Pedal-Powered Smart Bike

ECE 445 Final Paper

Alex Sirakides, Karl Kamar, and Anshul Desai
Team 29
TA: Yifan Chen
December 8th, 2020

Abstract

The following paper summarizes the motivations, conception, and processes that led to the realization of the Pedal-Powered Smart Bike project. In this document, we will cover each step of the project in detail by presenting graphs, calculations, and reasoning as well as commenting, in retrospect, on the challenges that were encountered along the way. As a result, this work will demonstrate the feasibility of this project and its possible industrial horizons if developed further.

Contents

1 Introduction 4
1.1 Objective 4
1.2 Functional Overview 4
2 Power Supply Module 5
2.1 Generation and Rotation System 5
2.2 Battery 8
2.3 Charge Controller 9
2.4 Buck Converter 10
3 Control Module 11
3.1 ATMEGA328P Microcontroller 11
3.2 Removal of MicroSD Card Reader 11
3.3 NEO-6M GPS Module 12
3.4 Statistic Functions 13
3.5 Ambient Light Sensor 14
4 Interface Module 15
4.1 Adafruit 2.7” 128x64 OLED Display 15
4.2 Reset Button 15
4.3 Turn Signals 16
5 Costs 17
6 Conclusions 18
6.1 Accomplishments 18
6.2 Ethical Consideration and Safety Hazards 18
6.3 Further Work 18
7 Citations 19
8 Requirement and Verification Table 20

1 Introduction

1.1 Objective

The objective of this project is to present a functional prototype for what we believe a modern-day bike should comport. In fact, as city populations grow faster and urban centers become denser [1], we must actively find solutions to ensure efficient and dignified ways for individuals to commute. Although personal vehicles and public transport are developed and massively used, the growing pressure on these two methods and their infrastructure is presenting cities with a new challenge. We thus believed that, looked over by the electrical and computer engineering industry, bicycles are one of the most efficient solutions to this issue [2].

In fact, in the past 100 years, bicycles have observed a fair change on the mechanical level. Lighter, smoother and more balanced bicycles have seen the light as different materials and more sophisticated gear and speed mechanisms were brought forth. However, the electrical and digital advancements that were exhaustively incorporated in the user interface and experience of automobiles has remained absent in bicycles. For bicycles to become increasingly used and successfully reduce traffic, they must become more attractive to the modern citizen. That can only be possible by making the bicycle a useful, self-sustaining, vehicle which offers the user a personalized experience by taking the modern automation and safety standards into account [3][4]. That is why we propose the pedal-powered smart bike as a solution to these problems. This bicycle is equipped with an automated lighting system, a liquid-crystal display (LCD) providing real time travel data to the user, and is self-powered by an incorporated generator and battery system.

1.2 Functional Overview

The pedal-powered smart bike's fundamental component is the generator. The generator is a brush motor, resting over the back wheel via a hinge system which is rotated by the bike through direct contact with the tire. The generator is connected to an 8Volt lead-acid battery which it pumps charge into. This connection is monitored and regulated by an intermediary charge controller to prevent battery degradation as well as reverse current which would revert our generator into a motor. This power generating system feeds the peripheral components of the user interface via a buck converter which steps the voltage down to 5 volts. These peripheral components include an ATMEGA328P-U microcontroller, an Adafruit Monochrome 1.3" 128x64 Organic light emitting diode (OLED) LCD, and a set of different light emitting diodes (LED) lights that serve as head, tail and turn signal light signals.

These components interact with the microcontroller and other control components to provide the user with a functional and useful interface. The head and tail lights are connected to both the microcontroller and an ambient light sensor. This sensor continuously provides surrounding luminosity data to the microcontroller which in turn is coded to respond by outputting an ON or OFF instruction to the head and tail lights. The turn signal lights are controlled by push buttons placed at the user's fingertips which are pressed to trigger a flashing behavior of the corresponding light. The LCD, in constant communication with a Neo 6M GPS module via the microcontroller, presents continuously updated personal information to the user including instantaneous speed, average trip speed, local time, trip time elapsed, heading, longitude and latitude. This information is relevant to each trip. The user can reset older data to record data for a new trip by pressing a reset button also placed at their thumb's reach. The whole system can be switched OFF or ON by a button placed at the center of the bike's frame which disconnects the battery from the charge controller and from the rest of the circuit. The programmed clock frequency for the code is 1Hz in accordance with the GPS module's internal clock frequency. The ATMEGA328P-U's 2kbyte SRAM and 1kbyte EEPROM was sufficient to contain the coding

and library information which was needed. This led us to the decision of dropping the external microSD reader which was supposed to store eventual excess memory data. This will be developed further in the chapters relative to the microcontroller and the GPS module.

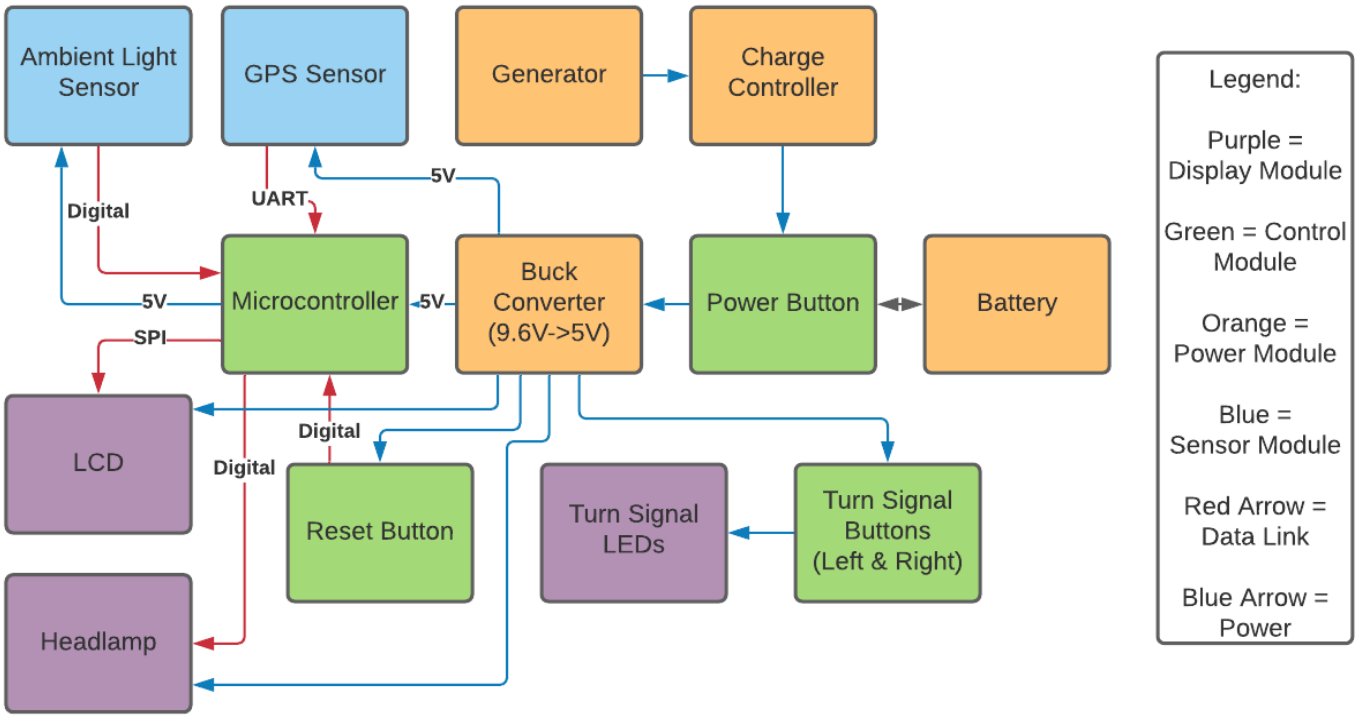

Figure 1. Functional Block Diagram of the Smart Bike's Final Design as Demonstrated

2 Power Supply Module

2.1 Generator and Rotation System

The generator is arguably the cornerstone of our project. It provides the bike with the self-sufficiency feature which allows it to never run out of charge. The generator chosen for this project is a direct current (DC) Brush Motor because these are the simplest and most straightforward to use. In fact, the generator was chosen over an alternator so as to avoid dealing with an alternating current (AC) output and voltage rectification since our bike's other components are only DC compatible.

Note: when referring to the component, we talk about the brush motor we purchased, however, once integrated into the system, the motor functions as a generator and will be referred to as such.

After choosing the 24 VDC Brush Motor, the next question to consider was the location of the generator on the bicycle. The place of the generator depends on the type of rotation system that would be used. Initially, a chain system was considered which would be attached to the bicycle's gear mechanism. That way the generator would be turned directly by the user's pedaling motion as the cogs turn. The generator would thus be placed at the front of the bike and ideally at the same level as the gear in order for most of the force to go into torque on the brush instead of pushing the generator downwards. After reflection over this question, and conversation with the machine shop technicians, we concluded that this setup might lead to excessive torque exerted on the generator's brush. Consequently, the generator would react by an increasing drag which could make it almost

impossible for the user to pedal the bike. This was further theoretically verified by using the basic equation for torque:

Torque = Force x Distance from Object x sin(θ) (2.1)

where θ is the angle between the force vector and the plane that it is exerted on. We will compare the result of this equation in the two proposed scenarios in a further part of this chapter.

As an alternative, we decided to design the system by using a hinge system. The generator would be attached to a metal rod which itself would be connected to the bike's rear frame by a hinge mechanism. The generator would then rest over the back tire of the bicycle with its brush as the contact point. By designing this mechanism, we would be relying on the friction between the brush and the tire to stimulate the rotation motion of the brush. The friction in this scenario would be a function of the normal reaction from the tire on the brush which is nothing else than the opposite and equal force of the generator's weight. This is proven by the friction equation as a function of normal reaction:

Friction = μk x Normal force (2.2)

where μk is the kinetic constant of friction between rubber and metal found from online sources[5].
In order to maximize the friction force and make it as constant and stable a value as possible, we added an indented steel cylinder at the place of the generator's brush. This cylinder being longer and thicker than the brush, it has a stabler and bigger contact area with the tire. This made the rotation motion smoother and more consistent as well. This friction force would then be the acting force used to calculate the torque using equation (2.1). Finally, to know the exact voltage output of at our generator terminal, we would use the values given in the generator's performance chart in Figure 2.1as a function of the exerted torque on the brush as follows:

Voltage = Power/Current x Efficiency (2.3)

We will now present our calculations and explain how our results compare and lead to our design choice.

Using equation (2.1) to calculate the average torque exerted on a bicycle gear, we use the following values for each component:

Force = Average Person Weight = 65kg x 9.81m.s^-2 = 637.7 N

Distance from Object = Average Pedal Size = 0.17 m

|sin(θ)| = 1 at maximum torque points and 0 at minimum torque point

Therefore,

Torque = 637.7 x 0.17 x 1 = 108.4 N-m at maximum point.

This torque value exerted at the base of the bike's pedal would then be translated into the chain's torque on the generator brush with some loss. However, this value is out of range with respect to our generator's torque tolerance which has a rapidly decreasing efficiency starting at 1.1N-m torque values. This means that this torque value is two orders of magnitude larger than the maximum recommended torque on our performance chart which would lead to excessive drag from the generator.
Calculating the torque exerted on the generator brush in the hinge system scenario can be done by using equation (2.2) and plugging its result into (2.1) as follows:

Normal Reaction = Generator Mass x g = 5.6lb x 0.453592kg/lb x 9.81 = 24.92 N

Friction = Dynamic Friction Constant x Normal Reaction = 0.5 x 24.92 = 12.46

Torque = Friction x sin(θ) x Generator Brush Length
= 12.46 x sin(90) x 1.1in x 0.0254in/m
= 0.348 N-m

When plugging this result into the generator performance chart visible in Figure 2.1, we observe that this torque yields the following electrical output values from the generator:

Power = 124W, **Current** = 6.6A, **Revolution Rate** = 3320 RPM and **Efficiency** = 76%

Now, plugging these values in equation (2.3), we obtain the following voltage output:

Voltage = 124/6.6 x 76/100 = 14.28V

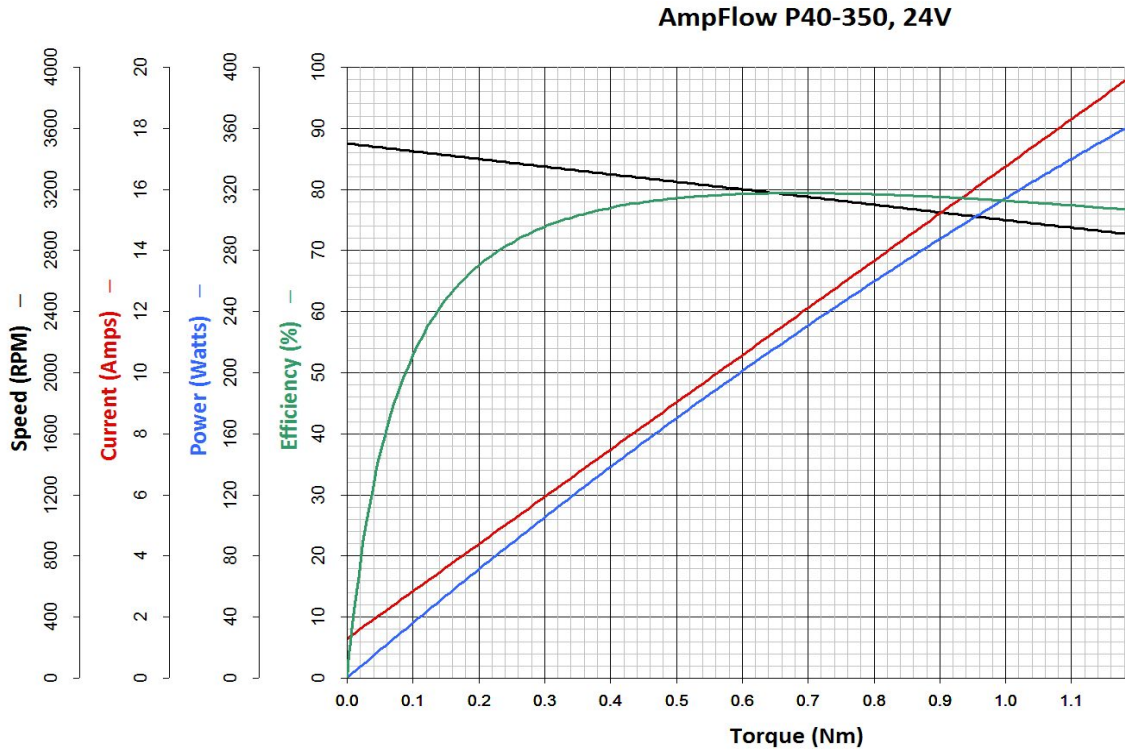

Figure 2.1. Performance Chart of the 24VDC Motor Used in the Project [6]

The verification process for the generator's functionality and requirements was fairly straightforward. After the design was installed and finalized at the machine shop, one of the team members, Alex, rode the bike with the generator system installed back home. He was able to bike fairly easily and only experienced minor drag resistance which is desirable for the exercise purpose of the bicycle. By this, we can verify that the torque does not exceed 5N-m as specified in our design document Requirements and Verification table for the Generator available in the Appendix . The second requirement that was to be verified is the minimum voltage output of the motor which was set as 9V. This value is based on the minimum charging voltage requirements of our 8V battery which will be developed in the corresponding following chapter. The battery being unable to charge at applied voltages inferior to 9Volts, this requirement was verified by observing the battery's terminal voltage displayed on the charge controller's seven segment display. In fact, as we pedalled the bike with our arms and holding the bicycle stationary and lifted above ground level contact, we could observe an increase in the battery's terminal voltage on the charge controller display proving that our generator's output was at least the minimum required charging voltage that the battery needs. Since this verification ended involving two

components that we have not presented yet, we will not explain this verification process further as it will be clarified in the following chapters.

2.2 Battery

The battery is the direct power source for the bike's peripheral components. Although the generator is the one that generates power to maintain the battery at a healthy charge level, the battery is used to power the rest of the circuit and that for multiple reasons.

The first reason which made us decide to use a battery is power consistency. This is for the case that the cyclist is stopped at a red light or on the side of the road. The generator would stop generating power and thus the whole circuit would turn off. This could also happen if at a moment the bicycle were to be slower than the needed speed to exert the minimal required torque on the generator brush. The consequence would be constant pausing and playing of the information collection from the GPS module and an inconsistency in displayed data. The second reason is the time required to power-up the circuit. If we were to solely rely on the generator's generated output, we would need a couple seconds for the circuit to power up. That adds to the fact that our GPS module needs some initial idle time to calibrate itself and get an accurate location and time fix which will be further developed

The battery value was decided based on the following factors: Voltage, Current, Weight. Initially, we had chosen a 12V 2.3Ah lead acid battery. This voltage value was sure to be enough for our needs, however, it weighed 4.2lbs. We thus decided to use a 8V 3.2Ah battery which weighed 2lbs. This is because our head and tail light specificities, being our main power consumers, don't need more than 5V to be powered but around 650mA of current need. We therefore opted for the 8V 3.2Ah which was safer current wise since all our components are powered in parallel at the battery output. This could be easily verified using Kirchhoff's Current Law with the following equation:

$$\Sigma \text{ Input Currents} = \Sigma \text{ Output Currents} \qquad (2.3)$$

In order to theoretically verify this fact, we placed the max current draws of our lights in the equation (2.3). To see if it would be smaller or equal to the maximum current draw of the battery of 3.2Ah. Our values are:

Headlight and Taillight Max Currents = 650mAh each
Turn Light Signal Max Currents = 50mAh each
OLED Adafruit Display Max Current = 150mAh
NEO-6M GPS Module Max Current = 67mAh
ATMega328P-U Max Current (16MHz) = 14mAh

We therefore obtain:

$$\Sigma \text{ Output Currents} = 650x2 + 50x2 + 150 + 67 + 14 = 1{,}481\text{mAh} = 1.48\text{Ah} < 3.2\text{Ah}$$

This is the first theoretical proof by calculations that our max current draw is inferior to half the max current draw of our battery [7]. Which makes our battery choice very adequate. This was proven when implementing the whole circuit with the battery. We thus satisfy our Requirement and Verification table for the battery as presented in our Design Document and available in the Appendix.

2.3 Charge Controller

The Charge Controller is necessary to ensure safe charging of the battery. Since the generator turns depending on the pedaling rate, it is very inconsistent which can lead to irregular spikes in the voltage output. Another concern is that the output of our generator is much greater than the max charging voltage of our battery (9.6V) [7]. The charge controller therefore makes sure the battery is not damaged by excessive voltage or current.

Our first plan for the charge controller was to build it ourselves with a chip and a whole circuit on a PCB board. We chose to use the charge controller chip LTC4020. This chip's datasheet [8] laid out the detailed calculations needed to be done to choose the right resistor values needed for the voltage and current dividing circuits which would be connected to the chip's different outputs. According to our battery data sheet, the max charging current of the battery must be 0.96A with the voltage value given above. To obtain these values as outputs of our charge controller circuit, a couple resistor values were calculated. Rcs, the charge current sensor, is defined in the data sheet as:

$$\mathbf{Rcs = 0.05/Ics_{MAX} = 0.05/0.96 = 0.0521\Omega} \quad (2.4)$$

This value limits the maximum output current to 0.96A as defined in the battery data sheet.
The next output values to be controlled are $V_{Float,}$ $Vout_{MAX}$, and $Vout_{MIN}$. V_{Float} the voltages which must be maintained at the battery terminals to prevent it from discharging. That is because our charge controlling chip was designed to charge a battery and then maintain it at a constant
charge instead of letting it discharge and recharge it. The equation to find the resistor ratio for our specific battery is as follows:

$$\mathbf{V_{Float} = 2.5(1 + R_{FB1}/R_{FB2})} \quad (2.5)$$

where V_{Float} = 9V, and R_{FB1} and R_{FB2} are connected to pins V_{BAT} and V_{FB} of the charge controller IC as shown in Figure 3.
This yields a ratio R_{FB1}/R_{FB2} = 2.65.
To calculate $Vout_{MAX}$ and $Vout_{MIN}$, a similar equation must be used to get the ratio of the resistances R_{MAX1}/R_{MAX2} and R_{MIN1}/R_{MIN2}. That is:

$$\mathbf{Vout_{MAX} = 2.75(1 + R_{MAX1}/R_{MAX2})} \quad (2.6)$$
$$\mathbf{Vout_{MIN} = 2.75(1 + R_{MIN1}/R_{MIN2})} \quad (2.7)$$

where R_{MAX1} and R_{MAX2} and R_{MIN1} and R_{MIN2} are connected to pins V_{FBMAX} and V_{FBMIN} of the chip respectively as shown in Figure 2.3 below.
Since the components of this board were all of QFN type, (Resistors of multiple kΩ, constant inductor of 15µH [8], and the chip itself, it made the process of soldering out of our reach since we don't have the necessary tools for that. A lot of the components were also very expensive due to their rare values (over $50 for some single components).We therefore resorted to ordering a prefabricated charge controller with adjustable settings based on Start and Stop Voltage values of the battery output being:

$$\mathbf{Start\ Voltage = Rated\ Voltage\ x\ 0.9 = 8\ x\ 0.9 = 7.2V} \quad (2.8)$$
$$\mathbf{Stop\ Voltage = Rated\ Voltage\ x\ 1.2 = 8\ x\ 1.2 = 9.6V} \quad (2.9)$$
$$\mathbf{Charging\ Current = 0.1\ x\ Rated\ Current = 3.2\ x\ 0.1 = 0.32A} \quad (2.10)$$

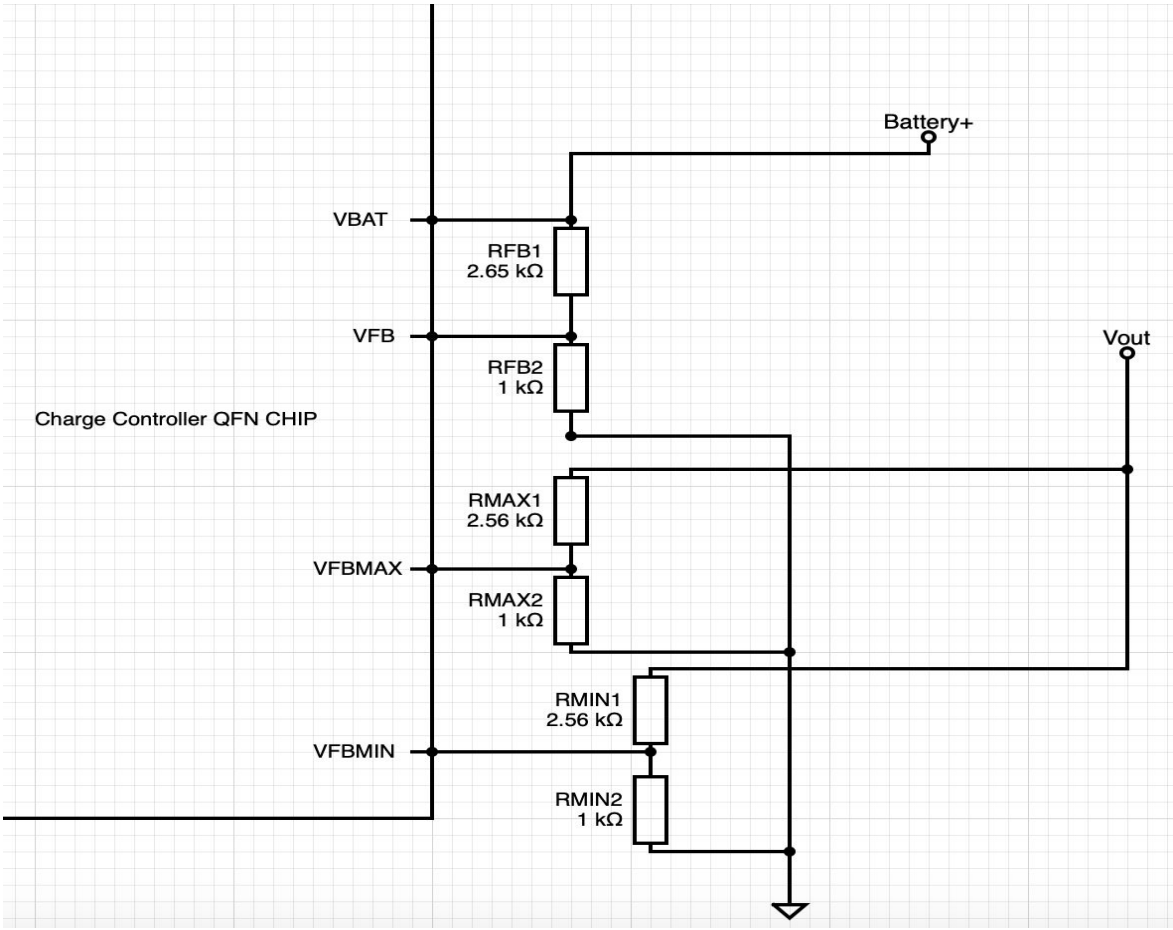

Figure 2.3. Schematic of the Resistances as Connected to the Pins of the Charge Controller QFN Chip

2.4 Buck Converter

The buck converter is the last component of the power module. It is directly connected to the VCC pin of the microcontroller and the positive ends of the other components (LCD Display, Lights, Buttons). Its use is to step the 8V voltage of the battery down to 5V. 5V being the voltage value at which all our components operate on. The only component that needs 3.3V is the ALS which gets its power from the microcontroller instead. The buck converter we have chosen also maintains the current below 3A which is an adequate value with respect to our 3.2Ah current battery. We had initially considered using a linear voltage regulator because it would have a more constant output. However, the downside of the voltage regulator is the excessive loss that is dissipated as heat. Not having the time or the money nor the space on the bike to install an efficient enough heat sink, we decided to opt for the cheaper and more straightforward buck converter.

This final piece of the module ensures a safe voltage value applied at the component terminals preventing damage as was verified in the final implementation.

3 Control Module

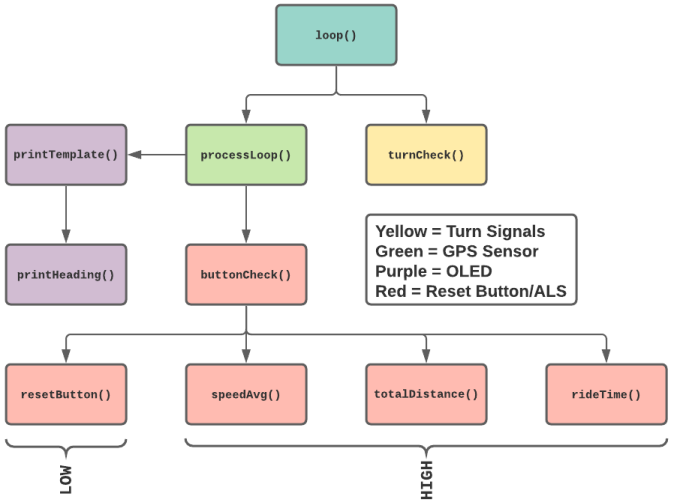

Figure 3.1. Control/Interface Module Function Tree

Figure 3.1 shows us the overall function tree of our microcontroller as it applies to both our Control and Interface Module. It is important to note, and will be further elaborated, that all functions but one operate on the update clock of the GPS sensor (`processLoop()`), rather than the clock of the microcontroller (`loop()`). Functions from this figure will also be referenced in the following module.

3.1 ATMEGA328P Microcontroller

The development process began with the selection of a microcontroller. The ATMEGA328P was selected as our microcontroller as it is the same microcontroller used by the Arduino Uno, enabling us to program, debug, and upload our sketch to our microcontroller directly from an Arduino. Our development process involved bootloading a number of naked ATMEGA328P microcontrollers, and rather than using a USBASP device as initially planned, we simply placed these microcontrollers onto our Arduino board, and uploaded our code via Arduino IDE. While most testing occurred on the Arduino board itself, when it came to finally testing and integrating with our entire system, we simply removed the microcontroller from our Arduino board, placed it onto a breadboard with two 22 pF capacitors, with a 16 MHz self-oscillating crystal which is utilized as an external clock by our microcontroller. A simple translation from Arduino Uno pins to ATMEGA328P pin was ensured (and often created problems when incorrect) in order to validate that the code interfaces with the external sensors, lights, and buttons correctly.

3.2 Removal of MicroSD Card Reader

The idea of implementing a MicroSD card reader stemmed from the idea of storing user path data with intentions of potentially allowing the user to retrieve this data on their personal devices to see a graphical representation of their journey as well as potential reverse geocoding via a large peer-submitted latitude-longitude to location database. It was quickly observed that while testing only the LCD, GPS sensor, and MicroSD card reader, SD card writes would only work synchronously instead of on interrupt and the delay against our 1 Hz clock update from our GPS sensor would cause a disruption of serial GPS data resulting in garbage values being processed by our microcontroller. This was verified by seeing randomized values being reflected on our LCD during early testing. Additionally, the microcontroller was simply too slow to process

reads from the SD card while having to write our display buffer and read data from the GPS sensor. While we found the need for path extraction and reverse geocoding to be ambitious and unnecessary, we later found that the microcontroller contained a 1 KB EEPROM which would be able to serve our needs as they relate to statistical calculations as well as other features which will be explained later. The unused pins of the card reader on our PCB were eventually repurposed for buttons, ambient light sensor, and lights which voided the need to redesign or create a new PCB.

3.3 NEO-6M GPS Module

The NEO-6M GPS module is one of the most widely used GPS sensors and only requires four pins, only of which two occupy digital inputs of the microcontroller. The interface with our microcontroller occurs through UART. In our PCB, we have the RX and TX pins of the GPS sensor mapped to the RX (2) and TX (3) pins of the microcontroller. It is important that we map RX to TX and TX to RX in order for correct interface. As the Arduino requires the use of the RX and TX pins to be emptied when uploading a sketch, during development we had leveraged a library called NeoSWSerial to allocate any pin I preferred when developing. The use of the NeoGPS library allowed for the fastest processing of NMEA GPS data. NeoGPS's included functions allow for much more accurate metrics with the inclusion of error margins to correct data. However, before even leveraging either of these libraries, we needed to verify the GPS sensor was working with our microcontroller. By checking against the default baud rate of 9600, we were able to use the serial monitor of Arduino IDE to display a variety of NMEA sentences. The NMEA sentence containing the most useful data as it related to our project was the $GPRMC sentence:

```
$GPRMC,220516,A,5133.82,N,00042.24,W,173.8,231.8,130694,004.2,W*70
```

Figure 3.2. Example of $GPRMC sentence

Looking at this sentence, we can extract data as it relates to location, heading, and even speed. For example, fields 3 and 4 describe our latitude and cardinal direction while field 7 describes our speed in knots. While this data is not that attractive nor digestible, NeoGPS allows us to reference different values as whatever datatype we pleace.

The GPS sensor is responsible for the main control loop of our system. `processLoop()` is the name of the function which is called dependent on the GPS sensor's software serial connection's availability which occurs at a frequency of 1 Hz. Since this function calls all of our other important functions, much of the control logic of the device is dependent on the time fix of the GPS. Fortunately, it takes little to no time for the GPS sensor to achieve this fix when outside. The pseudocode of our function is as follows:

```
void processLoop() { // Called when software serial connection for GPS is available (1 Hz)
   1. Save reset button state, ALS state, and GPS fix
   2. Call buttonCheck() THEN printTemplate()
   3. Save updated statistics (and associated variables) to EEPROM
}
```

Figure 3.3. processLoop() pseudocode

3.4 Statistic Functions

The three statistics calculated while our device is recording relates to average speed, ride time, and total distance traveled and are calculated when recording and on each 1 Hz update of the GPS sensor. This is because the functions are called by `buttonCheck()` which is called by `processLoop()`. Each function leverages the EEPROM which replaced our MicroSD card reader as discussed earlier. This also allows for our data to be saved on each update such that if the device loses power, we can still retain our statistics (as indicated by Step 3 in Figure 3.3). Other use cases involve factoring black space (time in which the device is powered off) out of our statistics and only based on type of current fix. Beginning with our `avgSpeed()` function:

```
void speedAvg() {
   1. If the current speed is more than 0.5 mph
   2. Then avgSpeed = ((prevCount * avgSpeed + [current speed]) / (prevCount + 1))...
   3. ...and increment prevCount
}
```

Figure 3.4. speedAvg() pseudocode

Here we see that our average speed is calculated merely as a running average of ongoing speeds. We ensure that we only calculate for speeds above half a mile per hour because the GPS speed tends to wander around less than half a mile per hour when stationary. This is so that stationary speeds are not included in the average. The use of floating-point variables allows us to process this calculation for near infinite time. We retain both the previous count and average speed to our EEPROM because the previous count is what is used to average out our speed and only increments when our specific conditional is met. Next we have our `totalDistance()` function:

```
void totalDistance() {
   1. If the current speed is more than 1 mph, and current location is valid
   2. Then add the difference in distance from previous and current location to total...
   3. ...and update prevLoc to current location
}
```

Figure 3.5. totalDistance() pseudocode

Similar to the previous algorithm, we set up a condition such that distance only increases with speeds above 1 mile per hour which is likely when a user is biking. Rather than maintaining previous count, we maintain the previous location and leverage NeoGPS' included distance formula. The benefit of using this formula is that for small distances, it knows to calculate square distance, however for large distances, it will opt for Haversine accounting for the roundness of the planet. In this case, we do not need to save the previous location to the EEPROM, just the total distance. Next we have `rideTime()`:

```
void rideTime() {
   1. If (reInitTime) and time is valid
   2. Then save start time, previously elapsed time to EEPROM, set reInitTime flag on
   3. Else if !(reInitTime) and time is valid
   4. Then calculate time difference from saved start time
   5. Add time difference and elapsed time if necessary
}
```

Figure 3.6. rideTime() pseudocode

Here we see a slightly more complex implementation. The use of reInitTime as a state identifier and forcing this state to true on startup ensures that we have a base timestamp for comparison. By not saving this

timestamp to the EEPROM, we remove any black space (time when device is off) from our calculation, since the base time will be overwritten on every startup. We instead save the previous time to the EEPROM as elapsed time which is factored into the time difference when the function reaches the next state. This occurs the first second after the base time is set.

3.5 Ambient Light Sensor

The ambient light sensor (ALS) is used to determine the behavior of the head and tail light. The ALS reads HIGH in low light and LOW in day time. On the other hand, our head and tail light operates in a five-state FSM, two of which being ON and OFF and three belonging to external modes not required by our needs. The states are such that on power up, the head and tail lights start in the ON state, then have three modes deemed unnecessary to our project, followed by the OFF state. To cycle through these states electronically, the buttons of the head and tail lights are pulled up by a 2M ohm resistor, effectively tying the button to high-z, and then pulled LOW in 5ms pulses by our microcontroller. Given the three unnecessary states, the microcontroller would have to pulse LOW four times to cycle around to the OFF state and then pulse LOW a single time to cycle to the ON state. Determinations for the output of the microcontroller were made based on the state of the ALS.

Recall that we read our ALS state in Figure 3.3, Step 1. We process our current and previous ALS state within our `buttonCheck()` function (1 Hz). The following pseudocode follows the modified state system described above:

```
void buttonCheck() {
      ...Reset Button Behavior…
   1. If ALScurr is HIGH, ALSprev is LOW, and !(alsInit): Pulse LOW once // Light now on
   2. Else if ALScurr is LOW, ALSprev is HIGH: Save initTime and set alsInit flag on
   3. Else if ALScurr is LOW, ALSprev is LOW, 5 or more seconds since initTime, and
      (alsInit): Pulse LOW four times and set alsInit flag off // Light now off
}
```

Figure 3.7. buttonCheck() pseudocode for ALS-FET behavior

As shown above, we make use of variables such as `ALSprev` and `alsInit` to capture the previous state of our ambient light sensor and utilize a flag to enforce a strict flow of states from 1 to 2 to 3 and then back to 1. As you can see, we check to see if five seconds have elapsed since our saved start time which requires that we utilize GPS data. This means that this behavior will be dependent on the `processLoop()` function and the ability for our GPS sensor to achieve a time fix at the minimum. However, this requirement is merely required for states 2 and 3, which enables the light to be turned on in darkness without needing a GPS fix of any kind. The only drawback is that the light can be briefly left on in daytime without a GPS fix. On the other hand, the tradeoff in favor of low light safety far outweighs the negligible power consumption that may occur for a short period of time. We also chose a time difference of five seconds to prevent potentially annoying on and off flickering in very specific conditions. This occurs as we transition from darkness to light and need to power off our lights which also promotes potential safety.

4 Interface Module

4.1 Adafruit 2.7” 128x64 OLED Display

When we originally planned to integrate our MicroSD card reader with our final circuit, we had connected it through hardware SPI in order to ensure the fastest write and read speeds. Fortunately, the Adafruit Display Driver for our SSD1325 display allowed us to connect our display through software SPI at half the draw speed. Luckily, since we did not require any frame-by-frame based animation, and only an refresh rate of 1 Hz, software SPI was plenty fast enough to meet our high-level requirements, regardless of the fact that we removed the MicroSD Card Reader. A requirement for our display was an SRAM of at least 1 KB from our microcontroller. The ATMEGA328P is equipped with around 2 KB SRAM, leaving 1 KB leftover for anything that might be needed. We were able to verify our overwritten pin assignments were working with our microcontroller by using provided test code from Adafruit. When we were able to see that the display worked with our PCB conditions, we got to work developing a template for our user display. The main function called by our OLED display is `printTemplate()` and its associated helper function `printHeading(float heading)` to correctly convert our heading from degrees to compass cardinality for our display. The pseudocode is as follows:

```
void printTemplate() {
   1. Clear display, set up cursor and styling for video buffer build
   2. Call printHeading(fix.heading()) and print heading and time at top row
   3. Print speed and average speed from calculated buttonCheck()
   4. Print distance from calculated buttonCheck()
   5. Print ride time from calculated buttonCheck()
   6. If location fix, print coordinates
   7. Display video buffer
}
```

Figure 4.1. printTemplate() pseudocode

4.2 Reset Button

As indicated by Step 1 of Figure 3.3, we save the state of our reset button at the beginning of our `processLoop()` function. Although the button is called a “reset” button, it’s a bit of a misnomer as the functionality of the button is more like that of a “record” button. We check and manage the state of our reset button through the `buttonCheck()` function which is called on GPS update (1 Hz). If HIGH (button-switch is depressed), we call all three of our statistical functions to update our statistical variables stored in flash memory so that by the time `printTemplate()` is called, it will print updated statistics. On the other hand, if LOW (button-switch is floating/un-depressed), we call our `resetButton()` function continuously whose pseudocode is as follows:

```
void resetButton() {
   1. Set average speed, ride time, trip distance, and prevCount/prevTime to 0
   2. Update start time for rideTime to current time (continuously)
   3. Update EEPROM with new values
}
```

Figure 4.2. resetButton() pseudocode

In order to ensure proper values were taken in, the reset pin is tied LOW via a 10k ohm pull-down resistor.

4.3 Turn Signals

The button-switches for our turn signals interface with our microcontroller through the unused pins of the MicroSD Card reader. We utilize two inputs from our buttons and produce our "flashing" behavior through a signal digital output. The turn signal function, `turnCheck()` is the only function that operates outside of the `processLoop()` function and rather through the microcontroller's 16 MHz crystal. This makes the functionality completely independent of the 1 Hz clock for the GPS sensor. This is required since the outputting flashing behavior is a 2 Hz square wave duty cycle. The following describes the pseudocode to `turnCheck()`:

```
void turnCheck() {
   1. Read states from turn signal buttons and output pin
   2. If either state is HIGH
   3. Then get current clock time and if 0.5s has passed since previous time, set flip
      output pin
}
```

Figure 4.3. turnCheck() pseudocode

The main reason we are able to implement this function asynchronously from `processLoop()` is because the code is simple enough where we can implement without delay and we also are not dealing with a state machine like we did with our ALS-FET system.

The implementation of the turn signals was slightly more complex than that of the head and tail lights. Like our head and tail lights, the turn signals were finite state machines, however, unlike our head and tail lights, the button on the turn signals could not be electronically controlled via a single pin. The reason for this is not particularly pertinent to the project but it did leave us with turn signals which effectively operated in two states; powered / ON and unpowered / OFF. To achieve the 1 Hz flashing indicated in the high level requirements, it was necessary to drive the LEDs via a FET. By utilizing an n-channel mosfet as the common ground of the two LEDs, it was possible to flash the LEDs using a square wave output from the microcontroller. From here, the power source for the LEDs was severed mechanically by the turn signal buttons and only provided power to the proper LED when the proper turn signal's button was depressed. This circuit can be seen in figure 4.4 below.

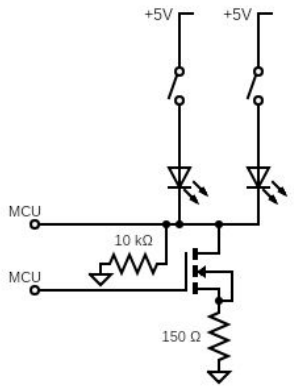

Figure 4.4. Turn Signal Circuit Design

5 Costs

The costs for this project are an addition of labor costs and material costs.
The labor costs for each of the teammates is as follows:

ideal salary (hourly rate) x actual hours spent x 2.5 (5.1)

where the 2.5 factor is to account for the other labor costs such as machine shop technicians and lab janitors.
Plugging values in equation (5.1) yields the following result:

35$/hr x 8hrs x 13wk x 2.5 x 3 = 27,300$

As for the component prices and other material, here are the costs in below Table 1:

Component	Retail Price	Department Paid	Team Paid
Charge Controller	$11.98	$0	$11.98
PCB Boards	$24	$24	$0
Microcontroller	$15.05	$0	$15.05
Ambient Light Sensor	$5.88	$0	$5.88
Neo 6M GPS Module	$26	$15	$11
Motor	$81.95	$81.95	$0
Battery	$31.90	$31.90	$0.00
QFN Charge Controller	$26	$0	$26
Buttons/Buck Converters/LEDs	$45	$0	$45
LCD Display	$49.95	$49.95	$0
Micro SD Card	$15	$15	$0
SD Card Reader	$7.5	$7.5	$0
TOTAL	**$268.23**	**$210.30**	**$57.93**

Table 1. Costs of Components and Other Materials and Portions Paid by the Department and the Team

6 Conclusions

6.1 Accomplishments

Our project exceeded expectations both in terms of power generated and the capabilities of our periphery components. We successfully integrated a brush motor as a generator within our system requiring very minimal torque input from the user and reaching near optimal performance per the specifications of the brush motors design specifications. This resulted in a system which is entirely self-sufficient, easy to use, and significantly upgrades the user experience of biking relative to anything on the market at this time. We were able to accomplish these goals utilizing only off the shelf components and modifications to the frame via the machine shop.

6.2 Ethical Consideration and Safety Hazards

Within our project, there are several safety hazards which needed to be addressed. First and foremost, the concern for loose wires for the user to unintentionally come into contact with. To address this issue, we made sure that all wire connections were covered via a piece of shrink-tube or electrical tape. All leads on through-hole board, PCBs, and other components were beyond the users range of access and, given the project is meant for outdoor use, final iterations of this project would require that all electronics be contained within a waterproof casing and thus inaccessible to the user. The other main point of concern for the user would be the potential for reverse current to the brush motor causing the generator to act as a motor and pedal the bike without user input. This is prevented by a set of diodes in parallel between the generator and the charge controller. In regards to the ethics of our design, there are no ethical considerations of note which pertain to our project.

6.3 Further Work

Further work for this project is quite expansive as there is still a lot of potential for both upgrades, optimization, and ease of use which could all be considered. The most important items to address would be: the need to complete weatherproof housings for the components, as this project is meant to be outside; the need to optimize the design by making the wiring, PCB, and lighting more permanent so that a user may unbox the product and easily install it on their own bike; and finally to make adjustments to the UI to include temperature, directions, and other useful information. We expect all of this would take this project from a fun idea into a functional product.

7 Citations

[1] un.org, Department of Economics and Social Affair, News, “Around 2.5 billion more people will be living in cities by 2050, projects new UN report”, 16, May 2018. [Online]. Available: https://www.un.org/development/desa/en/news/population/2018-world-urbanization-prospects.html

[2] Bicycling, “New Study Says Bicycles Are the Future of Urban Transportation”, Jan 15, 2020. [Online]. Available: https://www.bicycling.com/news/a30518994/deloitte-2020-study-bicycle-transportation/ https://www2.deloitte.com/content/dam/insights/us/articles/722835_tmt-predictions-2020/DI_TMT-Prediction-2020.pdf

[3] NHTSA, Bicycle Safety, “Overview”. [Online]. Available: https://www.nhtsa.gov/road-safety/bicycle-safety#:~:text=In%202018%2C%20there%20were%20857,fatalities%20on%20our%20nation%27s%20roadways.

[4] IIHS HLDI, Fatality Facts 2018, Bicyclists, December, 2019. [Online]. Available: https://www.iihs.org/topics/fatality-statistics/detail/bicyclists

[5] Tribology-ABC, “Coefficient of friction for a range of material combinations”.[Online]. Available: https://www.tribology-abc.com/abc/cof.htm

[6] AmpFlow, “P40-350, 24V Chart”. [Online]. Available: https://www.ampflow.com/P40-350_Chart.gif

[7] Battery Mart, “PS-832 8Volt 3.2Ah Rechargeable Lead Acid Battery.” [Online]. Available: https://www.batterymart.com/pdfs/sla-ps-832.pdf

[8] “LTC4020 55V Buck-Boost Multi-Chemistry Battery Charger”, Linear Technology.[Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/4020fd.pdf

Appendix - Requirement and Verification Table

Power Module

Requirement	Requirement
Pedals need to turn with < 5N-m of torque.	This can be verified by inspection.
Generator must generate 9 volts of power.	Prop up the back tire so that the bike may be tested in place with a voltmeter wired in parallel between it and the charge controller. 9V is the target voltage.
Prevents the battery from overcharging	Verifiable by tick from the relay. The display instantly shows a reduced voltage indicating battery disconnected.
Needs to have the capacity to power up the circuit.	System powers up when the on switch is flipped, while battery is charged.
Steps down 8V battery input to 5V output for the rest of the system.	Use multimeters to test input and output voltages of the buck converter.

GPS Sensor Module

MCU does not interface with garbage metrics from GPS Module	Ensure standing still doesn't change total distance, average speed, or the displayed coordinates on the LCD.

Display Module

Display updates at rate of 1 Hz.	Verified by update of the time on the display in seconds.
Display shows stated information when GPS has a fix.	Verified by inspection.

Control Module

Reset switch must reset data when un-depressed and record data when depressed.	Test by inspection. Unit testing performed on Arduino IDE.
Headlight and taillight turn on when the ambient light sensor detects low visibility.	Turn off lights and check whether the bike lights come on.
Headlight and taillight turn off after 30 seconds in bright conditions.	Turn lights on and check whether the bike lights turn off.
Average speed only calculated for speeds > 0.5 mph and total distance only calculated for speed > 1.0 mph	Conditionals included in each relevant function to check current speed and ensure that it is within bounds. Put in place due to wandering behavior of GPS module. Unit tested on Arduino IDE.