Human Powered Vehicle Bike Computer

June 2013

Bradley Shellnut
Eric Yaklin

Table of Contents

Introduction:

The purpose of this project is to create a custom bike computer for the Cal Poly Human Powered Vehicle team, which designs and implements Human Powered Vehicles (HPV). HPV's are "aerodynamic, highly engineered vehicles that may be for use on land, in the water or the air."[22] The main focus of the club is to create HPV's for the Human Powered Vehicle Challenge (HPVC) events put on by the ASME (founded as the American Society of Mechanical Engineers) every year. This project provides the HPV team with a computing device that is useful for their competitions. The basic functionalities of the device include collecting data during a race, showing this data to the user during a race, and enabling the user to analyze data after a race.

There are two main aspects of this project: the data collection and the data presentation. The data collection is done by using an Arduino Uno microcontroller and attached sensors and modules. A GPS module collects location data, a temperature sensor collects temperature data, and reed switches collect wheel and gear rotational data. This data is sent to an Android smartphone or tablet via Bluetooth communication. The information gathered from the Arduino is then displayed on the screen of the Android device (data presentation), including speed, temperature, distance traveled, lap time, and cadence. The Android device contains a custom-designed user-friendly application. The purpose of this application is to store and display data for different users by using a user-profile design. Different rider's profile information will be stored and accessed via external storage on the Android device (usually an SD card). Profiles will contain a name, a temperature in degrees Fahrenheit (for use with an external fan attached to the Arduino), and a picture.

There are two main modes of the application after profile creation/selection: race mode and street mode. In Race Mode, the user is prompted to enter wheel size and lap distance in order to calculate speed and lap count. Once the user presses the start button, the Android device begins communicating with the Arduino Uno. While the Arduino Uno is sending data to the Android device during a race, this data is being stored in a CSV file on external storage under the current rider's profile.

Also, if the recorded temperature exceeds the rider's specified temperature in their profile, the Android device sends the Arduino Uno a command to turn on an external fan for temperature control inside the fairing. After a race, a rider can attach the Android device to a laptop or PC and open the CSV files in a spreadsheet document program such as Microsoft Excel. This enables the rider to access data after a race so they can view trends and perform calculations. In Street Mode, the rider's position will be shown on a map using Google Maps. Street mode will also show the current and max speed of the rider. This mode does not communicate with the Arduino Uno since the information presented doesn't need to be as accurate as race mode. This mode also does not save data in a CSV file for later inspection and uses carrier data in order to access the Google Maps database.

In summary, this project provides the Cal Poly HPV team with a computational device that will benefit the team and possibly pave a way to more custom computer solutions to mechanical problems.

System Description & Implementation:

In this next section we will discuss our software implementation. This section will be broken up into the following parts: “High Level UI”, “Selecting Program Functionality”, and “Flow of Software.” “Flow of Software” will be split into multiple sections, each section explaining a separate activity in the app and the programming techniques used to create that screens functionality. There is also an additional section describing the Arduino software implementation.

High Level UI

After consulting our client we decided that our Android App’s UI had to allow the user to create a new profile and also select from a list of already created profiles. Allowing the user to select one of these options warranted creating a main screen to allow them to do just that. Once a profile has been created/selected the next screen needed to show their profile with options to change their current settings and move on to the main part of the App. Because our client needed to use the device in an environment that limited phone data capabilities we needed not only a data driven Google Maps interface but also an offline version of maps. Using this knowledge we decided that the last two screens our app would allow the user to go to would be accessed from the profile screen and allow the user to select a “Race” mode (no data, offline maps) or a “Street” mode (data, Google Maps). The high-level UI flow diagram can be seen below in Figure 1.

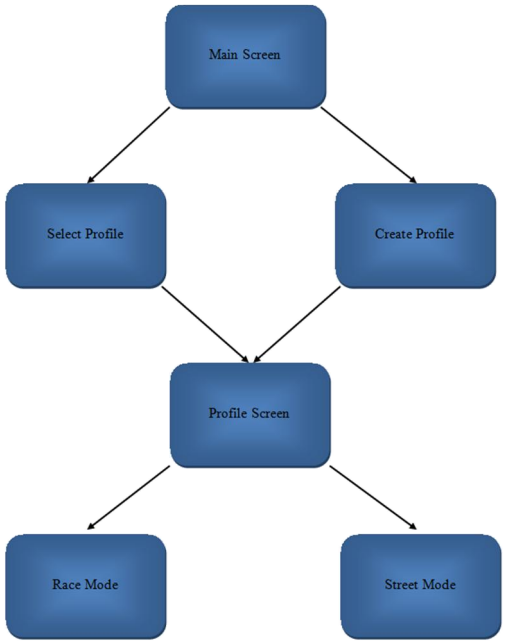

Figure 1: High level flow diagram of the App UI

Activity Overviews:

Once a high level flow diagram was created the next step was to figure out what functions needed to be performed in each screen depicted in the diagram. We concluded that the "Main Screen" needed to allow users to select either "Create" or "Select" a profile. The user also needed to be able to access a help screen which would display basic information on how to use the App.

In the "Select Profile" screen the user is able to select from a list of profiles. Each entry in the list includes the name of the profile and a profile image. The user also has the capability of deleting entries in the profile list.

While in the "Create Profile" screen, the user needs to enter a unique name for their profile and a temperature for the fan feature of our device. The user also has the option to take a picture that will be displayed with their profile or choose a picture that already exists in their phone's photo gallery. The user also has the choice not to take a picture and stick with the default profile picture. Once a profile has been selected/created the "Profile Screen" displays the users profile name and picture. This screen allows the user to change the settings they chose when they created their profile. From the profile screen the user has the choice of selecting race mode or street mode.

"Race Mode" displays the offline map, current temperature, number of laps, speed, start and stop buttons with a timer, and draws the current location and past locations on the map. This mode connects to the Arduino via Bluetooth and writes the collected data to a CSV file so the riders could export the data to Excel.

Lastly, the "Street Mode" displays Google Maps at the location of the user while also displaying the current speed, max speed, and draws the route the user has taken when in "Street Mode".

Flow of Software:

The following section describes in detail the features of the previous section.

The programming logic flow when moving from the main screen to either “Select Profile” or “Create a Profile” can be seen below in Figure 2.

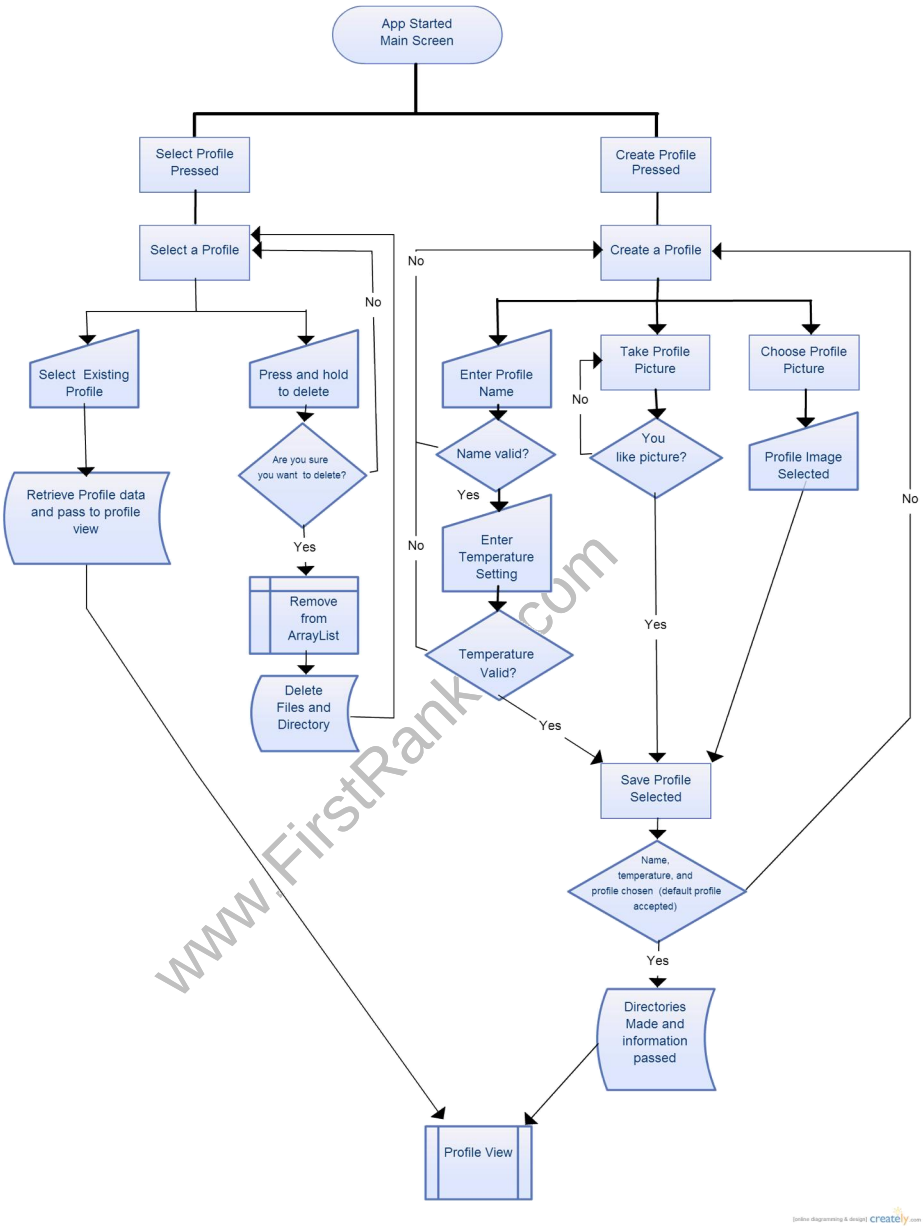

Figure 2: Programming Logic of Main Screen, Select Profile Screen, and Create a Profile Screen

Main Screen:

When the user launches the Android Application they will be presented with the main screen of the app. While at the main screen the user has the option to select the help button in the top right hand corner, select the "Create Profile" button, or select the "Select Profile" button. An image of the main screen can be seen below in Figure 3.

Figure 3: The Android Application Main Screen

Create Profile:

When the "Create Profile" button is pressed the Android activity is created and we begin the create profile branch of Figure 2. The user is then presented with a screen containing two text entry locations, a default image view, and three buttons (archive, camera, and save). A screenshot of this screen is shown in Figure 4 on the following page.

When the user clicks a text entry field the default Android keyboard pops up and allows the user to enter information. The user also has the option of taking a picture for their profile. Pressing the button with an image of a camera will launch the default Android camera App, allowing the user to take

a picture. Pressing the button with an image of a folder allows the user to select an image from their phone gallery as their profile picture. Once the information in the text fields is entered and the user

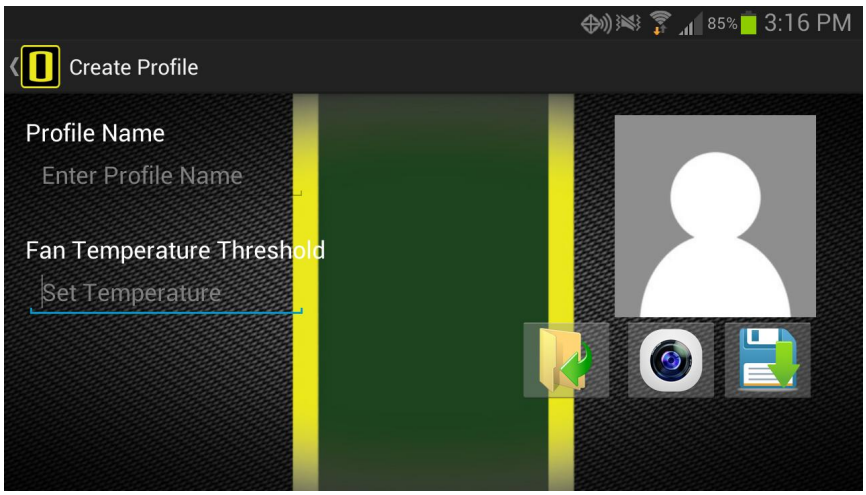

Figure 4: Create Profile Screen in the Android Application

has chosen a picture the user shall press the “save” button. When the save button is pressed, the code searches through our apps save directory. This directory is located in your phone’s internal memory, the /HPV (Human Powered Vehicle) directory. If the search through the directory finds that the name the user entered for their profile already exists the app writes a message to the screen prompting the user to change the name. If the profile name is available then the app checks to make sure the temperature the user entered is between 0 and 125 degrees Fahrenheit. If this is not the case, the app prompts the user to change the temperature. Lastly, the code checks to see if an image was taken/chosen. If everything passes these checks the code then creates a new directory within the /HPV directory. This directory’s name becomes whatever the user entered for their profile name. An example of a user’s directory structure would be sdcard0/HPV/Bob, with Bob being the users profile name. The app code then writes the profile image to the user’s directory and creates a “.csv” file containing the profile name and temperature setting.

Once all of this has been done, the code passes the user name and temperature to the “Profile View” (as shown in Figure 2) through the programming technique called Android Shared Preferences.

Select Profile:

When the “Select Profile” button is pressed the Android activity is created and we begin the “Select Profile” branch in Figure 2. The user is then presented with a screen containing a list of all the profiles in the app. The program loads the information needed to populate the list from the user’s phone memory in the app’s directory (“/HPV”). Using basic Java file I/O the program loops through the directories and stores the name and corresponding bitmap image in an ArrayList of custom profile objects.

Once the ArrayList is populated the program creates a separate entry in the list for each profile. Each entry contains the user’s profile image to the far left of the screen and the users profile name in the center of the screen. In order to accomplish these tasks the program uses Android base adapters and custom XML files. An example image of the “Select Profile” screen can be seen below in Figure 5.

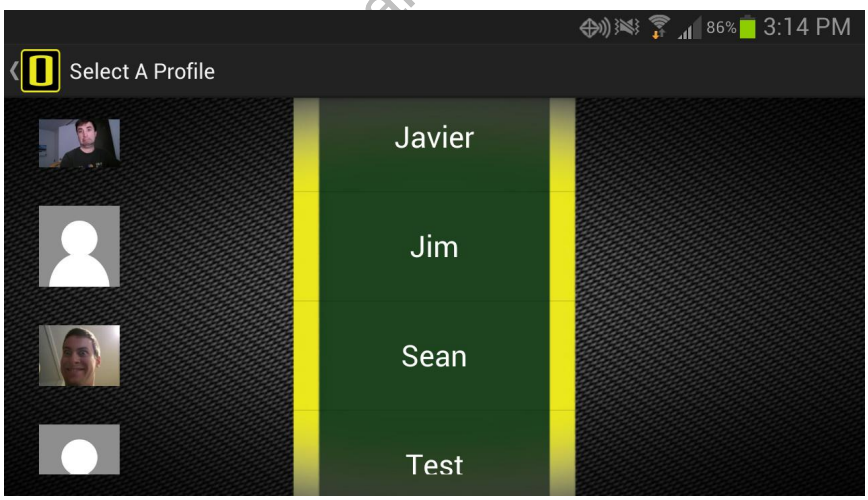

Figure 5: Select Profile Screen in the Android Application

From this screen the user has two options: Press and hold a name to delete the profile or tap a profile to load it. When the user decides to delete a profile they are prompted with a confirmation message making sure they want to delete their profile. If they choose yes then the program removes the entry from the list and deletes the user's data on the phone. An example image of the select and hold to delete can be seen below in Figure 6. When the user decides to choose a profile to load they tap the user and the program will pass the user's data to the "Profile View".

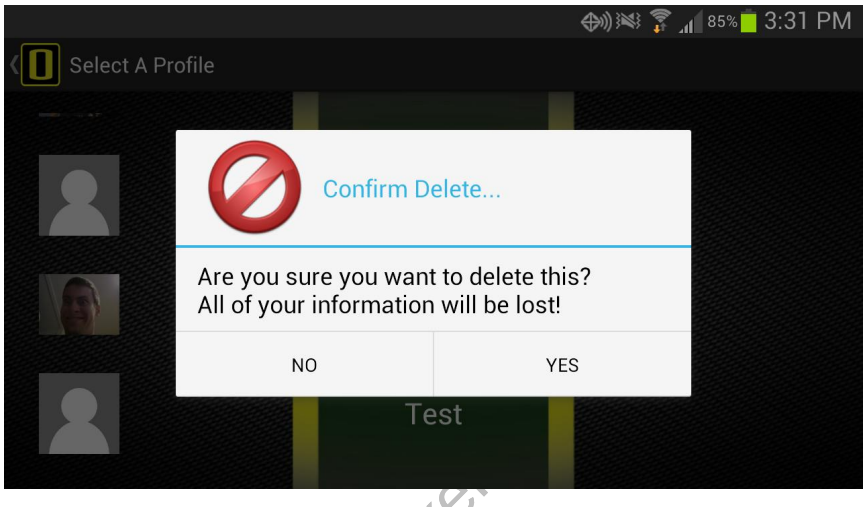

Figure 6: Deleting a Profile

Profile View:

The program flow diagram for the profile view can be seen in the next two pages in Figure 9. While the user is at the profile view screen they are presented with their profile image to the left, their name at the top of the screen, and four buttons. An image of this screen can be seen in Figure 7 on the next page. The user has the option of changing their profile picture by pressing the button "Change Picture". If the user wishes to change their profile name and/or temperature threshold they can press the settings button in the top right hand corner of the screen. An image of this option can be seen in Figure 8 on the next page. Lastly, the user can choose to enter the "Street" or "Race" mode. If the user chooses the race mode they will be prompted with a prompt similar to the change settings prompt asking for their bike's wheel diameter in inches and the distance of the race track in miles. Pressing

either of these two buttons will take the user to their respective Android Application activities as shown in Figure 9.

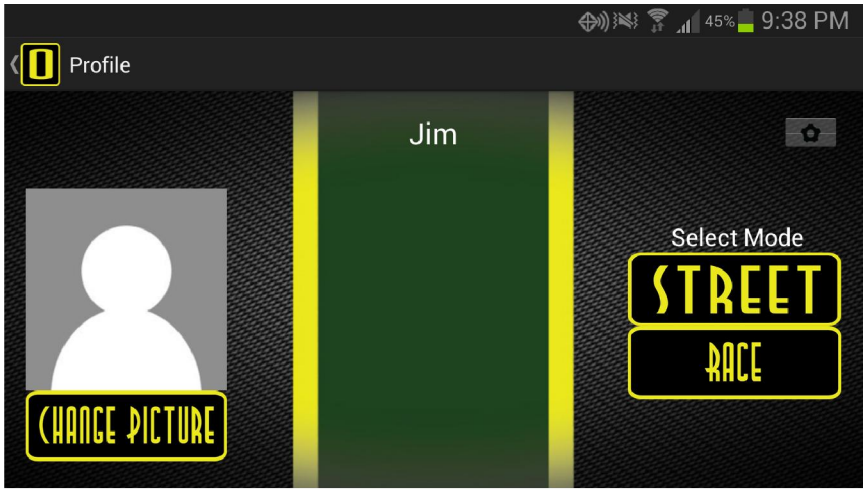

Figure 7: Profile Screen in the Android Application

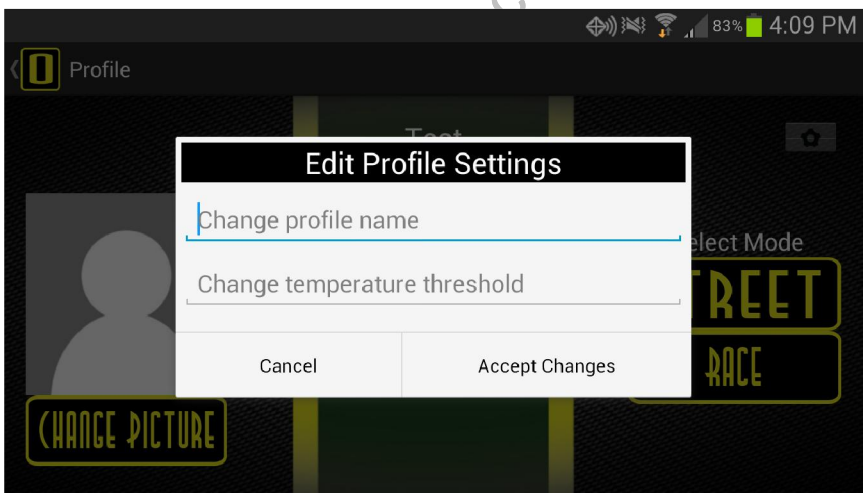

Figure 8: Edit Profile Settings

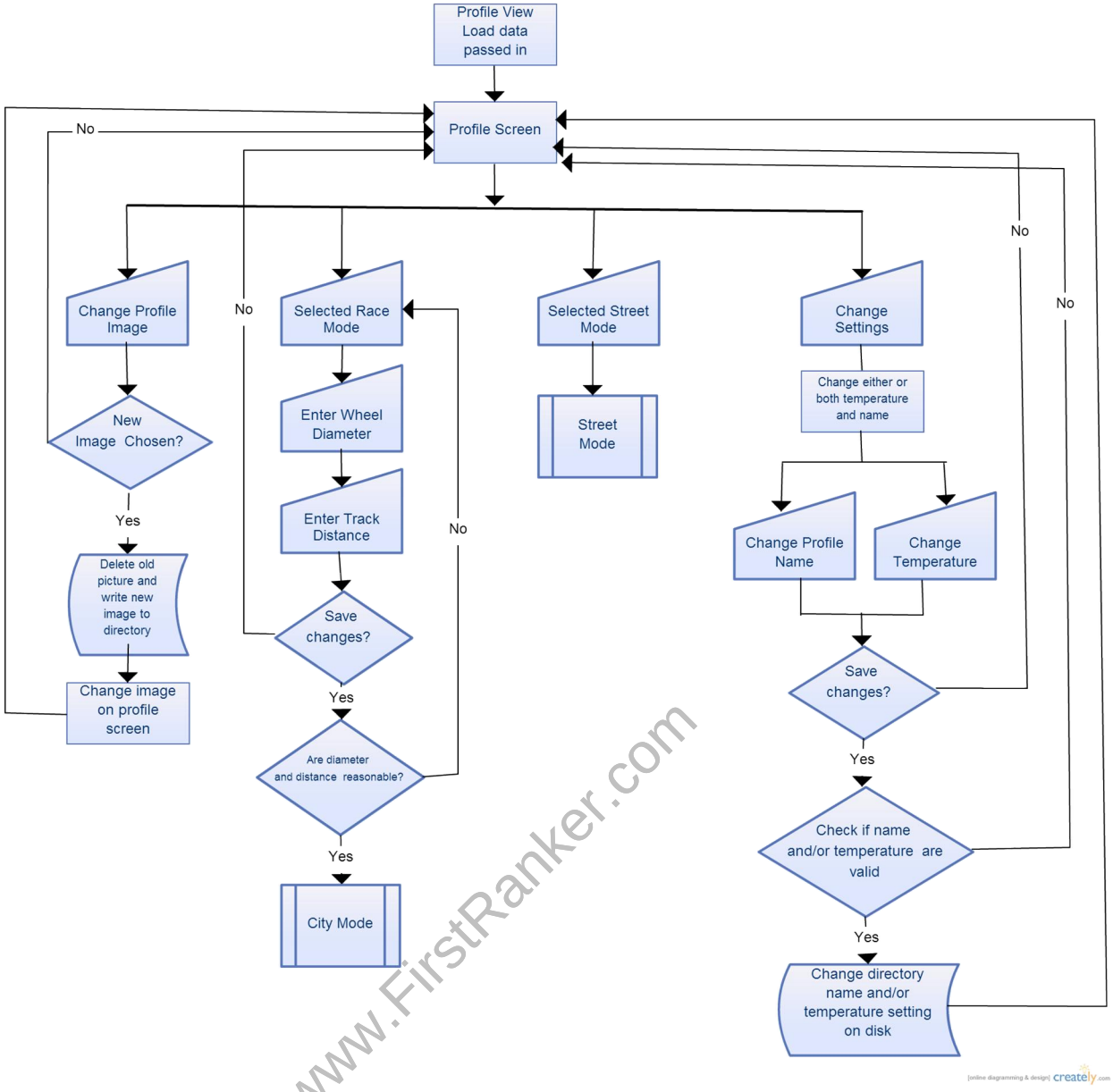

Figure 9: Profile View Program Flow Diagram

Street Mode:

The software flow diagram for the "Street Mode" can be seen on the next page in Figure 11. Street Mode uses the built in Android Google Maps API to display where the user currently is on a map. When the user enters this mode the program checks to make sure the GPS is turned on and prompts the user if it is not turned on. If GPS is turned on, the phone finds their location and centers the map on the location. The street mode screen displays the user's max speed they have attained and their current speed. It also checks with the GPS every 1.5 seconds to see if the user's position has changed. If the current position has changed then Google maps draws a line between those coordinates and the last recorded coordinates. An image of this screen can be seen below in Figure 10.

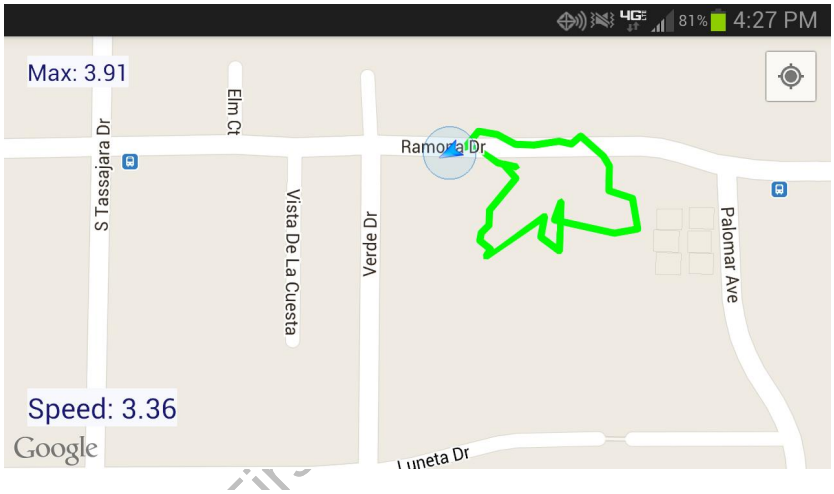

Figure 10: Street Mode with Max Speed and Current Speed shown

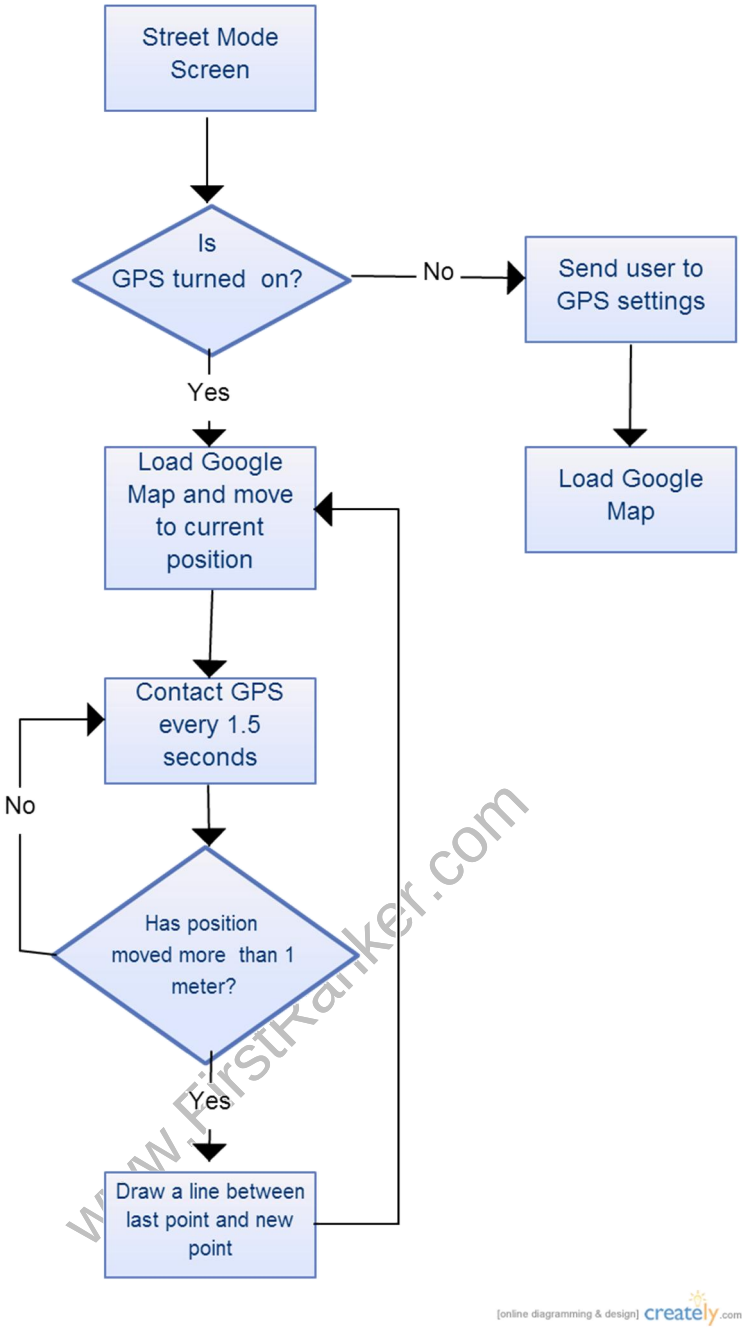

Figure 11: Street Mode Program Flow Diagram

Race Activity:

The Race Activity is the main focus of our project: bringing together the Arduino and Android to create a data collecting system. The process for starting the Race Activity begins when the user presses the “Race” button on the Profile Activity screen. This triggers a prompt to the user asking for a wheel size in inches and a lap distance in miles. If the user presses cancel, the Race Activity will not load and the user presses accept, the Race Activity will load.

When the Race Activity first loads it initializes variables, text labels, and the MapView object and also gets the current date and time in order to name the CSV file. It reads the race data and directory data from the settings that were passed in to the Race Activity from the Profile Activity. A new CSV file is created in the directory specified by the passed-in settings, with the title ”RACE<date>@<time>”. The file’s name contains the date and time so it is easy for the rider to know what race the file is associated with.

After initialization of data, the Race Activity attempts to connect to the Bluetooth module and begins listening for data. It first starts by creating a “BluetoothAdapter” object and checking to see if Bluetooth is enabled on the phone. If it is not, the Race Activity automatically enables it without the user’s consent and pairs with the “linvor” device. This is the default name of the Bluetooth module we are using. After enabling Bluetooth on the phone and choosing “linvor” as the Bluetooth device, the Race Activity attempts to connect to the device. The Race Activity makes a socket to connect to the default Bluetooth UUID, “00001101-0000-1000-8000-00805f9b34fb,” and attempts to connect to it. If this connection fails, an exception is thrown and the Race Activity attempts to connect three more times. If the Race Activity attempts to connect to the Bluetooth module a total of four times, the Race Activity quits, goes back to the Profile Activity, and displays a message to the user suggesting to pair to the correct device or turn the Arduino device on. If, however, a successful connection is made, input and output streams are created so data can be transmitted and received to and from the Bluetooth module.

After a successful connection the Race Activity starts a new thread that is used for listening to Bluetooth data.

At this point, nothing else happens with the Activity until the user presses the “Start” button on the screen. The “Start” button performs three main events: It starts the on-screen timer, tells the Arduino to start sending data, and finally tells the newly created thread that the Race Activity is ready to listen to data. If the stop button is pressed the on-screen timer stops, the Arduino is ordered to stop sending data and the input stream and file are closed.

While listening for data, the Race Activity uses the Java I/O “readFully()” method. This method blocks until a specified number of bytes are read into a buffer, 20 in our case for five four-byte floats. We decided to use this function because it is much easier than polling for the data. The data bytes are ordered as follows: 0-3 represent the time of a wheel rotation in milliseconds as a float, 4-7 represent the GPS longitude as a float, 8-11 represent the GPS latitude as a float, 12-15 represent the temperature in degrees Fahrenheit as a float, and lastly bytes 16-19 represent the time of a gear rotation as a float. After this buffer is filled, the data parsing begins. When parsing the data, the byte order was important to consider. They were copied in in Little Endian order, meaning that the least significant byte was read first. If they were read in using Big Endian, the data would be wrong.

After the bytes are read in and correctly placed in their corresponding variables, calculations for wheel speed, distance traveled, and number of laps completed are performed. In order to calculate the distance traveled, the Haversine formula was used. The Haversine formula is one of the most important navigational formulas, giving great-circle distances between two points on a sphere from their longitudes and latitudes [21]. Each time the distance is calculated it is added to the overall distance which is displayed on the screen. When reading the GPS data the float values were changed to doubles in order to keep the high accuracy precision that is needed when handling GPS coordinates. The total distance is also used to calculate the number of laps that have been completed. The time the wheel to

rotate and the wheel circumference are used to calculate wheel speed in miles per hour and the time to rotate the gear is used to calculate the cadence in rotations per second. After all necessary information has been collected and computed it is displayed on the screen and saved to the CSV file. The software flow chart for the Race Activity can be seen below in Figure 12.

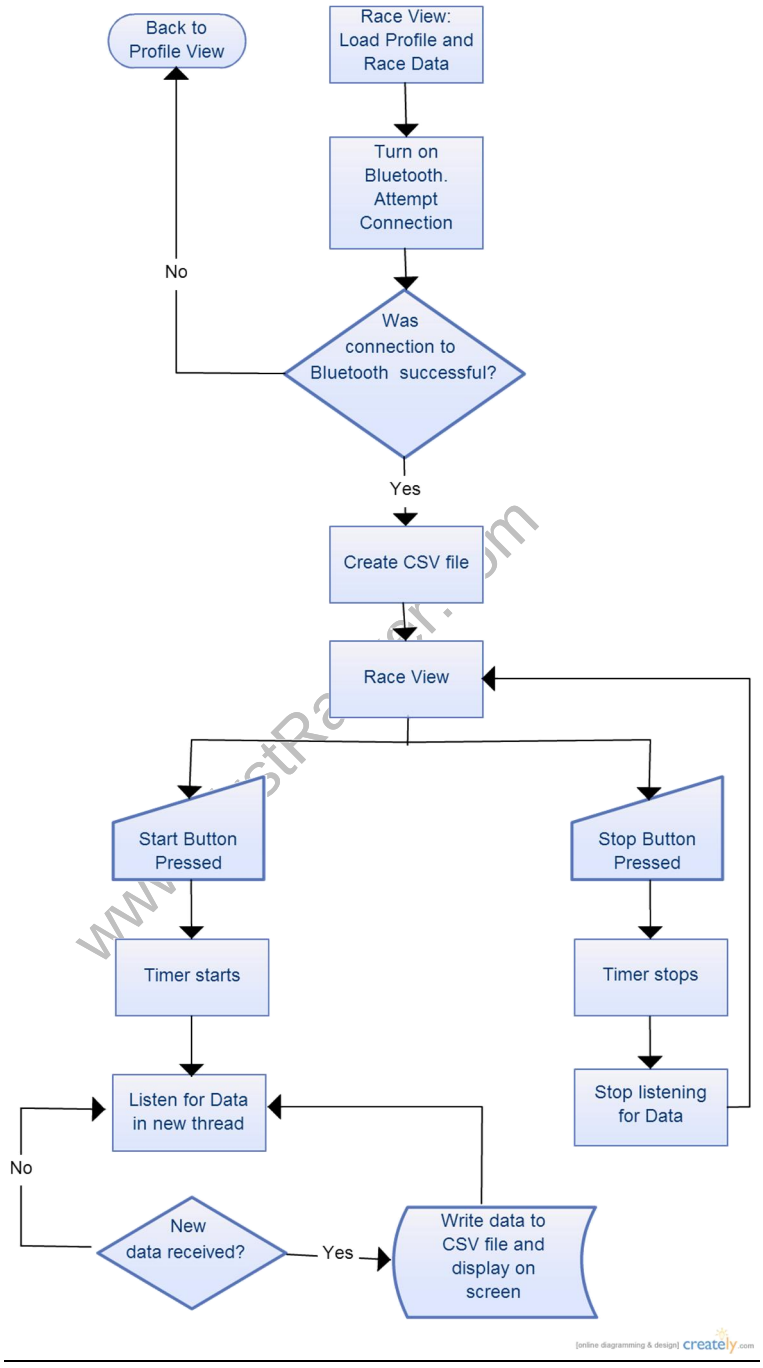

Figure 12: Software Flow Diagram for the Race Activity

Arduino Uno Software:

Separate from the Android application is the Arduino software. The Arduino acts mainly as a data collecting/forwarding device but serves other functions as well. Essentially, the Arduino collects data from the attached modules/sensors and sends this data to the Android device via Bluetooth. It also controls an attached fan that powers on when the temperature reaches a certain threshold. A detailed software flow diagram is displayed below.

The program begins by powering on the device. The program initializes the Software Serial pins used by the GPS and Bluetooth modules and also initializes external interrupts that will be fired whenever the reed switches are switched. After initialization, the program waits for a "transmit begin" signal to be received from the Android device. Once this signal is received, the program begins collecting and sending data approximately once every second. The GPS module's coordinates and the DS18B20's temperature data are read, along with the number of times the wheel and gear turned during that time frame. All of this data is packaged into a 20-byte packet as described in the "Race Activity" section above. The least significant byte is put in first, making the order Little Endian. This ordering was important to consider in the Race Activity since reading the bytes in the incorrect order can lead to bad data.

While the program is idling during the second that it's not sending data, it waits for signals to arrive from the Android device. The three main signals the Arduino checks for are: fan on, fan off, and transmit off, which are all self-explanatory. If the "transmit-off" signal is received, the Arduino turns off the fan and goes back to the "check for received signal" state until it needs to send data again or the Arduino is turned off.

Throughout the lifetime of the "transmit" state in the Arduino software, reed switches fire external interrupts that are handled by the Arduino. The time in between external interrupts is used to measure the wheel speed. Before transmitting the speed data the Arduino checks to see if the wheel speed has been updated recently. If it has not been updated for a specified amount of time, the Arduino

assumes the vehicle is stationary and sends zero as the wheel speed. A software flow diagram of the Arduino program can be seen below in Figure 13.

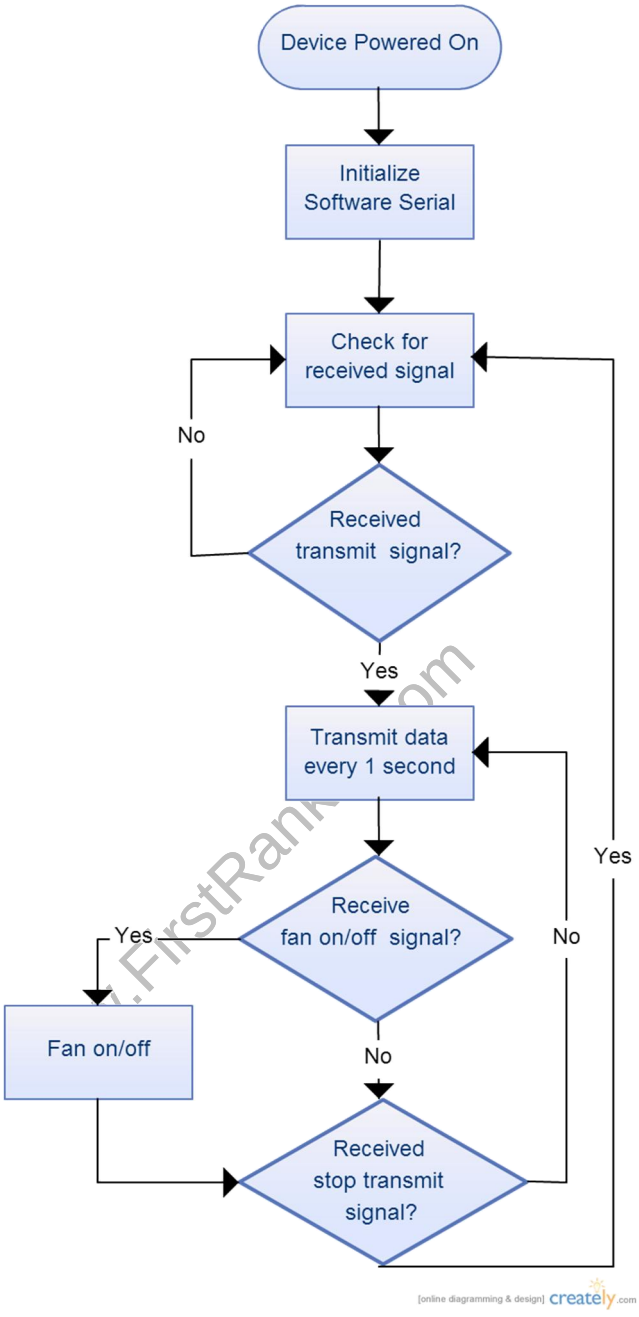

Figure 13: Software Flow Diagram for the Arduino Uno

Hardware Implementation:

Parts Design Decisions:

In order to collect data to be transmitted to the Android device, we integrated an Arduino Uno with an LS20031 GPS module, DS18B20 one-wire temperature sensor, JY-MCU Bluetooth module, and two reed switches. In addition to these parts, a 5V relay was used to power and control a 5V DC 50x50x10mm fan and an external battery pack with 4 AA batteries was used for powering the Arduino.

We chose the Arduino Uno microcontroller was because of its ease of programming, 5V output voltage, and ease of integration with other parts. The Arduino's language is simple to learn and easy to use, especially with the Arduino programming environment. It is C++ and C based, so many C++ and C programs can be written for use with the Arduino in combination with the built-in Arduino functions and macros. Along with the ease of programming, many parts have libraries and tutorials written for use with the Arduino family of microcontrollers. This makes it much easier on the programmer since libraries do not have to be written from scratch and parts can be integrated easily. These aspects make the Arduino a great microcontroller family for both beginners and experts to use to control their projects.

Of the many GPS modules to choose from we picked the LS20031 for its 1-2 meter accuracy and 5 Hz update frequency [9]. In any kind of racing situation, accuracy and speed are needed in order to get the best and most up-to-date data. This GPS module is also compatible with the TinyGPS library which was used in the project.

For the DS18B20 temperature sensor the accuracy and update frequency of reading temperature aren't as crucial as with position data. Because of this, we chose a standard temperature sensor that was easily integrated into microcontroller environments. It is compatible with the OneWire library.

For reading rotational speeds of objects, magnetic sensors are standard, namely reed switches and Hall Effect sensors. However, the two sensors are functionally different. When in contact with a magnetic field, a reed switch physically connects two pieces of metal together so current flows through

the circuit, acting like a physical switch would. On the other hand, when a Hall Effect sensor comes into contact with a magnetic field the output voltage increases or decreases. Because of the mechanical nature of reed switches they have much faster switch times. For this high-speed, high-accuracy environment we felt that a reed switch would be preferable to a Hall Effect sensor.

There are many different Bluetooth modules available on the market for use with microcontrollers. We chose the JY-MCU Bluetooth module because of its low cost, easy integration, and short signal range, which is about 10 meters. The JY-MCU was the quarter of the cost of similarly functioning Bluetooth modules such as the Bluetooth Mate Silver. The JY-MCU has a lack of documentation, however, so the low cost of the module comes with a price. The JY-MCU functioned easily in the project despite this lack of documentation.

We used a 5V relay and 5V DC 50x50x10mm fan in order to provide cooling to the rider. The 5V power for the fan was a good match for the 5V output voltage of the Arduino pins. However, the Arduino Uno digital pins don't provide enough power or current for the fan to run, but the 5V power output on the Arduino Uno does. A 5V relay helps accomplish this, which enables the fan to be hooked up to power and be controlled at the same time.

Finally, we used a 4 AA battery pack to power the Arduino Uno. Originally, a 9V battery was used to power the Arduino. When testing it with our device, however, we realized it did not provide nearly enough power and current for the 5V relay to switch. After some research, we decided that 4 AA batteries would best power the Arduino. Although 4 AA batteries provide 6V, which is less voltage than the 9V, they provide much more power and current. Also, the battery pack that holds the AA batteries has an on/off switch whereas the 9V battery had no such power control. This was better for our design since it is much easier to flip a switch than unplug a battery pack every time the Arduino needs to be powered down.

Black Box Diagram/ Schematic Description:

A black box diagram for our setup can be seen in the Figure 14 below. Input devices include the temperature sensor, reed switches, GPS module, battery pack, and Bluetooth module. Output devices include the 5V relay, DC fan, and Bluetooth module.

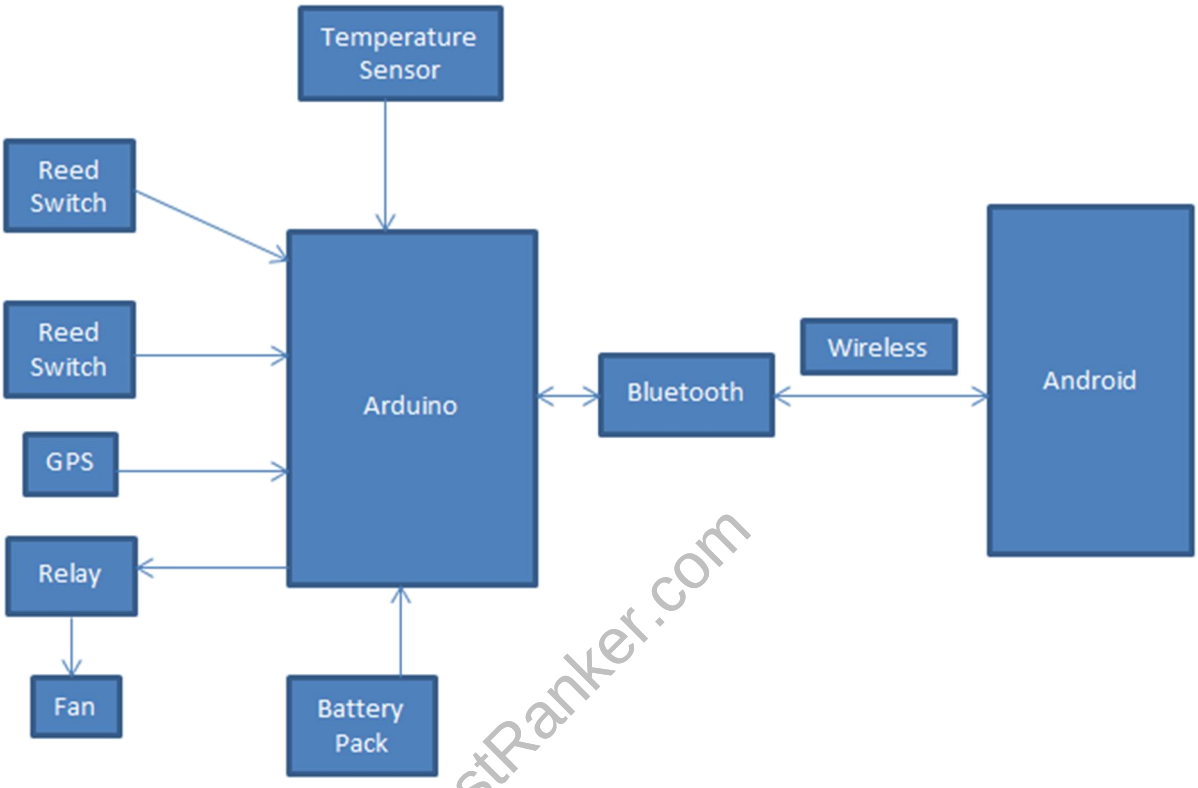

Figure 14: Black Box Diagram

A more detailed setup can be seen in the schematic on the next page in Figure 15, showing pin connections, power connections, and any other hardware not displayed in the black box diagram.

Most pin assignments were arbitrary, save for the reed switches. The Arduino Uno's external interrupt inputs are assigned to pins 3 and 2, so this is where the reed switches were connected since external interrupts are preferable to polling in high accuracy environments. Also attached to pins 2 and 3 are two 10kΩ pull-down resistors. This ensures that when the reed switches are not completing a circuit, pins 2 and 3 read a low value and don't mistakenly trigger an interrupt. The reed switches are

connected to the Arduino's 5V output voltage to ensure that 5 volts are given to pins 2 and 3 when the switch is triggered.

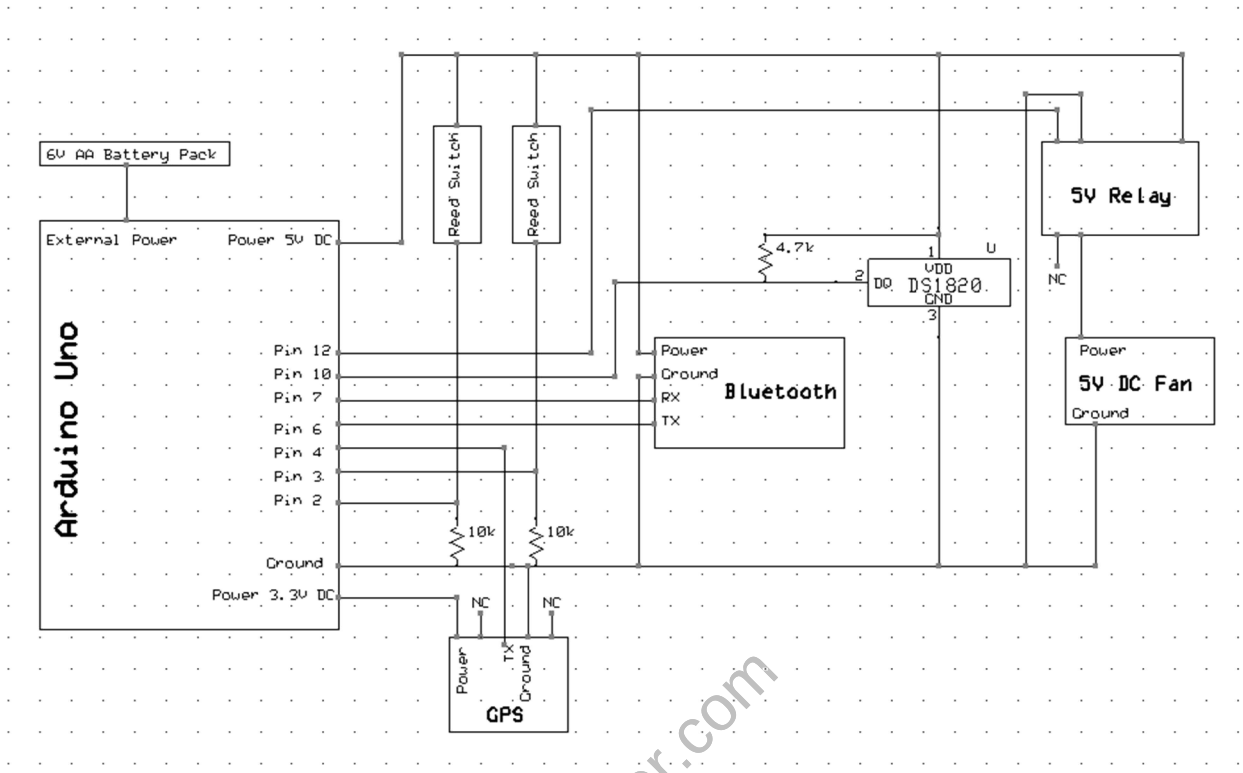

Figure 15: Arduino Uno and Components Schematic

Pin 4 reads the TX output from the GPS module and pin 5 is reserved for GPS RX input. However, since we are only reading input from the GPS module and not giving it commands via its RX pin, we are not using pin 5. The GPS module operates on 3.3V , so the power pin is connected to the Arduino Uno's 3.3V output [8]. The TX output of the GPS module is based off of 3.3V as well, however since this is considered a high voltage in an Arduino Uno input pin no external circuitry is needed. The Arduino's internal SoftwareSerial library was used with this module, so the hardware serial pins 0 and 1 were not needed for communication.

Pin 6 reads the TX output from the Bluetooth module and pin 7 outputs commands to the RX input. The Bluetooth module's power input is connected to the 5V output on the Arduino. Since the module operates on 5V, no extra circuitry was needed to interface it with the Arduino Uno. Like the GPS

module, the Bluetooth module uses the Arduino's SoftwareSerial library and thus was not connected to the hardware serial pins 0 and 1.

Pin 10 is connected to the one-wire pin of the DS18B20 temperature sensor. In order for the signal from the temperature sensor to be properly read, a 4.7KΩ pull-up resistor is connected between the one-wire pin and the power input pin of the temperature sensor [2]. Since the temperature sensor operates at 5V, the power input pin is connected to the 5V output rail of the Arduino Uno.

The 5V relay has five connections, four of which are used. The top left pins, one connected to pin 12 of the Arduino Uno and the other connected to ground, are used for inducing a coil in the relay. Once enough current is flowing through the coil, the switch in the relay is triggered, letting current flow from the top right pin of the relay to the right-most pin on the bottom left of the relay. Pin 12 of the Arduino serves as the triggering mechanism for the relay. When the output of pin 12 is high, the relay's switch completes the circuit. The top right pin of the relay is connected to the Arduino Uno's 5V output voltage rail, and lets that voltage and current carry over to the right-most bottom left pin where the fan's power input is attached. Simply connecting the fan's power input to one of the Arduino Uno's digital pins would not work since the digital pins output around 40-70 mA of current where the fan needs 200 mA. This is why the relay is needed. The relay ensures that the high current coming from the Arduino Uno's 5V output rail can power the fan while still being able to control the fan's power.

All components' ground pins are connected to the common Arduino Uno ground.

Project Enclosure Implementation:

Much thought was put into the final design of our project. Since the HPV team will have a different bike every year the components need to be compact and easily attached to the Arduino Uno, and the Arduino Uno needs to be easily placed somewhere on the bike. For our final project enclosure

we made sure as many components as possible could be housed within the same enclosure as the Arduino Uno.

Firstly, we chose to create our own custom shield for the Arduino Uno. With this, we were able to connect the relay and the reed switch's pull-down resistors to the Arduino without the need to create our own external circuit board for the relay and without the need to externally connect the resistors. This makes it much easier for the HPV team (and us) to connect the fan and the reed switches since less assembly is required. We also soldered male headers onto the shield so we could use female jumpers to connect the devices. The female jumpers we used also have a tight grip on headers and other jumpers connected to them, so there was no worry about any wires being dislocated.

Secondly, we chose to attach the Arduino Uno plus custom shield to an aluminum project box. The aluminum project box is sturdy but also easily drilled into. We drilled three holes on the bottom of the box lined up with the the Arduino Uno's and custom shield's three main screw holes. We were able to fasten the Arduino Uno down into the box using 3/4 inch #4 screws and appropriately sized nuts on the bottom of the box. Another hole we drilled in the box is lined up with the external power input to the Arduino Uno. This is so the battery pack can be inserted into the Arduino easily without having to open up the box every time the Arduino needs to be powered on or off.

Thirdly, we created our own pin array so all of the necessary parts can be externally attached. We used a female header array soldered to short wires that were then connected to female jumpers within the box. The female jumpers held a firm grip on the wires, so there was no worry about the wires being dislocated. A slot barely big enough to hold the pin array was drilled into the box so the pin array could be accessed externally. Finally, J. B. Weld was used on the pin array to ensure that it stays attached to the box. Photos of the final enclosure can be seen on the next page in Figures 16 and 17.

Figure 16: The Inside of the final enclosure, featuring the custom made Arduino Uno shield, battery pack with power switch, and the female header pin array.

Figure 17: The outside of the final enclosure, featuring the plug for the battery pack and the female header pin array attached to the enclosure with J.B. Weld.

For the temperature sensor we created a small custom circuit board to ensure that the pull-up resistor was properly connected.

Software Tools:

- For Arduino programming: Arduino 1.0.5 for Windows
    - Note: The SoftwareSerial.h library file must be edited. The #define on line 42 "_SS_MAX_RX_BUFF" must be changed from the value 64 to 128. This ensures that the RX buffer is large enough to store the data retrieved from the GPS module. If this is not included the GPS values will be read as "nan" or "ovf."
- For Android programming: Android ADT Bundle for Windows, which includes:
    - Eclipse + ADT plugin
    - Android SDK Tools
    - Android Platform-tools
    - Android Google Play Services
    - The latest Android platform
    - The latest Android system image for the emulator
- For creating schematics: ExpressSCH version 7.0.2 for Windows 7
- For creating software flow diagrams: Creately WebApp

Testing:

This project received ongoing, incremental testing throughout the development of both the Arduino device and the Android Application. Each component interfaced with the Arduino was individually tested before being used with the others. The components were gradually joined together, testing their combined functionalities together. Finally, all of the components were tested in a single application before using them as intended. For example, the GPS and temperature sensor were tested on their own and then brought together to display both module's data in the Arduino's serial terminal. Similar testing was done with the Android application. Testing was done on each new feature as it was

added to ensure proper functionality. For instance, when the feature of changing a profile name was added it was extensively tested to ensure that no bugs were hidden in the code.

There are always going to be bugs and mistakes found when testing. For our project, a big feature we had to test was displaying the correct wheel speed on the screen. Originally the Arduino code used the number of rotations per second to measure wheel speed rather than the amount of time in between rotations. This proved to be extremely inaccurate, only showing intervals of speeds rather than gradually increasing and decreasing speeds. After seeing this inaccurate behavior, the code was changed to measure the time in between rotations. This implementation was tested and proved to be a much better solution, matching almost exactly what a commercial reed switch displayed on an external computer. Another big feature that was fixed thanks to testing was external battery power. Originally a 9V battery was used to power the device. However, after some connectionless testing, we realized that a 9V battery did not give the Arduino Uno enough power to power the 5V relay. We instead used 4 AA batteries which provided enough voltage and power to ensure the functionality of the device. Without these and other tests, our project would have had many problems later on.

Related Works and Sources:

Senior Project vs Market Products:

There are many products in today's market that accomplish similar goals to our project. One such product on the market, currently priced at $249.95, is the Garmin Edge 500 Wireless Bike Computer [19]. Basic functionalities include capturing speed, time, distance, heart rate, and power. This data can then be pulled from the device via USB cable and viewed on a computer. The Garmin Edge 500 comes with a black and white LCD screen, optimal for viewing text and simple graphs. Our bike computer is similar in the fact that it records data for later use that can be accessed on an outside computer. Another product similar to the Garmin Edge 500 is the improved Garmin Edge 810 Bike

Computer [20]. In addition to the same capabilities as the Garmin Edge 500, it includes live GPS tracking, instant uploading of data, ride sharing capabilities, social network sharing, and weather updates. However, these capabilities come at a price: The Garmin Edge 810 is twice the price at $499.95. Our bike computer is similar to the Garmin Edge 810 in the fact that it includes live GPS tracking. However, the one feature that both Garmin bike computers lack is the separation of users. When data is recorded and stored, it is done for a certain user. The Garmin bike computers are for personal use whereas our application is for team use. Also, the Garmin bike computers record data that is relevant to weight loss whereas we record data that is relevant for the HPV team's race and their team performance. Our bike computer is a custom solution to a problem that bike computer companies have solved for personal use but not team use.

Related Works:

Matt Bell's Blog, Android and Arduino Bluetooth Communication: This project, created by Matt Bell and explained on his blog on January 2nd, 2012, involves creating a messaging application between an Android device and an Arduino Uno using a Silver Bluetooth Mate. The Arduino Uno uses the serial terminal in the Arduino software in order to send typed out messages to be displayed on the Android device. The serial terminal is also used to display the messages the Android device sends the Arduino. Matt Bell provides the pin connections, Arduino code, and Android Java code to run this project. This project was extremely helpful in first establishing a connection between the Arduino and the Android device. It provided the basic Arduino and Android code necessary to communicate via Bluetooth. This code was modified to fit the purpose of our project [7].

Androino!, instructables post by user metanurb: This project, created by user metanurb on instructables on March 10, 2012, involves an Arduino Uno communicating with an Android device using Bluetooth.

The Arduino Uno reads a value from an attached sensor and sends the data over to the Android device via Bluetooth. The Android device then displays this data on the screen. Metanurb used a Python script on the Android device in order to communicate with the Arduino Uno. This involved installing SL4A, which enables Python to be run on an Android device. Even though the project used a Python script instead of an Android application, it was helpful in the fact that it gave us a way to test communication between the Arduino and Android device[1].

Open Source Software:

Osmdroid Version 3.0.9: Provides tools/views to interact with OpenSTreetMap-Data. The OpenStreetMapView is an almost full/free replacement for Android's MapView class. It also enables the use of offline maps with custom tiles or the MOBAC program. In our application, it is used in Race mode to show the rider's position on an offline map. The offline map is obtained using MOBAC [14]

MOBAC Version 1.9.12: Short for "Mobile Atlas Creator." It is an open source (GPL) program which creates offline atlases for GPS handhelds and cell phone applications. MOBAC can use a large number of different online maps such as OpenStreetMap and other online map providers. In our application it is used in conjunction with Osmdroid to display offline maps to the rider [13].

TinyGPS Version 11: TinyGPS is designed to provide NMEA GPS functionality for Arduino users such as position, date, time, altitude, speed, and course. The library keeps resource consumption low by avoiding floating point dependencies and ignoring all but a few key GPS fields. In our application, TinyGPS is used to obtain the longitude and latitude readings from the attached GPS module, which are then passed to the Android device [15].

OneWire Version 2.1: OneWire lets you access 1-wire devices made by Maxim/Dallas, such as temperature sensors and ibutton secure memory. In our application, the library is used to read values from the DS18B20 temperature sensor, which are passed to the Android device [16].

StopWatch Class by Corey Goldberg, 2005: Creates an object that can be used like a stopwatch for wrapping blocks of code in timers. In our application, this class is used in Race mode to show the rider the current elapsed time of the race. It is also used when writing the data to the CSV file for a specific race [17].

Google Maps Android API: Allows you to access Google Maps on an Android device. Uses a map fragment to display the map and allows you to use all of the gestures that come with the native Google Map App [18].

Sources:

[1] Androino!, instructables post by user metanurb:

http://www.instructables.com/id/Androino-Talk-with-an-Arduino-from-your-Android-d/

[2] bildr.blog One Wire Digital Temperature Sensor Arduino Tutorial:

http://bildr.org/2011/07/ds18b20-arduino/

[3] Dr. David Janzen's Android App Course v3:

https://sites.google.com/site/androidappcoursev3/home

[4] Jimmanz ListView using BaseAdapter:

http://jimmanz.blogspot.com/2012/06/example-for-listview-using-baseadapter.html

[5] Lars Vogel Android ListView - Tutorial:

http://www.vogella.com/articles/AndroidListView/article.html

[6] Lars Vogel Android Camera API - Tutorial:

http://www.vogella.com/articles/AndroidCamera/article.html

[7] Matt Bell's Blog, Android and Arduino Bluetooth Communication:

http://bellcode.wordpress.com/2012/01/02/android-and-arduino-bluetooth-communication/

[8] Sparkfun LS20031 GPS Assembly Guide:

https://www.sparkfun.com/tutorials/176

[9] 66 Channel LS20031 GPS 5Hz Receiver Datasheet:

https://www.sparkfun.com/datasheets/GPS/Modules/LS20030~3_datasheet_v1.2.pdf

[10] DS18B20 One Wire Digital Temperature Sensor Datasheet:

http://datasheets.maximintegrated.com/en/ds/DS18B20.pdf

[11] JY-MCU Bluetooth Module Datasheet:

No datasheet could be found for this module. However, it is very similar to the Bluetooth Mate Silver, so the datasheet for the Bluetooth Mate Silver is provided instead. Functionality of the device was tested from source [1].

http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Wireless/Bluetooth/Bluetooth-RN-42-DS.pdf

[12] Reed Switch Datasheet:

https://www.sparkfun.com/datasheets/Components/Buttons/MDSR-4.pdf

[13] MOBAC Main Webpage:

http://mobac.sourceforge.net/

[14] OSMDroid Documentation:

https://code.google.com/p/osmdroid/

[15] TinyGPS Version 11 Documentation:

https://www.google.com/url?sa=f&rct=j&url=http://arduiniana.org/libraries/tinygps/&q=&esrc=s&ei=a_ekUcC6DsTTiwLOzoH4Bw&usg=AFQjCNFRZ6qPNxMA3OfIXNkoVizKgucgGQ

[16] OneWire Version 2.1 Documentation:

http://www.pjrc.com/teensy/td_libs_OneWire.html

[17] Java StopWatch Class Code and Docmuentation:

http://goldb.org/stopwatchjava.html

[18] Google Maps API Documentation:

https://developers.google.com/maps/documentation/android/

[19] Garmin Edge 500 Wireless Bike Computer Product Information:

http://www.rei.com/product/807089/garmin-edge-500-wireless-bike-computer

[20] Garmin Edge 810 GPS Bike Computer Product Information:

http://www.rei.com/product/855697/garmin-edge-810-gps-bike-computer

[21] Haversine Formula Wikipedia Page:

http://en.wikipedia.org/wiki/Haversine_formula

[22] Cal Poly Human Powered Vehicle Website

hpv.calpoly.edu

Conclusion:

We learned much from our senior project experience, especially about learning on your own and integrating two different engineering disciplines (mechanical and computer engineering). We learned how to make Android applications on our own with only the help of online resources and also different ways of measuring mechanical data with electronic sensors. This project was great experience for working in industry since many companies require engineers from different disciplines and backgrounds to work together. Another aspect of our project that we particularly enjoy is the fact that it will be in constant use by a team. Our project has a practical use and environment and will benefit the users

Our senior project was not without its difficulties, however. One of the greatest difficulties was creating an Android application from scratch with no prior experience. We had to rely on our own research skills and the information presented in numerous tutorials. Instead of having a vast background in Android programming we only worked off of bits and pieces we learned from tutorials along the way. This was a challenge, mostly because there were many simple solutions to problems we didn't know of because we hadn't encountered them or they weren't explained to us beforehand. Self-learning can be incredibly fun and rewarding, but it's not without its frustrations. Another difficulty we encountered was Bluetooth communication between the Android device and Arduino Uno. The JY-MCU Bluetooth module lacked in documentation, so many online tutorials were for other similar Bluetooth modules. Initial testing of the device proved difficult because the module never worked quite like the tutorials described. However, after much persistence we got the Bluetooth module working and all was well.

Finally, there is some future work that can be done on the project. Firstly, a more compact enclosure for the Arduino Uno and shield can be constructed so it is easier to place on the HPV. Secondly, the application could use some polish from a graphics design major specializing in app development. Finally, additional sensors/modules could be attached to the Arduino Uno for recording more statistics such as heart rate and calories burned.

Overall, we had a great experience with our senior project and feel that it has better shaped us as engineers. Learning on your own and working with other engineering disciplines is a lot of what working in industry is about, and we've done just that. We will take what we've learned from this experience and become even better engineers in the future.

Analysis of Senior Project Design

Summary of Functional Requirements: The project combines an Arduino Uno microcontroller and Android smart phone to make a computer system for the HPV team. The main functionality of the project is to collect data about the bike and present it to a rider via an Android application. The Arduino Uno uses an integrated GPS module, temperature sensor, and reed switches in order to collect data to be transmitted to the Android device via a Bluetooth module. The Android application takes this data and displays it to the user on the screen showing wheel speed, location on a map, distance traveled, temperature, and cadence. The application saves this data to a CSV file for later viewing. The application also features a user-profile system so each rider can have their own racing data saved to their own profile.

Primary Constraints: The main challenges of this project consisted of choosing inexpensive parts that suited our needs along with integrating communications between an Arduino Uno and an Android device. Many personal bike computers exist in the market but our implementation was unique to the Human Powered Vehicle team. Creating a cheap device was difficult considering our project required that an external GPS be used in order to gather information from outside the fairing of the bike, and GPS modules can be fairly expensive. Since timing and accuracy were primary factors in our project, choosing the right GPS that had fast data collection and a small error window took time and careful consideration.

Economic:

Original estimated cost of component parts: $120

Actual final cost of component parts: $148.72 for all materials purchased, including materials not used in the final implementation of the project.

Additional equipment cost: Equipment used in the development of our project include: A Windows PC with the ability to use the different programming tools listed in our report, a soldering iron, solder, various machining tools for cutting into aluminum. Also, an Arduino Uno and Android smart phone were not purchased since we already owned them.

Original estimated development time: Two quarters, not counting school breaks for design and implementation

Actual development time: Two quarters, not counting school breaks.

Environmental: The environmental impact of our project is similar to consumer electronics. The two main factors are production (soldering, wire scraps, plastic and metal manufacturing, etc.) and energy consumption (our device requires four AA batteries).

Manufacturability:

The metal encasing used to hold the Arduino device was bought pre-shaped and of default size. This solution, although inexpensive, creates an end product that is larger than it needs to be. The enclosure could be smaller and would allow for easier storage when using. The default metal enclosure also required holes to be drilled in order to allow access to the device.

Sustainability:

Issues for maintained use: Because the device will be used on bikes the constant motion and vibrations could loosen connections within the device. The use of the device in racing human powered vehicles

could also potentially break components if the device is incorrectly placed within the fairing and multiple ground impacts occur. These impacts could occur when the bike falls over at high speeds.

Project impact of sustainable resources: The project uses many electronic components and circuit boards which would need to be replaced when components failed. The project also requires the user have an Android smart phone in order to use the project App.

Upgrades for the design: Primary upgrades for this design might be to fabricate a custom shield for the project. The project enclosure could also be upgraded to better fit the device.

Ethical: The only ethical concern with the project is the possible use of the device in racing environments when such devices are not permitted.

Health and Safety: Main safety concerns arise when using the actual product in a racing or casual riding environment due to the fact that the device requires the user to look at a phone for information. This could be potentially unsafe if the user does not take necessary precautions when using the device while riding.

Social and Political: Although the creation of this device was for a senior project as well as an HPV competition, potential social issues can arise concerning fairness of the competition. Since the Cal Poly HPV team is using a device that could potentially help them improve designs and personal fitness while other teams aren't, this can be seen as an unfair advantage.

Development: During the development and analysis of the project we learned how to program in Java for Android devices. Although we knew how to program in Java we had never applied this to programming for Android devices. We learned how to program an Android application with the Android Development Tools (ADT) bundle for the Eclipse IDE. We also learned how to communicate with Bluetooth devices using Java and the Adruino programming language. Lastly, we learned how use the machining tools used to cut our devices metal enclosure.