Wearable Health Information and Tracking (WHIT)

Final Report Submitted To:
Professor Rose Gomar & Professor Carlos Rossa

April 10th , 2024

Authors:
NAHREEN TAHMID
ARYAN MATHUR
ARYAN LAXMAN SIROHI

Table of Contents

Table of Figures

List of Tables

1. Introduction

1.1. Project Purpose and Significance

Our project intends to assist individuals in monitoring the health and safety of their dependents by introducing an inexpensive and standalone health monitoring device designed specifically for their care. We have aptly named our device, based on the motivations and objectives, the W.H.I.T.; it stands for Wearable Health Information and Tracking.

In this report, we will delve into the various aspects of health monitoring and its importance. We will present the features, benefits and potential impacts of our device. We will highlight how it can streamline health monitoring for busy individuals and provide peace of mind knowing that their dependents' health is being watched over, even in their absence [1].

2. Background

2.1. Growing Importance of Health Monitoring

In an age where the pace of life is faster than ever before, the importance of health monitoring cannot be overstated [1]. Our health monitoring device is designed with young adults in mind, to empower them with the tools they need to ensure the well-being of their dependents.

The well-being of our loved ones, particularly our children and elderly family members, is a matter of great importance. As young adults, we are entrusted with the responsibility of those who depend on us, and their health is a responsibility we cannot afford to overlook. Health monitoring is the cornerstone of preventive care, ensuring that potential health issues are detected and addressed before they escalate into more significant problems [2].

Wearable devices have become integral to modern healthcare due to their developing impact on patient monitoring and overall well-being [3]. These compact and user-friendly gadgets present real-time insights into vital signs and activity levels, enabling healthcare professionals to provide personalized care and early intervention. By bridging the gap between healthcare providers and patients, wearables enhance the quality of healthcare delivery, encourage better health choices, and contribute to a more efficient and patient-centered healthcare system.

2.2. Current Devices and Functionalities

Many people are finding wearable health devices to be helpful technology. These devices are designed to be worn on the body and can monitor various aspects of an individual's health and well-being. They are lightweight and comfortable to wear, making them convenient for everyday use. These devices often connect to smartphones or other devices to provide users with real-time data and insights [4]. Common examples of wearable health devices include fitness trackers (e.g., Fitbit), smartwatches (e.g., Apple Watch), and medical wearables (e.g., steady Electrocardiogram (ECG) monitors) [5].

Individuals can use wearable health devices to keep track of their well-being and potentially catch health issues early. To ensure that one is getting the most out of these devices, it's important to select one that aligns with your personal health goals and seek advice from a healthcare professional [6].

3. Motivation

The motivation to build a better product primarily stems from the current shortcomings in the market. The primary targets of our critique will be the Apple Watch and Google Fitbit. Complimentary motivations will also be stated that overlap with existing functionalities in the Apple Watch SE and Google FitBit.

3.1. Problem #1: Cost of Comparable Devices

The cheapest entry point into the Apple ecosystem for a wearable health-tracking device is $329 for the Apple Watch SE [7]. This wearable also harbors a requirement to already own an Apple device potentially increasing the overall cost further.

Similarly, the cheapest entry point with the line of Google's Fitbit is at 130 dollars while operating at reduced functionality until a subscription is purchased [8].

3.2. Problem #2: Lack of Standalone Functionality

Neither the Fitbit or Apple Watch offers complete standalone functionality built into their wearable experience, both devices require a supporting device such as a mobile device or laptop to be connected to the wearable, and is only able to synchronize the device information to their data display platform once that device is in some predefined Bluetooth range. In the capacity of a wearable meant to be set up once and used thereafter, there is a very limited selection range on the market.

3.3. Problem #3: Poor Battery Performance

The battery life of the Apple Watch SE is rated to be up to eighteen hours in normal use [9]. The battery life of the Luxe 2 from Google is rated to be up to 4 days of normal use, however, there have been numerous complaints about the batteries used in Fitbits and their overall reliability [10].

3.4. Problem #4: Fair Use and Contextual Information

The Apple Watch SE and Fitbit Luxe 2 both also lack a comprehensive educational experience for the end-user. The wearables and their onboarding experiences rely on an existing understanding of the metrics and their respective significance. According to a paper published by PLOS Digit Health, they state "contextual information" as an area of concern with modern health-oriented wearable devices [5]. The Author of the paper, Stefano Canali, also goes on to state,

"wearable technology as a crucial tool for the remote and constant monitoring of the elderly and patients that need to practice social distancing, avoid hospital visits but require monitoring. Looking at current figures on the adoption of wearable devices, members of the population that fit into these categories are severely underrepresented and excluded by the application of this technology."

It is clear in this paper that fair use of these devices is a large issue in today's market. Fair use and distribution should be a larger cornerstone in the ongoing development of health monitoring wearables.

4. Objectives

1. **To develop a *standalone* wearable device:** Our main aim for this project is to create a wearable health monitoring device that can collect, display and synchronize data without the need for a supporting device to be around at all times. This device will use advanced technology and sensors to continuously gather and analyze health information. Our goal is to build a device that is easy to use and can help people keep track of their health more conveniently and accurately.

2. **To provide *real-time* health data to users**: In this project, we want to ensure that users get real-time health information. As the device collects data, users will be able to see their health information

immediately. By achieving this goal, we hope to empower users to make better decisions about their health and respond quickly to any issues.

3. **To enhance *user awareness* of their health status:** Another important goal of our project is to help users better understand their health. We plan to create user-friendly displays and features that show health data in a simple and easy-to-understand way. This will not only let users know how they're doing health-wise right now but also encourage them to pay more attention to their health and make healthier choices in the long run. To ensure that the health tips and information presented is correct and reliable, the team has consulted and cross-checked multiple reliable sources of information such as World Health organization (WHO), National Institutes of Health (NIH) to provide evidence-based information on different health fields.

4.1. Solutions to Current Problems

Solution #1: Reduced Cost

Throughout this project, we have used many different components and tools to develop the final product. Keeping in mind the first problem, as stated in Section 3, we have aimed to use cheap and readily available technologies. Using this approach, we have achieved a low-cost and durable product.

The final product boasts a 3D-printed casing made of durable Onyx material (which requires a Markforger printer). This material is a fusion of nylon and carbon fiber, making it as strong as Acrylonitrile Butadiene Styrene (ABS) and allowing for the printed items to be stiff, strong and smooth [12]. Similarly, the microcontroller, sensors and other devices used in the project implementation are cheap and readily available on Digikey for purchase. Furthermore, the integration of the parts uses open-source libraries and components to reduce overhead research and development costs.

Solution #2: Complete Standalone Device

In order for the device to operate as a standalone device, these set of objectives must be met:

- The device must be able to operate without the assistance of another person. The user should be able to use the device as intended without facing difficulties with the interface.
- Must have an initialization process between a primary device and the wearable to communicate vital information such as Wi-Fi credentials
- The device must be able to communicate vital information to users without a secondary device being nearby
- The device must be capable of performing basic functionality for extended periods of time without the requirement of a ‘synchronization’ with the supporting device

In order for our device to be able to operate with the presence of a secondary device such as a phone or laptop, Wi-Fi and Bluetooth functionalities are imperative. Bluetooth should be used in the first time setup in order to communicate vital information such as the Wi-Fi SSID and Password, alongside basic user metrics for onboard algorithms.

Comparable wearables are unable to stand alone and communicate metrics without a secondary device because they lack the ability to directly upload information. Wi-Fi is required to update the end-user of any alerts and/or changes to biometrics as without it, communication to any server cannot be established.

Solution #3: Efficient Battery Life and Performance

Optimizing power consumption involves integrating energy-efficient hardware and implementing software components that extend battery life and support longer use of the device without recharging. Additionally, we developed a low-power mode that selectively collects vital information, ensuring better battery utilization and an extended device life for improved user experience.

Solution #4: Increased Accessibility and Health Literacy
When developing the application and its supporting User Interface (UI), we integrated a panel regarding sensor readings' related health tips and knowledge. This was created with the intent to teach users about what certain metrics mean and how they can interpret them to better understand and control their physical well-being.

Note: It is important to acknowledge that accurate retention and proper use of these tips is not guaranteed and that everyone may interpret and use them differently. As people's minds are subjective, it's hard to ensure that everyone understands and follows the information provided exactly as intended. So, while our app aims to help, it can't guarantee that all users will use the tips in the same way.

5. Design Requirements

5.1. Functional Requirements

Note: All mentions of 'user' are associated with individuals who have created an account and linked their application with the watch. The interpretation of functional and non-functional requirements is based on the following reference, [13].

- The device should measure **Heart Rate** and **Blood Oxygen** using the Pulse Oximeter sensor.
- The device should **alert** the user of **abnormal vital values** and send supporting notifications and pop-ups through the application and device UI.
- The device should **alert** the user of a '**Fall Detected**' and send supporting notifications and pop-ups through the application and device UI.
- The device should provide **GPS location** updates and alert the user if they step outside a pre-set boundary. If such a motion is detected, the user should be alerted of the last reported location.
- The database should implement **AES encryption** and authentication mechanisms to safeguard sensitive health data stored & transmitted by the device to ensure **data security** and privacy.
- Improve user **health awareness** and health literacy through the use of the product and its supporting application.

5.2. Non-Functional Requirements

- Measure and update vital **Heart Rate and Blood Oxygen** information from the sensors every 10-15 seconds with 80% accuracy and reliability.
- Ensure **alerts** are sent to the user within 90 seconds of the measurement with 80% accuracy and reliability.
- Measure and update the **GPS** location every 5 minutes with 90% accuracy.
- Ensure the device **battery** life of at least 8 hours with 90% reliability.
- Ensure that data security and physical **product safety** regulations are met.
- Increase the **affordability** of the product to a wider target audience.

In the next few chapters, we will be looking into the different phases of the product's design in detail. Phase 1 will cover how the team decided on certain components and the selection process behind it. We will discuss the factors that were considered when selecting the various components for the product. Phase 2 will take a closer look at the initial development of the components and application. We will also further expand on the implementation of Bluetooth using the devices. This phase plays a crucial role in the development of the product and will provide a look into the technical aspects of the project. Phase 3 talks about the new design iteration and development of the new changes made. We will also take a look at the initial stages of the user interface. This phase is important as it lays the foundation for the final product. Phase 4 is the last in the series and talks about the final prototype design, the last stages of the user interface and application development. We will also cover how the product and the application communicate. This phase is critical as it marks the end of the design process.

6. Phase One: Initial Design

6.1. Goals

This iteration aims to develop a foundation for the project to build on for future enhancements.

1. System Design

The first key goal for this iteration is to plan out how the entire system will work at a high level through a deployment diagram. This aids in clearly defining and understanding the entire system so that the implementation process can be focussed.

2. Hardware Selection, Tools and Technologies

Another key goal is to select all of the required hardware as well as the tools and technologies that are necessary for implementation. The selection process will also involve explanations for why certain design choices were made.

3. Hardware Exploration

After baseline hardware components are selected, it is important to develop basic functionality with each component in order to solidify their place within the project. This came in the form of creating a simple codebase and basic connections to better test our hardware suite.

4. Initial User Interface

Continuing the planning phase of the project, the final significant goal for this iteration is to plan out the user interface of the mobile application through a mock-up in order to get a visual representation of how the user interface is intended to look and feel.

6.2. Implementation

In order to understand how this iteration was developed, tasks were divided into 3 building blocks that made up the outcome for this phase of implementation.

6.2.1. System Design

The first critical part of the planning phase was to get a basic idea of how the system will function with a deployment diagram which is shown below in Figure 6.2.

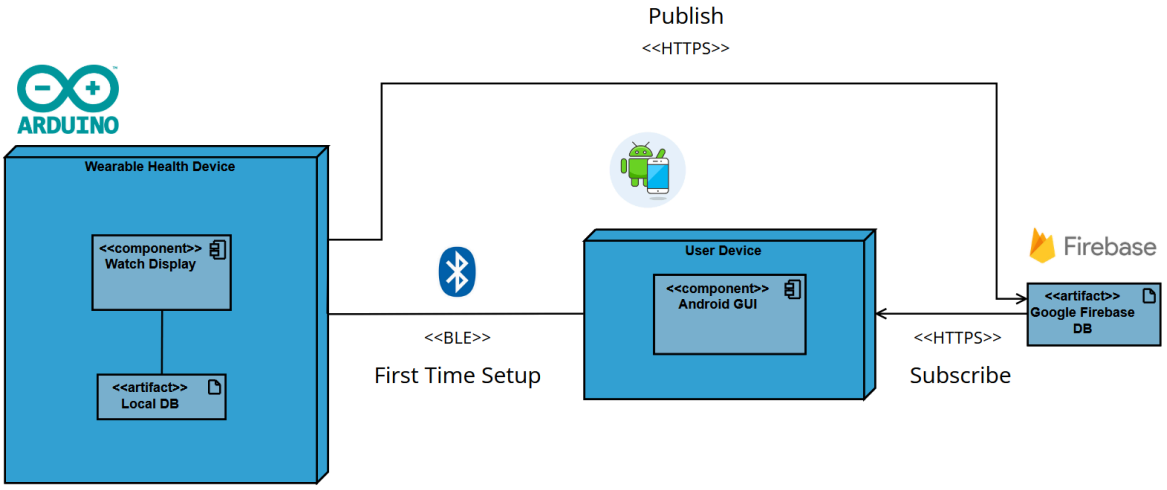


Figure 6.1: Deployment diagram showcasing high-level system architecture

The deployment diagram in Figure 6.1 gives a simple overview of the different nodes in our design as well as briefly illustrating the data flow for the collected user data. The device and the phone application nodes of the system are essential for functionality as the Arduino-powered device contains the sensors required for user health metrics, and the application is required to display the collected information in a concise package that can be shown to users anywhere.

For the mobile application, it is evident from Figure 6.1 in the central node that an Android application was selected as the platform to develop the application on. This was due to a number of factors, the first of which being that Android has a 70% global market share [14], which means that most people around the world use Android devices, therefore it makes sense to cater to the majority audience. Furthermore, native iOS development is restricted to only Apple computers and learning a brand new cross-platform framework such as Flutter or React Native introduces significant overhead that would cost significant time and resources. Therefore, it was simpler to develop on the Android platform which is solidified even further due to the fact that Android devices come with a JRE (Java Runtime Environment) which allows for Java to natively run on Android devices which simplifies development further due to the object-oriented nature of Java.

The third node in the deployment diagram is Google Firebase which enables the device to publish its data over HTTPS (Hypertext Transfer Protocol Secure) to the cloud which can be accessed from anywhere securely. Therefore, allowing authorized users to access the devices data from anywhere is extremely flexible as authorized users do not have to be near the device in order to monitor the device's parameters. This node is critical for the standalone objective of our device thus it is vital to be present in the deployment diagram. Alternatives such as MongoDB, AWS (Amazon Web Services), or Microsoft Azure were considered, however Amazon and Microsoft use a pay-as-you-go model where implementing our code would require making payments as resources are consumed. Google Firebase was free to start with and had very generous limits of their free tier. Firebase also includes far more services for free such as Authentication, a Real-Time Database, a longer term database called Cloud Firestore, and Firebase Functions which allows for backend functions to run for a small cost. Therefore, due to the plethora of features and low cost, it was more feasible to pursue implementation on Google Firebase.

In addition, the mobile application and the device follow a publish-subscribe model which is evident in the deployment diagram as the device publishes sensor data to the cloud database and the phone is able to use the same HTTPS protocol to retrieve the information that was published by the device, resulting in near real-time updates of current device data.

Finally, in order to cover the use case where the device loses an internet connection and is unable to publish its data to the Google Firebase cloud database, it will locally store as much data as possible until it gets an internet connection back, at which point it will publish the locally stored data in batches to the cloud database for the application to read the updated sensor data.

Once the overall system architecture is defined, the strategic hardware selection process can begin in order to identify the ideal components that would meet all functional requirements for our device.

6.2.2. Hardware Selection, Tools and Technologies

Microcontroller Selection

Since the hardware features entirely depended on which microcontroller was used, it was critical to select a microcontroller that supported all requirements for the hardware. Key factors included diverse I/O (Input/Output) support for connecting external devices such as a pulse oximeter. LiPo (Lithium Polymer) battery support, portable size and display support were additional specifications that needed to be satisfied. The size constraint was determined by using the average wrist size to ensure that the device would fit on most people.

The objective approach used to streamline the deciding process was to perform a weighted trade study to determine the microcontroller that would successfully accomplish the goals stated. Weighted trade studies work by defining categories that are the most important to satisfy and assigning a weight to them. Then a rating is assigned by comparing with the other options and multiplying the rating by the weight and then summing that result for all categories gives an overall percentage. The percentage represents the percentage of the categories that are covered by each device. So for the Arduino Nano BLE (Bluetooth Low Energy) Sense 67.5% of the project's requirements are covered by that device. Table 6.1 shown below displays the result of the weighted trade study and it is evident that two microcontrollers received the same score of 67.5% which were the Arduino Nano BLE Sense and the RP2040 MCU Board.

Category	Weight	Score Scale	Arduino Nano BLE Sense	Arduino Nano ESP32	Adafruit Flora V3	RP2040 MCU Board
Built-in Hardware	30	4 = Most Features 1 = Least Features	4	3	1	3
Cost	20	4 = Least Cost 1 = Highest Cost	1	2	4	3
Design	20	4 = Most In Favor 1 = Least In Favor	1	1	3	3.5

Extendibility	10	4 = Most Durable 1 = Least Durable	3	3	2	3
Documentation	20	4 = Most Documentation 1 = Least Documentation	4	4	2	1
Total (%)	**100**		**67.5%**	**65%**	**57.5%**	**67.5%**

Table 6.1: Weighted Trade Study for Microcontroller Selection

Upon completing the weighted trade study, it was determined that beginning implementation on both boards at the same time would provide more first-hand insight into the boards and a better selection would be made based on the experience gained. While this increased initial costs, it was important for the team to trial and test which board would work better for this project's application, and unfortunately this aspect was not determinable through research.

Pulse Oximeter Selection

The MAX30102 was a popular choice among similar projects as there was support with documentation and libraries in order to implement functionality rapidly which made this an easy choice.

Figure 6.2: MAX30102 Pulse Oximeter

GPS Module Selection

The GPS selected was the FGPMMOPA6B which was selected primarily due to the size constraint as other options had antennas for increased accuracy at the cost of losing space in the device. Other alternatives for the GPS such as the MIA-M10 were compact but required external antennas and making hardware connections would have been very difficult as contact points were small and required soldering, whereas the FGPMMOPA6B enabled rapid prototyping due to the fact that it resided on a development board right out of the box.

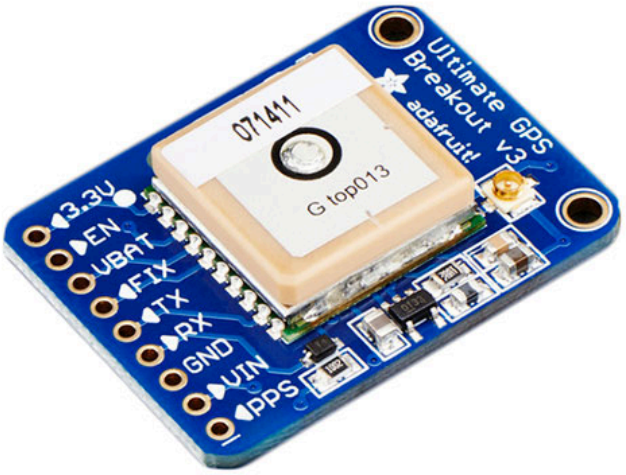


Figure 6.3: FGPMMOPA6B GPS Module

Battery Recharger Module Selection

One of the final hardware selections was the battery recharger board as there was no interface present on the Arduino to recharge a LiPo battery. Furthermore, the decision to choose a recharger board came from the fact that Arduinos and every microcontroller require a regulated 3.3V to function normally, otherwise they will receive too much or too little power depending on the battery. This is because LiPo batteries output an unregulated voltage of 3.7V at full charge which would overpower microcontrollers. This voltage gradually declines as the battery discharges, therefore a circuit is required to consistently supply 3.3V until the battery is unable to provide any power. Furthermore, recharger boards provide an output that allows for a connection to the microcontroller in order to read the current battery level.

Battery Selection

The LiPo battery itself was the remaining component but they are standardized so the only deciding factor for the battery was how much capacity was necessary in order to power the device for a minimum of 8 hours as defined by the functional requirements. For initial development, the microcontroller was powered over USB (Universal Serial Bus). This allowed for testing power consumption in order to identify the ideal battery capacity needed to meet the requirement for battery life while also adhering to size constraints. In addition, the health and safety of the user is highly important therefore we made sure to select batteries with built-in battery protections in addition to an external recharger board that allows for a robust and safe battery-operated device.

Figure 6.4: LiPo Battery

Tools and Technologies

There are a number of tools and technologies that were used to develop the entire system, and this section will detail all of the tools that we used for this iteration. From the cloud provider for the backend of the system to the development environment that was used to develop code for the application and the Arduino.

Firstly, the Arduino IDE (Integrated Development Environment) is used to develop and upload code to the Arduino and luckily both the Arduino Nano BLE Sense as well as the RP2040 and many other popular boards have support for the software to write and upload code to the boards. Integrated Development Environments are interfaces used to streamline the coding process as they contain features such as code autocompletion, syntax highlighting, built-in debugging, version control plugins for tools such as Github and many more features.

Android Studio is another IDE that is used to streamline application development, with code editors that show a preview of the application, and additional features such as emulators which can simulate the application functioning in a virtual environment. USB debugging is also a useful feature in Android Studio that uploads the application code to a physical Android device to allow for testing on an actual device and monitor behavior on real hardware.

Github is used to maintain the version history of code as this project was foreseen to have quite a large code base, it is important to maintain previous history and incrementally develop as necessary. Github is one of the most widely used tools that makes it easy to incrementally develop programs with several developers and a single code base.

Google Firebase has an online portal that provides a central interface to interact with all of Firebase's services in one easy-to-use platform. Upon project creation, an API (Application Programming Interface) key is generated and that needs to be imported locally into the Arduino and application code.

6.2.3. Hardware Exploration

Arduino Nano BLE Sense Exploration

Rationale: As outlined in our trade study, a decision was made to begin development into both microcontrollers. The priorities of the development in regard to the Arduino was to attain a better understanding of the Arudino's environment, capabilities, and available libraries/examples. While the RP2040 yields a higher potential in terms of its overall capability and featureset, it was necessary to have a back-up in the event the RP2040 did not meet the requirements of either the project or assisting documentation to aid in further development.

During this period, an emphasis was placed on developing baseline functionalities such as understanding how to utilize the Arduino's bluetooth chip, built-in IMU, and corresponding I/O.

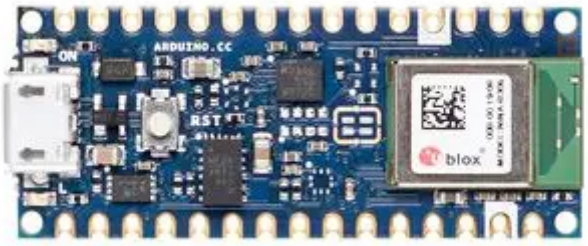


Figure 6.5: Arduino Nano BLE Sense

BLE Overhead: Development with the BLE chip of the Arduino was the first priority. Without a guarantee that data received from the sensors could be sent over, further development on any part of the project was a risk. Before development could begin, an overhead cost was incurred in relation to learning the fundamentals of how bluetooth transactions are conducted.

An assortment of online guides and tutorials was used in pursuit of a complete understanding for implementing BLE technology into our project. These were the conclusions:

Overview:

- BLE communicates through two types of devices: Central and peripheral
    - Peripherals advertise their presence
    - Central devices scan for and initiate connections with nearby peripherals

Service and Characteristics:

- A characteristic is a predefined location where a type of data is stored
    - It has a unique 32-bit long Universally Unique Identifier (UUID)
- A service is a collection of characteristics
- Once a central device discovers these characteristics it can:
    - Write information to the characteristic
    - Request information from the characteristic
    - Subscribe to updates

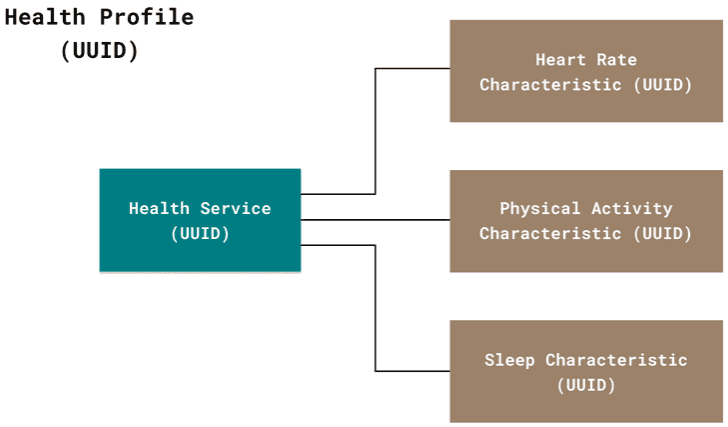


Figure 6.6: Sample BLE GATT Table design[7]

Data Exchange:

- Data can be exchanged three ways
    - *Reading*: occurs when a central device asks the peripheral device for specific information about a characteristic
    - *Writing*: occurs when a central device writes specific information into a peripheral devices
    - *Notifying*: occurs when a peripheral device offers information to the central device using a notification

Utilizing the fundamentals of BLE and various example code snippets, a basic C++ file was created for communication with other devices that support BLE. A code snippet is attached below:

```
BLEService healthService(HEALTH_SERVICE_UUID);

BLEIntCharacteristic heartRate(HEART_UUID, BLERead);
BLEIntCharacteristic physicalActivity(PHYSICAL_UUID, BLERead | BLENotify);
BLEIntCharacteristic sleep(SLEEP_UUID, BLERead | BLENotify);

void setup() {
  BLE.setLocalName(“ArduinoBLE");
  BLE.setAdvertisedService(healthService);

  healthService.addCharacteristic(heartRate);
  healthService.addCharacteristic(physicalActivity);
  healthService.addCharacteristic(sleep);

  BLE.addService(healthService);
  BLE.advertise();
}
```

Figure 6.7: BLE characteristic definition code snippet in Arduino

Through this code, a connection was established between the Arduino and a BLE testing app on a Samsung device. The app enabled the testing of reading from and writing to characteristics on the Arduino.

RP2040 Microcontroller Exploration

Rationale: Concurrently with the development of the Arduino, we also initiated work on the RP2040 to ensure flexibility in case the Arduino implementation did not meet our spatial requirements. The rationale behind this decision was to maintain agility in our project's evolution. In the event of space constraints, having the RP2040 as an alternative platform provided us with the opportunity for seamless transitions and iterations.

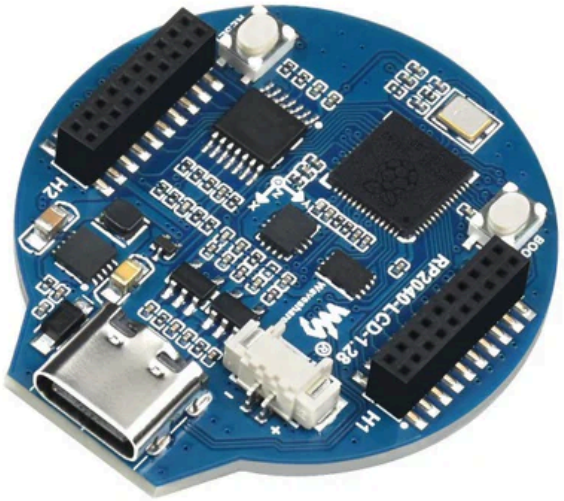


Figure 6.8: RP2040 Microcontroller Board

Beyond the capabilities of the Arduino, the RP2040 offered a significant advantage with its integrated battery recharger board, which effectively saved substantial space within our design. This feature not only addressed potential spatial limitations but also enhanced the overall efficiency and portability of the device.

Furthermore, the inclusion of the GC9A01 LCD display module directly on the RP2040 board was a game-changer. This addition opened up exciting possibilities for our project, as it allowed for the seamless integration of an advanced user interface. With this LCD display module, our project could potentially evolve into offering users a sophisticated and intuitive interface to complement the diverse range of functionalities we aimed to incorporate. This enhancement significantly broadened the scope and potential applications of our project, positioning it for greater versatility and usability in various contexts.

User Interface Overhead: In order to harness the on-board LCD overhead had to be taken in regard to learning the fundamentals of display engineering. This overhead consisted of key aspects such as display drivers, bitmap graphics, refresh rates, and other related concepts.

A large focus was placed on understanding display drivers, which serve as intermediaries between the RP2040 chip and the display panel. The TFT library serves as a crucial bridge providing a comprehensive suite of functions and utilities for controlling various aspects of the display. Through extensive experimentation and calibration, proficiency was gained in utilizing its feature set to manipulate individual and groups of pixels, rendering images, and managing display refresh rates.

Specifically, the library was used in large-part to draw geometric shapes, text, and images onto the display utilizing display cache. Through these objects, a rudimentary UI was developed as a placeholder in the scenario of further development in this direction.

Challenges: In our project's microcontroller selection process, the RP2040 initially held promise as a potential candidate, despite its placement as the second choice due to concerns regarding documentation availability. However, these concerns proved significant as we encountered limitations in accessing comprehensive documentation and examples for the RP2040, particularly concerning its integration with other components.

The decision to abandon further work on the RP2040 stemmed from practical challenges we faced, notably the inability to effectively harness a stable power source from the onboard battery regulator to power essential components such as the pulse oximeter and GPS module. This technical hurdle underscored the importance of compatibility between microcontroller and peripheral devices, which the RP2040's lack of documentation hindered.

Moreover, while the exploration of display engineering held merit, it became apparent that the significant learning curve associated with this endeavor was consuming a disproportionate amount of development time. Given our project's scope and priorities, it became imperative to reallocate resources towards addressing more critical functionalities.

The decision to discontinue work on the RP2040 was driven by documentation limitations and integration challenges. This shift in focus allowed us to prioritize our minimum viable product that could be achieved with the Arduino.

Sensor Connectivity

To implement the physical functionalities of the device, the team made use of the Arduino microcontroller board's in-built IMU. To do this, research and experiments with multiple different algorithms that used both the 3D accelerometer and gyroscope of the sensor were done. The algorithm included calculating the acceleration vector using the sum-of-squares method with respect to the acceleration of Gravity (in terms of G) and a threshold. After multiple rounds of testing and calibration of the threshold, the Pedometer algorithm was functional.

Similarly, in setting up the MAX30102 pulse oximeter sensor with the Arduino Sense Rev 2 microcontroller, the team powered the sensor via the 3V3 output pin to maintain the voltage supply. After verifying the connection between the sensor and the microcontroller, different Arduino libraries were used in testing, including those created by Adafruit and DFRobot. With the Adafruit library, we encountered issues with register assignments and inconsistent readings, and hence selected the DFRobot library for its compatibility. Using the DFRobot library, interfacing between the MAX30102 sensor and the Arduino Sense rev 2 microcontroller was intuitive and easy to set up for receiving raw readings. The raw readings received from the sensor are the change in absorbance or intensity of the reflection from the Red and Infrared wavelengths from the photodiodes. During initial testing, the team ensured consistently reliable heart rate and blood oxygen measurements.

Furthermore, implementing the GPS module was done by utilizing the Adafruit library and an in-built example code that was used as a foundation for the team to build on. The team integrated the provided code to allow the module to read raw data from the sensor using the UART protocol effectively. This data included the longitude, latitude, time and satellite status.

Challenges: While the pedometer was functional and worked as intended during testing, at times the sensor was either too sensitive or did not pick up the readings at all. This is not a factor that the team could have accounted for, nor was it fixable using the team member's skills and knowledge.

6.2.4. App User Interface Mock-up and Initial Development

At this point, the first node in the deployment diagram from Figure 1 has been planned out and it is ready for initial implementation. With regards to the center node of the deployment diagram, the

mobile application aspect of the system, it is vital to first develop a plan to understand what is expected of the application.

Planning out a mobile application involves an initial mock-up that does not have any functionality, it simply shows the key pages of an application as well as the layout of each page and what data to include on each page. This process also simplifies the implementation process as there is a predefined guideline that can be followed to build rather than leaving ambiguity in design choices. There are several tools and technologies available to develop a mock-up, however, after researching several alternatives such as Moqups it was very simple to implement a prototype using Figma due to its drag-and-drop nature and accessibility as it has a lot of features for free. The figure below shows a screenshot of the mock-up that was built using Figma's built-in tools.

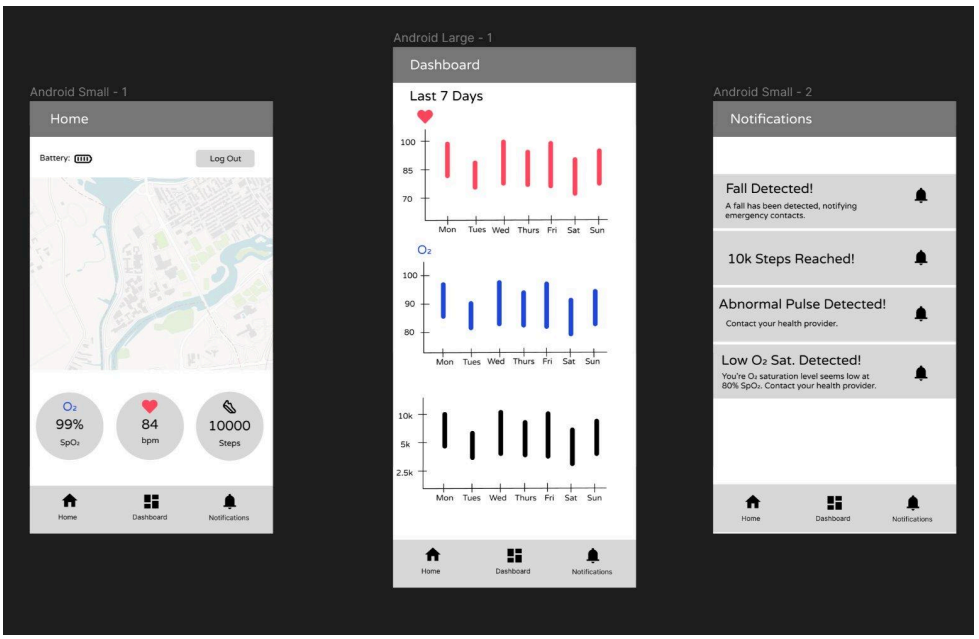


Figure 6.9: Mock-up of mobile application built in Figma

The first page of the application (left panel in Figure 6.10) shows the Home page that users will be greeted with, showing all key parameters of the application in a single page. The top left of the page shows a battery indicator, displaying the current battery percentage of the device. On the top right, there is a logout button, which logs the user out of their account and takes them to the login page of the app. In the center of the page, there is a map showing the current location of the device, and below there are 3 circles, each showing the real-time values of each parameter. The left circle shows the current oxygen saturation level of the user based on the device's readings, the center circle shows the current heart rate, and the right circle shows the current steps that the user has taken on that particular day.

The center panel in Figure 6.10 in the application shows the Dashboard page where historical data is displayed in an easy-to-read page outlining the historical values that have been collected by the device. The top of the page shows a label that reads "Last 7 Days", this would be an area for the user to select how far in history to view their data. For example, a user may want to see the historical data from the past month, or the past year, therefore this area will allow them to change that. The 3 graphs that are on the page display the past heart rate, oxygen saturation, and step data from top to bottom respectively.

The far right panel in Figure 6.10 shows the notifications that the device has raised to the user and to authorized users. The user will be able to remove notifications and see all historical notifications raised by the device for simply viewing the history.

Implementation

Translating this prototype into a functional application involved understanding how mobile apps are developed through the extensive documentation that Android provides, as well as analyzing open-source projects to gain a real-world understanding of what methods were used to implement various features.

To provide an overview of how applications are developed in Android, it is important to understand the various layers behind an Android application. The highest layer is the actual user interface that users see, which has certain layouts (e.g. LinearLayout for linearly arranged components, ScrollView layouts for scrollable containers, etc.), and UI (User Interface) elements such as buttons and several containers. The next layer is the lifecycle of the UI which are methods that are called upon a page loading, and navigating to different pages as well as bringing new pages onto the screen. The background methods and functions used for navigation are not visible to the user, but it allows the user to navigate throughout the application. Finally, there are the background functions that execute events such as updating the data displayed on a page, loading a map, logging users in with a background service, Bluetooth connectivity and many more.

The first layer of an Android application is the user interface which includes the layouts, containers and UI elements that appear on the screen. In Android applications, the elements that appear on the page are defined in XML (Extensible Markup Language) where each element has attributes assigned to it. The XML code for the logout button is shown below and it is clear to see that the button has some metadata associated with it. The id attribute is used to uniquely identify that element from all other elements on that page, this allows a specific button to be referenced in Java code so that additional functionality can be added. Other attributes are relatively self explanatory such as font, width, and height, as well as the text. The constraint attributes are used to lock an element in the page with respect to other elements, this is mainly used to ensure that different screen sizes do not affect the layout of the page. The onClick attribute is the function that is called when the button is tapped. The logOut function has to be in the Java code that runs on that particular page which is also known as an “activity” which will be explained in the following paragraph.

```
<Button
   android:id="@+id/logOut"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:fontFamily="@font/varela_round"
   android:onClick="logOut"
   android:text="Log Out"
   android:textAllCaps="false"
   app:layout_constraintEnd_toEndOf="parent"
   app:layout_constraintHorizontal_bias="0.949"
   app:layout_constraintStart_toStartOf="parent"
   app:layout_constraintTop_toTopOf="parent"
   tools:ignore="HardcodedText" />
```

Figure 6.10: XML code defining Log Out button

The following layer of Android applications is the lifecycle of the user interface which requires clarifying terminology that will be used. Firstly, an "Activity" is a Java class that is invoked when an application is launched, and when a user navigates to a different activity within the application. In the context of this application, the Login page is the main activity when the application is launched, and the sign up page is another activity, The home page that the user sees upon successfully logging in has a slightly different architecture in terms of the interface layout. This new architecture leverages "Fragments" which are also Java classes and essentially mini applications that run within a parent activity. The home page contains the navigation tray that is visible in Figure 6.11 at the bottom of each panel. This tray is part of the parent home page activity, and clicking on each page (e.g. Home, Notifications, Dashboard) fragments are used to maintain the running context of each page.

Figure 6.12 and Figure 6.13 below shows the lifecycle of Activities and Fragments respectively. Both images show the functions that are executed at each state of the application. For example, when the application is first launched to the login page, the onCreate(), onStart(), onResume() methods run in that order to make sure all components have been initialized properly. When the user logs in successfully, the process is stopped and destroyed while the home page activity loads and launches the same OnCreate(), onStart(), and onResume() methods in the context of the new activity.

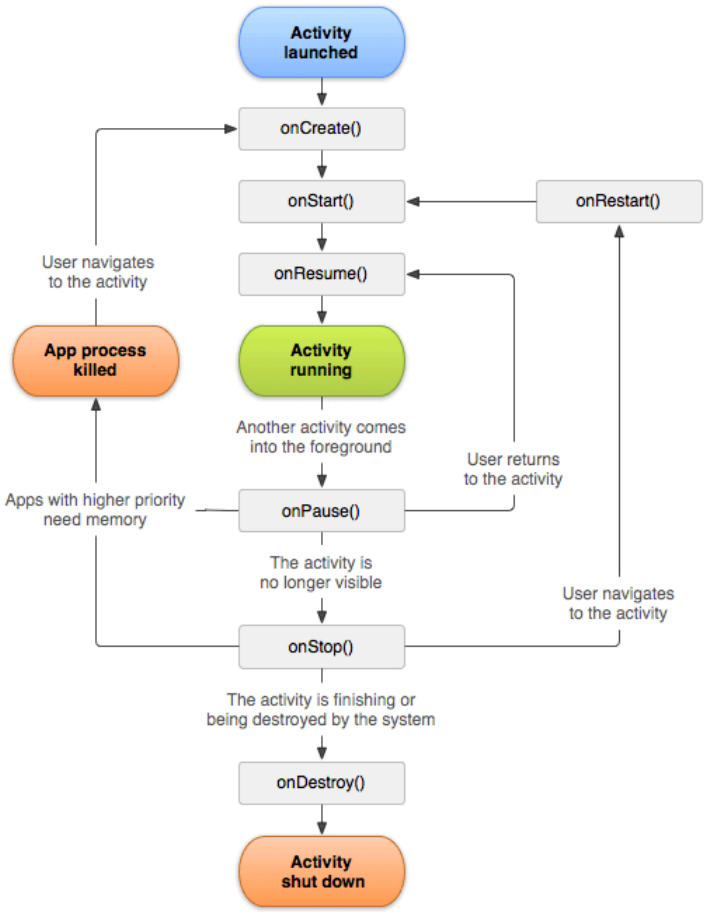


Figure 6.11: Activity Lifecycle in Android Applications [9]

Similarly, fragments have their own life cycles as they run in their own context within parent activities. As mentioned before, the home page contains 3 fragments (Home, Dashboard, Notifications) that are running when the corresponding fragment is clicked on by the user which calls all the methods in the order specified by the picture in Figure 6.13 below.

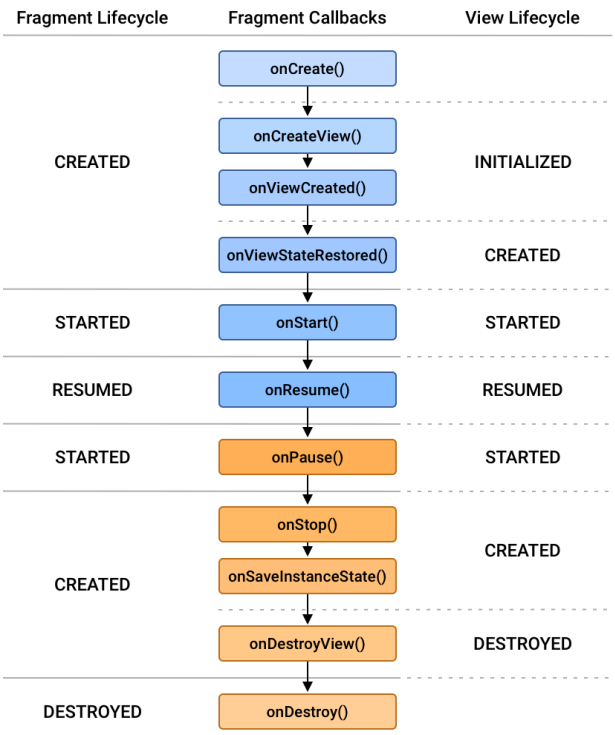


Figure 6.12: Fragment Lifecycle in Android Applications [10]

It is also important to note that these functions work like normal Java methods that can be overridden to add additional functionality or manually override existing functionality. This will be important when backend functions need to run as a user navigates throughout the user interface.

The third layer of mobile application development is the backend methods that are used for data updates, bluetooth data exchange, and other background tasks required for the user interface. More details about all of the methods used in the background of the application will be discussed in phase 2 as most of that functionality is implemented in phase 2.

Challenges:

- Based on the presented information, it is very clear to become lost in the code as there are a lot of moving parts to the application which needs careful consideration during development. In this phase of development, it was very difficult to get several components to work properly all at the same time as the app crashed or components would not appear as expected due to constraints or lack thereof.
- Furthermore, using the Google Maps API was another challenge as a Google Cloud account needed to be created in order to get the API key for using Google Maps inside the application and the map required specific methods called “callbacks” to be implemented which was not

outlined in documentation, therefore online forums proved to be useful to implement the map in the app.

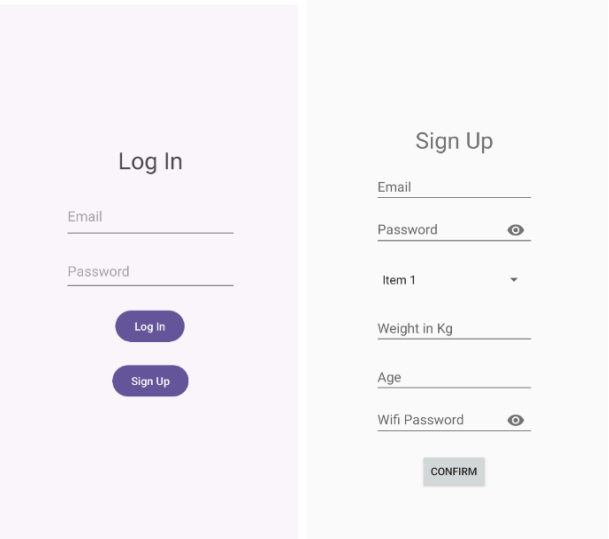


Figure 6.14: Log In and Sign Up Activities

Figure 6.14 above shows the main activities that the user sees upon login or sign up. The Sign Up button on the Log In activity launches the Sign Up activity so that an account can be created for the user through Firebase Authentication.

Firebase Authentication handles all account creation and stores emails and passwords securely on Google's servers, therefore accounts are managed and secured through Google.

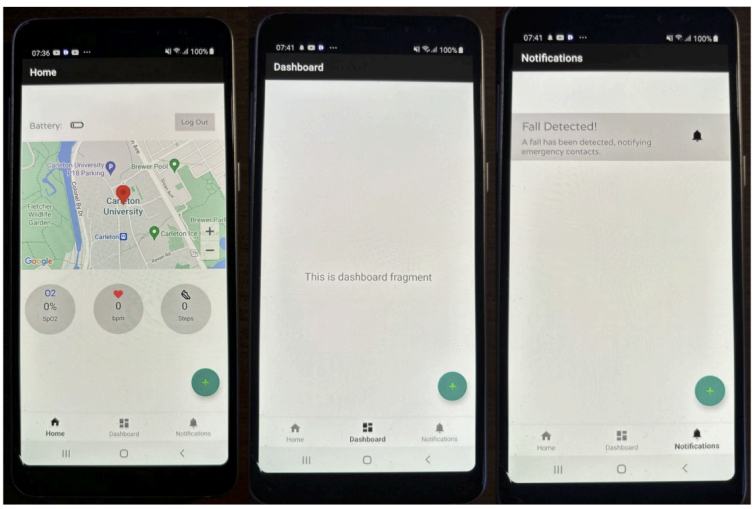


Figure 6.15: Phase 1 Implementation of Application

Figure 6.15 above shows the result of the implementation of the app by the end of Phase 1 and it is evident that the layout follows directly from the Figma mock-up that was created which highlights how vital it was to plan the interface out before building the app as it enabled focussed development. There is a new element that is visible in the image which is the floating "+" icon on the bottom right of each fragment. This is where a user is able to add the device using bluetooth to initialize the pairing

process and share key details between the wearable device and the phone. The interface for adding a new device will be covered in the next phase of implementation.

7. Phase Two: Arduino

7.1. Goals

1. Prototyping and Connection Update (Adafruit Breadboard)

A key challenge in our previous phase was in regard to the sensor connections. Due to loose connections between the breakout boards, headers, and the breadboard, connections were unable to be established reliably. A decision was made to purchase rapid prototyping breadboards and solder connections to aid in further testing and software development.

2. Sensor Readings

After a reliable connection was established between each of the breakout boards and Arduino, parsing sensor values and converting them into usable metrics was a priority. This goal marked the inception of our intensive embedded software suite alongside basic algorithm development.

3. Bluetooth Low Energy

After a reliable first prototype was constructed and sensor values were being received, the next step was to communicate them via Bluetooth Low Energy (BLE). An emphasis was placed on testing and viability of the BLE implementation as problems were detected in regard to the real-time capabilities of the BLE chip on the Arduino.

7.2. Implementation

7.2.1. Prototyping and Connection Update

Testing of the individual components and putting them together onto a breadboard posed multiple issues, including the size of each component, unreliable power connections and weak wires that broke after every stage of testing (which can take a while to build).

In order to address the most pressing issue, power connections to the components, the team decided that the best way to ensure reliable electrical connections was to solder the components onto the Adafruit prototyping breadboard. This provided a more stable platform for wiring and ensured consistent power supply to the devices. The decision came after realizing that the initial setup was not suitable for reliable performance from the sensors used.

Transitioning to the Adafruit breadboard was a large step for the project, and required a very steep learning curve in soldering for the team members. Significant time was spent learning and practicing how to use the Soldering Iron and Wick before implementing the prototype. The circuit diagram of the prototype is shown below.

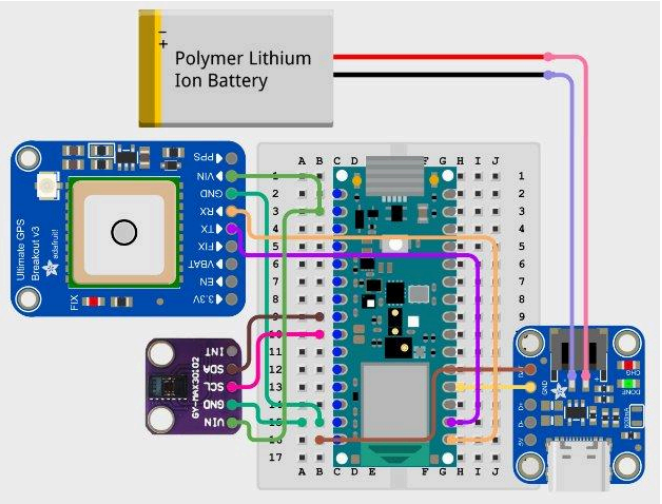


Figure 7.1. Circuit Diagram of Arduino Prototype

As seen in the figure above, all of the connections shown were soldered onto the backside of the Adafruit prototype board.

Challenge: The prototype was not a practical device due to the oversized components and poor organization of the sensors on the board, making it impractical for its intended use. The bulky size of the components resulted in an excessively large prototype, making it bulky and unaesthetic.

Since the device created had a cluster of sensors all over the board, getting measurements from the pulse oximeter quickly became a problem. It required a finger to be placed close enough to the sensor LED to produce reliable readings. As a result, one of the device's primary functionalities had become limited and was not effective in reading pulse and oxygen saturation levels consistently and accurately. This posed a significant issue, making it difficult to rely on this prototype design for accurate measurements.

The GPS module used in the prototype was somewhat large, which made it difficult to create a compact and wearable Computer-Aided Design (CAD) casing design. The module was taller than what was ideal for this specific project, and this made it inappropriate to be integrated into a portable device. This made the overall device less user-friendly and limited its use or appeal in real-world applications.

Computer-Aided Design

The CAD design process involved creating a 3D model of the prototype using the Autodesk Fusion 360 software. The design drafted mainly focused on the block-like structure that would sit on top of the wristband. The circular structure that was made was only for visual purposes, of what including a wristband may look like. It was to give a better idea of the overall design. The dimensions of the main structure were kept at 82mm x 51mm, with a height of around 10mm.

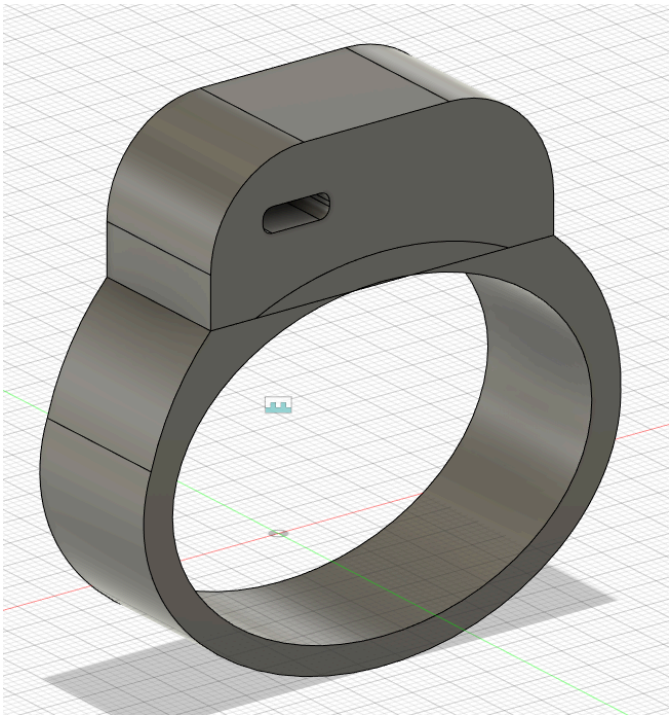


Figure 7.2. CAD Design of First Prototype

However, it's important to mention that the CAD prototype was significantly large, and was not practical for everyday use by the target audience. While the prototype served its purpose in providing a good illustration of the design, further iterations may have been required to refine the prototype and its dimensions to make it more user-friendly.

7.2.2. Sensor Readings

Pulse Oximeter

The Arduino program created to support the MAX30102 Pulse Oximeter sensor configures the sensor settings such as photodiode LED brightness, sample average, and sample rate. It initializes the I2C communication protocol through the SDA and SCL lines of the sensor and microcontroller and then checks if the sensor is connected correctly. The main loop reads the raw intensity values of the Red and infrared wavelengths from the sensor. It then uses an algorithm of ratios of the 2 values to calculate the blood oxygen and the heart rate. The last part of the loop is to then take the heart rate and oxygen saturation levels and print them on the serial monitor. This development helps to continuously monitor heart rate and oxygen saturation levels.

GPS Module

To build on the foundation code that was provided in the Adafruit library as an example code, the team worked on recalibrating the GPS module and increasing its sensitivity, which included increasing the sample rate of the device. While the execution of the program did work initially in reading the longitude and latitude values, due to discrepancies such as being indoors, clouds and the number of satellites around resulted in high reliability and low accuracy from the GPS overall, even after multiple rounds of calibration of testing. Most GPS modules available encounter common difficulties due to weak signals caused by buildings and structures that obstruct the satellite reception [15].

7.2.3. BLE Low Energy

Overview

The objective of the BLE development at this point in the project was to establish a reliable connection flow between the first prototype of the app and the Arduino. The primary goal being a two-way communication lane being created to handle first time setup credentials and user information. Building on the understanding of BLE gained in the hardware exploration step of phase one, research was continued into GATT (Generic Attribute Profile) tables, publisher-subscriber dynamics, and efficient bluetooth operation.

Arduino BLE

The Arduino BLE framework mentioned in 6.2.3 was expanded upon with further functionality. While a basic framework gave the arduino functionality to be connected to and create a basic data structure, it needed refinement to be pushed into a production environment.

First, core UUIDs (Universally Unique Identifiers) were established to provide labels to specific characteristics that the app could then search for.. The list of UUIDs was as follows:

```
#define SERVICE_UUID "19B10000-E8F2-537E-4F6C-D104768A1214"
#define DEVICEID_UUID "19B10001-E8F2-537E-4F6C-D104768A1215"
#define EMAIL_UUID "19B10001-E8F2-537E-4F6C-D104768A1216"
#define WIFISSID_UUID "19B10001-E8F2-537E-4F6C-D104768A1217"
#define WIFIPWD_UUID "19B10001-E8F2-537E-4F6C-D104768A1218"
#define AGE_UUID "19B10001-E8F2-537E-4F6C-D104768A1219"
#define SEX_UUID  "19B10001-E8F2-537E-4F6C-D104768A1220"
#define WEIGHT_UUID  "19B10001-E8F2-537E-4F6C-D104768A1221"
```

Figure 7.3. UUID definitions in Arduino

It was important for the UUIDs to be established early to proactively fight against imprudent errors in our codebase. After establishing them, a skeleton header file was constructed. The initial organization of the header file was a basis for the remaining code to be built. Thus, it was important to have a logical layout. The UUIDs were included to begin with, then the setup function was defined, “bluetoothSetup()”.

This function would serve as the inception point for any new outgoing connection request from the device. The most common use would be during our first time setup flow. The setup function goes through several sub-tasks to correctly initialize bluetooth:

1. Import <BLEDevice.h> to utilize pre-defined functions for bluetooth
2. Initialize the BLE service and create the server the service will run on
3. Create each individual characteristic given the pre-defined UUIDs
    a. Also establish read, write and notify permissions

```
wifiPWD = pService->createCharacteristic(
  WIFIPWD_UUID,
  BLECharacteristic::PROPERTY_READ | BLECharacteristic::PROPERTY_WRITE);
```

Figure 7.4. Characteristic creation in Arduino

Fig: *Example of how to define a single characteristic with read and write properties*

4. Set default values for each characteristic to negate NULL value errors upon reading the given value.
    a. This error could appear on the Arduino self-referencing or the app attempting to discover a service with no value given
5. Start the service after all characteristics have been initialized
6. Begin advertising the service with an agreed upon service UUID

These subtasks would encompass the fundamental creation flow of a bluetooth service with all relevant characteristics. The bluetooth service was integration tested with a third-party app to verify basic functionality before an attempt was made to integrate with the WHIT companion app.

Companion App BLE

As mentioned in Phase 1, bluetooth functionality is a part of the third layer in mobile application development, where background tasks are executed independently from the user interface functionality. This section aims to outline the step by step approach to implementing the bluetooth pairing and data exchange process on Android.

Firstly, upon clicking the aforementioned green floating action button from Figure 6.15 that intends to invoke the add device activity to register a new device via bluetooth, a page opens up that scans for devices using bluetooth low energy. The interface is shown below in Figure 7.5 after clicking the green "+" button:

Figure 7.5. User Interface for adding a new device using BLE

Figure 7.5 above shows the user interface that is launched upon clicking the green "+" icon. The refresh icon on the top left is used to rescan the nearby areas for bluetooth devices that are broadcasting that they are ready to initialize the pairing process.

The process for implementing bluetooth in Android is outlined below, beginning from tapping the refresh icon to initialize the bluetooth scan:

1. Upon clicking the refresh button, a method called refreshList is called in the AddDeviceActivity Java class as that is the current execution context for this part of the

application. The method begins the scanning process if it has not been initiated yet through a built-in BLE library in Android.

- The below code snippet retrieves the hardware bluetooth adapters for the current device which provides the code with an interface to execute methods that manage the bluetooth functionality
- The second last line is the method that starts the scanning process for finding bluetooth devices that are ready to pair

```
bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
bleScanner = bluetoothAdapter.getBluetoothLeScanner();
bleScanner.startScan(leScanCallback);
BluetoothDevice device = result.getDevice();
```

Figure 7.6: Bluetooth Low Energy code snippet

2. Once a scan has completed after scanning for a set period of time, another method gets automatically called (i.e. a callback is executed), in this case, the leScanCallback method is called as defined above. The bluetooth device can be retrieved by executing the getDevice method on the result of the scan. The final line in the code snippet above is executed in the leScanCallback callback method that allows the application to store the current bluetooth device object after a successful scan.
3. Based on the results of the scan, if the UUID found in the scan matches the UUID of the wearable device, it is clear that the wearable has been found in the scan. UUIDs are predefined and shared between the application and the wearable allowing the application to identify if the wearable has been successfully found.
4. Upon successful identification of the wearable, the refreshList method is called again, to update the interface to show the user a button that initiates the connection process between the application and the wearable. Figure 7.7 below shows a screenshot showing what a successful scan would show the user.

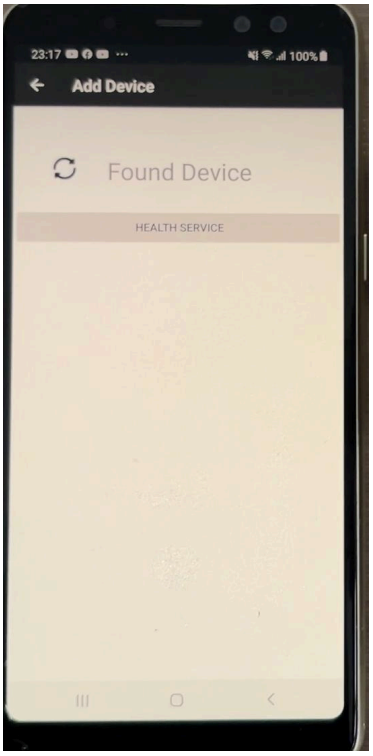


Figure 7.7: User Interface showing successful BLE scan result

5. Once the button labeled “HEALTH SERVICE” is clicked, it initializes the connection process that enables data transmission between the application and the wearable device. The connectToDevice method is executed upon the button click which calls a bluetooth method called connectGatt which returns the gatt object to allow other gatt methods to be executed. Below is a code snippet showing the method being called and the callback that is bound to the connectGatt method after it is done executing.

```
gatt = device.connectGatt(this, false, gattCallback);
```

Figure 7.8: Gatt connection code snippet

6. The gattCallback visible in the snippet above is responsible for handling all subsequent bluetooth interactions, the most notable functions that are handled by this callback are the following: **onConnectionStateChange** (handle what to do when bluetooth connection state has changed), **onServicesDiscovered** (handle what to do when all services have been discovered by the application), **onCharacteristicsRead** (handle what to do when reading data that has been sent by the wearable)
7. When the connectGatt method has completed, the connection has been established successfully and the connection state has changed and the onConnectionStateChange method gets executed.
8. Inside of the onConnectionStateChange method, the following code snippet is executed if the new connection state is “Connected”:
    - This method returns all of the services that the wearable device has broadcasted, in this case there is only one service called the “Health Service” and it is identified by its UUID as mentioned before

```
gatt.discoverServices();
```

Figure 7.9: Gatt service discovery code snippet

9. The result of the discoverServices triggers the onServicesDiscovered callback method which is subsequently executed. At this stage the Health Service UUID can be used to retrieve all the child characteristics and the data associated with those characteristics. In this project, the device id is sent from the wearable to the application upon first setup and all other characteristics (wifiSSID, wifiPwd, email, age, sex, weight) are sent to the wearable from the application. Below is a code snippet of how characteristics are read from once services have been discovered:

```
BluetoothGattService service = gatt.getService(healthService);
deviceIdCharacteristic = service.getCharacteristic(deviceIdUuid);
gatt.readCharacteristic(deviceIdCharacteristic);
```

Figure 7.10: Reading characteristics code snippet

10. When the readCharacteristic method is called, the final gattCallback method is automatically triggered which is the onCharacteristicRead method. At this point, the value for device id can be read and stored in the application and the parameters that need to be sent to the wearable can be written to the GATT table. Below is a code snippet showing a sample of how the device id is read and how the email is sent to the wearable:
    - getStringValue is a method used to retrieve the original string that was sent by the wearable as data is sent over as bytes
    - It is important to note that normal strings and integers and other data types need to be converted to bytes before being sent over bluetooth as data is transmitted as raw 1s and 0s over the air.

```
// Reading and Outputting Device ID
String value = deviceIdCharacteristic.getStringValue(0);
Log.d("Device ID", value);

// Writing Email to GATT table
emailBytes = MainActivity.email.getBytes(StandardCharsets.UTF_8);
emailCharacteristic.setValue(emailBytes);
Log.d("Email Address", MainActivity.email);
```

Figure 7.11: Device ID retrieval and email code snippet

11. A level of optimization was also performed at this stage in order for the one time setup process to be seamless and as fast as possible. Each characteristic that was written to the GATT table of the wearable device did so in its own thread. This allowed for writing all characteristics to the device at the same time which sped up the setup for the device. Below is a sample of how the email is sent over in its own thread:
    - Note: characteristics are written repeatedly to the GATT table as values may not appear in just one attempt, it may take several attempts to successfully write on the GATT table therefore this thread loops until the write operation is successful which would cause the emailSuccess boolean to be true and stop the thread from execution using the removeCallbacks function.
    - writeCharacteristics is a method that triggers the onCharacteristicsWrite callback which is automatically executed upon successful execution of the writeCharacteristic method.
        - Therefore, the emailSuccess variable is assigned to true when the onCharacteristicsWrite method is called as that reflects a successful write operation.

```
Runnable writeDataEmail = new Runnable() {
    @Override
    public void run() {
        if (emailSuccess) {
            writeDataEmailHandler.removeCallbacks(this);
        } else {
            gatt.writeCharacteristic(emailCharacteristic);
            writeDataEmailHandler.postDelayed(this, 10);
        }

    }
};
writeDataEmailHandler.postDelayed(writeDataEmail,10);
```

Figure 7.12: Sending email over BLE implementation snippet

12. If connection fails, there is a retryConnection method that attempts to reconnect with the device 10 times automatically before giving up and the user would have to retry manually. Apart from the code mentioned, there is error handling in place to ask for bluetooth permission from the user and ensuring that the bluetooth scanning process is not initiated if the user keeps pressing the refresh button amongst other quality of life features.

Challenges

In order to test the Arduino BLE implementation, a bluetooth testing application was used on a random phone as a peripheral to read the device id that was sent by the device. Similarly the application's bluetooth functionality was tested using another phone that acted like the

wearable device and broadcasted itself as the wearable device in order to validate if data was received successfully. However this presented problems with bluetooth when the wearable was tested with the application:

1. The application was entirely unable to connect with the wearable at times as the scanning period would run out, and the user had to reinitiate the adding device process
2. If the device was connected and the app was reopened, and the wearable was turned off, the application would still somehow find the wearable even though it was off.
3. Upon establishing a connection successfully, the Arduino would hang at the characteristic read state for over a minute, which would make the user believe that there was something wrong. Since the application was multithreaded to be faster, the Arduino's bluetooth module seemed to be the limiting factor in the slow bluetooth process.

8. Architecture Restructuring

8.1. Limitations and Issues with Arduino

The following highlights all of the problems that were observed and encountered with the Arduino hardware being the center of the problem.

1. Bluetooth connectivity took unbearably long which severely impacted the user experience and efficiency.
2. External display as well as external recharger board is required which significantly increases the size of the wearable device and adds several failure points.
3. The Arduino CPU contains only 1 core which heavily limits the parallelism that can be implemented and reduces the accuracy of the device
4. The shape of the Arduino is not ideal for a wearable as it is a shape that is not ideal for a watch style device. The rectangular shape creates sharp corners and makes the device harder to wear often.
5. The Arduino also only has bluetooth and no Wifi which requires the device to be close to a device in ideal conditions whereas the goal is to make the device standalone where it can function entirely away from a device.
6. Without wifi, the GPS module was required in order to get a location however the GPS was unable to find a fix indoors and in overcast weather, therefore it was effectively useless as it was almost never functioning. Wifi would allow for complete elimination of the GPS module thus saving space and also enabling standalone functionality for the wearable.
7. The designed prototype used too many large-sized components compared to the small Adafruit prototyping board. Looking back at the dimensions and sizing of the design, it was not practical nor functional for the target audience and would have used an unnecessary amount of space on the user's wrist.
8. Due to the sensors being large and so close to one another, not only did the prototype look clumsy, but the Pulse Oximeter sensor could only be accessed by inserting a finger between 2 other components. This is impractical for this project's implementation as the requirement is to have the sensor autonomously read the vitals of the user, without having to manually use or tinker with the device. This resulted in the pulse oximeter reading inconsistent values during testing, rendering it unreliable.
9. The GPS module's size impeded the design of a compact and wearable casing, reducing the device's portability and user-friendliness. Although the CAD design process provided a visual representation, the resulting prototype was impractically large.

Overall, due to the reasons mentioned above, the first prototype and design choices needed to be changed for this project to meet its functional and non-functional requirements, and for the team to create a compact, practical and user-friendly device.

8.2. Design Changes and Reasoning

- **Removing the Arduino Nano BLE Sense Rev 2**

  Removing the Arduino Nano BLE Sense Rev 2 from the project design was imperative due to numerous critical issues observed with its hardware. Firstly, the Bluetooth connectivity's latency severely undermines user experience and reduces the device's overall efficiency. The need for an LCD display and a recharger board significantly increases the device's size and introduces more potential failure points, reducing its reliability and usability.

  In addition, the Arduino's single-core CPU severely constrains parallel processing capabilities and compromises the device's accuracy and performance. The Arduino's inherent rectangular shape makes it unsuitable for a wearable device, resulting in the team having to work around the microcontroller's size to create a bulky and inappropriate device casing. Its lack of Wi-Fi connectivity limits the device's standalone functionality and forces dependency on proximity to another device, reducing its autonomy and practicality. Integration of Wi-Fi capability would eliminate the need for the GPS module, freeing up space, enabling the device's standalone operation and enhancing the device's overall functionality and appeal.

- **Removing GPS Module**

  Removing the GPS module from the new design was considered necessary due to several clear reasons. Firstly, the GPS module contributed significantly to the overall size and bulkiness of the device, making it less appealing for wearable applications. Its wholly large size made it difficult to integrate into a compact casing and ultimately compromised the portability and comfort of the device for users. By removing the GPS module, the team was able to streamline the device's form factor, making it more sleek and comfortable for the user's everyday wear and activities. Furthermore, eliminating the GPS module helped reduce production costs, making the device more affordable and accessible to a wider range of users.

- **Removing Recharger Board**

  The team decided that removing the recharger board is essential to streamline the wearable device's design and enhance its reliability. The inclusion of an external recharger board increases the device's size and complexity. Relying on internal charging mechanisms, such as a built-in board, eliminates the need for additional external components, making the device more compact and aesthetically pleasing.

- **Integrating ESP32-S3 Touch LCD 1.28**

  The ESP32-S3 Touch LCD 1.28 microcontroller board has several features that make it an ideal fit for this project. Firstly, its enhanced processing capabilities, featuring a dual-core Xtensa LX7 CPU running at up to 240 MHz, provide good computational power for the real-time sensor data processing and analysis required for the device. This allows for efficient handling of sensor data, ensuring accurate measurements of pulse and oxygen saturation levels.

Figure 8.1: ESP-32 S3 Microcontroller

Additionally, the ESP32-S3's built-in Wi-Fi and Bluetooth connectivity allow seamless communication with other devices over large distances, eliminating the need for external modules and enhancing the device's standalone functionality. Furthermore, its compact, common circular form factor and low power consumption make it well-suited for wearable applications, ensuring long battery life and a potentially compact and comfortable casing design for users.

In addition, it would be important to note that at the time the team started implementing the first 2 phases, the ESP32-S3 was not available on online markets for purchase and hence wasn't a viable option at the time. The microcontroller was only released for purchase much later during development and encouraged the architecture restructuring of the device.

9. Phase Three: ESP32-S3

9.1. Goals

1. Build Watch LCD User Interface

As mentioned in the previous section, one of the challenges we faced was to find and start building an LCD interface that was suitable for the purposes of the project. In finding the ESP-32 S3, one of the first main goals the team had was to start establishing the user interface of the touch LCD on the microcontroller. This was done through Light and Versatile Graphics Library (LVGL).

2. Implement Multiple Pulse Oximeters

While using one pulse oximeter was functional and worked fairly well in the development of the previous prototype, we faced many issues as outlined in the last section. After careful research and experimentation, the team decided to implement two pulse oximeters to address and reduce many of the issues faced previously and implement an averaging algorithm.

3. Update Application User Interface

This section will outline the key changes and rearrangements that were made upon reevaluation of the user interface. Initial implementation shows the Dashboard page with 3 graphs all on one page which proved to be difficult to read. Therefore the user interface is rearranged to provide a dedicated page for pulse, oxygen saturation level and steps allowing for additional functionality to be implemented.

4. Accomplish Fall Detection

Another goal for this phase was to create, test and establish a program that would detect if the user had fallen and send an associated notification or alert on both the watch user interface and the application.

5. Integrate Battery into Prototype

Moving onto a new microcontroller included removing the previous battery that was being used due to the difference in power usage by the new prototype's components and size. As such, another goal was to find and integrate a new, compact and reliable battery component into the new prototype design.

6. Draft Watch CAD Casing

As a team, we set a goal to draft the new CAD designs for a prototype with a new structure and dimensions while working towards other milestones.

7. Establish Wi-Fi Connection and Port Old BLE functionality

As a result of switching micro-controllers, previous BLE functionality is required to be ported onto the new device. Alongside this, since the Wi-Fi chip is integrated to the Printed Circuit Board (PCB) , basic Wi-Fi functionality should also be prioritized in order to interface with the REST API being constructed.

9.2. Implementation

9.2.1. Build Watch LCD User Interface

Overview

In order to implement a functional touchscreen-compatible UI, LVGL was used. It is a free and open-source platform that builds on the TFT libraries that were researched in phase one of our project. While only a limited amount of code was transferable between the R&D conducted on the RP2040, some assets such as custom bitmap images could be transferred as the display was identical. The library offers an exceptional amount of configurable elements while maintaining a low memory footprint.

Some key features include:

- Building blocks such as buttons, lists, and sliders
- Graphical animations, opacity, and smooth scrolling
- Customizable graphical elements via custom CSS-like styling
- Written in C allowing complete flexibility when it came to system-based alterations
- Free and open source under MIT license

Components

In this section, the components used in creating the GUI will be outlined. Code snippets will be provided for reference.

Attributes;
Attributes are types of modifiers that can be applied to different components of the LVGL library. They are a basic type of element that is used in the styling and positioning of different elements on the display screen. Some examples of attributes are:

- Position
- Size
- Parent
- Styles
- Event Handlers

Certain objects also have specific attributes such as a slider containing a maximum and minimum value attribute.

Objects:
The basic building block in LVGL is called an object. Almost all predefined objects are derived from the superclass “Object”. All objects are referenced using type lv_obj_t and it can be used as a pointer. Objects can be assigned attributes on initialization such as position and size. Certain objects such as slides can be given minimum and maximum values.

Objects can be given parent-child structures with one-another. Every object then has by extension, exactly one parent. On the contrary a parent can have any number of children. Objects move relative to one another, if a child is located in the center of a parent object, it will move in relation to the parent's center point, not the layer above it.

Objects can be created and deleted dynamically at runtime in order to suppress Random Access Memory (RAM) concerns or aid in functionality. This dynamic creation can be used in the settings screen, all corresponding widgets located within, and any notification screens.

Each object is constructed from multiple parts. For example a slider may contain the following parts:

- LV_PART_MAIN: the background
- LV_PART_SCROLLBAR: the scrollbar
- LV_PART_INDICATOR: the indicator of what value the scroll bar is currently at
- LV_PART_KNOB: an indicator of where to touch to change the scroll bar value

The primary purpose of using parts is to allow for styling and custom events to be tied to each of the widgets. This could be useful with custom sliders to control brightness or toggle WiFi/BLE.

Objects can also be given states in order to be referenced later, some examples of this are:

- LV_STATE_DEFAULT: normal state
- LV_STATE_CHECKED: toggled state
- LV_STATE_USER_1: custom state

These states are typically assigned automatically but can be controlled manually. This may be useful to auto-switch to battery saver modes while also letting the user know of a change via state change of controls.

Images

Images in LVGL can be imported via custom C files that hold bitmap values of each pixel designating their state and a list of colors associated with each pixel. Images are treating like objects once imported, the import process is as follows:

1. Take desired PNG and convert it to the required C bitmap file structure:

```
const LV_ATTRIBUTE_MEM_ALIGN LV_ATTRIBUTE_LARGE_CONST uint8_t map[] = {
  0x00, 0xFF, 0x00, 0xFF,  // Green, Blue, Green, Blue
  0xFF, 0x00, 0xFF, 0x00,  // Blue, Green, Blue, Green
  0xFF, 0x00, 0x00, 0xFF,  // Blue, Green, Red, Blue
  0x00, 0xFF, 0xFF, 0x00,  // Green, Blue, Red, Green
};

const lv_img_dsc_t img_ = {
  .header.always_zero = 0,
  .header.w = 4,
  .header.h = 4,
  .data_size = sizeof(map),
  .header.cf = LV_IMG_CF_TRUE_COLOR,
  .data = map,
};
```

Figure 9.1: LVGL code snippet showing bitmap implementation

   This conversion can be done via a proprietary LVGL website called LVGL bitmapper [8]
2. Proceed to import the image file as a LVGL object
3. Apply any corresponding formatting

Layers:

In LVGL, layers are created based on the order of widget creation. Each layer can hold a single object or can be rasterized to a single layer to save memory.

By default each object is drawn on top of a previous object with its own layer. Each new layer and its objects can cover the previous objects. These objects can overlap dynamically and only cover the area they are designated to take as can be seen below in Figure 9.2. The button on top is only covering the bottom left of the button underneath it.

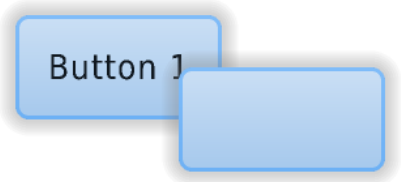


Figure 9.2: Layered buttons illustration

The order of these layers can also be changed provided a direct reference to the object is attainable. There are also two special layers designated to persistently be on the top of the screen, layer_top and layer_sys. The difference between these two layers is insignificant for the scope of this project, however, they will be particularly useful in displaying alerts and important notifications.

If the *CLICK* attribute is enabled, the layer_top will act as a screen sized button, which also may be useful for dismissing alerts and similar use cases. There is also a bottom layer located below each of the existing layers that can be identified without an object. It's useful for setting background colors and properties.

Styles:

Styles are used to group the modifiers applied to an object and are heavily inspired by CSS used in web development. The concept is as follows:

- Each lv_style_t object can hold multiple properties such as:
  - Border color
  - Border width
  - Text color
  - Text size
- Styles can be assigned to objects so long as the object has the relevant attributes available to modify
- The last applied style to an object takes precedence at run-time
  - This is particularly important when changing effects of an element at runtime (alert)
- Transitions can also be applied to the style when a state change happens to an object

However, styles do not need to be applied in chunks via the style_t object. Styles can also be individually applied to specific parts of an object via the part selectors such as LV_PART_MAIN.

To initialize a style the following code can be used:

```
static lv_style_t style_btn;
lv_style_init(&style_btn);
lv_style_set_bg_color(&style_btn, lv_color_hex(0x115588));
lv_style_set_bg_opa(&style_btn, LV_OPA_50);
lv_style_set_border_width(&style_btn, 2);
lv_style_set_border_color(&style_btn, lv_color_black());
```

Figure 9.3: LVGL code snippet showing style initialization

Not all of the specifics of this code snippet will be covered, however, it is useful to see a style being created for a button, modified, and then applied. Special attention should be paid to how the opacity and color of the button are set in the style as these attributes are very useful in building an intuitive UI. For example, a button should have its opacity reduced along with its events changed if the button's functionality is disabled.

Some of the notable styles that may be of use are the following:

- Background
- Border
- Outline
- Shadow
- Padding
- Width and Height
- X and Y Translation

Scrolling

Scrolling in LVGL is very intuitive and self-contained. Any layer can be scrolled with however many objects are located within it. If the active layer is larger than the display, scroll bars and corresponding functionality will be automatically enabled allowing the user to access all parts of the layer through scrolling.

It is however possible to make elements also non-scrollable. This is useful in the case of an object being intentionally larger than the display to minimize gaps on the borders of the screen.

If an object cant be scrolled past the boundaries of its parent, a flag can be raised within the object to enable LV_OBJ_FLAG_SCROLL_ENABLE, this will allow the object to enable a effect showing the object slowing down the further it leaves its bounds and rebounding to its original position. This will be useful in displaying to the user intuitively that there is no more content beyond this point.

Similarly, while scrolling through an element, snapping can be utilized by enabling the LV_OBJ_FLAG_SNAPPABLE flag. This designates the object as snappable, causing a scroll that ends close to this object to cause it to be the focus of the screen. This property could prove of use while developing the fundamental skeleton of the UI. Many further flags can also be enabled such as setting the maximum number of scrolls or which element to automatically scroll on an event change.

Animations

Animations can be programmed to automatically changing the values of attributes and corresponding styles. The animations in LVGL work by calling an animator function that is responsible for the value change. Animations can range from being basic single value changes on event, to complex maneuvers that demand more CPU and memory usage.

Animation speeds can be controlled via changing the animation path attribute, examples:

- lv_anim_path_linear()
- lv_anim_path_step ()
- lv_anim_path_ease_in ()

Custom animation timelines can also be created, this is currently beyond the scope of the framework UI, however, can be utilized for advanced scrolling and reset animations.

Example code of how to build an animation object can be found below:

```
lv_anim_t a;
lv_anim_init(&a);
lv_anim_set_var(&a, label);
lv_anim_set_values(&a, lv_obj_get_x(label), 100);
lv_anim_set_duration(&a, 500);
lv_anim_set_exec_cb(&a, anim_x_cb);
lv_anim_set_path_cb(&a, lv_anim_path_overshoot);
lv_anim_start(&a);
```

Figure 9.4: LVGL code snippet showing animation objects

Events

Events are used to inform the end-user that some changes, either internal or external, have taken place. Events are flag based triggers that can be linked to corresponding callback functions similar to

interrupts in embedded software. They play a pivotal role in orchestrating animation queues, functionality shifts, and raising exceptions on the user interface.

Events can be bound to objects based on certain actions such as being clicked, dragged, or released.

An example of how a event is tied to an object with a callback function can be found below:

```
lv_obj_add_event_cb(btn, btn_event_cb, LV_EVENT_CLICKED, NULL);

void btn_event_cb(lv_event_t * e)
{
    printf("Clicked\n");
}
```

Figure 9.5: LVGL code snippet showing button click handler sample callback

UI Fundamentals

While designing the UI for the WHIT, careful consideration was given to the readability, useability, and most importantly functionality of the UI. These are the design tenets that were used contextually during prototyping.

- Readable display
    - The WHIT's LCD display is relatively small, careful attention should be paid to font-size, color, and contrast
    - Corresponding to the use-cases of the WHIT, the wearer may have compromised eyesight, a potentially large-font and high-contrast mode should be considered
- Prioritize important information
    - On tangent with the previous tenant, only critical information should be placed on the home screen of the UI to prevent visual clutter. Being a health-focused wearable, BPM, SP02, and Step count should take precedence.
- Intuitive navigation
    - Navigation should be intuitive with clear visual markers for boundaries and easy to contextualize symbols for the settings menu
- Consistency
    - Consistency between the types of symbols used between the wearable UI and the companion app UI

By incorporating basic principles, a user-friendly and intuitive UI can be created that displays relevant information with clarity and consistency.

Basic structure of the UI

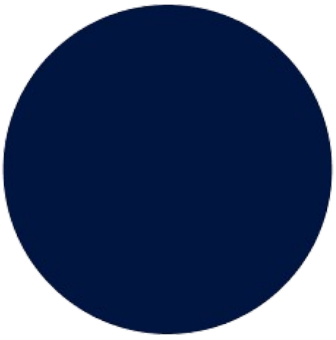
At the point in the development, a decision was made to develop a tile based UI where the user could swipe up or down to navigate. Similar to flipping pages in a portrait notebook, each "tile" could contain different information about the device.

Some early ideas were:

- Tile for basic metrics, time, date, and battery status
- Tile for access to kick-off a manual reading of BPM and SPO2
- Tile for current location shown on a map or as current location reading

- Tile for calorie tracker based on current number of steps and bpm over time
- Tile for a settings page to control brightness, connectivity options, and enable battery saver mode

In order to create this gallery-like UI design a background layer was created with a semi-dark color to increase contrast to any elements above it.



```
lv_obj_t  panel = lv_obj_create(lv_scr_act());
lv_obj_set_size(panel, 240, 240);
```

Figure 9.6: LVGL code snippet and diagram showing UI background code

This background panel would also serve as the parent to all other objects being placed within the UI. The next element included was the battery indicator, a small bar outlining the border of the UI will represent what percentage the battery is charged to.

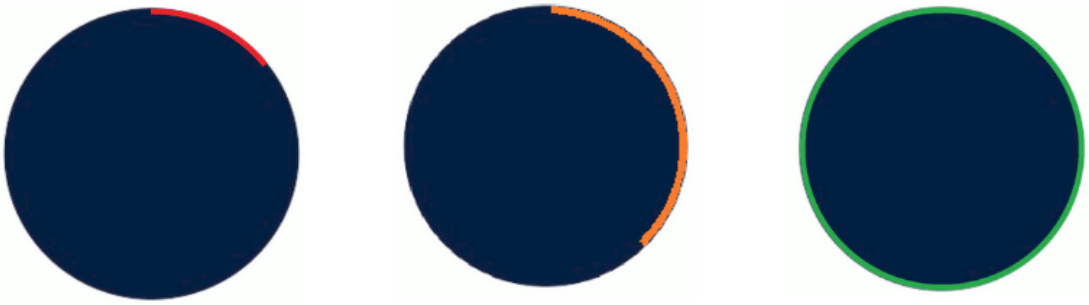


Figure 9.7: UI screenshot showing 3 battery states, low, medium, and full charge

```
battery = lv_arc_create(lv_scr_act());
lv_obj_set_size(battery, 240, 240);
lv_arc_set_rotation(battery, 270);
lv_arc_set_bg_angles(battery, 0, 360);
```

Figure 9.8: LVGL code snippet of battery indicator arc

The battery bar would also have an animation tied to it, changing the value of the bar based on the most recent battery value. More information about animations can be found under “animations” above.

```
void changeArcValue(lv_obj_t *arc, int32_t stopVal) {
  lv_anim_t anim;
  lv_anim_init(&anim);
  lv_anim_set_var(&anim, arc);
  lv_anim_set_exec_cb(&anim, set_angle);
  lv_anim_set_time(&anim, 1000);
  lv_anim_start(&anim);
}
```

Figure 9.9: LVGL code snippet showing how battery level updates on UI

```
Implementation of color change omitted for brevity
```

Once the implementation of the battery indicator was complete, development of the scrollable tiles could begin. In order to implement a scrollable set of tiles, sub-objects must be created with a circular shape in the size of the empty area between the battery indicator. This was approximately 220px wide in diameter.

The corresponding flags were raised in the parent object to enable scrolling, dictate the direction of the scroll, and enable snapping to each of the tiles.

```
lv_obj_set_scroll_snap_y(panel, LV_SCROLL_SNAP_CENTER);
lv_obj_set_scroll_snap_x(panel, LV_SCROLL_SNAP_CENTER);
lv_obj_set_flex_flow(panel, LV_FLEX_FLOW_COLUMN);
lv_obj_set_style_outline_width(panel, 0, LV_PART_MAIN);
lv_obj_set_scrollbar_mode(panel, LV_SCROLLBAR_MODE_OFF);
```

Figure 9.10: LVGL code snippet showing scroll enablement

After this was set, it would enable all sub-objects to scroll in a vertical manner and be organized one above the other.

The sub-objects now needed to be created, a simple loop was utilized and sub-objects were created in the same fashion as the main background panel, with a slight reduction in diameter and different color.

```
for (i = 0; i < 2; i++) {
    lv_obj_t *subPanel = lv_obj_create(panel);
    ....
}
```

Figure 9.11: LVGL code snippet showing subpanel creation for loop

Then each subPanel that is created can act as a parent for all other sub-objects. The first panel that is created is the home panel that contains all basic information related to biometrics and system status.

The elements utilized on the home page are as follows:

- Labels for all text based information
- Images to represent symbols for bpm, spo2, and steps
- Font Style objects to alter size and color of labels

Creating a label is fairly straightforward, however, configuring the font used for the lab is an involved task. First, the internal lvgl_configuration file must be altered to pre-load the bitmap values for different font types and sizes. This is because each character within the specified font library is a small image with encoded bitmap values. Effectively making every letter a highly space-efficient image. This is what the configuration file looks like:

```
/*Montserrat fonts with ASCII range and some symbols using bpp = 4
 *https://fonts.google.com/specimen/Montserrat*/
#define LV_FONT_MONTSERRAT_8  0
#define LV_FONT_MONTSERRAT_10 0
#define LV_FONT_MONTSERRAT_12 1
#define LV_FONT_MONTSERRAT_14 1
#define LV_FONT_MONTSERRAT_16 1
#define LV_FONT_MONTSERRAT_18 0
#define LV_FONT_MONTSERRAT_20 0
#define LV_FONT_MONTSERRAT_22 1
#define LV_FONT_MONTSERRAT_24 0
#define LV_FONT_MONTSERRAT_26 0
#define LV_FONT_MONTSERRAT_28 1
#define LV_FONT_MONTSERRAT_30 0
#define LV_FONT_MONTSERRAT_32 0
#define LV_FONT_MONTSERRAT_34 0
#define LV_FONT_MONTSERRAT_36 0
#define LV_FONT_MONTSERRAT_38 0
#define LV_FONT_MONTSERRAT_40 1
#define LV_FONT_MONTSERRAT_42 0
#define LV_FONT_MONTSERRAT_44 0
#define LV_FONT_MONTSERRAT_46 0
#define LV_FONT_MONTSERRAT_48 0
```

Figure 9.12: Arduino code snippet showing font style and size definitions

As can be seen in the Figure 9.12 above, the font style and size must be pre-enabled before code compilation for it to be used in runtime. The font used was montserrat as it was deemed to be the most readable after testing with various options. The sizes enabled were 12, 14, 16, 22, 28, 40.

Time, date and biometric label placeholders were then created and placed as below:

Figure 9.13: Diagram showing the homepage of the wearable

```
timeLabel = lv_label_create(subPanel);
lv_label_set_text(timeLabel, "0634");
lv_obj_align(timeLabel, LV_ALIGN_CENTER, -40, 0);
lv_label_set_long_mode(timeLabel, LV_LABEL_LONG_WRAP);
lv_obj_set_width(timeLabel, 52);
lv_obj_set_style_text_align(timeLabel, LV_TEXT_ALIGN_CENTER, 0);

static lv_style_t style;
lv_style_init(&style);
lv_style_set_text_font(&style, &lv_font_montserrat_40);
```

```
lv_obj_add_style(timeLabel, &style, 0);
```

Figure 9.14: LVGL code snippet showing creation of time label

As can be seen, the time and date seem contextualized as they are on the face of a watch, however, the three metrics on the right side need images/symbols that correspond to their purpose for clarity. In order to this, images need to be utilized alongside bitmaps, and chroma keys. For the brevity of this report, image import and configuration will not be expanded upon, however, extensive documentation is available on LVGLs website [8].

Figure 9.15: Diagram showing completed homepage for watch UI

Images were imported for Bpm, Sp02, Steps, and a placeholder settings button. At this point, basic UI functionality has been achieved. Methods to update the labels in realtime will be developed after sensor functionality is further developed in the project.

9.2.2. Implement Multiple Pulse Oximeters

The motivation behind using multiple pulse oximeters was to reduce the issues faced when using only one. These included erroneous readings from the sensor, consistent default no-value readings and ambient light issues that affected the readings calculated by the program. In order to address all these issues the idea was to add on another pulse oximeter so that the program would take an average of both the sensors to return a more reliable and accurate reading. In addition, if one of the pulse oximeters was to revert back to default values, the reading from the other sensor would be used, and lastly, this would make the pulse oximeter reading algorithm more functional overall.

To integrate the two MAX30102 Pulse Oximeters, the team had to consider that since the two sensors were the same, they had the same device address and used the same I2C communication protocol. Initially, this posed a challenge as the microcontroller only has one set of SDA and SCL lines. However, using a parallel circuit to share the I2C bus, and taking turns to turn the sensors on (one after another, in a cycle), the goal was achieved.

The algorithm was composed on an Arduino program. As the ESP-32 S3 only has one dedicated SDA and SCL line, each of the sensors was assigned to a different pair of GPIO pins. The program took turns defining those pins as the SDA and SCL lines and using the I2C bus to send over the readings. To set up, both the sensors were configured to their ideal LED brightness and sampling rate values. In order to conserve power and retain more battery life, the team had also created a separate program to only turn the sensor 'ON' when its GPIO pins were active on the I2C bus. This was also done to take

samples of the sensor readings one after another for the averaging algorithm. The algorithm is shown in the figure below.

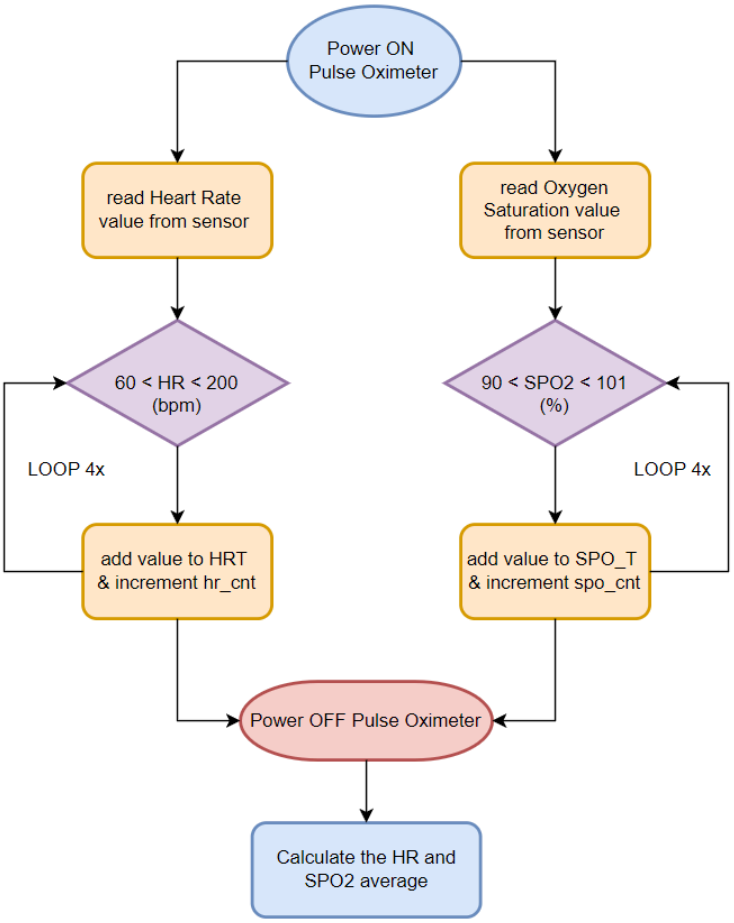


Figure 9.16. Algorithm of Averaging the Pulse Oximeter Readings

9.2.3. Update Application User Interface

To justify the reason for rearranging the user interface, it is essential to observe the limitations of the previous phase of the user interface. In Figure 9.17 below, a screenshot of the pre-existing dashboard page is provided to illustrate the issues with presenting historical data in this method.

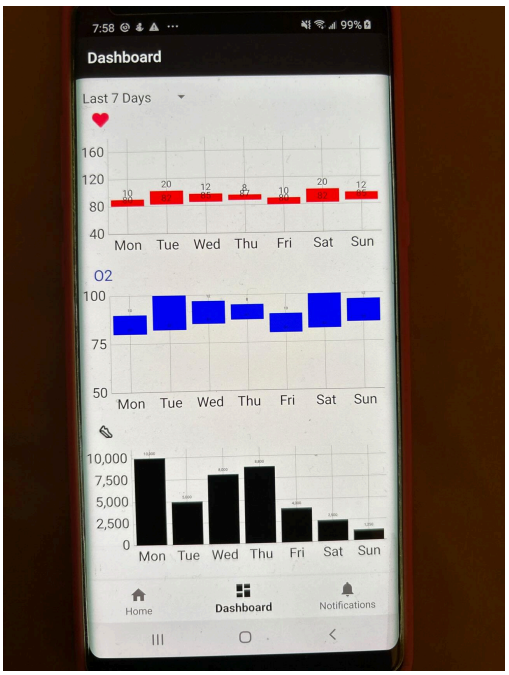


Figure 9.17: Image of Dashboard page with initial design

Figure 9.17 clearly displays the 3 graphs that are quite small and could be difficult to read and does not allow for additional information to be provided to the user for each parameter (e.g. pulse, oxygen saturation level, and steps). After brainstorming ideas for rearranging the UI, it was determined that the navigation tray visible at the bottom of the page in Figure 9.17 would contain Home, Notifications, and Settings in that order from left to right.

This would eliminate the dashboard page, therefore all the historical data would be accessed by clicking on the real time pulse, oxygen saturation level, as well as the steps circles that are visible on the home page. Thus each historic graph would correspond to the correct parameter, clicking on the pulse circle would open the historical pulse data with its graph. This allowed for a larger and clearer graph to be displayed, while also adding space to add information that would provide more insight to the data that is collected by the device. Furthermore, historical data can be viewed in different time ranges, from the past 7 days, all the way to the past 2 years and users can select where to modify the time ranges in the top of the page in Figure 9.17 by tapping the label that says “Last 7 Days”.

When a user needs to view historical and other insights in their pulse over time, they simply tap on the circle showing their current heart rate on the home page which is visible in Figure 6.2.4.5 on the left image. This opens the Heart Rate Detail page, which can be seen in Figure 9.18 shown below.

The detail page shows a larger graph with a date range that can be dynamically changed by tapping the top left of the page on the “Last 7 Days” label. The next element below the graph displays the current resting heart rate of the user based on their age, weight and sex which was sent to the device over bluetooth during the initial setup process. The algorithm for calculating the resting heart rate is implemented on the watch and sent to the application. More details on the exact data flow will be provided in Phase 4 of the implementation. There is also another element on the page which educates the user on what the values collected by the device mean. For the user in Figure 9.18, a healthy resting heart range has been dynamically identified based on their unique parameters of age, weight and sex. This allows the user to understand the data that is being presented to them

Similar to the Heart Rate Detail page, there is a Oxygen Saturation Detail page which once again shows historical data similarly to the heart rate detail page, and it shows the current oxygen saturation level, and the final element tells the user what oxygen saturation level means and what a healthy range looks like for that specific user. The steps detail page also works in a similar manner to the aforementioned detail pages, however the key difference is that the steps data collected gets converted into the distance traveled for that day, which gives the user a better perspective on what the steps data collected means.

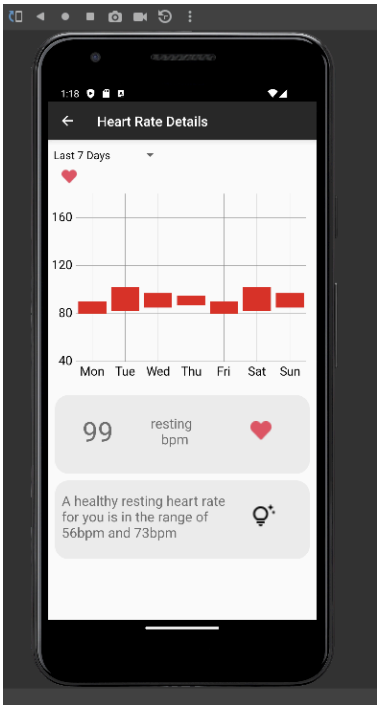


Figure 9.18: Image of the details page for heart rate

With the removal of the Dashboard page, it allowed for a Settings page to be added instead of the Dashboard page which allowed users to manage authorized users to provide or revoke access to the device data to certain individuals.

Overall, the reorganization of the dashboard page allowed the user to focus on each parameter individually, and allowed the user to become more educated on what data gets collected and what that means for their health. This also allowed the Dashboard page to be removed and a Settings page to be added in place of that and streamlined the user experience significantly.

9.2.4. Accomplish Fall Detection

To implement the product's key features, the team started the program, testing and validation of the Fall detection algorithm. Falls are common among elderly people and can lead to potentially serious injuries. The fall detection feature of the device helps by quickly identifying falls using the program developed and alerting caregivers or emergency services. This ensures timely help in case of emergencies, providing peace of mind for both the users and their families.

This algorithm was built on the Arduino platform and used features from the pedometer as well. Initially, the program checks if the acceleration in the downward direction crosses a predefined threshold with respect to a number of G's (gravity's acceleration, 9.81m/s). If that condition is

satisfied, the program checks to see if the user is still moving or walking, as this would indicate that the user simply shook their hands with a lot of force - as one would do when shaking a bottle for example. If this condition is not satisfied, then the program deems the user's action as a 'fall'. This response is displayed on the watch as an alert with a countdown timer, after which the device will start to call or notify the user's caretakers or the police. The device also uses Bluetooth to send out this response to the application and creates a notification alert. The flowchart for this algorithm is shown in the figure below. The caretakers mentioned in this scenario would be the readers of the user's data, which can be set up in the application by the user.

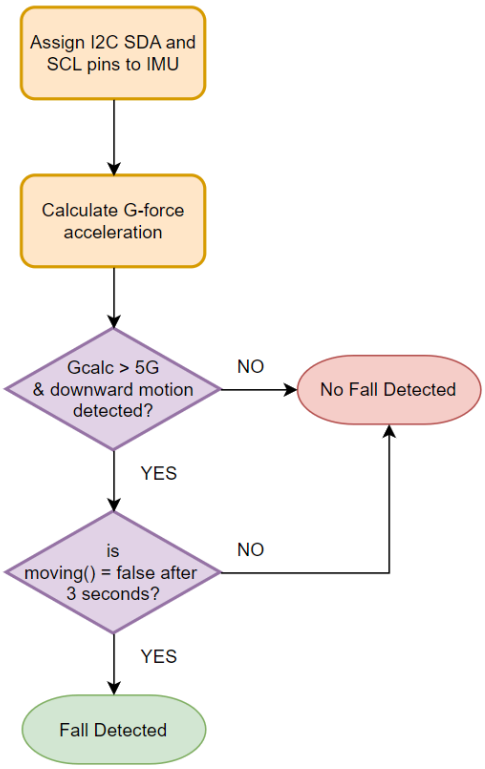


Figure 9.19. Fall Detection Algorithm

9.2.5. Integrate Battery into Prototype

The decision to integrate a rechargeable battery in the watch device project is embedded in the device's practicality and wireless requirements. Rechargeable batteries offer cost-effectiveness in the long run, as they can be recharged multiple times, eliminating the need for frequent battery replacements. And even if required, they can be easily replaced - as can be done with other mobile devices.

This feature enhances the user experience by providing continuous usage without interruptions due to battery depletion and the constant need to recharge. It also eliminates the necessity to be near an outlet and gives the user more free-range movement when using the device in their daily activities. Moreover, this battery component aligns with modern consumer preferences for eco-friendly products, enhancing the market appeal of the device.

The battery utilized in this device is a DC 3.7V Rechargeable Lithium Polymer Replacement Battery. The battery we chose has a capacity of 220mAh, which supports all the power consumption of the

sensors. After conducting several rounds of testing with various battery models, sizes and capacities, our team decided to proceed with this particular one as it lasted up to approximately 8 hours.

9.2.6. Draft Watch CAD Casing

The team spent many hours brainstorming and researching different designs for the new prototype's casing. After considering various options, they decided to follow the shape and design of watches found commonly online. This included features such as a circular display and side lugs to support the watch and help the product fit on the user's wrist. To turn their vision into a reality, the team turned to Fusion 360 to create intricate designs, test various configurations, and make adjustments in real time. After several iterations, the team finally settled on a design that met all the product's requirements at the time. The first draft of the design is shown in the figure below.

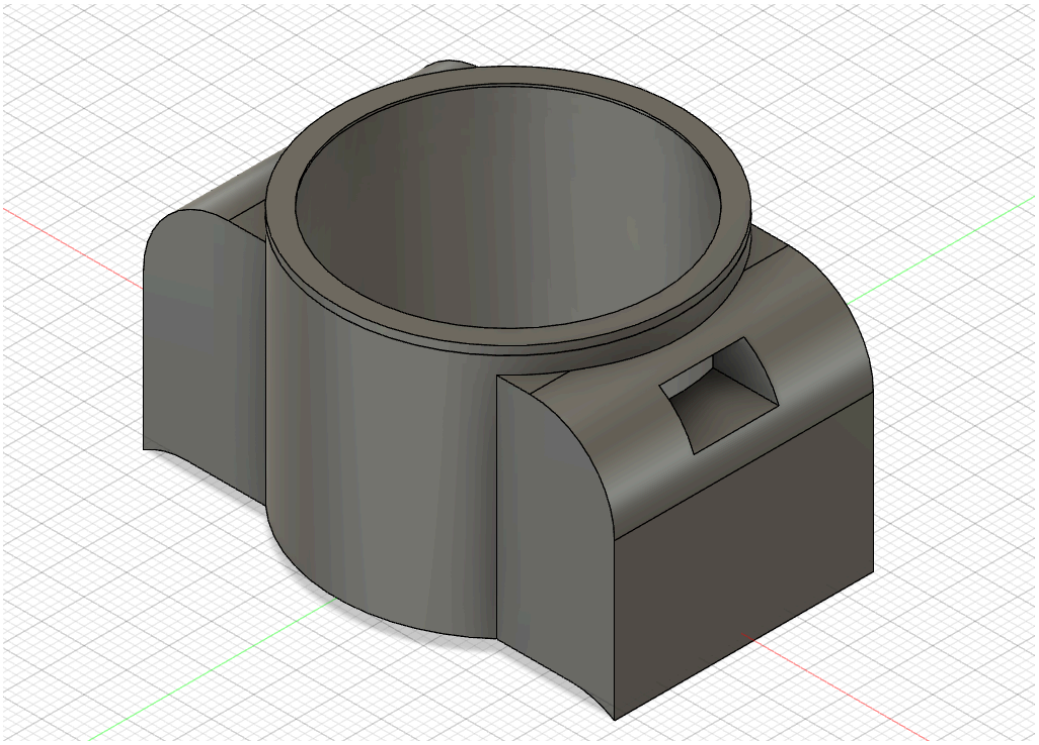


Figure 9.20. CAD Casing for First Prototype

As can be seen in the figure above, the watch casing created followed a cylindrical shape for the main body while the side lugs are extensions of that body. On the right side of the watch, a rectangular hole was also made to allow for a charger cable to be put through for recharging the device. While this design had well-thought-out features, its bulkiness, size and fit on the user's wrist did not meet the team's standards. In addition, due to the closed nature of the main body (cylindrical part), inserting and setting the components was a seemingly difficult task. In the same manner, removing to test and restructure the components was also difficult. This posed a challenge to the group as solidifying the microcontroller board circuitry required multiple test rounds, during which the components would have to be removed. Furthermore, in order to ensure product and hardware safety, the circuit needed to be in a position where the team would be able to tinker and use tools such as electrical tape to keep them in place. This was not achievable with this casing prototype. Ultimately, the team decided to create a simpler and shrunken-down model that more closely resembled the watches seen online.

9.2.7. Establish Wi-Fi Connection and Port Old BLE functionality

Porting BLE functionality

When porting BLE functionality, the overhead on BLE was not an expense as the fundamental principles of the technology did not change. The primary change was the libraries being used in order to harness the bluetooth chip within the PCB. Some slight syntax changes were made to adapt the code to the ESP32 platform. The header file remained almost entirely unchanged.

A small degree of testing was conducted to verify functionality of the BLE chip. After testing and verification was complete, it was clear that the new bluetooth chip was significantly faster at the initial connection process. The companion app was able to discover the service and all characteristics within 10 seconds.

Wi-Fi Connectivity

In order to establish Wi-Fi connectivity, the <WiFi.h> library was used. The connection process was straightforward.

10. Phase Four: Integration

10.1. Goals

1. **Integrate Historical Data**

   In order for the device to effectively monitor and provide helpful health advice to the user, it is necessary to save their historical data on a secure database. The team's objective was to establish a safe and reliable way to store and display the user's past vital and physical levels on the supporting application.

2. **Google Geolocation API**

   Due to the previous issues, the team decided to use the Google Geolocation API instead of a physical GPS module for this prototype. The team's goal was to create an Arduino program that would be used to monitor the user's live location, making the device smaller and cheaper.

3. **Update the Prototype CAD Design**

   Looking at the design hurdles faced with the previous CAD iteration, some minor tweaks were made to the prototype to enhance the device's aesthetic, overall feel and user comfort. It was an important goal to accomplish to create an appeal towards the product.

10.2. Implementation

10.2.1. Integrate Historical Data

In order to understand the data flow for health data in this project. It is vital to update the deployment diagram to show the additional layers of security added in this phase. In Figure 10.1 below, the updated deployment diagram is shown with an updated database and a custom REST API (Representational State Transfer Application Programming Interface).

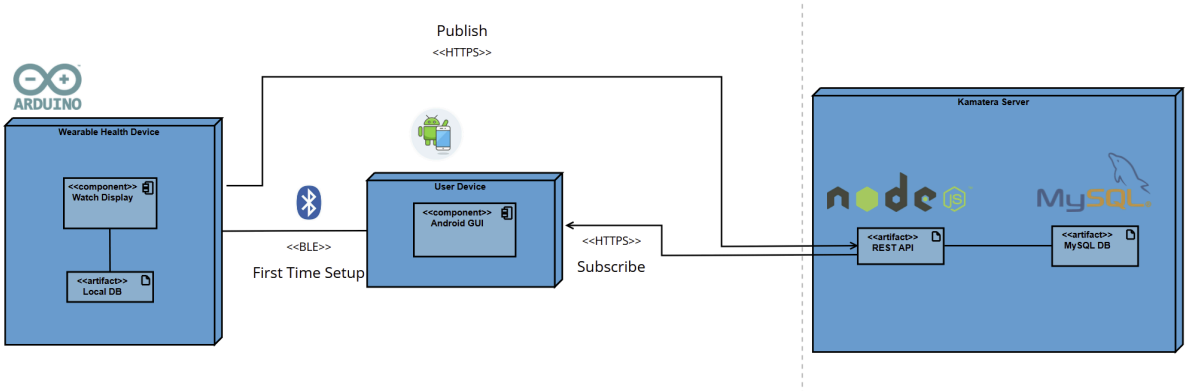


Figure 10.1. Updated deployment diagram

It is evident that there is an additional element in the database; NodeJs, which is used to build APIs that ensures that users are not able to directly execute queries on the database. There should be a layer in place that protects malicious users from injecting their own code into the database which is the purpose of the API. The shift to the MySQL database allowed our data model to simplify significantly from Google Firebase's JSON (JavaScript Object Notation) architecture to rows and columns, which allowed for data to be logged much more simply and processing the data on the server side became far easier.

Due to the limitation of the processing power on both the application as well as the wearable device, it is best to offload calculations on historical data to be performed on the server side, which is where the API comes in. REST APIs support a variety of different method types such as GET, POST, UPDATE, DELETE, which provides the application and the wearable with a very simple interface to execute methods on the server. For example, if the application needs to receive averages over the last month, the data can be retrieved and processed all on the server in one simple GET request rather than performing all the processing on the application or the wearable.

In order to implement this, a cloud provider called Kamaterra was used to create a free MySQL server which contained the data model that is visible in Figure 10.2 below.

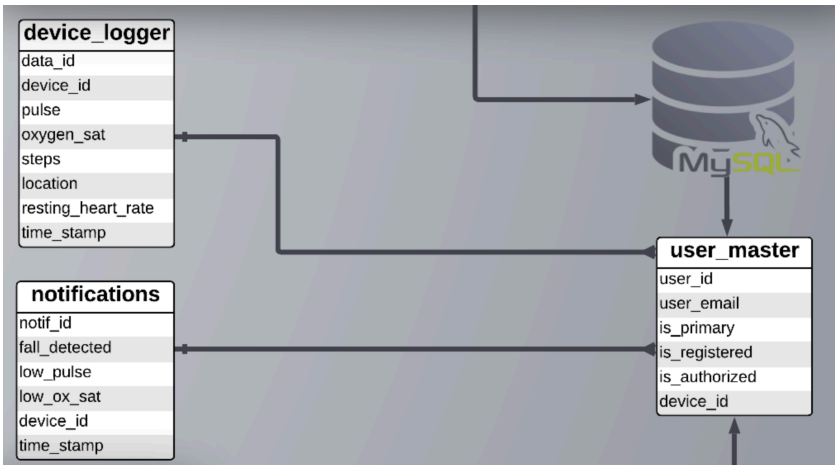


Figure 10.2. Data Schema for MySQL server

There are 3 tables that are present in the database, user_master which stores all user data, device_logger, which stores all logged data in a raw format, so that further processing is simple to implement and finally, the notifications table which stores all notifications raised by device.

On the same server as the MySQL server, a NodeJs server was created in order to build the REST API which the application and the wearable would interact with. GET methods are used to retrieve information from the server to the client and POST methods are used to send new information to the server. UPDATE methods are self explanatory, they update information and DELETE methods delete info. In Figure 10.3 below, a sample code snippet of a GET method that is implemented on the node server is provided.

```
app.get('/getDailyHighLowPulse', authenticateApiKey, (req, res) => {
    const email = apiKeyMap.get(req.headers['x-api-key'])
     const query = 'SELECT MAX(pulse) AS max_pulse, MIN(pulse) AS min_pulse,
DAYNAME(time_stamp) as day_of_week FROM (SELECT dl.pulse, dl.time_stamp FROM
WHIT.device_logger dl INNER JOIN WHIT.user_master um on dl.device_id =
um.device_id WHERE um.user_email = ? AND dl.time_stamp >= DATE_SUB(NOW(),
INTERVAL 7 DAY)) AS subquery GROUP BY DAYNAME(time_stamp);';
    db.query(query, [email], (error, results, fields) => {
        if (error) {
            console.error('Could not get daily high/low');
            res.status(500).json({'message' : 'query error'});
            return;
        }
        res.json({"results" : results});
    });
});
```

Figure 10.3. GET method implementation in NodeJs REST API

The most obvious thing to notice in Figure 10.3 is the call to the function authenticateApiKey. This is the first thing that happens when a request is made to the server. The server checks if the device requesting to execute this method has an API key.

When a user is created on the application, the app calls a register method on the server over HTTPS which is a secure method of sending data over the internet as the requests are authenticated using SSL (Secure Sockets Layer) certificates. For the purpose of this project, self-signed certificates were used but normally a certificate authority would be responsible for ensuring that requests made are secured.

The register method uses the email that is passed to it and generates an API key for that email which essentially binds an API key to an email which ensures that users are not able to access each other's information from any request. These generated API keys are then stored in an encrypted file which is encrypted using AES (Advanced Encryption Standard) to ensure that even if the server is compromised, API Keys and therefore users, would not be compromised.

To recap, requests are made securely over HTTPS to the server with an API Key that gets generated upon sign up. That API Key is then shared with the wearable over bluetooth for initial sign up which

is also encrypted. All subsequent methods are made simply by passing the API Key to the server and the REST API validates the API Key and returns the results of the queries that it executes. If parameters are passed to the server, those details are passed in a query using parameterized variables, which prevents SQL injection attacks. Overall these 3 layers of security help to minimize the risk of sensitive health data being compromised.

There is a suite of methods implemented in the server for the application to retrieve historical data and update other user data in the database via the REST API. This allows for calculations such as heart rate averages for example, to be offloaded from the client side to the server side which frees up resources for the user experience.

10.2.2. Google Geolocation API

To implement GPS tracking in this project, the ESP32-S3 microcontroller uses the Google Geolocation API to get accurate GPS information using WiFi and a series of processes. This program was created on the Arduino platform of the device.

Firstly, the ESP32 device sends a request containing information on nearby Wi-Fi access points. This information includes MAC addresses and signal strengths of nearby routers. These serve as reference points for the API to triangulate the device's approximate location based on signal strengths and known locations of the reference points, all through Google's database. After receiving the request, the Google Geolocation API utilizes complex algorithms to analyze the provided data and calculate the device's geographical coordinates. This process involves comparing the signal strengths received from nearby Wi-Fi access points and cellular towers with a database of known locations. By associating this information with the database, the API can determine the device's location with a high grade of accuracy.

Once its calculations are complete, the Geolocation API returns a JSON response to the device, containing the estimated latitude and longitude coordinates, and an accuracy radius indicating the margin of error for the calculated location. It also returns other information such as the current time, date and specifics of the calculated location, such as country, city, zip code and more. Aligning with the requirements of the project, the team decided to take only the longitude and latitude coordinate values from this program. These numerical values are sent over WiFi to the application to be displayed on the user interface, to track the user's position.

10.2.3. Update the Prototype CAD Design

The requirement for the latest CAD design was to fit the ESP-32 S3's inherent shape and keep the circular shape, as seen on common watches available on the market. Similarly, due to the shape of the microcontroller, one side of the watch casing has a rectangular protruding end. As a team, we decided to keep this feature of the microcontroller and reflect it in the CAD design as it also functioned as a port to the USB-C battery recharger circuit. As such, all elements of the design were made to aid the functionality of the product without making it too bulky or adding unnecessary extra components. The entire watch casing was 3D printed using an outsourced Markforger printer. The material that the watch casing was made in is Onyx. This material is much stiffer and stronger than commonly used Polylactic Acid (PLA) or Acrylonitrile Butadiene Styrene (ABS), and also ensures that the print is precise and its edges are smooth, as compared to using cheap materials. This ensured that the design created was simple, comfortable and elegant looking. It also makes the device and product design more appealing to the user wearing and using it. The final CAD casing design is shown below.

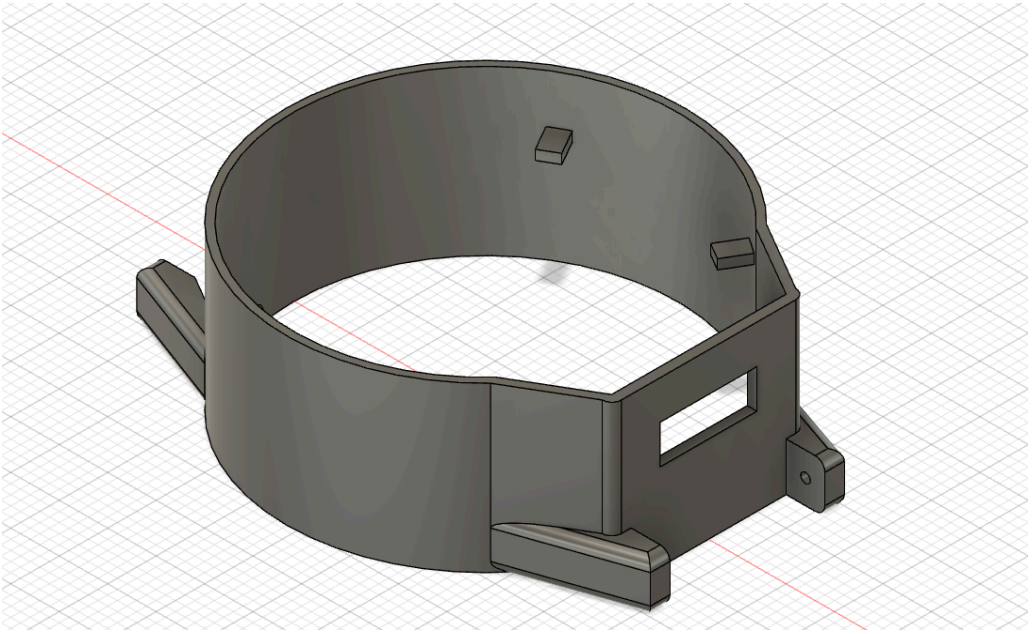


Figure 10.4. Final CAD Design for Body of Watch Casing

The main body of the watch casing, as depicted in the figure above, was made to be much shorter and smaller in size than the one created previously. This watch casing had a length of 53mm, a width or diameter of 41mm and a height of 17mm. Similar to the previous version of the watch casing this also has a rectangular hole for the USB-C charger hole, to recharge the device as needed. The side lugs on this design are only at the bottom of the main body, this makes the model compatible with almost all watch straps found online. To make sure that the watch straps and the associated bar would work with this model, the distance between the hole in the opposing lugs is around 18mm, which is proportional to 16mm width watch straps. Furthermore, the side lugs are also rounded out to enhance the aesthetic of the model and reduce the chances of the user getting nicked or cut due to the sharp edges of the model. The same was done with all the sharp-looking edges of the design. In addition, inside the main body, there are also small protruding notches. these were designed to hold the display of the ESP-32 microcontroller to the top surface of the casing. It enhances the look of the watch and makes interfacing with the touch LCD much easier for users - as opposed to having to insert one's fingers into the casing.

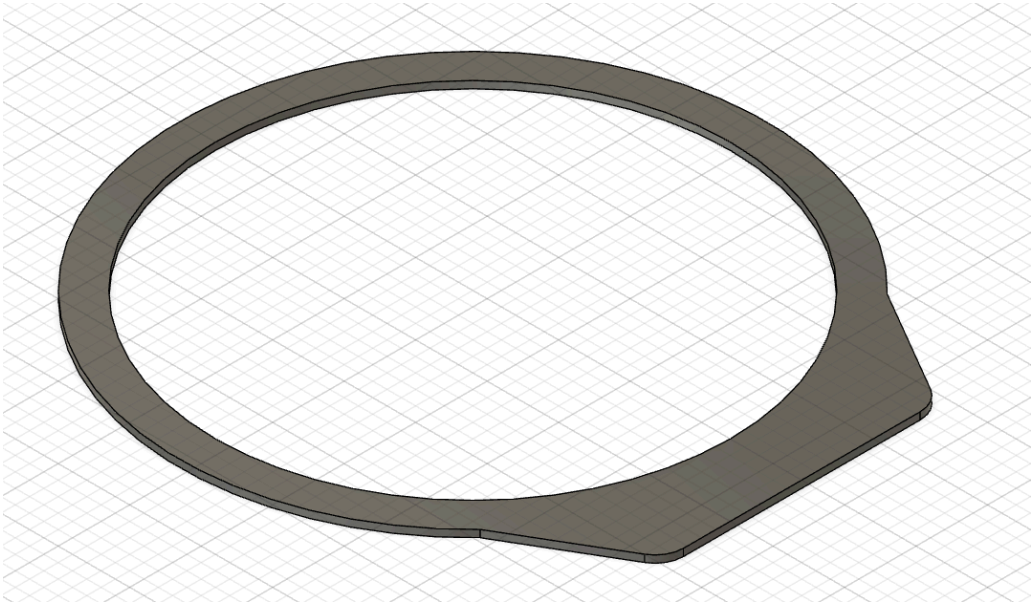


Figure 10.5. Final CAD Design for Top Lip of Watch Casing

The top lip of the watch casing shown above was made mainly for visual purposes, and to aid the aesthetic of the design. As the microcontroller's LCD display was pushed to the top of the casing, it created a small problem as the display's edges were still visible from certain angles and gave the overall design a somewhat unfinished look. To solve this problem, the team added a small lip to the top of the casing that covered the display edges and made the design look sleek and clean. The lip not only improved the design but also protected the display from damage.

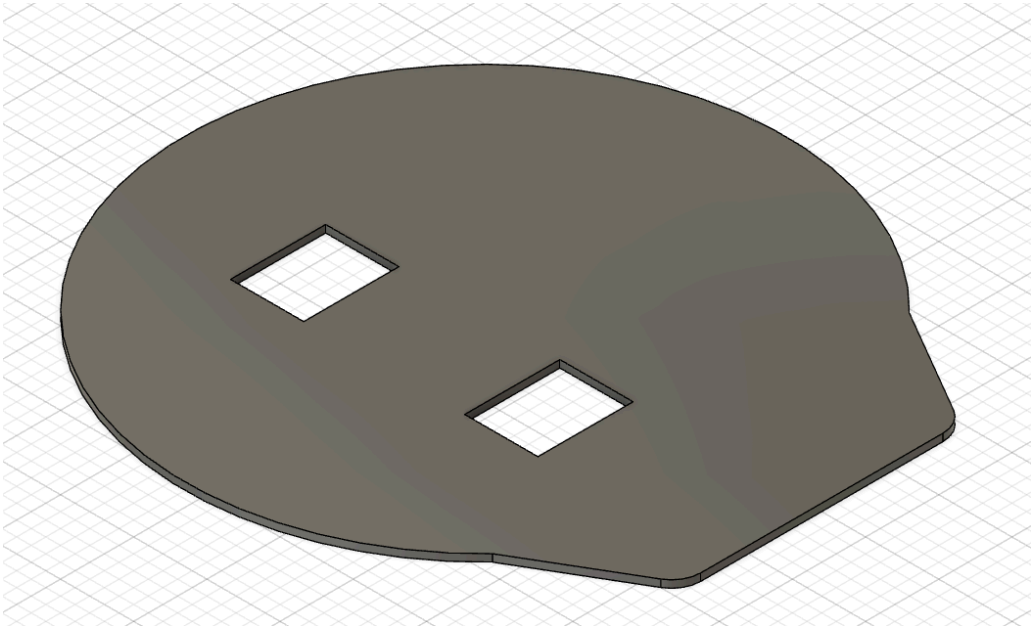


Figure 10.6. Final CAD Design for Bottom Surface of Watch Casing

The bottom surface of the watch casing was a critical feature of the model. The requirements for this model were to allow the MAX30102 pulse oximeter sensors to be able to sit flush with the user's skin while being sturdy and thick enough to uphold all the components of the device without any object underneath as support. This was a difficult requirement since the width of the sensors is about 1mm which meant that the thickness of the piece would have to be smaller to allow the sensor through.

Keeping that in mind, the team decided to keep the thickness of the bottom surface to 0.8mm. This allowed the piece to have some sturdiness and slo flexibility without snapping too easily. In addition, the model also showcases two equally sized holes for the sensors to be put through. They were measured to be the exact length and width dimensions of the sensor so that there are no unnecessary gaps in the piece. This further kept the model looking clean and elegant.

11. Safety

11.1. Product Safety

Ensuring product safety for the WHIT required careful measures for each component of the overall device.

For the rechargeable lithium polymer battery used, the safety measures taken included selecting a reputable manufacturer that adheres to the industry standards. Fortunately, the battery chosen has safety features built into the battery. It utilizes the Seiko IC chip that helps prevent the battery from being overcharged or overcurrent, which can damage the battery or even cause it to catch fire. It also protects against short circuits, which can be dangerous. Furthermore, the battery unit includes high-temperature resistant silicone cables that are designed to withstand high temperatures without melting or degrading. This is important to ensure safe operation, especially if the device generates a lot of heat during use.

With the ESP32 microcontroller, safety measures involved exhaustive testing of the firmware to prevent potential software exposures and using secure communication protocols to protect user data. In terms of physical safety, the microcontroller board utilized a 12-pin jumper cable to connect to the pulse oximeters. Since the implementation only required 4 GPIO pins to be used for the sensors and 2 for ground and power respectively, the rest of the wires were cut short, dipped in hot glue (as it is a good insulator) and covered with electrical tape to prevent possible short circuits.

For the pulse oximeters, extensive testing, including stress and environmental testing, was conducted to identify and address any safety concerns. The sensors were surrounded with hot glue to ensure that they stayed in place inside the device, and were further blanketed in electrical tape to prevent short circuits with other components.

Overall, the team was able to produce a sound and standardly safe product without compromising the device's performance or structure.

11.2. Manufacturing Safety

All members of the team received extensive training in working with electrical components and tools such as the soldering equipment, wiring and other instruments. All electrical connections and device manufacturing was done in a lab room that included safety equipment such as fire extinguishers, heat-protective gloves, safety glasses, etc. This was done in the Maker Lab of Carleton University.

12. Testing and Validation

12.1. Hardware Laboratory Testing

The group conducted various lab tests to ensure the functionality, reliability, and safety of the hardware components. Key testing categories:

- Functional Testing: This involved testing each hardware component, such as the pulse oximeter's ability to accurately measure blood oxygen saturation and heart rate.

- Intra-Device Testing: These tests were done to ensure a seamless operation among all hardware components. For example, the team ensured that the IMU and Pulse Oximeter were able to work simultaneously while both sensors used the one I2C bus of the ESp-32.
- Reliability Testing: Stress tests were done to ensure the hardware's reliability over time and under different conditions. This included subjecting the Fall Detection algorithm to vigorous movements to ensure that its alert system functions when intended.
- Durability Testing: The durability of hardware components was a huge priority to the team, as this product is meant to be worn through all kinds of daily activities. For example, the CAD casing material used is resistant to scratches and breaking on impact.
- Safety Testing: The team ensured compliance with safety standards and regulations, for each component and the device as a whole. This included ensuring user safety with proper insulation and protection against electric shocks for the jumper wires using electrical tape.
- Usability Testing: The group focused a lot on testing the user experience, including the readability of displays and the comfort of wearing the watch.

Through the extensive lab tests, the group was able to ensure that the project's hardware components met quality standards, performed reliably when in use, and were safe for use by users.

12.2. Software Unit Testing

Unit tests are a crucial component of the software development process. They play an important role in ensuring the reliability, maintainability, and robustness of software.

- **Early Detection of Issues:** Unit tests were written to check the individual units of code in isolation. They help in the early detection of bugs, making it easier to fix issues before they feed into other parts of the software.
- **Regression Testing:** Unit regression tests act as a safety net when new changes are made to the codebase. By running these tests after each code change, the team was able to quickly identify if new code broke existing functionality, ensuring that the software remained stable throughout the development process.
- **Enhanced Code Quality:** Writing unit tests forced the team to think about code design, structure, and modularity. Resulting in cleaner, more maintainable, and better-structured code.
- **Documentation**: Unit tests serve as live documentation for the codebase. It provided clear examples of how different functions and components are intended to work.
- **Faster Debugging**: When any unit test failed, it provided immediate feedback about the specific issue, making debugging much faster and more efficient for the team.
- **Support for Refactoring**: Unit tests make it safer to refactor code. The team was able to develop and make new changes with confidence, knowing that if their modifications break existing functionality, the unit tests will catch it.

Development of Unit Tests:

- Test Framework Selection: the framework used for the application is JUnit. To test the ESP-32, the team implemented incremental functional testing.
- Test Coverage: Critical sections of the code were covered by unit tests. This included functions, methods, or classes that perform specific tasks or calculations, such as, the averaging of Heart Rate measurements from the sensors.
- Creation of Test Cases: The team wrote test cases for each unit of code. A test case typically consisted of simulation input data, expected outputs, and assertions to verify that the code behaves as expected.

- Consider edge cases, boundary conditions, and typical usage scenarios.
- Test Maintenance: Unit tests were maintained alongside the code being tested. When code changes are made, updates to the corresponding unit tests are made to reflect the changes in functionality.
- Code Reviews: The team incorporated regular code reviews into the development process to ensure that unit tests are adequately covering the code.

By incorporating unit tests into the software development process, the team not only improved the quality of the software developed but also streamlined the process, reducing debugging time, and built a more robust and maintainable codebase.

12.3. User Trials

Due to time constraints, the team was unable to perform user trails on volunteers or students of the University, given their consent. An outline of the proposed user trials is written below.

1. User Trial Execution:
    - We planned to recruit various participants and provide them with data collection tools. Instruct them to use the tools in their daily routines and tracking relevant health metrics.
2. Data Collection and Analysis:
    - Collect user-generated data and feedback during the trial.
    - Analyze the data to evaluate system performance, user experience, and accuracy, including comparing user-generated data with reference data, if available.
3. Iterative Improvement:
    - Use insights from the trial data and feedback to make iterative improvements to the system. Address identified issues, enhance user experience, and ensure ongoing accuracy.

13. Project Timeline and Progress

13.1. Estimated Timeline

Milestone	Description	Assignee	Deadline
Overall Project Plan	Create a project plan and discuss goals to achieve through design. Decide who the target audience is and alter design accordingly. Discuss further innovation ideas.	P: Aryan Mathur S: Aryan Laxman	September 2023
Finalize components and design choices	Discuss the components available online and on-campus and choose appropriately for design form factor	All members	October 2023
Project Proposal	Create a project proposal that outlines design, methodology and timeline	P: Nahreen Tahmid S: Aryan Mathur	October 2023
Connect and Build device for initial testing	Start creating flexible hardware connections and check for data transfer and size issues	P: Nahreen Tahmid S: Aryan Mathur	November 2023
Integrate software onto device	Implement the software into the device and test operation.	P: Aryan Laxman S: Aryan Mathur	November 2023
Build software to monitor and analyze data	Develop, build and test an application that can monitor the device's operation	P: Aryan Laxman S: Aryan Mathur	December 2023
Finalize first functional	3D print and compartmentalize the first	All members	December 2023

prototype	prototype		
Adjust device design	Adjust required software, hardware and design according to design requirements and feedback from supervisors	P: Aryan Mathur S: Aryan Laxman	January 2024
Start user testing	Start user testing for functionality, wearability and durability	P: Aryan Mathur S: Nahreen Tahmid	February 2024
Finalize device build, software and other factors	Create the final product using user feedback and making the required changes for improved use.	All members	March 2024
Final Report	Create the final report encompassing the entirety of the project's development.	All members	April 2024

Table 13.1. Timeline Descriptions and Deadlines

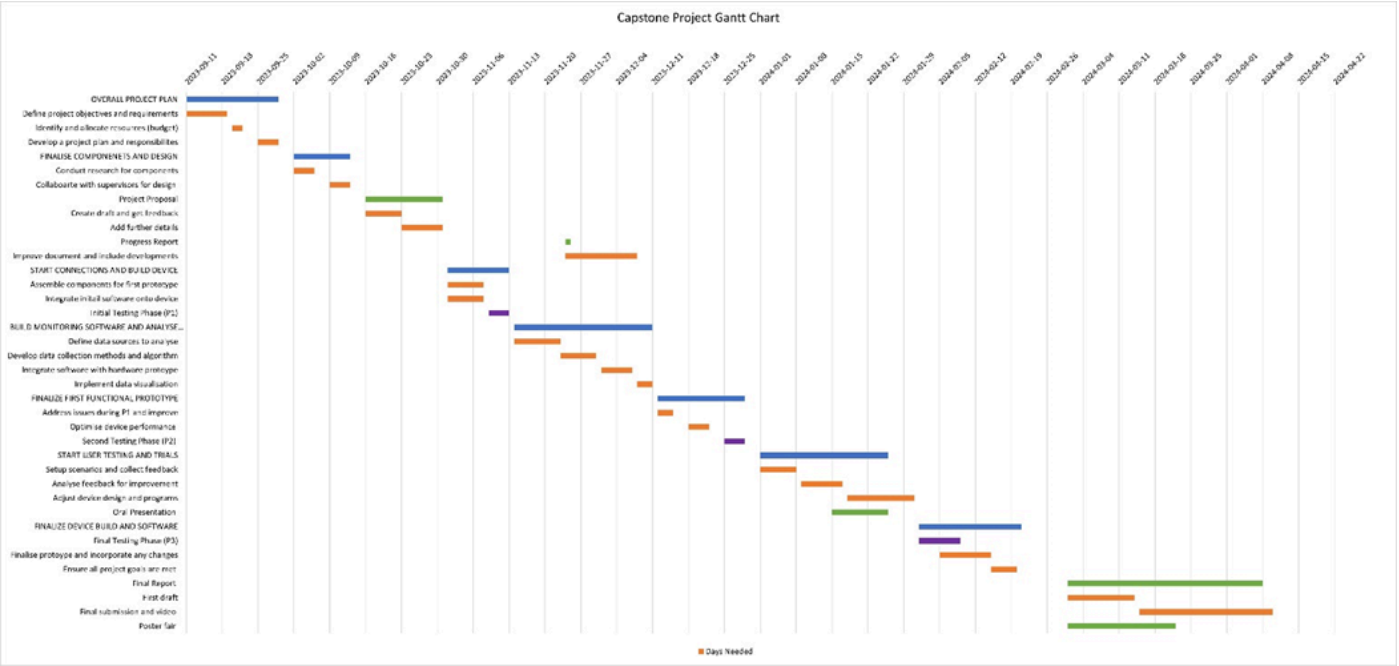


Figure 13.1. Gantt Chart of Project Timeline

13.2. Budgeting

13.2.1. Itemized Budget for Components

Item	Amount Required	Unit Price ($)
Arduino Nano BLE Sense Rev 2	1	58.03
Pulse Oximeter	2	27.52
Adafruit GPS Module	1	42.91
LiPo Battery	1	9.95
Recharging Module	1	5.95
ESP-32 S3 Touch LCD 1.28	1	40
RP2040	1	30

Solid Core Wires	1	25
3D Printed Casing	1	16

Table 13.2 Itemized Budget

13.2.2. Laboratory Testing Expenses

As all team members have access to the laboratory and all its equipment and tools in the Carleton University maker lab, all expenses related to testing and hardware building will be of little to no charge.

14. Project Management

14.1. Team and Organization

The team consists of three members: Aryan Mathur, Aryan Laxman Sirohi and Nahreen Tahmid. All members of the group are students at Carleton University as full-time fourth-year students. Aryan Laxman and Aryan Mathur are both studying Computer Systems Engineering. This gives the two members a strong foundation in embedded systems and software development required to complete this project. Nahreen Tahmid is studying Biomedical and Electrical Engineering. This gives her a strong understanding of project design, research and engineering circuitry.

The supervisors for this Capstone project are Professor Shaghayegh Gomar and Professor Carlos Rossa. Our supervisors have been helpful throughout the course of our capstone project. They've helped us by providing the resources needed and providing useful advice. With their guidance, our project has been a success, and we've learned a lot from them. We're thankful for all their support.

This project combines all the foundational aspects of both engineering majors. Its implementation of hardware using microcontrollers and compatible devices uses the traditional computer systems course’s projects, as well as its software development. Similarly, the health data collection and analysis included in the project is an extension of what the biomedical research and instrumentation courses include.

14.2. Relevance to Academic Programs

All members of the group are students at Carleton University as a full-time fourth year student. Aryan Laxman and Aryan Mathur are both studying Computer Systems Engineering. This gives the two members a strong foundation in embedded systems and software development required to complete this project. Nahreen is studying Biomedical and Electrical Engineering. This gives her a strong understanding of project design, research and engineering circuitry.

Aryan Mathur has exposure to backend programming in C#, C++, Java, JavaScript, Python, and SQL along with website development using .NET MVC, which includes a strong grasp of Test-Driven Development (TDD). Aryan Laxman’s strengths lie in designing and developing projects using Python and real-time control systems using Java. Nahreen has extensive experience using Verilog, C and Python in designing and verifying a system’s functionality and has worked on projects in collecting health data acquisition. Each member of the group has a good grasp of circuitry, prototyping and testing through course work and

This project combines all the foundational aspects of both engineering majors. Its implementation of hardware using microcontrollers and compatible devices uses the traditional computer systems course’s projects, as well as its software development. Similarly, the health data collection and analysis included in the project is an extension of what the biomedical research and instrumentation courses include.

14.3. Teamwork Strategy

Our team intends to follow a weekly update schedule along with brief meetings with all the members on a regular basis to keep everyone updated on any progress or issues as they arise to mitigate future collateral damage or delay in the project's development. Meetings with our supervisors are also held on a weekly basis to get constant feedback and continue a good line of communication to allow for steady improvements and feedback.

All meeting descriptions, agenda and times are recorded by Nahreen on a document consisting of relevant papers, documents created and topics of discussion. All code and programs created are shared over a common git repository by all members of the team. The team holds regular weekly meetings over Discord to discuss and collaborate on current tasks and share progress.

Git Repository Link: https://github.com/nahreen02/capstone
Application GitHub Repo: https://github.com/laxman-22/Capstone

15. Conclusion

15.1. Summary

In conclusion, the development of our biodata tracking device represents a significant stride towards enhancing the quality of life for individuals with dependents who are vulnerable to health conditions, sudden physiological changes, or chronic diseases. By continuously monitoring heart rate and other vital signs, our innovative device empowers caregivers and healthcare providers to detect potential issues in real-time, enabling early intervention and tailored care [3]. As we move forward, the potential applications of this technology are boundless, promising a future where proactive health management becomes a reality, ensuring the well-being and safety of our loved ones. This project is not just about technology, it's about providing peace of mind, improving healthcare, and fostering a more caring and connected society.

15.2. Importance in Healthcare Sector

This project holds ample importance in the healthcare sector due to its potential to revolutionize patient care and management. In a time where healthcare resources are often strained, and the population is growing rapidly, the power to remotely monitor an individual's vital signs and health data becomes an essential tool for healthcare providers. It allows for early detection of health issues, facilitating timely interventions and easing the burden on the healthcare system [16].

Moreover, for individuals with dependents susceptible to health conditions, such a device offers peace of mind, ensuring their loved ones are safe and well cared for. Additionally, the continuous data generated by these devices can contribute to better personalized and data-driven healthcare, enhancing treatment plans and results [17]. In essence, this project bridges the gap between healthcare and technology, offering a promising route for improving the overall well-being of individuals and the efficiency of healthcare delivery systems.

16. References

[1] T. H. Tulchinsky and E. A. Varavikova, “Measuring, Monitoring, and Evaluating the Health of a Population,” The New Public Health, pp. 91–147, 2014, doi: https://doi.org/10.1016/b978-0-12-415766-8.00003-3.

[2] “Tapping the Potential of Wearables in Early Disease Detection,” Fitbit Health Solutions, Oct. 12, 2023. https://healthsolutions.fitbit.com/blog/tapping-the-potential-of-wearables-in-early-disease-detection/ (accessed Oct. 28, 2023).

[3] L. Lu et al., “Wearable Health Devices in Health Care: Narrative Systematic Review,” JMIR mHealth and uHealth, vol. 8, no. 11, p. e18907, Nov. 2020, doi: https://doi.org/10.2196/18907.

[4] K. Yasar, “What is wearable technology? - Definition from WhatIs.com,” SearchMobileComputing, May 2022. https://www.techtarget.com/searchmobilecomputing/definition/wearable-technology

[5] “Save Big & Bring Joy.” Fitbit Official Site for Activity Trackers & More, www.fitbit.com/global/en-ca/home. Accessed 8 Dec. 2023.

[6] H. S. Kang and M. Exworthy, “Wearing the Future—Wearables to Empower Users to Take Greater Responsibility for Their Health and Care: Scoping Review,” JMIR mHealth and uHealth, vol. 10, no. 7, p. e35684, Jul. 2022, doi: https://doi.org/10.2196/35684.

[7]“Apple Watch SE,” Apple (Canada). https://www.apple.com/ca/apple-watch-se/

[8] “Fitbit LuxeTM,” Fitbit.com, 2024. https://www.fitbit.com/global/en-ca/products/trackers/luxe?sku=422BKBK (accessed Apr. 11, 2024).

[9] “Apple Watch - Battery,” Apple (Canada). https://www.apple.com/ca/watch/battery/

[10] S. Harding, “‘Time to move on’: Fitbit owners fed up with battery problems, Google response,” Ars Technica, Jan. 31, 2024. https://arstechnica.com/gadgets/2024/01/time-to-move-on-fitbit-owners-fed-up-with-battery-problems-google-response/

[11] S. Canali, V. Schiaffonati, and A. Aliverti, “Challenges and recommendations for wearable devices in digital health: Data quality, interoperability, health equity, fairness,” PLOS Digital Health, vol. 1, no. 10, Oct. 2022, doi: https://doi.org/10.1371/journal.pdig.0000104.

[12] “Introducing Our New Markforged Material: Onyx,” Markforged. https://markforged.com/resources/blog/introducing-our-new-markforged-material-onyx

[13] Altexsoft, “Functional and Non-functional Requirements: Specification an,” AltexSoft, Nov. 30, 2023. https://www.altexsoft.com/blog/functional-and-non-functional-requirements-specification-and-types/

[14] “iPhone vs. Android User & Revenue Statistics (2024),” Backlinko, 13-Mar-2024. [Online]. Available: https://backlinko.com/iphone-vs-android-statistics. [Accessed: 10-Apr-2024].

[15] J. Howarth, “iPhone vs Android User Stats (2023 Data),” Exploding Topics, Oct. 13, 2023. https://explodingtopics.com/blog/iphone-android-users

[16] E. Wittenberg, A. Saada, and L. A. Prosser, “How Illness Affects Family Members: A Qualitative Interview Survey,” The Patient - Patient-Centered Outcomes Research, vol. 6, no. 4, pp. 257–268, Oct. 2013, doi: https://doi.org/10.1007/s40271-013-0030-3.

[17] W.-H. Wang and W.-S. Hsu, “Integrating Artificial Intelligence and Wearable IoT System in Long-Term Care Environments,” Sensors, vol. 23, no. 13, p. 5913, Jan. 2023, doi: https://doi.org/10.3390/s23135913.