The Smart Bike

MECH 307 - Group 32

December 9, 2016

Design Team

Nate Keisling

Chris Sawyer

Sean Visocky

Rob Bescheinen-Hajdas

Design Summary

The idea behind this project was to create the ultimate commuter bicycle, a bicycle which helps the rider safely, effectively, and efficiently flow with traffic. The system includes a LIDAR module to detect cars behind the rider, a smart LED headlight and taillight, and an 800 Watt electric motor driving a freewheel crank. The electric motor is programmed to work in both a pedal-assist mode and throttle-controlled mode. In pedal-assist, the motor adds natural pulses of power to augment, but not override, the physical input of the rider. In throttle mode, the motor power is controlled directly by the hand-throttle. The main safety feature is the LIDAR system, which provides real time proximity awareness roughly 300 degrees around the rear of the bike. The system is interfaced via Bluetooth communication to a smartphone mounted on the handlebars which a displays a polar plot of LIDAR data, the throttle state, the bike's speed, and the currently selected light and motor modes.

System Details

LIDAR Awareness System

The LIDAR system consists of a time-of-flight infrared laser rangefinder mounted to the output shaft of a gearbox actuated by a stepper motor running at constant speed. It provides real time awareness of objects in a 40 meter radius. The Lidar works by sending out rapid laser pulses, which are reflected off the first object they hit, and then sent back to the unit. By calculating the amount of time it takes for the laser to return, the distance can be measured. The LIDAR is mounted on a rotating gearbox controlled by a stepper motor. By spinning the LIDAR, a 360 degree field of view is generated, though about 60 degrees of this field is obscured by the rider. The LIDAR speed is controlled by an Easy Driver stepper motor and a 2:1 torque reduction gearbox. The pulsed step input is provided very consistently by a self-contained 555 timer circuit, shown in Figure 2, ensuring that the stepper spins at a predictable speed at all times. The information from the LIDAR is then sent to the first Arduino Uno which then processes and packages the information and sends it to the Master Arduino Uno, which passes it onto the phone interface via Bluetooth where it is interpreted and presented as a polar graph. The physical LIDAR unit is shown in Figure 1. For about one third of the cost of a pre-built 2D LIDAR module, our LIDAR module has similar capabilities at roughly only 50% larger. The original sketch of the LIDAR box is shown in Figure 3. It was heavily modified in the shop to better reflect the properties of the LEXAN material, which was well suited to bending.

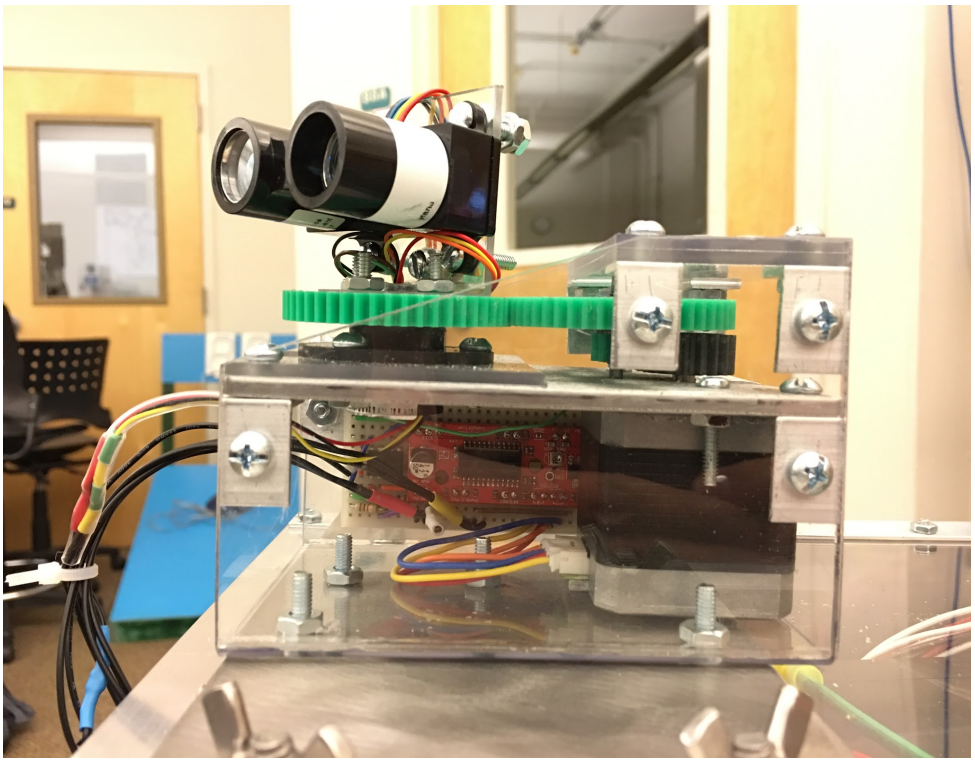


Figure 1: The LIDAR Module, gearbox, stepper motor, and 555 control circuit

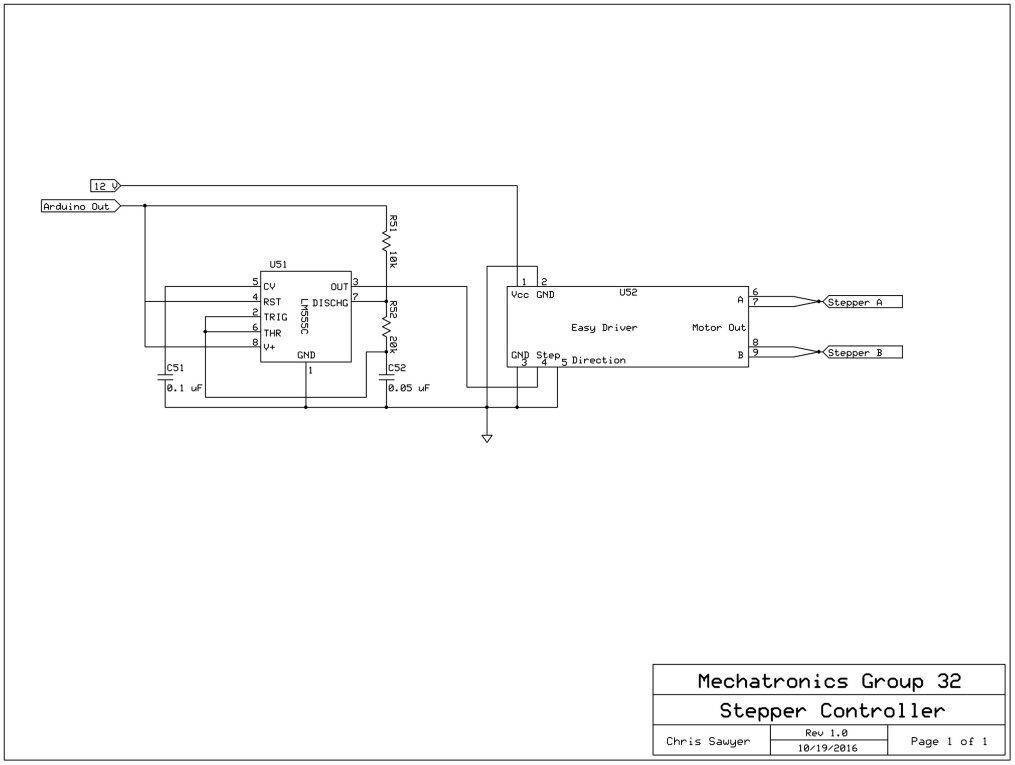


Figure 2: 555 Circuit used to drive the Easy Driver

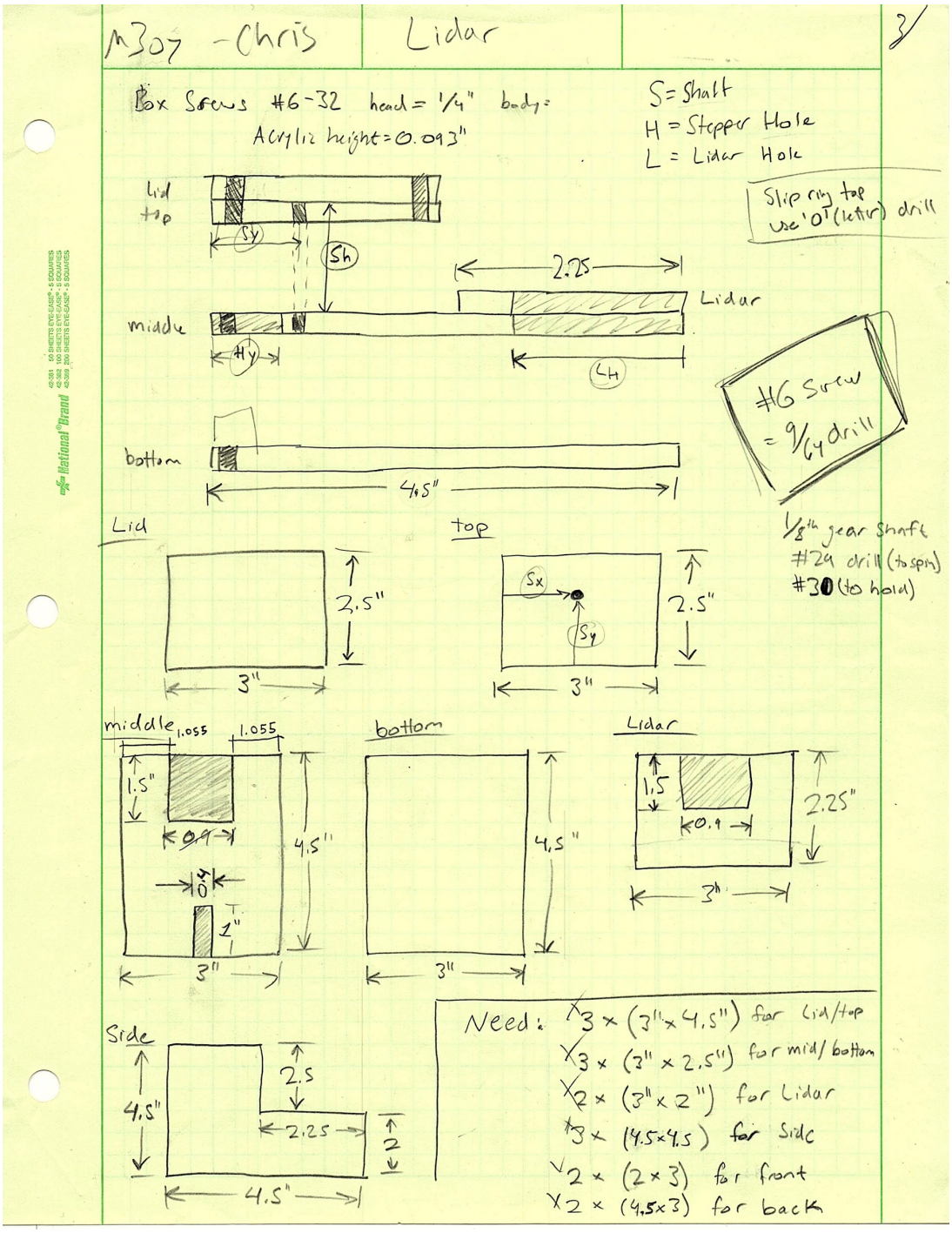


Figure 3: Preliminary LIDAR Blueprint

Motor System

A 36 Volt, 800 Watt MY1020 motor with a maximum current draw of 29.2 Amp was purchased to provide power to bike and is governed by an associated 1000 Watt motor controller designed for the MY1000 series motors. The motor provides power to the bike's existing drivetrain with roughly a 7:1 speed reduction, which gives the motor the benefit to use the bike's existing gear ratio. The drive sprocket is mounted to a freewheel crank that allows the motor to power the drivetrain of the bike, without forcing the rider to pedal. In fact, the motor can run at full speed with the pedals remaining perfectly stationary. The motor is mounted to the bike using a welded stainless steel bracket and three large bolts which pass through the neutral axis on the downtube of the bike. This mounting arrangement and drivetrain is shown in Figure 4. In order to accept the drive sprocket, the smallest of the three existing chain rings was removed and the derailleur was tuned to respect the new two ring arrangement. It was determined that the smallest chain ring had negligible benefit given the incredible torque assist provided by the motor. In addition, top speed was estimated using the conversion ratio shown below:

$$Bike\ Speed = \frac{Engine\ RPM \times Tire\ Diameter \times Drive\ Ratio}{Trans.Ratio \times 336}$$

$$Bike\ Speed = \frac{1000\ RPM\ \times 26" \times 2}{7.27 \times 336}$$

$$Bike\ Speed = 21.28\ mph$$

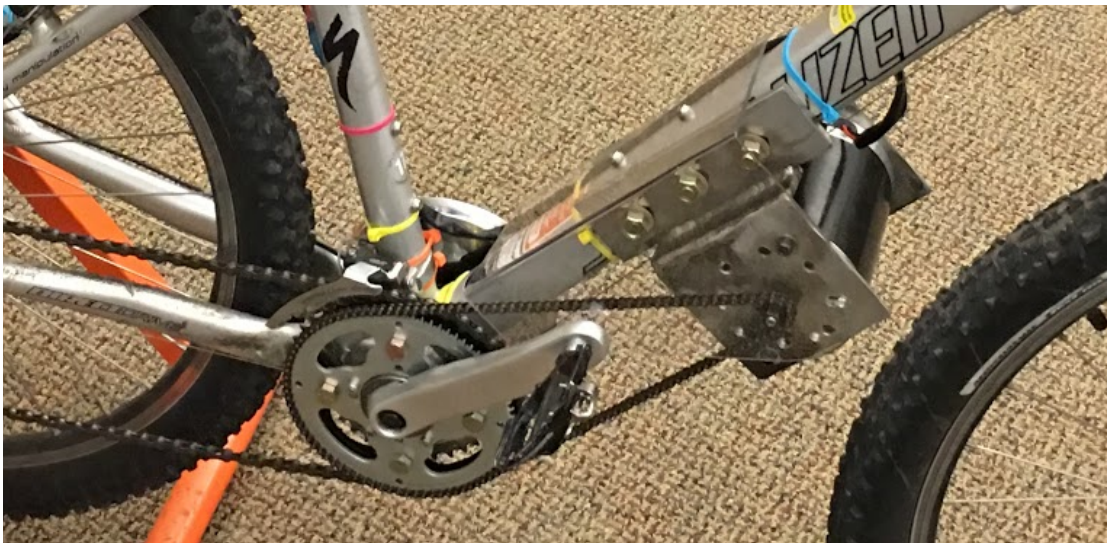


Figure 4: The Motor, chain, lexan chain-guard, and flywheel-crank assembly

Motor Modes

The motor has two modes. In Throttle Mode, a signal relay on the PCB allows the throttle on the handlebar to send its signal directly to the motor controller, giving the user direct, nuanced control over the speed of the motor. In Assistive Mode, the throttle signal to the motor controller is switched out and replaced by an output pin on the dedicated motor PIC which reads the magnetic-reed-switch cadence sensors on the left side of the bottom bracket, shown in Figure 5. When one of the two magnets passes the cadence sensor, the PIC provides the motor controller a voltage which uses PWM to ramp the motor up to full speed, where it is momentarily held before being turned off. This whole process only lasts for a fraction of a second. This pulse corresponds to the sinusoidal power input created by the motion of the rider's legs and provides a natural assistance that varies directly with the pedalling speed of the rider. Because the PWM output takes time, it is possible to miss a cadence reading if the user is pedalling faster than about 110 rpm. To combat this, a data flip flop holds the value of the cadence sensor until it is read and reset by the motor PIC. The catch is that, during testing, we found that that system encouraged the rider to pedal above about 100 rpm, which would cause the motor to hold at roughly 60% duty cycle, which is enough that the riders input is less than about 100 Watts. This meant that the system wasn't really assistive and did not respond to environmental needs or the needs of the rider, such as a hill or the desire to accelerate- it simply held at 60% duty cycle. By ignoring the flip flop and reading the cadence data directly, however, the assistive mode encouraged the rider to stay at around 60-80 rpm, a much more natural cadence. At this speed, the user's input is roughly 40% to the motor's 60%, meaning there is significant potential for the user to ramp up and accelerate if necessary. This is a rare, but useful example of where signal attenuation is desirable in a system.

Figure 5: Cadence sensors (yellow and orange zip ties) and crank plate holding two diametrically opposed magnets

Drive Sprocket and Freewheel

In order to have the motor to run without the pedals moving as mentioned above, a custom built drive sprocket and freewheel adapter were made. An 80 tooth drive sprocket was interfaced to a freewheel mechanism via several bolts. The drive sprocket is then bolted to the existing bike chainrings and spaced with washers to give appropriate clearance between the bike and motor chains. The freewheel is then threaded to the crank, and attached to the bike via the bottom bracket. The smallest chainring on the front crank was removed to allow for enough room to mount the assembly. Removal of

this chainring slightly limits the lowest speed range, but it can still be achieved by running the motor at a lower RPM. A longer bottom bracket is used to provide the extra clearance the drive sprocket needs. The 80 tooth drive sprocket provides a gear reduction of 7.27 from the motor to the drive sprocket, which helps the motor run at a higher efficiency, and limits the torque spikes caused by the sudden acceleration of the motor. The completed drive sprocket assembly is shown in Figure 6.

Figure 6: Freewheel Crank Assembly

Visual Output and User Interface

The rider interacts with the systems on the bike primarily through the front console which contains a phone mount, two SPTT switches, and a USB charging cable for the phone. The phone, an old LG G2 in this case, is mounted with easy access to the user between the handles at the front of the bike. The phone has a custom graphical interface designed in the Unity game engine (C#) which displays the LIDAR information as a polar graph, the bike speed, throttle position, and the switch positions for the motor and lights. The phone also emits a sound recording of the state of the lights or motor when toggled. This gives the rider audio feedback which means they do not necessarily have to look down at the phone to confirm their selection. The phone interface communicates via Bluetooth to the Master Arduino which sends the LIDAR information. This handlebar console is shown in Figure 7.

Figure 7: Front console with light control switch on the left and motor control switch on the right, and the phone readout in the red phone mount.

LED Headlight and Tail lights

Highly visible smart LED headlights and taillights will automatically engage in low light conditions via a photoresistor mounted on the back end of the bike, but can also be turned on or off manually at any time. These LED lights are identical USB powered bike lights that are normally button activated. They were torn down and modified to accept a generic 5 Volt source and the button was removed and a hardwired signal cable was soldered in place, allowing for the light to be actuated by the light controller PIC. They contain a small logic board which handles secondary modes such as strobing or "low-power mode". The board logic is toggled on the negative edge of the switch movement and to handle this the PIC was required; otherwise they could have been connected directly by the switch. The lights are approximately 1200 Lumens and have had custom diffusers fitted. The diffuser in the front creates an interference pattern that spreads the light out horizontally which provides the cyclist with a large illumination area in front of them. The tail light has a diffuser which tints the light beam red and spreads the light out vertically. The advantage to this is the light will be more visible to oncoming cars as the focal range will be oriented preferentially in the direction of travel to aid in cyclist detection for other vehicles. The headlight and taillight are shown on the following page in Figure 8.

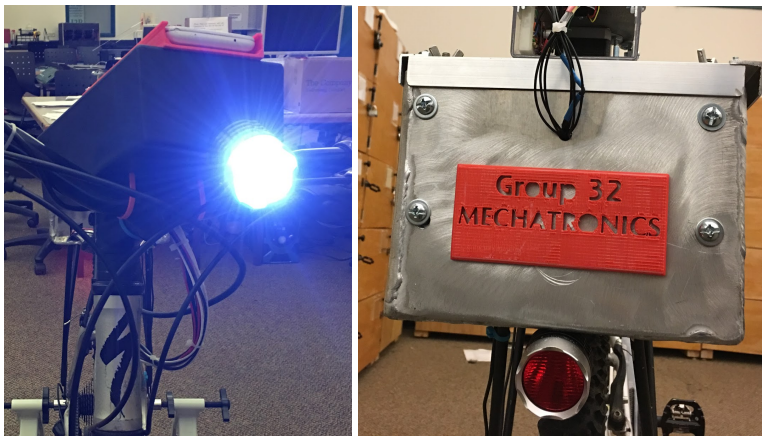


Figure 8: Headlight and Taillight

Power Supply and Electronics

The entire power supply for the bike is contained in three 12 Volt, 12 Amp-Hour Lead Acid batteries wired in series providing 36 Volts and 432 Watt hours. These batteries were selected for their high energy density as well as relatively low cost. The lead acid batteries also provided a level of resilience and helped with the design considerations to illustrate marketability in multiple applications. Lithium ion batteries were considered for the better size to energy ratio, but ultimately rejected due to high cost and their sensitivity to cell-to-cell disbalance. The battery rating provides power to the motor and subsystems for approximately 30 minutes if running the motor at full power. In pedal assist mode, the lifetime of the batteries is expected to be over two hours. The bike has a custom charging port located at the back of the bike in order to charge the batteries after use. This port is wired through a DPDT switch in order to isolate the batteries when charging and operating the bike.

However, none of the other electronics on the board can run off of 36 Volts. It would be cumbersome and inefficient to have a secondary power supply, so a custom printed circuit board was

designed to accommodate all of the electronics, including (but not limited to) both PICs, a Bluetooth module, a data flip flop, a transistor DIP, a ribbon cable for power and signal distribution, and (most importantly) three DC to DC converters. Two of these converters step 36 Volts down to 5 Volts and the third steps the 36 Volts down to 12 Volts for the stepper motor. This printed circuit board is shown in Figure 9 and Figure 10. The wiring schematics for the PCB are shown following it in Figures 11 through 14. Countless design considerations come to light once a custom PCB is on the table. Looking at Figure 9, the different colors represent different layers. The four-layer board had a top layer of traces in red and a bottom layer of traces in green. The middle two layers were power and ground planes and connections to these planes are shown with the white **+** and **-** throughout the board. The yellow part outlines and text is a silkscreen layer. Designing this PCB required extensive research regarding components and their specifications. The three vertical rectangles labeled U11, U12, and U13 were DC-DC converters capable of handling our 36 Volt source and stepping it down to anywhere from 2 Volts to 12 Volts, depending on the value of an attached resistor (R11, R12, and R13). The converters also required aggressive signal conditioning on the input and output. Capacitors C11, C12, and C13 were each 1,000 microfarad capacitors the size of a thumb; in order to handle the incoming 36 volts. C14, C15, and C16 could be smaller because of the smaller output voltages, but had to be 2,200 microfarad capacitors, as required by the manufacturer of the DC-DC converters. In addition, you can see the large green traces on the left side of the board. Trace length is extremely important on PCBs and have to be carefully selected based on the current load of the trace. Those large green traces were specified in order to accommodate the 9 amps of max draw possible from the three DC-DC converters. The large green polygonal traces were designed to accommodate the unusual 3-pin output of the DC-DC converters, which the manufacturer specified to be electrically connected. Moderately large red traces coming from RLY24 were selected to accommodate up to 2 amps each- they are supplying power to the headlight and taillight. Another interesting design quirk that cannot easily be seen in Figure 6 but can be made out slightly as a faded white rectangle on the back of the board in Figure 10, was the need to split the power plane. It could not be guaranteed that the two 5 volt DC-DC converters would have the exact same voltage value. If they did not, they could not be safely wired in parallel. Therefore, the top 5V source is electrically isolated from the bottom 5V source. Components had to be cleverly arranged to draw from the appropriate power plane. RLY24, for example, straddles the two power planes in order to utilize one source for the headlight and one source for the taillight as to not approach the 3 amp limit of each source. All of the electronics on the entire bike had a common ground at the negative terminal of the batteries, so separate ground planes were not required.

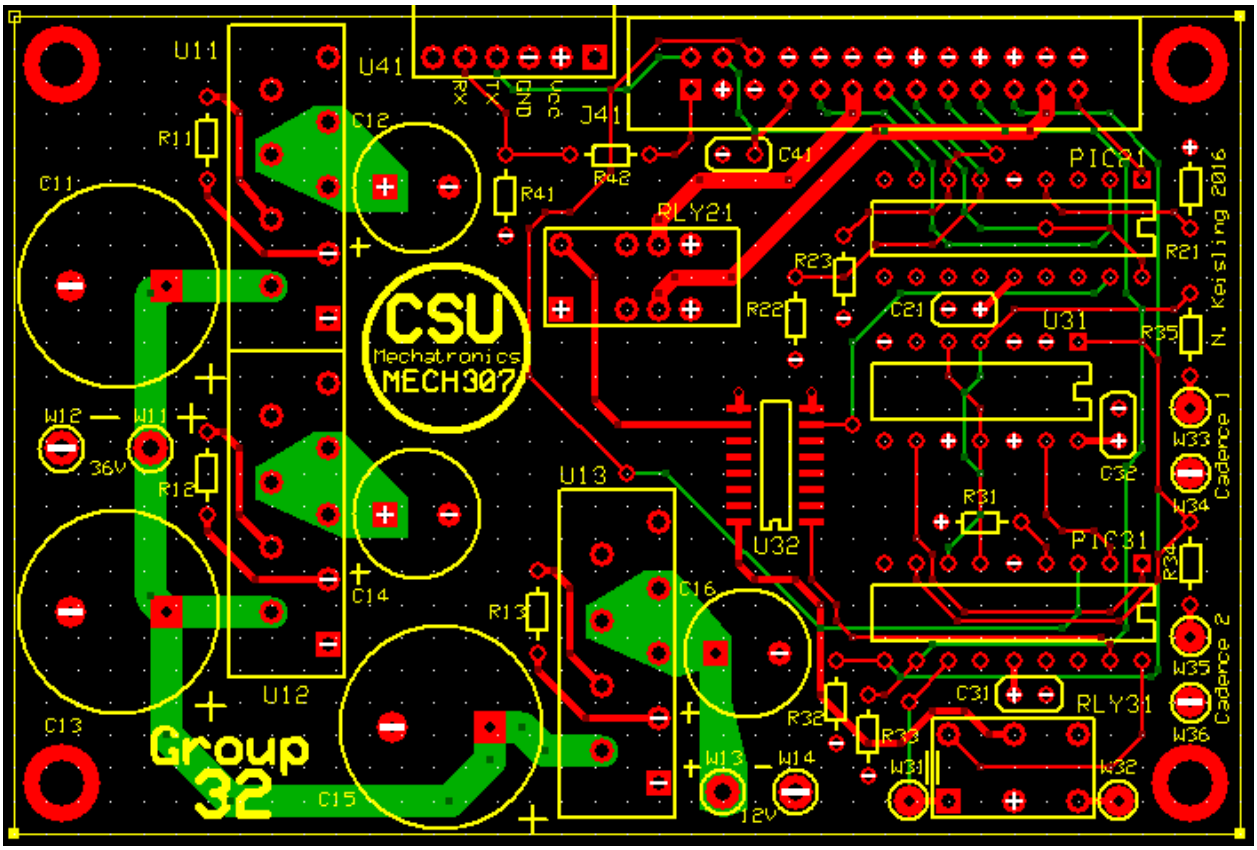


Figure 9: PCB Schematic

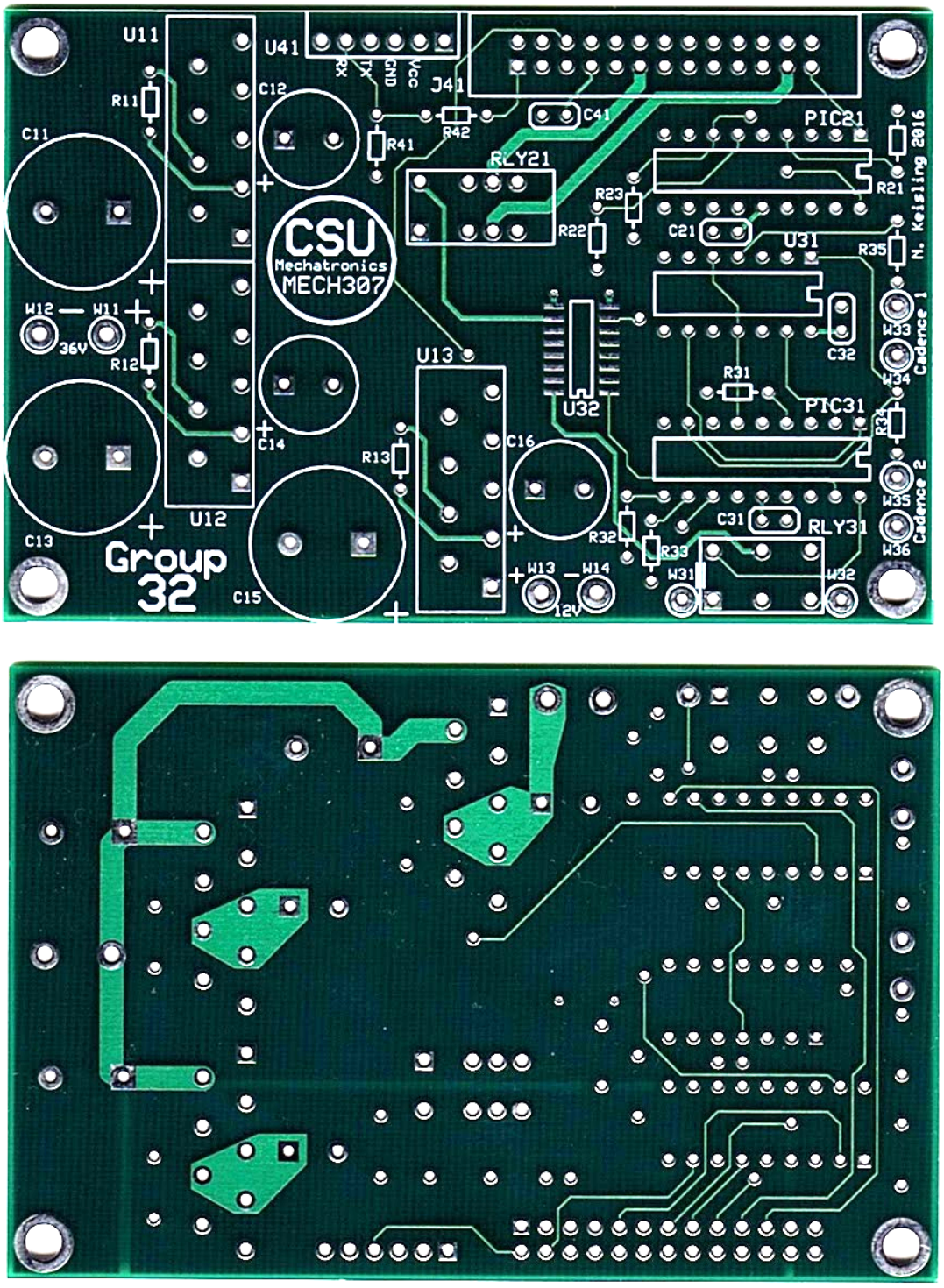


Figure 10: Front and Back of Custom Printed Circuit Board

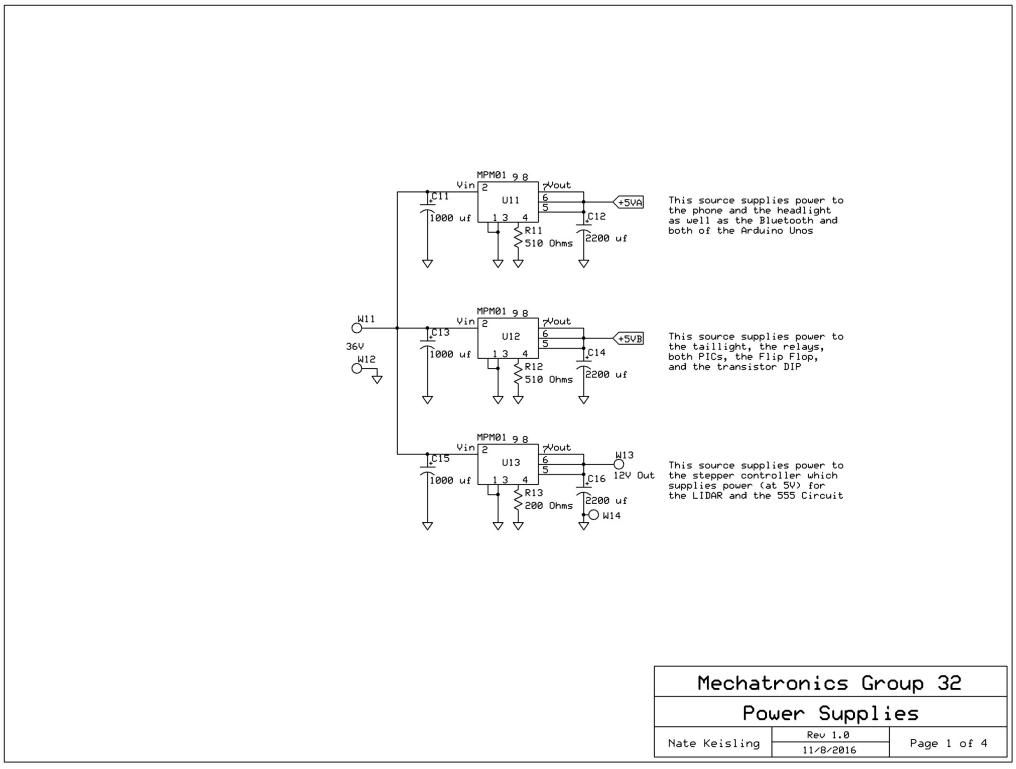


Figure 11: Power Distribution using Three DC-DC Converters

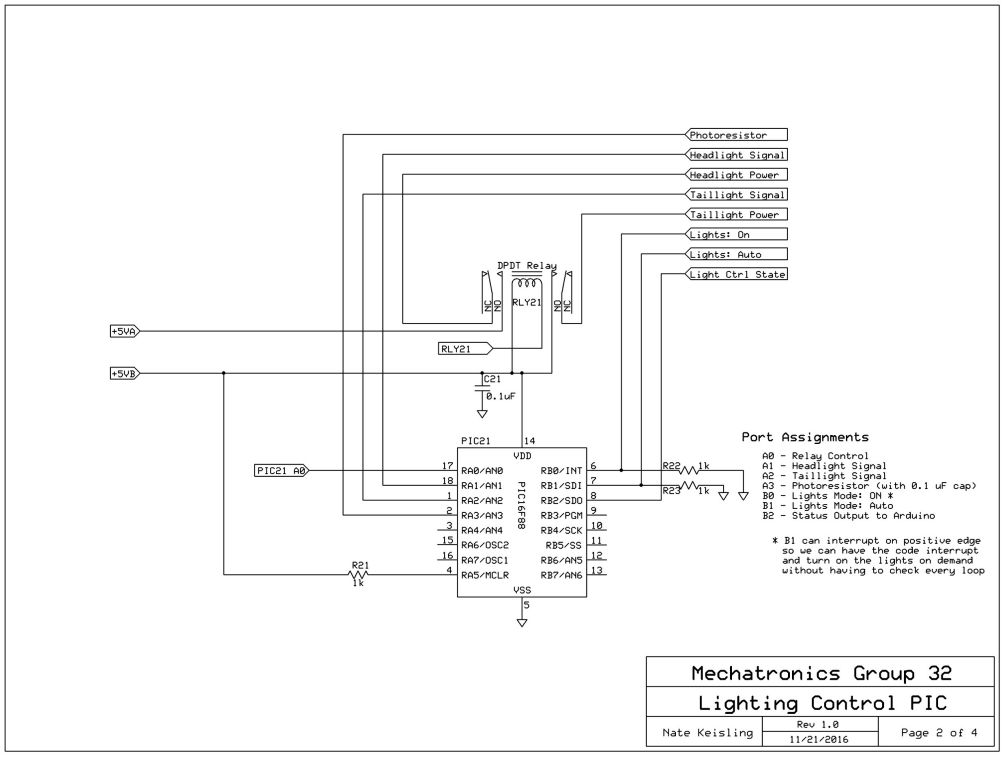


Figure 12: Headlight and Taillight Control PIC

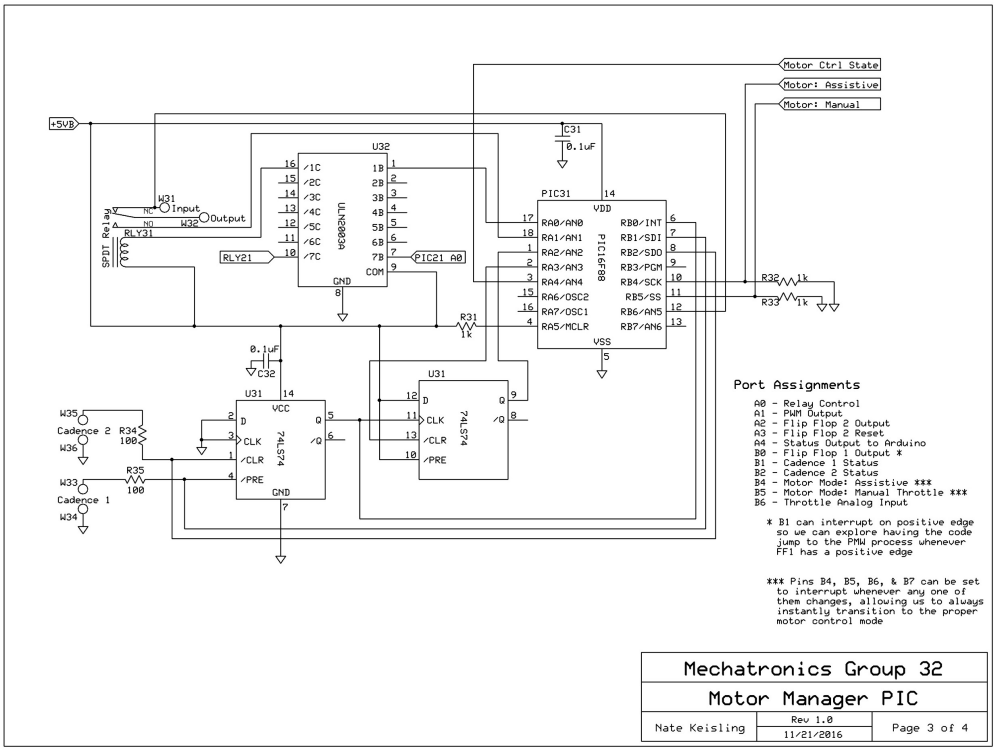


Figure 13: Motor Manager PIC and associated support circuitry

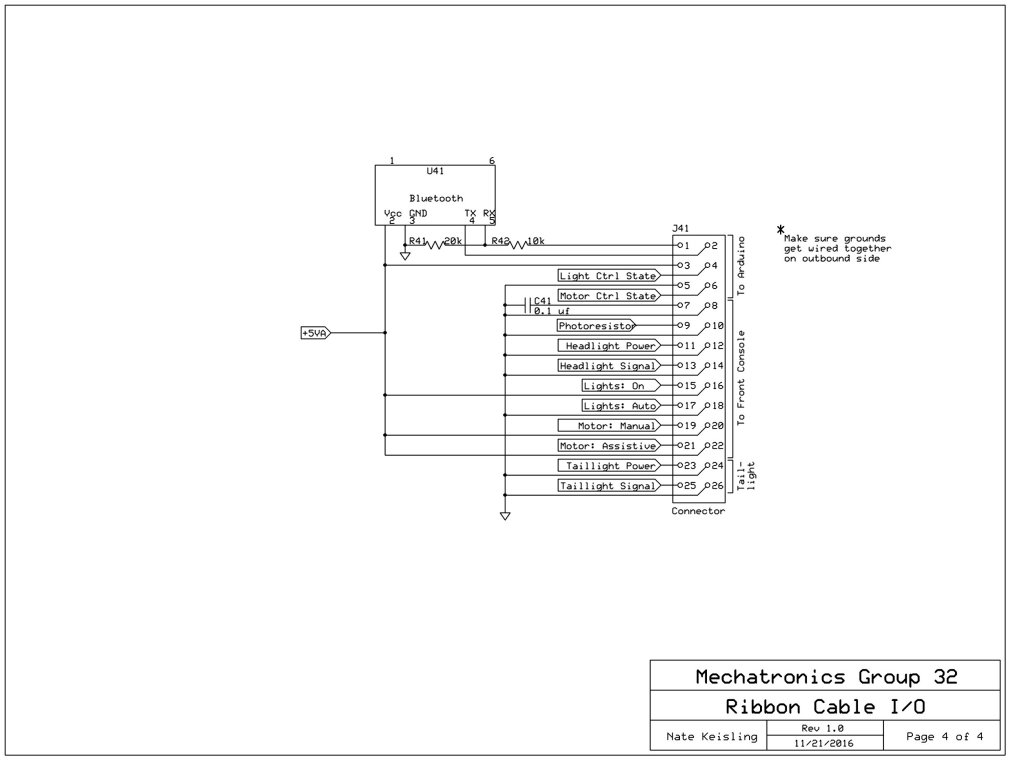


Figure 14: Ribbon Cable Input/Output and the Bluetooth Module

Functional Diagram

The following page is a diagram depicting connections between major components.

Red lines are digital logic connections or analogue inputs, in the case of the sensors.

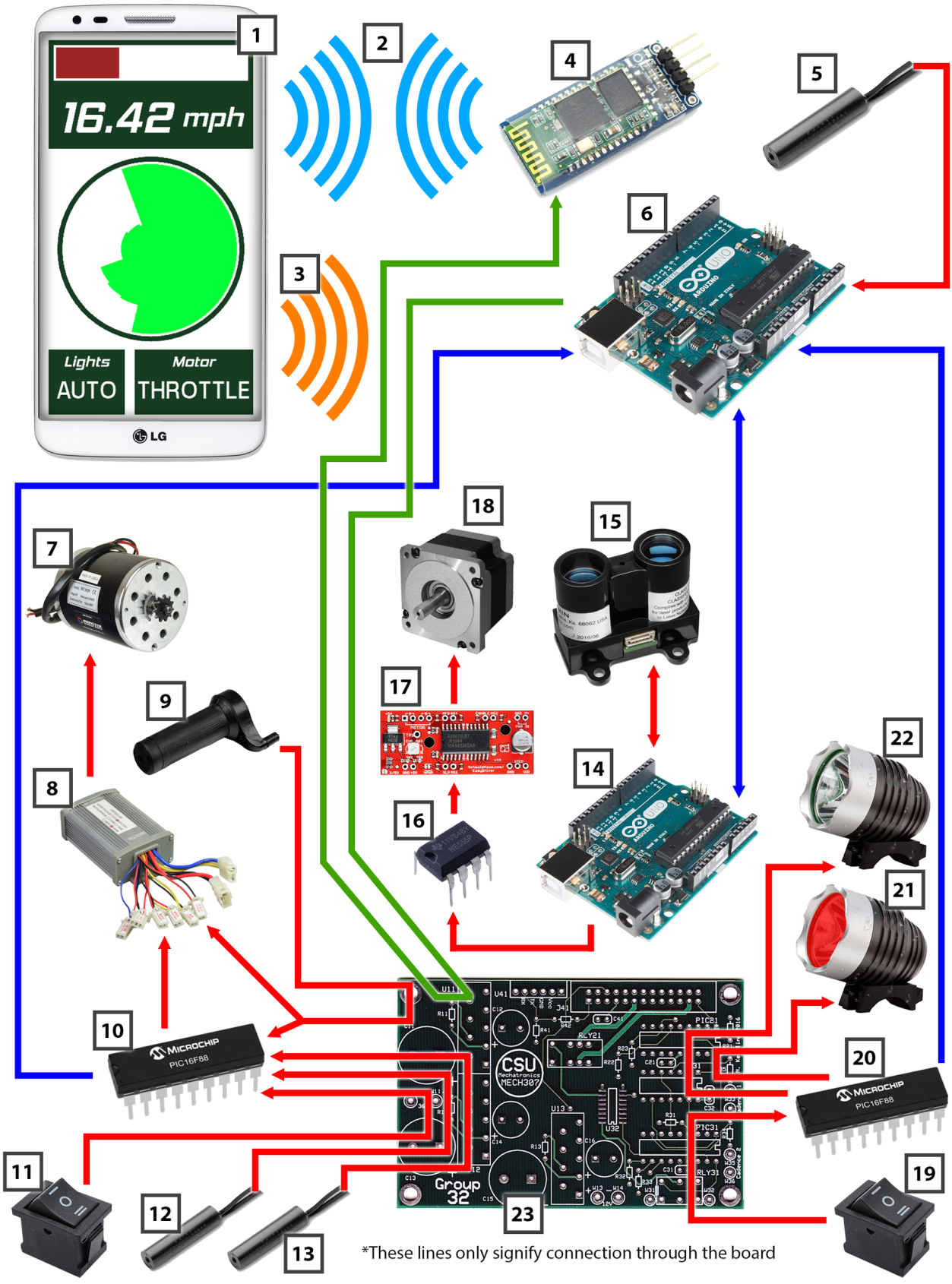
Blue lines are for serial communication between the microcontrollers. The two PIC's and the LIDAR-Arduino Uno send information to the main Arduino Uno.

The green line is the serial connection from the main Arduino Uno to the Bluetooth module. This information combines the information read from the blue serial lines.

Components:

1. LG L2 Android Phone with custom app
2. Bluetooth signal
3. Audio output from phone
4. HC-06 Bluetooth Module
5. REED Sensor for wheel speed
6. Arduino Uno which handles information coordination
7. 800 Watt MY1020G Motor
8. 1000 Watt Motor Controller
9. Electric Scooter Throttle
10. PIC 16F88 which handles the throttle and motor controller
11. SPTT Switch to set motor-mode. (Off, Throttle, or Assistive)
12. REED Sensor 1 for assistive mode
13. REED Sensor 2 for assistive mode
14. Arduino Uno which handles the LIDAR system
15. LIDAR Module
16. 555 Timer to send pulses to the stepper controller
17. Easy Driver Stepper Controller
18. Stepper motor
19. SPTT Switch to set lights-mode. (Off, On, or Automatic)
20. PIC 16F88 which handles the lighting system
21. Rear Running Light
22. Headlight
23. Our purpose-built printed circuit board

Note: The blue serial lines also travel through the custom PCB but are shown on the edges of the diagram for clarity.



*These lines only signify connection through the board

Design Evaluation

Output Display

- Phone Application displaying LIDAR Information, motor mode, and light positions
- LED Headlight and Taillight activated by photoresistor

The included Android App acts as a readout for the system. The phone connects to the coordination Arduino though an HC-06 Bluetooth radio to display the following information to the rider: throttle position, speed of the bicycle, a radial plot of the LIDAR information, and the state of the motor and lighting control switches. This system is a rather advanced form of output display and has been largely successful other than some glitches with the polar LIDAR display.

Audio Output Device

- The phone app plays a descriptive phase when the motor and lighting control switches are changed.

This functionality is to allow the rider to focus on the road while still being sure of what they're doing with the controls. This required a significant amount of research regarding the integration of audio cues in a phone application.

This is highly functional, but not particularly loud. In the future, the phone's primary purpose would be to process the information from the LIDAR, track a kinematic model of the surrounding area, and alert the rider of any imminent collisions using an alarm output. That advanced level of data acquisition and live analysis was determined to be even farther outside the scope of this already out of scope project.

Manual User Input

- Phone Application
- 2 SPDP Switches to control the LED Lights and the state of the motor controller
- Throttle Input
- Key Switch to charge the batteries and turn system on

There are two SPDT switches on the control panel which control the lighting system and the state of the motor controller as well as a throttle. The lights can be on, off, or controlled automatically. The motor controller is either off, controlled by the throttle, or run in a purely assistive mode. These phrases are also displayed in the app readout. Use of the phone as a touch input was evaluated and considered a triviality. We chose not to go this route because it is not convenient to operate a touchscreen while riding a bike- physical switches will always be far superior under such conditions.The other user input is the throttle control used in the pure throttle motor mode. This is a hall-effect rotational position sensor. Both the throttle and the two switches functioned flawlessly.

Automatic Sensor

- LIDAR System to detect surrounding environment
- Photoresistors to automatically turn LED's on and off
- Reed Switches to help with the motors pedal assist mode

LIDAR, in the form of a laser rangefinder being rotated by a stepper motor at a particular rpm, scans the surrounding area and builds a map of the riders blind spots. It works to a primarily demonstrative extent, and would only be of moderate use on the actual road. This was always the intent- making the

LIDAR do precision polling would have increased the data by an order of magnitude and that could not be handled by our Arduinos.

Reed Switches provides accurate crank rpm (cadence) for use matching the rider's power input in Assist Mode and calculating the speed of the bicycle. There are cadence sensors attached to the flywheel crank assembly to sense when and how quickly the rider is pedaling to precisely power the motor in assistive mode. These sensors work flawlessly. Another reed switch reads rotations of the back wheel for speed purposes and also works flawlessly.

A Photoresistor determines ambient light levels and turns the lighting system on or off. We had some problems with it misbehaving indoors, but it's largely functional.

Actuators, Mechanisms, and Hardware

- 800 Watt brushed DC motor to propel rider
- Stepper motor to drive the LIDAR module
- Freewheeling Crank Assembly

An 800 Watt brushed DC motor drives the front chain rings. This motor works to perfection. It can get the bike up to around 20-25 mph under pure throttle and provides a perfect, natural, balanced assistance in assistive mode. Stepper motor drives the LIDAR module. Actuated by a 555 timer. Works perfectly- spins at a constant, known rpm. The Freewheeling crank assembly allows front chain rings to be driven by motor without turning pedals. Also function perfectly- at full motor power the pedals can be perfectly stationary. Even when the motor pulses on abruptly, no pulse can be felt by the rider.
The motor mount is welded from stainless steel plate and custom fit to the down tube of the bike as well as the front and rear plate of the motor. This helps with mitigating and compensating for the added torque of the motor and the holes drilled through the aluminum down tube.
Pass-through cable slip ring assembly allows output cabling from LIDAR module to remain static while the module rotates continuously. Also works flawlessly in conjunction with a custom designed gear assembly connected to the stepper motor.

Battery box at the rear of the bike safely and securely holds all three batteries and all of the support electronics. Though with consideration for the significant added weight of the batteries, a quarter inch aluminum plate was chosen to provide a shell with the relative strength of steel without the weight. This box is designed to separate and place the bulk of the weight of the batteries directly between and orthogonal to the main support over the rear axle and be held with a forward tilt to below the seat and fastened with 12.9 metric bolts. With additional holes, there would be adequate airflow to dissipate any potential heat built up due to the current draw of the motor and electronics. The box also provides adequate space to mount the electronics in an efficient manner, but it is not weatherproof.

Logic, Processing, and Control

Both flowcharts describing the code and important sections of the code itself can be seen in the appendix. These are certainly the best way to understand the code. A rough outline of the logic follows:

Motor PIC

- Reads the motor-control switch state
- Reads the throttle position
- Reads the cadence sensors' states
- Handles the motor controller. It has 3 states:
    - connected to the PIC, turned off - "Off"
    - connected directly to the throttle - "On"
    - connected to the PIC, waiting for a cadence read to trigger a ramping-pulse - "Assistive"
- Spams the switch state and throttle-position over serial every loop

Lights PIC

- Reads the lights-control switch state
- Reads the photoresistor
- Turns on or turns off the lights based on the selected mode (On, Off, Auto) with this data
- Spams the switch state over serial every loop

LIDAR Arduino

- Provides power to the 555-Timer Circuit connected to the STEP pin of the Stepper's EasyDriver
    - This provides a constant rotational speed without needing any logic from the Arduino
    - (If we had more time, a method for sending PWM signals to the step line from the arduino was devised and tested that let us play music with the stepper-motor acting as the speaker. If steps are sent faster than the motor can move, it produces a high-pitched whine. The frequency of these pulses directly relates to the pitch sounded. Oh well.)
- Pulses the LIDAR, reads the returned distance and the time since the last measurement
- Caches this information until the Main Arduino asks for it to be sent
    - Then sends the information over serial and deletes it to make room for more data

Main Arduino

- Uses an interrupt to read the REED switch to get wheel RPM, and from that bike speed
- Waits until it reads a full packet from the Lights PIC, stores it
- Waits until it reads a full packet from the Motor PIC, stores it
- Sends a signal to the LIDAR Arduino asking for it to send the cached LIDAR information
- Listens to the LIDAR Arduino's information
    - The PICs send much shorter strings than the LIDAR Arduino does so they can spam theirs while the LIDAR Arduino waits to send the big string only once.
- Builds one string containing all of this information
- Sends this string to the phone through the bluetooth module

Arduino Phone App built in Unity3D

- Uses a bluetooth library to connect to the bluetooth module and read strings into the app
- Parses the string into integers. Stores partial strings to be completed next loop due to streaming
- Displays the information about the throttle, the switch states, bike speed, and the LIDAR plot
- Plays audio files though the phone speakers when a switch is toggled

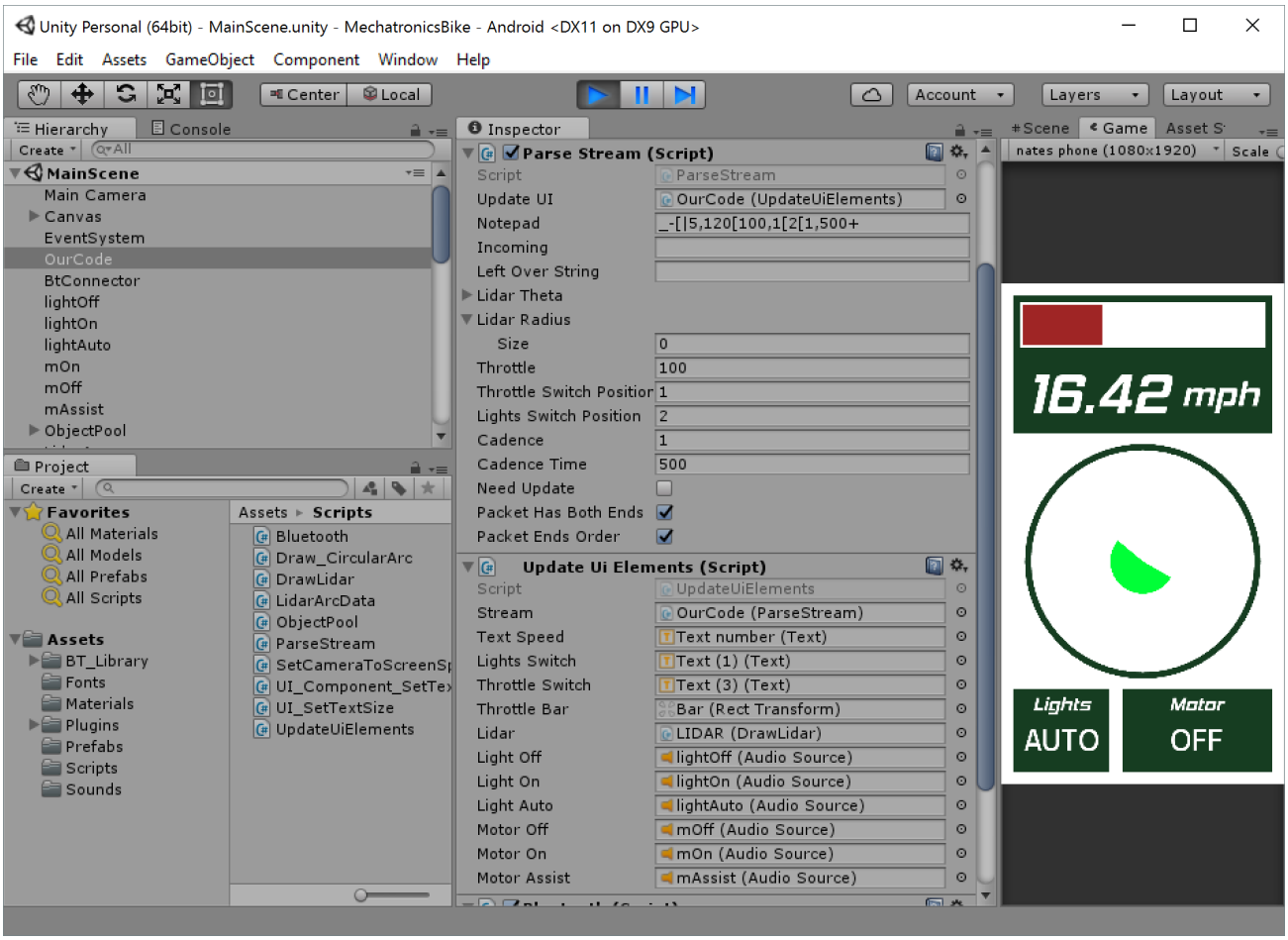


Figure 15: The Unity 3D Editor and basic setup of Unity GameObjects

Figure 16: The Android App with mockup of realistic LIDAR information

Partial Parts List

Overall costs, including valuation of the bike and the phone, are in the range of approximately $700.

Item	Supplier	Cost
2004 Specialized Rockhopper Mountain Bike	On Hand	Worth $100, Free
LG G2 Smartphone (Circa 2013)	On Hand	Worth $25, Free
Custom Printed PCB	ExpressPCB	$89
PCB Components	DigiKey	$60
2x PIC16F88	Class Supplies	Free
2x Arduino Uno	Class Supplies & On Hand	Free
LIDAR Module	Sparkfun	$150
Easy Driver Stepper Controller	Sparkfun	$15
800 Watt MY1020G Motor	Monster Scooter Parts	$90
1000 Watt Motor Controller	Monster Scooter Parts	$50
12 Volt Razor Batteries	Amazon	$90
Headlight/Taillight	Amazon	$30
HC-06 Bluetooth Module	Amazon	$6
Freewheel	EbikeParts	$15
Cranks	EbikeParts	$20
Drive Sprocket	EbikeParts	$20
Raw Materials: Lexan/Aluminum/Steel/Fasteners	Varies, much on hand	$50
Total		$685

Lessons Learned

Team Dynamics and Management

Overall, our group severely struggled to get the ball rolling early in the semester. Our project concept was clear and agreed upon from the beginning and a rough time table was set, then the first deadline was immediately failed. The following deadlines crumbled soon after. This would become a running theme with the project and all scheduling and deadlines were quickly thrown out the window in favor of a "this is your job, get it done, don't screw it up" mentality. This was the direct result of the biggest lesson we learned, which was to put more effort into the planning phase of the project and to make and agree upon a detailed schedule before diving into the project.

Another huge facet of this issue was our failure to fully conceptualize and research every part of the project as a unit and understand potential stumbling blocks on the whole. We had a tendency to instead jump in and research a part of the project, design it, build it, solve problems as they came up, finish it, and then repeat the process for the next part of the project. For certain things, this was okay. Things like mounting the LIDAR module to the battery box or the front console to the handlebars can be figured out on the fly with minimal wasted time and minimal planning. Many other things could not be treated with such nonchalance, and it should be clear that poor planning was a common theme in the following sections.

Finally, a core issue we faced was deciding on a project before fully understanding the effort level that each group member was willing or able to bring to the table. This comes as our biggest piece of advice: **Decide on a project appropriately difficult relative to the average ambition of the group as a whole, not relative to the most ambitious members.** Failing to do so causes more issues than you might at first realize. One such supplemental issue is the bulk of the work falling on the more ambitious group members, though it should be understood that this is not always the fault of the less ambitious group members. There were certainly times in this project where the more ambitious members took on extra work because doing so was perceived as easier and more effective than having to conceptualize the work for another member and then check on their progress. It is important to have faith in your fellow group members, but it is even more important to openly discuss group members failures and reassess your faith in other group members directly.

Mechanicals

One of the most significant delays in the project was the freewheel crank. It was scheduled to be done before the halfway point, yet wasn't finalized until the deadline was two weeks away. An abstract failure to plan was the direct fault of these delays. It was presumed by the assigned group member that the required parts could simply be purchased, assembled, and installed. When it came to installation, however, no measurements had been made to confirm that the new crank assembly would fit on the bottom bracket. Not only did it not fit, it was off by several inches. A special bottom bracket was ordered with a longer shaft, yet no formal measurements were ever made to confirm that this was an adequate solution. It was not. The team opted to ditch the smallest chainring, which we knew we didn't need anyways, but this still wasn't enough to fit the assembly. A clever inversion of the freewheel and drive sprocket assembly and some associated modifications to the crank allowed the assembly to fit on the crankshaft, but the chainrings rubbed against the frame of the bike and large bolts used in assembly would not allow the assembly to rotate, something rudimentary measurements would have anticipated. The bolts and the frame were grinded down to allow for enough clearance to be functional. With proper

measurements, detailed drawings, and appropriate planning, this 8-week affair that caused several other delays could have been avoided entirely.

The rest of the build process only served to continue to reinforce the basics; measure twice, drill pilot holes, the right tool for the right job, keep it simple, don't rush it, etc etc. All lessons that shouldn't be overlooked but should also be considered fundamental. Further, when determining materials, more considerations could be taken to optimize weight to strength and space considerations can be made to streamline the package without creating such a difference in the mechanics of how the bike performs when it is ridden. Understand the functional environment of the device you are building and make appropriate design adjustments. It is easy to say that a certain thickness or type of material will be "more than adequate" in terms of strength or machinability, but is it optimized? Could you get away with less? Even if it is a back-of-the-napkin estimation, analyze all your design choices for efficiency. Specifically, we are referring to the 1/4" thick aluminum used to construct our battery housing.

Another issue was communicative failures that lead to the entire scrapping of what had the potential to be the iconic centerpiece of the bike. Early on, the plan was to 3D print a somewhat large box designed to form fit to the handlebars and elegantly and ergonomically hold the phone, headlight, and control switches. The first issue was leaving this assigned to the same individual who was handling the crank assembly. The number of issues and delays faced in regards to the crank should have immediately been grounds to redistribute work. Instead, the console design was left on that individual's shoulders and it arrived late, missing major design elements that were repeatedly and explicitly discussed, and in a form not yet ready for 3D printing. To make matters worse, no one in the 3D printing studio bothered to mention that our reserved printer was essentially broken. What was to be an 18-hour print failed in the first 6. The design was scrapped and desperately simplified. The lessons learned there are twofold: First, don't be afraid to redistribute work. Second, all design elements need to be on paper, ideally in sketch form. Discussing design elements in person with no paperwork, no matter how often, is a recipe for elements being forgotten or misunderstood. Having design elements explicitly laid out with notes and sketches also eases the transition in the event that work needs to be redistributed.

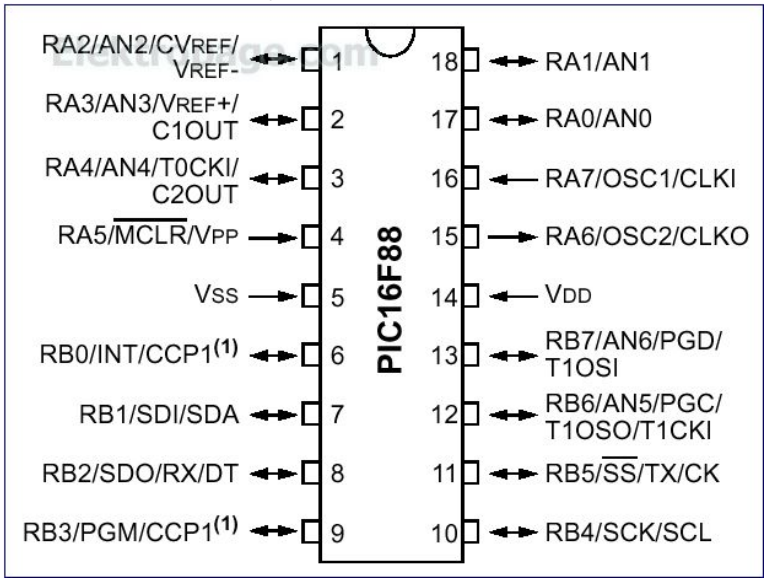
Electronics

Fortunately, this is where things went right, through lessons still abound. The electronics system and the use of a custom PCB was an ambitious addition to the project, but an obvious one given the electrical constraints. The corner we painted ourselves into was that the PCB design had to be flawless. There was not enough time or money to fix the design and have them reprinted, so we were forced to work with whatever came from the manufacturer. Changing design constraints and poor planning meant that the electrical schematics that would become the PCB were up in the air all the way until the PCB design was completed. In fact, moments before the PCB order was placed, yet another change was made and the design was quickly updated. In the end, this meant that the functionality of the PCB defined the potential functionality for the whole bike- if it had major design flaws, the bike would simply not work and there would be very little recourse. So the first piece of advice on this subject is to avoid these make-or-break situations at all cost. If they can't be avoided, put them first above all else. Let them dictate the course of the rest of the project as we did. With many hours of double checking schematics and some advice from a relative who happens to be exceptionally experienced electrical engineer, the PCB's arrived and functioned perfectly from top to bottom and provided an incredibly stable platform with which the rest of the bike was built off of.

That being said, our failure to do significant research up front significantly blunted the effectiveness of the LIDAR system. One Arduino Uno was completely overwhelmed with what we expected it to because of serial limitations- it only has one hardware serial bus and could not possibly receive *and* send data from the LIDAR to the phone fast enough to be even demonstrative. The introduction of the second Uno allowed for us package the data nicely enough that the master Uno could handle sending enough data to the phone to at least provide the proof of concept. If we had planned better from the beginning, these limitations would have been clear and we would have immediately researched the use of a much more advanced microcontroller such as the Arduino Mega, (which has 3 hardware serial ports), or even a fully fledged computer, such as a Raspberry Pi. Instead, with a month remaining, we were forced to decide between an expensive and unfamiliar upgrade or allowing the LIDAR to be more demonstrative than actually functional on the road. It was a hard choice that could have easily been avoided with better planning. It was also interesting to us that in the end, a fully realized LIDAR module and associated phone app would have been a fully realized and exceptionally well received Mechatronics project in its own right.

Code

The process of writing code is a constant back-and-forth between having an idea, testing the idea, and consulting the manual when you realize that your basic understanding of the how to implement the idea is mostly wrong. That said, I ran into more than a few roadblocks when writing in the three languages used to animate all of the aforementioned hardware. So, I'm going to list a few key ideas in hopes that you'll be able to benefit from them. Good luck! -Chris

- **Actually read the manuals. Look them up, save them, and read them.** Once you've actually read them, when you have a question you'll be able to search for it directly instead of through google. Incredible secret: the people who answer forum questions do so by having read the manual.
    - Example: I wasted 5 hours with bad analogue data from a PIC because I didn't know that the Analogue AN pinouts were different from the A or B registers. RA1 happened to line up with AN1 in the example code so I assumed that was the same for all pins.That wouldn't have been a problem if I had read the few lines above this figure:
    - 

- **Do tests with your hardware before you make assumptions about how you can structure your software.** We initially assumed that the stepper motor and LIDAR could both be controlled with the same Arduino. The stepper control is just an OFF/ON/OFF pulse and the LIDAR, by definition, works at the speed of light. However, it turns out that the processing hardware under the laser does not work as quickly as we had hoped. No matter how we configured it, the stepper would be jerky and jittery. This forced us to build the separate 555-Timer circuit to control the stepper because the LIDAR just takes too much time to actuate.

- **Understand exactly where the data in your system is going to flow before you plan which microcontrollers you are going to use.** Had we planned this ahead of time, we would have used just one Arduino Mega instead of two Arduino Unos because the Mega has three hardware serial ports instead of one, like the Uno. Reading the LIDAR data needed the Arduino's active attention so extra serial ports could have passively recorded data from the PICs. That would have been a massive efficiency improvement in the serial data sending structure because all of the lidar data could have been stored in the same memory that it would be read from when sending it over bluetooth, instead of waiting for a serial transfer.

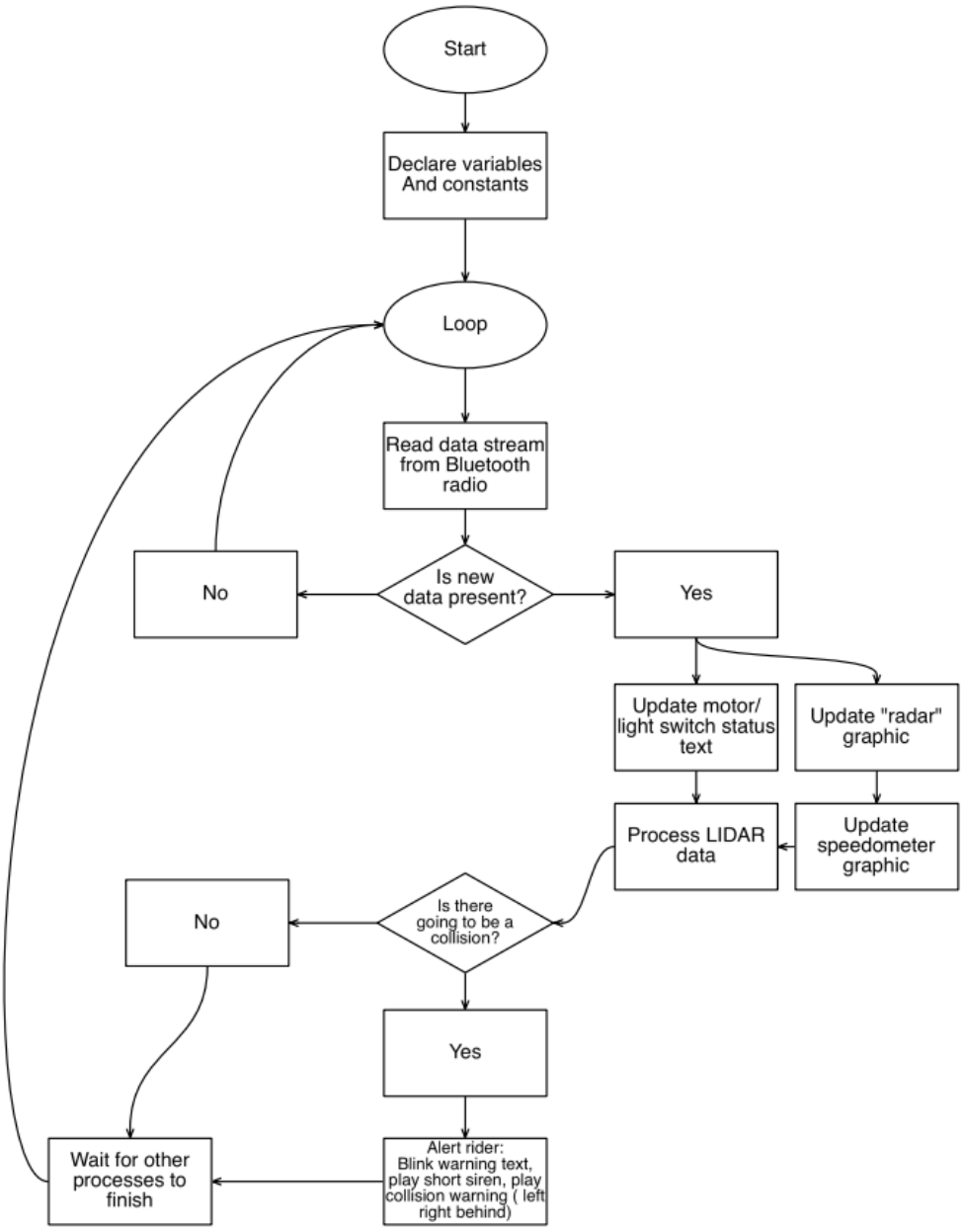
- **Most microcontrollers are single-threaded**, meaning that they can only handle a single task at a time. This may not sound like much of a problem, but it's more restrictive than it sounds. Even if your task is very simple, like reading a switch, if there is a heavier task that also needs to happen at the same time it will block the simple task or at the very least make it happen in an unacceptable amount of time. The number of things that *need* to happen at once directly dictates how many parallel processors you need, be they on single-processor microcontrollers or 8 core computers.

- Know that in almost all circumstances **your first plan will not work. This is absolutely normal.** Give yourself enough time to come up with a second idea and actually build it too.

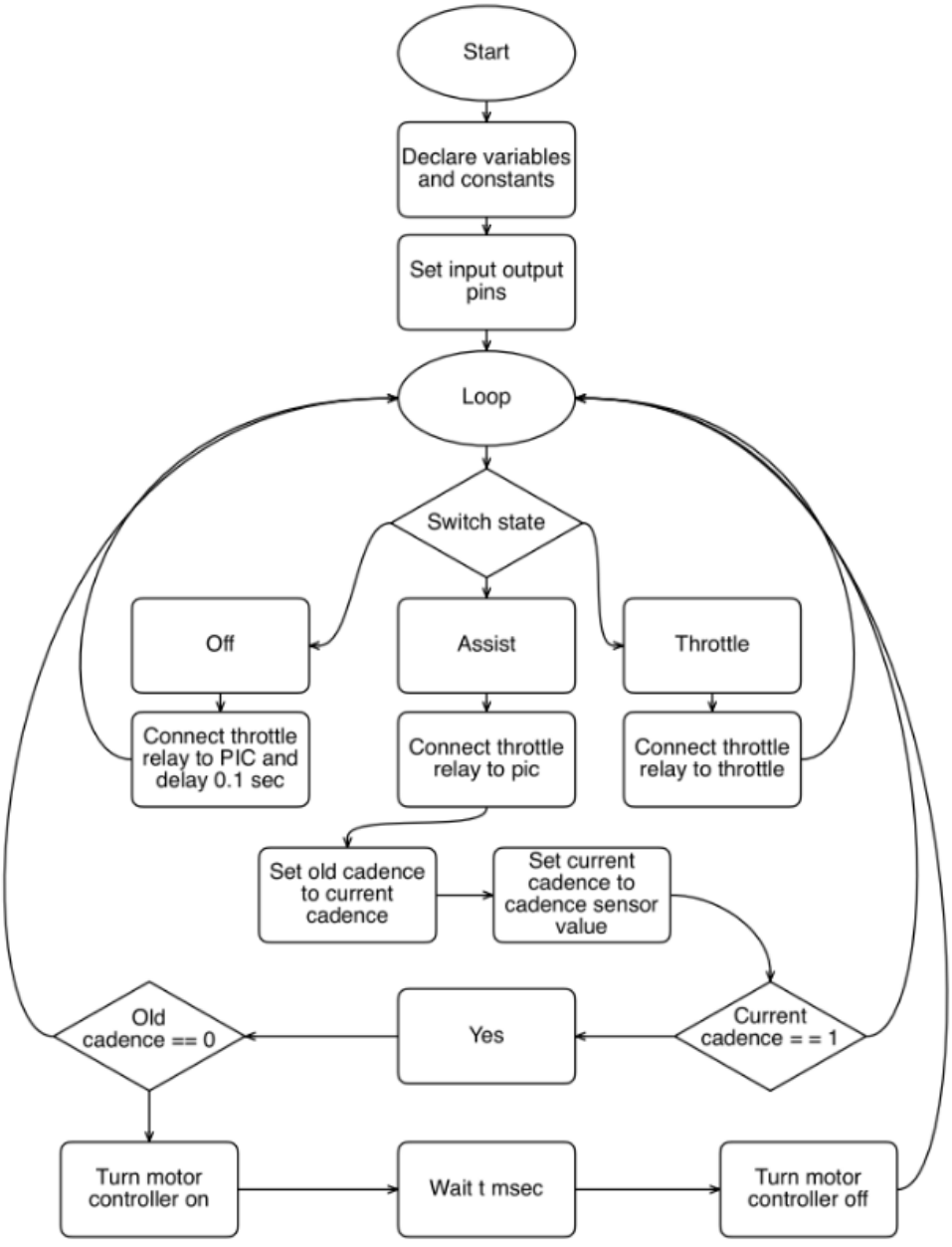
Code Flow Charts

Note: These flowcharts are slightly outdated and only provide a cursory representation of the logic at play. Actual code, which follows, should be examined for a full understanding.

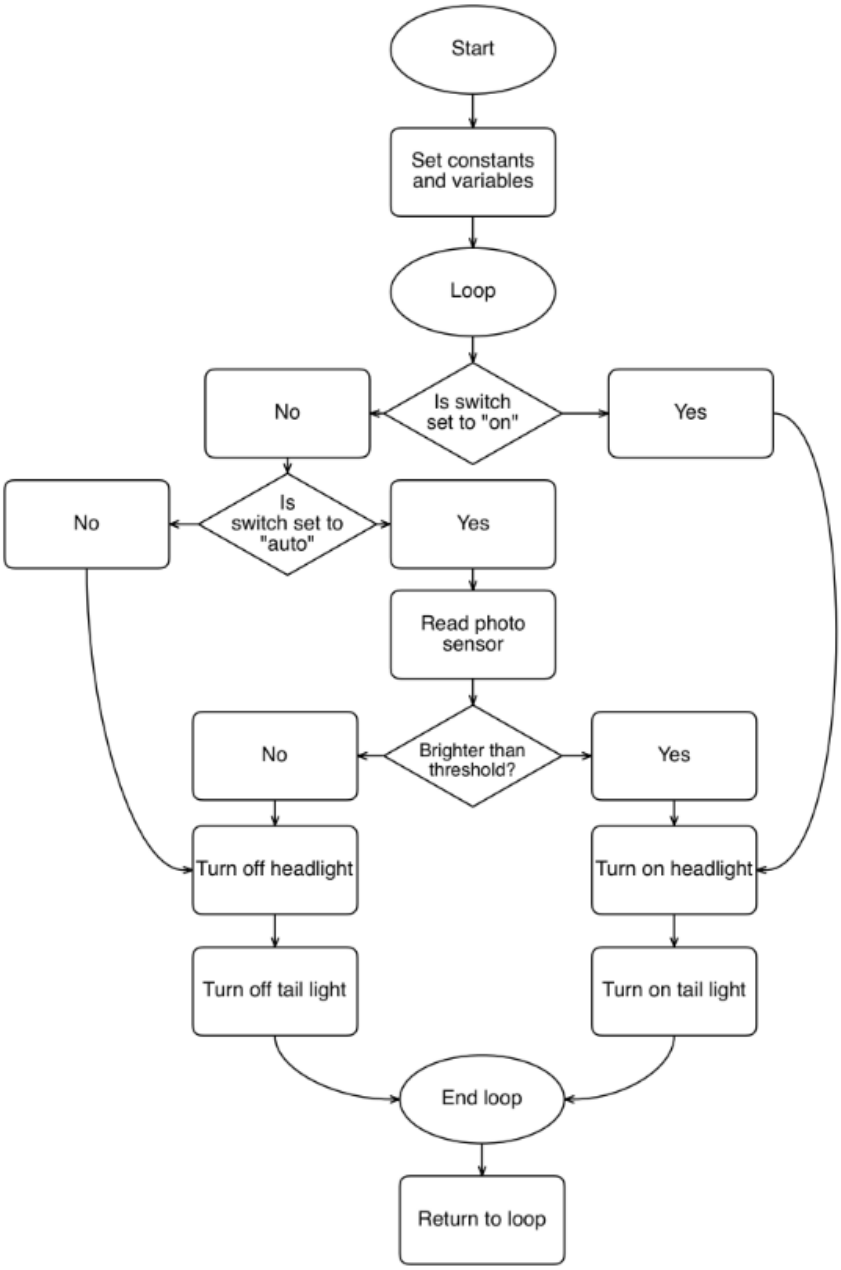
Phone App



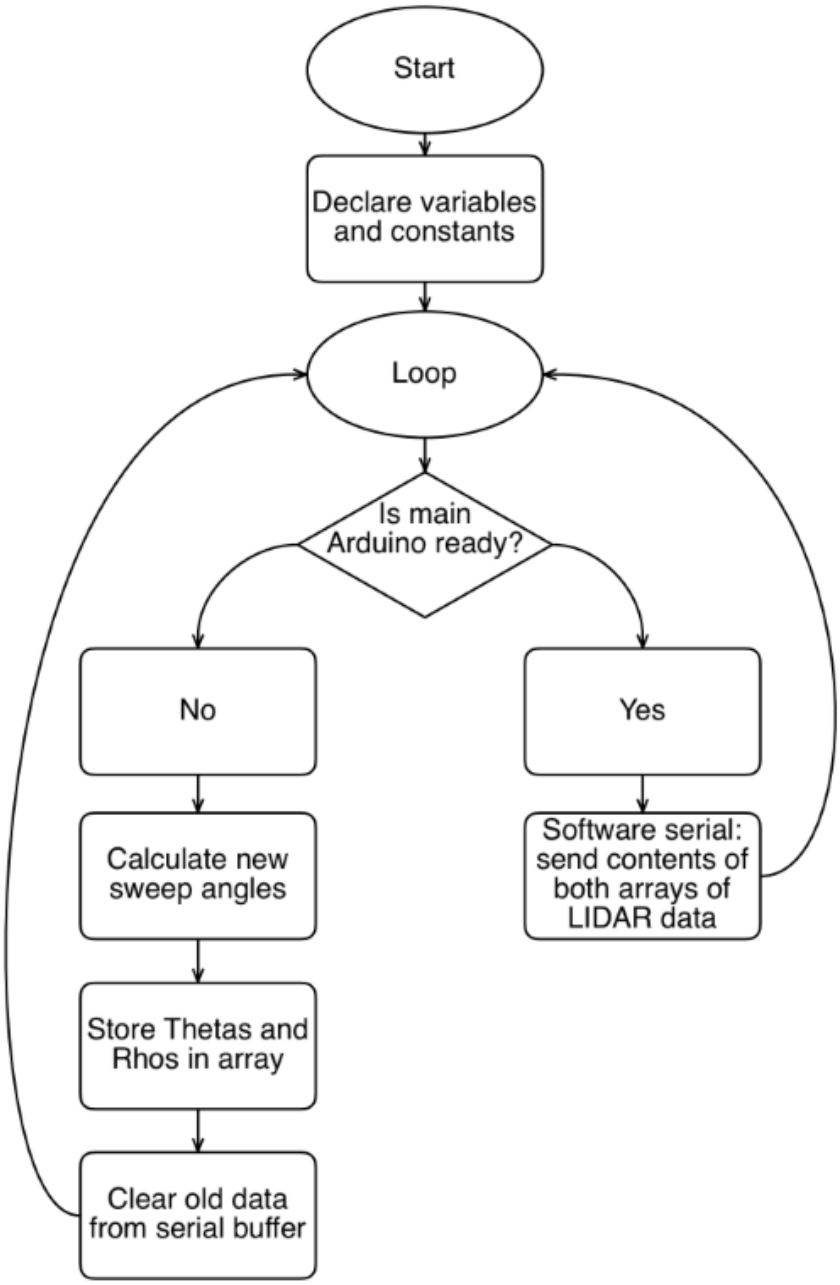
Throttle PIC



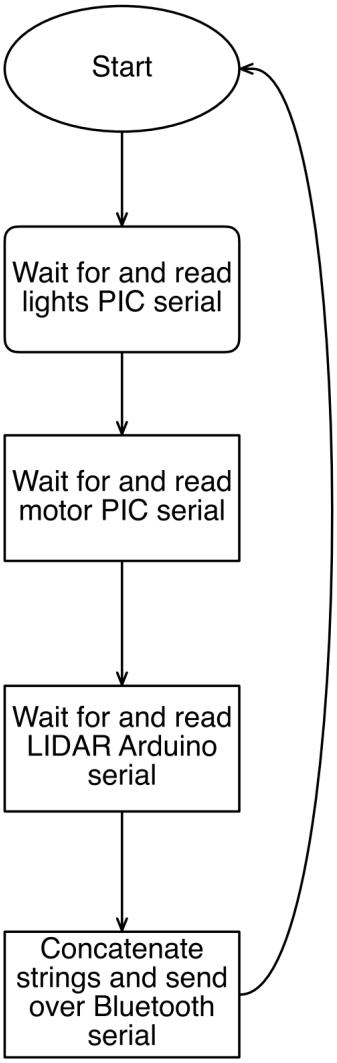
Headlight and Taillight PIC



LIDAR's Secondary Arduino



Main, Coordination Arduino



```
'/*
' *    Uses modified "PIC16F88 code template for MECH307 Labs"
' *    Template:  http://www.engr.colostate.edu/~dga/mech307/handouts/PIC16F88_template.bas
' */

#CONFIG
    __CONFIG _CONFIG1, _INTRC_IO & _PWRTE_ON & _MCLR_OFF & _LVP_OFF
#ENDCONFIG

' Set the internal oscillator frequency to 8 MHz
DEFINE OSC 8
OSCCON.4 = 1
OSCCON.5 = 1
OSCCON.6 = 1

' Turn off A/D converter
ANSEL=0

'/*
' * Copyright Chris Sawyer, 2016
' * License: The following original code can be freely used for educational purposes; any other use is forbidden.
' *          Referenced libraries are covered under their original licenses.
' */

TRISA = %00001000 ' 1=in 0=out
TRISB = %00000011

OUT_RELAY_CONTROL var PortA.0
OUT_HEADLIGHT_SIGNAL var PortA.1
OUT_TAILLIGHT_SIGNAL var PortA.2
IN_PHOTORESISTOR var PortA.3
IN_LIGHTS_ON_SWITCH var PortB.1
IN_LIGHTS_AUTO_SWITCH var PortB.0
OUT_SERIAL var PortB.2
baud_rate Con 2 ' 9600 baud-rate mode for serial communication, assuming 8 MHz

' set all ouput pins low on boot
low OUT_RELAY_CONTROL
low OUT_HEADLIGHT_SIGNAL
low OUT_TAILLIGHT_SIGNAL
low OUT_SERIAL


char var byte 'tmp byte to hold chars for seding over serial

_ON con 0
_OFF con 1
_AUTO con 2
```

```
_THRESHHOLD con 127
_PHOTORESISTOR_SCALE con 255
_LIGHTDELAY con 50 'ms to wait between NEG edge pulses for light state

lightButtonState var byte
internalLightState var byte
photoResistorValue var byte


lightButtonState = _OFF
internalLightState = _OFF


myloop:

    if (IN_LIGHTS_ON_SWITCH == 1) then
        ' Turn ON lights
        lightButtonState = _ON
        goSub turnLightsON

    elseif (IN_LIGHTS_AUTO_SWITCH == 1) then
        ' Read photoresistor then turn on or turn off lights
        lightButtonState = _AUTO
        'goSub handleAutoLights

        gosub handleAutoLights

    else
        ' Turn OFF lights
        lightButtonState = _OFF
        goSub turnLightsOff
    endif

    gosub sendSerial

goto myloop


''' FUNCTIONS ''''''''''''''''''''''''

turnLightsOn:
    if (internalLightState == _ON) then

        return 'change nothing if this is already the correct state

    else
        internalLightState = _ON

        ' turn ON the lights power relay
```

```
        HIGH OUT_RELAY_CONTROL
        pause(50)
        ' pulse the light signal lines once to "turn them on"
        low OUT_HEADLIGHT_SIGNAL
        low OUT_TAILLIGHT_SIGNAL
        pause(50)
        high OUT_HEADLIGHT_SIGNAL
        high OUT_TAILLIGHT_SIGNAL
    endif
return



turnLightsOff:
    ' turn OFF the lights power relay
    if (internalLightState == _OFF) then
        return 'change nothing if this is already the correct state
    else
        internalLightState = _OFF
        low OUT_HEADLIGHT_SIGNAL
        low OUT_TAILLIGHT_SIGNAL
        low OUT_RELAY_CONTROL
    endif
return



handleAutoLights:

    POT IN_PHOTORESISTOR, _PHOTORESISTOR_SCALE, photoResistorValue
    if (photoResistorValue > _THRESHHOLD) then
        gosub turnLightsOn
    else
        gosub turnLightsOff
    endif

return



sendSerial:

    LookUp 0,["["],char
    Serout OUT_SERIAL, baud_rate, [char]

    Serout OUT_SERIAL, baud_rate, [#lightButtonState]
return
```

```
'/*
' *    Uses modified "PIC16F88 code template for MECH307 Labs"
' *    Template:  http://www.engr.colostate.edu/~dga/mech307/handouts/PIC16F88_template.bas
' */

#CONFIG
    __CONFIG _CONFIG1, _INTRC_IO & _PWRTE_ON & _MCLR_OFF & _LVP_OFF
#ENDCONFIG

' Set the internal oscillator frequency to 8 MHz
DEFINE OSC 8
OSCCON.4 = 1
OSCCON.5 = 1
OSCCON.6 = 1

'/*
' * Copyright Chris Sawyer, 2016
' * License: The following original code can be freely used for educational purposes; any other use is forbidden.
' *          Referenced libraries are covered under their original licenses.
' */

TRISA = %00000100 ' 1=in 0=out
TRISB = %01110111

' Turn on A/D for the throttle
ANSEL=0
ANSEL.5 = 1 ' throttle is AN5
ADCON1.7 = 1 ' have the 10 bits be right-justified
DEFINE ADC_BITS 10 ' AN5 is a 10-bit A/D
AD_word Var WORD ' word from the A/D converter (10 bits padded with 6 0's)
AD_scaled Var BYTE


OUT_RELAY_CONTROL var PortA.0
OUT_PWM_OUTPUT var PortA.1
IN_FLIPFLOP2_OUTPUT var PortA.2
OUT_FLIPFLOP2_RESET var PortA.3
OUT_SERIAL var PortA.4
baud_rate Con 2 ' 9600 baud-rate mode for serial communication
IN_FLIPFLOP1_OUTPUT var PortB.0
IN_CADENCE1_STATUS var PortB.1
IN_CADENCE2_STATUS var PortB.2
IN_MOTOR_MODE_ASSISTIVE var PortB.4
IN_MOTOR_MODE_THROTTLE var PortB.5
IN_THROTTLE var PortB.6

' set all ouput pins low on boot
low OUT_RELAY_CONTROL
low OUT_PWM_OUTPUT
```

```
low OUT_FLIPFLOP2_RESET
low OUT_SERIAL


_THROTTLE con 0
_OFF con 1
_ASSIST con 2

throttleButtonState var byte
oldCadence var byte

' These are used in the PWM easing loop
' in MICRO-seconds
totalTimeUS  var byte
pwmTimeUS var byte
pwmTimeTotalUS var byte
dummyLoop var byte

char var byte ' holds character to send over serial


myloop:

    if (IN_MOTOR_MODE_THROTTLE == 1) then
        throttleButtonState = _THROTTLE

        gosub connectThrottleToMotor

        ' measure throttle voltage
        ADCIN 5, ad_word ' throttle is AN5
        ad_scaled = ad_word/4   ' convert pointlesly huge word to byte

    elseif (IN_MOTOR_MODE_ASSISTIVE == 1) then
        ad_scaled = 0
        throttleButtonState = _ASSIST
        gosub controlMotorWithCadence

    else
        ad_scaled = 0
        throttleButtonState = _OFF
        gosub disconnectThrottleFromMotor

    endif


    gosub sendSerial

goto myloop
```

```
controlMotorWithCadence:

    gosub disconnectThrottleFromMotor

    ' If the cadence sensor sees a positive edge
    if ((oldCadence == 0) && (IN_CADENCE2_STATUS == 1)) then
        ' 15 minimum
        ' ramp up
        for pwmTimeUS = 0 to 19
            high OUT_PWM_OUTPUT
            pause pwmTimeUS
            low  OUT_PWM_OUTPUT
            pause pwmTimeUS
        next pwmTimeUS

        ' hold
        for dummyLoop = 0 to 4 '4
            high OUT_PWM_OUTPUT
            pause pwmTimeUS
            low  OUT_PWM_OUTPUT
            pause pwmTimeUS
        next dummyLoop

    endif

    oldCadence = IN_CADENCE2_STATUS

return

connectThrottleToMotor:
    ' Connect the throttle voltage to the motor signal line with
    '   the relay (this also disconnects it from the PIC's OUT_PWM)
    '   Also turn the PWM control line low so it doesn't start moving
    '   the motor when the switch is put into a PIC-controlled mode
    low OUT_PWM_OUTPUT
    low OUT_RELAY_CONTROL
return



disconnectThrottleFromMotor:
    ' Disconnect the throttle voltage from the motor signal line with
    '   the relay (this also connects it to the PIC's OUT_PWM)
    '   Then, turn the PWM control line low so we don't move the motor
    low OUT_PWM_OUTPUT
    high OUT_RELAY_CONTROL
return
```

```
sendSerial:
    LookUp 0,["["],char
    Serout OUT_SERIAL, baud_rate, [char]

    Serout OUT_SERIAL, baud_rate, [#ad_scaled]'tmp not AD_scaled

    LookUp 0,[","],char
    Serout OUT_SERIAL, baud_rate, [char]

    Serout OUT_SERIAL, baud_rate, [#throttleButtonState]
return
```

```
/*
* Copyright Chris Sawyer, 2016
* License: The following original code can be freely used for educational purposes; any other use is forbidden.
*          Referenced libraries are covered under their original licenses.
*/

#include <SoftwareSerial.h>

#define ArduinoBAUD 115200

//// PINS /////////////////////////////////////////////
#define triggerPin 2
#define monitorPin 3
#define rxMainArduino 0
#define txMainArduino 1
#define stepper555Power 13
#define stepperStep 12
///////////////////////////////////////////////////////

unsigned long pulseWidth = 0;

long loopStartTime = 0;
long loopEndTime = 0;

unsigned long distance = 0;
long degreesSwept = 0;

String toPrint = "[";

void setup() {
        Serial.begin(ArduinoBAUD);

        pinMode(triggerPin, OUTPUT);
        digitalWrite(triggerPin, LOW);
        pinMode(monitorPin, INPUT);

        pinMode(stepper555Power, OUTPUT);
        pinMode(stepperStep, OUTPUT);
        digitalWrite(stepper555Power, HIGH); //Spin the motor
        digitalWrite(stepperStep, LOW);

}
void loop() {

        distance = GetDistance();

        loopEndTime = millis();
        degreesSwept = RPS * 360 * (loopEndTime - loopStartTime) / 1000.0;
        loopStartTime = millis();
```

```
        // already has "["
        toPrint += "|";
        toPrint += degreesSwept;
        toPrint += ",";
        toPrint += distance;

        delay(50);
        CheckAskedForData(); // to possibly send

}

unsigned long GetDistance() {
        // This function uses Garmin's example code
        // http://static.garmin.com/pumac/LIDAR_Lite_v3_Operation_Manual_and_Technical_Specifications.pdf

        pulseWidth = pulseIn(monitorPin, HIGH); // Count how long the pulse is high in microseconds
        pulseWidth = pulseWidth / 10; // 10usec = 1 cm of distance
        return pulseWidth; // Print the distance
}

void CheckAskedForData() {
        if (Serial.available() > 0) {

                // We're waiting to read an 'a' to trigger the data sending
                if (((char)Serial.read()) == 'a') {

                        ////// Clear the serial buffer
                        while (Serial.available()) {  //is there anything to read?
                                char getData = Serial.read();  //if yes, read it
                        }   // but don't do anything with it.
                        //////

                        // Send the data
                        Serial.println(toPrint);
                        toPrint = "[";
                }
        }
}
```

```
/*
* Copyright Chris Sawyer, 2016
* License: The following original code can be freely used for educational purposes; any other use is forbidden.
*          Referenced libraries are covered under their original licenses.
*/

#include <SoftwareSerial.h>

#define BluetoothBAUD 115200
#define ArduinoBAUD 115200
#define PicBAUD 9600

//// PINS ////////////////////////////////////////////
#define rxBluetooth 12
#define txBluetooth 13
#define rxLightsPic 11
#define txLightsPic 10 // no connection, just for SoftwareSerial.h
#define rxThrottlePic 9
#define txThrottlePic 8 // no connection, just for SoftwareSerial.h
#define rxLidarArduino 0
#define txLidarArduino 1
#define rxSpeedCadenceSensor 2
#define pwrSpeedCadenceSensor 3
// All of these are SoftwareSerial except for the cadence, which is either an interrupt or hardware serial depending on
//    if interrupts are too slow and break softwareserial
// interrupt: use pins 2 or 3
// serial: use  RX pin 0
// https://www.arduino.cc/en/Reference/AttachInterrupt
//////////////////////////////////////////////////////

// only one of these can run a at a time!
SoftwareSerial bluetoothSerial(rxBluetooth, txBluetooth); //RX, TX
SoftwareSerial lidarArduinoSerial(rxLidarArduino, txLidarArduino);
SoftwareSerial lightPicSerial(rxLightsPic, txLightsPic);
SoftwareSerial throttlePicSerial(rxThrottlePic, txThrottlePic);


char lastRead, lastLastRead;
String serialBuffer;
bool gotFirstBracket, gotEndBracket;
int dummyLidarDistance = 400;
String strReturn;

bool readRPM = false;
long timeS, timeE;

void setup() {
        bluetoothSerial.begin(BluetoothBAUD);
        Serial.begin(ArduinoBAUD);
```

```
        pinMode(pwrSpeedCadenceSensor, OUTPUT);
        pinMode(rxSpeedCadenceSensor, INPUT);
        digitalWrite(pwrSpeedCadenceSensor, HIGH);
        attachInterrupt(digitalPinToInterrupt(rxSpeedCadenceSensor), rpm, RISING);
        timeS = millis();
}


void rpm() {
        readRPM = true;
        timeE = millis();
}

void loop() {

        //'serpator start lidar lidar lidar throttle lights cadence end
        //'_-[|16,120|30,100|25,160[100,1[2[45,200+

        ////////////////////////////////////////////////////////
        // Gather all information and send over bluetooth serial
        ////////////////////////////////////////////////////////

        // Start packet
        bluetoothSerial.print("_-");

        strReturn = ReadLidarFromSerial();
        bluetoothSerial.print(strReturn);


        // Send throttle: "[255Position,STATE"
        strReturn = WaitForPicPacket(throttlePicSerial);
        bluetoothSerial.print(strReturn);

        // Send lights: "[STATE"
        strReturn = WaitForPicPacket(lightPicSerial);
        bluetoothSerial.print(strReturn);

        // Send cadence: "[numberWheelRevs,timeMsSinceLastMeasurement"
        if (readRPM) {
                readRPM = false;
                bluetoothSerial.print("[");
                bluetoothSerial.print("1");
                bluetoothSerial.print(",");
                bluetoothSerial.print((timeS - timeE));
                timeS = millis();
        }
        else {
                bluetoothSerial.print("[0,1");
        }
```

```
        // End packet
        bluetoothSerial.println("+");
        bluetoothSerial.end();
}

String leftoverSerial = "";
String partialPacket = "";
String serialString = "";
bool reading = false;

String ReadLidarFromSerial() {
        serialString = "";
        Serial.println('a');
        while (Serial.available() == 0) {} // wait for serial
        if (Serial.available() > 0) {
                serialString = Serial.readStringUntil('\n');
                serialString.remove(serialString.length() - 1);//remove the '\n' char from the end of the string
        }
        return serialString;
}



String WaitForPicPacket(SoftwareSerial serialConnection) {
        serialBuffer = "";

        gotFirstBracket = false;
        gotEndBracket = false;

        bluetoothSerial.end();
        serialConnection.begin(PicBAUD);

        // wait until we read a bracket, signifiying the start of a packet
        while (!gotFirstBracket) {
                if (serialConnection.available()) {
                        lastRead = (char)serialConnection.read();
                        if (lastRead == '[') {
                                gotFirstBracket = true;
                        }
                }
        }

        serialBuffer = "[";

        // read until we read another bracket, signifiying the start of the next packet and the end of this packet
        while (!gotEndBracket) {
                if (serialConnection.available()) {
                        lastRead = (char)serialConnection.read();
```

```
                        if (lastRead == '[') {
                                gotEndBracket = true;
                        }
                        else {
                                // Don't want to add an extra '[' on the end of the packet. That's the start of the next one!
                                serialBuffer += lastRead;
                        }
                }
        }

        serialConnection.end();
        bluetoothSerial.begin(BluetoothBAUD);

        return serialBuffer;
}
```

```
/*
* Copyright Chris Sawyer, 2016
* License: The following original code can be freely used for educational purposes; any other use is forbidden.
*          Referenced libraries are covered under their original licenses.
*/

using UnityEngine;
using System.Collections;
using System;
using System.Collections.Generic;

using UnityEngine.UI;

public class ParseStream : MonoBehaviour
{

    /*
    STREAM FORMAT:

        serpator start [|lidar |lidar |lidar [throttle [lights [cadence end
        public string incoming = "_-[|16,120|30,100|25,160[100,1[2[45,200+";

    SUBSTRING FORMATS:

        LIDAR, from 2nd Arduino
        [|angleChange,distance|angleChange,distange.....

        throttle, from PIC1
        [throttle,switch
            THROTTLE:   min:40    max:220
            SWITCH:     0   THROTTLE
                        1   OFF
                        2   ASSIST

        lights, from PIC2
        [switch
            SWITCH:    0   on
                       1   off
                       2   auto

        cadence, from Main Arduino
        [wheel revs, time ms since last count
    */

    public UpdateUiElements updateUI;

    const string SEPERATOR = "_";
    const string START = "-";
    const string END = "+";
```

```
    const string SOURCE_SEPERATR = "["; // between sources/chunks of information
    const string LIDAR_SEPERATOR = "|"; // to differentiate couple pairs in the lidar chunk
    const string INNERSEPERATOR = ","; // to differentiate couple elements

    public string notepad = "_-[|16,120|30,100|25,160[100,1[2[45,200+"; // For Debug in the Unity Editor

    public string incoming = "";
    public string leftOverString = "";

    public List<int> lidarTheta = new List<int>();
    public List<int> lidarRadius = new List<int>();

    public int throttle = 0;
    public int throttleSwitchPosition = 0;
    public int lightsSwitchPosition = 0;
    public int cadence = 0;
    public int cadenceTime = 0;

    public bool needUpdate = false;

    void Update()
    {
        if (needUpdate || (incoming.Length!=0))
        {
            Parse();
            updateUI.UpdateUI();
            needUpdate = false;
        }
    }

    public bool packetHasBothEnds, packetEndsOrder;

    private void Parse()
    {
        // if there was part of a new packet at the end of the last incoming string, add it before the new string to complete the packet
        // We can just grab whatever is in the Bluetooth serial buffer and parse the complete packets, but save the last bit for the next loop.
        if (leftOverString.Length != 0)
        {
            incoming = leftOverString + incoming;
            leftOverString = "";
        }

        string[] sections = incoming.Split(new string[] { SEPERATOR }, StringSplitOptions.RemoveEmptyEntries);

        foreach (string s in sections)
        {
            // Make sure this substring has both start and end strings, "-" and "+"
            packetHasBothEnds = (s.Contains(START) && s.Contains(END));
            packetEndsOrder = (s.IndexOf(START) < s.IndexOf(END));
```

```
            if ((packetHasBothEnds) && (packetEndsOrder))
            {
                int start = s.IndexOf(START);
                int end = s.IndexOf(END);

                string chunk = s.Substring(start + 1, end - start - 1);

                string[] sources = chunk.Split(new string[] { SOURCE_SEPERATR }, StringSplitOptions.RemoveEmptyEntries);

                string[] lidarData = sources[0].Split(new string[] { LIDAR_SEPERATOR }, StringSplitOptions.RemoveEmptyEntries);
                foreach (string l in lidarData)
                {
                    int commaIndex = l.IndexOf(INNERSEPERATOR);
                    lidarTheta.Add(int.Parse(l.Remove(commaIndex)));
                    lidarRadius.Add(int.Parse(l.Remove(0, commaIndex + 1)));
                }

                int commaIndexThrottle = sources[1].IndexOf(INNERSEPERATOR);

                throttle = int.Parse(sources[1].Remove(commaIndexThrottle));
                throttleSwitchPosition = int.Parse(sources[1].Remove(0, commaIndexThrottle + 1));

                lightsSwitchPosition = int.Parse(sources[2]);

                int commaIndexCadence = sources[3].IndexOf(INNERSEPERATOR);
                cadence = int.Parse(sources[3].Remove(commaIndexCadence));
                cadenceTime = int.Parse(sources[3].Remove(0, commaIndexCadence + 1));
            }
            else
            {
                if (s.Contains(START) && !s.Contains(END)) // if this is the BEGGINING of an incomplete {} section
                {
                    leftOverString += s; // save it for the next loop
                }
            }

        }

        incoming = "";
    }

}
```

```
/*
* Copyright Chris Sawyer, 2016
* License: The following original code can be freely used for educational purposes; any other use is forbidden.
*          Referenced libraries are covered under their original licenses.
*/

using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class DrawLidar : MonoBehaviour {

    public ObjectPool pool;

    public GameObject test1;
    public float x, y, w, h;

    void Start () {
        x = GetComponent<RectTransform>().position.x;
        y = GetComponent<RectTransform>().position.y;
        w = GetComponent<RectTransform>().rect.width;
        h = GetComponent<RectTransform>().rect.height;
    }

    private List<GameObject> nonPooledPrefabs = new List<GameObject>();
    private GameObject tmpGameObject;
    private LidarArcData tmpLidarArcData;

    private int currentStartAngle = 0;
    private int tmpAngle;
    private int tmpRadius;
    private int lengthOfDataAtTimeOfNewAdd;

    public void DrawNewArc(int radius, int sweepAngle)
    {
        // Get arc from pool
        GameObject arc = GetArc();
        nonPooledPrefabs.Add(arc);

        tmpAngle = currentStartAngle + sweepAngle;
        while (tmpAngle >= 360)
            tmpAngle -= 360;


        // Give the arc prefab its properties
        arc.transform.position = new Vector3(x, y, 0);
        arc.GetComponent<Draw_CircularArc>().DeltaStart = currentStartAngle;
        arc.GetComponent<Draw_CircularArc>().DeltaSize = sweepAngle;
```

```
        tmpRadius = (414 * radius / 400); //426 pixels for full length, 0-400cm range
        if (tmpRadius > 414)
            tmpRadius = 414;

        arc.GetComponent<Draw_CircularArc>().OuterRadius = tmpRadius;
        arc.GetComponent<Draw_CircularArc>().UpdateMesh();

        // Store the prefab's geometry data
        arc.GetComponent<LidarArcData>().startAngle = currentStartAngle;
        arc.GetComponent<LidarArcData>().sweepAngle = sweepAngle;
        arc.GetComponent<LidarArcData>().radius = tmpRadius;
        arc.GetComponent<LidarArcData>().life = 1f/2.2f; //2.2 RPS
        arc.GetComponent<LidarArcData>().drawLidarListIndex = nonPooledPrefabs.Count;

        // Check to see if this arc is overlapping any of the others, and if so return them to the pool
        // There is also a timer on each arc that deletes the arc if it exists for longer than the
        //   LIDAR would physically take to rotate back to its start position.
        for ( int i=0; i< nonPooledPrefabs.Count; i++)
        {

            tmpGameObject = nonPooledPrefabs[i];
            if (tmpGameObject != null)
            {
                tmpLidarArcData = tmpGameObject.GetComponent<LidarArcData>();
                float otherStartAngle = tmpLidarArcData.startAngle;
                float otherEndAngle = otherStartAngle + tmpLidarArcData.sweepAngle;

                int newStartAngle = currentStartAngle;
                int newEndAngle = newStartAngle + sweepAngle;

                // Normalize to [0...360]
                while (otherEndAngle >= 360)
                    otherEndAngle -= 360;

                while (newEndAngle >= 360)
                    newEndAngle -= 360;

                // This is the actual overlap checking and timer logic
                if (
                    (
                        ((newStartAngle >= otherEndAngle) && (newStartAngle < otherEndAngle))
                        ||
                        ((newEndAngle >= otherStartAngle) && (newEndAngle < otherEndAngle))
                    )
                    ||
                    (tmpLidarArcData.life<=0)

                )
                {
```

```
                    RemoveArc(nonPooledPrefabs[i]);
                    nonPooledPrefabs.RemoveAt(i--);
                }

            }
        }

        currentStartAngle += sweepAngle;
        while (currentStartAngle >= 360)
            currentStartAngle -= 360;
    }

    private GameObject GetArc()
    {
        return pool.GetObjectForType("LidarArc", true);
    }

    private void RemoveArc(GameObject arc)
    {
        pool.PoolObject(arc);
    }
}
```

```
/*
* Copyright Chris Sawyer, 2016
* License: The following original code can be freely used for educational purposes; any other use is forbidden.
*          Referenced libraries are covered under their original licenses.
*/

using UnityEngine;
using System.Collections;
using UnityEngine.UI;

public class UpdateUiElements : MonoBehaviour {

    public ParseStream stream;
    public Text textSpeed, lightsSwitch, throttleSwitch;
    public RectTransform throttleBar;
    public DrawLidar lidar;
    public AudioSource lightOff, lightOn, lightAuto, motorOff, motorOn, motorAssist;

    // BIKE CONSTANTS
    // speed [mi/hr] = rotations*sec circumfrence/sec  *  ft/inch * mi/ft  * sec/min * min/hr   =   2*pi*r * 1/12*1/5280 * 60*60
    float BIKE_TIRE_RADIUS = 23; //in inches

    string[] LIGHT_MODES = { "ON", "OFF", "AUTO"};
    string[] THROTTLE_MODES = { "THROTTLE", "OFF", "ASSIST" };

    int oldSL = -1, oldST = -1;

    public void UpdateUI()
    {

        while (stream.lidarRadius.Count > 0)
        {
            lidar.DrawNewArc(stream.lidarRadius[0], stream.lidarTheta[0]);
            stream.lidarRadius.RemoveAt(0);
            stream.lidarTheta.RemoveAt(0);
        }

        float speed = stream.cadence / (stream.cadenceTime / 1000f) * (2 * Mathf.PI * BIKE_TIRE_RADIUS * 1 / 12f * 1 / 5280f) * (60 * 60);
        textSpeed.text = string.Format("{0:0.00}", speed);

        // Take throttle voltage and scale the readout to match
        // normlalize to Throttle's weird voltage range. (min:47, max:215)
        // normalize new range from 1 to (215-47+1)=169, with the 1 to keep it visible at 0
        float rangeTop = 220; // constants determined from experimentation with our specific throttle
        float rangeBottom = 40;
        float throttlePercent = (stream.throttle - rangeBottom) / (rangeTop - rangeBottom + 1);
        throttleBar.anchorMax = new Vector2(throttlePercent, 1);

        lightsSwitch.text = LIGHT_MODES[stream.lightsSwitchPosition];
```

```
        throttleSwitch.text = THROTTLE_MODES[stream.throttleSwitchPosition];

        if (oldSL == -1)
        {
            // First loop, need to set dummy data
            oldSL = stream.lightsSwitchPosition;
            oldST = stream.throttleSwitchPosition;
        }

        if (oldSL != stream.lightsSwitchPosition)
        {
            switch (stream.lightsSwitchPosition)
            {
                case 0:
                    lightOn.Play();
                    break;
                case 1:
                    lightOff.Play();
                    break;
                case 2:
                    lightAuto.Play();
                    break;
            }
        }
        if (oldST != stream.throttleSwitchPosition)
        {

            switch (stream.throttleSwitchPosition)
            {
                case 0:
                    motorOn.Play();
                    break;
                case 1:
                    motorOff.Play();
                    break;
                case 2:
                    motorAssist.Play();
                    break;
            }

        }

        oldSL = stream.lightsSwitchPosition;
        oldST = stream.throttleSwitchPosition;
    }

}
```