

NOISE MONITORING SYSTEM



Team Members: *Liam Platt, Paul Wehbe, and Rich Patino*

Faculty Advisor: *Krista Hill*

Spring 2021

TABLE OF CONTENTS

1. PROJECT OVERVIEW.....	1
1.1 PROJECT OUTLINE	1
1.2 PROJECT BACKGROUND	1
1.3 OBJECTIVES	2
2. DETAILED USE CASE	4
3. CONSTRAINTS AND REQUIREMENTS	6
4. RISK ANALYSIS AND INVESTIGATION	9
4.1 KEY PRINCIPLES OR TECHNOLOGIES AND RISK ANALYSIS	9
4.2 INVESTIGATIONS	9
4.2.1 Sound Level Meter Investigation: Literature	10
4.2.2 Sound Level Meter Investigation: Mockup	11
4.2.3 Flask Web Application Investigation:	12
4.2.4 Database Investigation:	13
5. PROJECT BREAKDOWN AND SCHEDULE	16
5.1 PROJECT BREAKDOWN	16
5.2 DESIGN EVIDENCE GROUPINGS AND CONTRIBUTORS	17
5.3 SCHEDULE	17
6. DETAILED DESIGN	19
6.1 DESIGN EVIDENCE	19
6.1.1 Sound Level Meter	19
6.1.2: Dashboard: Front End Design.....	32
6.1.3: Dashboard: Back End Design	37
6.1.4: Notification System Design	43
6.2 BILL OF MATERIALS	48
7. TESTING METHODOLOGY AND RESULTS	50
7.1 TESTING METHODOLOGY	50
7.1.1 Frequency Response	50
7.1.2 Linearity.....	51
7.1.3 End-to-End Testing: Notification System	52
7.2 RESULTS	54

7.2.1 Sound Level Meter Testing Results	54
7.2.2 End-To-End Results.....	57
7.2.3 Noise Monitor Enclosure Results.....	57
7.2.5 Dashboard Web Interface Results.....	58
7.2.6 Notification System Results	59
8. ATTRIBUTES AND REFERENCES	60
8.1 ATTRIBUTIONS	60
8.1.1 Sound Level Meter Attributions	60
8.1.2: Chart,js.....	61
8.1.3: Notification System Attribution	62
8.2 REFERENCES.....	63
9. REFLECTIONS AND CONCLUSIONS	64
9.1 GREATER IMPACT	64
9.2 KNOWLEDGE GAINED	65
9.3 CONCLUSION	66

1. Project Overview

1.1 Project Outline

Our project consists of an integrated noise monitoring system designed to be utilized in residential spaces where noise can have a lasting and detrimental impact on residents. Specifically, this project is aimed at controlling and monitoring the often-excessive noise found in on-campus dormitory housing. The health effects of such noise are quantifiable by long-term hearing loss, disturbance of sleep and rest, social and behavioral problems, as well as temporary and permanent physiological and psychological conditions.

Our aims are to both reduce the work of residential staff and improve student housing on campus by streamlining the way in which noise complaints are handled. By utilizing a system of noise monitors spread throughout a building, residential staff can be notified when a resident is not following the University policies with regards to “quiet hours”. These are policies that restrict excessive noise late at night and are included in the University's 2020-2021 Student Handbook [5]. The noise monitoring units will be integrated through a centralized dashboard which serves as a secured database for noise level readings and as an interactive webpage for residential assistants.

The main target user of this project is residential staff. They are the ones who will be interacting directly with our system. The dashboard website itself will serve as their point of contact and all operations such as viewing data, notifying residents, and checking status’ will be done through it. This allows residential advisors to actively address noise from anywhere and make informed decisions about reoccurring problems.

1.2 Project Background

As mentioned previously, noise pollution can have short term and long-term side effects. For a student, such effects can be amplified- with lack of sleep and rest being linked to decreased academic performance. Psychological effects such as anxiety, depression, and stress can cause significant issues in personal wellbeing, to an already vulnerable population. In fact, the World Health Organization lists those “dealing with complex cognitive tasks” as a particularly vulnerable subgroup. Because of the reliance on studying done out of class during the COVID-19 pandemic this has further become an issue for students trying to succeed in their coursework.

At the University of Hartford in particular, a few major sources were found to have contributed the most to noise pollution on campus. Noise created by sources such as construction, road noise, large gatherings, and entertainment systems play a detrimental role in student wellbeing. While road noise is a major contributor, existing dormitories can’t be moved and adding any sort of sound dampening measures is oftentimes costly, especially in older historic buildings. Activities related to construction also, admittedly, play a huge role in noise pollution but are often in aim of improving campus life for students. Our project takes aim at the later of the two sources in hopes of improving student life and academic performance.

Since college dormitory housing is often tightly packed with adjacent units, the idea of noise monitoring as both a measure of wellbeing and as a deterrent becomes much more reasonable. With most noise complaints being handled by residential advisors the act of policing overly detrimental noise pollution is a challenging task. RAs at the University of Hartford are expected to uphold quiet hours from 10p.m.-10a.m. on weeknights and from 1a.m.-10a.m. on weekends[5]. This can become a taxing duty with residential staff having to address noise complaints that occur at any time of the night.

As with any project, overall impact and the ethical implications that follow were considered and used to guide overall design. One such ethical implication was privacy. This is common thread in many Internet-of-Things (IoT) projects and becomes particularly problematic when the sensor involved is a microphone. Specifically, for this project, concerns around unwanted ‘listening-in’ or monitoring of audio in residential spaces were raised. Because of this, certain design decisions were made to preserve privacy and create a secure network of sensors.

1.3 Objectives

Our objective for this project was to design a low-cost network of noise monitoring devices capable of measuring equivalent noise levels within a dormitory setting. As part of this, one working prototype noise monitor was to be fully constructed. To prove the capability of our project as a full-scale integrated system, such as the one in Figure 1.3.1, multiple noise monitors were implemented on breadboard as well. An interactive website for Residential Assistants to view noise levels; both current and past was also constructed. This serves as the point-of-contact for the user to login, notify residents, and view other information pertinent to residential life staff.



Figure 1.2.1: Use case of a noise monitoring network implemented throughout a city.

Work on this project was done by Paul Wehbe, Rich Patino, and Liam Platt. Liam Platt is an Audio Engineering Technology student with a passion for audio processing, both digital and analog. His work area included the design, development, and implementation

of the noise monitors themselves. Liam also worked on developing testing suites for said noise monitors.

Paul Wehbe is an Electrical Engineering student with a passion in communications and signal processing also both digital and analog. His work area throughout the project included contributions in the design of the noise monitors dashboard, which included the designing of the authenticated login/logout, front-end development using Adobe XD and Webflow, and working on developing the charts used within the dashboard.

Rich Patino is a Computer Engineering student with an interest in computer programming and hardware implementation. His work area throughout the project included commitments to creating back-end development with Google Firebase and Microsoft Visual Studio, design of the live dashboard web application, and developing email and text message system that was used to notify the resident.

2. Detailed Use Case

Our use case is an indoor dormitory setting at the University of Hartford. To anyone who has ever lived in college housing you know that sleep can be hard to come by on certain days of the week. This can be especially frustrating if you have early morning classes or simply just want some peace and quiet. As explained in the previous section, noise pollution can have some long-term psychological effects. These symptoms are especially hard on college aged individuals. Our project aimed to create a network of sensors, in hopes of streamlining the way in which noise complaints are handled.

Influencing our overall use case was our past experience as students and discussions with current residential assistants who would be directly interacting with the system. Based on these discussions our use case evolved.

Residential assistants would login to a centralized website upon starting their hours on duty. A list of all current rooms and noise indicators would be available and if the residential assistant deems it necessary, they can notify the residents of a room. Frequent violation of school policies on quiet hours would be easily viewed, and thus used as an indicator of when to send said message. The message would show up on both the residents cell phone and email and be directly associated with their student ID number.

This message would ideally influence the residents to quiet down and adhere to school policies. Residential assistants could easily send another follow-up message to indicate that future action could be taken in student conduct. One of our projects aims is to allow residents the time to correct any excessive noise before a noise complaint was filed. The following diagram shows our use case in action, with a noisy pair of students disturbing the studying of their neighbors.

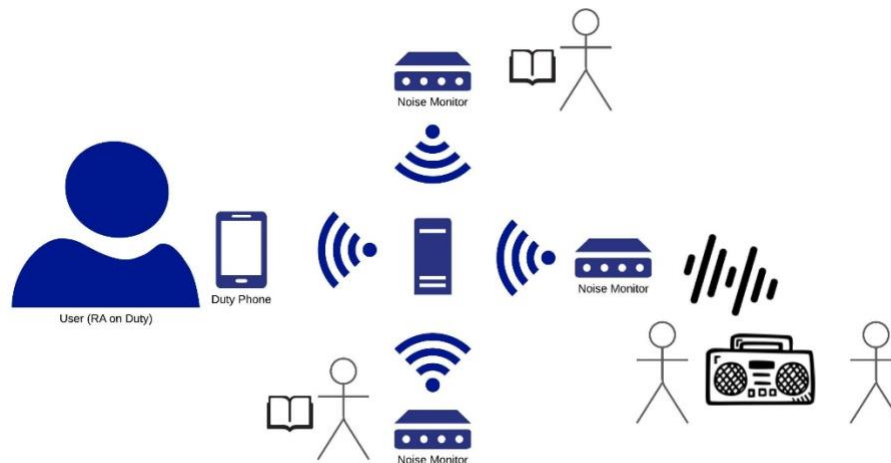


Figure 2.1: Use Case Diagram for Dorms at the University of Hartford

Our project would be ideally implemented on a larger scale. With multiple sensors set up across parts of the dormitory, noise complaints could be handled effectively. There would be no need for RA's to knock on doors or for students to balance getting sleep or being a 'party killer'. Due to our design, integration of hundreds of sensors could be made. Our system would provide a centralized history of noise trends on campus and serve as a marker of student wellbeing. Residential advisors would be able to send notifications directly to residents, lessening their workload overall.

3. Constraints and Requirements

Using the use-case of indoors in a multi-occupant dorm at the University of Hartford, our team was able to identify some specific requirements and constraints for our project. They range from compliance with certain sound level metering standards, to compliance with privacy concerns from RA's and residents. As specific details of the project came to fruition, our team also amended those requirements in order to get a clearer understanding of our goals.

As seen in item numbers 1.4 and 1.5, design changes were made with regards to a power supply and the needs of the noise monitor units. Due to the needs specified in our use case, the move from battery operated to a wired connection provided longer term stability and allowed for easy maintenance of the noise monitors. An alarm sound was also deemed to be too invasive for students, and thus was not incorporated into our final design.

Seen in item numbers 1.1-1.3 are the constraints put together based on IEC and ANSI standards. The specific standard referenced in our design was IEC-61672. While the standard could not be directly accessed, various resources referencing the characteristics defined within it were found. Such characteristics include the device having an overload and underload indicator present, a 70dB+ linear dynamic range, and a frequency response within an expected deviation over a number of defined frequency bands.

Item numbers 1.6-1.8 show the rest of requirements laid out for our noise monitor. The need for an IR sensor was prompted by questions on students tampering with the device, whether that be covering the microphone to obscure measurements or trying to manually shut down the device. The IR sensor included in the final design was able to notify the dashboard when various cloth materials were placed over the microphone. The next two requirements detailed how the device should operate. We decided that a continuously operating device needed to be implemented. We also decided that the method of communicating with the dashboard would be through a wireless connection to the schools' network.

	Item	Requirement	Test Method	Results	Completed	Partial	Missing
1.0 Performance	1.1	Linearity through operating range	Added: Test tone at 8kHz applied over full operating range	Linearity deviated significantly at high and low sound pressure levels. All other tests passed		X	
	1.2	Flat frequency response from 20-16kHz	Added: Acoustic test at full frequency range and Python program testing full range	Frequency response was within allowed deviation for all bands below 16kHz	X		
	1.3	Over/Under Indication	Added: Test at acoustic overload at 120dB (SPL) and noise floor at 30dB (SPL)	Overload triggers indicator Noise floor could not be reached in testing environment.	X		
	1.4	Beep as loud as smoke detector	Changed, N/A: Replaced with electronic notifications.	N/A			X
	1.5	Battery Operated	Changed - 5V AC power supply	N/A			X
	1.6	IR sensor triggers when covered.	Covering with a variety of fabrics that would 'dampen' sound.	Reacted to every material put within 5cm of the sensor.	X		
	1.7	Connects to WIFI	Send message through school network.	Sent message to dashboard	X		
	1.8	Continuous noise monitoring	Run over long period of time.	Ran through 3 days of operation.	X		
2.0 Physical	2.1	Three LED indicators indicate current level	Tests at 60dB, 75dB, and 95dB	Indicators responded correctly.	X		
	2.2	Must attach to any wall in a typical home	Test mounting on drywall and with command strips	Device can be mounted on any wall using a mount and screws.	X		
3.0 User Interface	3.1	Notifies admin through electronic notification	Modified- Admin notifies the resident using dashboard	"NOTIFY" button in dorm room page of dashboard	X		
	3.2	DIY install and setup	Setup and disconnected multiple times.	Monitor booted into program and ran correctly with no action needed.	X		
4.0 Environmental	4.1	Operating temperature between 32-100°F	Manufacturer's data sheet	Pi runs from -40-85°C. Microphone operates from -40 -100°C	X		
	4.2	Easily recycled	Manufacturer's data	Made from injected plastic molding	X		
5.0 Cost	5.1	Cost under \$100	BOM	Cost at \$31.90	X		

Figure 3.1: Project Requirements and Validation Table

Communications from Server to Sensors Requirements:

1. Reliable connection:
 - a. Set and forget.
2. Non-constant relaying of data
3. Latency (time it takes for server to process data) on the order of 1-2 minutes. Needs to be balanced with the sampling of noise level data.
4. No collision of data if sensor send data at same time.

When designing the noise monitor, we wanted the noise monitor to start operating automatically, where the noise monitor begins reading and recording noise level/connection status, along with sending that information over to the server. The team was able to solve this requirement by having a command run the noise monitors code automatically once powered while connecting to the server via HTTPS requests. For demonstration of the project the team did not run into the collision of data problem but believe that this would not have been an issue seeing how we were able to successfully configure the noise monitor through Firebase.

Database:

1. User authentication for RA's and staff
 - a. Secure and private.
 - b. Quick and simple login.
 - c. Each RA gets own user.
2. Data storage
 - a. Noise levels, time and date information, Wi-Fi state, and power status
 - b. Could store longer history of data.

For the noises.info dashboard, we needed multiple requirements, the basic requirements being having a server, host, database, and incorporate authenticated login. The solution to this requirement was incorporating Google Firebase to take on all those roles as the server, host, database, and authentication. The team wanted an authenticated login that was again secure, private, and simple. Firebase offered easy setup authentication, where when the team was developing the dashboard, we were able to incorporate email/password login. The Firebase platform has what is called the Firestore Database, in this database we were able to store noise levels/connection status along with their timestamp, having this stored into the database Firebase also allows for cloud storage where the team was able to incorporate back tracking of those alerts.

4. Risk Analysis and Investigation

4.1 Key Principles or Technologies and Risk Analysis

A risk assessment was conducted in the early stages of the project. This was used to pinpoint the riskiest elements of our design. Areas where the team had little collective knowledge and no firsthand experience are denoted by red. Areas where every member has knowledge, but lacked any experience are denoted by yellow. Finally, areas where we had both experience and knowledge are denoted by green.

Table 4.1: Risk assessment done at beginning of project. Shows the experience and knowledge of each team member in each design area. LP denotes Liam Platt, RP denotes Rich Patino, and PW denotes Paul Wehbe.

	0 Know Little About	0 Have Never Done
	1 Know About	1 Have Done Before
	Knowledge	Experience
Programming	(RP, PW, LP)1	1
Microchip Implementation	(RP, PW)1	1
Internet of Things	(RP, PW, LP)1	0
WiFi/Bluetooth Implementation	(RP, PW, LP)1	0
PCB Design and Fabrication	(LP,PW)1	0
Filter Design and Implementation	(PW,RP,LP)1	1
Mounting and Packaging	(RP,PW,LP)1	0
Preamplifier	(LP)1	1
Audio Rectifiers and Averaging Circuits	(LP)1	1
SLM Implementation	(LP)1	0

4.2 Investigations

To mitigate the riskiest areas of design and those with mild risk, our team conducted investigations into some of the key principles of the project. Key principles are those in which the project relies on, and lack of knowledge and/or experience in these areas lead to unnecessary risk.

4.2.1 Sound Level Meter Investigation: Literature

Given the lack of experience in sound level meter implementation, our group conducted research into effective design for that element. A sound level meter or SLM is made up of a few key principles- a preamplifier, an audio rectifying circuit, and a logarithm computational unit. When dealing with digital audio, this processing can instead be applied through an algorithm.

The roadmap for developing each corresponding algorithm was laid out at the end of the fall semester of 2020. This was influenced by research on existing SLM designs in both the analog and digital domains. One paper published by the Ogor Gresovnik[], provided a in depth overview of the elements that make up SLM algorithms. Detailed within this resource are the underlying equations for a few of the processes later implemented in our design.

Shown below is the equation for the calculation of A-weighted sound pressure levels with respect to a reference sound pressure level.

$$L_A(t) = 10 \log \left(\frac{p_A(t)}{p_0} \right)^2 dB$$

Figure 4.2.1: Equation for computation of dBA sound pressure levels, with reference to 20uPa.

What is important to note in this figure is that p_0 is referenced to 20uPa. A pascal is simply a measure of acoustic pressure and is often expressed as W/m^2 . This measurement is directly related to a SPL reading of 94dB at 1kHz and is one of the building blocks of our SLM design.

Based upon this equation is the equivalent continuous sound pressure level equation which was to be implemented within our final design. This type of reading was decided on for our final design after consulting with Professor Owen King from the Universities Acoustics Department. During the investigation phase of our project, Professor King suggested that an A-weighted measurement would be most applicable in an indoor setting and would be of interest when considering the ‘annoyance’ of any sound event. This is shown in Figure TODO below and was taken directly from Gresovnik’s paper.

$$L_{Aeq,T} = 10 \lg \left[\frac{\frac{1}{T} \int_{t_1}^{t_2} p_A^2(t) dt}{p_0^2} \right] dB; \quad T = t_2 - t_1,$$

Figure 4.2.2: Equation for the computation of equivalent noise levels over a period of T. Also referred to as LAEQ.

A host of low-power and highly accurate sound level meters were also used as a reference in our design. Due to the use case dictating a compact, low-cost design with long-term reliability, we spent a significant amount of time referencing materials with similar constraints.

One such paper was published by the *Polytechnic University of Madrid*[]. Published in January of 2020, it detailed a similar project goal: implementation of a low-cost integrated system of noise monitoring devices. Unlike our project, these devices were utilized in an outdoor environment and mainly tasked with monitoring road and city traffic noise in the city of Malaga. However, this paper detailed how digital signal processing in a fixed-point or floating-point architecture can reduce overall cost and increase the ability for large scale implementation. This heavily influenced the path the project took.

Mentioned later is the choice of the I2S protocol as a means of serial communication for our project. This was heavily influenced by our literature review on serial communications protocols. The communication protocols investigated were SPI (Serial Peripheral Interface), UART (Universal Asynchronous receiver-transmitter), I2C (Inter-Integrated Circuit), and I2S (Inter-Integrated Circuit Sound Bus). The original documentation for the I2S protocol by Phillips Semiconductors[] and literature on bandwidths of each protocol was referenced in this review[].

4.2.2 Sound Level Meter Investigation: Mockup

Throughout the project, mockups were constructed as a way of demonstrating progress and mitigating risk. The first mockup for the noise monitor was constructed in Fall of 2020 and showed the overall shape and profile of the device's enclosure. Seen below in Figure 4.2.3, is the initial housing design built from cardboard. It shows the proposed shape of size of our enclosure and would later be referenced when choosing the final prototypes housing.

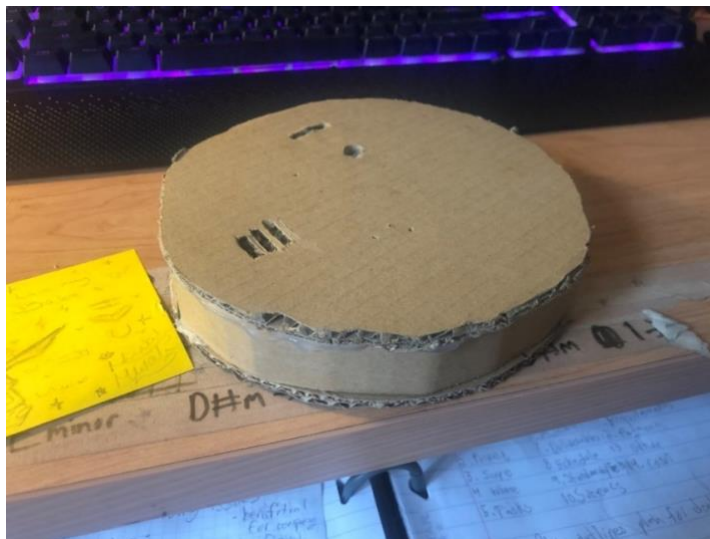


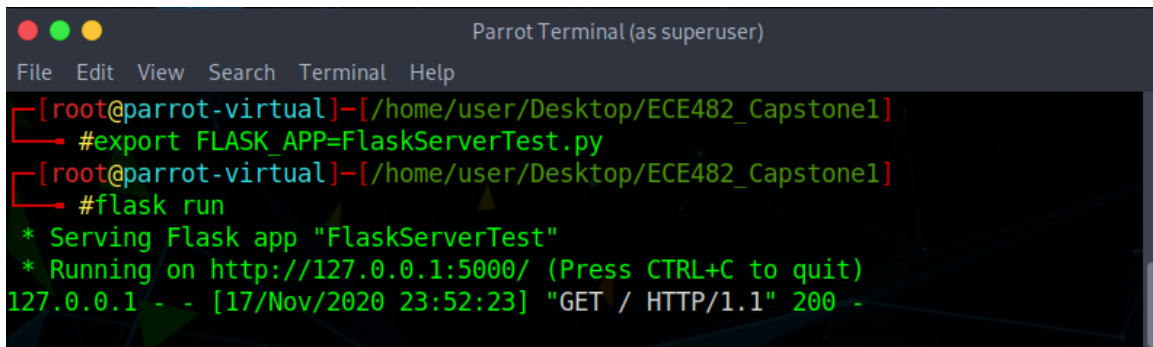
Figure 4.2.3: Mockup enclosure constructed from cardboard.

4.2.3 Flask Web Application Investigation:

The Flask server is a website application framework which has a purpose to provide all necessary tools when creating a website. An investigation that our team assessed was to create a simple python program followed by running the program onto the Flask server (Shown in Figure 4.2.7). After deciding to use the Linux operating system, the Flask application was installed using the Linux shell or terminal line by entering basic commands that were documented on [7]. The Python Integrated Development and Learning Environment known as IDLE was used to write the python code followed by then saving the file to an appropriate location on the Linux machine. The Python code that was generated contained a string that included: "Hello Rich, Liam, and Paul!" which was ran through the Flask server by then entering `flask run` into the Linux shell terminal window (Shown in Figure 4.2.6).

```
from flask import Flask          #imports flask library from the
Flask class
app = Flask(__name__)
@app.route("/")
def home():
return "Hello Rich, Liam, and Paul!"
# enter "export FLASK_APP=FlaskServerTest.py"
# enter "flask run" in terminal to run
```

Figure 4.2.4: Python Code Generated



The screenshot shows a Parrot Terminal window titled "Parrot Terminal (as superuser)". The terminal output is as follows:

```
[root@parrot-virtual]-[/home/user/Desktop/ECE482_Capstone1]
#export FLASK_APP=FlaskServerTest.py
[root@parrot-virtual]-[/home/user/Desktop/ECE482_Capstone1]
#flask run
* Serving Flask app "FlaskServerTest"
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [17/Nov/2020 23:52:23] "GET / HTTP/1.1" 200 -
```

Figure 4.2.5: Running Flask Server from Linux Command Shell

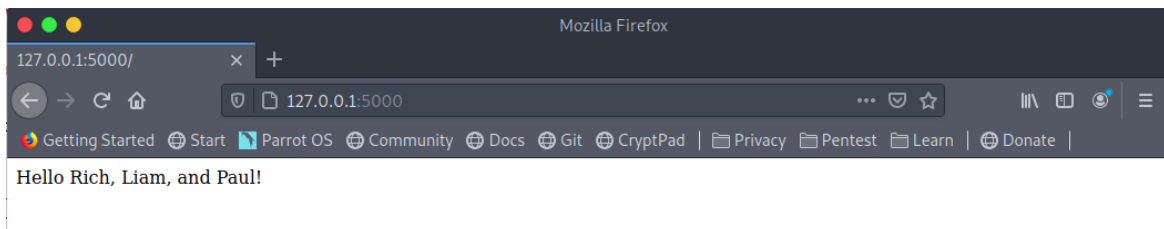


Figure 4.2.6: Program Output from Flask Server

4.2.4 Database Investigation:

Google Firebase is a back end as a service, providing developers with most if not all tools and services to help develop mobile and web application built on Google's infrastructure. Some of the key features Firebase provides are real-time database, crash reporting, authentication, cloud functions, cloud storage, hosting, and performance monitoring. A NoSQL database is used to store and sync data between users and devices in real-time using the cloud functions Firebase provides. Crash reporting is a feature of Firebase to diagnose problems in the mobile application along with the detailed reports of bugs and crashes. Authentication through Firebase can manage users in a simple and secure way. Being able to use authentication in multiple ways such as email and password, third-party providers like Google or Facebook can easily be configured to the authentication system directly. The cloud functions Firebase provides allow developers to customize back-end code without needing to change anything within the server. The cloud storage through firebase allows developers to store and share audio, video, images, and other user generated content at an affordable price.

The reason the team chose Google Firebase over Flask was because Firebase offered all in one functionality for building the noises.info dashboard. Having all the tools in one platform necessary to build the dashboard web interface into one platform like firebase allowed the team to easily configure Firebase with Microsoft Visual Studio instead of using all separate tools to make the dashboard work. Along with the all in one functionality feature that Firebase offers, the platform as a whole is very user intuitive and easy to get around, when their were tasks the team did not know how to complete there are many tutorials and walk throughs online to help as resources.

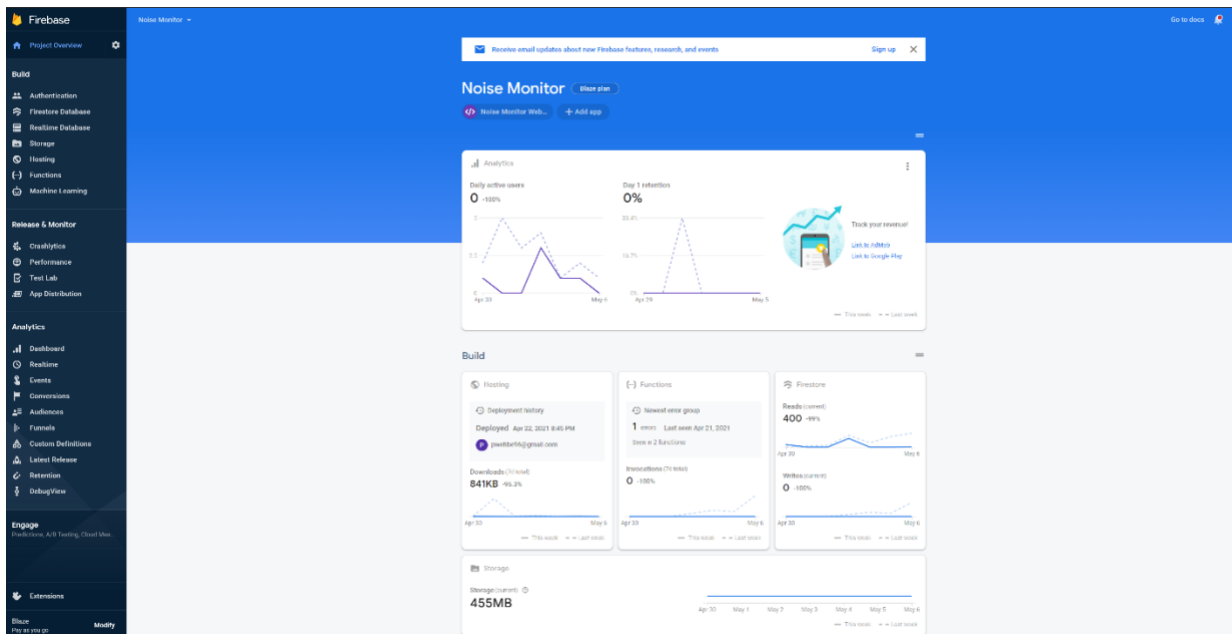


Figure 4.2.7: Google Firebase Console for Noise Monitor

The features the team used for the noises.info dashboard were the server, hosting, database, and authentication. The Google Firebase remote server is used to run the dashboard remotely and securely at an affordable price. If the team had resorted to going the Flask route, the hosting would have been done locally and not anywhere as secure as the Google infrastructure.

Hosting through Firebase allowed the team to display the real-time data being sent over from the noise monitor to the noises.info dashboard quickly and accurately, seeing that Firebase Hosting is a production-grade web content hosting for developers the team felt hosting through firebase would give this project the best results.

Using the Firestore Database within Firebase allowed the team to store the alerts sent over from the noise monitor into the database which would then be showed live onto the noises.info dashboard. Along with storing dorm information, such as student ID, phone numbers, and emails of the residents within the specific dorms.

Authentication through Firebase can manage users in a simple and secure way. The team was able to set up an authenticated login using a test email and password, if the dashboard becomes live and in use by the university, the authentication setup with the RAs Hartford.edu email will have been simplified due to how simple the Firebase setup is. Below are a few images displaying some of the information stored within the Firestore Database for the noises.info dashboard.

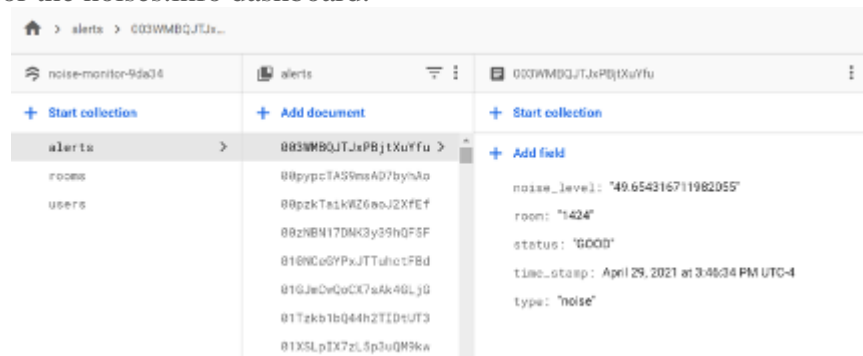


Figure 4.2.8: Firestore Database: Alerts from noise monitor

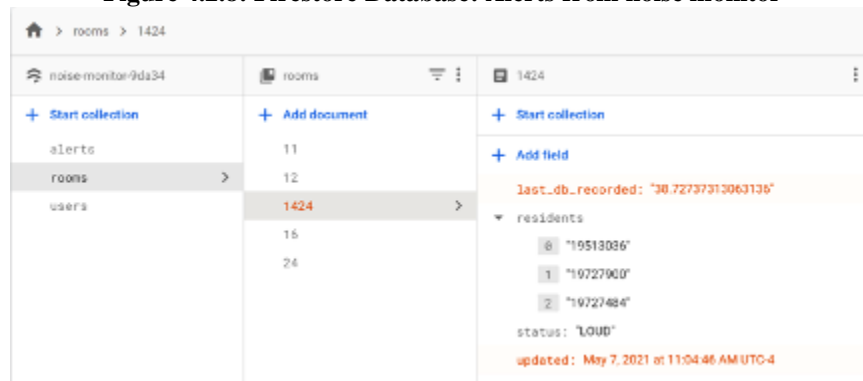


Figure 4.2.9: Firestore Database, Dorm room 1424 residents and status

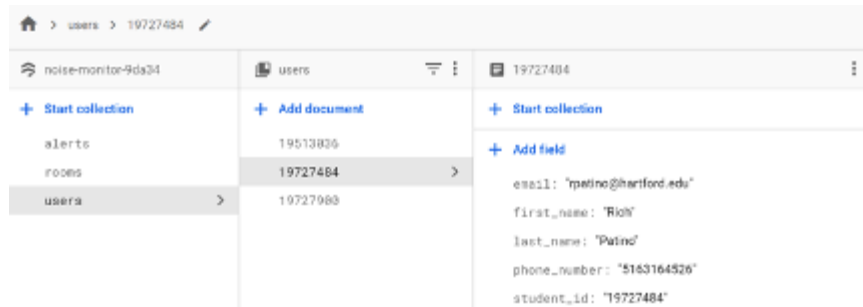


Figure 4.2.10: Firestore Database: Users/Resident Information

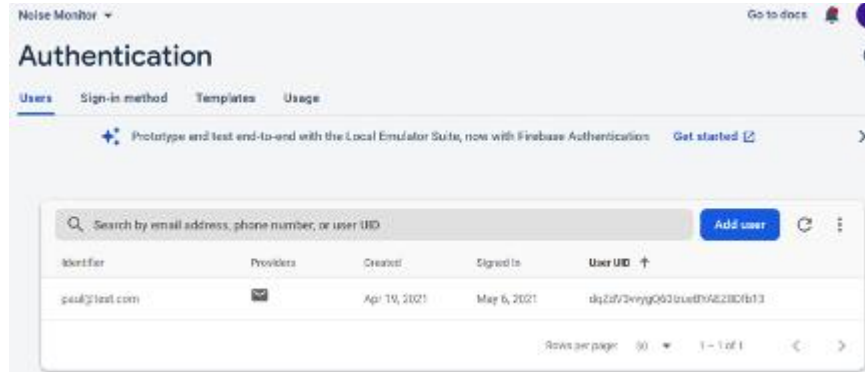


Figure 4.2.11: Google Firebase: Authentication Setup Page

5. Project Breakdown and Schedule

5.1 Project Breakdown

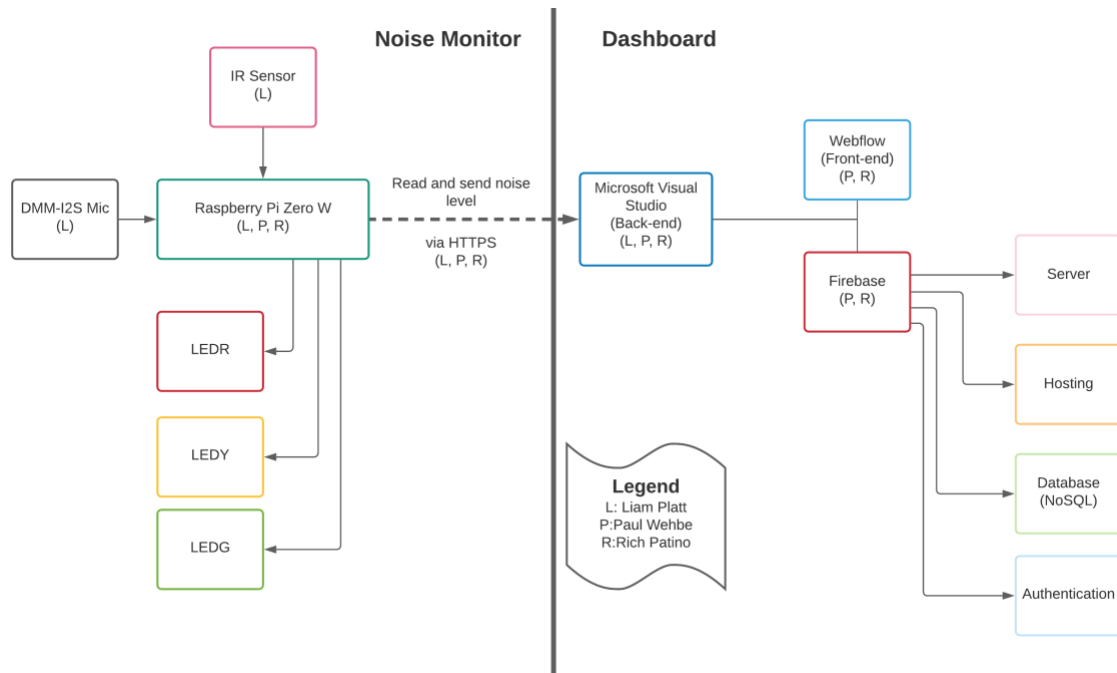


Figure 5.1.1: Block diagram showing the elements of our design.

1) Noise Monitor

- Raspberry Pi Zero W** – Used to connect all electronics components together and integrates with dashboard via Wi-Fi connectivity.
- DMM-4026-B-I2S-EB-R** - Microphone used to measure noise levels
- 3 LEDs** – Displays the room status by turning on one of the three LEDs (Green – Room is at appropriate noise level, Yellow- Noise level getting too loud, Red- Room is too loud).
- Infrared Sensor** – Notifies RA if microphone is covered.

2) Dashboard

- Adobe XD** – Used to design user interface for the dashboard
- Webflow** – Integrated with Adobe XD to ensure dashboard was fully functional.
- Microsoft Visual Studio**- Receives noise levels from the noise monitor and sends the data to Google Firebase.
- Google Firebase**
 - Server** – Works with Firebase's CLI to interact between files between the client and hosting servers.
 - Hosting**- Easy to use static web-hosting provider ensures that the web application is delivered quickly and securely.
 - Database**- A noSQL database allows data to be transferred and stored.

- iv. **User Authentication** – Allows RAs to log into the dashboard and view the current noise status of his/her assigned residences.

5.2 Design Evidence Groupings and Contributors

The team had two major design evidence groupings, the noise monitor, and the dashboard. The noise monitor as stated above in section 5.1 includes the raspberry pi zero w as our microcontroller that acted as the physical noise monitor. Attached to the pi were LED indicators (green, yellow, red) for the various noise levels. Along with an IR sensor and the DMM I2S microphone that was digitally filtered by Liam Platt. The components were pieced together and enclosed in a modified smoke detector, also done by Liam Platt.

The dashboard was building using three major components, Webflow, Google Firebase, and Microsoft Visual Studio. Webflow is a service for website building and hosting, the company has their own online visual editor platform that allows users to design, build, and launch websites, Paul Wehbe was successfully able to design implement the dashboard design using Webflow and transferring the files into Microsoft Visual Studio. Firebase is a platform developed by Google, acting as a server, host, database, and authentication for the dashboard. Both Paul Wehbe and Rich Patino were able to use Firebase and call for information that was stored in the database to the back-end of the dashboard. Items such as noise alerts (with time stamps), resident information such as phone numbers and emails were able to be stored into the Firebase and use them for the notification functions as well as the charts used throughout the dashboard.

5.3 Schedule

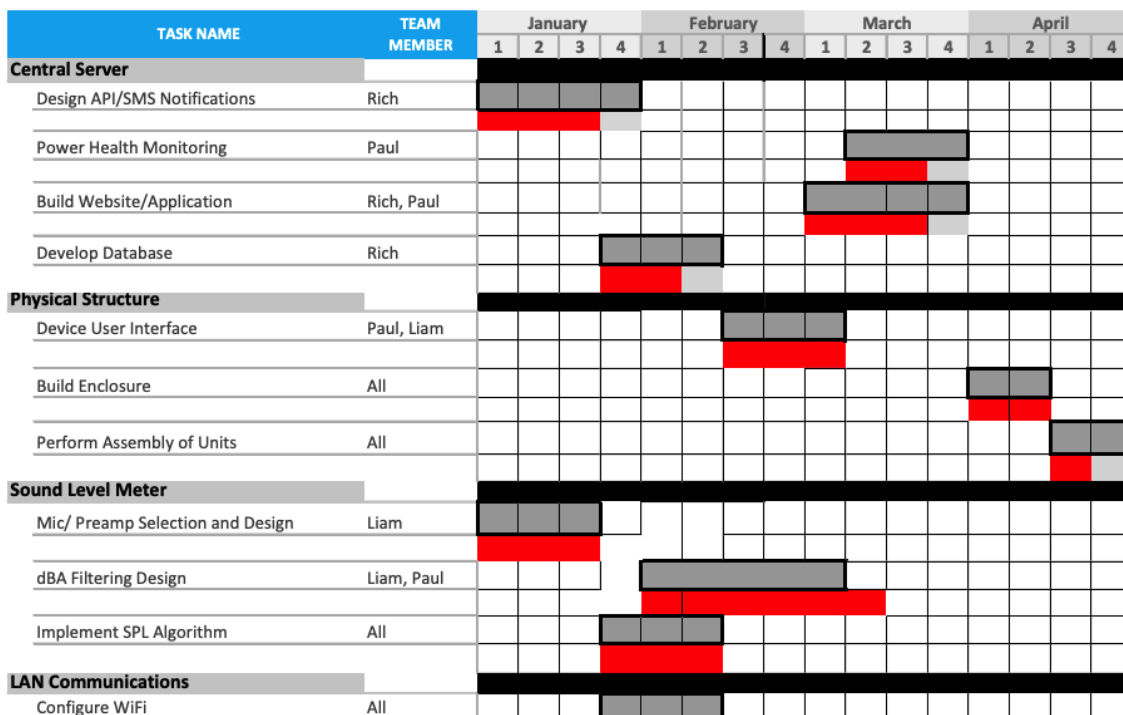


Figure 5.3.1: Gantt Chart illustrating the projects timeline. Areas in red are fully completed tasks and

reflect the actual time for completion. Area in gray are the initial projected schedule we had put together.

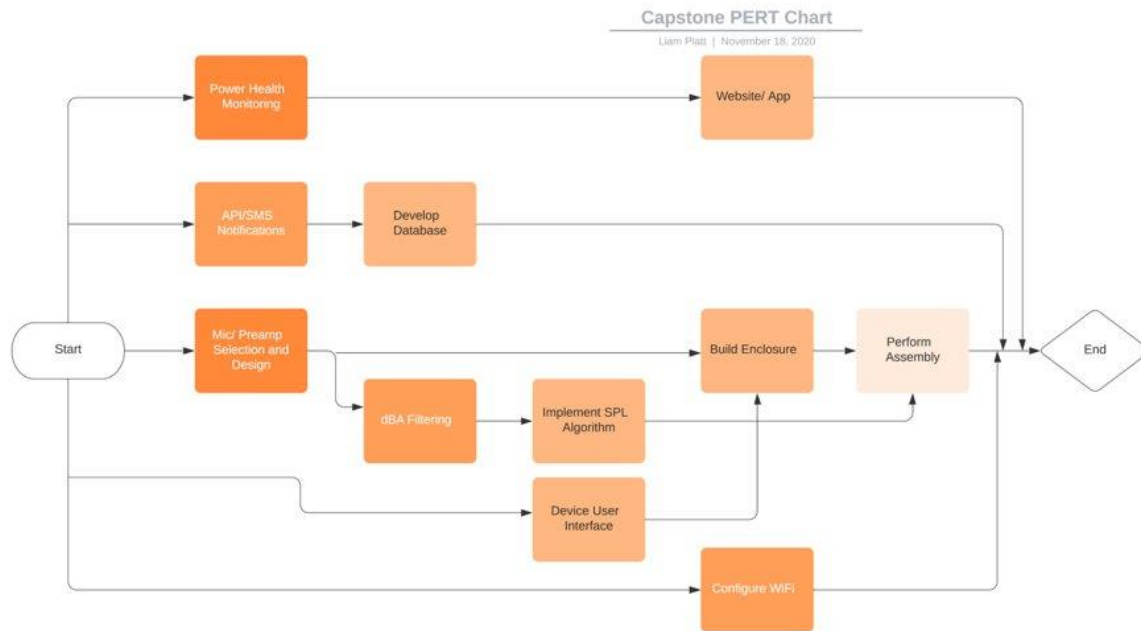


Figure 5.3.2: PERT Chart illustrating the order in which the design process happened.

6. Detailed Design

6.1 Design Evidence

6.1.1 Sound Level Meter

Our project consisted of three major design areas- a noise monitor, a central database, and a dashboard. With each area crucial to the project's success, research and design of each element was of utmost importance. At its basis, the noise monitor itself involved a few key elements that interact with each other and the rest of the system. Each element itself could be considered a marker of project success, and because of this each individual piece of the design needed to work. A careful division of this design area itself was made and each element was tackled independently of each other before integrating together.

6.1.1.1: Requirements and Constraints

There are three sets of requirements that were laid out for this design area: requirements regarding the microphone itself, the 'user interface' of the noise monitors, and those related to the interconnection of the noise monitors with the central dashboard. The first sub-section of this design area is the microphone and its related processing. Being an IoT project, the transducer itself was a main area of consideration. In the early stages of the design, a major decision was made to utilize digital processing rather than its analog equivalent.

Keeping this in consideration, requirements for the microphone were laid out in full. Since digital processing was to be used, we needed an analog to digital converter. This could be interfaced in one of two ways: either by utilizing an electret condenser in series with both a preamp and the analog to digital converter or by utilizing a microphone with a built-in codec, cutting out the need for extra components. The following shows the requirements constructed for the microphone.

Microphone Requirements:

1. Meets Class-2 SLM standards listed in IEC-61672.
2. Omni-directional polar pattern
3. Wide band frequency ranges from ~20Hz to 17kHz.
4. Signal to noise ratio of above 60dB
5. Small profile- the mic circuit should be under 1-cubic inch to fit in assembly.
6. Acoustic overload above 110dB for accurate reading in expected range (Typical PA systems can push out 110dB-115dB at max during concerts). Setting the limit high for worst case scenario, allows guaranteed accuracy of data.
7. Easy to maintain-
 - 7.1. Resistant to dust
 - 7.2. Resistant to humidity/temperature changes.

8. Operating temperatures from 0-120°C
9. Has I2S serial digital communication.
10. Solder point assembly.

One requirement listed was the use of I2S serial communication. Afterall, digital data coming from the microphone had to be relayed to the microprocessor, in this case, the Raspberry Pi. Serial communications such as I2C, SPI, and UART were considered, but ultimately the I2S protocol was chosen.

The main differences between similar communication methods is the ability of the I2S bus to preserve high bandwidth signals, deliver 24-bit words, and support 2-channel (stereo) audio. In Table 6.1.1 below is a comparison showing the difference between the protocols considered. Although UART and SPI both offer high-speed modes with speeds high enough to carry the full bandwidth of an audio signal, they were not considered. The struggle of working with the word-lengths supported by the two protocols would have outweighed any benefits they offered.

Table 6.1.1: Common serial communication protocols.

Method	Standard Data Rate (Mbps)	Bits per word	Numbers of words per sample (24-bit audio)	Bandwidth
I2C	.1	8	3	~4.167kHz
UART	.115	8	3	~4.79kHz
SPI	50	8	3	~2MHz
I2S	2.048-4.096	16, 24, 32	1	85.3kHz

After deciding on a communications protocol, component selection for the microphone itself was made with the initial selection of a low-cost MEMS microphone. The main reasons driving this decision were that the microphone needed to be small profile, resistant to indoor environmental conditions, and have a built in A/D converter supporting I2S data. At first, our team had decided on a the compact ICS43434 microphone featuring most of the requirements listed above. However, the frequency response of this selection was non ideal, with a sharp roll off happening below 100Hz.

While this would be just fine in most situations, in our use case it would have vastly underestimated the low frequency content present in any given room. Because of this a wider bandwidth MEMS microphone (DMM-4026-B-I2S-EB-R) was selected to replace the initial choice. This would have been a major setback, but because of the commonality in communications protocol it proved to be negligible.

6.1.1.2: Overall Design

The overall design of the sound level meter can be broken into six separate elements, shown below. The following section will discuss the digital processing constructed for each.

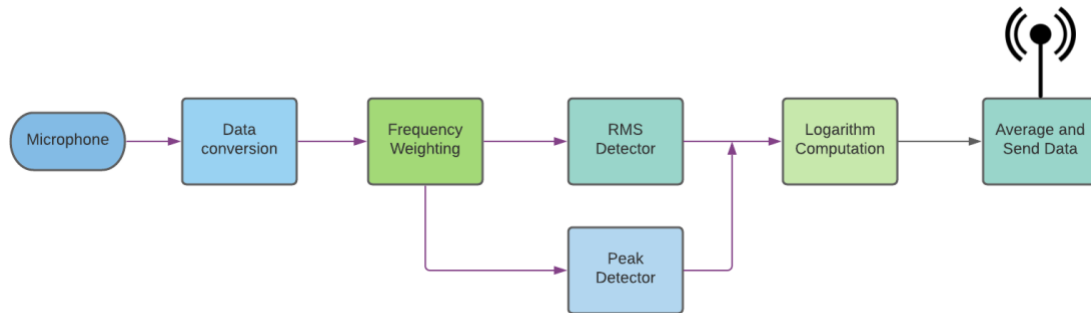


Figure 6.1.1: Block diagram of noise monitor processing.

Data Conversion:

The first subsection is data conversion. The sampled data coming from the microphone is blocked in groups of 375 samples and put into a buffer for unloading. Since the standard I2S bit-depth was 24-bit and the Raspberry Pi's PyAudio library only supported 16-bit and 32-bit operation, the samples had to be converted to usable values. Initially this mismatch hadn't been caught, and initial prototypes of the noise monitor were unable to read below 70dB.

This is primarily due to the calculation of a reference amplitude at start up. Influencing this calculation was the BD,

As seen in Figure 6.1.3, each block is iterated over, and a bit-wise shift is introduced, effectively performing division by 2^8 . This shift to the right by eight binary digits allows for further processing to be performed accurately. It is also important to note that the bit-wise shift operator requires minimal CPU, increasing the overall programs speed.

```
#Convert values from 32 bit to 24 bit by shifting 8 bits
#                                     <-----usable----->
#Data being read in will be in form 00000000 00000000 00000000 00000000
#We shift over the data 8 bits to the right, effectively dividing by 2**8
for n in range(len(data)):
    sample = (data[n] >> (BD_samp-BD))
    data_shifted.append(sample)
shifted = squaresum(weight_signal(data_shifted))
return shifted
```

Figure 6.1.2: Python code for shifting samples from 32-bit LE to 24-bit LE.

Frequency Weighting:

After being converted into a usable form, the data is then run through a frequency weighting filter. The one chosen for this design was an a-weighted filter. This specific type of filter mimics the response of the human ear, allowing for the sound level meter to behave in a fashion reflecting what is being heard. The characteristics of this type of filter are listed below and are given by IEC and ANSI standards.

Table 6.1.2: A-Weighted frequency response as defined by ANSI and IEC standards.

Frequency	Number Of Poles	Attenuation at 1kHz
20.6Hz	2	1.9997 dB
108Hz	1	
738Hz	1	
12.2kHz	2	

Using the given characteristics, an infinite impulse response (IIR) filter was constructed using the bilinear transformation method. This was initially designed in Matlab but was ported over to Python to run on the Raspberry Pi.

To design the filter, an analog prototype of the filter is first constructed, with the critical frequencies listed above and incorporating the attenuation needed at 1kHz. The original ‘s’ domain transfer function given by ANSI standards is as follows:

$$H_A(s) = \frac{k_A \cdot s^4}{(s + 129.4)^2(s + 676.7)(s + 4636)(s + 76655)^2}$$

Figure 6.1.3: ‘S’ domain transfer function for a-weighted filter.

Each pole of the filter is repeated convolved to produce the final B and A coefficients used in the weighting filter. Because of the specifications of this filter, an FIR filter would be far less efficient. This is because producing the digital equivalent using a feedforward design requires far more coefficients, possibly hundreds. In contrast the IIR filter designed only utilizes 7 coefficients. The frequency response of the filter designed is shown below.

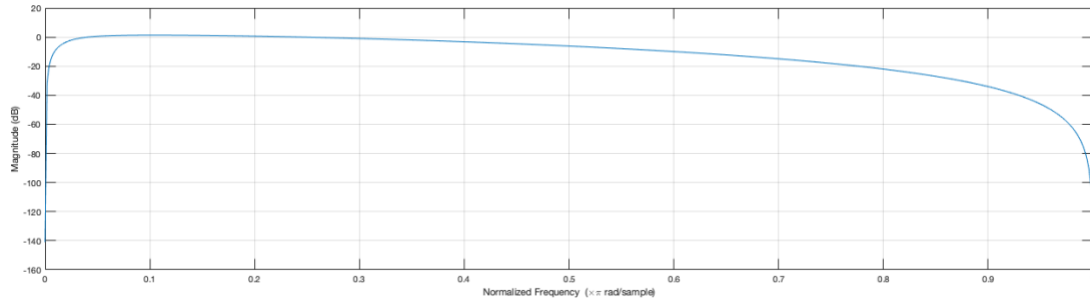


Figure 6.1.4: Measured frequency response of a-weighted filter designed. Illustrated using the Matlab ‘freqz’ function.

Another filter was also constructed, with its goal to block the DC offset present in the I2S data. This is common within microphones of this type and had to be accounted for. To construct this filter, the MATLAB *ellip* function was used. The frequency response for said filter is shown in Figure TODO. A passband ripple of 0.5dB and a stopband attenuation of 80dB was determined to be acceptable in the confines of the project.

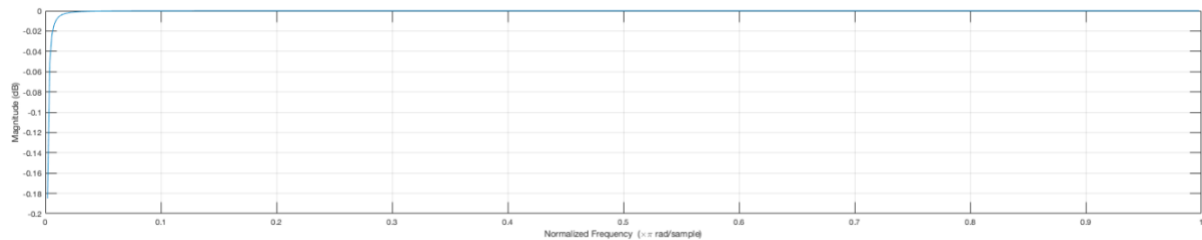


Figure 6.1.5: Measured frequency response of DC blocker filter designed used Matlab ‘ellip’ and ‘freqz’ functions.

The final implementation of both filters was performed by the *lfilter* function found within the *scipy.signal* library. This filters data using a direct-form II transposed IIR filter. The difference equation for this form, as denoted by *scipy* documentation is listed below:

$$a[0]*y[n] = b[0]*x[n] + b[1]*x[n-1] + \dots + b[M]*x[n-M] - a[1]*y[n-1] - \dots - a[N]*y[n-N]$$

Figure 6.1.6: Scipy ‘lfilter’ implementation in direct-form II transposed.

RMS Detector:

After compensating for the human ear's frequency response, the signal is then thread through an RMS algorithm. This does two things: it eliminates the ability for audio to be intercepted in any meaningful manner and it prepares our signal for accumulation over the devices sampling period. Given that the ethical issue of privacy was raised and incorporated into our projects requirements, this first result is crucial. At any given time, the device ‘buffer’ will only be filled with 7.8ms of audio, meaning that conversations and speech are inaccessible.

The RMS detector also serves the important purpose of averaging data over a period of time. As an example, this operation is shown both over a period of 1 second and over the devices sampling period of 7.8ms.

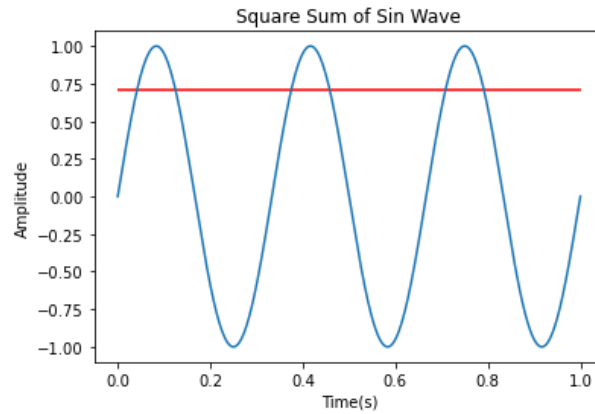


Figure 6.1.7: Square sum of a sinusoidal wave over 1 second. Used as proof of concept in early stages of design.

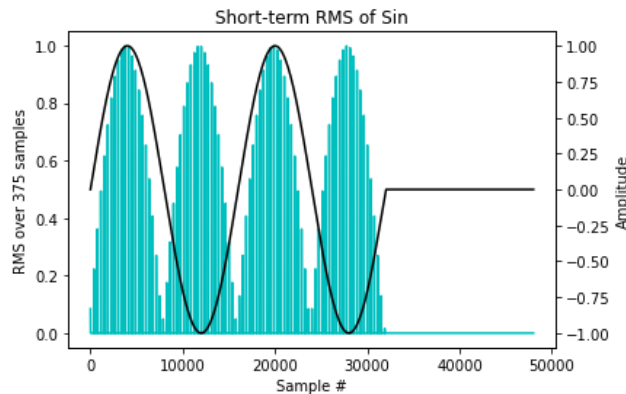


Figure 6.1.8: Short term RMS of a sinusoidal wave taken over blocks of 375 samples.

While implementing the RMS algorithm, the least CPU intensive option was pursued. Using research and some innately “Pythonic” syntax, we can effectively capture short-term RMS values. This operation is shown in the figure below. It is important to note that this function simply returns the squared sum of all elements in an array and does not return RMS values itself. Instead that operation is done inside the main program by simply taking the square root of the squared sum divided by the sampling period.

```
120 def squaresum(n) :
121     # Iterate i from 1
122     # and n finding
123     # square of i and
124     # add to sum.
125     sm = sum(i*i for i in n)
126     return sm
```

Figure 6.1.9: Python code implementing a sum of squares formula. Found to be lowest computational cost implementation in Python.

This of course, follows along with the mathematical formula for RMS of a signal. This is shown as reference in the figure below.

Root Mean Square

$$x_{rms} = \sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2} = \sqrt{\frac{x_1^2 + x_2^2 + \dots + x_n^2}{n}}$$

Figure 6.1.10: Root Mean Square equation.

Logarithmic Computation and Peak Detector:

The next element in the sound level meters design was the process of computing decibel values and using those values as a peak indicator. This was by far the most critical element to get right. Without it, the linearity of our device would be non-existent, and any value read in would be null and void. The heart of this element is shown below.

```
#Iterate through 128 times, each time collecting 375 samples
#
#Each time a check that calculates short term spl values is used
#to verify it isnt above mic OL or below mic noise level
#
#Add the square A-weighted level to our leq_sum
#When enough samples have been taken in calculate leq_db over 1
minute
for i in range(int_time):
    if IR_SENSOR.is_pressed:
        #print("IR SENSOR COVERED")
        rled.on()
    else:
        ind.off()
    #Read in block of 375 samples and store the weighted squaresum
    value
    sqr_sum = read_samples()
    short_RMS = sqrt(sqr_sum/bank_size)
    short_spl = OS + ref_1k + 20 * log10(short_RMS/mic_ref)
    if short_spl > mic_OL:
        short_spl = 120
        rled.on()
    elif short_spl < NF:
        short_spl = 30
        rled.on()
    long_samples += 1
    long_sum += short RMS
```

Figure 6.1.11: Main processing block of noise monitor program. This shows the peak detection implementation as well as the operation for computing RMS and SPL values.

The process starts by first declaring an iteration period of 128 readings. Blocks of 375 samples are read in 128 times, meaning that a total of 6000 samples are included in each reading. An intermediate SPL value is computed at each iteration, ensuring that the reading coming in are below the acoustic overload point and above the noise floor of the microphone. At the end of the loop, the short-term RMS is accumulated for use later.

Averaging Data:

After accumulating readings over a set period, the program then computes the equivalent noise level. This is done in a unique way, influenced by our use case. Every RMS value computed previously is average together, and then pushed into a buffer. This serves two purposes.

Firstly, this cuts down on the total computational cost of the program. Originally, the design had computed short SPL readings at every iteration through the main loop. This had forced the use of a computationally expensive function to average said readings. This cannot be done arithmetically as such would approximately equate to the loudest reading within the period being averaged over. This would be undesirable, as residential advisors don't need to know about short fluctuations in sound pressure level. Instead, the values had to be effectively converted back to their RMS counterparts, costing CPU time that was not necessary. This can be seen in Figure TODO.

```
162 def avg_db(avg):
163     """
164     """
165     avgsum = 0
166     N=len(avg);
167     for j in range(N):
168         avgsum += 10**((avg[j])/20)
169     avg = 20 * log10(avgsum/(N))
170     return (avg)
```

Figure 6.1.12: Function used to average decibels. This was found to be too computationally expensive and its use was minimized.

To get around this, the RMS values themselves are stored in the buffer and later averaged to produce an LEQ value over a period of roughly two minutes. Unique to this algorithm, is the utilization of a moving average of RMS values. By doing this, we further filter out short fluctuations within the sound pressure readings.

```

#After reading 128*375 samples, compute 'short' sound pressure readings.
#Store in array to be used for a moving average
long_rms = (long_sum/(long_samples))
long_sum = 0
long_samples = 0
avg.append(long_rms)
#If we've read in enough data->start taking a moving avg
#Moving average acts as a filter where quick fluctuations in sound pressure
#are ignored
#Append this moving average to a list to average and send to dashboard
if leq_buffer > window_size-1+OF:
    #Average the readings in moving buffer and push into list
    #FIFO
    average = OS + ref_1k + 20 * log10((sum(avg[0:window_size])/len(avg))/mic_ref)
    long_avg.append(average)
    #print(f"LAEQ: {average} at {datetime.datetime.now()}")
    display_leds(average)
    avg.pop(0)
if send_buffer >= num_sec-(60/send_rate):
    #Average the long_spl readings in moving buffer
    t0 = datetime.datetime.now()
    leq_db = (avg_db(long_avg))
    #Check if IR sensor was covered, use LOUD as a status
    if sensorPressed > 128:
        status = "LOUD"
    else:
        #Otherwise, set status accordingly
        if leq_db >= high_thr:
            status = "LOUD"
        elif leq_db < high_thr and leq_db >= low_thr:
            status = "WARNING"
        else:
            status = "GOOD"
    #Send status to dashboard
    send_status(1424, leq_db, status)

```

Figure 6.1.13: Python code for averaging of data using a moving buffer. The buffer operates in a first in first out format, with averaging done over a window size of 10. This was found to produce the best results and ignored short fluctuations in SPL.

Sending Data:

The final step within this design was the averaging of our moving buffer. This was done utilizing the *avg_db* function listed in Figure TODO. It was found that the most stable and accurate readings over our accumulation period were given when applying this operation. While LAEQ values are typically computed by simply accumulating a ‘sum’ of RMS values, for our project’s use case this was only partially followed. As mentioned in Section 1, the goal of the project was to give RA’s a clearer picture of continuous violations in noise policy. Having the program less susceptible to loud events that occur in a short period of time aids in this.

The exchange of data occurs within the *send_status* function. Using the Python requests library an HTTPS ‘GET’ request is made, and upon a successful handshake with the database the data is sent. There are three parameters that are being sent to the database. The first is a room number for identification of where the noise monitor is located. The next is the equivalent noise level which will be sent in LAEQ dB. The final parameter is the status and specifies what the general range of the noise levels in the room is for that reporting period. This can either be “LOUD” for over 90dB, “WARNING” for over 70dB but under 90dB, or “GOOD” for anywhere under 70dB.

```

256 def send_status(room, noise, status):
257     PARAMS = {'room':room,'noise':noise, 'status': status}
258     try:
259         r = requests.get(url = URL, params = PARAMS)
260         if r.status_code == 200:
261             print("Data successfully sent to server.")
262         else:
263             print("Failed to send data to server")
264     except IOError:
265         print("Failed to senf data to server. Will try again soon.")
266     return

```

Figure 6.1.14: Python function to send current noise level readings, room number, and status to the dashboard using an HTTPS GET request.

The connection of the device also played a role in this design element. After research into the Universities network, we were able to implement a wpa supplicant file. What this file does is tell the device the proper username, password, and security features of the network. Because the school operates an enterprise WPA-2 network, a traditional connection could not be made. The following is a screenshot of the wpa_supplicant file used to connect to the schools network upon startup.

```

ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
country=US

network={
    ssid="hawknnet"
    proto=RSN
    key_mgmt=WPA-EAP
    identity="platt"
    password=
    phase1="peaplabel=0"
    phase2="auth=MSCHAPV2"
    priority=1
}

network={
    ssid=
    psk=
    priority=3
}

network={
    ssid="uhdevices"
    psk=
    priority=2
}

```

Figure 6.1.15: wpa_supplicant file located on Raspberry Pi. Serves the function of connecting to the school network and providing backup connections.

As seen in the figure above, there are three different network connection options, each with their own priority. The priority network is the schools WPA-2, and if available, the noise monitor will connect to it on first try. If unavailable, the noise monitor will then connect to the backup network, the schools WPA network. This is typically used for gaming consoles, televisions, and other devices, but was used for our project as an easy way to securely access the dashboard.

6.1.1.3: Design Issues

One of the major issues run into was the Raspberry Pi's lack of support for Inter-IC-sound (I2S). The Raspberry Pi Zero we had selected to use in each unit did not come with any official drivers like they would with other serial communications like SPI and I2C. This of course proved to be a significant hiccup in the initial stages of the project. Rather than being forced to switch the microphone component already selected, a solution was found using an installation guide provided by Adafruit [6].

By modifying the ALSA (Advanced Linux Sound Architecture) device drivers and utilizing the data lines and clock pins already available on the Raspberry Pi, the I2S microphone was able to be interfaced properly. An ALSA "dummy-device" was created to make the program work with Python's PyAudio library.

Much like other means of serial data communication, the I2S protocol is intended to interface two electronic devices. This allows for the effective communication of data whether that data is coming from a sensor or being sent from one microcontroller to another. Specifically, the I2S protocol was created for the high-quality communication between digital audio devices. This protocol consists of a serial data line (SD), word select line (WS), a left/right (L/R) control line, and a master clock. This may sound familiar for those with experience with any other form of pulse density modulation (PDM). Shown in Figure 6.1.14 is a timing diagram of the I2S peripheral.

The word-clock (WC) line selects which channel the data is being received on. For our project, this line was tied directly to ground since we used only one I2S device. Since the data format inherently supports a second channel, a second microphone could be easily implemented. This gave added flexibility in our final design and could be used later to implement processing for sound localization.

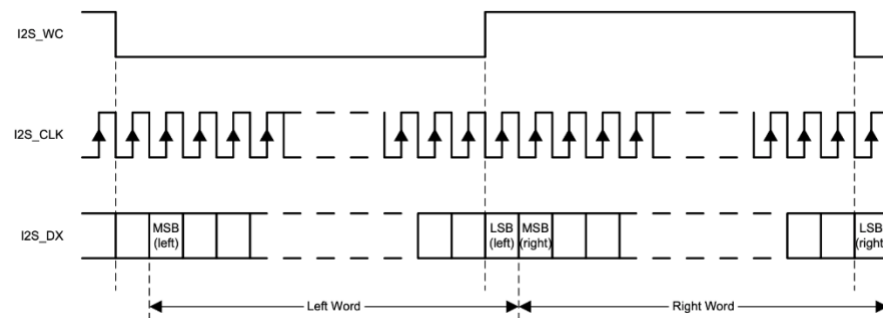


Figure 6.1.16: Timing diagram for I2S serial communication. WC line is tied to ground when a second channel is not being used.

The sampled data being received by the Raspberry Pi is in 24-bit little-endian format, with the leftmost bit being the most-significant bit (MSB) and the rightmost bit being the least-significant bit (LSB). This is what is known as little endian (LE) format. However, this bit-depth did not line up with the bit-depth choices supported by the Python libraries used in our program.

To fix this, a simple bit-wise shift was introduced on each sample of the data. Given that the difference in bit-depth was 8 bits and that the data being received was in little-endian format, a right shift of 8-bits would effectively scale each sample. While another option would have been dividing each sample by 2^8 , bitwise shifts require far less processing power and because of this were chosen as the method of scaling input data. This process can be seen in Figure 6.1.15.

```
#convert value from 32 bit to 24 bit by shifting 8 bits
#                                     <-----usable----->
#Data being read in will be in form 00000000 00000000 00000000 00000000
#We shift over the data 8 bits to the right, effectively dividing by 2**8
#Applies a dc blocker to signal using direct difference equation
for n in range(len(data)):
    sample = (data[n] >> (BD_samp-BD))
    data_shifted.append(sample)
```

Figure 6.1.17: Shifting of samples performed by a right bit-wise shift.

6.1.1.4: Alternate Approaches Considered

As mentioned previously, our team had made a significant design decision to pursue digital processing rather than analog. This was due to a few important characteristics that digital processing offered:

Firstly, the widespread implementation of numerous SLM's on a level that would be required for our use case required the stability that digital processing offered.

Secondly, our team had no experience in PCB board design and an analog sound level meter would all but require this. This posed a significant risk to our project, and one that could be easily avoided.

Thirdly, the implementation of digital processing and an I2S microphone eliminates the need for a PCB board, meaning that space can be saved in favor of a low-profile design. This was especially crucial when fitting the soldered circuit boards for the IR sensor and LED components and the Raspberry Pi inside of a smoke detector housing.

The following is the initial circuit design for the weighting filter. This would have been one of four circuits needed for an analog design. Its characteristics were confirmed as part of the investigation detailed in Section 4.2.1.

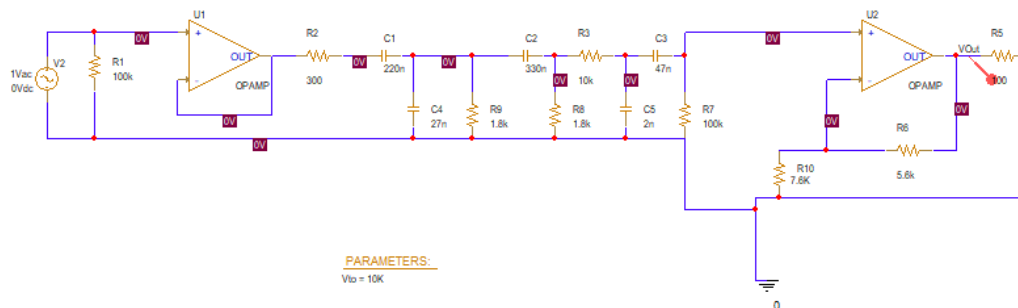


Figure 6.1.18: Initial analog a-weighted filter design made in Cadence PSPICE. Features passive filtering, a buffer amp, and an output gain amplifier.

6.1.1.5: Preliminary Results and Validation

Preliminary testing was done throughout the design of the noise monitor to verify the right path was being followed. We were able to verify clear audio was being sampled from the microphone by setting up a simple Python script to capture the sampled data and output it to a wav file. The test was conducted with a 1kHz sinusoidal wave being played back near the device. The time domain and frequency domain signal are shown below. A clean sin wave was reproduced faithfully.

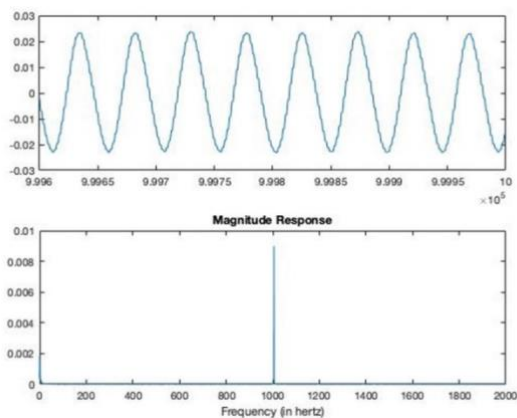


Figure 6.1.21: 1kHz sin wave tested through noise monitor. The frequency and time domain signals are shown, with the sin wave being reproduced clearly.

6.1.2: Dashboard: Front End Design

Front-End Design Layout (Adobe XD)

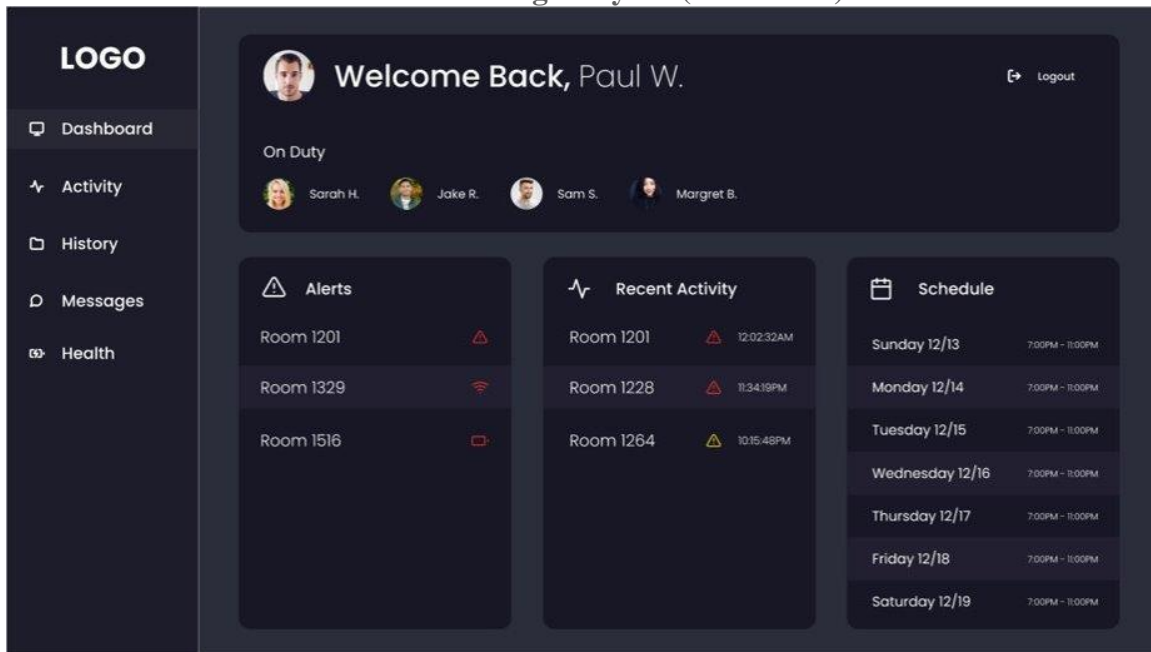


Figure 6.1.22: Adobe XD: Dashboard Page

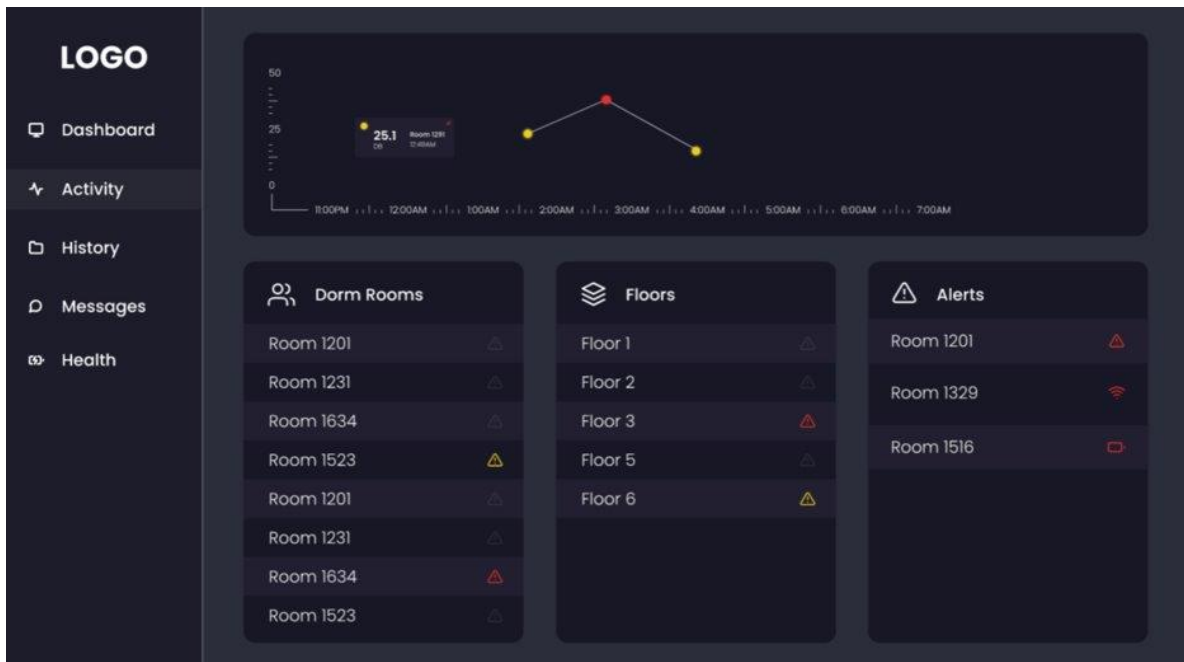


Figure 6.1.23: Adobe XD: Activity Page

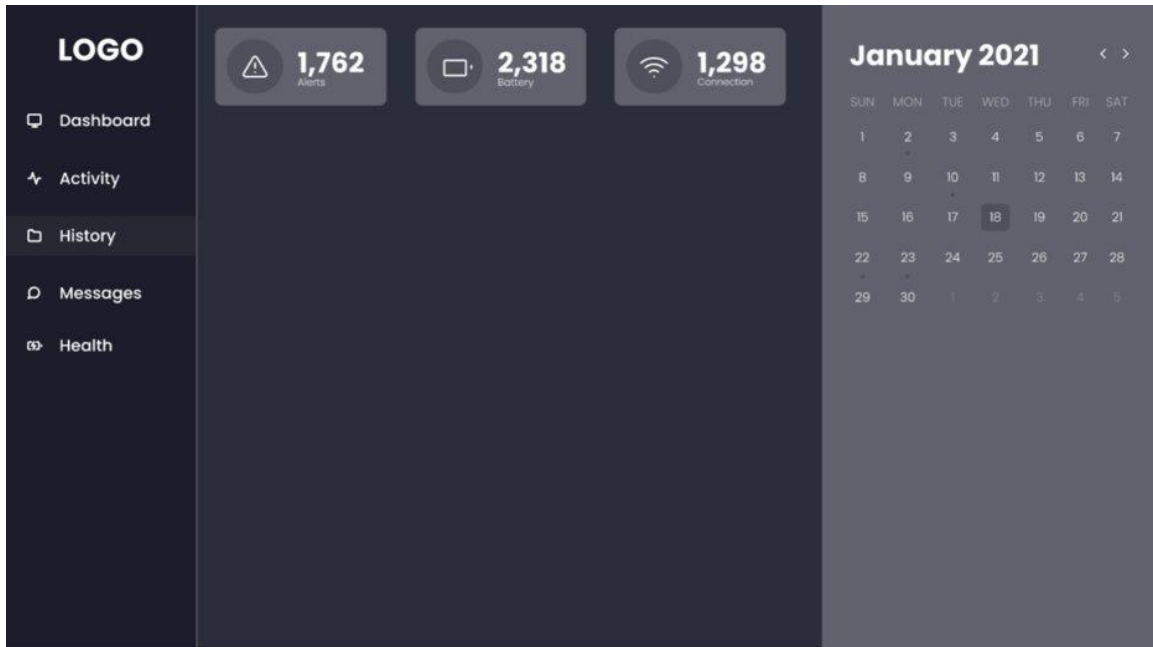


Figure 6.1.24: Adobe XD: History Page

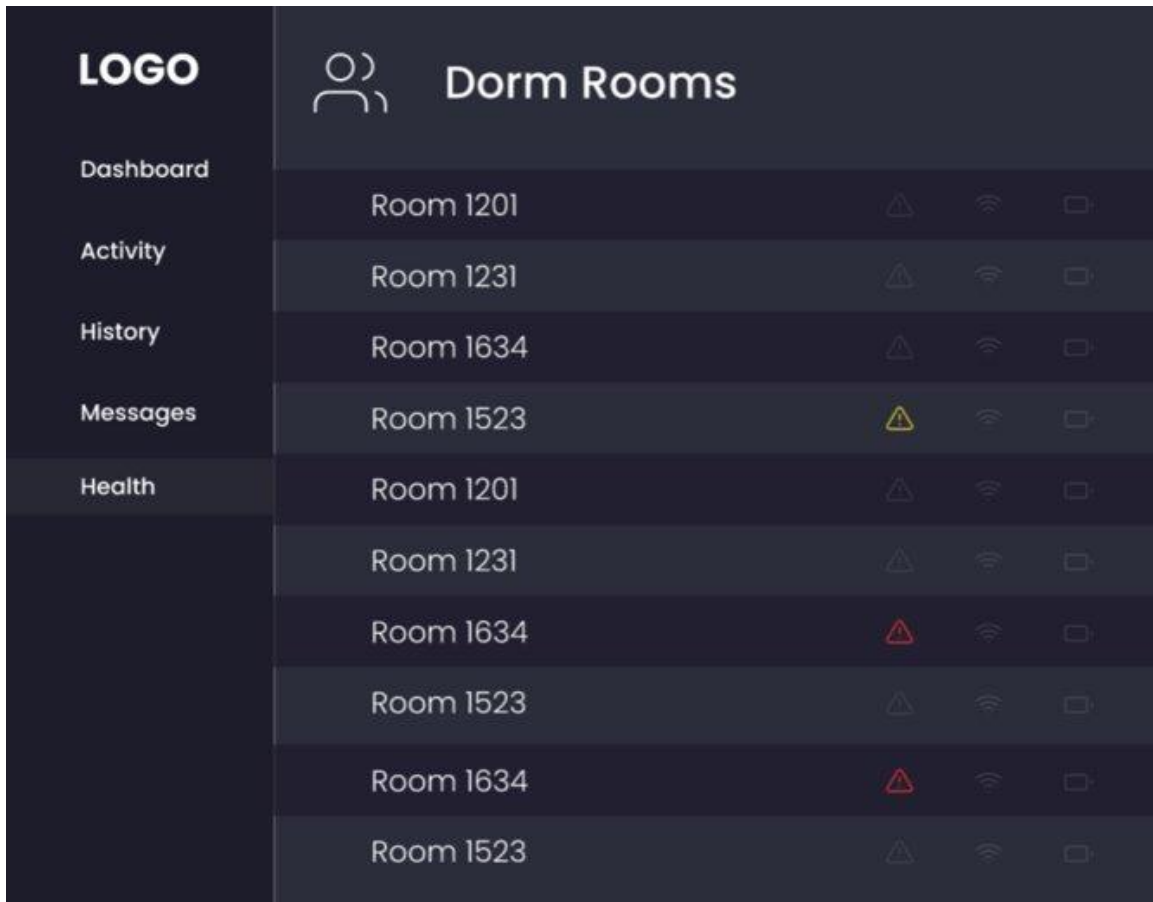


Figure 6.1.25: Adobe XD: Health Page

When going over design ideas of the dashboard that was going to be used for the noise monitor project, the team had the vision that the dashboard would have the multiple pages shown above, a home page, activity, history, messages, and health page.

It was a great innovative idea to add the messages page so RAs would easily be able to communication with each other on duty through the dashboard, for the project's purpose this was unnecessary and will have taken time away from the main purpose of the project by adding the messenger tab onto the dashboard. Below the initial Webflow design of the front-end was created. Eliminating the messenger tab and make some revisions on the actual look of the dashboard seeing that this was done on a different platform, the exact replica of what was done on Adobe XD was not entirely possible.

Initial Webflow Front-End Design:

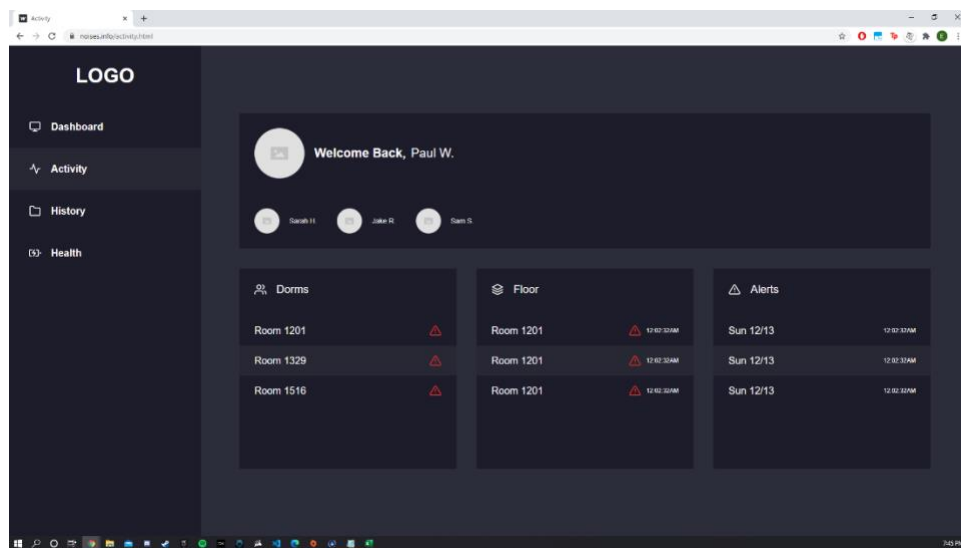


Figure 6.1.26: Webflow: Activity Page

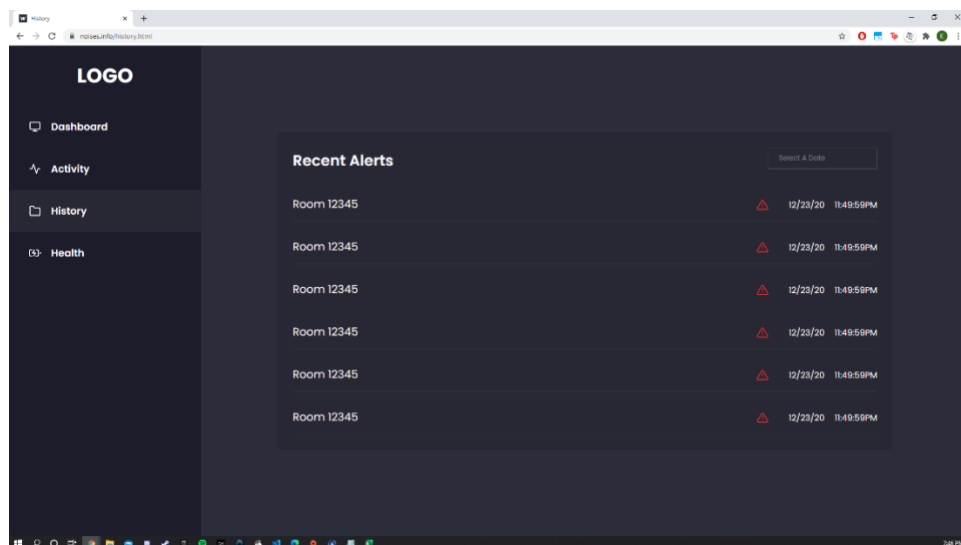


Figure 6.1.27: Webflow: History Page

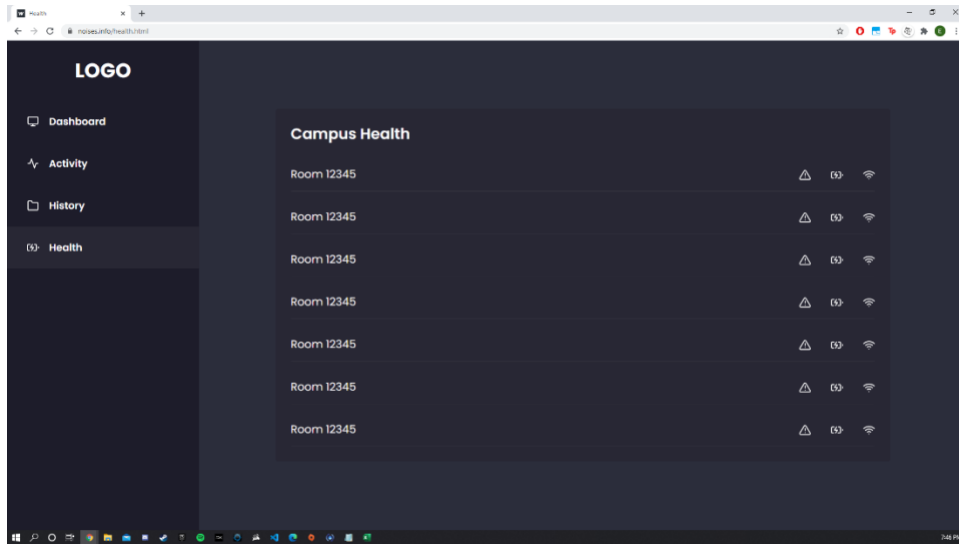


Figure 6.1.28: Webflow: Health Page

The screenshots above show the initial Webflow front-end design of the dashboard, where the team was able to almost replicate what was designed on Adobe XD onto the front-end visual designer used (Webflow).

The main difference from the Adobe XD design to the initial Webflow design was removing the messages tab, to optimize the dashboard by removing this unnecessary tool. The Activities page gave detail on room status from each individual dorm and floor along with the time stamp to the noise change. Where the history page was almost similar just showing the most recent noise level alerts. The health page displayed the noise level, Wi-Fi and battery connection status of all the dorms within the building.

Final Webflow Front-End Design:

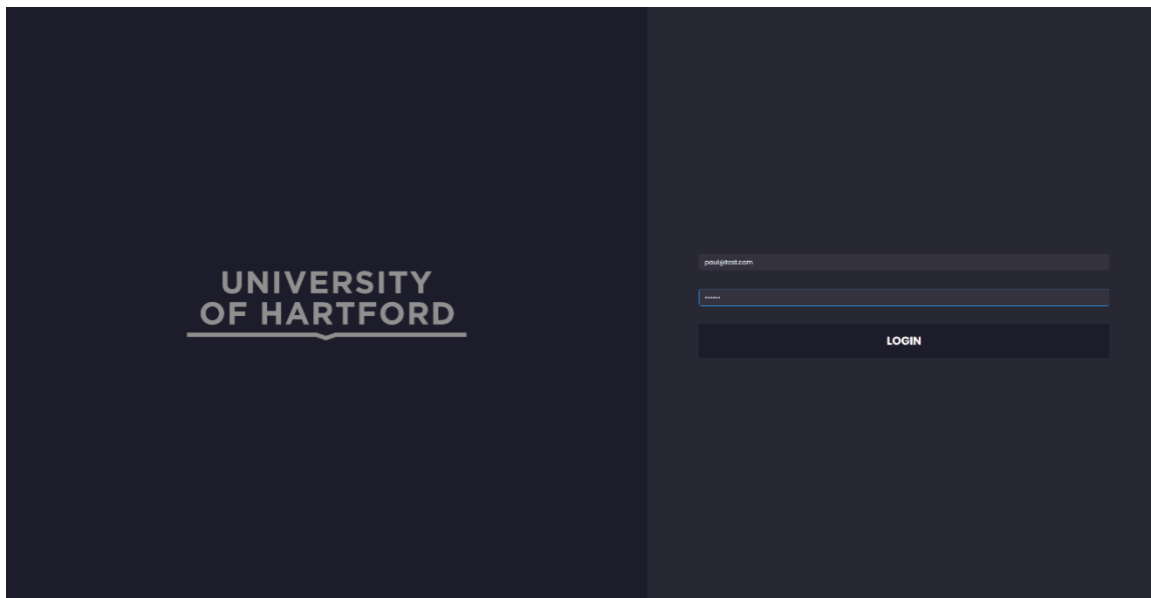


Figure 6.1.29: Webflow: Login Page

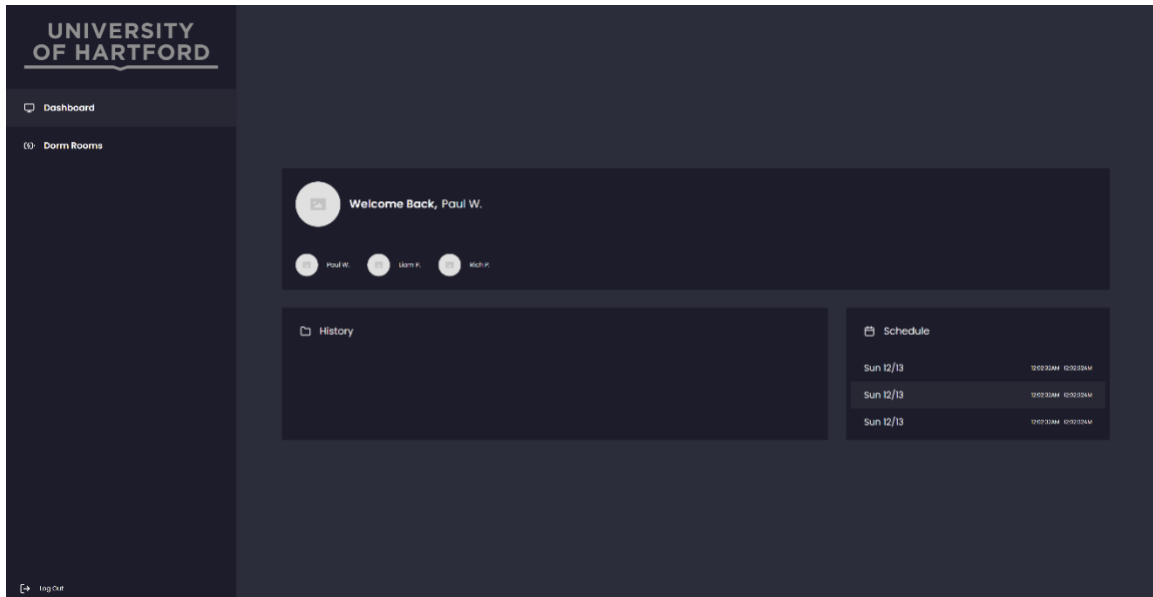


Figure 6.1.30: Webflow: Dashboard Page

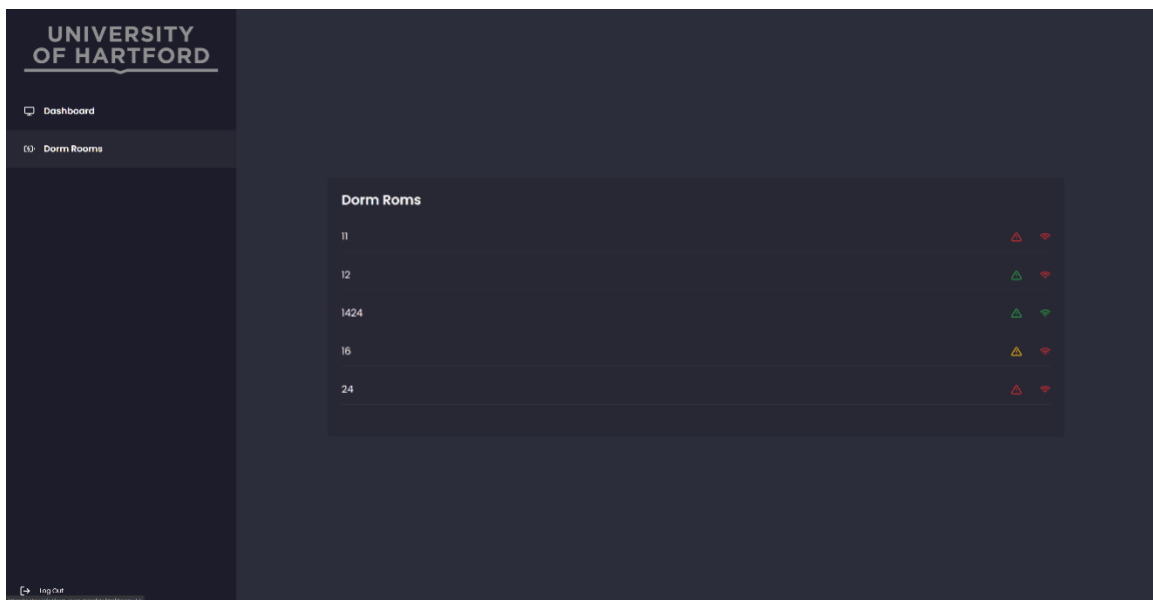


Figure 6.1.31: Webflow: Dorm Room Page



Figure 6.1.32: Room 1424 Activity Page

The screenshots above show the final revision of the dashboard. The team chose to optimize the dashboard by removing some of the unnecessary and repetitive tabs within the dashboard and adding an authenticated login.

The team was able to add an authenticated login page for RAs to be able to login securely with their own username and password. Seeing that the first Weblow run through had an activity, history, and health page, the team decided it was better to consolidate the three tabs into one tab, where in the final revision it is labeled as the “Dorm Rooms” tab shown above. Within the tab, the RA can live monitor all the dorm rooms within the building on that single page. When a specific dorm room is selected, the current noise and connection status is displayed on the top right corner of the page, along with a line chart showing you the decibel readings and time stamp.

6.1.3: Dashboard: Back End Design

The notify button on the bottom right-hand corner of room 1424’s activity page is used to notify the residents of that dorm by email and SMS messages when the dorm room is too loud, giving the residents a fair warning before there is an actual noise complaint.

```
//Login Authentication Using Firebase Username/Password

function login(){
  var email = document.getElementById("UNAME").value;
  var password = document.getElementById("PASSWORD").value;

  firebase.auth().signInWithEmailAndPassword(email, password)
    .then((userCredential) => {
      // Signed in
      var user = userCredential.user;
    })
    .catch((error) => {
      // ...
    })
}
```



```

        console.log('signed in');

        window.location = "index.html";
        // ...
    })
    .catch((error) => {
        var errorCode = error.code;
        var errorMessage = error.message;
        alert(errorMessage);

    });
}

//Log Off Authentication Using Firebase
function logOff(){
    firebase.auth().signOut().then(() => {
        // Sign-out successful.
        window.location = 'login.html';
        alert('Logged out successfully.');
```

```

    }).catch((error) => {
        // An error happened.
    });
}

firebase.auth().onAuthStateChanged(firebaseUser => {
    if(firebaseUser && window.location.pathname == '/login.html'){
        window.location = 'index.html';
    }else if(!firebaseUser && window.location.pathname != '/login.html'){
        window.location = 'login.html';
    }
});

```

Figure 6.1.33: Login.js: Authentication

Above is the JavaScript code written in the back-end design for the dashboard. Where the username and password are verified to authenticate the login, the username and password are both verified through firebase. Shown below.

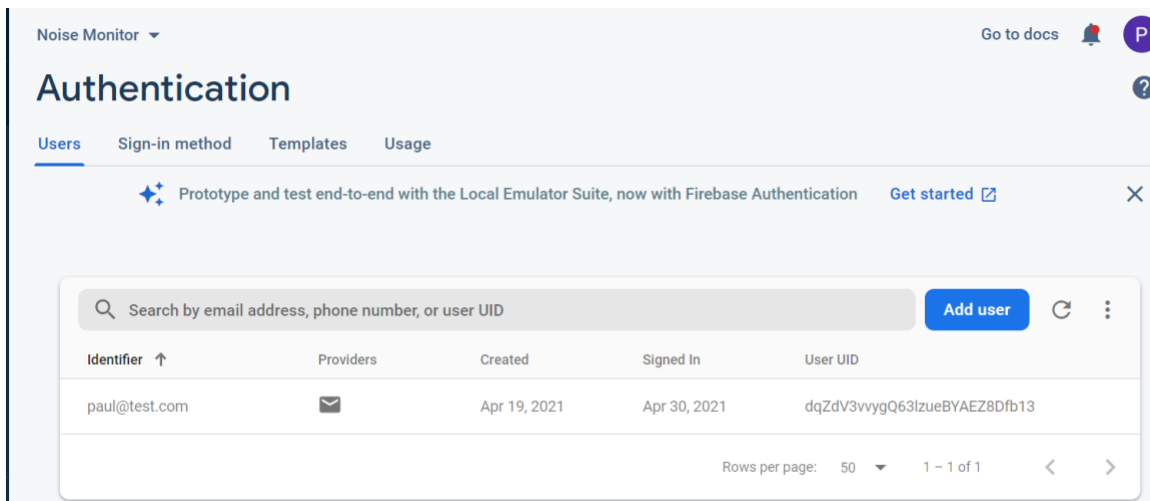


Figure 6.1.34: Webflow: Login Page

Chart JavaScript (Back-End)

```
//Chart JS
function createChart(data){
  var ctx = document.getElementById('myChart').getContext('2d');
  myChart = new Chart(ctx, {
    type: 'line',
    data: {
      // labels: [new Date('2019-01-01'), new Date('2019-01-
02'),new Date('2019-02-05')],
      datasets: [{
        label: 'My Line',
        data: data,
        borderColor: 'rgb(75, 192, 192)',
        backgroundColor:'rgb(75, 192, 192,0.5)',
        lineTension: 1
      }]
    },
    options: {
      responsive: true,
      title:{
        display:false,
        text:"Noise over time"
      },
      legend:{
        display:false
      },
      elements: {
        point:{
          radius: 0
        }
      },
      annotation: {
        annotations: [{
```

```

        type: 'line',
        mode: 'horizontal',
        scaleID: 'Y',
        value: 90,
        borderColor: '#E53030',
        borderWidth: 2,
        label: {
            enabled: true,
            content: 'Loud'
        }
    },
    {
        type: 'line',
        mode: 'horizontal',
        scaleID: 'Y',
        value: 70,
        borderColor: '#FFC107',
        borderWidth: 2,
        label: {
            enabled: true,
            content: 'Warning'
        }
    }
],
},
scales: {
    xAxes: [{
        type: 'time',
        display: true,
        ticks: {
            source: 'auto',
            maxTicksLimit: 24,
            fontColor: 'white'
        },
        scaleLabel: {
            display: false,
            labelString: 'Date'
        },
        gridLines: {
            display: false,
            color: 'white'
        },
    }],
    yAxes: [{
        display: true,
        id: 'Y',
        ticks: {
            fontColor: "white",
            suggestedMax: 95,
        },
        scaleLabel: {

```

```

        display: true,
        labelString: 'Decibels',
        fontColor: 'white'
    },
    gridLines: {
        display: false,
        color: 'white'
    },
    }
}
}
});
}

```

Figure 6.1.35: Webflow: Chart JavaScript Code

The JS code above is what the team designed to display the chart showing the noise levels along with the time stamp for each individual dorm room.

Where the function creates a line chart displaying the noise over time in decibels. Along with the line chart the JS is written to label both the x and y axis' and can display the two limit lines labeled "Warning" and "Loud". The noise over time is automatically averaged out from one time stamp to the next on the x axis with the given information sent over from firebase within the past 24 hours of the monitor being run.

Calendar Function

```

//Calendar Function
function updateDate(){
    room_alerts =[];

    var currentDate = $( "#datepicker" ).datepicker( "getDate" );
    var endDay = new Date(currentDate.getTime() + (24 * 60 * 60 * 1000));

    var docRef = firebase.firestore().collection("alerts")
    .where("room", "==", room)
    .where("time_stamp", ">", currentDate)
    .where("time_stamp", "<", endDay)
    .get()
    .then((querySnapshot) => {
        querySnapshot.forEach((doc) => {
            // doc.data() is never undefined for query doc snapshots
            var data = {
                x:doc.data().time_stamp.toDate(),
                y:parseFloat(doc.data().noise_level)
            };

            room_alerts.push(data);
        });

        if(myChart){
            myChart.destroy();
        }
    })
}

```

```

    createChart(room_alerts);
  })
  .catch((error) => {
    console.log("Error getting documents: ", error);
  });
}

```

Figure 6.1.36: Webflow: Calendar JavaScript Code

The function above was created so an RA using the dashboard can have and backtrack the noise and connection status history for each dorm.

The function queries the rooms data history from the firestore data base into the dashboards back-end. Where this information is already stored into firebase with the room number along with its noise levels and connection status of that day had already been stored into firebase. Once the data is brought over from firebase into the dashboards back-end it is then displayed onto the chart explained previously.

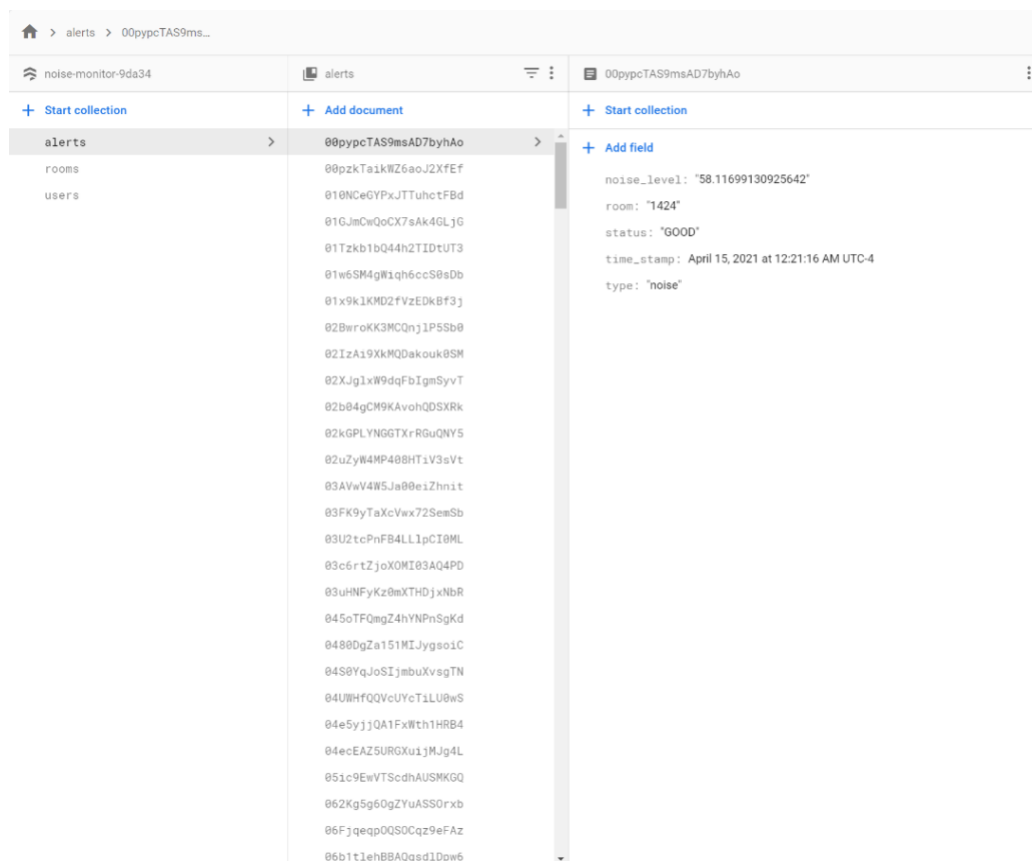


Figure 6.1.37 Firebase Alerts Section

6.1.4: Notification System Design

Notification Function and Flowchart:

As our project began to evolve throughout the semester, our team decided to write the notification system in JavaScript rather than using the Python programming language. One main advantage of using JS for the email/SMS system was that the noise monitor would only be focused on reading and sending noise levels in dB to the back-end using HTTP requests. Therefore, the back-end in Visual Studio in connection with Google Firebase allowed us to improve our notification system further.

The figure below indicates the three residents who currently live in room 1424 as shown in Google Firebase. The users section displays the student ID numbers of the residents followed by variables containing their information. The *email* and *phone_number* variables are used with the JavaScript code to ensure that the email/SMS message is sent to the correct occupant living in the dorm room.

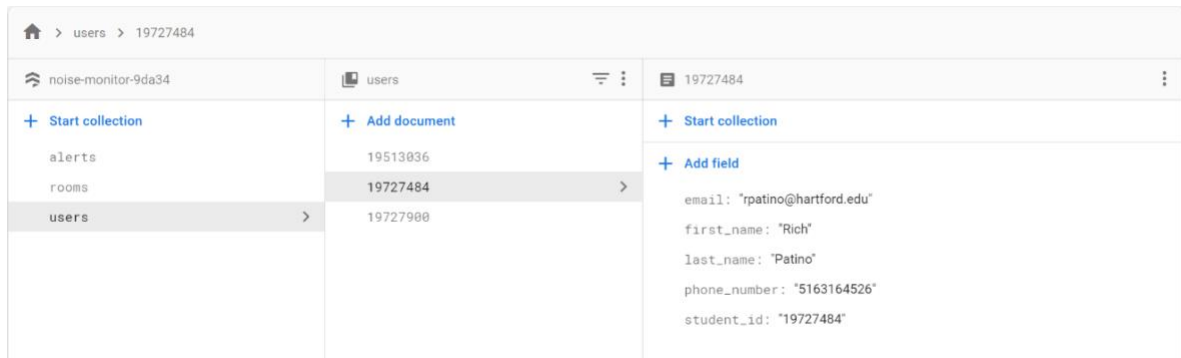


Figure 6.1.28: Webflow: Login Page

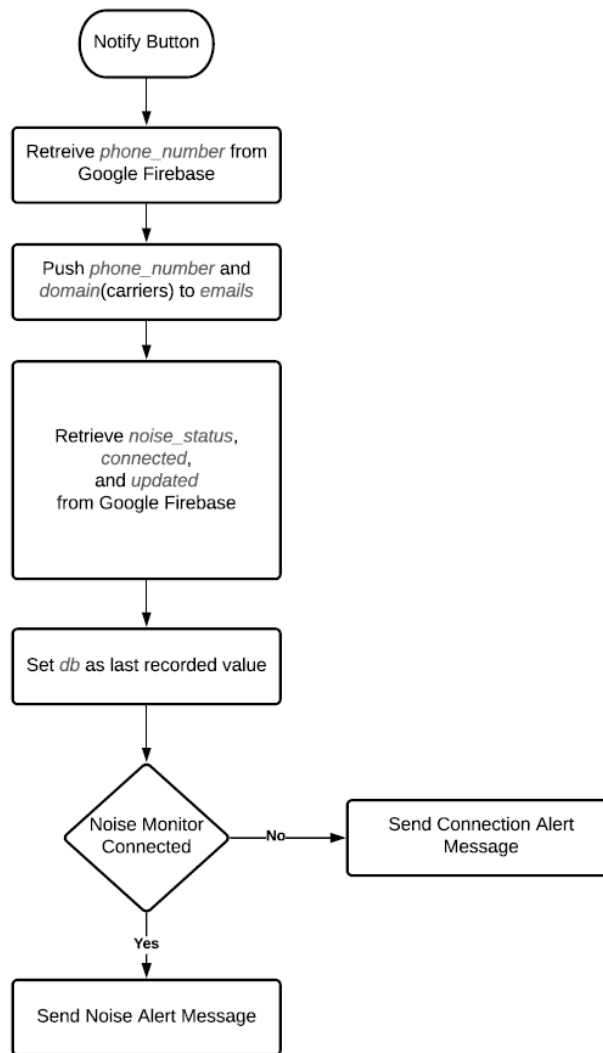


Figure 6.1.32: Notification System Flowchart

In our initial design for our SMS messages, a message was previously going to be sent to one phone number. The problem that we quickly realized was that the carrier of the resistant's cellular phone must have been known before sending any message. As shown in the updated flowchart, Google Firebase stores the phone number of the resident using the `phone_number` variable. Next, an `Array.push()` method is used to take the empty array of `emails` and combine the `phone_number` and `domain` variables to return an `email`. This feature makes it simpler when adding a phone number because if the university decides to add a phone number to Firebase they would not have to worry about the carrier because the notification system JS code takes care of that.

Next, the `noise_status`, `connected`, and `updated` variables were queried with Google Firebase. Our group also thought it would be best to include the decibel reading from the noise monitor into the email/SMS message. To do this, we created a variable called `db` to use the `Math.round()` function to round the

last_db_recorded value from Firebase. Lastly, as shown in the flowchart, a decision box was generated asking if the noise monitor was connected. An If/Else statement was created in JS stating if the noise monitor was connected, then the text message will display the current status. The else statement indicates that if the noise monitor is not connected, the body of the message will state that the current room number is not properly connected.

```
function generateEmails(data){
    let domains = ["@txt.att.net", "@tmomail.net", "@messaging.sprintpcs.com", "@vtext.com", "@vmobl.com"];
    let emails = [];

    for(i =0; i < domains.length;i++){
        emails.push(data.phone_number+domains[i]);
    }
    return emails;
}
//send email
function sendEmail(data){

    //get recipients
    let phone_emails = generateEmails(data);

    //get status
    let noise_status = roomData.status;
    let connected = roomData.connected;
    let updated = roomData.updated;

    //get decibels
    let db = Math.round(roomData.last_db_recorded);

    //Body
    var body;
    if(connected){
        //noise alert
        body = "ALERT! Room "+room+"\n Status: "+noise_status+"\n Last Decibel reading: "+db+"\n Last Updated: "+updated.toString();
    }else{
        //connection alert
        body = "ALERT! Room "+room+" is not properly connected. Please check connection.\n Last Status: "+noise_status+"\n Last Decibel reading: "+db+"\n Last Updated: "+updated.toString();
    }
}
```

Figure 6.1.33: Notification System JavaScript Code

The JavaScript code below displays where the message will be going and who the message is from. When sending a message through an email we decided to use the SMTP (Simple Method Transfer Protocol) server. The purpose of this server is to serve as a link between the sender and receiver so that a message can be sent successfully. When our team

enabled SMTP for the uhartreslife1@gmail.com account, it was critical that we enabled two-factor authentication followed by creating an “App Password”. Therefore, instead of inserting the real password for the email account in the “Password” field, the 16-digit app password was placed there giving the server permission to access the account. As previously mentioned, the “To:” field points to the email addresses of the residents stored in google firebase. The remaining section of this JavaScript code indicates the subject that will be displayed on each message and the appropriate body that will be included.

```
Email.send
({
  Host: "smtp.gmail.com",
  Username: "uhartreslife1@gmail.com",
  Password: "yynnufjqccvcjfyyc",
  To: data.email,
  Bcc: phone_emails,
  From: "uhartreslife1@gmail.com",
  Subject: "University of Hartford Residential Life",
  Body: body ,
})
.then(function (message) {
});
}
```

Figure 6.1.34: Notification System JavaScript Code Continued

Notification Button:

Once the residential assistant is certain that the resident is being too loud, they will click the *Notify* button on the room page of the dashboard. The function below is the JavaScript code that is ran once the residential assistant makes their final decision. First, *roomData.residents* is set to the variable *resident IDs* which runs a for loop from 0 until the amount of residentIDs is met. Within the for loop, the *users* (containing the student ID numbers) which is obtained from Google Firebase stores the data as an array, *residentIDs*. Therefore if the document in Google Firebase containing the resident IDs exist, then the JavaScript code will call the *sendEmail()* function and a pop on the screen will display: “mail sent successfully”. If the ID numbers of the residential students do not exist then there will be a pop up on the screen indicating: “No such document!”.

```
function sendNotifications(){
  let residentIDs = roomData.residents;
  for (i = 0; i < residentIDs.length; i++) {
```

```

        var docRef = firebase.firestore().collection("users").doc(resident
IDs[i]);
        docRef.get().then((doc) => {
            if (doc.exists) {

                setEmail(doc.data());

            } else {
                // doc.data() will be undefined in this case
                console.log("No such document!");
            }
        }).catch((error) => {
            console.log("Error getting document:", error);
        });

    }

    alert("mail sent successfully");

}

```

Figure 6.1.35: Notification Button JavaScript Code

Email and Text Message Output:

After the RA decides to notify the resident for being too loud, the resident will receive both an email and text message once the notify button located on the dashboard web application is clicked by the residential assistant. The first message on the figure on the left-hand side displays the text message alert for when the RA alerts the student that their current noise level in the room is “Good”. As mentioned previously, the SMS message will indicate the room number, noise status, last decibel reading, and time. These message indications were received by the variables stored within Google Firebase which were used when writing the JavaScript code in Microsoft Visual Studio. An identical message alert was sent to the student’s email (shown on the right-hand side) displaying the same information as what was shown in the text message alert. The second text message alert on the left-hand side displays what the resident may receive if the noise monitor is disconnected from Wi-Fi or if the device is disconnected from a power source. The message indicates that the device is “not properly connected. Please check connection” and displays the room number, last decimal reading and last noise status.

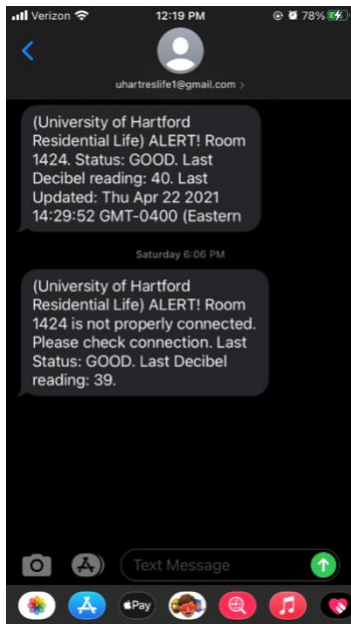


Figure 6.1.36: Text Message Output

[External]: University of Hartford Residential Life

 uhartreslife1@gmail.com <uhartreslife1@gmail.com> 4/22/2021 2:31 PM

To: Patino, Richard

****External Email****

ALERT! **Room 1424**. Status: GOOD. Last Decibel reading: 40. Last Updated: Thu Apr 22 2021 14:29:52 GMT-0400 (Eastern Daylight Time)

Figure 6.1.37: Email Message Output

6.2 Bill of Materials

Table 6.2: Bill of materials

Item	Description	Quantity	Price (per unit)	Purchase/Made	Vendor	Design Done	Ordered/Made	Delivery Date
1.0	Noise Monitor							
1.1	Raspberry Pi Zero W	1	\$10.00	P	Amazon	X	X	Fall 2020
1.2	DMM-4026-B-I2S-EB-R I2S Mic	1	\$3.32	P	Digikey	X	X	Spring 2021
1.3	LEDs	3	\$0.36	P	Amazon	X	X	Fall 2020
1.4	IR Sensor	1	\$1.95	P	Adafruit	X	X	Spring 2021
1.5	First Alert PR700 Smoke Detector Enclosure	1	\$14.90	P	Amazon	-	X	Spring 2021
2.0	Dashboard/Server							
2.1	Front End	-	NA	M	-	X	X	Spring 2021
2.2	Back End/Server	-	NA	M	-	X	X	Spring 2021
2.3	Notification System	-	NA	M	-	X	X	Spring 2021
	Total	\$30.53						

Shown above is our groups bill of materials. As indicated by the total, our project came in significantly under budget. While this was not our intent from the onset, as the project evolved, we were able to keep the cost of our components minimal with the use of digital processing. The enclosure itself represents the bulk of the cost, and could certainly be manufactured for cheaper, if a design was made up.

7. Testing Methodology and Results

7.1 Testing Methodology

As part of verifying that our project meets all the requirements listed in Section 3, testing procedures for measuring the accuracy of the noise monitor as a Type-2 SLM (sound level meter) were implemented through software and acoustical tests. While the IEC-61672 standards were not accessible within the confines of our project, our team used two standard testing procedures to verify the devices accuracy. Separate tests were also implemented to test end-to-end connectivity and overall system function.

7.1.1 Frequency Response

The first major test conducted was a test of the overall frequency characteristics of the sound level meter program, with specific considerations being taken in accounting for the a-weighted frequency response. As laid out by the IEC-61672 standard, the noise monitor was tested with 94dB SPL (−26dBFS) sinusoids at frequencies ranging from 20Hz-20kHz. Since testing within a laboratory setting was unavailable, testing was instead conducted within a Python program.

After synthesizing each signal with a fixed length of 1 second, each test signal was inputted into the final SPL algorithm. This process can be seen in Figure 3.1 below where short-term SPL readings are computed, and then averaged over a period of 1 second. The results of this test are shown in Table **TODO**.

```
for sin in que:
    spl = []
    for i in range(375, len(sin)):
        sig_chunk = squaresum(sin[i-375:i])
        short_spl = 0S + ref_1k + 20 * np.log10(sig_chunk/mic_ref)
        rms = sig_chunk/2**23
        normalized_amp.append((20*np.log10(rms)))
        spl.append(short_spl)
    print(".", end = "")
    avg = avg_db(spl)
    shorts.append(avg)
    i+=1
```

Figure 7.1.1: Computation of LAEQ values for sinusoidal waves spanning frequencies of 20-20kHz.

In addition to the testing done digitally, frequency response testing was also conducted acoustically. A separate test program was written to gather acoustic sound pressure levels, check them against expected values, and write the test results to a CSV file. This testing was conducted in the Dana audio studios in talent room 1. This allowed for an acoustic noise floor well below that of a typical room and minimized the effect of room modes on the testing.

The setup for the acoustic testing was as follows. A simple powered speaker was placed in a sealed room with both the noise monitor and a reference SPL meter positioned ~150mm from the speaker's low frequency driver. While the devices were coupled together, each microphone was placed at a distance of 10mm from each other. Each

device was set to read in values in dBA SPL with a time constant of 1 second, typically referred to as a ‘slow’ response.



Figure 7.1.2: Testing setup for both frequency response and linearity. Reference microphone used was a type-2 SLM (MicW i437L). A cardboard mounting system was employed to hold both microphones in close proximity.

The testing suite was then run with frequencies ranging from 20-20kHz. Each sinusoidal wave was outputted through the speaker and verified to be at 94dB using the reference SPL meter. Due to the limitations of the speaker itself, frequencies of below 63Hz and above 16kHz could not be produced. These frequencies bands were omitted entirely.

7.1.2 Linearity

Testing was also conducted to evaluate the noise monitors linearity. To evaluate linearity, a fixed frequency sinusoidal wave of 8kHz was input into the setup listed in Section 7.1.1. Rather than change the frequency, the test called for producing the same sine wave over a range of SPL values. This, of course, tests whether the microphone can accurately output a SPL reading over its full dynamic range. For the microphone selected for our project, the dynamic range spanned from 30dB-120dB SPL. The mapping of those sound pressure levels to both dBFS (full scale) and integer values are shown below in Figure

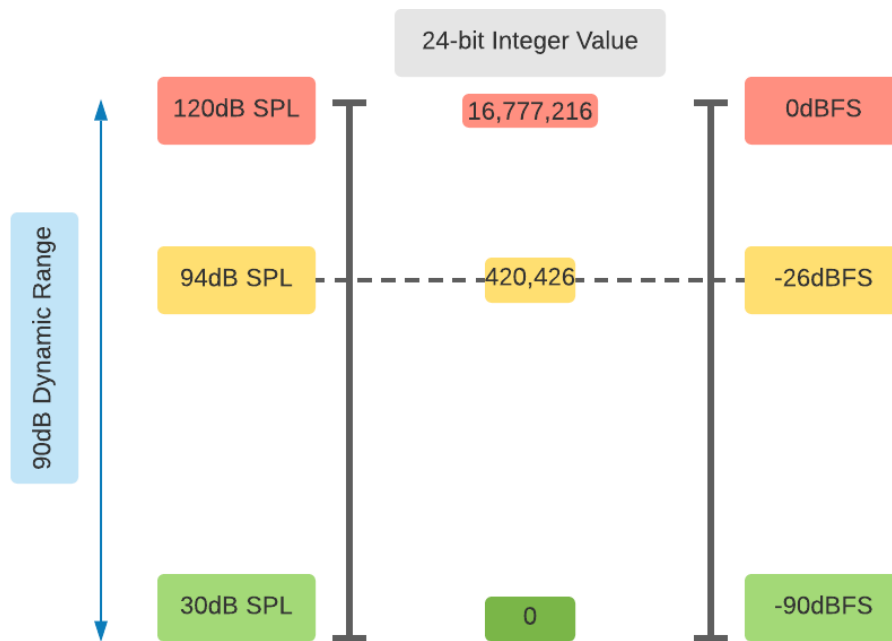


Figure 7.1.3: Mapping of acoustic SPL values to 24-bit integer values and full-scale decibels (dBFS).

7.1.3 End-to-End Testing: Notification System

Prior to sending an email and text message through JavaScript as described in section 6.1.3, our team initially decided to send an alert message to an RA. Through our previous use case, sending an email alert and text message to the residential assistant was indeed tested to make our project work in an end-to-end functionality test. The python code displayed below (Shown in Figure 7.1.5), the code that was tested for send an email and SMS to an RA by sending the message to multiple recipients. This feature was done by including `msg['to'] = ",".join(to)` which was responsible for combining the email or phone number string and assigning them to the variable `to`.

```
#function is created
def email_alert(subject, body, to):
    msg = EmailMessage()
    msg.set_content(body)
    msg['subject'] = subject
    msg['to'] = ",".join(to) #used for multiple recipients

    user = "uhartreslife@gmail.com" #your email
    msg['from'] = user
    password = "ngcagjbxsyalmgge"

    server = smtplib.SMTP("smtp.gmail.com", 587) # This will start our email server
    server.starttls() # Starting the server
    server.login(user, password) # Now we need to login
    server.send_message(msg)

    server.quit() #quit server

# If this main program is ran then alert is sent.
if __name__ == '__main__':
    to = ["rpatino@hartford.edu", "6753489015@txt.att.net"] #send to multiple addresses
    email_alert("University of Hartford Residential Life", "ALERT! Regents Park N204B STATUS: LOUD!\n", to)
```

Figure 7.1.5 – Initial Notification Python Code

The figure below displays the initial text message output that our team decided to send. The message alert includes the name of the university's residential life followed by an alert message indication the room number and location, and lastly the noise status. To make our notification system more useful and informative, our new design of the notification system include detailed information within the messages (Discussed in Section 6.1.4).



Figure 7.1.4 – Initial Test Message Output from Python Code

7.2 Results

7.2.1 Sound Level Meter Testing Results

Results from the testing procedures above were beyond what was expected. The linearity of the device through a range of 74dB was confirmed. Frequency response for almost every band was within the tolerances listed by the IEC, with only two frequency bands deviating beyond that. Because manufacturer data states that the microphone unit-to-unit consistency is +/-1dB across a frequency range of 20Hz-20kHz, it was concluded that widespread implementation of our noise monitors was feasible. With some extra equalization applied, or a change in the sound level meters mechanic mounting, our project could fully meet type-2 SLM standards. Based on our digital test of the system, we believe that even at maximum deviation the response would also fall within allowed ranges.

In Table 7.2.1, the digital frequency response test results are shown. The deviation within each band was consistent enough that an offset of -0.4 dB would push all bands deviation further towards zero-deviation.

Table 7.2.1: Digital frequency response tests conducted. Each band is weighted and thus needs to be offset by the amount defined by the filter.

Frequency (Hz)	Frequency Weighting (dB)	Read Level (dB)	Expected Level (dBA)	Deviation (dBA)	Type 2 Upper Tolerance (dBA)	Type 2 Lower Tolerance (dBA)	Pass/Fail
20	-50.5	43.95	43.5	0.453	3.5	3.5	PASS
40	-34.6	59.76	59.4	0.361	2.5	2.5	PASS
80	-22.5	72.08	71.5	0.578	2.5	2.5	PASS
100	-19.1	75.34	74.9	0.438	2	2	PASS
125	-16.1	78.32	77.9	0.416	2	2	PASS
250	-8.6	85.81	85.4	0.412	1.9	1.9	PASS
500	-3.2	91.23	90.8	0.435	1.9	1.9	PASS
1k	0	94.35	94	0.351	1.4	1.4	PASS
2k	1.2	95.70	95.2	0.498	2.6	2.6	PASS
4k	1	95.44	95	0.443	3.6	3.6	PASS
5k	0.5	95.03	94.5	0.529	4.1	4.1	PASS
8k	-1.1	93.35	92.9	0.449	5.6	5.6	PASS
10k	-2.5	92.03	91.5	0.530	5.6	NA	PASS
12.5k	-4.3	90.27	89.7	0.572	6	NA	PASS
16k	-6.6	87.74	87.4	0.337	6	NA	PASS
20k	-9.3	85.12	84.7	0.417	6	NA	PASS

Table 7.2.2 reflects the acoustic tests conducted with regards to frequency response. While two bands fell outside of their allowed deviation, a simple filter could be implemented to correct this. Frequency bands below 63Hz and above 16kHz could not be tested, as the speaker used could not reproduce such frequencies. Figure 7.2.1 shows the acoustic results in graphical form.

Table 7.2.2: Acoustical testing data collected. Tolerances as well as overall deviation are indicated.

Frequency (Hz)	Read Level (dBA)	Expected Level (dBA)	Deviation (dBA)	Type 2 Upper Tolerance (dBA)	Type 2 Lower Tolerance (dBA)	Pass/Fail
63	81.32	94	-12.68	2.5	2.5	FAIL
80	95.27	94	1.27	2.5	2.5	PASS
100	95.57	94	1.57	2	2	PASS
125	93.44	94	-0.56	2	2	PASS
160	92.91	94	-1.09	2	2	PASS
200	92.62	94	-1.38	2	2	PASS
250	93.51	94	-0.49	1.9	1.9	PASS
315	94.82	94	0.82	1.9	1.9	PASS
400	92.61	94	-1.39	1.9	1.9	PASS
500	92.45	94	-1.55	1.9	1.9	PASS
630	93.02	94	-0.98	1.9	1.9	PASS
800	92.10	94	-1.90	1.9	1.9	PASS
1k	94.96	94	0.96	1.4	1.4	PASS
1.25k	93.98	94	-0.02	1.9	1.9	PASS
1.6k	92.63	94	-1.37	2.6	2.6	PASS
2k	92.67	94	-1.33	2.6	2.6	PASS
2.5k	92.60	94	-1.40	3.1	3.1	PASS
3.15k	94.25	94	0.25	3.1	3.1	PASS
4k	93.13	94	-0.86	3.6	3.6	PASS
5k	95.16	94	0.66	4.1	4.1	PASS
6.3k	98.18	94	4.18	5.1	5.1	PASS
8k	97.66	94	3.66	5.6	5.6	PASS
10k	99.28	94	5.28	5.6	NA	PASS
12.5k	100.89	94	6.89	6	NA	FAIL
16k	98.96	94	4.96	6	NA	PASS

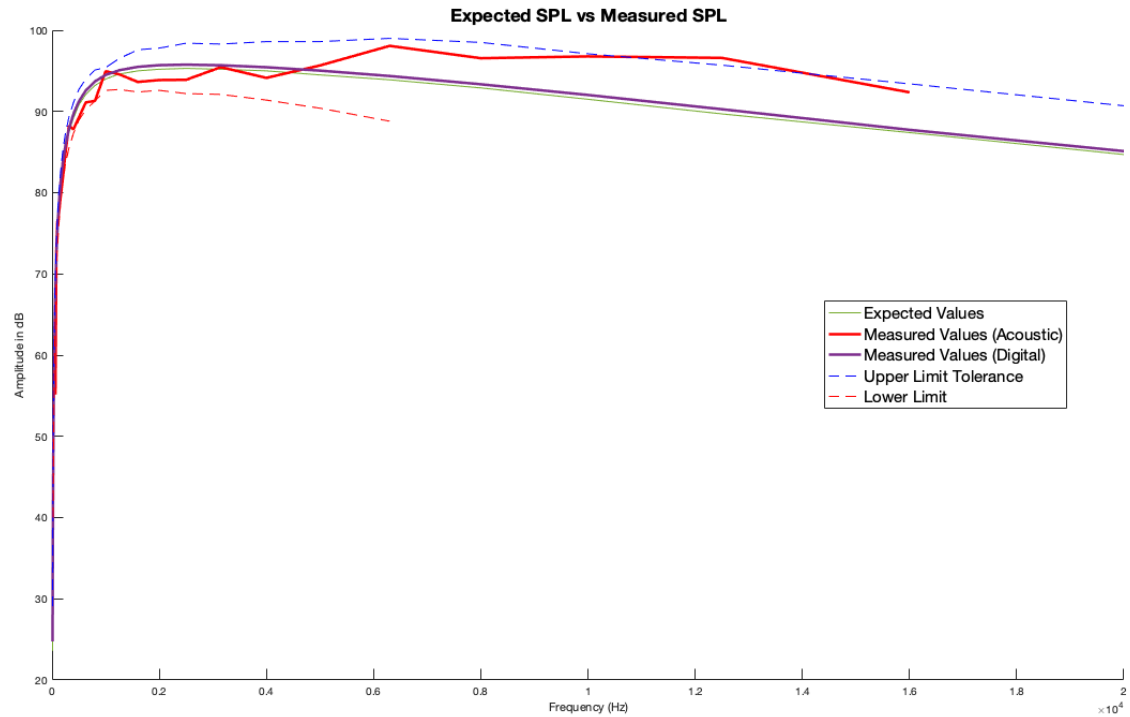


Figure 7.2.1: Graphical representation of acoustic testing results. The upper tolerance limit for Type-2 SLMs is shown with a blue dashed line, while the lower limit tolerance is indicated by a red dashed line. Areas where these lines end indicate that there were no upper/lower limits defined.

Table 7.2.2: Acoustic linearity testing results. Bands above 116dB SPL failed this test and deviated beyond the +/- 1.5dB tolerance allowed for a type-2 instrument.

Applied SPL (dBA)	Read Level (dBA)	PASS/FAIL
40	41.97	PASS
49.8	51.02	PASS
55	56.19	PASS
60	61.63	PASS
65	61.38	PASS
70	71.44	PASS
75	76.04	PASS
80	81.05	PASS
85	86.36	PASS
90	91.97	PASS
95	96.24	PASS
100	100.75	PASS
105	106.35	PASS
110	111.17	PASS
115	114.14	PASS
116	114.64	FAIL

117	114.64	FAIL
119	114.64	FAIL
120	114.64	FAIL

Our linearity test confirmed that our device maintains a relative linearity in a range of 85dB, from 40-115dB. While the device failed tests above 115dB, such a range is not needed within the scope of our project. Readings below 40dB could not be taken, as the noise floor of the testing place was just above 36dB.

7.2.2 End-To-End Results

As a team we can say that our project was successful and fully implemented by creating a functional live dashboard web application, created an enclosure that includes a Raspberry Pi Zero W, microphone, IR sensor, and 3 LEDs that all act as the noise monitor, and a working notification system that sends both an email and text message to the resident once the residential assistant decides to notify the resident

7.2.3 Noise Monitor Enclosure Results

Designing an enclosure was very important for the practicality of the noise monitor because our team wanted all the required components to be hidden from the resident living in the dorm room. A physical requirement for the enclosure was that it must be able to be installed on any surface. Therefore, our smoke-detector shell was a perfect match to fit our requirement because its hard plastic material and mounting plate makes the device easy to setup. As shown in Figure 7.2.4 below, several cutouts were done to accommodate for the three LED indicators, the AC power cord, microphone, and the infrared sensor.



Figure 7.2.2 – Final Noise Monitor Enclosure

7.2.5 Dashboard Web Interface Results

The team was able to add an authenticated login page for RAs to be able to login securely with their own username and password. Seeing that the first Webflow run through had an activity, history, and health page, the team decided it was better to consolidate the three tabs into one tab, where in the final revision it is labeled as the “Dorm Rooms” tab shown above. Within the tab, the RA can live monitor all the dorm rooms within the building on that single page. When a specific dorm room is selected, the current noise and connection status is displayed on the top right corner of the page, along with a line chart showing you the decibel readings and time stamp.



Figure 7.2.3 Room 1424 Status Page

7.2.6 Notification System Results

To complete our end-to-end result of the noise monitor system that is meant to be installed in a dorm room within a universities resident hall, a notification alert to the resident was required. It was concluded that once the “Notify” button on the dashboard was clicked by the RA, a fully detailed text message and email would include information such as the name of the university, the room number, noise status, last decibel reading, the time the message was sent, and finally the last decibel reading that was recorded from Google Firebase.

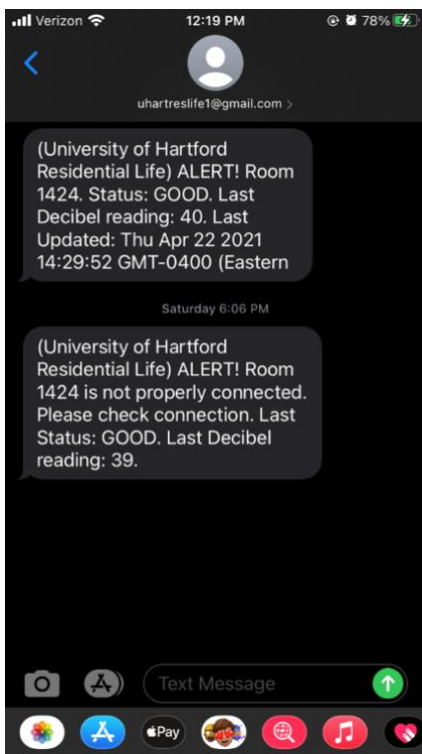


Figure 7.2.4: Text Message Output

[External]: University of Hartford Residential Life

 **uhartreslife1@gmail.com** <uhartreslife1@gmail.com> 
4/22/2021 2:31 PM

To: Patino, Richard

****External Email****

ALERT! Room 1424. Status: GOOD. Last Decibel reading: 40. Last Updated: Thu Apr 22 2021 14:29:52 GMT-0400 (Eastern Daylight Time)

Figure 7.2.5: Email Message Output

8. Attributes and References

8.1 Attributions

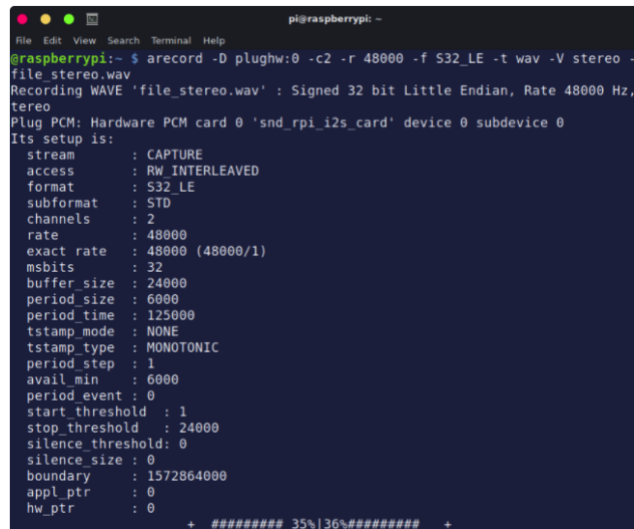
8.1.1 Sound Level Meter Attributions

As mentioned in Section 6, several attributions can be made for our design of the noise monitor. One such attribution is a paper published on sound level meter algorithms, that was used in development of our program [9]. It detailed all of the equations used for calculating acoustic measurements, and also referenced the design of digital filters for frequency weighting. The equation actually implemented is shown in the following figure.

$$L_{Aeq,T} = 10 \lg \left[\frac{\frac{1}{T} \int_{t_1}^{t_2} p_A^2(t) dt}{p_0^2} \right] dB; \quad T = t_2 - t_1,$$

Figure 8.1.1: Equation used for calculation of weighted equivalent noise levels. This formula was implemented directly in various revisions of our code.

As mentioned in the previous sections, the utilization of an installation guide from Adafruit Industries [6] allowed our Raspberry Pi to be interfaced directly with the I2S microphone. Through modifications to the ALSA device drivers and the creation of a ‘dummy’ soundcard, I2S support was successfully added to our project. This was a major part of our overall success and without it would have required the use of a different microprocessor. A snippet from the end of the installation process is shown in the following figure for added clarification.



```
pi@raspberrypi: ~
File Edit View Search Terminal Help
raspberrypi:~$ arecord -D plughw:0 -c2 -r 48000 -f S32_LE -t wav -V stereo -v
file_stereo.wav
Recording WAVE 'file_stereo.wav': Signed 32 bit Little Endian, Rate 48000 Hz, Stereo
Plug PCM: Hardware PCM card 0 'snd_rpi_i2s_card' device 0 subdevice 0
Its setup is:
  stream      : CAPTURE
  access      : RW_INTERLEAVED
  format      : S32_LE
  subformat   : STD
  channels     : 2
  rate        : 48000
  exact rate  : 48000 (48000/1)
  msbits      : 32
  buffer_size : 24000
  period_size : 6000
  period_time : 125000
  tstamp_mode : NONE
  tstamp_type : MONOTONIC
  period_step : 1
  avail_min   : 6000
  period_event : 0
  start_threshold : 1
  stop_threshold  : 24000
  silence_threshold: 0
  silence_size    : 0
  boundary        : 1572864000
  appl_ptr        : 0
  hw_ptr          : 0
+ ##### 35%|36%##### +
```

Figure 8.1.2: Adafruit made install of an I2S driver for Raspberry Pi. Figure shows the commands for recording an audio file with a specified format.

While none of the team members were acoustics/mechanical majors, some significant research was done in mounting the MEMS microphone within its housing. Gasketing material was used as an acoustic sealant and served the purpose of creating an acoustic channel with a fixed resonance and isolated the microphone from the rest of the components. A document put out by Infineon was used in reference for this step of our project. It detailed mechanical designs for a MEMS microphone that produced the most favorable acoustic responses.

The following figure is a diagram from said paper showing the mechanical design we implemented for our prototype. Notable in this design is the use of an acoustic sealant, and a fine mesh to protect from dust and other environmental conditions.

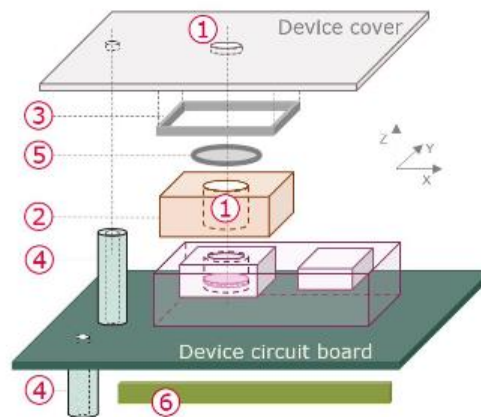


Figure 8.1.3: Diagram used to construct the physical mounting of the MEMS microphone in its enclosure. The acoustic gasket is denoted by 2, while the mesh screen is denoted by 5.

8.1.2: *Chart.js*

There were multiple sources the team had for incorporating the line chart used to track the noise levels in dB throughout the night. The entire back-end of the noises.info dashboard was developed in JavaScript, the team needed to hardcode this line chart or look at multiple resources and work off those attributions. Using stack overflow the team was able to find similar code for our specific use case, where the time stamp was along the x axis and edited the y axis to the decibel readings. Having those resources to look at separately along with documentation on the chartjs.org website, we were able to manipulate a line chart into the one the team is using to live monitor each room.

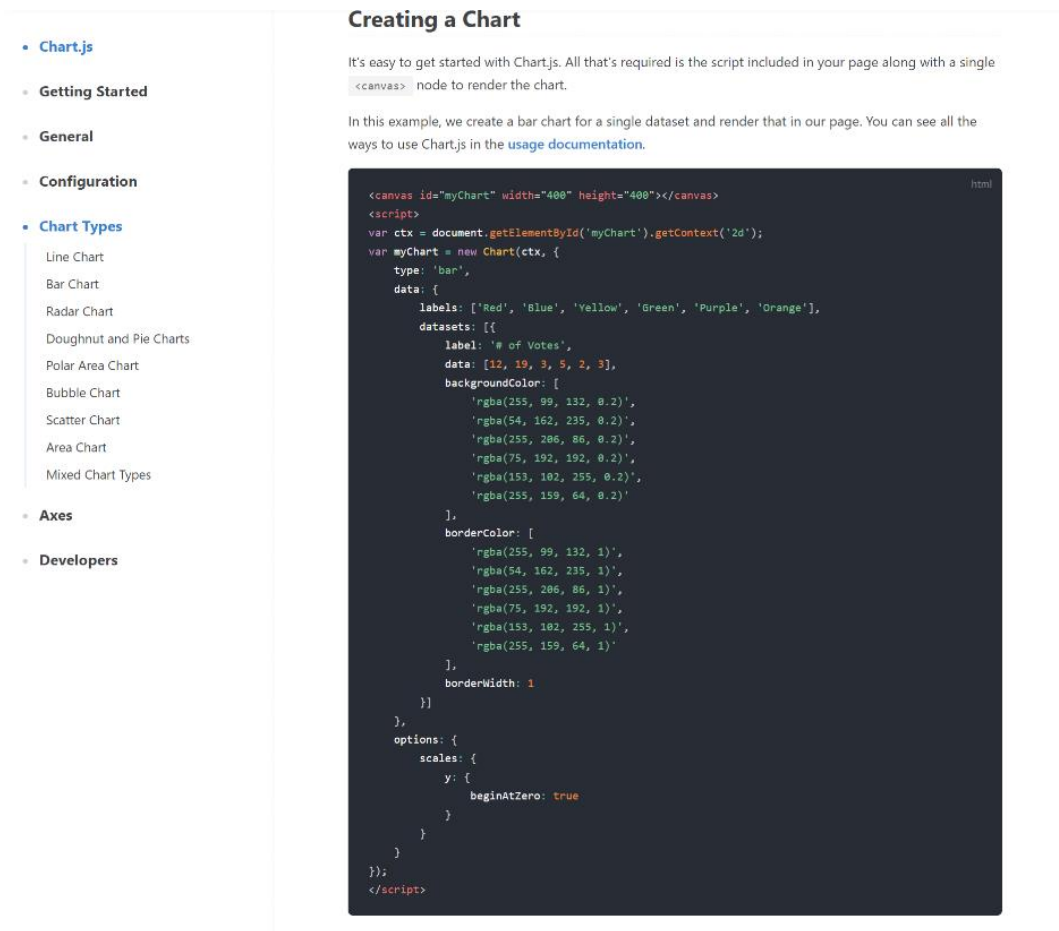


Figure 8.1.4 Chart.js.org Documentation

8.1.3: Notification System Attribution

When designing our notification system, a difficult feature to incorporate was sending the message alert to a phone through the phone numbers carrier. Our team decided to research attributions on sending an SMS to a phone using any programming language [19]. To complete this, the proper SMS gateway was needed. As mentioned in section 6, the JavaScript code was modified to accommodate all carriers, resulting in the host to just enter the resident's phone number into Google Firebase. Figure 8.3.2 below shows the most common carrier SMS gateways.

SMS Gateways for each Carrier

- AT&T: [number]@txt.att.net
- Sprint: [number]@messaging.sprintpcs.com or [number]@pm.sprint.com
- T-Mobile: [number]@tmomail.net
- Verizon: [number]@vtext.com
- Boost Mobile: [number]@myboostmobile.com
- Cricket: [number]@sms.mycricket.com
- Metro PCS: [number]@mymetropcs.com
- Tracfone: [number]@mmst5.tracfone.com
- U.S. Cellular: [number]@email.uscc.net
- Virgin Mobile: [number]@vmobl.com

8.2 References

- [1] <https://www.sciencedirect.com/topics/engineering/sound-level-meter>
- [2] https://ec.europa.eu/health/scientific_committees/opinions_layman/en/hearing-loss-personal-music-player-mp3/1-3/2-sound-measurement-decibel.htm
- [4] https://www.onosokki.co.jp/English/hp_e/whats_new/SV_rpt/SV_7/sv7_5.htm
- [5] https://www.hartford.edu/currentstudents/_files/6.5_current_students_student_handbook.pdf
- [6] <https://cdn-learn.adafruit.com/downloads/pdf/adafruit-i2s-mems-microphone-breakout.pdf>
- [7] <https://flask.palletsprojects.com/en/1.1.x/>
- [8] <https://www.who.int/docstore/peh/noise/Comnoise-1.pdf>
- [9] http://www2.arnes.si/~ljc3m2/igor/doc/rep/02_06_sound_level_meter_algorithms.pdf
- [10] https://www.who.int/occupational_health/publications/noise6.pdf
- [11] <https://medium.com/oceanize-geeks/firebase-realtime-nosql-database-c8b30eba2220>
- [12] <https://firebase.google.com/docs/hosting>
- [13] <https://www.educative.io/edpresso/what-is-firebase>
- [14] <https://firebase.google.com/docs/database>
- [15] <https://firebase.google.com/pricing>
- [16] <https://firebase.google.com/docs/auth>
- [17] <https://www.verypossible.com/insights/bluetooth-vs.-wi-fi-for-iot-which-is-better#:~:text=Wi%2DFi%20has%20a%20maximum,numerical%20values%20from%20IoT%20sensors.>
- [18] <https://www.chartjs.org/docs/latest/>
- [19] <https://dev.to/mraza007/sending-sms-using-python-jkd>

9. Reflections and Conclusions

9.1 Greater Impact

Our project tackles the issue of noise pollution on college campuses. Being one of the most vulnerable groups with respect to mental wellbeing, college students are especially susceptible to the effects of noise pollution. Whether studying or recharging after a long day of classes, students benefit from a quiet environment. Oftentimes this is not possible due to excessive noise from neighboring dorms. Our project aims to tackle this in a way that both respects student's privacy and increases their wellbeing.

One design decision made to ensure privacy was the use of HTTPS requests as a way of connecting the noise monitors to the dashboard. In the early stages of the project, a concern was raised that the noise monitors could be listened in on. This concern drove our overall design in the direction of secure communication. The HTTPS protocol requires that a handshake be made between the dashboard and the noise monitor, ensuring that data is not sent to the correct place.

Another design decision made to combat this privacy issue was the use of a 'blocking' mode of sampling. By using non-continuous sampling of audio, only 8ms of audio data are captured at one time. These samples are then immediately used to compute a root mean square value and discarded. The next block of samples is then read in and so on, ensuring that only 8ms of audio data would be accessible at any given time.

The greater impact of our project can be expanded from our use case. The need for low-cost noise monitoring in cities and buildings is a growing trend. With the push to urbanization in the United States and elsewhere, noise pollution has become a problem. It is a marker of public health and because of this, our project would be a viable option for measuring public wellbeing. It could be used in various settings, both outdoors and indoors as a way of cheaply and quickly giving towns, cities, and states the ability to collect valuable information on public health.

9.2 Knowledge Gained

Throughout the past two semesters, so many components of the project were new to all of us on the team. All of us having different engineering concentration backgrounds we were not familiar with the same things, which obviously helped, along with us having to do our own research on a lot of the components for this noise monitor project.

Referring to section 4.1 where the Risk Analysis is shown, the main components to our projects were listed above. Where neither of the team members had much knowledge or experience with IoT projects in general, Wi-Fi/Bluetooth implementation, mounting and packing of a device like this and dB microphone implementation.

Given that those were just about half of the projects main components the team had to do further investigation on each of those components. The entire team had to contribute equal investigation on IoT projects in general, then the Wi-Fi/Bluetooth implementation component was then brought up throughout the research. For this IoT project the team chose to go the Wi-Fi route with the noise monitor, seeing that Raspberry Pi being used already had the functionality in there and the connection was more fitting for the project, the team was able to quickly narrow down which would be used.

When comparing both the advantages and disadvantages of each connection Wi-Fi made more sense in the use case of the noise monitor. Wi-Fi speed being a major advantage to Bluetooth where while Bluetooth has a maximum speed of only 3 Mbps, Wi-Fi has a maximum speed of at least 54 Mbps, over 18 times as fast of a connection giving the dashboard a much faster and accurate response to the readings. Bluetooth for most IoT projects would be sufficient, Wi-Fi can add an additional layer of security using certain protocols that cannot be used for Bluetooth such as WPA, WEP, WPA₂ and WPA₃.

Range being the last important advantage, Wi-Fi typical has a much bigger range than Bluetooth would, typically Bluetooth have ranges no more than about 33 feet, where Wi-Fi range depends on a few factors, such as antenna type, frequency, transmission power, and how many routers are in use, the University of Hartford already has Wi-Fi implemented throughout the entire university include the dorms where the monitors will be in use.

Although the entire group was familiar with filter design and implementation, the group chose to go the digital filtering route, where initially the group believed it to be an easy enough task to finish fast, later realizing that it took would take a big part of the investigation along with most of the time allotted for the sound level meter implementation.

Specifically, this design element required knowledge of acoustic measurements, algorithmic design, and knowledge of digital filtering. This was something that we investigated over the course of the fall and into January. Since none of our team members had any experience with filtering in the digital domain, it was a challenge to learn how to design and use a filter in our program. In the end, the team was able to overcome this and design an IIR filter with the characteristics needed.

9.3 Conclusion

Our project addressed the problem of noise pollution on college campuses. Specifically, it was designed and developed for the University of Hartford, with the goal user being residential assistants. Our project grew to involve the development of an interactive webpage where residential assistants can view and monitor current and past noise levels across any number of rooms. Features of the final webpages design included: live updated graphs, secure login for staff, the ability to view previous noise levels on a day-by-day basis, and the implementation of a notification button that automatically sends out an email and SMS message directly to residents.

Apart from the design implemented for the frontend of the website, a backend design was also constructed to allow its full functionality. The webpages back-end included the ability to store and access previous data: including residents' names, ID numbers, contact information, room numbers, noise levels, and timestamp information. This involved the integration of various functions and programs with the user interface of the webpage.

To fully implement this design, the development of a noise monitor was also completed. Its design featured digital processing: including filtering, logarithmic computation, and various other signal processing techniques. Based on the Raspberry Pi Zero W and a digital I2S microphone, the noise monitors were able to meet most of the requirements needed to be classified as a Type-2 sound level meter. Overall cost per unit was kept under \$33, ensuring the projects' ability to be implemented cost-effectively on a large scale.

While our project was implemented fully, there are few addition improvements that could be made. Specifically, additional work could be done to improve the accuracy of the noise monitors to fully fit within the definitions of a Type 2 sound level meter.

Working with the Raspberry Pi Zero W, which features only one processor, a real time implementation was not fully realized. While utilizing a blocking system of sampling audio provided the necessary accuracy, implementing the program in real-time on a processor such as the ESP32 would further improve accuracy. In addition, more design could be done with respect to the mounting of the MEMS microphone within the noise monitors housing. With proper acoustic design implemented for the MEMS sound channel, we could reduce the resonance observed above 14kHz.

In addition, added features could also be implemented using a multicore design. Due to the I2S protocol featuring two channels, an additional microphone could be implemented so noise localization could be performed. This could serve to locate whether the sound was coming from an adjacent unit. Additionally, analysis of the audio coming into the system could be performed to notify campus public safety of events associated with break-ins such as shattering glass and gunshots.