip.com/downloads/aemDocuments/documents/APID/ProductDocuments/DataSheets/21373C.pdf

#### 4.4.7. Revision Table

| Date Revised | Who? | Reason Revised/Revisions Made |
|---|---|---|
| 1/17/2023 | James Beans | Creation of Block Validation Draft |
| 2/8/2023 | James Beans | Edits added from feedback from reviewers of the rough draft |
| 3/14/2023 | James Beans | Edits added for interfaces. |

### 4.5. Microcontroller

#### 4.5.1. Description

This is the block that serves as the hardware-side brain for the system. It takes in inputs from the power  supply block and the sensor block and physically sends the information to the software side of the  system. This device receives power from the power block. This block can be broken down into two major components: the ESP module and the communication module. Below is a block diagram detailing how these components communicate, as well as a few lesser modules. It’s important to note this block contains *just the hardware*. Being the ESP32 is a microcontroller, it will  contain code designed to collect, interpret, package and send the information. This is *not* handled in this  block and will not be discussed here.

#### 4.5.2. Design

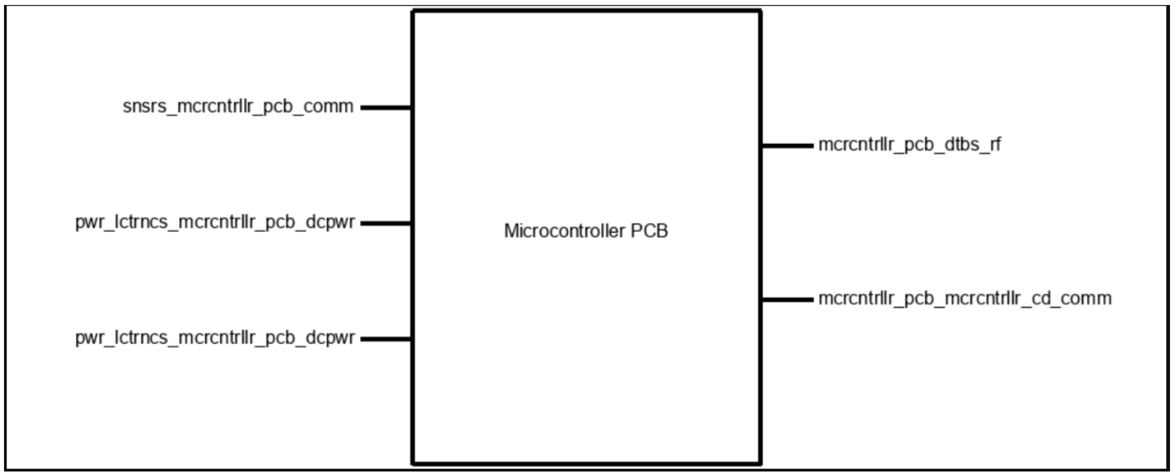

snsrs_mcrcntrllr_pcb_comm
pwr_lctrncs_mcrcntrllr_pcb_dcpwr
pwr_lctrncs_mcrcntrllr_pcb_dcpwr
Microcontroller PCB
mcrcntrllr_pcb_dtbs_rf
mcrcntrllr_pcb_mcrcntrllr_cd_comm

**Figure 4.9: Microcontroller Black Box Diagram**

The circuit represented by the diagram above is largely based on an actual ESP32. The only difference in  fact is that it has been stripped of all functionality not required by our project. This includes streamlining  of active pins, removal of the 5V to 3.3V converter, and the removal of a few connectors and buttons.  Preserved are the ESP module and its USB and UART functionality, as well as a reset button and an  enable button. Added are three connectors and status indicators for power and sensor input.

This block contains two major components and a few minor components. First to be discussed will be  the ESP32 module.

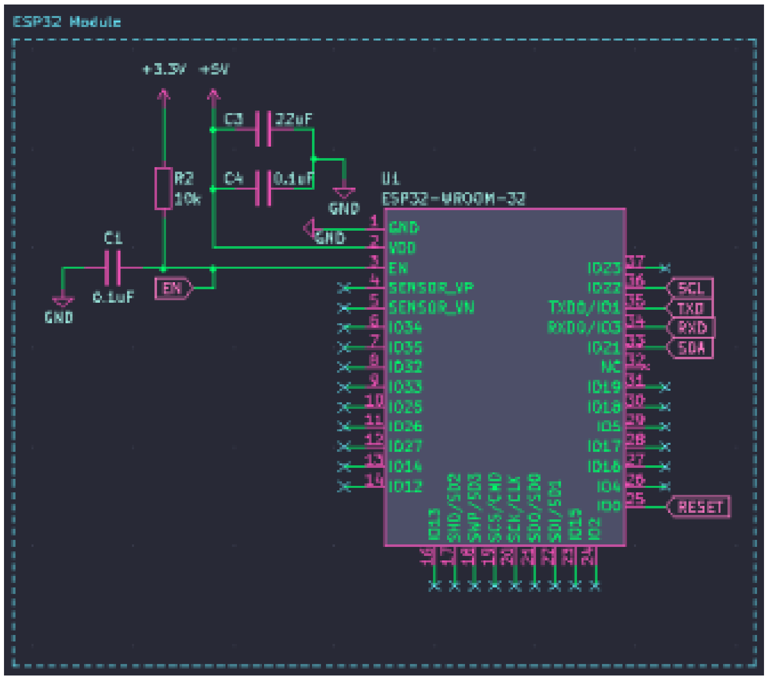


**Figure 4.10: ESP32 Module**

This component contains the actual processor and recommended circuitry powering and enabling the  chip. Used connections are labeled as nets, and all others are marked as not connected (NC or “x”). The  chip receives 5v at Vdd, with two capacitors acting as decoupling capacitors. The 5v is necessary to keep  the processor in “Awake” mode in order to operate at full power. The enable pin is connected between a pull up resistor at 3.3v and another decoupling capacitor. The only other pins used are the dedicated I2C pins (IO22 and IO21), the dedicated UART pins (IO1 and IO3) and the reset pin (IO0).

The second major component is the USB and UART, shown below.

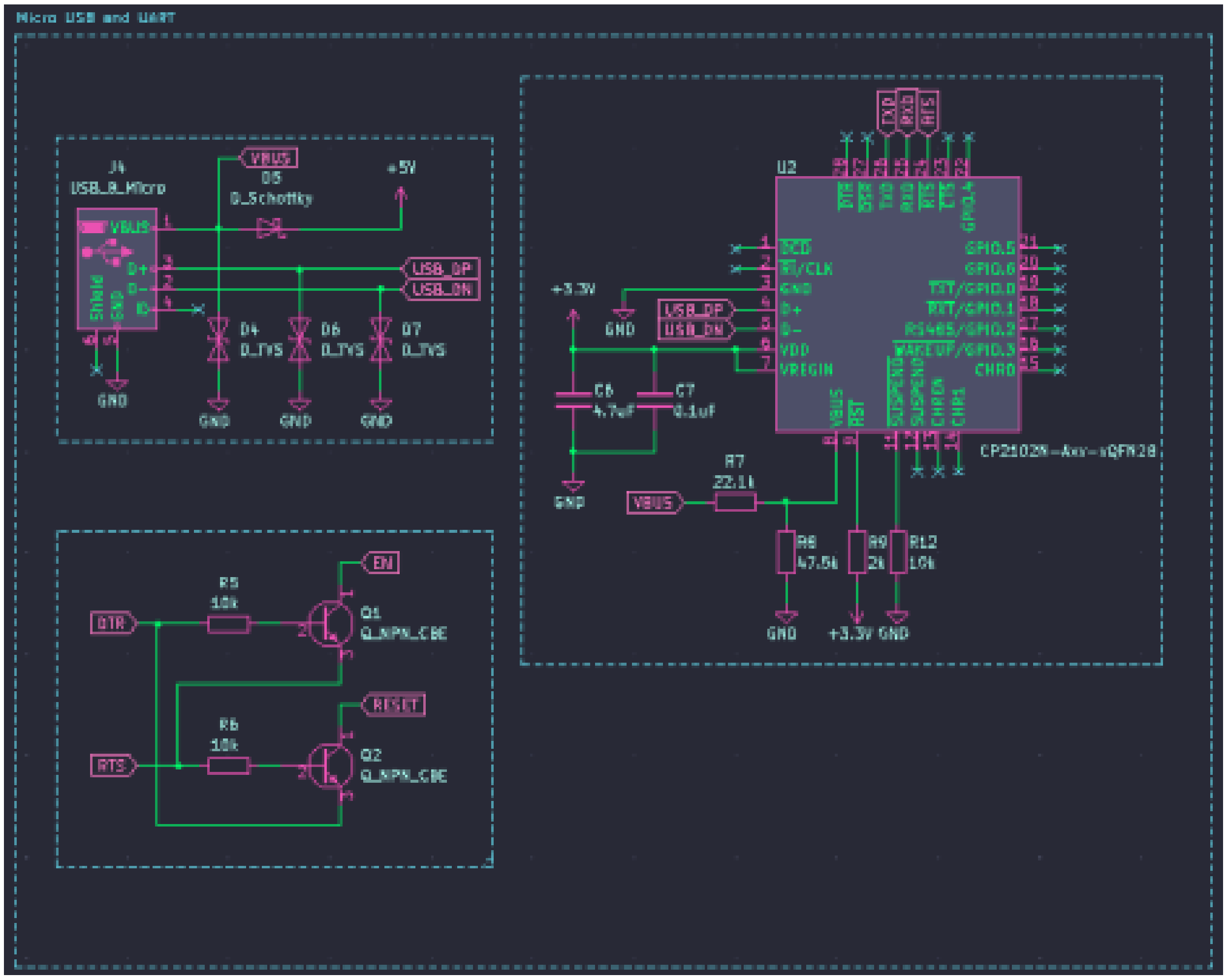


**Figure 4.11: USB and UART**

This component is based around programming the processor. It includes a micro USB port, a UART to  USB converter, and some logic circuitry enabling the programmer.

Moving onto the lesser components are the button assemblies, pictured below.

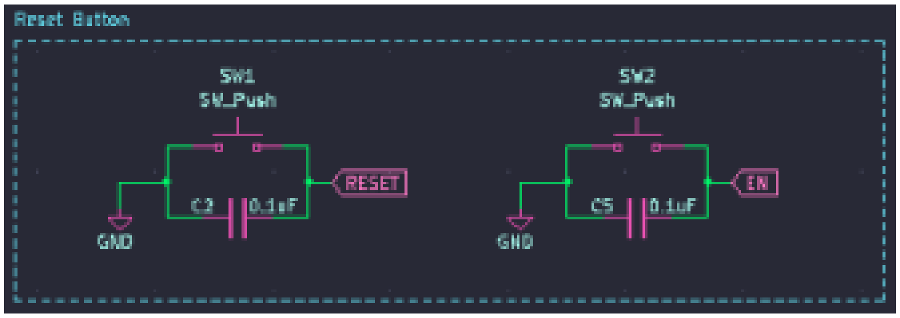


**Figure 4.12: Buttons**

The typical ESP32 contains two buttons: one for reset and another for enable. I chose to include both in my design. The reset button is a necessity in any microcontroller, so it was a straightforward design choice. I chose to include the enable as well in I needed it for the programming as well. Their design is  simple enough, including a button with one end connected to ground and the other connected to their associated pin. There is also a decoupling capacitor to help with debouncing.

The final components are the connectors shown below.

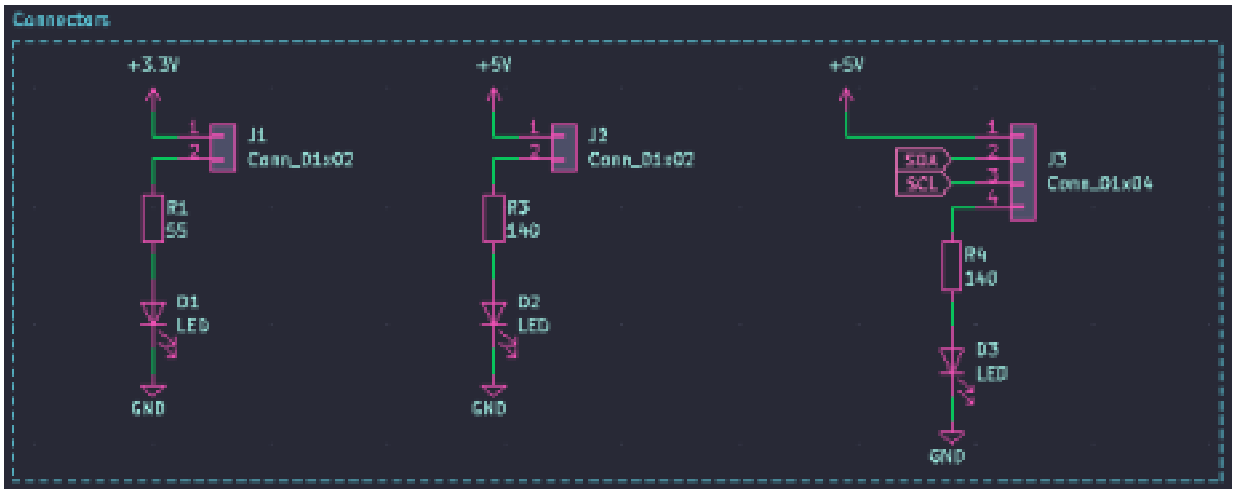


**Figure 4.13: Connectors**

This is the only component of the entire block I designed from scratch. Normally, the ESP32 is powered  through USB and utilizes a series of header pins for communication. I chose to remove the header pins  to streamline the design. As a result of the power not being included in the block and not having these header pins, I needed a way to modularize the inputs. This came in the form of connectors! The two  two-pin connectors are used for 3.3V and 5V power from the power block. The four-pin connector is  used for pulling in an I2C signal from the temperature/humidity sensors. Each of the sensors includes a  status LED in order to determine whether or not it's connected. Too many times chasing down shorts to  leave those out.

Finally, pictured below is the layout for the PCB.

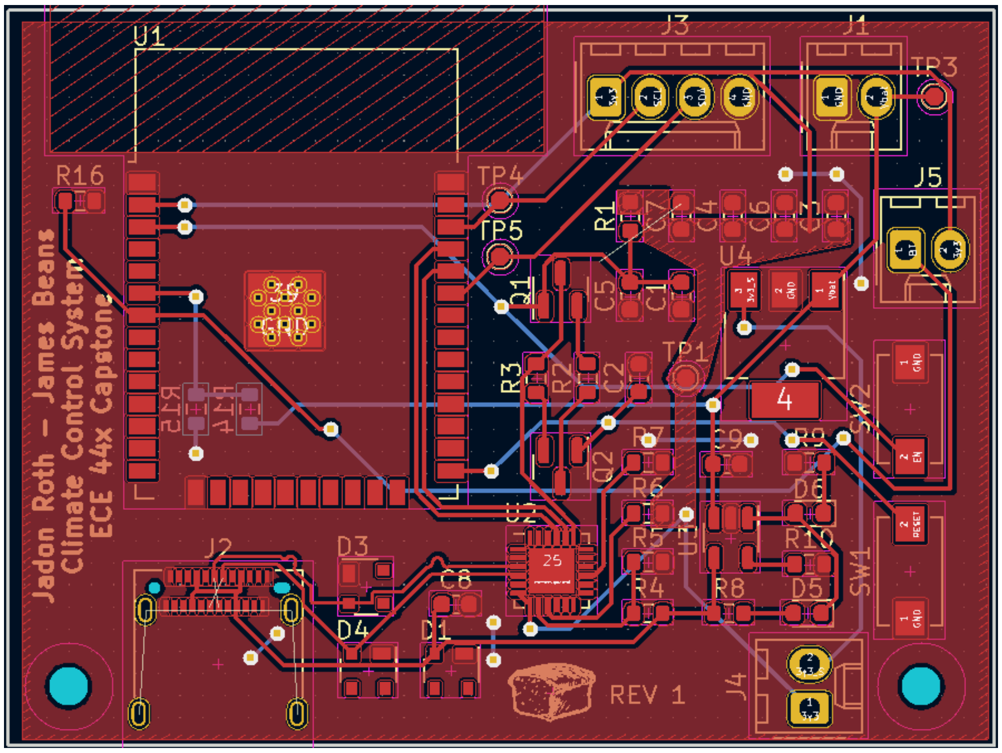


**Figure 4.14: Layout**

### 4.5.3. General Validation

The main purpose of having this block designed the way it is was to reduce cost and to conserve space. One of our system requirements is that the entire system must fit inside a 3" by 3" by 1" cube. As such, a store bought ESP32 is already more than an inch tall, and that is space we don't have. This inspired the custom design while occupying nearly half the space in width and height, nearly a quarter as tall [1]

All the components chosen were chosen due to availability and ease of assembly.

The design itself was fairly simple, as the ESP32 is a well-documented device and all the information is  widely available.

In terms of performance, this block hits all the notes required of it.

In terms of our partner requirements, we hit all the requirements as well.

### 4.5.4. Interface Validation

| Interface Property | Why is this interface property this value? | Why do you know that your design details <u>for this block</u> above meet or exceed each property? |
|---|---|---|

**Table 4.16: g_mcrcntrllr_rf : Input**

| | | |
|---|---|---|
| Messages: wifi information (struct) ex. username: mywifi, pw: 1234 | Wifi credentials are presented to the microcontroller in the form of an ssid and password | The ESP32 has the ability to receive wireless data in multiple different structures |
| Messages: sensor name (string) ex. Indoor | Sensor name is stored in the microcontroller and used to send data to the cloud database | The ESP32 has the ability to receive string data over a wireless connection |
| Messages: reset (bool) ex. false | A boolean is used for resetting the sensor to limit the amount of data sent | The ESP32 has the ability to receive boolean values over a wireless connection |

**Table 4.17: snsrs_mcrcntrllr_comm : Input**

| | | |
|---|---|---|
| Messages: Humidity | Microcontroller needs humidity data to send to the application | The ESP32 can receive I2C communication in the form of messages from the sensor |
| Messages: Temperature | Microcontroller needs temperature data to send to the application | The ESP32 can receive I2C communication in the form of messages from the sensor |
| Protocol: I2C | The sensor uses I2C to communicate with the microcontroller | The ESP32 list I2C communion in the datasheet |

**Table 4.18: pwr_lctrncs_mcrcntrllr_dcpwr : Input**

| | | |
|---|---|---|
| Inominal: 30mA | This will be the average current our microcontroller can see | Average current draw according to the ESP32 datasheet |
| Ipeak: 1.5A | This is the largest current that our microcontroller can support | Largest amount of current the ESP32 can draw for operation according to the datasheet |
| Vmax: 5.1V | The maximum voltage our microcontroller can use to function reliably | ESP32 can run with this max voltage according to the datasheet |
| Vmin: 4.6V | The minimum voltage our microcontroller can use to function reliably | ESP32 can run with this min voltage according to the datasheet |
| Vnominal: 5V | This is the necessary voltage that our microcontroller requires. | The typical operating voltage of our ESP32 is 5V, and the circuit can support this voltage. |

**Table 4.19: mcrcntrllr_dtbs_rf : Output**

| | | |
|---|---|---|
| Messages: humidity (double) ex 80.5 | The humidity readings provided by the sensor will be transmitted as a double to the database | ESP32 has the ability to send data over a WIFI connection to a remote server |
| Messages: temperature (double) ex. 72.1 | The temperature readings provided by the sensor will be transmitted as a double to the database | ESP32 has the ability to send data over a WIFI connection to a remote server |
| Messages: tag (string) ex. Indoor | The sensor name will be sent as a tag to the database | ESP32 has the ability to send data over a WIFI connection to a remote server |

**Table 4.20: mcrcntrllr_mcrcntrllr_cd_comm : Output**

| | | |
|---|---|---|
| Messages: Time | Time readings from the timesync server | ESP32 has the ability time sync with a WIFI connection |

| | | |
|---|---|---|
| Messages: Humidity | The humidity readings are packaged and sent to the microcontroller code | Internally, the ESP32 transfers I2C data to memory and into the code |
| Messages: Wifi information ex. username: mywifi, pw: 1234 | Wifi information is in the form of ssid and password | The ESP32 stores wifi information in its internal flash storage to use with the wifi provisioning functions in the code |
| Messages: Temperature | The temperature readings are packaged and sent to the microcontroller code | Internally, the ESP32 transfers I2C data to memory and into the code |
| Messages: Node name ex. Indoor | Node name is in string form for database tag | The ESP32 code needs a tag to distinguish the sensor data in the database |

**Table 4.21: mcrcntrllr_cd_mcrcntrllr_comm : Input**

| | | |
|---|---|---|
| Messages: Time | Time data is needed for accurate sensor data readings | ESP32 can package the time data into a datapoint for the cloud database write over WIFI |
| Messages: Humidity | The humidity readings in double form are received | ESP32 can package the humidity data into a datapoint for the cloud database write over WIFI |
| Messages: Temperature | The temperature readings in double form are received | ESP32 can package the temperature data into a datapoint for the cloud database write over WIFI |

|  |  |  |
|---|---|---|
| Messages: Node name ex. Indoor | Node name is received as a string for database tag | ESP32 can package the name data into a datapoint for the cloud database write over WIFI |

### 4.5.5. Verification Plan

This is the verification plan for the microcontroller PCB block, it will over all the inputs and outputs for the block:

1. Set power supply to nominal values (30mA, 5V)
2. Connect sensor and conform readings for temperature and humidity
3. Make sure wifi info, node name, and reset boolean are initialized and stored
4. Run code and confirm ESP32 is correctly powered and data is uploaded to the database
5. Set power supply to max (5.1V, 1.5A) and run steps 2 through 4
6. Set power supply to min (4.6V) and run steps 2 through 4

### 4.5.6. References and File Links

[1] F. Waqar, "Sensor System for Self-Driven In-Home Climate Control (ECE)," *EECS Project Portal*, Sep-2022. [Online]. Available: https://eecs.engineering.oregonstate.edu/capstone/submission/pages/viewSingleProject.php?id=9vBa7B1brA8Osgxc.

ESP32 Datasheet:
https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

### 4.5.7. Revision Table

| **Date Revised** | **Who?** | **Reason Revised/Revisions Made** |
|---|---|---|

|  |  |  |
|---|---|---|
| 03/04/2023 | Jadon | Feedback review and update |
| 01/20/2023 | Jadon | Initial creation |

## 4.6. Microcontroller Code

### 4.6.1. Description

This block consists of the program that lives on the microcontroller. The primary function of the program is to collect data from the sensors via I2C and instruct the microcontroller to ship the data off to the online database. The code will also provision the wifi of the microcontroller and receive data from the GUI wirelessly.

This is done by setting a timer on the device and executing certain tasks at certain times. Every 30 seconds the microcontroller is instructed to send data to the database and an onboard heater status is toggled. Every minute data is polled from the sensor and added to a queue of data.

This code was written in such a way that it not only collects data on a schedule, but also has the ability to store data onboard and remove the data in a timely fashion in the event of a network disconnect. When the code is not sending, it is looking for wireless data transmissions from the application.

### 4.6.2. Design

This section details the design of the code block. Figure 4.15 below shows the black box diagram for the block. There is one input, mcrcntrllr_mcrcntrllr_cd_comm, which takes in sensor data, time, and wifi information, and one output, mcrcntrllr_cd_mcrcntrllr_comm, which sends database information back to the microcontroller.

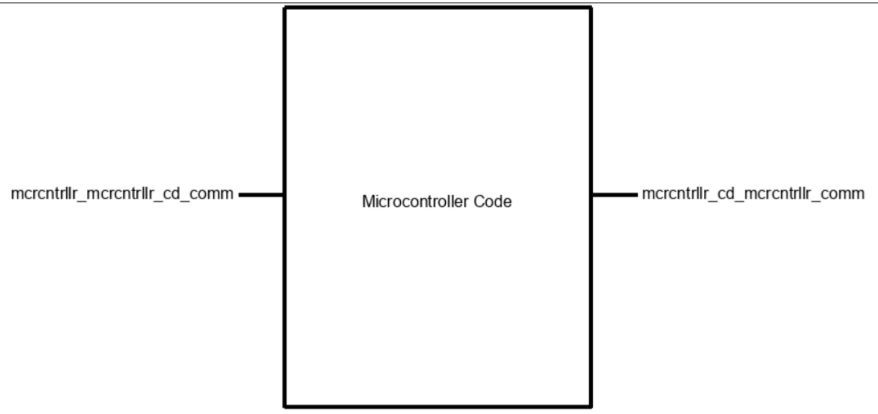


**Figure 4.15: Microcontroller Code Black Box Diagram**

Figure 4.16 is the flow chart depicting the layout of the microcontroller code. In the setup function, the code will begin the wifi provisioning event. Once this event is completed, or the wifi credentials are already saved, the loop is executed. Every 30 seconds, the sensor heater state is changed and the sensor data stored in the queue is sent to the database. Every 60 seconds sensor values are read and stored in the queue along with a time element, that way if the wifi goes down values are still stored.

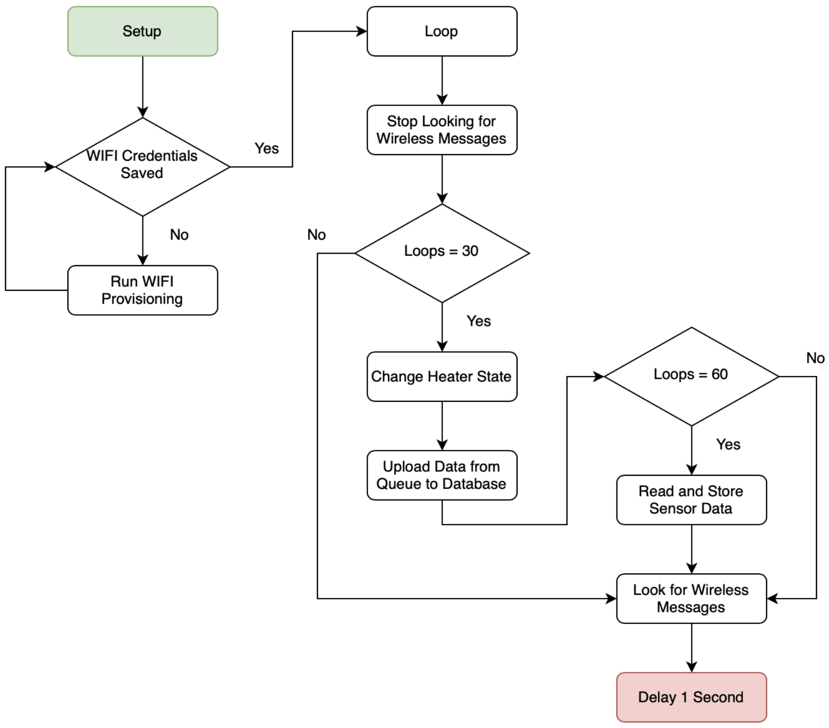

Setup
WIFI Credentials Saved
Yes
No
Run WIFI Provisioning
Loop
Stop Looking for Wireless Messages
Loops = 30
No
Yes
Change Heater State
Upload Data from Queue to Database
Loops = 60
No
Yes
Read and Store Sensor Data
Look for Wireless Messages
Delay 1 Second

**Figure 4.16: Microcontroller Code Flow Chart**

### 4.6.3. General Validation

This block meets the needs of the system. The microcontroller code gathers sensor data, specifically temperature and humidity data, which is a project partner requirement of our system [1]. The code also directs wireless communication between the application and the microcontroller, which is another project partner requirement, namely device pairing and unpairing [1]. Finally, the code sends data to the database, which helps the system fulfill the graphical data visualization requirement listed on the project page [1].

This design will work for the system in many situations. Looping through the code will allow different functions to run continuously at specific times. Changing heater states every 30 seconds makes sure that our temperature and humidity values are the most accurate. Uploading code regularly makes sure that our GUI is updated with the latest information. Having a setup function allows the device to check for wifi connection, which is a critical component of our data collection system.

We could have used different coding techniques, for example Expressif IDF, for more complex code but that would have added unnecessary complications to our simple sensor reader and data uploader code block. We can also handle all the necessary wireless communications using the Arduino platform.

### 4.6.4. Interface Validation

This section details the interface validation for the microcontroller block

**Table 4.20: mcrcntrllr_mcrcntrllr_cd_comm : Input**

| **Interface Property** | **Why is this interface property this value?** | **Why do you know that your design details <u>for this block</u> above meet or exceed each property?** |
|---|---|---|
| Messages: Time | Time readings from the timesync server | The code uses timesync function to sync the time to the pacific time zone using the server |
| Messages: Humidity | The humidity readings are given to the code from the microcontroller | Microcontroller code gathers the humidity readings from the sensor to package into database points |
| Messages: Wifi information ex. username: mywifi, pw: 1234 | Wifi information is in the form of ssid and password | Code uses wifi provisioning library to gather wifi information from the GUI application |
| Messages: Temperature | The temperature readings are given to the code from the microcontroller | Microcontroller code gathers the temperature readings from the sensor to package into database points |

| | | |
|---|---|---|
| Messages: Node name ex. Indoor | Node name is sent to the code in string form | Microcontroller code gathers node name from GUI code over wireless connection |

**Table 4.21: mcrcntrllr_cd_mcrcntrllr_comm : Output**

| | | |
|---|---|---|
| Messages: Time | Time data is needed to created datapoint for the database | ESP32 can package the time data into a datapoint for the cloud database write over WIFI |
| Messages: Humidity | The humidity readings in double form are sent to be packaged into datapoint | ESP32 can package the humidity data into a datapoint for the cloud database write over WIFI |
| Messages: Temperature | The temperature readings in double form are sent to be packaged into datapoint | ESP32 can package the temperature data into a datapoint for the cloud database write over WIFI |
| Messages: Node name ex. Indoor | Node name is needed to tag database points | ESP32 can package the name data into a datapoint for the cloud database write over WIFI |

### 4.6.5. Verification Plan

The section details the verification plan for the microcontroller block:
1. Check that the microcontroller is connecting to the local network
2. Check that the time is syncing correctly with the server
3. Check that the node name is updated within the code (can be hard coded)
4. Use dummy data to replicate the temperature and humidity
5. Run the code for 5 minutes (Ten 30 second intervals, Five 60 second intervals)

6. Confirm that time, temperature, and humidity is being stored in the queue
7. Confirm that point are being written to the database

#### 4.6.6. References and File Links

[1] F. Waqar, "Sensor System for Self-Driven In-Home Climate Control (ECE)," *EECS Project Portal*, Sep-2022. [Online]. Available: https://eecs.engineering.oregonstate.edu/capstone/submission/pages/viewSingleProject.php?id=9vBa7B1brA8Osgxc.

#### 4.6.7. Revision Table

| Date Revised | Who? | Reason Revised/Revisions Made |
|---|---|---|
| | | |
| 03/04/2023 | Jadon | Initial Creation |

### 4.7. Sensors

#### 4.7.1. Description

This block consists of the physical sensors that will be feeding information to the microcontroller which will parse and then send the data to the software side.

As the goal of the system is to report and make decisions upon temperature and humidity data, these sensors are therefore required to read the local temperature and humidity of any given location.

#### 4.7.2. Design

The design for this block is fairly simple: find a device that can read in the desired data and output it in an ordered fashion.

We chose to use the SHT-30 sensor for this purpose. This device is cheap, efficient and well within margin of error. If purchased in the correct package, it even comes with a prefab water-proof packaging which will fulfill our project partner requirement of being waterproof. Figure 4.17 shows the black box diagram of the sensor for our system.

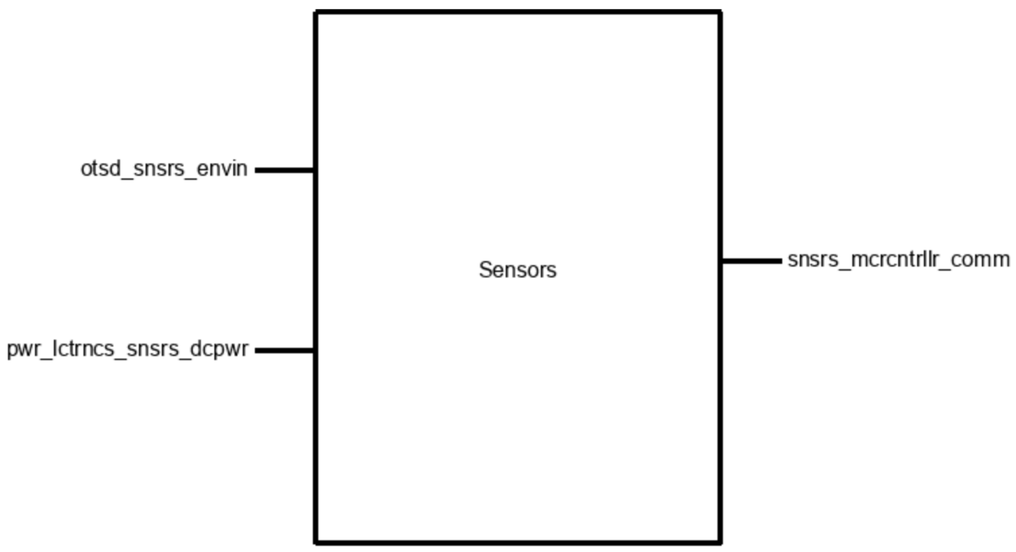


**Figure 4.17: Sensor Black Box Diagram**

### 4.7.3. General Validation

As stated above, this device is relatively cheap and efficient.

This device and package only costs $25. Compared to similar packages at $50 and up, this device saves the bank. This is important because we have a goal of keeping the system cost at under $300. We need to build a grand total of three of these nodes, with each requiring a sensor module. As such, cheaper is better [1].

The efficiency on this model was also an important factor in choosing this device. The sensor has a rated tolerance of ±0.5°C for temperature and ±2% for humidity. The next closest module was rated for ±1°C and ±5%. For a system where accurately pulling and operating on local temperature and humidity data is of utmost importance, having tight tolerances is a must. If we didn’t have these tight tolerances, we would be defeating the purpose of pulling local data in the first place [2].

### 4.7.4. Interface Validation

This section details the interface validations, two inputs and one output:

| Interface Property | Why is this interface this value? | Why do you know that your design details <u>for this block</u> above meet or exceed each property? |
|---|---|---|

**Table 4.22: otsd_snsrs_envin : Input**

| | | |
|---|---|---|
| Humidity: Space Dependent | Outside conditions have moisture in the air dependent on weather | The sensor will read this data value |
| Temperature (Absolute): Space Dependent | Outside conditions have temperature dependent on weather | The sensor will read this data value |

**Table 4.23: snsrs_mcrcntrllr_comm : Output**

| | | |
|---|---|---|
| Messages: Humidity | Microcontroller needs humidity data to send to the application | The sensor will send this value according to the datasheet |
| Messages: Temperature | Microcontroller needs temperature data to send to the application | The sensor will send this value according to the datasheet |
| Protocol: I2C | The sensor uses I2C to communicate with the microcontroller | The datasheet lists I2C as the sensors communication protocol |

**Table 4.24: pwr_lctrncs_snsrs_dcpwr : Input**

| | | |
|---|---|---|
| Inominal: 3mA | The is the average current drawn by the sensor according to the datasheet | The power electronics block has the ability to supply this current |
| Ipeak: 100mA | This is the maximum possible current drawn by a pin on the sensor | The block has the ability to draw this much current according to the datasheet |
| Vmax: 5V | Maximum rating according to the datasheet | The sensor block can handle this voltage |

| | | |
|---|---|---|
| Vmin: 3V | Minimum rating according to the datasheet | The sensor block can still run at this voltage |
| Vnominal: 3.3V | Average rating according to the datasheet | The linear regulator of the power supply will drop voltage to 3.3 for the sensor |

### 4.7.5. Verification Plan

This section details the verification plan for the sensor:

1. Set up power supply to run 3.3 V at 3mA to supply for the sensor
2. Confirm that microcontroller receives temperature and humidity data
3. Set power supply to 5V at 100 mA
4. Confirm that microcontroller receives temperature and humidity data
5. Set power supply to 3V
6. Confirm that microcontroller receives temperature and humidity data

### 4.7.6. References and File Links

[1] F. Waqar, "Sensor System for Self-Driven In-Home Climate Control (ECE)," *EECS Project Portal*, Sep-2022. [Online]. Available: https://eecs.engineering.oregonstate.edu/capstone/submission/pages/viewSingleProject.php?id=9vBa7B1brA8Osgxc.

[2] "Sht-30 mesh-protected weather-proof temperature/humidity sensor," *Adafruit Industries*. [Online]. Available: https://www.adafruit.com/product/4099?gclid=Cj0KCQiAjbagBhD3ARIsANRrqEvGKtVbXrLidh9HGwOH-cOX-GIw4mZLwPhFOvTOC5HB9hegqcKkrskaAgc4EALw_wcB

Datasheet File:
https://cdn-shop.adafruit.com/product-files/4099/C13024-002+datasheet.pdf

### 4.7.7. Revision Table

| Date Revised | Who? | Reason Revised/Revisions Made |
|---|---|---|
| 03/04/2023 | Jadon | Feedback Review |
| 01/20/2023 | Jadon | Initial Creation |

## 4.8. Computation

### 4.8.1. Description

The computation block will take in data from the indoor and outdoor temperature sensors to determine if windows need to be opened or closed. The user will be able to set a desired temperature using the application. If the indoor temperature rises above the desired temp and the outdoor temperature is below the indoor temp, the block will tell the GUI that windows need to be opened. Humidity will also be used in the computation block by integrating into a dew point calculation. This will be used to determine if the outdoor air is too moisture heavy compared to the indoor moisture. If the outdoor dew point is close to the indoor dew point, the windows can be opened if the temperatures are correct. The calculation block also determines if a notification needs to be sent using the previous window state. If the window state changes, the block will output the corresponding notification integer which is stored and then used by the notification block. The window status string is based on the window calculation and is used within the UI to clearly indicate the window status to the user.

### 4.8.2. Design

This section includes the black box diagram of the calculation block as well as a flow chart describing the code progression within the block. Figure 4.X, the black box diagram, shows that the block has one input, g_cmpttn_data, and one output, cmpttn_g_data. This is technically a bidirectional connection between the code block and the GUI. Input consists of float values which are: desired temperature, indoor and outdoor temperature, and indoor and outdoor humidity. The computation block outputs integer, string, and boolean data. The notification integer values determine what notification needs to be sent. The string is used to update the window status in the UI, and the boolean stores the state of the window for reference.

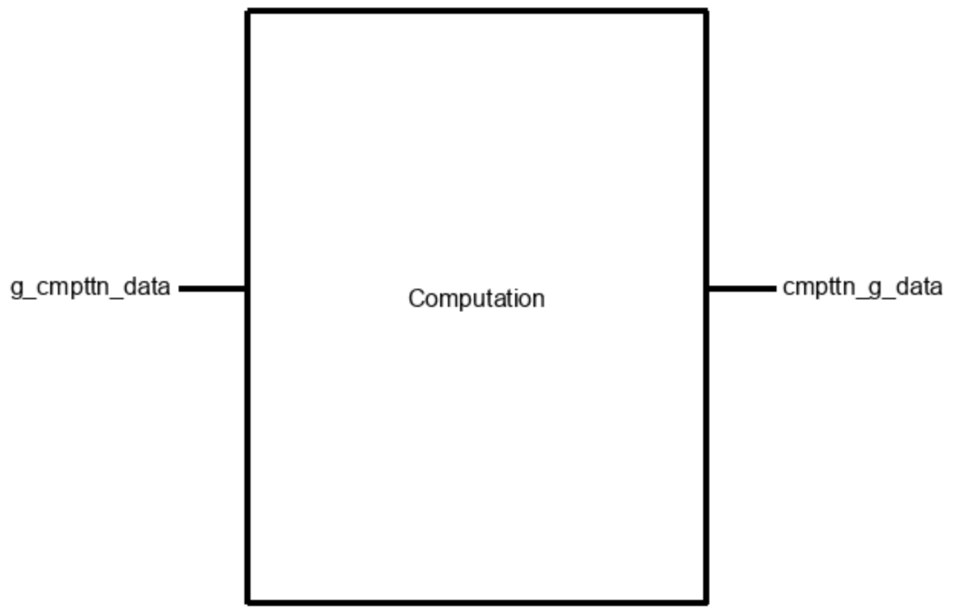


**Figure 4.18: Black Box Computation Diagram**

Figure 4.19 below showcases the flow chart describing the layout of the code block. At the top, fahrenheit temperature values are converted into celsius values, this allows the code to make a proper dew point calculation. The formula for calculating dew point is: (temp * 17.625)/(temp + 243.04) + log(hmd/100). Once this dew point is calculated for indoor and outdoor conditions, a simple if statement is run to determine if the windows need to be opened or closed. If the indoor temp is greater than 1.5 C of the desired temperature and outdoor temp is lower than indoor temp, the first condition is passed. If the outdoor dew point is lower than indoor dew point + 5 (meaning they are similar but outdoor not too large), then the second if passes and the windows need to be opened. From there, window status boolean and string is changed to open and the notification integer is updated (send if state changed, don't send if no change). These values are stored in the GUI database for further use by the application.

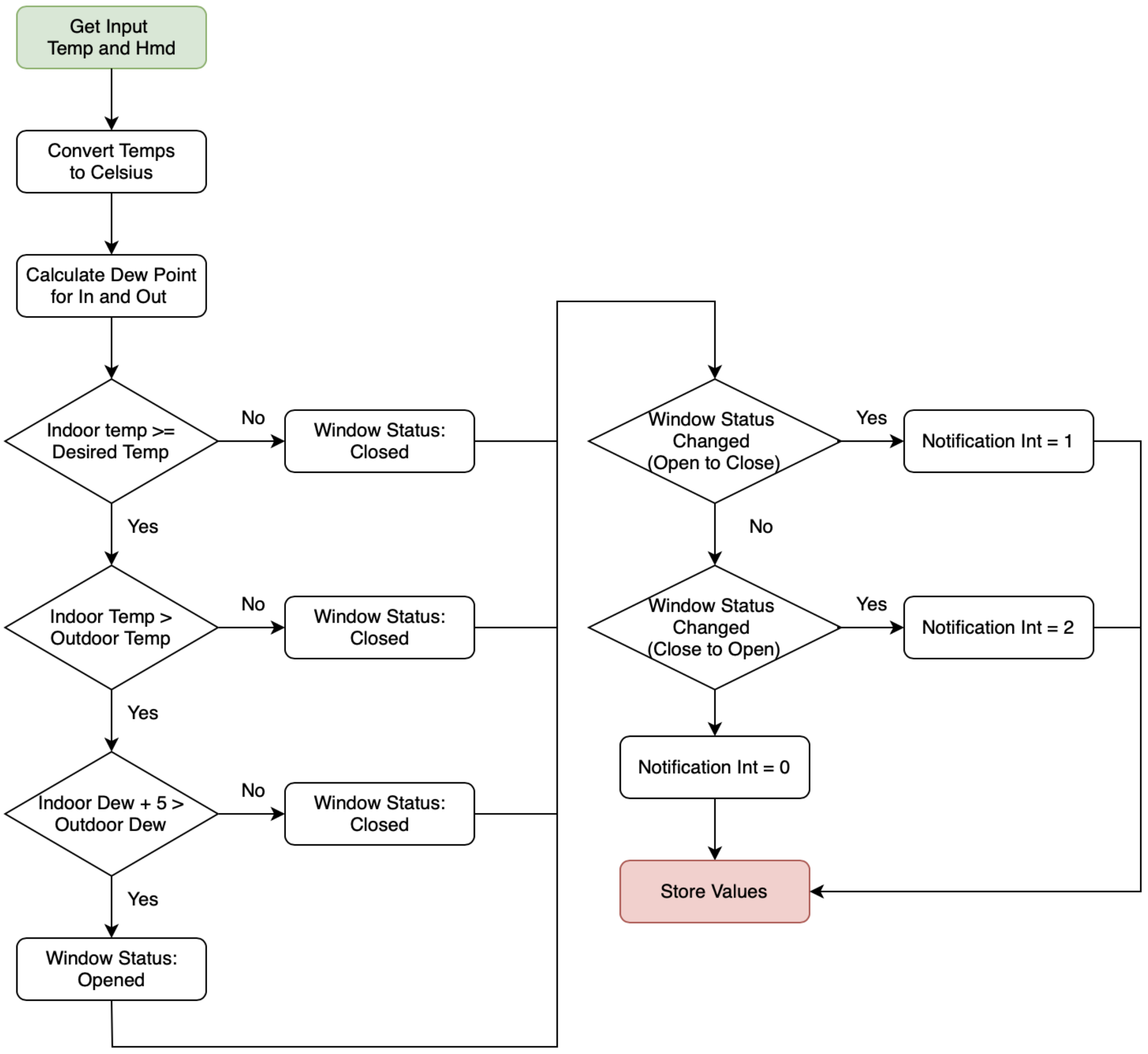


**Figure 4.19: Calculation Code Flow Chart**

### 4.8.3. General Validation

The calculation block is necessary for our project due to our project partner requirements and the nature of our project. The main goal of our system is to notify a user when their windows need to be opened or closed [1]. In order to accomplish this, we must have a code block that calculates when the windows must be opened based on the data provided by our sensors. The calculation block accomplishes this within the background thread of the application using data from the cloud database. By consistently updating the data in the application, we can make updated calculations to get the proper window status and statifly our requirement. Our project partners also wanted the user to be notified outside the GUI, which is why the calculation block includes a notification integer that is determined by the previous window state [1]. This integer is passed to the notification block for accurate and timely notifications for the user. The calculation block is an integral and necessary part of the system.

We decided to implement our calculations on the application due to the ease of data transfer and communication between the sensors. This allowed us to use more complex calculations, like dew point calculations, in this block. Dew point is a much better metric than relative humidity when looking at the moisture in the air [2]. We did not want a user to open their windows when the moisture content was too high outside, therefore we used this calculation to compare indoor and outdoor moisture. If the moisture levels were similar, windows can be opened without creating dampness in the user’ s house.

We could also have implemented this on one of the esp32 sensors, however this would have added needless complicated networking for this simple of a project and may have slowed down the results of the calculations. It also would not have allowed users to get notification when they were away from the house.

### 4.8.4. Interface Validation

This section contains the two interfaces and their properties. The tables below talk about each property value and why the design details meet each property.

| **Interface Property** | **Why is this interface this value?** | **Why do you know that your design details <u>for this block</u> above meet or exceed each property?** |
|---|---|---|

**Table 4.25: g_cmpttn_data : Input**

| | | |
|---|---|---|
| Messages: outdoor hmd (double) ex. 75.8 | Outdoor humidity is a double in order to calculate the most accurate information for the user | When performing calculations in Java, the numbers involved must be double or float values or else the final result may be truncated, giving inaccurate readings. |
| Messages: indoor temp (double) ex. 75.2 | Indoor temperature is a double in order to calculate the most accurate information for the user | When performing calculations in Java, the numbers involved must be double or float values or else the final result may be truncated, giving inaccurate readings. |
| Messages: desired temp (float) ex. 71 | Desired temperature is this value to make it easy for the user to | The block takes in the desired temperature to check if the interior |

|  |  |  |
|---|---|---|
|  | enter the value. The value is a float to aid in future calculations. | temperature gets too high. The design for the block utilizes the float to perform accurate calculations. |
| Messages: outdoor temp (double) ex. 68.9 | Outdoor temperature is a double in order to calculate the most accurate information for the user | When performing calculations in Java, the numbers involved must be double or float values or else the final result may be truncated, giving inaccurate readings. |
| Messages: indoor hmd (double) ex. 57.3 | Indoor humidity is a double in order to calculate the most accurate information for the user | When performing calculations in Java, the numbers involved must be double or float values or else the final result may be truncated, giving inaccurate readings. |

**Table 4.26: cmpttn_g_data : Output**

|  |  |  |
|---|---|---|
| Messages: notification type (int) ex. 0 (0 is none, 1 is open window, 2 is close window) | The notification type integer is used to determine the notification sent to the user. An int is used since three states are needed for window state: open, close, or no change. | The design of the block sets the integer to 0 unless a state is changed, therefore it meets the expectations for this property. |
| Messages: window status (boolean) ex. false (true is open, false is closed) | The window status boolean is used since the current window status just needs two states, opened or closed. | The design of this block sets the boolean true if all the sensor specifications are met or defaults to false. The design meets expectations for the window status of the system. |
| Messages: window string (String) ex. "Closed" | The window string changes along with the window status boolean, it works the same way but allows the UI to easily update. | The design of this block for this property is the same as the boolean, therefore it continues to meet expectations. |

#### 4.8.5. Verification Plan

This section details the verification plan for the computation block. The plan will go through the process of verifying that the input interfaces produce the output interfaces.

1. Open Android studio and build the project. Once the project has been built, open the background worker class and scroll to the windowCalculation() function.

2. Call function in the worker class and enter dummy data for indoor and outdoor temperature and humidity into the function call (data: 75, 45, 68, 80).

3. Open application on device. Tap on the window status text box and enter the desired temperature in the pop-up window.

4. Refresh activity to run background worker class. Look at the log in android studio to verify that each of the three outputs are printed to the log (window status is “false”, string is “closed” and int = 0)

5. Change the dummy data in the function call (usually just change outdoor humidity = 50) and open the main page again.

6. Look at android studio logs and verify output changed accordingly (window status now “true”, string is “open” and int = 1)

### 4.8.6. References and File Links

[1] F. Waqar, “Sensor System for Self-Driven In-Home Climate Control (ECE),” *EECS Project Portal*, Sep-2022. [Online]. Available: https://eecs.engineering.oregonstate.edu/capstone/submission/pages/viewSingleProject.php?id=9vBa7B1brA8Osgxc.

[2] “Dew Point vs Humidity,” *NOAA: US Dept of Commerce*, 26-Jan-2021. [Online]. Available: https://www.weather.gov/arx/why_dewpoint_vs_humidity

### 4.8.7. Revision Table

| **Date Revised** | **Who?** | **Reason Revised/Revisions Made** |
|---|---|---|
| | | |
| 01/20/2023 | Blake | Initial Creation |

## 4.9. Database

### 4.9.1. Description

The database block allows our system to collect, store, and display sensor data over a longer time period. Temperature and humidity data from our sensor comes into the block, where it is packaged by the microcontroller using relevant tags and identifying information, such as username and node name (indoor/outdoor). This packaged data is then sent to our time series cloud database, Influx DB, over WIFI where data is stored with a timestamp the moment it arrives. I chose Influx DB due to its wide range of applications and code examples for both Arduino (ESP32) and Java (Android application) as well as the automatic timestamp collection component, which is important for our project.

In the database, different accounts (usernames) will be tagged within a single bucket of data and the sensor name will be tagged underneath the user for further organization. Once the data is stored, our Android application will query the database using these tags to collect temperature and humidity data as well as the timestamp from specific sensors. This data will be used in our GUI to create time graphs for all the sensors attached to a user. One of our project partner requirements was the ability to visualize sensor data in our application, and the database block allow the creation of this time-based graph with time on the x-axis and temp/hmd data on the y-axis. We will also use the temperature and humidity values in the computation block to determine when the windows need to be opened or closed. This was another major requirement of the system (window notifications) and the database block helps accomplish this by gathering and sending critical data for the calculations. Finally, one of our system requirements is the ability to show current temperature data, and the database block is needed to update the user interface with the most updated and last time stamped data point.

### 4.9.2. Design

This section includes the black box diagram of the database block as well as the flow chart of the code needed to execute both the data upload and to query the sensor data. Figure 4.20, the black box diagram below, shows that the block has one input, mcrcntrllr_dtbs_rf, and one output, dtbs_g_rf. The input to the database block comes from the ESP32 microcontroller sensor code. Its form is floating point values that store temperature and humidity data and tag strings that store node names. For example, mcrcntrllr_dtbs_rf will contain temp = 72.1, hmd = 80.6, measurement = values, and tag = outdoor. The output from the database, also seen in Figure 4.20, goes to the GUI block. Its form is also floating point values for temperature and humidity (same as input) along with the same measurement and tag values in the form of strings. The output differs from the input in that a new timestamp element is included in the form a Date object, which is the standard time element in Java.

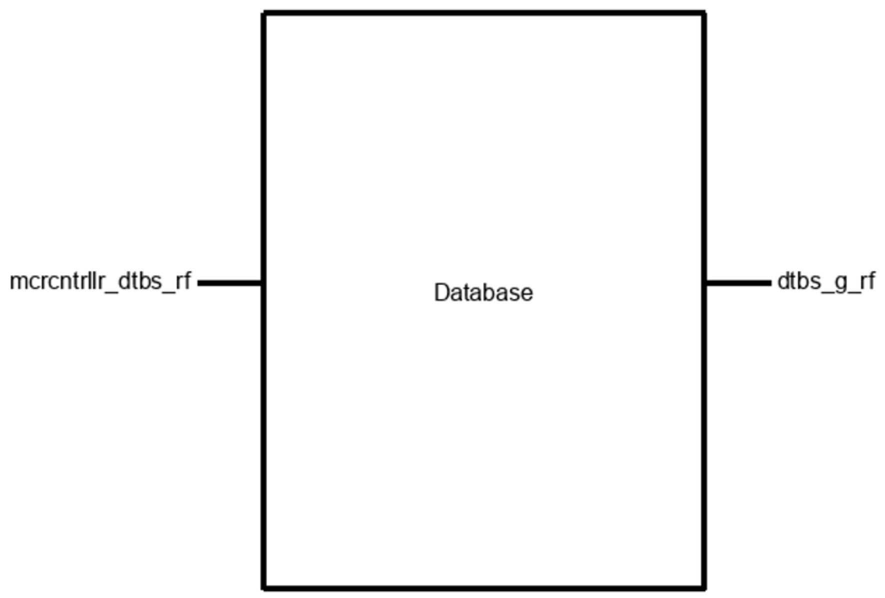
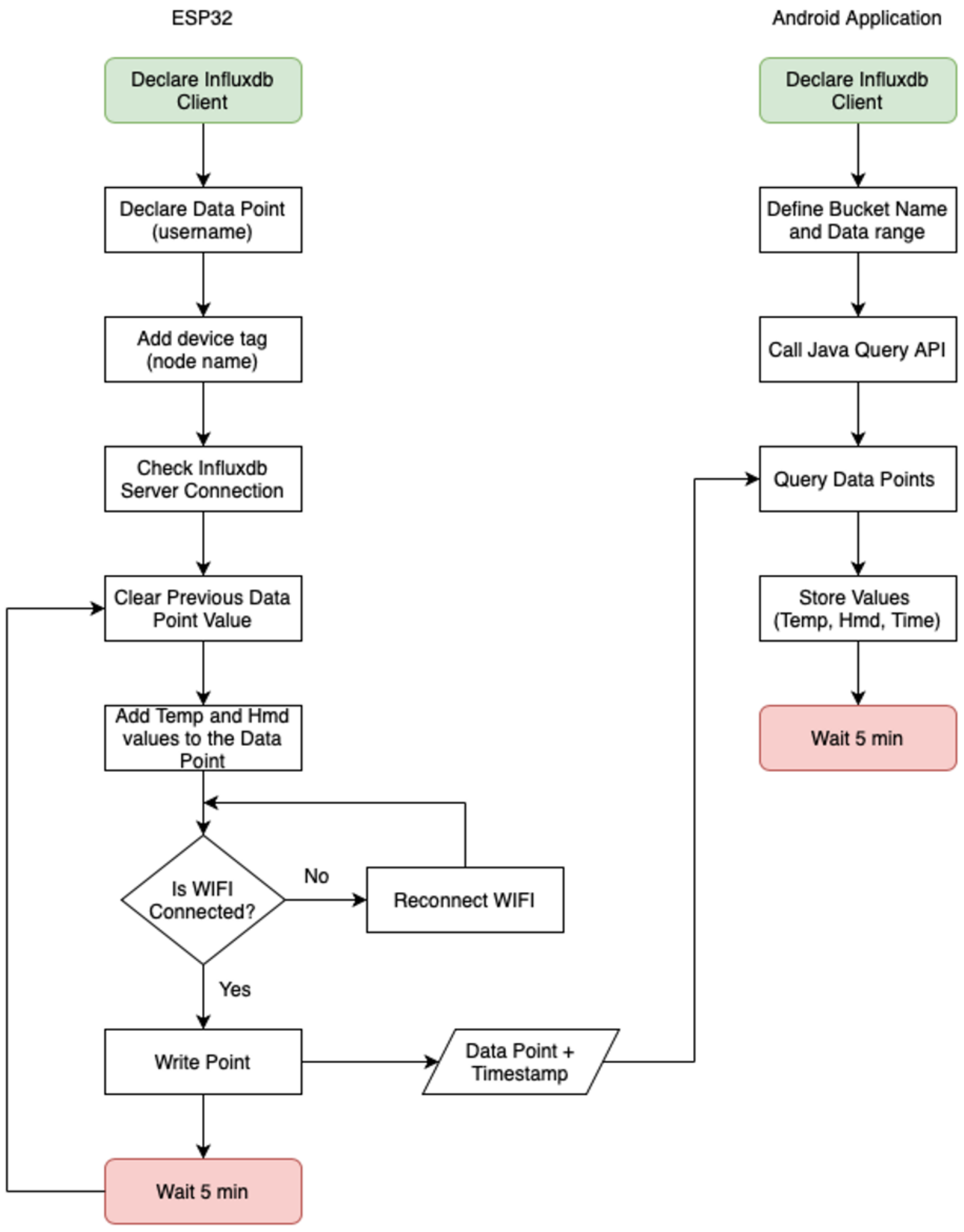


**Figure 4.20: Black Box of Database Block**

Figure 2 showcases the interior flow of the data from the ESP32 to the application. There are two separate components to the database block: data upload and data query. The data upload section happens on the ESP32. Once it enters the block, the code will add the input interface properties to an Influx dB data point, which is a class built for the Arduino IDE. The data point contains a measurement, fields for temp and hmd, and a tag for the sensor location. After that, an Influx DB client will be created using the cloud storage URL, our account code, and a token for certification. After WIFI and server connections are rechecked and possibly reestablished, the point is written to the database using the Influx DB client. The upload loop will run every 5 min to ensure the most updated data while also saving battery.

From this point, Figure 4.21 shows that a timestamp in the form Year-Date-Time is added to the datapoint. The cloud database will store the data for up to 30 days. Getting data from the server is called a query. The right half of Figure 2 shows how our android application will query the data. On the app, a new Influx DB client is created using the same information on the ESP32 (URL, account, token, bucket name). Then, the Influx DB Query Java API uses a series of information, such as bucket name (which is our case is “Sensor Data”), date range (-1 day), and measurement (or account name) to query the data using a structure called flux. The query creates a flux table, which is used is organize the data that is queried. Values in each table row are pulled out and used for various UI elements within the application. Items like current temperature and humidity data are used to update the internal databases which control UI elements on the main screen, and timestamps with temp/hmd values are used to create graphs on the info pages. This query process will happen in a background thread and continually update the UI elements every 5 minutes, or when the page is resumed to ensure the user always sees the most updated data.



ESP32
Declare Influxdb Client
Declare Data Point (username)
Add device tag (node name)
Check Influxdb Server Connection
Clear Previous Data Point Value
Add Temp and Hmd values to the Data Point
Is WIFI Connected?
No
Reconnect WIFI
Yes
Write Point
Data Point + Timestamp
Wait 5 min
Android Application
Declare Influxdb Client
Define Bucket Name and Data range
Call Java Query API
Query Data Points
Store Values (Temp, Hmd, Time)
Wait 5 min

**Figure 4.21: Arduino + Application Code Flow Chart**

### 4.9.3. General Validation

The general validation section describes why the design from above fits the needs of the system. Needing both a hardware sensor and an application for user interaction, having a cloud database for data storage is the best solution and helps with the connection between the two subsystems (as shown in Figure 2 from the design).

The database block is an important part of the climate control sensor system. One of the requirements of the system is to visualize sensor data within a graphical user interface. Our project partners want graphical data visualization to help the users of our system see the temperature and humidity fluctuations outside and within their house [1]. This will help them better understand the impact of opening and closing their windows. Design decisions that allow this to happen is the use of a time series database. Time data is necessary to create accurate graphs in our application, and Influx DB automatically timestamps data points and is accurate to the second [2]. I was also very familiar with Influx DB writing and queries since I used it for many data applications during my summer internship.

Another requirement of the system is the ability to allow users to create accounts. Having a cloud-based storage that connects with both the sensors and the application allows the system to have organized data locations for individual users. Local storage could have been an option, but all the data would have been locally accessed (home WIFI only) and there would be no need for accounts. System performance is also much better with a cloud-based storage system since data can be accessed from anywhere, allowing users to be notified outside of their house [3]. This cloud storage foundation could potentially allow automatic window opening and closing in future iterations of the system.

In terms of the reasons for the design of the block, we made a choice to build an android application because of the availability, cost, and ease of the use of the platform. ESP32 modules have built-in wireless connections, which make it simple to connect to our database. Influx DB is also a free service with significant amounts of documentation for both Arduino and Java. Splitting the block into two sections, write and query, made implementing the database much easier. We also didn't need the application to write anything to the database or the ESP32 to query data, so those code block are not included.

One alternate solution for this block would be to cut out the cloud-based storage solution and directly send temperature and humidity data to the application on the local network. The system could use the WIFI to send JSON packets with temp, hmd, and name data to the application. The application could then store this data on the device for a few days and update the UI elements as needed. As discussed above, this solution has future limitations, but it is viable.

### 4.9.4. Interface Validation

This section details the interface properties for both the input and the output. As discussed in previous sections, the system requires data storage and transfer to complete system requirements like data visualization and updated information in the GUI. Properties like temperature and humidity data use a float format to get the most accurate and easy to read numbers, while other messages like measurements and tags are string objects to find data relating to the correct sensor subsystem. Both input and output interfaces use Influx DB libraries to create and query points in each language, and the time stamp is outputted as a Date object to make time graphs easier to create on the application.

**Table 4.27. dtbs_g_rf : Output**

| **Interface Property** | **Why is this interface this value?** | **Why do you know that your design details <u>for this block</u> above meet or exceed each property?** |
|---|---|---|
| Messages: temperature (float) ex. 72.1 | The temperature and humidity data were chosen to be float based for our sensor accuracy. Rounding to the tenths place helps users to visualize the data easily within our user interface. We also want the data to be easily to graph and visualize, and this form helps us achieve that goal. | This data format is easiest to write and query from the database. While the database can handle all forms of data, numbers with only a few decimal places are best. |
| Messages: tag (string) ex. user1, indoor | Tags for the data were chosen because we needed to separate data by sensor and by account. For Influx DB, tags are normally strings for ease of use when querying the output. | This meets the standards for Influx DB, tags must be string data. I also know it is easiest to use tags to display cleanly to the user. |
| Messages: time stamp ex. 2023-1-12T20:36:45 | The time stamp was chosen for the output due to the requirement to visualize data in our application. This format is what comes standard with Influx DB and can be easily put into a Date object for graph creation. | Our design specifies time as a key data value for our GUI. Using this interface value from the time series database is best for our purposes of data visualization and using time as a component in our window open and close calculations. |
| Messages: humidity (float) ex. 80.5 | The humidity data were chosen to be float based for our sensor accuracy. Rounding to the tenths place helps users to visualize the data easily within our user interface. We also want the data to be easily | This data format is easiest to write and query from the database. While the database can handle all forms of data, numbers with only a few decimal places are best. |

| | | |
|---|---|---|
| | to graph and visualize, and this form helps us achieve that goal. | |

**Table 4.28. mcrcntrllr_dtbs_rf : Input**

| | | |
|---|---|---|
| Messages: temperature (double) ex. 72.1, humidity (double) ex. 80.5 | The temperature and humidity data were chosen to be a double based on our sensor accuracy. Rounding to the tenths place helps users to visualize the data easily within our user interface. We also want the data to be easily to graph and visualize, and this form helps us achieve that goal. | Based on past experiences with Influx DB, I know that this data format is easiest to write and query from the database. While the database can handle all forms of data, numbers with only a few decimal places are best. |
| Messages: tag (string) ex. user1, indoor | Tags for the data were chosen because we needed to separate data by sensor and by account. For Influx DB, tags are normally strings for ease of use when querying the output. | This meets the standards for Influx DB, tags must be string data. I also know it is easiest to use tags over changing a data field or adding it to the end of the sensor data. |
| Protocol: Influxdb Client Library (Arduino) | The ESP32 uses the Arduino IDE for compile and code upload. The Influx DB client library is part of the Arduino system and is necessary for creating clients and writing data points in our code. | The Influx DB Client Library on GitHub gives example on how to create data points and write data. I chose to implement this database because it uses time stamps and I worked with it previously [5]. |

### 4.9.5. Verification Plan

This section details the verification plan for the database block. The verification plan will go through the process of making sure each input interface property makes the corresponding output interface property. For example, if the temperature is 72.1 at the input, the output must also show 72.1 as the temperature. This is important for accurate UI data within our application.

**ESP32 Verification**
1. Power up ESP32 using USB power. Confirm that Influx Client information (URL, token, account code, bucket) are correct.

2. Name the measurement (point) “values” to simulate account information. Name tag “Indoor” to simulate sensor name.
3. Create float values for temperature and humidity. Write a code block within the loop that increases both data values every minute. This will simulate float data coming from the sensor and show that changing data on the input will change the output.
4. Compile and run ESP32 code. Let the microcontroller write pseudo sensor data to Influx DB for at least 5 minutes. This allows the database to populate with values and time stamps that are graphable.
5. Log into Influx DB cloud account. Find the “Sensor Data” bucket and use an online query to confirm that data has been uploaded in the proper location.

**Android Application Verification**
6. Open Android studio and check that Influx Client Data (URL, token, account code, bucket) have the correct values that match the values on the ESP32 in both the Worker class (background data query on a timer) and the node info activity (where the graphs are located).
7. Plug in a test device to the computer. Build and run the application on the device.
8. Look at data points to see if they match the input data in the system log. More specifically, check that time values are accurate for the current time (UTC), the sensor name matches the input sensor name, and temperature and humidity match the input values.
9. Check the temperature and humidity graphs in the info node activity properly populated with the data being queried from the database. Check that the most recent temperature and humidity data are show on the main screen.

### 4.9.6. References and File Links

[1] F. Waqar, “Sensor System for Self-Driven In-Home Climate Control (ECE),” *EECS Project Portal*, Sep-2022. [Online]. Available: https://eecs.engineering.oregonstate.edu/capstone/submission/pages/viewSingleProject.php?id=9vBa7B1brA8Osgxc.

[2] “Time Series Database,” *Hazelcast*. [Online]. Available: https://hazelcast.com/glossary/time-series-database/.

[3] S. Naseem, “Here's what really matters in cloud vs local storage.,” *Communication Square LLC*, 10-Nov-2022. [Online]. Available: https://www.communicationsquare.com/news/cloud-vs-local-storage/.

[4] Influxdata, “Influxdata/influxdb-client-java: Influxdb 2 JVM based clients,” *GitHub*, 03-Nov-2022. [Online]. Available: https://github.com/influxdata/influxdb-client-java.

[5] T. Schuerg, “influxdb-client-for-arduino,” *GitHub*, 14-Oct-2022. [Online]. Available: https://github.com/tobiasschuerg/InfluxDB-Client-for-Arduino.

### 4.9.7. Revision Table

| Date Revised | Who? | Reason Revised/Revisions Made |
|---|---|---|
| 02/08/2023 | Blake | Updated Description with relevant system requirements, added specific figure descriptions to Design section, added citation to General Validation, added table labels, added intro paragraph to sections 3-5, updated verification to include UI element checking. |
| 01/20/2023 | Blake | Initial Creation |

## 5. System Verification Evidence

### 5.1. Universal Constraints

#### 5.1.1. The system may not include a breadboard

Our system meets the breadboard requirement since it does not contain a breadboard. Our system only includes a battery, PCB, sensor and switches as our main hardware and everything is connected using the PCB.

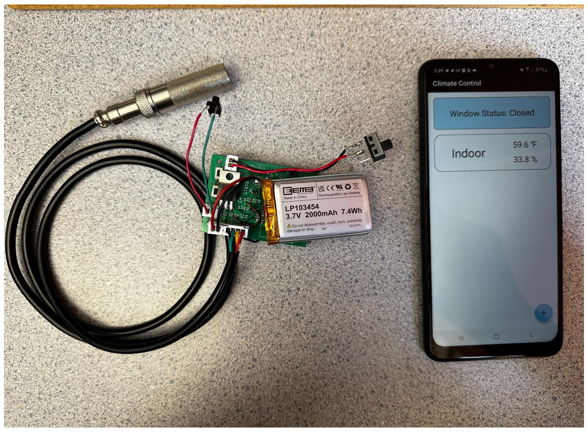


**Figure 5.1: Complete System (shows no breadboard)**

#### 5.1.2. The final system must contain a student designed PCB.

Our system meets the student designed PCB requirement. Our sensor PCB was designed by Jaden and includes our ESP32 and battery power circuits as well as connectors to our sensors, battery and buttons. It contains 214 smd pads and sits inside our enclosure as the main hardware component of our system.

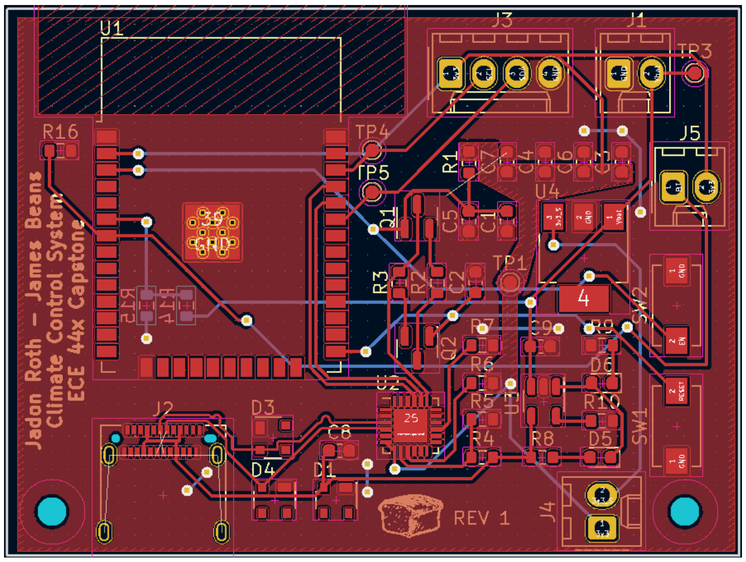


**Figure 5.2: PCB Design Schematic**

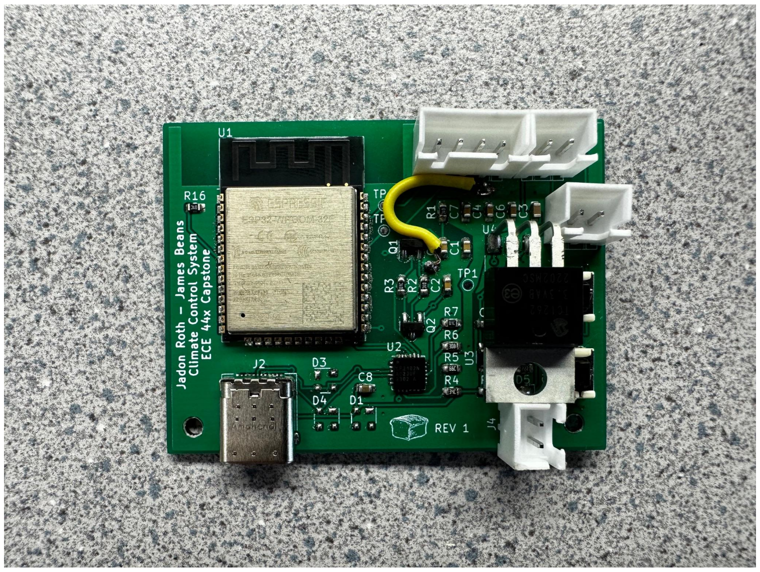


**Figure 5.3: Fully Assembled PCB**

## 5.1.3. All connections to PCBs must use connectors.

Our system meets the connector requirement. Our PCB uses connectors to power and send data from our sensors, battery, and button peripherals. The image below showcases our finished PCB with all our components connected to it. Additional wires were used to correct errors and were not part of the original design.

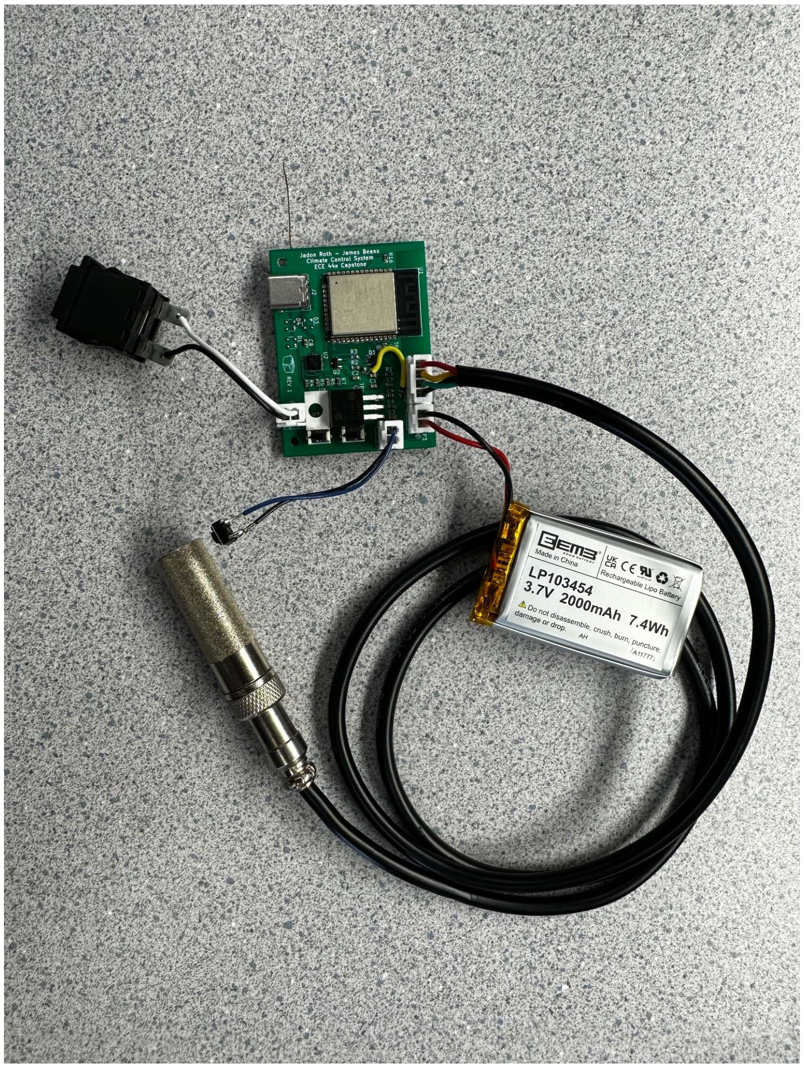

Made in China
Rechargeable Lipo Battery
LP103454
3.7V 2000mAh 7.4Wh
Do not disassemble, crush, burn, puncture, damage or drop.

**Figure 5.4: PCB with connectors**

### 5.1.4. The system may be no more than 50% built from purchased 'modules.'

**Table 5.1: Modules**

| Block | Built/Bought |
|---|---|

| | |
|---|---|
| GUI | Built |
| Notifications | Built |
| Enclosure | Built |
| Power Electronics | Built |
| Microcontroller | Bought |
| Microcontroller Code | Built |
| Sensors | Bought |
| Computation | Built |
| Database | Built |

**Total Percentage of Blocks Built: 77.8%**

#### 5.1.5. All power supplies in the system must be at least 65% efficient.

We use an LDO to power all the components in our system. The battery provides 3.7 volts, our LDO drops that down to 3.3 V. From the datasheet, the power loss equation provided is Pd = (Vin_max - Vout_max) * Iout. Calculations are shown below:

Vin_max = 3.7V

Vout_max = 3.3V

Iout = 500mA

Pd = (3.7V - 3.3V) *0.5 A = 0.2 W

Pout = 3.3V*0.5A = 1.65 W

Pin = Pout + Pd = 1.65W + 0.2W = 1.85W

Efficiency = Pout/Pin = 1.65/1.85 = 0.8918 = **89.18%**

### 5.2. Requirements

#### 5.2.1. Battery

5.2.1.1. **Project Partner Requirement:** Rechargeable battery
5.2.1.2. **Engineering Requirement:** The system will operate for at least 1 week on a single charge.
5.2.1.3. **Testing Method:** Test

5.2.1.4. **Verification Process:**
1) Detached from wall power 2) Put DMM inline with battery 3) turn on device 4) Find average amount of current battery is drawing 5) Use to calculate how many hours device will run on single charge
**Pass Condition:** Device will power for 7 or more days (168 hrs)

5.2.1.5. **Testing Evidence:** Average battery draw = 11.27mA (see image below). 2000mAh/11.27mA = **177.46 hrs > 168 hrs**. Therefore the device will run for 1 week on a single charge.

Video Link: https://youtu.be/gqC4mDICtEo (Verified 05/09/2023)

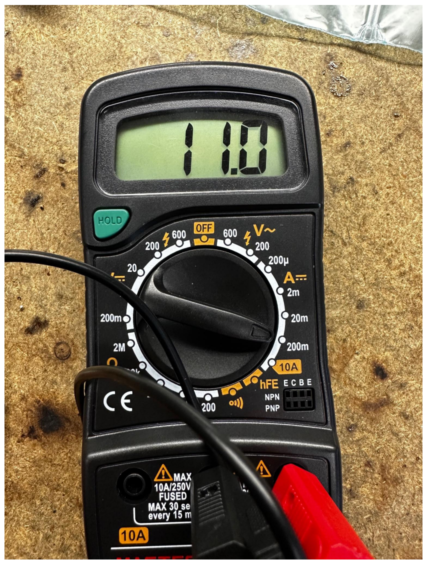


**Figure 5.5: Battery Test Evidence (Verified 05/05/2023)**

### 5.2.2. Device Labeling

5.2.2.1. **Project Partner Requirement:** Ability to mark a certain module as indoor or outdoor

5.2.2.2. **Engineering Requirement:** The system will have the ability to name device subsystems within the application.

5.2.2.3. **Testing Method:** Inspection

5.2.2.4. **Verification Process:**
1) Open the application and add device 2) Name device 3) Check that device name is correct
**Pass Condition:** Device subsystems are correctly labeled in the application

5.2.2.5. **Testing Evidence:**

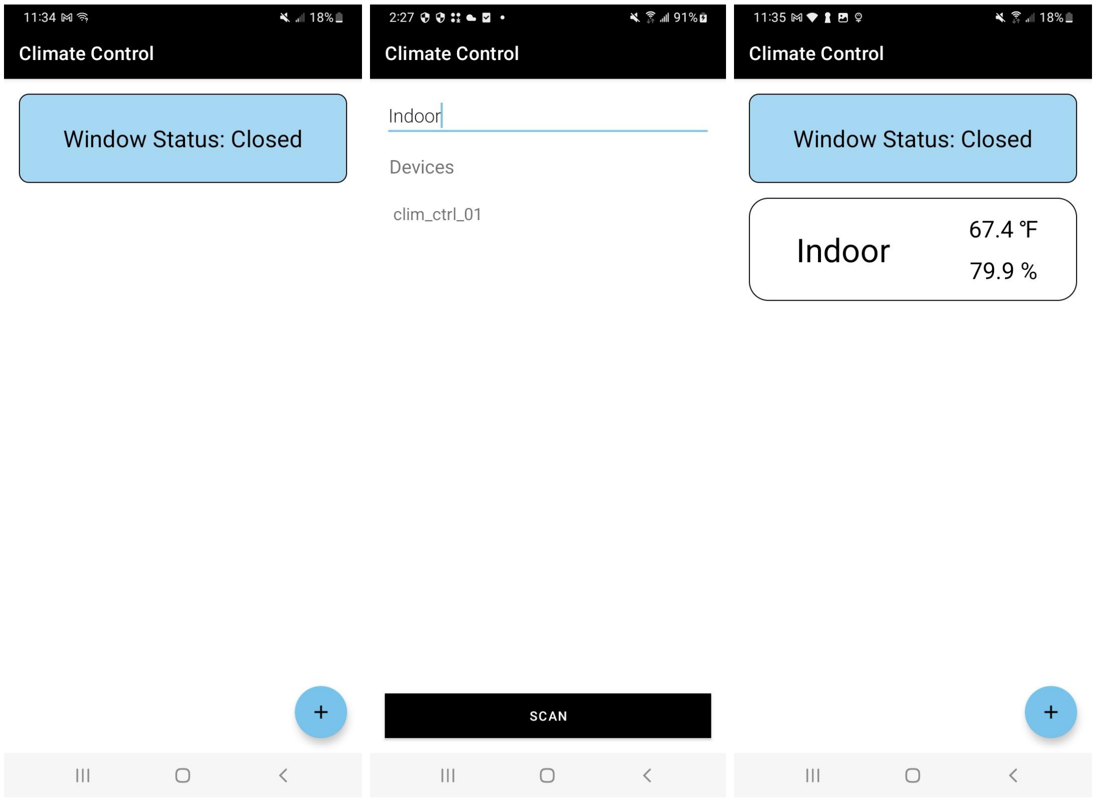


**Figure 5.6: Device Labeling Screen Shots (Verified 05/05/2023)**

### 5.2.3. Device Size

**5.2.3.1. Project Partner Requirement:** Module and Housing must fit within a 3”x3”x1” space

**5.2.3.2. Engineering Requirement:** The system device subsystem will be no larger than 3x3x1 inches of space.

**5.2.3.3. Testing Method:** Inspection

**5.2.3.4. Verification Process:**
Use a ruler to measure each side and check dimensions are smaller than 3x3x1 inch.
**Pass Condition:** Device size within listed specifications

**5.2.3.5. Testing Evidence:**

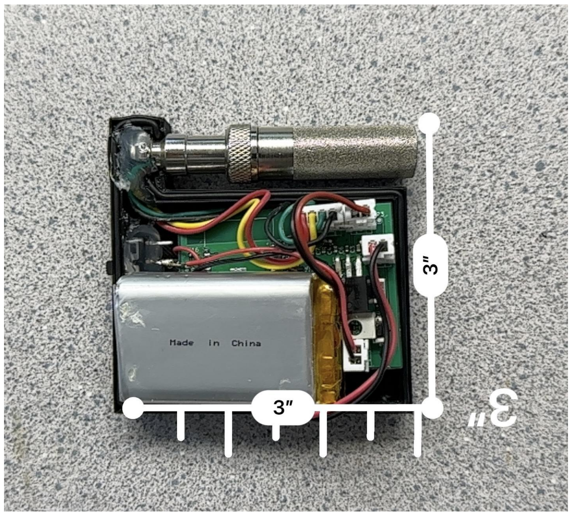


**Figure 5.7: Device Size Evidence (05/11/23)**

### 5.2.4. GUI

**5.2.4.1. Project Partner Requirement:** GUI interface

**5.2.4.2. Engineering Requirement:** The system will show the current temperature data and at least 9 out of 10 users report they can see the current temperature easily.

**5.2.4.3. Testing Method:** Demonstration

**5.2.4.4. Verification Process:**
1) Take a screenshot of the working home screen 2) Create Google form asking the question "Is the temperature provided by the GUI easily readable?" 3) Confirm that 9 out of 10 users answer yes
**Pass Condition:** 90% of users answer yes to the question "Is the temperature provided by the GUI easily readable?"

**5.2.4.5. Testing Evidence:**

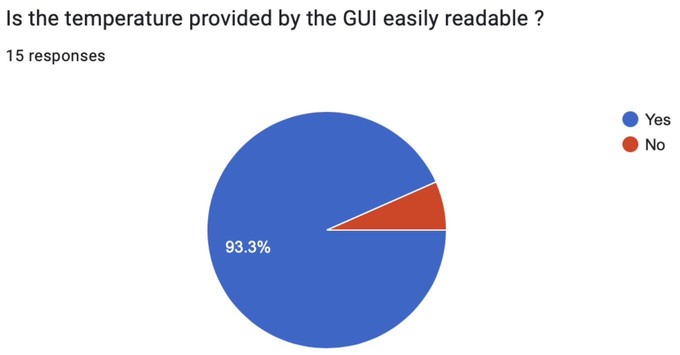


**Figure 5.8: GUI Form Pie Chart**

Video Link: https://youtu.be/EU-kbv03sAU (Verified 05/07/2023)

Form Link:
https://docs.google.com/forms/d/e/1FAIpQLSdVVWecX0Qr9RtCPfkTS1m3DCHSPrjn0bqIk7gHzpNzbKKWYQ/viewform?usp=sf_link

### 5.2.5. Notifications

**5.2.5.1.** **Project Partner Requirement:** Notifies user when to open or close windows

**5.2.5.2.** **Engineering Requirement:** The system will notify users of requested change in window status in a way that at least 9 out of 10 of users report was easy to understand.

**5.2.5.3.** **Testing Method:** Demonstration

**5.2.5.4.** **Verification Process:**
1) Take a screenshot of open and closed notifications 2) Create a Google form with a screenshot that asks "Are the window status notifications easy to understand?" 3) Confirm that 9 out of 10 users answer yes
**Pass Condition:** 90% of users answer yes to the question "Are the window status notifications easy to understand?"

**5.2.5.5.** **Testing Evidence:**

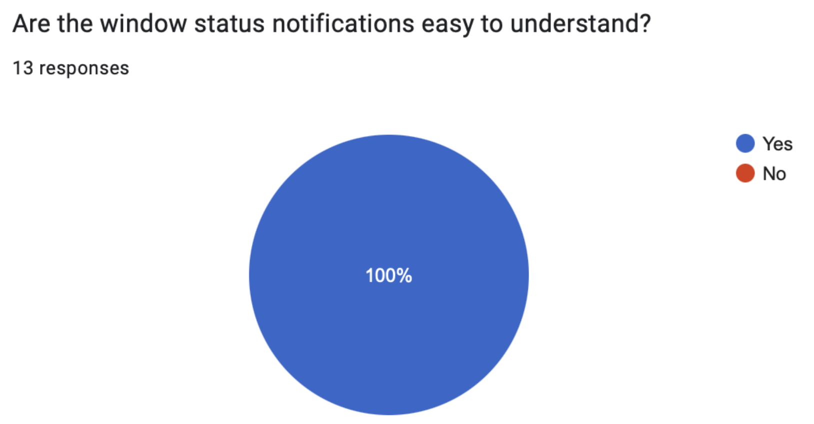


**Figure 5.9: Notification Form Pie Chart**

Video Link: https://youtu.be/jl7MP--Ijcw (Verified 05/07/2023)

Form Link:
https://docs.google.com/forms/d/e/1FAIpQLSdGiHRDYi3s8qDJfXtWrIp0X4u4RLiRaR2uts7qFV_po4u08g/viewform?usp=sf_link

### 5.2.6. Sensors

**5.2.6.1.** **Project Partner Requirement:** Temperature and humidity sensors

**5.2.6.2.** **Engineering Requirement:** The system will gather temperature and humidity data that is accurate within 5 degrees Celsius and 10% humidity.

**5.2.6.3.** **Testing Method:** Test

**5.2.6.4.** **Verification Process:**
1) Get an already accurate device and measure current temperature and humidity values 2) Compare values gathered by the system to these collected numbers

**5.2.6.5.** **Pass Condition:** System values are within 5 degrees C and 10% humidity of collected values

**5.2.6.6.** **Testing Evidence:** Values read from accurate device (Kestrel): 74.9 ℉ and 50.5 % humidity. Values read from our system: 73.9 ℉ and 45.9 % humidity.

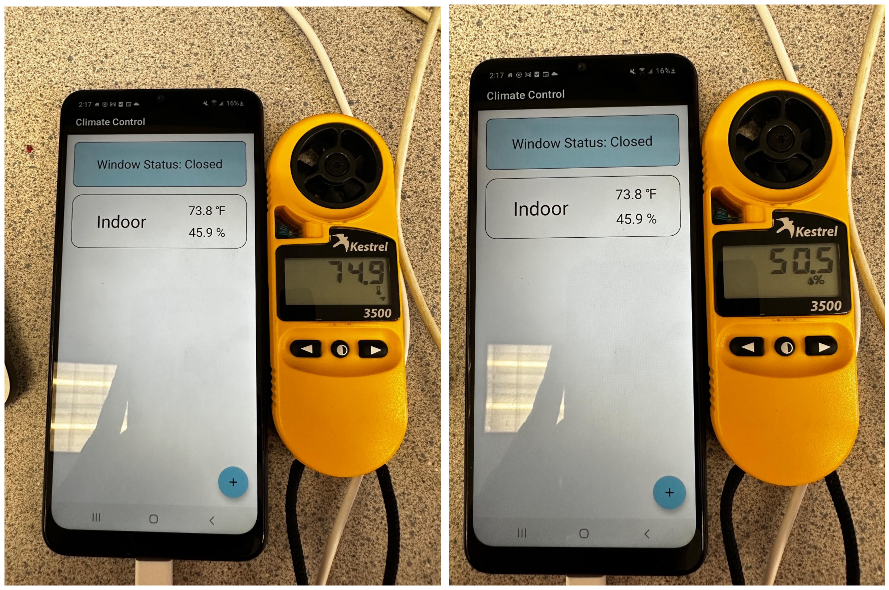


**Figure 5.10: Sensor Accuracy Evidence (05/10/2023)**

### 5.2.7. Weather Proof Enclosure

**5.2.7.1.** **Project Partner Requirement:** Weatherproof (IPX4) container

**5.2.7.2.** **Engineering Requirement:** The system device subsystem will be weatherproof to an IPX4 water resistance (water splashes from any direction).

**5.2.7.3.** **Testing Method:** Test

**5.2.7.4. Verification Process:**
1) Splash water on each device subsystem from all directions 2) Check that no water is inside the enclosure
**Pass Condition:** No water is inside the enclosure

**5.2.7.5. Testing Evidence:**

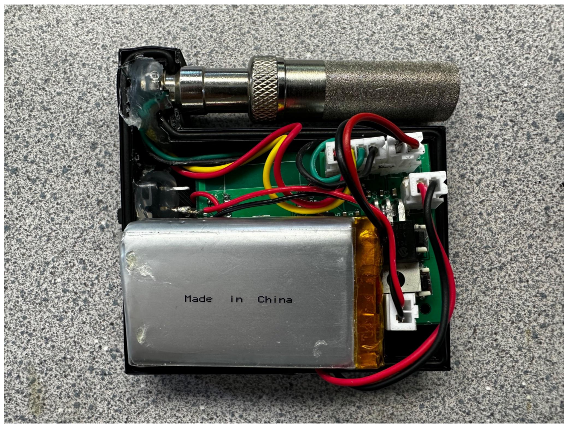


**Figure 5.11: Inside Enclousure (05/11/2023)**

### 5.2.8. Wireless

**5.2.8.1. Project Partner Requirement:** Wireless communication to transmit data

**5.2.8.2. Engineering Requirement:** The system device subsystem will pair with the application subsystem wirelessly

**5.2.8.3. Testing Method:** Test

**5.2.8.4. Verification Process:**
1) Turn on device subsystem 2) Open application and start pairing process 3) Check the process completes wirelessly
**Pass Condition:** Device subsystem temperature and humidity data is viewed within the application subsystem

**5.2.8.5. Testing Evidence:**

Video Link: [https://youtu.be/oUATXdjgrNo](https://youtu.be/oUATXdjgrNo) (Verified 05/09/2023)

## 5.3. References and File Links

### 5.3.1. References (IEEE)

### 5.3.2. File Links

### 5.4. Revision Table

| Date Revised | Who? | Reason Revised/Revisions Made |
|---|---|---|
| 05/07/23 | Blake | Added Testing Evidence to Battery/GUI/Notifications |
| 03/12/23 | Blake | Initial Creation |

# 6. <u>Project Closing</u>

## 6.1. Future Recommendations

This section details the items we are recommending a future team working on the project to complete and consider. It will look at three different types of recommendations: technical recommendations, global impact recommendations, and teamwork recommendations.

### 6.1.1. Technical Recommendations

One recommendation we would make is a more efficient PCB design. Currently, the PCB has a total width of about 2’’ by 1.5’’. This could be reduced further with a design that utilized both sides of the board, or even a three layer PCB [1]. In addition, quite a few of the routing options, while acceptable, leave the device susceptible to communication errors. This is due to data paths crossing each other and high frequency lines next to each other. This could be resolved by simply adding another layer or better utilizing the two existing layers. The USB communication could also be slowed down to better the communication efficiency [2].

A second recommendation for the project would be to work on improving the enclosure. Currently the enclosure is a 3D printed 2 piece enclosure that sandwiches the parts within. With a limited size of 3”x3”x1”, the enclosure could be decreased in size even more that what it is currently designed as. Additionally, since the temperature and humidity sensor is externally mounted, a better secure brace should be made to cradle the sensor to prevent it from coming loose when falling from tall heights. This can be done by looking at common practices for bracing enclosures [3].

A third technical recommendation for calculation improvement using more data points, such as the feels like temperature, cloudiness, wind, pressure, aside from the main data points which are the temperature and humidity. These data points can be collected using the open weather api [4]. Once these are pulled, an improved calculation block could take this data and recommend window closing when it is raining for example or the sun is directly on a window.

Another recommendation is improving upon the wireless systems and connections between the application and the sensor. Currently, our system sets up a wifi network with a bluetooth connection and then uses our database to transfer our data. To improve this system, a team could implement a direct connection between the subsystems using a protocol like HTTP [5]. This would allow the project to send more information when the user is connected locally and potentially speed up the data transfer process.

#### 6.1.2. Global Impact Recommendations

Being an IoT device, our system interacts with the outside world quite often. When you open your windows, one of the most important conditions for health is air quality. Poor air quality can be dangerous for people inside the home, where a concentration of polluted air can lead to shortness of breath, coughing, chest pain, nausea, fatigue and asthma attacks for those at risk. An easy solution exists for this that we didn't have time to implement, location based weather data. If you asked the user for their location, a future group could use a weather api to find the local air quality and use that data to make a decision for the user's windows. You can learn more about the weather api on the website below [4].

Another recommendation we have for a future group is about the cost of our device. Our sensors use many chips and modules to allow our team to fulfill all our requirements. Due to the global chip shortage, these components are rapidly increasing in price and are limited in availability. Our temperature and humidity sensor cost $25 per unit. With this cost, a sensor could cost between $50 and $100 to build which could potentially be too much for buyers. Our solution would be to look at part prices to get the cheapest and potentially make your own waterproof sensor (SHT30 sensors without waterproofing are much cheaper) [6].

#### 6.1.3. Teamwork Recommendations

When it comes to teamwork recommendations, our group had two main suggestions to help future groups. The first recommendation is weekly work times. Our group had regularly scheduled meetings with our project partners, and 2 hours to work during class, but we lacked dedicated time to work as a group without distractions. While we worked well as individuals, creating the entire system was difficult without dedicated work time. One solution could be to set up a weekly time using a website like when to meet and dedicate at least 2 hours to working together to do group work [7]. Another solution would be to create a list with tasks that are checked off when a group member completes the task, that way work can be evenly divided among group members.

Our second recommendation is to create a place where all documents are stored early in the process, particularly when it comes to code. Early in the fall term, our group decided to create a google drive to store all of our documents, but storing more complex files, like code and pcb files, became a challenge. Later in spring term, we created a Github repo to store these files and be able to collaborate on them. As a team, we should have done this much sooner in order to collaborate more efficiently [8].

[1] EMS Solutions, "How to reduce PCB size," *EMS Solutions*, 29-Aug-2022. [Online]. Available: https://www.myemssolutions.com/how-to-reduce-pcb-size/

[2] Sherly and T. King, "How to fix slow USB transfer speed and speed up USB in Windows 10/8/7," *EaseUS*, 22-Feb-2023. [Online]. Available: https://www.easeus.com/computer-instruction/fix-slow-usb-transfer-speed.html

[3] D. Papp, "Simple tips for better 3D-printed enclosures," Hackaday, 16-Nov-2020. [Online]. Available: https://hackaday.com/2020/11/16/simple-tips-for-better-3d-printed-enclosures/

[4] "How to start to work with openweather API - openweathermap." [Online]. Available: https://openweathermap.org/appid

[5] "Esp32 HTTP GET and HTTP post with Arduino IDE," *Random Nerd Tutorials*, 27-Oct-2022. [Online]. Available: https://randomnerdtutorials.com/esp32-http-get-post-arduino/

[6] "Sht30-dis-b2.5ks: Digi-Key Electronics," *Digi*. [Online]. Available: https://www.digikey.com/en/products/detail/sensirion-ag/SHT30-DIS-B2-5KS/5872250

[7] *When2meet*. [Online]. Available: https://www.when2meet.com/

[8] "Create a repo," *GitHub Docs*. [Online]. Available: https://docs.github.com/en/get-started/quickstart/create-a-repo

### 6.2. Project Artifact Summary with Links

**Project Repo Link:** https://github.com/NikeGolf36/SeniorDesign

This Github project repo contains all the schematics, code, and documentation for the project. Within the main repo, additional repos are linked which contain our application code, microcontroller code and pcb layout. Enclosure files and any additional documentation (including this document) are present as well.

**Application Repo Link:** https://github.com/NikeGolf36/ClimateControlApp

This Github repo contains the android application code. Cloning this repo and opening in android studio will allow the user to download the application. A readme is included, which walks the user through the process.

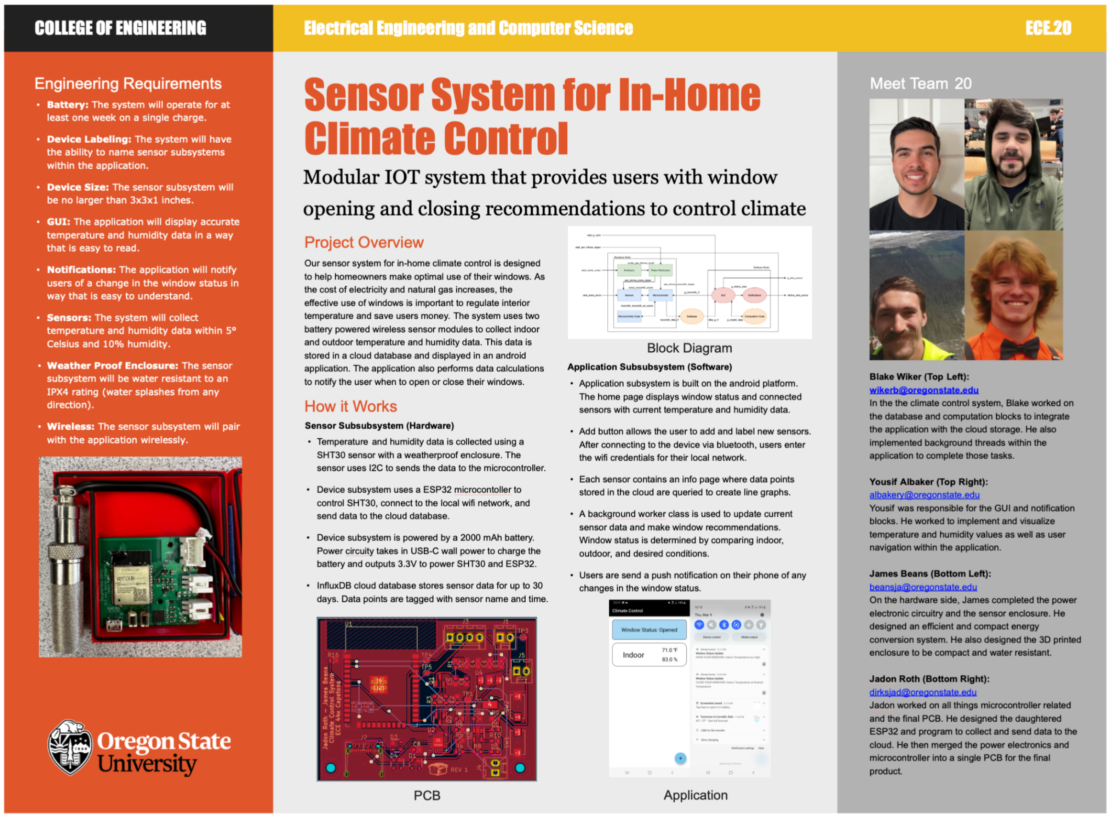
**ESP32 Repo Link:** https://github.com/NikeGolf36/ESP32_Sensor

This Github repo contains the C++ code to the esp32 microcontroller. The code can be run using the Arduino IDE.

**PCB Repo Link:** https://github.com/NikeGolf36/Sensor_PCB

This Github repo contains the Kicad project, schematic, and PCB files for the sensor system. A user will be able to clone this repo and create the gerber files within the Kicad application.

## 6.3. Presentation Materials



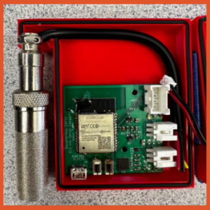
COLLEGE OF ENGINEERING

Electrical Engineering and Computer Science

ECE.20

Engineering Requirements

- **Battery:** The system will operate for at least one week on a single charge.
- **Device Labeling:** The system will have the ability to name sensor subsystems within the application.
- **Device Size:** The sensor subsystem will be no larger than 3x3x1 inches.
- **GUI:** The application will display accurate temperature and humidity data in a way that is easy to read.
- **Notifications:** The application will notify users of a change in the window status in way that is easy to understand.
- **Sensors:** The system will collect temperature and humidity data within 5° Celsius and 10% humidity.
- **Weather Proof Enclosure:** The sensor subsystem will be water resistant to an IPX4 rating (water splashes from any direction).
- **Wireless:** The sensor subsystem will pair with the application wirelessly.



Oregon State University

**Sensor System for In-Home Climate Control**

Modular IOT system that provides users with window opening and closing recommendations to control climate

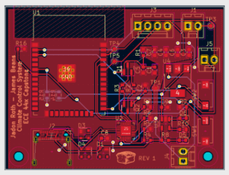
Project Overview

Our sensor system for in-home climate control is designed to help homeowners make optimal use of their windows. As the cost of electricity and natural gas increases, the effective use of windows is important to regulate interior temperature and save users money. The system uses two battery powered wireless sensor modules to collect indoor and outdoor temperature and humidity data. This data is stored in a cloud database and displayed in an android application. The application also performs data calculations to notify the user when to open or close their windows.

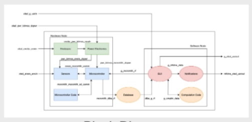
How it Works

**Sensor Subsubsystem (Hardware)**

- Temperature and humidity data is collected using a SHT30 sensor with a weatherproof enclosure. The sensor uses I2C to sends the data to the microcontroller.
- Device subsystem uses a ESP32 microcontoller to control SHT30, connect to the local wifi network, and send data to the cloud database.
- Device subsystem is powered by a 2000 mAh battery. Power circuity takes in USB-C wall power to charge the battery and outputs 3.3V to power SHT30 and ESP32.
- InfluxDB cloud database stores sensor data for up to 30 days. Data points are tagged with sensor name and time.



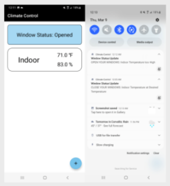
PCB



Block Diagram

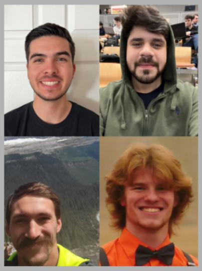
**Application Subsubsystem (Software)**

- Application subsystem is built on the android platform. The home page displays window status and connected sensors with current temperature and humidity data.
- Add button allows the user to add and label new sensors. After connecting to the device via bluetooth, users enter the wifi credentials for their local network.
- Each sensor contains an info page where data points stored in the cloud are queried to create line graphs.
- A background worker class is used to update current sensor data and make window recommendations. Window status is determined by comparing indoor, outdoor, and desired conditions.
- Users are send a push notification on their phone of any changes in the window status.



Application

Meet Team 20



**Blake Wiker (Top Left):**
wikerb@oregonstate.edu
In the the climate control system, Blake worked on the database and computation blocks to integrate the application with the cloud storage. He also implemented background threads within the application to complete those tasks.

**Yousif Albaker (Top Right):**
albakery@oregonstate.edu
Yousif was responsible for the GUI and notification blocks. He worked to implement and visualize temperature and humidity values as well as user navigation within the application.

**James Beans (Bottom Left):**
beansja@oregonstate.edu
On the hardware side, James completed the power electronic circuitry and the sensor enclosure. He designed an efficient and compact energy conversion system. He also designed the 3D printed enclosure to be compact and water resistant.

**Jadon Roth (Bottom Right):**
dirksjad@oregonstate.edu
Jadon worked on all things microcontroller related and the final PCB. He designed the daughtered ESP32 and program to collect and send data to the cloud. He then merged the power electronics and microcontroller into a single PCB for the final product.

**Project Showcase Link:**
https://eecs.engineering.oregonstate.edu/project-showcase/projects/?id=K11SITWJuzWW8SQn