Bike++ Embedded System: Project Report

Group: #30
Members: Rohail Kabani, Uzair Shafiq, Azfaar Qureshi
Course: ECE150 - 001
Date: Monday, December 4th, 2017

Table of Contents

Project Overview

The Bike++ embedded system is designed with one goal in mind; to improve the overall biking experience of any user. Whether it is an athlete working to achieve their dreams, a fitness enthusiast looking for extra functionality or a casual biker interested in the useful safety features the system has to offer, the Bike++ product implements something useful for every level of user.

The Bike++ device has three main components that come together in achieving the goal of the product;

1. An automatic bike lock to ensure the security of the bike. The lock takes input from the handlebars of the bike, and locks or unlocks automatically, based on the input. (This would use a potentiometer attached to the base of the handlebar, that turns with the turns of the handlebar. However, since this was not mechanically achievable, we read inputs from an external potentiometer attached to the breadboard).
2. A gyroscope for detecting axes tilt over time. The gyroscope sends the data it collects to the Omega for processing of statistics, which are useful for determining figures such as the calories the biker burned over the course of their trip. In addition, the gyroscope determines the incline/decline the user bikes at and computes the average at each of the three states, incline, decline, and flat; along with the total average, and the maximum and minimum angle the user bikes at.
3. A system of automatic blinker lights, that detect the tilt of the bike's handlebars to determine if the user intends on turning, and if they do, the lights turn on to indicate the intention to others. This is especially useful in busy cities at night time, when bikers are most at risk. For experienced and inexperienced bikers alike, it can be difficult to indicate such an intention, and sometimes a biker may not notice a driver behind them, as they may be listening to music, for example. These automatic blinker lights can help drivers and bikers both stay safe.

Our biggest roadblock in achieving all the functionalities we would have liked for this embedded system branched from one root cause; The Onion Omega IoT device. Most notably, the lack of documentation on different extensions for the device (such as the Bluetooth library, including its setup and implementation) made it difficult to even begin understanding the way it works at the level required to start developing for it. Once we had downloaded and installed all the libraries required, we went through their setup on the terminal and connected all the hardware components together (which in itself was a grand feat), there was no reliable documentation available online to actually begin developing for it. The group came to this realization after spending over three entire days attempting to communicate with the Omega via the Bluetooth extension. As well, the gyroscope we have is unable to communicate serially with other hardware components, and can only use I2C communication. The Omega, on the other

hand, works the opposite; it can communicate serially, but I2C protocols were not working, even after following the tutorial posted on piazza. The solution we implemented to overcome this was using an Arduino to collect the gyroscope readings, then transferring this data serially to the Omega for processing; the arduino can read and transmit both forms of data, so it acts as a data transfer bridge.

Ideally, if our project could do everything we originally planned, it would pair via Bluetooth with a “trusted device”, that is initially set by the bike owner. This may be a phone, laptop or some other Bluetooth enabled device. In the presence of the device, the bike would automatically unlock, and of course then inversely, when the trusted device is not detected nearby, the bike lock would clamp shut. This would utilize the Onion Omega Bluetooth expansion to test the signal strength of the trusted device, allowing our code to make a decision of whether or not to remain locked based upon that input. As well, if we were developing this system for the market, we would ideally create a webpage to initialize the Bike++ system. Because utilizing Bluetooth signal strength to lock or unlock the bike is not a viable solution as stated above, we overcame that with another protocol; utilizing the handlebar to create a “passcode” for the bike. For this new solution, when initializing the system, the user would log into the site and choose a combination for the lock. An example combination would be: Right>Right>Left>Right>Left. This means to lock or unlock the bike, the user must, in this order, perform twists/turns via the handlebar. Using the potentiometer, this input will be detected and the according action (locking/unlocking the bike) will be performed. Since we had already developed an implementation for the automatic blinker lights that also ideally utilize a potentiometer linked to the handlebar of the bike, implementing this functionality just involves manipulating pre-existing code.

There are some other small, mostly design based features that could not be fully realized due to their mechanical natures. Specifically, creating an effective bike mount that is perfectly out of the way of the user, and attaching the potentiometer hardware to the bike’s handlebars. If this project were to be sent out to market, the latter would be very crucial because that is what is necessary for the potentiometer to actually “get input” from the handlebars, but since this project was based more on hardware and software components, we prioritized the gyroscope, turning lights and general code and hardware behind bike lock system over the mechanical features that could be implemented to improve user friendliness.

As a result of the aforementioned effective prioritization, we were able to deliver all the core features and functionalities that we had originally planned for the embedded system. Although the implementation of the feature may be slightly different than we had originally planned (which is bound to happen in such a project), the three core features are still present; the automatic bike lock, automated turning signals and gyroscope with accompanying statistics.

System Design

The following is an overview of the complete hardware and software system design.

Hardware

Arduino Uno: Used to collect raw data values from various hardware components such as the potentiometer and gyroscope.

- Ideally, we planned on using the Arduino Dock 2 with the Onion Omega, but due to some incompatibilities such as digital versus analog output type and I2C versus Serial data communication, we were unable to use the dock.

Onion Omega: Communicates serially with the Arduino to gather the data collected from the gyroscope and potentiometer. Computes statistics for the gyroscope, controls LED lights and other functionalities related to the potentiometer and maintains (prints/reads) log files to track program flow over the duration of the program runtime.

- Does not directly communicate with potentiometer because the Onion Omega can only read digital value inputs, our implementation requires analog input.
    - Attempted troubleshooting via piazza posts and online research, but no reliable solution was found.

Potentiometer: Used to track the degree of rotation of the bike handlebar. This information is used in the automated bike lock and automated turning signals.

- Mechanical challenge: Unable to mount potentiometer to the actual handlebar, so it is connected to the breadboard instead.

Gyroscope: Used to collect bike tilt (yaw, pitch and roll), and accelerometer data over time. This raw data is collected by the Arduino, then transferred serially to the Onion Omega to be processed. Using a “buckets” system, where the data is sorted by value into buckets like a histogram, when the program is finished running (the bike user has completed their ride), ideally we would use the buckets to determine the amount of time for which the biker was riding at each bucket angle and compute the calories they burned using that information.

Lights: Part of the embedded system functionality, automatically controlled by code executed based on potentiometer readings.

Bike Lock Hardware: Since this product was mostly a proof-of-concept design, we used cardboard to make the lock motor housing and the lock itself. When power is sent through the motor, it spins and the cardboard hook-shaped lock swings into place between the spokes, locking the bike and preventing movement. Then, to unlock, the hook simply swings the other

way. The bike would lock when the user enters their “trusted password” using the handlebar to communicate with the system (turning their handlebar to follow a certain pre-set pattern, and upon recognition of this the bike locks or unlocks).

- When attaching a battery straight to the motor, it worked fine. However, when plugging it into the omega GPIO pins, they were unable to power the system. At first, we thought this might have been a problem with the pins or motor themselves, but after conducting research came to the conclusion that the GPIO pins were unable to provide enough power to the motor. After conducting more research, we learned we likely needed to create circuits using MOSFET Transistors that would act like voltage-actuated switches; this would allow us to attach the GPIO pins to the bridge of the transistor and control it, and so the battery could be directly connected to the motor in series, allowing it to provide enough power to run. Due to our lack of experience with hardware and circuits, however, we were unable to follow through with this plan.

Wires/Resistors/Breadboard: Used to organize and maintain the integrity of the hardware components, preventing current overload on components.

Software

Gyroscope Data Collection: Script to collect raw data from the gyroscope using I2C protocols, and writing the data to a file that can later be read. The gyroscope is connected to an Arduino for compatibility reasons, which then transfers the data to the Omega for processing.

Gyroscope Statistics: Serially transferring Gyroscope data from Arduino to the Omega for processing. The information can be read like a file, allowing for statistics to be computed over the dataset. Specifically, we want to consider the tilt of the bike over the course of the ride, on intervals rather than an overall average. This allows us to determine for what intervals the biker was moving uphill, downhill or on a flat surface, which is valuable information we can use to calculate more accurate measurements, like the calories burned by the bike user over the ride.

Log Files: By calling a log file method that writes the current status and “location” of the computer thread every time it enters a new function, exits or enters a loop or loops again, along with a timestamp of when it does so, we can keep track of the program flow. To make log files more readable, every log entry also comes with a tag, such as “(CODE FLOW)” which represents the program is entering a new function or looping, or “(FATAL)” which means the program had a fatal error that has caused a crash, along with a description of what line the program was at when this happened. This allows for much easier code debugging and system failure diagnosing. In an embedded systems project like Bike++ where there are many commands executing and many functionalities working together, such log files are imperative in determining the root causes of program failures.

Program Timer: For logging, statistics, and the program watchdog, we need to keep track of time. Having timestamps on the logs written to the log files will allow for easier code debugging and troubleshooting. As well, for statistics we must take into consideration the time interval over which the bike user rode uphill, downhill or on a flat surface. Using the time.h and ctime C/C++ libraries, we can determine the current time down to the nearest second to ensure precise log files. Other programs in the embedded system call this function, which returns an appropriate timestamp.

Program WatchDog: Checks for response from each component of the program at set time intervals. If that component does not respond (write to log files) for a certain threshold of time, an appropriate action is taken; When debugging, this action is to stop the program, and write to the log file why it stopped the program (which piece of code was not executing correctly).

Automated Blinker Lights: Script to collect and analyze potentiometer reading over time, using analog input method. The device was calibrated such that if the threshold for the twist of the handlebar (potentiometer) is exceeded, the blinker lights for the appropriate direction begin to flash. The time intervals between flashes, the animation for easy visibility (so drivers can easily tell a biker is up ahead, especially at night) and the general code for lighting up specific lights based on potentiometer reading must come together to successfully create this portion of the embedded system. As well, using these potentiometer readings, the automated bike lock is controlled.

Automated Bike Lock: Ideally, we wanted the omega to use it's bluetooth expansion to connect with any bluetooth enabled device of the owner, such as a laptop, phone, or perhaps both (in case one was out of battery and so could not be recognized). Upon detection of the trusted bluetooth device nearby, the lock would automatically open. Upon realization of the lack of documentation behind the omega, we adjusted the design such that the lock is controlled by a potentiometer, the same one that would be attached to the handlebar for the automated turning lights. When setting up the Bike++ system, the user would enter a trusted sequence of handlebar twists, and upon successfully inputting this same sequence, the lock would lock or unlock.

- Since we are not experienced with circuits and hardware, we were unable to implement our design to control the bike lock; A two series circuits, with transistors controlled by the GPIO pins of the omega, acting as switches between the battery and the motor. When one circuit is given power, the lock would swing in one direction, and when the other is powered, the lock swings in the other direction (to lock or unlock accordingly).

Software Design

State Diagram(s)

Figure 1: state machine of parsing Serial line Data

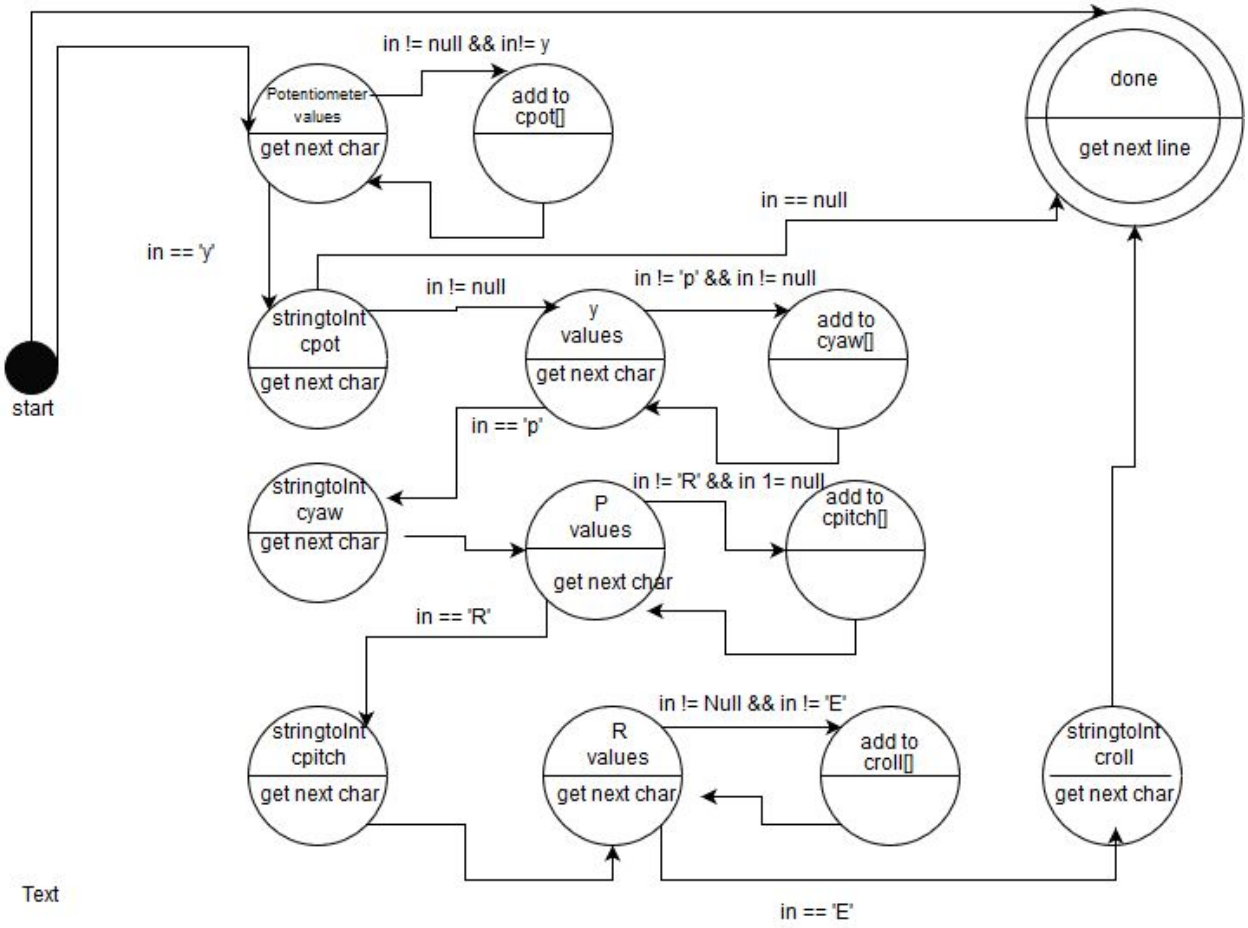
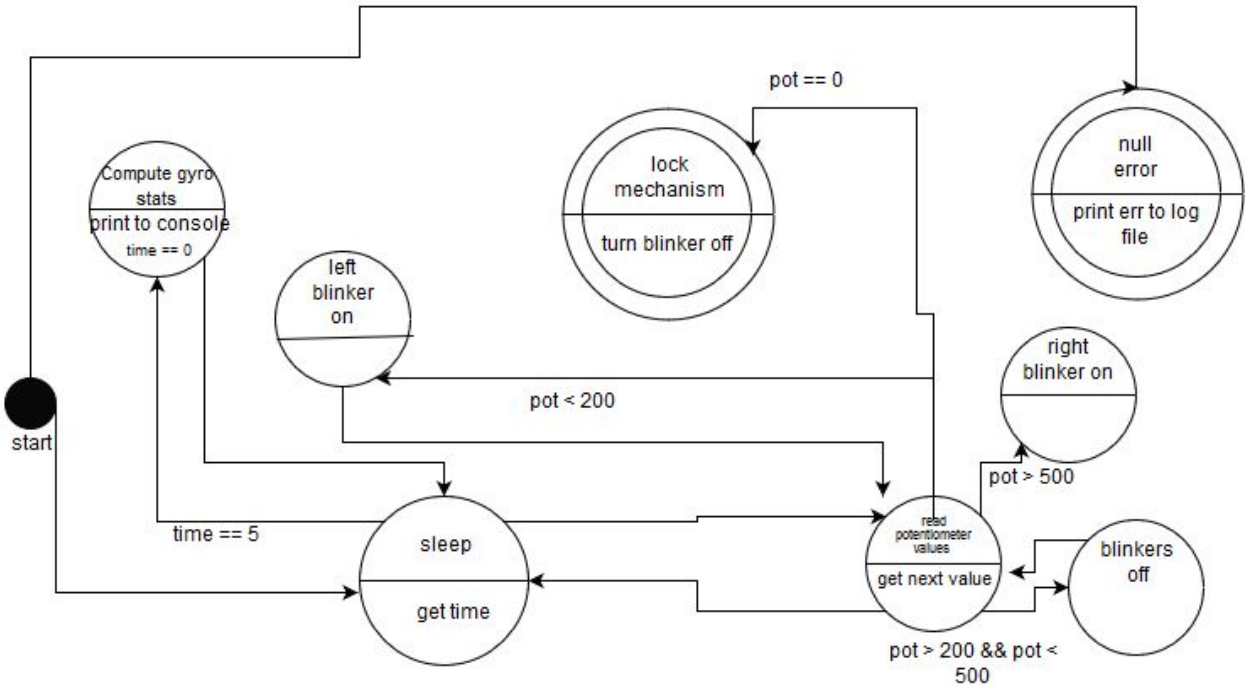
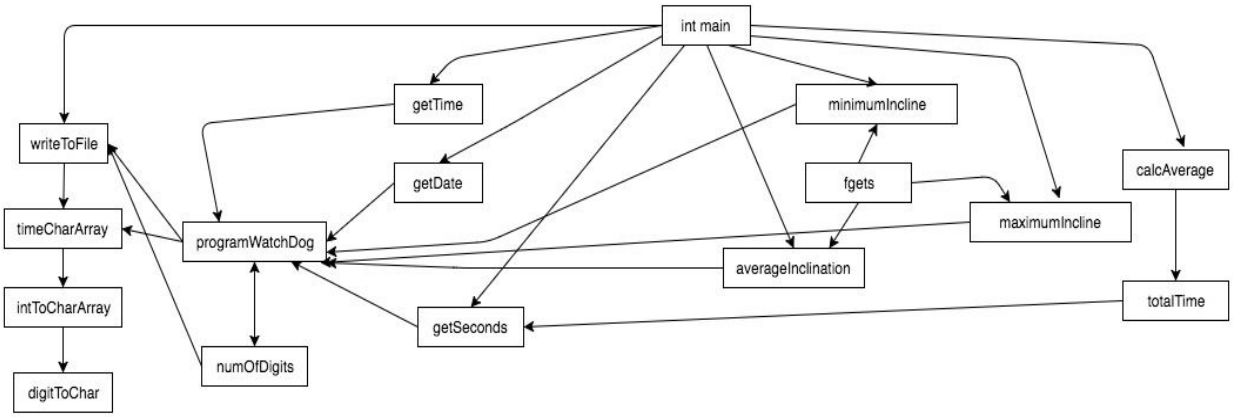


Figure 2: State diagram of program after it parses Serial data



Function Call Tree



Classes and/or Structs

- *struct* Accelerometer
    - float x. This value stores the values computed from the accelerometer that are responsible for the g-force acting on the x axis.
    - float y. This value stores the values computed from the accelerometer that are responsible for the g-force acting on the y axis.
    - float z. This value stores the values computed from the accelerometer that are responsible for the g-force acting on the z axis.
- *struct* Gyroscope

    - float yaw. This value is responsible for storing the gyroscope value that is communicated from the Arduino, and represents the directional value (north, south, west, east) of the bike.
    - float pitch. This value is responsible for storing the gyroscope value that is communicated from the Arduino, and represents the degree of incline of the bike (range -270 to 270).
    - float roll. This value is responsible for storing the gyroscope value that is communicated from the Arduino, and represents the rotation of the bike with respect to the forward direction.

- *struct* Date
    - int month. This value stores the month date.
    - int day. This value stores the month day.
    - int year. This value stores the month year.
- *struct* Time
    - int hour. This value is store time in terms of hours.
    - int minute. This value is store time in terms of minutes.
    - int second. This value is store time in terms of seconds.
- *struct* TheFloat
    - This struct stores all the necessary information needed to be stored when converting a string to a float
    - *int* floatNegative = 1; This value stores whether or not the number before ‘e’ in a float is negative or not.
    - *bool* decimalPoint = false; This value stores whether or not the number contains a decimal or not.
    - *bool* minus = false; This value stores whether or not the number after the ‘e’ or the power is negative.
    - *bool* plus = false; This value stores whether or not the number after the ‘e’ or the power is positive.
    - *bool* exponential = false; The value is set to true if 'e' shows up.
    - *int* exponent = 0; The value stores the numerical value of the exponent, or the number after the ‘e’.
    - *int* exponentNegative = 1; This value is set to -1 if the part after 'e' is negative.
    - *float* postexp = 0; This value stores the value after the exponent.
    - *float* preDecimal = 0; This stores the number before the decimal.
    - *float* postDecimal = 0; This stores the number after the decimal point to the point before the ‘e’.
    - *float* postDecimalDigits = 0; This value stores the number of digits the decimal contains.

System-Independent Components

Gyroscope statistics. Although this piece of software is used by our embedded system to compute statistics over the dataset collected by the gyroscope, it is not exclusive to the gyroscope; to test this software, any dataset can be used, and the code is transferrable to other hardware, too, so it is system-independent.

Log Files - The log file function simply writes to the log file based on the “location” of the program thread. Since it does not rely on any specific hardware or other software (but does require some software to be running in order to have something to write to the log files), it is still considered independent of the system.

Program Timer - Simply returns a timestamp of the time when the function is called. This software functions completely independently, and to test it, the function can simply be caused and the array of characters can be printed to the console, to ensure that the timestamps are accurate and formatted correctly.

Program WatchDog - This software is also completely unreliant on any hardware components. It simply keeps track of the last time it has been pinged by any of the other software “parts” involved in our project, and if it has not been pinged over a certain duration, it causes the program to terminate, and prints why to the console for debugging purposes.

System-Dependant Components

Automated Bike Lock and Automated Turning Lights - Both these software components rely on readings from the potentiometer attached to the breadboard. They would both access this data, and process the information differently. The automated bike lock would keep track of the past “x” (any number, chosen by user) twists/turns of the handlebars, and upon detection of a successful sequence, lock or unlock the bike. The automated turning lights do not need to store information from past twists or turns of the handlebar, but instead only need to process instantaneous potentiometer data. Regardless, both pieces of software rely on readings directly given to them through the potentiometer, making them system-dependent software components.

Gyroscope Data Collection - This piece of software reads the data given to it by the gyroscope hardware, and then takes this data and serially transmits it to the omega for processing. Unlike the gyroscope statistics, this piece of software directly works with the gyroscope, and if the gyroscope was nonfunctional, this code would not work either. This piece of software acts like an abstraction tool between the gyroscope hardware and gyroscope statistics software.

Logging Infrastructure

Logging was a very useful feature in debugging our system. Over the course of the development of the embedded system, we realized there are a lot of parts that need to work together just as well as they do separately. Just because a piece of code works on it's own, does not mean it will also work when integrated with other software. We experienced this when we brought our three main components of the product together. The code would sometimes compile, sometimes not, sometimes run for 10 seconds before crashing due to a segmentation fault, and other times working perfectly for an hour. We needed a more effective way of analyzing almost a thousand lines of code, especially when we did not all write the code together, so we did not understand it entirely (we split the software development into parts). In order to effectively log the program flow, we created a writeToFile function that writes the precise moment that the computer is doing a particular task. Every time the code enters a new function, a new loop or loops again, it calls the writeToFile function that prints exactly that to a log file. This allowed us to easily determine, for example, what component of the program was causing segmentation faults. In our code that reads and works with inputs from the gyroscope and potentiometer, we had delays. We noticed that after about 33 seconds of running, the program would crash when executing these lines of code. We came to the determination that there was some memory leak or program fault caused by the delay; perhaps the system thread was becoming asynchronized due to the delays. In order to fix this, we took the delays out of our code, and instead used manual timers. Now, the code runs every clock of the processor, but will not do anything unless a counter variable hits a certain count (approximately once a second), and we can work accordingly. Thanks to effective logging files that print to the logFile what the program is doing at every stage that it is running, with a timestamp, we were able to easily fix this flaw in our code, and learned the value of log files, especially in complicated embedded systems where so many components come together and are working at the same time.

Testing

Since the Bike++ embedded system consists of three major separate Bike upgrades, each of them had to be tested separately, then together, to ensure proper functionality. Following is the process undergone to test each of the components:

Gyroscope & Accompanying Statistics:

The overall general functionality of the gyroscope hardware and its related software was to track the tilt of the bike over time. Using this information, we can tell if the biker was riding uphill, downhill, or on a flat surface. This information is crucial in calculating more accurate statistics, such as the calories the biker burned on their ride,

that pure software applications can not accurately determine. Therefore, there were two main stages of testing this part of the embedded system; first, testing that the gyroscope can accurately measure its angle of incline in a two-dimensional space at a certain instance in time, and secondly, that it can do so over longer periods of time too (like on a biking trip). To test the former, the group took the finished bike apparatus to the Skate Park in the Waterloo Park, located near campus. To start, the gyroscope was calibrated such that flat ground would not be recognized as an incline. Then, by measuring the angle of some ramps at the park using a protractor (which is not amazingly accurate, but will do for this test) we compared the gyroscope readings to the angles we manually measured, and found that the gyroscope was generally accurate within just a couple of degrees to what we measured, and the readings were comfortably within the accuracy range required. The datasheet for this first test is attached as an excel file. Note that it is entirely possible that the gyroscope is more accurate than the angles manually measured using a protractor, which may be a cause for discrepancy. Since in our code we sort the data into buckets, however, small variances should not affect the accuracy results significantly. Also, since some readings are above the expected and others are below, the dataset isn't all skewed in one direction, so there is no further manual adjustment we can perform to further strengthen the data. After testing the gyroscope and accompanying statistics at the skate park, it was now time to actually take the bike for a ride. Since at the time we had not yet created an actual housing for the hardware, and instead it was taped onto the seat of the bike, we could not ride it; Instead, one group member held the bike and walked alongside it, with another holding the laptop nearby, and the third timing the trip. We went to a nearby park over the weekend in Mississauga to test that the system is functioning correctly. The bike started at the top of a hill, and was walked down to the bottom, where there was a temporary plateau. After walking on this flat ground for around 20 seconds, with the bike alongside, there was another small hill, which we went up and over, followed by a flat region again. At the very end of the ride, in excitement, we lifted the front wheel of the bike, pointing it to the sky. It should be noted that the gyroscope we used does not depend on or measure altitude; instead, it is measuring the angle at which the rider is moving at a certain time, which if graphed, should look like the derivative of a function measuring altitude over time. After inputting the data points received by the gyroscope on excel and graphing the relationship, we were able to come to the conclusion that the gyroscope is likely working very well. Clearly, as can be seen on the graph, the bike starts moving downhill for the first ~20 seconds of the ride, which is why the angle is negative (below horizontal). It gradually decreases angle over time, because the hill was curved and went from steeper to flat ground. After ~20 more seconds on near flat ground, we went over a small hill, then back down, and onto flat land again. The graph of the curve perfectly represents this. Therefore, from these two tests we were able to conclude that the gyroscope and statistics are fully functional. (Refer to appendix page 25 for excel datasheet and graph of gyroscope readings used for this analysis).

Automated Bike Locking System:

The way the automated bike locking system was initially designed to work was by monitoring bluetooth signal strength to determine proximity between the user's phone and the bike. When the signal strength is strong, it suggests that the owner is close to their bike and the lock will automatically open and when the signal strength is weak then the bike will automatically lock. However, in oversight of the hardware restrictions of the omega, we ended up buying two components that only transfer data over I2C protocol whereas the omega only had the capability to support one device at a time. We decided the gyroscope was of greater importance than the locking system and left this as a proof of concept. Our locking system was quite simple: a cardboard scythe-like protrusion that would rotate into the wheel spokes to lock the bike and rotate all the way back to unlock the bike. The GPIO pins on the omega weren't capable of providing enough current to rotate the motor so instead we connected a MOSFET transistor to an external power source and connected the gate to the GPIO pin. This way we could use the transistor as a switch and control the motion of the motor. However, we weren't experienced enough electrically to create a circuit capable of switching the ground and source of the motor so we can rotate the motor backwards. To test what we had, we outputted a voltage from our GPIO pins on the omega for a few seconds to see if the motor would spin. If it did, then all we had to do with the bluetooth was to set the correct 'if' condition. We know we can correctly parse data serially as we were doing so for the gyroscope. Since we did not have bluetooth data we added a special 'lock value' from the potentiometer. When the potentiometer was rotated 90 degrees to the horizontal, then the bike lock would start.

Automated Blinker Lights:

Along with the gyroscope data the arduino also sent the current value of the potentiometer circuit since the Omega did not have any analog pins. The potentiometer changes resistance depending on the angle of rotation of the device, which means we can draw a correlation from resistance values to the angle of rotation. Ideally, the potentiometer would be connected to the bike handle, but that was purely a mechanical challenge so we focused on the electrical and software side of things. We testing this component in parts; first we had to make sure we could correctly parse the correct string to get the values of the potentiometer and correctly store and update the value every clock cycle. Once we accurately did that, we would make sure we could turn on and off the correct gpio pins. Lastly we connected the circuitry to the corresponding GPIO pin and testing the entire system as a whole.

Limitations

Our embedded system project has some limitations that we did not foresee. Firstly, the project does not utilize the Onion Omega bluetooth expansion as we had originally hoped to achieve. As well, since we are inexperienced with hardware and circuits, we were unable to

implement the solution to the bike locking system that we encountered; Since GPIO pins do not provide enough power to the motor, we would ideally attach the lock directly to batteries, and use a transistor attached to the GPIO pins to act as a switch, allowing us to easily lock and unlock the bike. In a market-level product, of course, a biker expects to be able to actually ride their bike with the Bike++ embedded system adding features. However, since we wanted to focus on perfecting software and hardware components of the system itself, we were unable to create a proper housing for the embedded system, so if the user wanted to ride their bike with the system installed as we had for the demonstration, it would be… uncomfortable. Finally, one last problem we encountered was that our gyroscope did not provide velocity data as we originally had expected, only acceleration information. Because of this, we were not able to combine the two major stats (velocity and bike tilt) to calculate an extremely accurate measurement of the number of calories a rider burns on their trip. However, other than these aspects, our embedded system works exactly as expected; the potentiometer works and the arduino collects data from it and the gyroscope, serially transferring this information to the omega for processing. The omega outputs to the console, and the values are accurate as expected, the bike lock swings in and out, preventing the spokes from turning when locked, and the logging infrastructure effectively writes to files, allowing for easy debugging.

Our gyroscope also introduced some limitations; it was a little unpredictable at times. Sometimes, it would stop reading values, and just print 0’s and -1’s to the terminal, or other times, seemingly completely random values. After re-calibrating, these faults would dissipate temporarily, but would often soon occur again. Luckily, we were able to find a permanent solution online; to re-solder the gyroscope onto the breadboard using a different mount and different technique, and to also solder the wires to the apparatus to ensure it never loses signal. It turned out that it was not the software or even the hardware of the gyroscope itself that was causing this problem, but just the connection between the gyroscope and the arduino. This led to a very important lesson learned, that will be discussed in the next section.

Lessons Learned

Since we are not used to working with hardware and circuits, we did not expect the troubles we had with the hardware components of the project. The majority of the work that went into this project stemmed from problem solving the hardware components; trying to set up the omega bluetooth library, learning that the gyroscope can not serially transmit data, but rather only using I2C communication protocols, figuring out that the GPIO pins cannot provide enough current to properly propel the motor, and much more; and so the biggest lesson we learned was the importance of researching and having a deep understanding of the hardware involved in a project before jumping into it. Unlike pure software projects that rely on algorithms and logic that can, if not easily, at least instantly be implemented to solve a problem, if the wrong hardware is purchased, it cannot be fixed, and so proper care and consideration has to

be put into each of the components involved. As well, there is generally much less documentation to be found online when working with hardware than when working with software, and so it is important to choose parts that may have more information to be found online; for example, to control all the parts of this project, it may have been easier to use a single arduino over the omega, just because there is a lot more documentation online for how the arduino works, and how different parts work with it. When working with hardware, it is very important to have a realistic timeline for the project. When we were working on the weekend, we expected to be able to complete the majority of the project over those two days. However, as we got deeper into the project, we realized that there are a lot more considerations that go into working with hardware, and so a lot of our time was spent doing research online. Therefore, knowing when and where to find resources to help is also an important lesson we learned, as well as to keep a realistic timeline; If the software behind the project can be completed in a day, and the members of the group are not too familiar with the hardware, expect it to take multiple days to complete the project.

When working with our gyroscope, we learned a very important lesson; hardware components are not like software ones, in that it is often difficult to figure out where a problem is stemming from. In software, thanks to sophisticated compilers and debuggers, it can be quite simple to locate at least the function call or general code that is not working properly. However, when we were trying to determine what was causing our gyroscope to act strange, we did not know where to begin; perhaps the gyroscope hardware itself short circuited, causing random values to be transmitted to the terminal; or maybe it was the software behind the statistics, the connection between the arduino and the gyroscope, the connection between the omega and the arduino, or some other general interference, which we learned could be a causing factor, too; There are many different paths to consider when debugging such a system.

In terms of software, there was a lot to learn, too. Most notably, the importance of logging infrastructure. In our labs, we complete assignments with one main purpose; whether it's a sorting algorithm or a homework queue, there is only one component to troubleshoot. However, in an embedded systems project such as this one, where many components must work together, without having proper logging infrastructure, it can be nearly impossible to troubleshoot problems with the code. We learned that it is a really bad idea to implement logging infrastructure near the end of the project; instead, it should be implemented at the very beginning, so that problems can be detected and fixed as they arise.

Appendix

A. Abridged version of Code:

```
#include <stdlib.h>            //used for testing and for general functions
```

```
#include <math.h>           //used for statistical computation
#include <time.h>           //used to gather time/date of current day
#include <unistd.h>         //require for use of other libraries
#include <iomanip>          //required for table formatting
#include <string.h>         //required for use of strings
#include <fcntl.h>          // required for reading Arduino data
#include <ugpio/ugpio.h>    // required to read input/output of omega's GPIO pins
#include <fstream>          //required for file reading/writing

////////////////////////////////////////////////////////////////////////////////
//////////////STRUCTS///////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
struct Accelerometer {
  float x;
  float y;
  float z;
};

struct Gyroscope {
  float yaw;
  float pitch;
  float roll;
};

struct Date {
  int month;
  int day;
  int year;
};

struct Time {
  int hour;
  int minute;
  int second;
};

struct TheFloat {
  int floatNegative = 1;
  bool decimalPoint = false;
  bool minus = false;
  bool plus = false;
  bool exponential = false; // set to true if 'e' shows up
  int exponent = 0; // this is the exponent
```

```
  int exponentNegative = 1; // set to -1 if part after 'e' is negative
  float postexp = 0;
  float preDecimal = 0; // the number before the decimal
  float postDecimal = 0; // the number after the decimal
  float postDecimalDigits = 0; // # of digits postDecimal has
};

////////////////////////////////////////////////////////////////////////////////
//////////////Declaration ////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

Time getTime ();
Date getDate ();
Time timeElapsed (const Time startTime, const Date startDate, const Time time1, const Date
date1);
Time timeElapsed (const Time startTime, const Time time1);
void averageInclination (const Gyroscope gyro, float avgIncline[], Gyroscope& previousGyro);
void maximumIncline (const Gyroscope gyro, float& max);
void minimumIncline (const Gyroscope gyro, float& min);
float totalTime (float avgIncline[]);
float calcAvg (int avgIncline, float totalTime);
int getSeconds();

////////////////////////////////////////////////////////////////////////////////
//////////////GLOBAL VARIABLES////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

int startingTime = getSeconds();
Date startingDate = getDate();
int offset = 10;
unsigned int usecs = 100000;

////////////////////////////////////////////////////////////////////////////////
//////////////FUNCTIONS////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

char digitToChar (int digit){   // Turning a single digit into a character
}

void intToCharArray (int x, char charArray[], int numDigits){
        //converts an integer into a string
 }
```

```
void timeCharArray (char timeStamp[]) {
	//update char array with current time stamp. Used in log file writing
}

void writeToFile (char type[], char message[]){
	//opens stream to log file and writes message
}

Time getTime () {
	//returns current time using time.h library
}

int getSeconds () {
	//returns current time in seconds using time.h library
}

Date getDate () {
	//returns year, month, date and day
}

Time timeElapsed (const Time startTime, const Date startDate, const Time time1, const Date
date1) {
	//finds time elapsed since start of program. Used in stats calculations
};

Time timeElapsed (const Time startTime, const Time time1) {
	//keep track of time elapsed since start of program
}

void averageInclination (const Gyroscope gyro, float avgIncline[], Gyroscope& previousGyro) {
	//find average incline over set of data and time
}

void maximumIncline (const Gyroscope gyro, float& max) {
	//find max incline value over set of data
}

void minimumIncline (const Gyroscope gyro, float& min) {
	//find min incline value over set of data
}

float totalTime (float avgIncline[]) {
	 //find total time elapsed
```

```
            writeToFile(firstWord, secondWord); //write to log file
}

float calcAvg (int avgIncline, float totalTime) {
        //find average inclination over period of time
        writeToFile(firstWord, secondWord); //write to log file
}

bool stringToFloat(const char input[], float& value) {
        //converts a string to float
         writeToFile(firstWord, secondWord);  //write to log file
}

int stringToInt(char input[]) {
        //converts a string of numbers to an int.
        writeToFile(firstWord, secondWord);  //write to log file
}

int numOfDigits (int x){
        // Tells us the number of digits of the integer. Used as helper function
        writeToFile(firstWord, secondWord);
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

int main () {
  float avgIncline[3]; //array initializiation that will be used for storing the total amount of time
                          inclined, declined, and flat
  for (int i = 0; i < 3; i++) { //set initial index values to 0
        avgIncline[i] = 0;
  }

  //create variables
  float max = 0;
  float min = 0;
  float totalT = 0;

  Gyroscope previousGyro = { //base value for initial gyroscope reading
        .yaw = 0,
        .pitch = 0,
        .roll = 0
```

```
};

int startSeconds = getSeconds(); //get the starting seconds

//file reading
time_t t = time(0);
struct tm * now = localtime( & t );
  char logname[] = "LOGFILE.txt";

int i = 0;

//initialize variables used for storing gpio I/O
int rq1, rv1, rq2, rq3, rq4, rv2, rv3, rv4;
int gpio1, gpio2, gpio3, gpio4;
gpio1 = 1;
        ...
gpio4 = 7;
 gpio_free(gpio1);
        ...
 gpio_free(gpio4);


// check if gpio is already exported
if ((rq1 = gpio_is_requested(gpio1)) < 0 || (rq2 = gpio_is_requested(gpio2)) < 0 || (rq3 =
gpio_is_requested(gpio3)) < 0 || (rq4 = gpio_is_requested(gpio4)) < 0)
{
        //log that gpio is already used
}
// export the gpio
if (!rq1) {
        printf("> exporting gpio\n");
        if ((rv1 = gpio_request(gpio1, NULL)) < 0 || (rv2 = gpio_request(gpio2, NULL)) < 0 || (rv3
        = gpio_request(gpio3, NULL)) < 0 || (rv4 = gpio_request(gpio4, NULL))< 0)
        {
                //gpio already requested!
        }
}

// set to input direction of gpio pins

if ((rv1 = gpio_direction_output(gpio1, 0)) < 0 || ((rv2 = gpio_direction_output(gpio2, 0)) < 0) ||
(rv3 = gpio_direction_output(gpio3, 0)) < 0 || (rv4 = gpio_direction_output(gpio4, 0))){
        perror("gpio_direction_input");
```

```
}

bool val = true;
int analoguesize = 20; //max line of serial input data
while (true) {
        Gyroscope gyr;
        FILE * bytes;
        bytes = fopen("/dev/ttyS1", "r"); //open serial port, set permissions to read only

        char line[analoguesize];

        char* read = fgets(line, analoguesize, bytes);
        char cpot[6];
                …       // create char array place-holders for various input data
        char cPitch[6];
        int tempIndex = 0;
        int pot = -1;
        for(int x = 0; x < 6; x++){
        cpot[x] = 0;
        …   // auto initalize every value to null so we dont have to manually add null terminator
        cPitch[x] = 0;
        }

        // if success then read is not null, else read also contains same val as line
        if(read){
        int i = 0;

        while(line[i]){
        if(line[i] == 'q'){ //pot value found
        int x = i;
        while(line[x] != 'Y' && line[x]){ //keep on adding chars to cpot until you reach 'Y'. extra
         error check in case you reach end of line
        x++;
        cpot[tempIndex] = line[x];
        tempIndex++;
        }
        pot = stringToInt(cpot);
        tempIndex = 0;
        // cout << "POT: " << pot;
        } else if(line[i] == 'Y'){ //repeat for each delimiter
        ...
        i++;
```

```
        if(pot != -1){ // we are actually reading true potentiometer value error check
        if(pot >500){ //turn right blinker on
        int setval1 = gpio_set_value(gpio1, 1);
        int setval2 = gpio_set_value(gpio2, 0);

} else if( pot > 0 && pot < 300){ //turn left blinker on
         int setval1 = gpio_set_value(gpio1, 0);
        int setval2 = gpio_set_value(gpio2, 1);
} else if(pot == 0){ //start lock procedure
                int lock = gpio_set_value(gpio3, 1);
                int setval1 = gpio_set_value(gpio1, 0);
                int setval2 = gpio_set_value(gpio2, 0);
} else { //turn all blinkers off
                 int setval1 = gpio_set_value(gpio1, 0);
                 int setval2 = gpio_set_value(gpio2, 0);
}

        }

      //stats calculations
      int tempSeconds = getSeconds();
      if (tempSeconds != startSeconds) {
      int currentTime = getSeconds();

      averageInclination(gyr, avgIncline, previousGyro);
      maximumIncline(gyr, max);
      minimumIncline(gyr, min);

      totalT = totalTime(avgIncline);

      float avIncline = calcAvg(avgIncline[1], totalT);
      float avDecline = calcAvg(avgIncline[2], totalT);
      float avFlat = calcAvg(avgIncline[0], totalT);

      int nameWidth = 25;
      int numWidth = 25;
      char separator = ' ';

      Cout << [TABLE FORMATTED GYROSCOPE DATA] << endl;

      previousGyro = gyr;
      startingTime = currentTime;
      startSeconds = tempSeconds;
```

```
            }
    }
        pot = 0;

        //do gpio stuff here

} else {
    //log error
}

    }

    // unexport the gpio
    gpio_free(gpio1);
            ...
    gpio_free(gpio4);

            return 0;
}
```

B. Full Version of Code

https://github.com/rkabani19/bikePlusPlus-hardware

C. Peer Contribution

Rohail Kabani:

- Mainly responsible for the software component
- Created stats functions
- Created the structs
- Co-programmed main function
- Assisted with reading of serial data
- Refactored code for more readability
- Responsible for setting up gyroscope, and arduino
- Co-assisted with the final wiring, and soldering

Azfaar Qureshi

- Responsible for majority of the GPIO input/output
- Set up motors, and helped create the mechanical proof-of-concept for the bike lock
- Co-programmed main function
- Setup file reading, and reading Arduino input

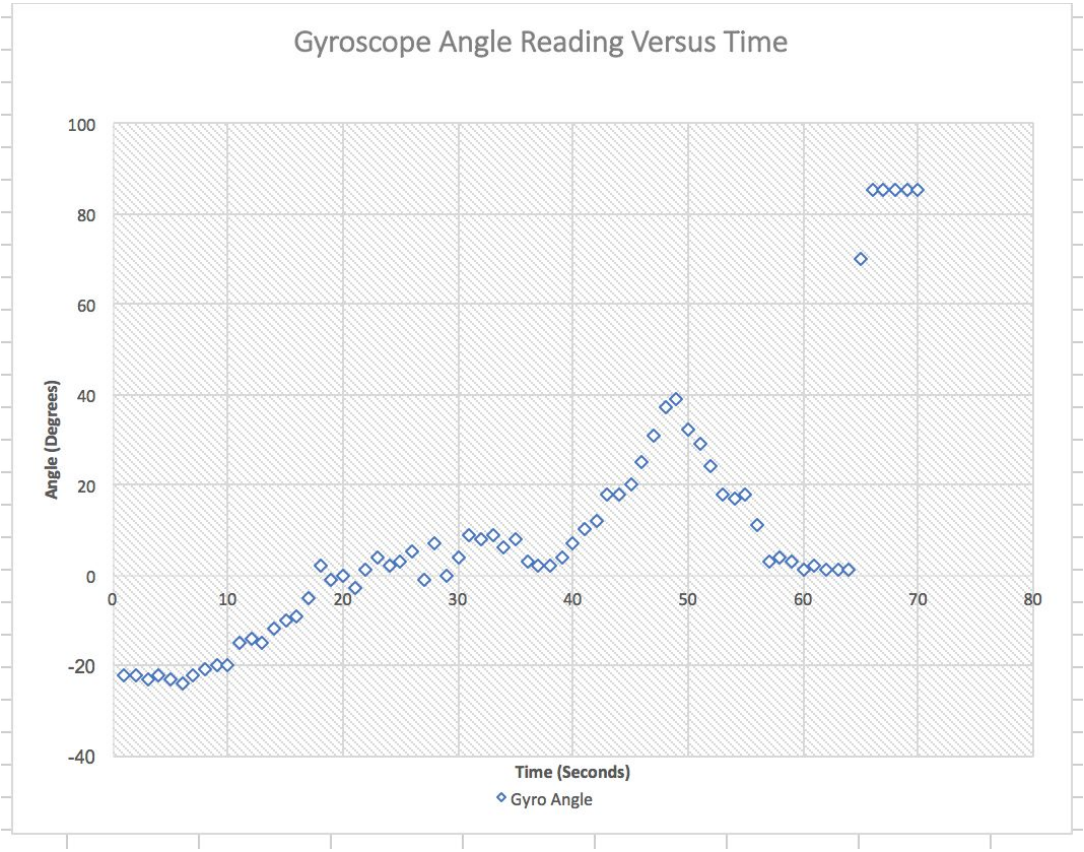
- Created blinker light animation
- Setup cross compiler
- Co-assisted with the final wiring, and soldering

Uzair Shafiq:

- Worked on helper functions software, such as the timeStamp maker, intToCharArray, digitToChar converter, and more. Also ensured that these functions worked properly and synchronously with the main functions that called them
- Created logging infrastructure for the program to help debugging
- Co-programmed main function
- Setup file writing function
- Wrote the project overview, system design, testing (for gyroscope), software design (system-independant and dependant components), logging infrastructure, limitations and lessons learned for the project report

D. Output Plot

	A	B	C	D	E	F	G	H	I
1	Measured Angle of Ramp			Gyroscope reading for angle of ramp					
2	~18°	(Stairs)		~17°			Since it is pointless to measure the angle		
3	~25°	(Small ramp)		~27°			using the gyroscope multiple times then		
4	~42°	(Large Ramp)		~45°			take the average, instead, the gyroscope		
5	~79°	(Top of the quarterpipe)		~77°			was allowed to stabalize in it's readings,		
6							then the reading was recorded.		
7		**Note**: The readings fluctuated a little over time. The gyroscope's							
8		consistancy is not very strong, but the accuracy of the readings							
9		when outliers are omitted is quite strong.							
10			**SideNote**: It is very possible that the gyroscope readings are more						
11			accurate than our measured readings using the protractor.						
12			**SideNote:** Sometimes our gyroscope read a little higher						
13			than expected, other times a little lower; Since the offset						
14			is not in the same direction (always higher or always lower),						
15			it is unlikely that there is a calibration fault.						
16			**SideNote:** The gyroscope measures angles to four significant						
17			digits, but the decimal values fluctuate slightly. So, the value was						
18			rounded and a tilda (~) was recorded to keep that in mind.						
19									



E. External Code/Libraries Used

https://github.com/mhei/libugpio/blob/master/src/ugpio.h
https://github.com/OnionIoT/i2c-exp-driver
https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050