# Cover Page

# 

# SMARTCYCLE:

# AN OUTDOOR WORKOUT WITH RESULTS

# Team:

# Liz Callahan

# Jimmy Duke

# Alex Kaup

# Joe “Shaggy” Lubetski

# John Sexton

# Table of Contents

[**Cover Page 1**](#_heading=h.gjdgxs)

[**2 Table of Contents 2**](#_heading=)

[**3 Introduction**](#_heading=) **3**

[**4 Detailed System Requirements**](#_heading=) **6**

[**5 Detailed project description**](#_heading=) **8**

[5.1 System theory of operation](#_heading=) 8

[5.2 System Block diagram](#_heading=) 10

[5.3 Detailed design/operation of GPS](#_heading=) 10

[5.4 Detailed operation of Accelerometer](#_heading=) 14

[5.5 Detailed operation of Heart Rate Sensor 17](#_heading=h.9mo45rccd8g7)

[5.6 Detailed operation of LCD 22](#_heading=h.oihmfgo8q8ti)

[5.7 Detailed operation of Bluetooth Unity App 24](#_heading=h.5w4jzfqjbwl2)

[5.8 Interfaces](#_heading=) 30

[**6 System Integration Testing**](#_heading=) **33**

[**7 Users Manual/Installation Manual**](#_heading=) **36**

[7.1 How to install your product](#_heading=) 36

[7.2 How to setup your product](#_heading=) 36

[7.3 How the user can tell if the product is working](#_heading=) 37

[7.4 How the user can troubleshoot the product](#_heading=) 37

[**8 To-Market Design Changes**](#_heading=) **37**

[**9 Conclusions**](#_heading=) **45**

[**10 Appendices**](#_heading=) **46**

# Introduction

*3.1 Problem Description*

In a world of rapid technological advancement and a society of increasingly sedentary human existence, maintaining a regular and healthy pattern of physical activity has become more important and more of a conscious effort than ever before. Exercise is important not just for everyone’s physical health, but for their mental health as well. As people’s lives become increasingly compartmentalized, it is also essential and prudent with current technology available that they can understand exactly what health benefits they receive from the method of exercise they choose.

Among the rapidly increasing quality, accuracy, and user-friendliness of fitness technology, one classic example has always stood out as an extremely convenient and marvelous tool – the stationary exercise bike. What makes these bikes stand out from other fitness tech is their excellent integration of sensors, advanced computation, and appealing user interfaces to give the average gym attendee a more informed workout – often displaying several statistics such as current heart rate, speed (from rpm), theoretical distance traveled, and many others.

These exercise bikes are great for boosting one’s cardiovascular health and overall wellbeing in an informed way; however, not everyone finds riding a stationary device inside for an extended period of time to be particularly enjoyable. Some may wish for a more mobile, dynamic, or even outdoor experience with which to get their exercise, something a typical exercise bike cannot provide. However, a standard bicycle cannot provide the helpful information that these exercise bikes do so effectively without some kind of external system or modification. There are existing products that can accomplish this, such as smart watches with heart rate and location tracking capabilities, but these tend to be rather expensive. Moreover, stationary exercise bikes themselves also tend to be rather expensive – for example, higher-end products such as Peloton and SoulCycle bikes can range in price from $1,500 to $2,500. These products also tend to take up a lot of space that one might not have readily available in their home as they might for an outdoor bicycle.

Our project, the SmartCycle, seeks to provide the user with the best of both worlds. It would provide the user with accurate and helpful information one might obtain through riding an exercise bike while also allowing them to enjoy the outdoors and a more dynamic experience. It would also save them money and space compared to the alternatives. Another benefit of this sort of system is the ability to use it as a means of transportation in certain places, meaning that one could theoretically get in a cycling workout with accurate data while also getting them to their destination, rather than having to purely compartmentalize this aspect of their daily routine. Overall, this product could make one’s cycling/cardiovascular exercise routine more enjoyable, more convenient, and more accessible.

*3.2 High Level Description of System*

The SmartCycle uses data from several different sensors and devices, all connected using a central microprocessor. The main devices at play are a heart rate monitor, a GPS device, and an accelerometer. The means of display consist of both an LCD screen that is directly connected as well as a smartphone app that is connected wirelessly through Bluetooth. The heart rate sensor is located on one of the handlebars of the bike, while the LCD screen, main board, and other devices are encased together and sit atop the frame of the bike just below the handlebars for the rider to easily view. The compactness of the system makes it easy to install and remove and does not affect the agility or safety of the rider at all. Ideally, the user would be able to see on the screen their current heart rate (updated every couple of seconds) obtained from the heart rate monitor, their distance traveled and speed in miles per hour both calculated from GPS data, and finally the angle of incline that the bike is traveling obtained from the accelerometer. In addition to this data, the phone app displays miles-split, calories burned and average heart rate. It also stores this data for the last five rides that the user logs into the system. All of these properties of the system give the rider an experience that is easy, enjoyable, and well-informed.

*3.3 Expectation vs. Reality*

Overall, the system connected very well and all devices and sensors were able to gather and display data in some form. In terms of different subsystems, some aspects of the final product ended up working more effectively or accurately than others. The GPS gave fairly precise coordinates, and with the right methodology, the distance traveled and current speed displayed when riding the bike ended up being very accurate. The accelerometer would also give very accurate angle data whenever we tilted the bike up and down and would update frequently, not getting stuck on certain values.

The heart rate monitor, however, did not produce the exact results we had hoped for. It would only detect an accurate or even reasonable heart rate level for certain people that we tested, and oftentimes would either default to the minimum or maximum heart rate that the program specified. It also measured far more accurately from the placement of a person’s thumb directly on the sensor than for the palm of their hand. Overall, it showed signs of promise, but could have been more reliable.

The LCD showed no major problems in the final product. The values we hoped to show were displayed in a clear and visually appealing way, and the values updated consistently in the time frames specified without any interruptions. The smartphone app also displayed and saved this same data correctly and consistently, provided the phone remained within a reasonable radius of the bike for bluetooth connectivity.

# Detailed System Requirements

1. General Requirements
   1. The system shall integrate with a general use bicycle without interfering with normal function
   2. The system shall not introduce any additional hazards to the cyclist
   3. The installation shall require no advanced tools
   4. The installation time shall be under 20 minutes
   5. The weight of the system shall not exceed 2.25kg so as to not unbalance the bicycle’s weight distribution
   6. The board shall be encased in a 3D printed box that connects to the LCD panel on the handlebars
   7. The wiring and sensors shall be encased in insulated material
2. Hardware Requirements
   1. The system shall measure data on bike speed to within 0.2 mph
   2. The system shall measure data on pedal speed to within 0.1 rps
   3. The system shall measure data on bike incline to within 5 degrees
   4. The system shall measure heart rate of the user to within 10 bpm
   5. The system shall contain an LCD display
   6. The system shall contain sufficient buttons to be able to fully control the LCD display
   7. The LCD panel shall be sufficiently large to display:
3. Total distance traveled
4. Average speed
5. Heart Rate
6. Time Elapsed
7. Calories Burned
8. Previous Mile Split
9. Device Battery Level
10. Software Requirements
    1. The microprocessor shall regularly command measurement updates from hardware
    2. The microprocessor shall save off sufficient data to calculate:
11. Total distance traveled
12. Average speed
13. Heart Rate
14. Time Elapsed
15. Calories Burned
16. Previous Mile Split
    1. The microprocessor shall be able to connect via Bluetooth to a single cellular phone to push data to it (within a range of 10m)
17. Environmental Factors
    1. The system shall operate unimpeded in temperatures from 32°F to 100°F
    2. The system shall operate in rainy weather
18. Power Requirements
    1. The device shall be powered using a 2Ah, 3.7V LiPo rechargeable battery
    2. The device shall be powered continuously for at minimum 8 hours
    3. The device shall be able to be turned off when not in use
    4. The device shall be able to be charged via a USB C charger

# Detailed project description

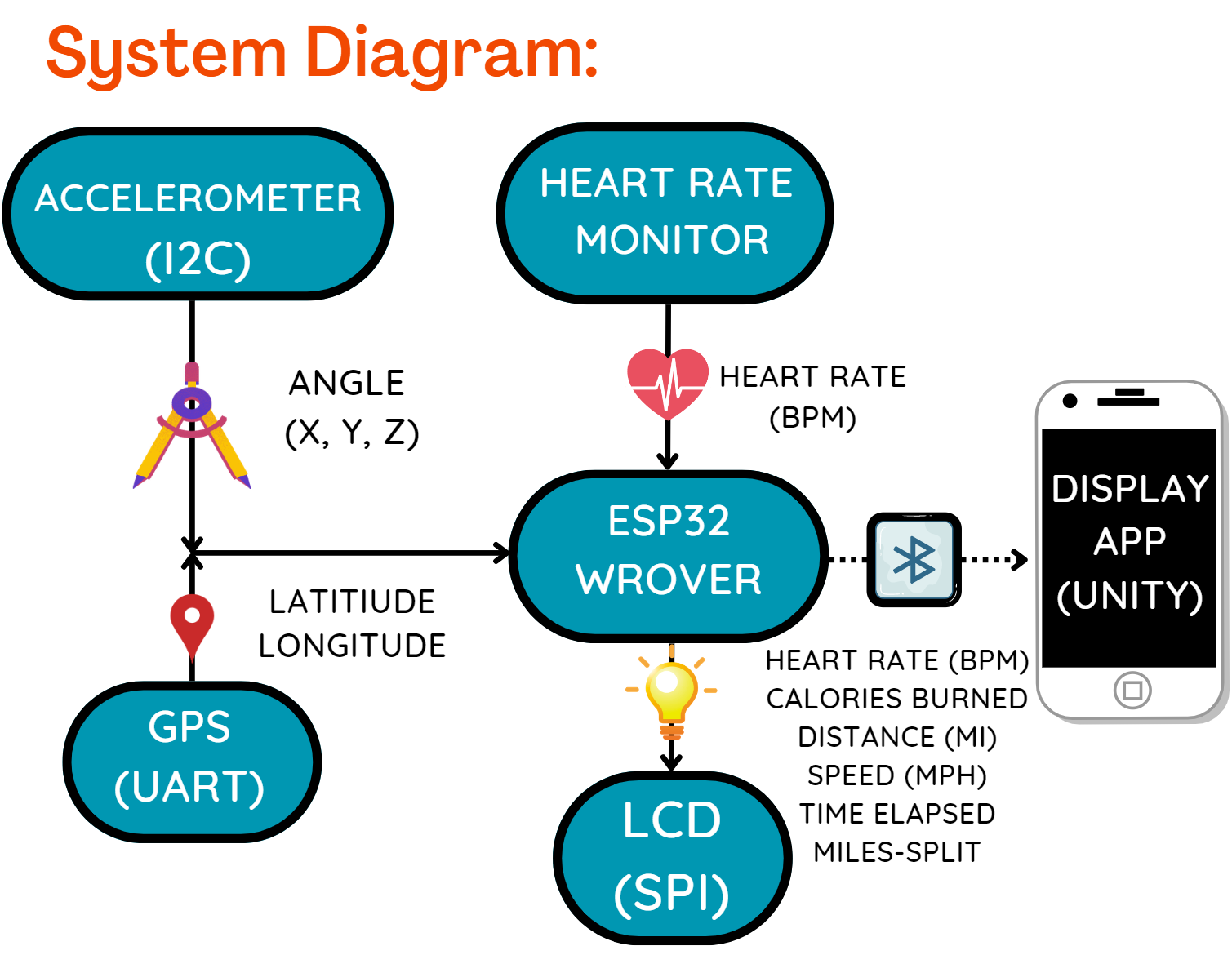
## System theory of operation

Our overall system is designed to be simple, easy to install, and easy to operate and understand from the user’s standpoint. The system contains one box that is attached to the center on the bike handlebars using a bike clamp, with one peripheral piece (the HRM) on the handlebar. Using the ESP32 WROVER microcontroller, the system takes in and parses data from the GPS sensor, accelerometer, and HRM, and outputs data to the LCD screen mounted on the bike, and the connected Unity app on the user’s iPhone. The LCD screen is set up to display the most important information for the user: the speed in miles per hour (MPH), the heart rate in beats per minute (BPM), the total distance traveled in miles, and the incline of the bike in degrees. The iPhone app, connected through Bluetooth, has the ability to show the user more data, as well as store workout data for future viewing. The app shows speed, heart rate, distance, time elapsed, mile split time, calories burned, and incline. It also stores the most five recent rides that the user can save using a button.



**Figure 1. Final Prototype of the SmartCycle System during Demonstration**

## System Block diagram



**Figure 2: System Block Diagram**

## Detailed design/operation of GPS

*5.3.1 Overview*

The Global Positioning System (GPS) FGPMMOPA6H chip is the part of the system being used to collect both location and speed data. It functions by using an antenna to transmit and receive signals to and from multiple GPS satellites which then triangulate the position. While the GPS is unable to receive signals from the satellites while inside of a building, it proved to be one of the most reliable and accurate sensors in the design once it was fully integrated.

*5.3.2 Requirements*

Some of the key requirements that directly impacted the design and choice to use the GPS were:

1.2 The system shall not introduce any additional hazards to the cyclist

2.1 The system shall measure data on bike speed to within 0.2 mph

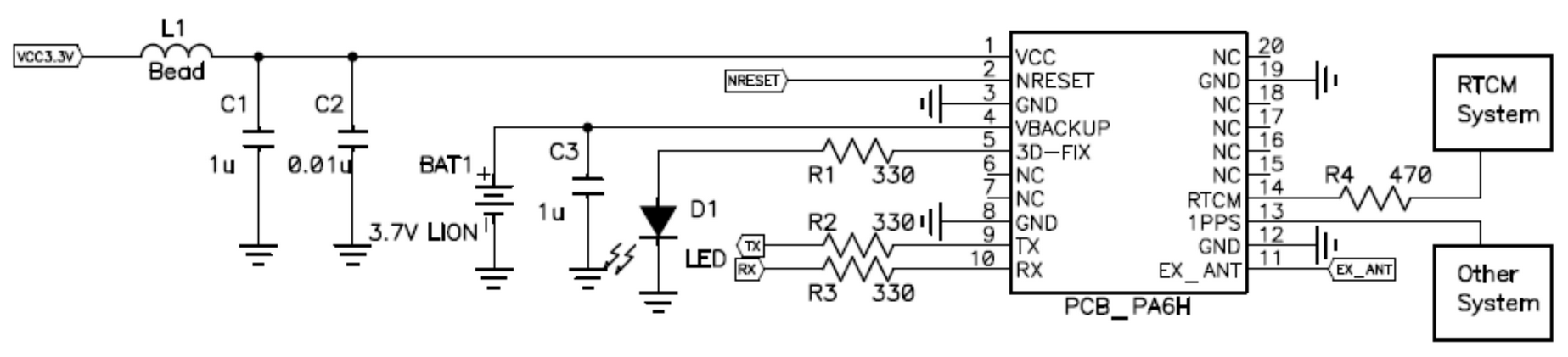
3.2 The microprocessor shall save off sufficient data to calculate:

A. Total distance traveled

In the early stages of development, the GPS chip was not included. Rather, a hall effect sensor was to be positioned near the wheel, and the number of rotations per minute was to be determined via the voltage spikes from magnets attached to the wheel. However, after more discussion of the mounting of the sensor and magnets, it was determined that the existence of more wires and connections near the wheels had the potential to introduce additional hazards to the cyclist, violating requirement 1.2. It was for this reason that the GPS was originally considered and tested. After some initial testing, it was determined that the GPS was quite accurate at reading the current location, and thus would be a good alternative to the hall effect sensor.

*5.3.3 Hardware*

Shown below in figure 3 is the reference circuit diagram that was followed to design our circuit board.



**Figure 3: GPS Reference Circuit**

As can be seen, there are very few externals other than a few capacitors to help keep VCC fully DC. In addition, pins 9 and 10 are TX and RX UART communications. As such, the communications between the GPS chip and the ESP was done through UART serial communications.

*5.3.4 Communications*

The exact messages received from the UART communications are in the form of the one shown below. They are based on National Marine Electronics Association (NMEA) standards for sending location data. One of the benefits of this particular GPS chip is that a library came with it. This library was helpful as it both did all of the string parsing automatically, and it had sample code which was incredibly helpful in getting the GPS initially started. The automatic parsing helped turn a message that is almost impossible to decipher into one that is obvious:

Unparsed Data:

$PGTOP,11,2\*6E

$GPGGA,050526.000,4141.8722,N,08614.4739,W,1,04,2.06,198.5,M,-33.9,M,,\*56

$GPRMC,050526.000,A,4141.8722,N,08614.4739,W,0.13,42.79,070323,,,A\*4A

Parsed data:

Time: 05:05:26.000

Date: 7/3/2023

Fix: 1 quality: 1

Location: 4141.8720N, 8614.4736W

Speed (knots): 0.13

Angle: 42.79

Altitude: 198.50

Satellites: 4

Antenna status: 2

This makes it incredibly easy to pull out specific data as needed without the headache of parsing the string manually. The library creates a GPS object, and from there it is possible to pull the latitude and longitude data (in degrees) by simply calling GPS.latitudeDegrees and GPS.longitudeDegrees.

*5.3.5 Software*

Within the software, the GPS data was constantly being parsed and the GPS object updated at a rate of 1 Hz. Every two seconds within the loop, the program then used the new data to update variables and displays. The current speed of the system was taken directly from GPS.velo and then converted to mph. Total distance traveled was calculated with a bit more logic. Every two seconds, the code compared the new location to the previous “anchor point.” It checked to make sure the new location was at least 0.01 miles away from this last known point. If it was, then the new location was saved as the new anchor point, and the distance traveled was incremented. If it was not, then the new location was ignored. This helped prevent the noise from the GPS being integrated and added to the total distance traveled.

*5.3.6 Subsystem Testing*

This subsystem was tested by taking it outside and watching the distance and speed data change as the user moved around. The distance data proved to be quite accurate, and the speed data was good, if a bit inconsistent.

## Detailed operation of Accelerometer

*5.4.1 Overview*

Within the design, the accelerometer is responsible for measuring incline with respect to flat ground. This incline data is displayed to the user, and in future iterations of the design will likely be used to calculate total distance ascended and descended as trip metrics. In addition to the IC chip being used containing an accelerometer, it also has a gyroscope. While the data being pulled to the ESP is just accelerometer data, the gyroscope is being used on the IC chip to eliminate some of the noise from the accelerometer.

In the original design, the sensor responsible for incline data was a gyroscope, however that proved to be unable to handle quick jerks and accumulated errors over time. In addition, before the accelerometer was finalized in the design, an attempt was made to use altitude data from the GPS. This was data that was provided by the GPS, and in theory knowing the altitude would just as easily help identify how much of the trip was uphill and downhill. However, after multiple tests, this too proved to drift too much to be useful. Even when placed stationary on the ground, the altitude data being given by the GPS varied by as much as 5 meters, and that doesn’t include the initial 10-15 meter calibration before the altitude settled. In the end, the accelerometer was decided upon as the solution which would most quickly and easily fix this problem.

*5.4.2 Requirements*

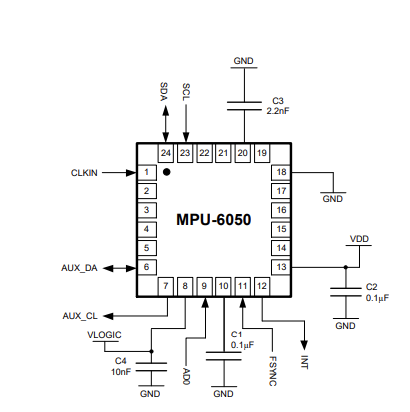
The only requirement driving the accelerometer design was:

2.3 The system shall measure data on bike incline to within 5 degrees.

This requirement was easily met through tests of it lying on a flat surface and reading 0, as well as at a 45-, and 90-degree angle and accurately reflecting the values to within 3 degrees.

*5.4.3 Hardware*

The reference circuit provided in the datasheet for the accelerometer is as shown below. One thing to note is that while there are a lot of pins on the chip, only a few of them ended up being used in the design. FSYNC, INT, AUX\_DA, and AUX\_CL were all not used. As such, our circuit was only concerned with the SDA/SCL pins for communication as well as voltage and ground connections.



**Figure 4: Accelerometer Reference Circuit**

*5.4.4 Communication / Software*

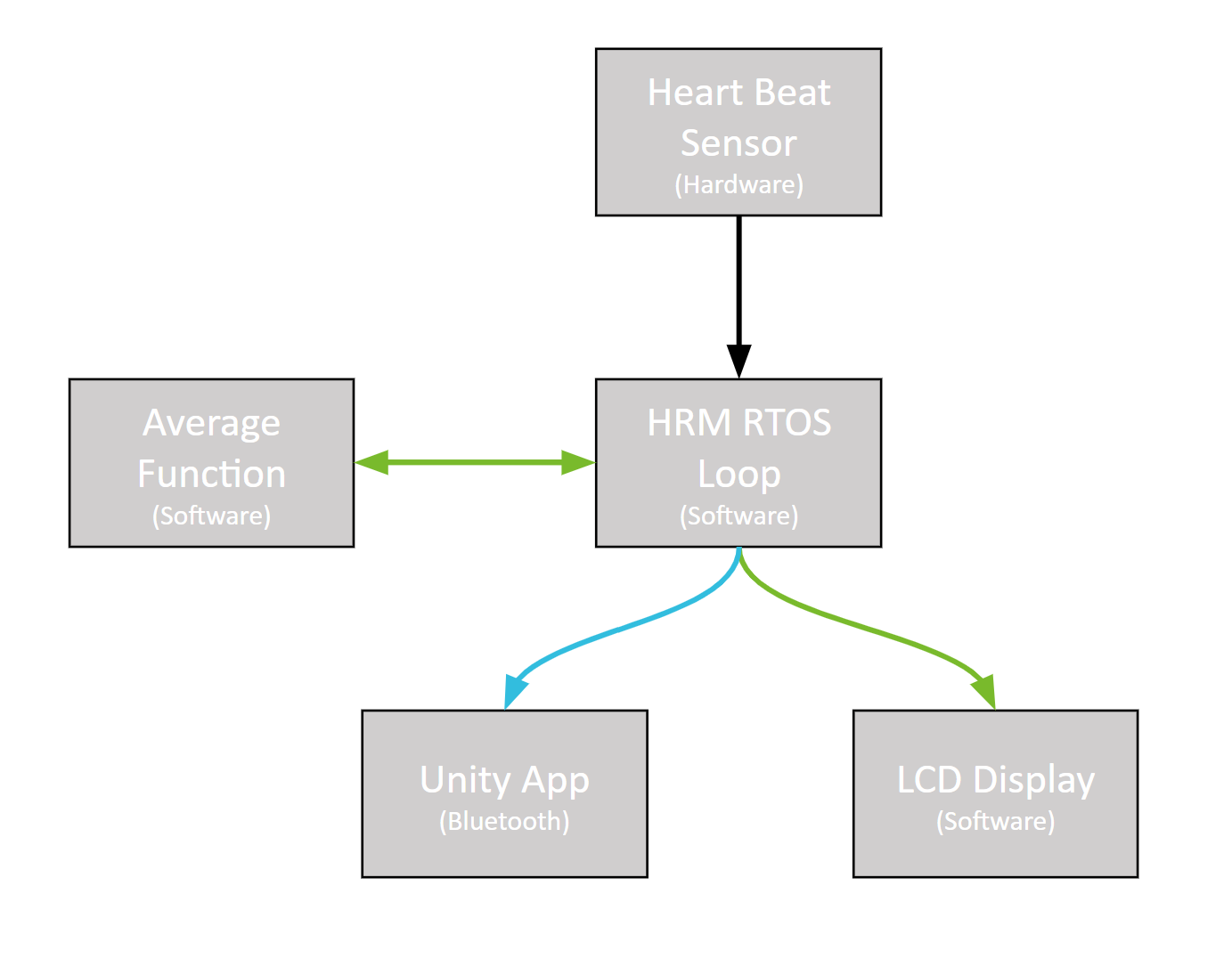
Communication of data between the accelerometer and the ESP is done entirely through I2C using Arduino wire commands. Data is pulled once every two seconds and then parsed and displayed to the user. Once data is pulled, it is mapped from its original min and max values to new min and max values of -90 to 90. From there, the inverse tangent of them is taken. For the desired forward angle of the bike based on how the part is mounted on the board, the equation is atan(-xAng / -zAng)+90.

## Detailed operation of Heart Rate Sensor

## 5.5.1 Overview

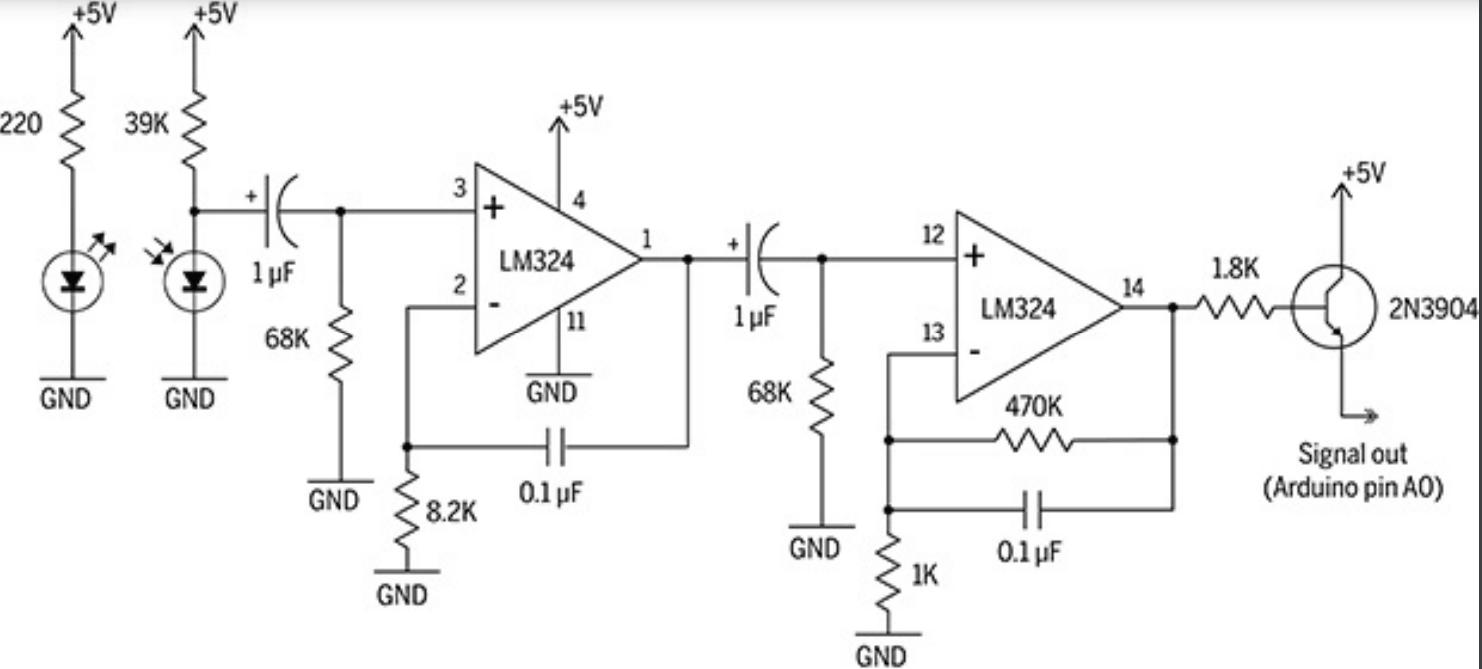
The heart rate monitor (HRM) works by reading in a light value using a coupled system of either an infrared (IR) sensor or a green light emitting diode (LED) and circuitry using an op-amp to amplify the signal created. Changes in reflected light in the blood in a person’s finger are able to be sensed through this circuitry and transferred through an analog signal back to the microcontroller of the system. This signal includes spikes in reflected light when blood is pumped quickly through the body with each heartbeat, and this spike, shown in the analog signal, can be used to measure the heart rate of a person using it.

In our final design for Demo Day, we used a prebuilt HRM, which connected to our board with three wires: power, ground, and an analog signal. The HRM was placed on the right handlebar of the bike and was connected to the board in the center of the bike. Through the ESP microcontroller, the data collected could be sampled and used to calculate heart rate in beats per minute (BPM). As shown in Figure 5 below, the HRM fed data, through the ESP32 microcontroller, into an RTOS software loop.

**Figure 5: Block Diagram of Software and Hardware Interfaces of HRM**

In this loop, the code triggered when the sampled signal was higher than a set threshold that indicated a heartbeat was being sensed, and then the time of this spike was subtracted from the time of the previous spike. This time difference was then calculated into a BPM, and this value was stored in a vector of heart rate measurements, with some logic to prevent values below 30 BPM and higher than 220 BPM from being stored (values outside this range were highly unlikely to be accurate). Every eight seconds, the data stored in this vector was averaged, using a separate average function in the code, and this data was stored in an average heart rate variable. This variable then was used to display both on the LCD, and was transferred using Bluetooth to the iPhone Unity App.

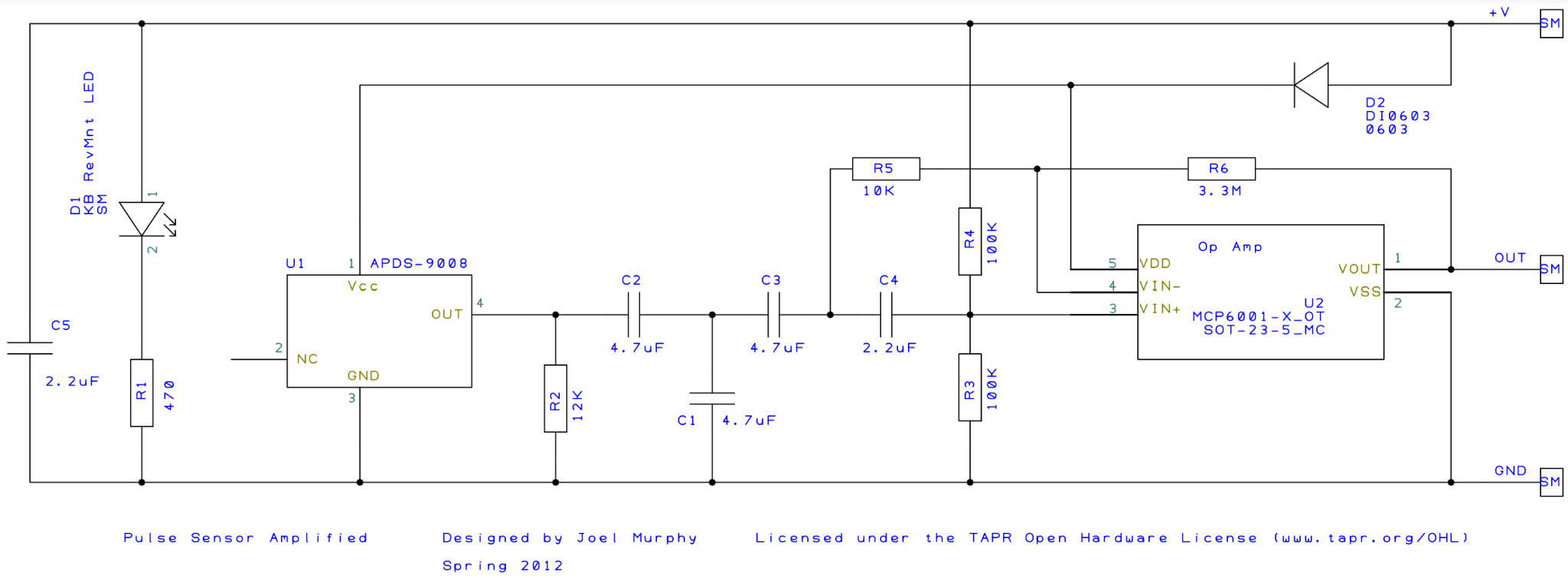
*5.5.2 Design Changes*

In our original design, we used the schematic shown below in Figure 5, which uses the IR sensor in the circuit rather than the green LED. 

**Figure 6: Heart Rate Monitor Schematic in Initial Design**

Our plan was to have two sensors, attached to each handlebar, that would connect to the same op-amp on our main board, and data from both sensors would be averaged together to create a more accurate heart rate reading. Our design had two peripheral boards that just contained the IR sensor, photodetector, and resistors, and three wires (power, ground, and signal) that connected to the main board in the center of the bike. This design made the boards on the handlebars less bulky and allowed us to use one op-amp. Unfortunately, we were not able to get this circuit design to give accurate and consistent heart rate data and did not have time to build a new board with different circuitry before Demo Day.

For this reason, our final design involved the use of one pre-built HRM located on the right handlebar, which uses a green LED coupled with a photodetector. The schematic for the circuit of this HRM is shown below in Figure 7.



**Figure 7: Pre-Built Heart Rate Monitor Circuit Used in Final Design**

*5.5.3 System Requirements and Future Changes*

The main system requirement for the HRM was requirement 2.4: The system shall measure heart rate of the user to within 10 bpm. We had some issues on Demo Day showing this requirement to be working consistently. Based on the way our software was set up, if the HRM was not getting a good signal, the heart rate would be shown as 214 BPM. Oftentimes when we were demoing our HRM, it would show heart rates within our requirements, and then spike up to 214 BPM, and then back down. We also had issues with the HRM giving accurate results depending on which person was using the device. The device would work well for some people, but not for others. On the software side, the issue of heart rate spikes could be mostly mitigated by more logic in the software, and more time testing the device on many people to make sure it was giving consistent results. Having a second sensor also would have been beneficial towards giving accurate and consistent results, especially since our main issue was getting a bad signal.

Another main issue with the HRM data that prevented getting consistent results was that during testing, the range of the signal data varied greatly depending who was testing the device and how they had their hand on the device. Since our code used relatively simple data measurements, just defining a threshold and calculating heart rate off of that threshold, it limited the accuracy of the device for different people using the device. This could be mitigated both on the software side and on the installation side. On the software side, more complex code could be used that would define the threshold based on how high the signal peaks got in the last period of time, rather than a defined value that never changes. This software would be difficult to implement but could result in a much more accurate heart rate. On the installation side, we could take measures to cover parts of the sensor that could be getting noise so that they do not affect the heart rate reading.

Overall, on Demo Day, we did see some periods of time with very accurate heart rate values. During these periods of time, the requirement for heart rate accuracy was met. However, these results were not consistent throughout the day. Given more time, we have pointed out specific ways we would mitigate the issues we saw to make the HRM consistently within our required specifications.

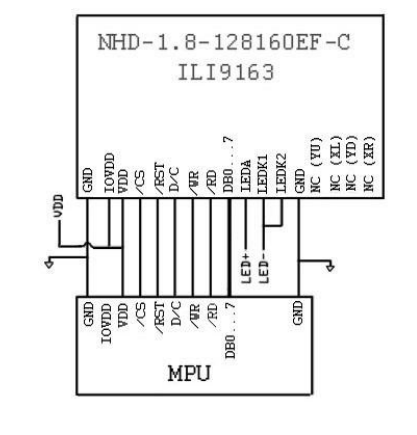
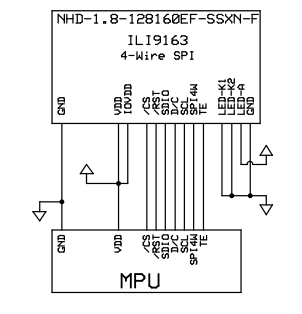
## Detailed operation of LCD

## 5.6.1 Overview

The liquid crystal display (LCD) screen is connected directly to the board through several pin connections and a semi-peripheral interface (SPI). We chose to use a device with SPI (NHD-1.8-128160EF-SSXN-F) because it required far fewer pin connections than one without it and was therefore easier to connect and troubleshoot. We also chose a 1.8-inch size because it was cheap, large enough to display important data clearly, and small enough not to interfere with the user’s experience. In terms of software, the display of data from the HRM, GPS, and accelerometer was very intuitive: the code (based in an Arduino framework just like the HRM, GPS, and accelerometer) simply referenced the variables that were being periodically updated in the RTOS and printed out the new values on loop. The code also set different customization parameters for the display of these values using the library that was available: one could set the text size, font, and color based on what was defined in the library using built-in functions. In our final product on demo day, we had the screen display four pieces of data: speed in MPH, heart rate in BPM, distance to the nearest hundredth mile, and finally, angle of incline in degrees.

*5.6.2 Design Changes*

As was mentioned before, we decided to switch from our original use of a 24-pin LCD (NHD-1.8-128160EF-CSXN-F) to an SPI-based one that only utilized 16 pins which allowed us to save space for other outputs from the board. The wiring diagrams for both of these are shown below.

**Figure 8. Wiring Diagrams for 8-Bit Parallel-based (left) and SPI-based LCD.**

The key difference between these two LCD’s in terms of pins is the replacement of the 8 data bus pins (DB0, DB1,..., DB7) with a singular interface pin (SPI4W) that performs the same functionality.

Additionally, the most complicated aspect of getting the LCD to work was not software-based, but instead getting the device to power on enough to run the program when connected to our board. Eventually we learned that because of a pair of resistors creating an unnecessary voltage divide, the LCD was not receiving sufficient power to run the program written for it for an extended period of time, so we decided to short those connections. Once we did so, the program was running smoothly, and the data was displayed clearly. While this is not a change directly to the LCD, it was a change required of the board to accommodate its full operation.

*5.6.3 System Requirements and Future Changes*

The LCD we chose, of course, satisfied our requirement of having an LCD to begin with, but there were a few shortcomings with the final product. We only ended up displaying four of the data points we initially set out to display on the screen because the screen ended up being too small for more data to be visible. However, all of the data was still available in the smartphone app. We also failed to meet our initial requirement of the LCD having a button-based interface for the user to be able to control it directly. We decided not to implement this aspect because it likely would have been a safety hazard for the rider if they had decided to adjust the display as they were riding. If we were to make a significant change to this subsystem, we would either choose a larger screen size so that more data could be displayed, or we would have the program toggle between screens that display different information periodically.

## Detailed operation of Bluetooth Unity App

*5.7.1 Overview*

The Bluetooth App serves as a data collection and display mechanism for the bike rider, and it currently uses Unity software running through Apple’s TestFlight on an iPhone. If this product was on the market, the app would be downloaded using the App Store or Google Play, and all necessary Bluetooth functionality would be loaded onto the board before installation. The app serves as a display for distance, speed, time, calories, mile split, and heart rate data, and it saves data from the last five rides a user has taken in order to track progress. Much of this Bluetooth

*5.7.2 Requirements*

The requirements for Bluetooth are related to the following:

3.1 The microprocessor shall regularly command measurement updates from hardware

3.2 The microprocessor shall save off sufficient data to calculate:

1. Total distance traveled
2. Average speed
3. Heart Rate
4. Time Elapsed
5. Calories Burned
6. Previous Mile Split

3.3 The microprocessor shall be able to connect via Bluetooth to a single cellular phone to push data to it (within a range of 10m)

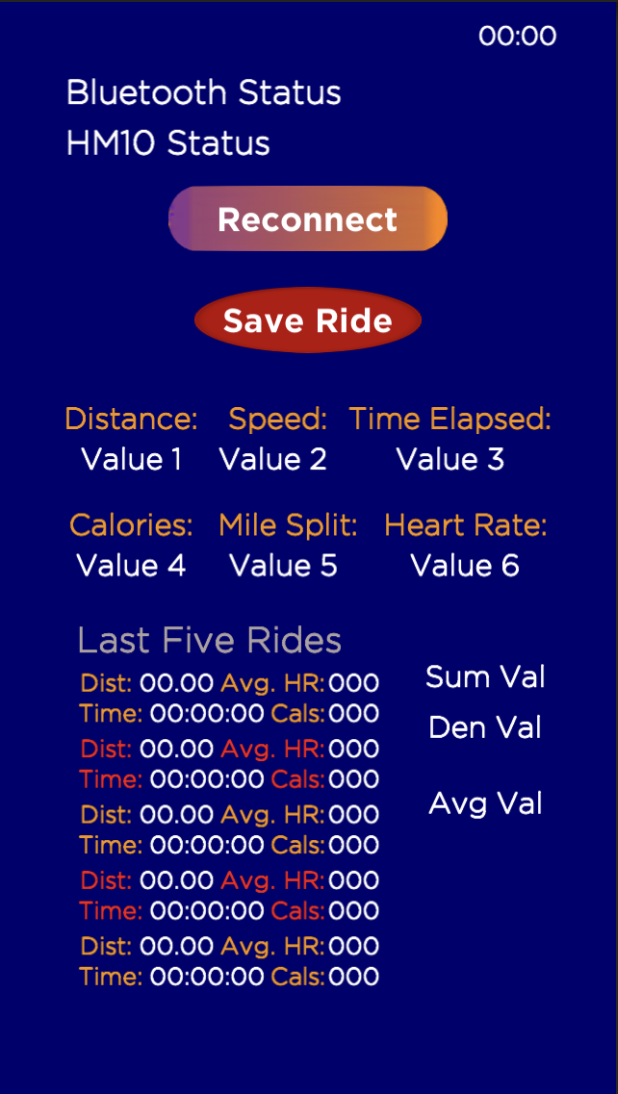
All three of these requirements were met except for the calculations of calories burned and previous mile split. While there was not an issue with sending the data back and forth from the device to the app, we ran into some issues with calculations on the device side, not the Bluetooth side, so the app was never able to display mile split and calories burned.

*5.7.3 Design Changes*

We originally were hoping to include customizable health informatics (weight, height, age, and sex) and goals in our app as well as a live Google Maps display; however, time restraints due to hardware and variable calculation troubleshooting led to a simpler app being developed. We also originally planned to store all data sent from rides and were hoping to have a separate, viewable menu of previous health data for users, but again, due to troubleshooting other components of the project, we were not able to implement this feature by Demo Day. Because of all of this, we opted for the simpler, proof-of-concept version of the app that displays the health data live and saves the previous five rides of a user so that they can track their progress, and we used static inputs for health variables such as weight, height, age, and sex.

*5.7.4 Unity (C#) Software*

The Unity app and C# code provide three main functionalities: displaying the status and facilitating the connection of Bluetooth, updating values in the app live once Bluetooth is connected, and storing trip data. The Unity editor view of the app can be seen in Figure 9.



**Figure 9. Unity Editor View of App (with Placeholder Text Object Values).**

First, the C# code reads for Bluetooth and displays to the app when the Bluetooth is connected. Additionally, the app has a “Reconnect” button that can be pressed to re-search for the Bluetooth device with a given initialized name. The C# code scans for any Bluetooth devices with the given name, and if it finds the device, it updates the state of the Bluetooth in the code in order to open up a communication channel.

Second, the C# code updates values to the app live once connected. By reading commands sent in the form of “Sr#:---” (or “Sr#:--:--:--” for time), the app is able to use a delimiter to collect the data and display it on the phone screen. “Sensors” 1-6 are used for displaying live data updates, 7-8 are used to display heart rate averaging debugging metrics, and 9 is used display the calculated average heart rate data from the current ride later stored in the app.

Third, the C# code provides a “Save Ride” button that, when pressed, stores total time biked, distance traveled, average heart rate for the ride, and calories burned during the ride. We had issues getting the calories burned to calculate and be displayed; however, we were able to get the other three variables saved in the app for the previous five rides saved. In the code, when the “Save Ride” button is pressed, the top-most saved values (lastDist1, lastHR1, lastCal1, lastTime1) are updated with the current values (sensor1value/distance, avgVal/heart rate, sensor4value/calories, sensor3value/total time), and each other set of saved value is replaced with the previous saved values (lastDist3 replaced with previous lastDist2, lastCal5 replaced with previous lastCal4, previous values from fifth entry deleted).

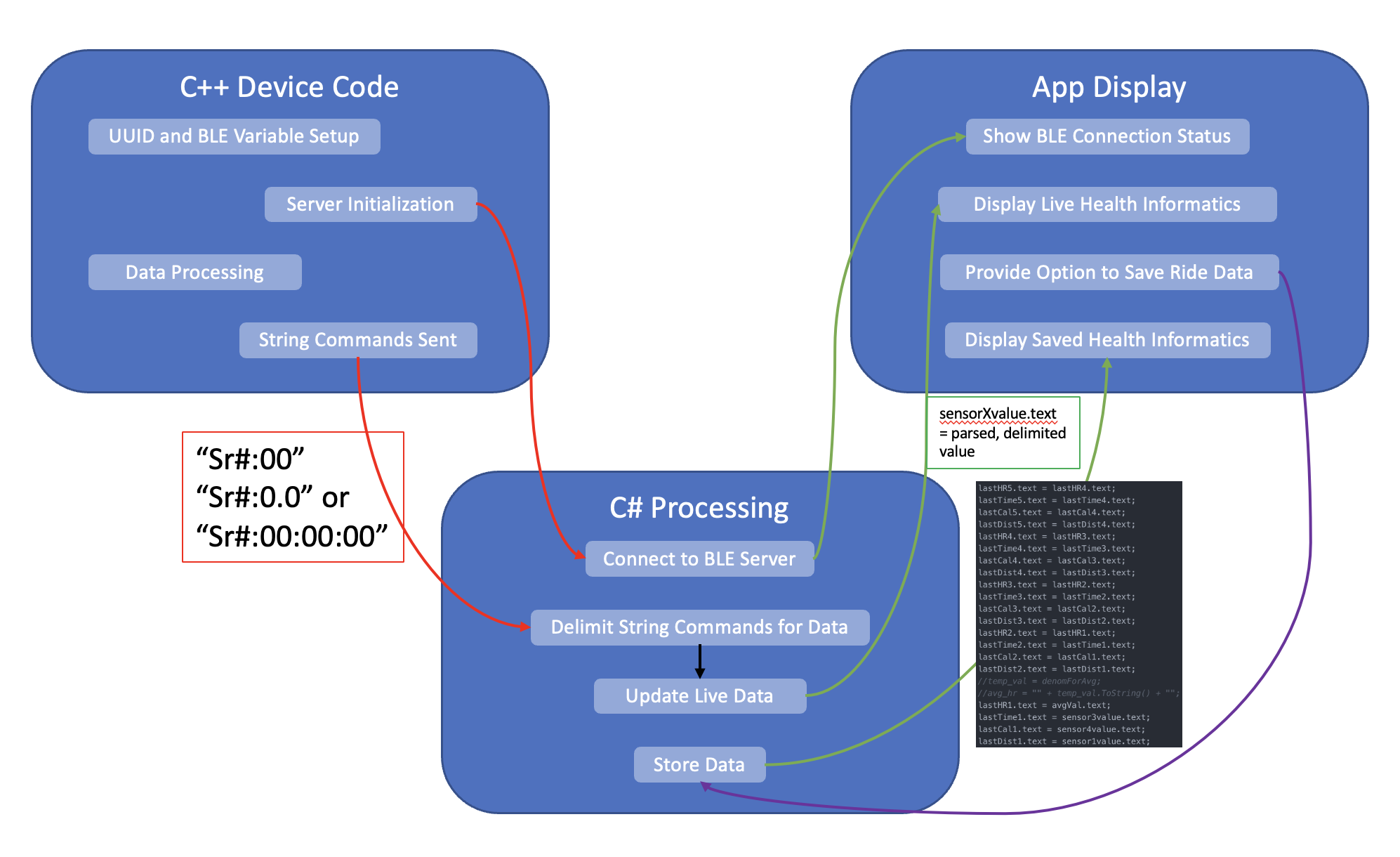
*5.7.5 System (C++) Software*

The system software communicates with the phone app by assigning a name (also called a UUID) to the microcontroller, in our case it is “6E400001-B5A3-F393-E0A9-E50E24DCCA9E,” and reading that name over Bluetooth in the C# code in the Unity app. Once that name is recognized, the phone and device can communicate with each other and update variables. These are initialized in the device code in the setup() function, where a BLE server is created, the UUID address is initialized, and the ability to write to Bluetooth is setup. This initialization can be seen in Figure 10.



**Figure 10. BLE Server Initialization Code Snippet.**

The system software sends nine different variables to the phone app in order to display live data and save trip data. These variables in the code are Total\_Distance\_Traveled, GPS.speed, Current\_elapsed\_time, calories, CurrentSplitTime\_minutes, avBPM, HR\_sum, HR\_denom, and HR\_stored\_avg. Each variable is sent in a string that has the format “Sr#:00”, and for time, the variable is sent specifically as “Sr#:00:00:00” in order to be parsed correctly. A flow chart for Bluetooth communication can be seen in Figure 11.



**Figure 11. Flow Chart for Bluetooth Application.**

Total\_Distance\_Traveled, GPS.speed, Current\_elapsed\_time, calories, CurrentSplitTime\_minutes, and avBPM are updated and displayed periodically as they would be in the on-the-market product, while HR\_sum, HR\_denom, and HR\_stored\_avg are updated and displayed periodically for troubleshooting purposes. HR\_sum, HR\_denom, and HR\_stored\_avg are variables stored and processed in the device code so that the average heart rate over a given ride can be calculated and stored long-term in the app. We attempted to do this averaging calculation within the C# Unity code, but it did not work, so the device C++ code was our next-best option. We displayed these variables in the app so that we could troubleshoot and were still troubleshooting up to Demo Day, so we never uploaded a new version of the app where these calculations were being done behind the scenes and without being seen in the app.

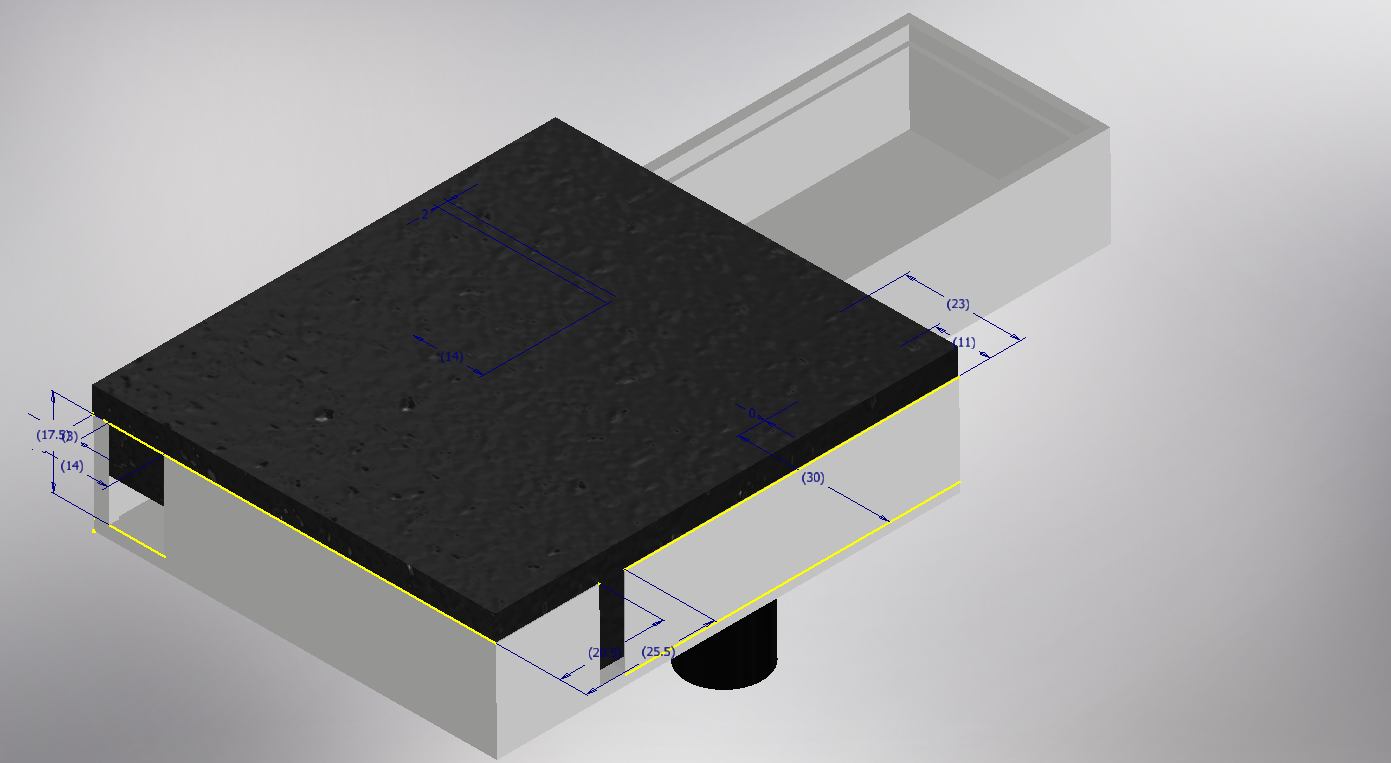
*5.7.6 Future Changes*

In the future, we would like to add calorie and previous mile split time calculations accurately and would like to have a broader capability to save data. Specifically on the app side, we could integrate multiple pages and menus to look at previous trip data as well as setting custom health stats (height, weight, sex, etc.) and goals (miles per week/month, weight goal, etc.) to better motivate and guide a user to make the most of their bike riding trips. We would also want to implement a login feature so that each individual would have privacy and access to their own health data, and we discussed possibly embedding a navigational feature with Google or Apple maps at a later point.

## Interfaces

*5.8.1 3D Printed Casing*

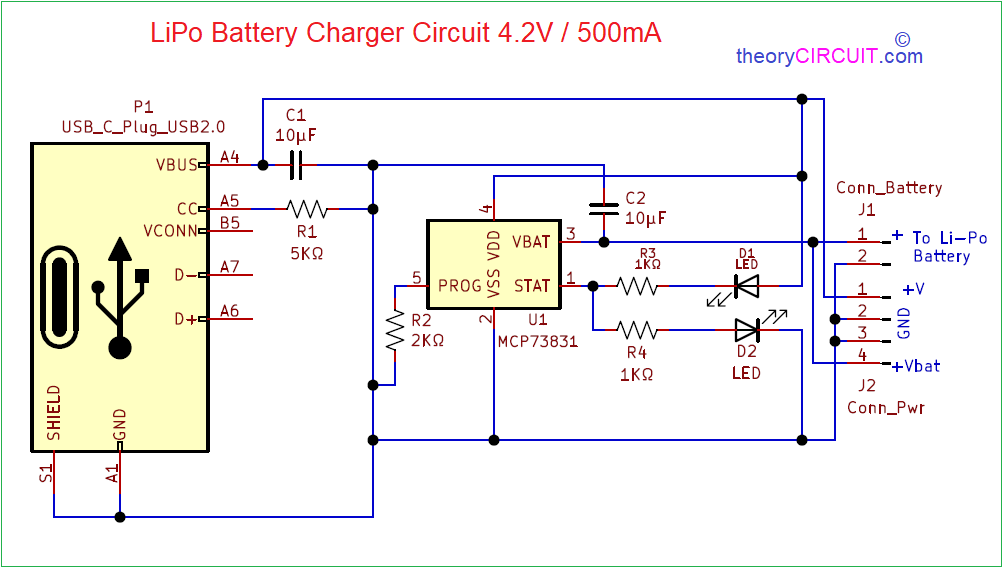
In our requirements, we stated that our design would be encapsulated in a 3D printed case. This case was designed in Autodesk Inventor in 3 separate pieces that can be seen assembled in the figure below. The pieces, while not explicitly weatherproofed in their current state, could easily be modified with some watertight adhesive for a finalized product. The product was made out of ABS plastic as not to attenuate the RF signals emitted by the GPS and Bluetooth components. There were no design changes as this part was designed very late in the development process, but some simple movement of different sections would be needed in an updated design to fully allow for charging the battery through the case and to let the LCD lay flat in its housing.



**Figure 12. Visual Assembly of the Custom 3D Printed Casing**

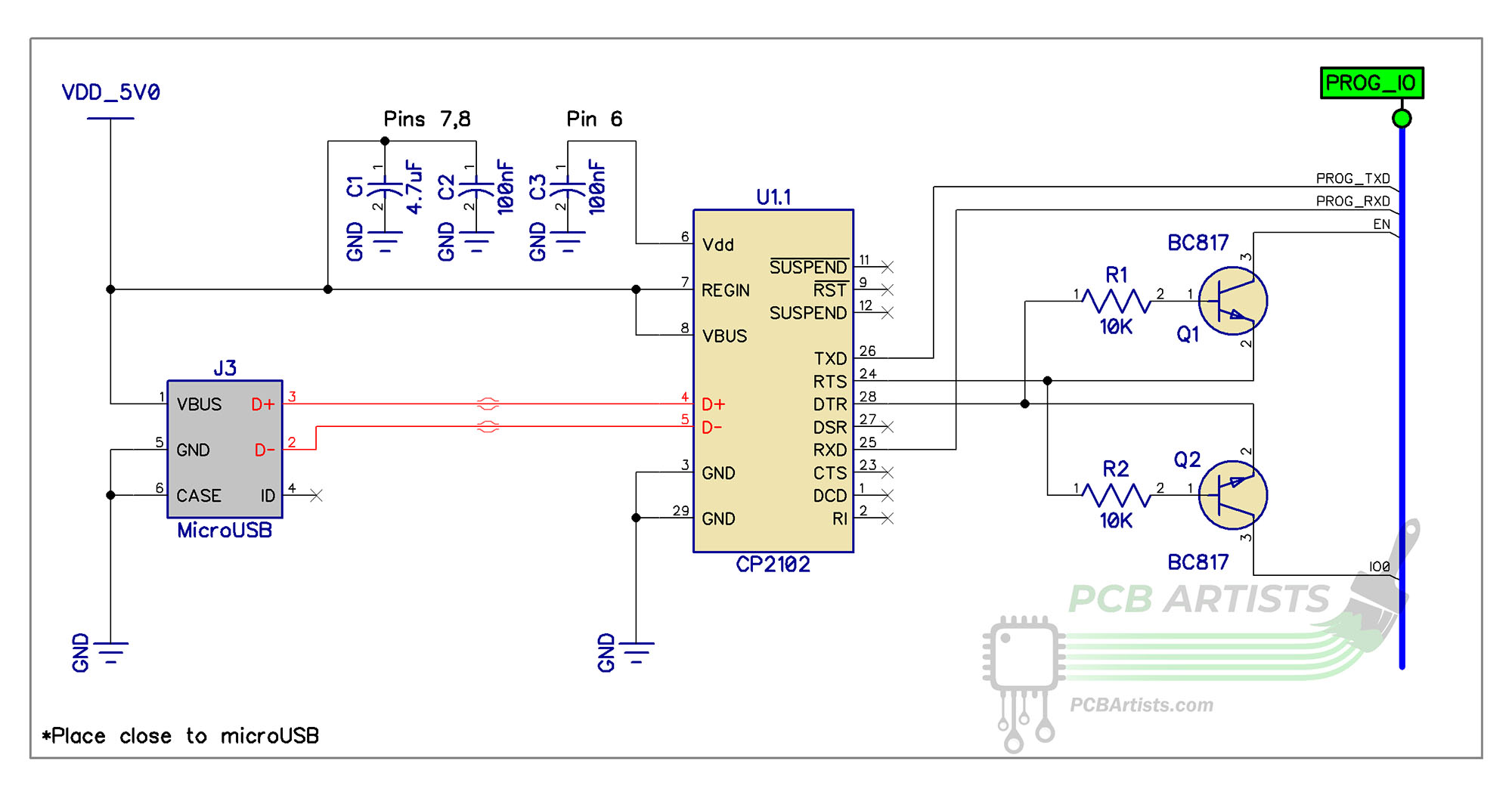
*5.8.2 Battery Charging Circuit*

One of our system requirements, requirement 5.4, was to have the unit charge via an onboard USB-C port. To implement this, we used a design from theorycircuit.com as there was no content in the curriculum to teach us this prior to the creation of our board. The only modification we made to the circuit was to remove the pin header connections seen on the right edge of the figure below that depicts the circuit we used. The circuit functioned well and was able to inform us through the LEDs whether power was coming from the USB-C or the connected LiPo battery. Supporting the brains of this operation that is the MCP73831, there are 2 DC voltage coupling capacitors, a resistor that alters the charging rate connected to pin 5 of the MCP chip, and 2 LEDs that indicate whether the battery is powering the circuit or being charged by the USB-C port. Since the programming happens with a physical component, no code was required from the ESP32. This however, meant that there was not a way that we were able to figure out how to measure and display the expected battery life remaining as we said in requirement 2.7.G. In the future, designing a way to implement this feature would be essential for a more robust product.



**Figure 13. Battery Charging Circuit Design Implemented**

*5.8.3 CP2102 Programming Circuit*

Much like the battery management circuit, we did not have much information to go on for designing a programming circuit. We wanted to be able to program our microcontroller while it was still inside of the casing so we opted to use a USB to serial converter chip in the CP2102 with the hope of programming through the USB-C port we had already added to charge our battery. While this was not an explicit requirement, we figured this would make things easier overall. This was the farthest thing from the truth as both attempts to build our board failed to let us program through the USB-C port. Whether that was from poor connections on the USB-C receptacle or the CP2102 or a bad circuit design overall. The circuit we decided to use is depicted below in Figure 11.

**Figure 14. Programming Circuit Dependent on the CP2102 Chip**

In future iterations of the design, the CP2102 was too much of a hassle since we never received any value from its addition to the design and the end user would never have to program their unit. An external programming system that connects directly to the Rx/Tx pins on the ESP32 through traces that go to pin headers should work as they take up less board space than the CP2102 and its supporting components.

# System Integration Testing

## Integrated Subsystems Testing / Design Requirements Verification

In general, many of the subsystems functioned independently as either a sensor or a display of some kind. Due to this, it was straightforward to do a lot of individual testing, and few larger integrated tests were done. These small tests typically consisted of print statements to the serial monitor and comparing those results against the expected values. Tests like this were done multiple times for both the GPS as well as the heart rate sensor. However, there were some tests that did still employ all of the different subsystems working together.

*Walking Test*

The first of the fully system tests conducted was once the 3D printed board was completed. At this point, a package for the board was complete, and the LCD was functional. This allowed the design to be safely and easily transported around campus while checking functionality of the heart rate sensor, GPS, and LCD to all be working with the ESP microcontroller.

This test helped in particular with checking the requirements for speed and distance. Both were displayed, and both made sense given the walking route taken. The speed oscillated a bit between 2mph and 3.5mph, so it wasn’t consistent, but our average walking speed was somewhere near 3 mph, so this was not terrible data. The heart rate sensor was able to track our heart rate increase from somewhere near 60 up to somewhere closer to 80 as we were walking. While the recorded number did give out sometimes and shoot up to a number in the 200’s or down into the 40’s, it overall performed pretty well. In addition, this test proved LCD capability to display heart rate and speed data.

Potentially the most important requirement tested was the ability of the system to be powered off of the battery. This proved to be a success.

*Biking Test*

The biking test was the most important test conducted as it showed the system working in its fully realized form. This test saw the system mounted onto a bike and then ridden around one of the quads to test all parts of active system function. The results were incredibly promising and helped test nearly all of the requirements.

This test in particular helped validate all requirements in section 1 of the detailed system requirements. The device was easy to mount, light, protected, and did not create any risk to the biker. This fulfills requirements 1.1 through 1.7.

From section 2, requirements that were confidently met were 2.3, 2.5, and 2.7 A, B, C, F. Requirements 2.1 and 2.4 were likely met, but another measuring device was not used to be able to say so definitely. Requirements 2.2 and 2.6 were not met as they were both obsolete by this point in the design process. Furthermore, 2.7 D and E were not included in the LCD as the tradeoff was made between more information and legibility. Both of these data points are on full display in the app to be viewed by the user after finishing biking. 2.7 G was not included as there was a lack of design foresight as to how to detect a low battery.

All requirements from section 3 were validated in this test through the Unity App. The requirements from section 4 were not able to be tested as the weather conditions were clear. However, datasheet specifications of different components indicate requirement 4.1 detailing temperature range should be met. From section 5, requirements met include 5.1, 5.3, and 5.4. There was never a test run to see the full lifetime of the batter to test requirement 5.2. However, the calculations conducted lead to the conclusion that this is the case.

# Users Manual/Installation Manual

## How to install your product

7.1.1 Out of the box, the system’s battery needs to be charged up to full from the on board rechargeable lithium polymer battery and its charging circuit.

7.1.2 Ideally the product would arrive with the 3D printed case fully assembled. The first step is to affix the mounting pegs to an aftermarket mounting system intended for headlights like we did, or we develop a similar mounting system with screws going through two semicircle pieces that surround the middle of the handlebars of the bike.

7.1.3 Once the physical systems are ready, the last step before the system is ready to track a ride is open up our app that we developed for iPhone and connect to the Bluetooth in our ESP32.

## How to setup your product

7.2.1 Download our app with TestFlight in the meantime but in the future this app would be available in the App Store once the product goes commercial.

7.2.2 After installation, powering the device on and take the bike with unit attached outside for GPS to connect to satellites. Once the GPS is locked, the speed data will change to a nonzero number letting the user know that data it being collected from the GPS.

## How the user can tell if the product is working

7.3.1 Once the unit is attached to the bike and powered on, taking the combination outside for a moment to see the GPS data start to update as an increase in distance traveled or see the speed number change from 0.0 to a higher value.

## How the user can troubleshoot the product

7.4.1 Charging the battery will alleviate the problems of the GPS not fixing to satellites and the powering off of other parts of the device.

7.4.2 For other issues that may arise, opening the lid of the case to ensure all components are still connected such as the LCD being still connected to the FPC connector.

# To-Market Design Changes

*8.1 Customer Discovery*

The first design change we would need to make would not be to the actual design itself, but to the process of design. In other words, if we were to make a commercially available product, we would need to focus first on the customer before focusing on technical details. While we already do have our prototype, we also must explore the broader pain points and problems we are hoping to solve.

We have our own personal experience to go off of with biking on campus and using stationary bikes to work out; however, we do not have a broader sense of what users for this product would want or need, especially when compared to current options. Considering all of this, we think that the first step in developing go-to-market design changes would be to conduct 30-40 user discovery interviews.

These interviews would challenge the pain points we believe our product aims to address by speaking to groups of individuals who are frequently active, are stationary bike users, and/or are standard bike users. We would likely target a younger demographic, as more of the younger workforce is commuting to work on bikes, and we would also target college students who are bike riders. After gathering our interviewees, we would ask a standard set of questions.

We would first ask about current user experience and satisfaction with that experience. We would determine first whether the interviewee was a stationary bike rider, a standard bike rider, or not a bike rider. Based on this determination, we would then learn more about their specific experience. If the interviewee is both a stationary and standard bike rider, we would ask them both sets of questions for those classifications.

If the interviewee is a stationary bike rider, we would ask them which brand of stationary bike they use, why they like working out on the stationary bike, how satisfied they are with the health statistics displayed on the stationary bike (scale of 1-10), what their overall satisfaction is using a stationary bike to exercise (scale of 1-10), and, if they are not also a standard bike rider, why they prefer the stationary bike to a standard bike for exercise.

If the interviewee is a standard bike rider, we would ask them in what context they use their bikes (commuting, hiking, leisure, etc.), why they like biking, what their overall satisfaction is with their current biking experience (scale of 1-10), and, if they are not also a stationary bike rider, why they prefer the standard bike to a stationary bike for exercise. Additionally, we would ask if they use any third-party health informatics solutions while they ride their bike, what brand or solution they use, what their overall satisfaction with that solution is (scale of 1-10), and why they use that solution/health informatics in general.

If the interviewee is not a bike rider, we would ask them why they do not ride a bike, what might motivate them to begin riding either a stationary bike or standard bike, and what they look for in their exercise equipment. We would ask them if they use health informatics solutions for any other types of exercise or active leisure activities they participate in, and if they did, we would ask similar questions about brand, satisfaction, and motivation for use that we would ask those in the bike rider classification.

After collecting data specific to biker rider classification, we would ask questions relating to the problems and pain points we believe need to be addressed. These would include questions about cost of exercise equipment and the desire for more visible health statistics to be displayed on moving bikes.

We would first ask how much the interviewee would be comfortable spending on exercise equipment and fitness monthly and annually. Additionally, if they are a bike rider, we would ask how much they spend monthly and annually on their bike(s) and what the specific breakdown of their costs is. After understanding their spending habits, we would then ask how they feel about the amount of money they are spending on fitness, specifically with their biking. We would ask them if they think they spend too little, too much, or an adequate amount on their health and on their biking, and we would ask them if they feel like proper training or exercising technology is too expensive (asking on a scale of 1-10 how much cost bothers them).

We would then follow-up by asking for how sufficient they feel the technology they use is towards understanding and tracking their progress and health. We would ask on a scale of 1-10 how well they feel like their health technology benefits their wellbeing and keeps them informed, and we would also ask how much it bothers them if they use technology that doesn’t fully meet their needs. Additionally, we would ask how convenient using their technology is and how simple they would want a solution (as opposed to having multiple solutions integrated together like an Apple Watch or Fitbit with a system on a bike).

Finally, after asking questions pertaining to our ideated problem, we would ask SmartCycle-specific questions to get a sense of what features we need to change or improve versus what we need to keep. We would demo our current solution to interviewees, and we would ask them both broad and specific questions related to the product.

Starting with the broad questions, we would ask them in general what they liked and what they disliked as well as what they would hope to see implemented in the future. We would also ask them to rate on a scale of 1-10 their satisfaction with our solution in terms of ease of use and data access.

We would follow up the broad questions with specific questions about the cost and data display of SmartCycle. We would first ask for the maximum and minimum price that the interviewee would pay for a product like ours (specific numbers). We would then follow up by asking if they thought enough data was displayed and if they felt like our solution was all-encompassing for health informatics (on a scale of 1-5, 5 being fully-encompassing - if they said anything not 5, ask them for what specific data they would want displayed or saved that we don’t currently provide).

While this might be a less technical addition to our design process, it is a necessary one in order to go to market. John’s experience with LifeDrive customer discovery would aid in the customer discovery for SmartCycle, and future design choices and customer needs would be clarified by these interviews. While we certainly have our own fixes and design changes that we want to implement separately from this customer discovery, we would not be able to go to market until we better understood the customer.

*8.2 Competitor Research*

We would also need to research our competitors in order to determine what would differentiate our product from others available on the market. This step would come after we analyzed our customer discovery data as well as our own internal design reviews. After we determined what changes we would want to make, we would need to check for the IP, advertised features, and competitive advantages of other products, and we would also need to compare costs between those competitors and our own device. Once we determined features and cost, we could then better understand feasible next steps to move forward.

*8.3 Design Changes*

While we still would need to do our customary discovery and competitor research, we have learned through this project that there definitely are some immediate changes that we think are necessary to make our product market ready.

*8.3.1 USB and Programmer Selection*

One significant challenge we had with our board design was that our USB-C jack and programming chip did not work to program our board. In order to upload any code, we had to solder wires directly to the ports on our ESP32-WROVER microcontroller and use a mini-programmer board in the lab. This issue developed largely because the data pin pads used for the connections on the USB-C were extremely small and under the jack, so we could not fix any bridging or other connectivity issues. While we were never able to completely confirm that the issue originated from the USB-C, we knew that our programming chip had soldered correctly on our second board build and was not the problem. The USB-C gave power and charged the battery, but it did not work for the data pins for programming.

This issue could be solved one of two ways. We could get a USB-C or USB connector with enough data pins to work with our programming chip that had larger pads, or we could get a microcontroller that has programming functionality embedded in it and use a bootloader with a simpler USB jack. For the LifeDrive boards, John has used the latter option, and it has worked well.

*8.3.2 Heart Rate Monitoring*

Another significant challenge we faced with our solution was obtaining consistent heart rate data. While our custom-made heart rate boards didn’t work in time for demo day, the prebuilt sensor we used also had a consistency issue. In order to mitigate this, we could implement one or more of three strategies. The first strategy we could enact would be to change the software and logic surrounding the threshold of our heart rate calculation code. We could have a moving threshold (or moving average peak) as opposed to a static threshold so that the sensor readings could adapt based on the user and how the user interacts or holds the sensor. This might give more accurate results and could better account for the inconsistency we had when different people tested the sensor on demo day.

The second strategy we could implement would be to add a second sensor for accuracy purposes. If we had two separate heart rate sensors, we could use them to compare to each other and parse out more accurate data. Setting one on each handlebar of the bike would ensure that readings could be obtained, and checking the readings against each other would theoretically allow for better accuracy.

The third and final strategy we could implement would be to integrate our board to an Apple Watch or Fitbit via Bluetooth and collect heart rate sensor data that way. Wearable devices have fairly advanced technology already embedded within them that would allow us to gather accurate heart rate readings and display those on our LCD and phone app, and this data would also help better inform other calculations such as calories burned.

*8.3.3 Accelerometer Implementation and Incline Display Changes*

We do not currently display anything with our accelerometer besides the angle at which it is tilted, and that does not provide much useful information to a biker. A next step for accelerometer integration would be to calculate percent incline, a useful metric for bikers, and display it on the LCD and phone app. Using incline grades helps to determine workout difficulty and can also help with more accurate calorie calculations.

*8.3.4 Board Layout Changes*

In our first board design, there is quite a bit of empty space. We could save money and time by redesigning the board to be smaller, and we could strategically place our capacitors and resistors in such a way that we would have less bottom trace connections and vias. This would improve efficiency, save money, and lower risk of traces shorting each other.

*8.3.5 LCD Additions*

If we continued to use the LCD in our design, we would like to display more data to the user. This additional data would be percent incline as discussed in 8.3.3 along with calories burned, which we do not currently display.

Another option we could implement would be to eliminate the LCD from our design entirely and solely rely on the phone display to collect and view data. We would mount the phone on the bike along with the device and would save money on the overall cost of the system.

*8.3.6 Bluetooth Additions*

The Bluetooth app would be the most variable depending on our customer discovery interviews, as we would want to get user input before changing anything with the GUI. However, we would like to add more data visualization, unlimited saved rides, and a navigation feature to provide a more complete software solution.

# Conclusions

Our project intends to create a user friendly, simple way for users to track workout data outside while riding a bike. We implemented our design using three sensors: a HRM, a GPS, and an accelerometer. Using the ESP32 Wrover microcontroller, we connect these sensors to an LCD screen and an iPhone Unity app that both show pertinent workout data. The LCD shows current speed, heart rate, distance traveled, and incline data. The app shows all of these data points, as well as time elapsed, calories burned, and mile split. The app also shows average data for the past five workouts that have been saved using a button on the app.

The implementation of our project has been relatively successful. We were able to meet all of our system requirements, though the system requirement 2.4, about accuracy of heart rate, was not always consistent. We described in section 5.3 future changes we would like to make to hit this requirement consistently. We also did not end up meeting requirements 2.2 and 2.4, because design changes we made due to time constraints and ease of rider use made these requirements no longer relevant.

Main alterations we would make to our product if we were to bring it to market would include a lengthy customer discovery process and competitor research to match our product to specific needs in the market. This would allow our product to be as successful as possible, and for it to meet a real market need. We would also spend time creating design changes to the HRM by using a different circuit on the board and adding a second sensor, changing accelerometer data to be in percent grade, making changes to the board layout to make the system more compact, adding more information to the LCD screen, and expanding the iPhone app to have a better user interface and increased capabilities.

# Appendices

Original Design Source Links:

[Eagle Schematic and Board Files](https://seniordesign.ee.nd.edu/2023/DesignTeams/gearshifter/SmartCycle%20Eagle%20Files.zip)

[Software for the ESP32](https://seniordesign.ee.nd.edu/2023/DesignTeams/gearshifter/SmartCycleFinalCode.zip)

[Software for the Unity App](https://seniordesign.ee.nd.edu/2023/DesignTeams/gearshifter/SmartCycle%20Final%20Unity.zip)

Datasheet Links:

[ESP32 Wrover Datasheet](https://seniordesign.ee.nd.edu/2023/DesignTeams/gearshifter/esp32-wrover-e_esp32-wrover-ie_datasheet_en.pdf)

[LCD Datasheet](https://seniordesign.ee.nd.edu/2023/DesignTeams/gearshifter/NHD-1.8-128160EF-SSXN-F.pdf)

[GPS Datasheet](https://seniordesign.ee.nd.edu/2023/DesignTeams/gearshifter/GlobalTop-FGPMMOPA6H-Datasheet-V0A.pdf)

[Accelerometer Datasheet](https://seniordesign.ee.nd.edu/2023/DesignTeams/gearshifter/MPU-6050-Datasheet1.pdf)