2017 Mercury Remote Robotics Competition

Team 4b – Final Report

Joseph Austin

Jonathan Ballew

Kamran Coulter

Kyle Edwards

The purpose of this document is to describe and justify the design of team Catch-a-Ride's Mercury Robotics Competition robot. The primary goal of this project was to design a mobile robot that could complete the 2017 Mercury Remote Robotics competition track and all objectives with minimal fault. The robot was designed as a capstone project for ECEN 4024. This document details the design and implementation of the motion platform, manipulator arm and effector, electronics hardware and power distribution, and the control and sensor feedback hardware and software. This document describes the extent to which the design met the project specifications and offers a critique of any systems that failed to meet specifications or could be improved.

ECEN 4024 – Capstone Design

Oklahoma State University

5/5/2017

Table of Contents

2017 Mercury Remote Robotics Competition

Team 4b – Final Report

1 Introduction

Written By: Joseph Austin

The goal of this project was to design a mobile robot that was able to compete in the 2017 Mercury Remote Robotics Challenge and complete every objective of the course with minimal fault. The robot was designed as a capstone design project for ECEN 4024. The robot needed to be able to navigate a dark tunnel, secure a lag bolt, navigate a slalom and 30 degree inclined ramp, deposit the lag bolt, and complete a timed sprint, all with minimal obstacle collision. The ability of the robot to complete these objectives without obstacle collision determines the team's score in the competition.

Moreover, as a capstone design project, the design was expected to be unique and go beyond minimal functionality to compete. The robot was expected to demonstrate careful consideration of design constraints and construction techniques appropriate for a team of graduating students in electrical engineering. The system needed to fully integrate a motion platform, any sensing and imaging, and control hardware as opposed to loosely-connected subsystems. Lastly, the operation of the robot needed to be performed through an integrated user interface with careful attention to usability and human factors.

The robot's design is broken down into four primary subsystems:

- Motion platform and control hardware
- The manipulator and end effector
- Power distribution and regulation
- Communications and control software

This document first describes the background information and rules of the competition. This document then details and justifies the mechanical design, electronics hardware, power distribution systems, and control and communications software implemented in the design of the robot.

1.1 Competition background

Written By: Kamran Coulter

The Mercury Remote Robotics Challenge is an international robotics competition hosted at Oklahoma State University. This year's competition was the 8th annual competition, and teams assembled from the United States, Colombia, Brazil, and Mexico. The competition was founded by faculty and students from Oklahoma State University with the hope that students could learn complex robotic systems by designing mobile robots capable of completing a wide range of objectives. The operating distance and internet connectivity requirements of the challenge impose many engineering challenges on the design of a competition robot. Minimizing communication and control latency and implementing responsive mechanical systems are of utmost importance during the design phase as teams attempt to build a motion platform that can be controlled from a minimum distance of 50 miles via the Internet. The competition attempts to simulate some of the challenges engineers face when designing systems to explore other planets. [1]

2 Competition Details – 2017

Written By: Kamran Coulter

The Mercury Remote Robotics Challenge course changes each year, but each year robots must be able to navigate a course and manipulate a payload in some fashion. The following sections give background information on the 2017 competition regarding the course, its objectives, and its scoring system.

2.1 Competition Track

Written By: Kamran Coulter

The 2017 competition track is shown in Figure 1. The track features six distinct challenges: the tunnel, payload securing, slalom, see-saw, payload depositing, and a timed sprint. The score for each robot is calculated from how well the robot is able to complete each challenge, which is described in the Competition Rules section to follow. The track is constructed using three-inch-tall pieces of flexible foam board taped 24 inches apart to a carpeted surface in the Noble Research Center (NRC) at Oklahoma State University. The robots in this year's competition start in front of the tunnel, which is 18 inches wide by 12 inches tall. The tunnel imposes the challenge of fitting the robot in a narrow enclosure and navigating it in a dark environment. After the tunnel, a team's robot must make a sharp right turn into the payload pick-up zone, where a 3 inch lag bolt is placed in whichever orientation the team desires. Once the bolt is secure, the robot needs to follow a smooth curved section to the slalom consisting of two pylons separated by 18 inches. The robot must navigate the slalom by following directional arrows taped to the ground.

Following the slalom, a smooth curve brings the robot to the seesaw. The operator has the choice to either to go over the 30 degree inclined seesaw or bypass it entirely. The steeply inclined seesaw is constructed from smooth plywood, making it challenging to both ascend and descend without dropping the payload. The robot then needs to deposit the bolt in the payload drop-off zone: an 8-inch long face that sits 6.2 inches off of the ground at a 45 degree angle. The face had three holes of increasing diameter from 0.5 to 2 inches. Lastly, the robot completes the course by driving a laser-timed sprint section to the finish. [1]

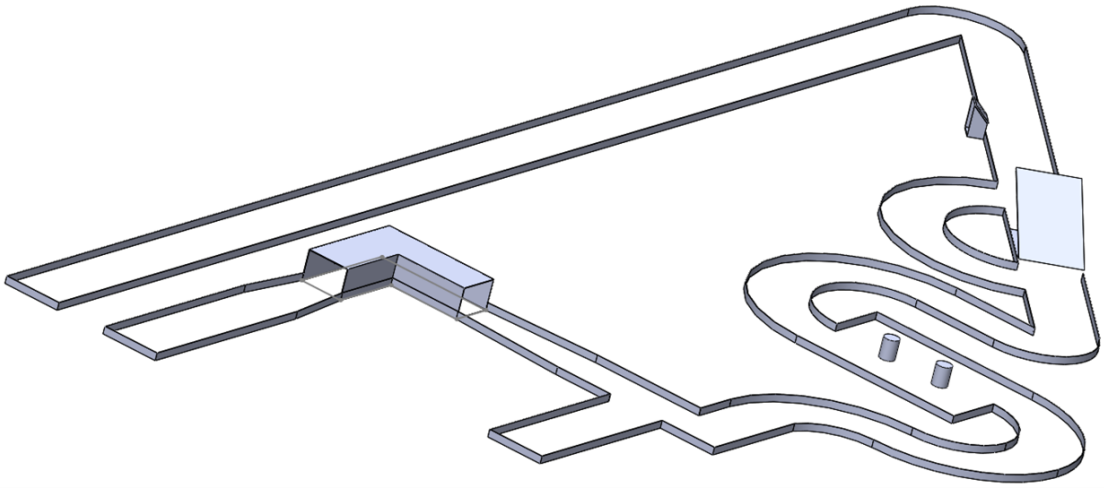


Figure 1: 2017 Mercury Competition Track

2.2 Competition Network

Written By: Joseph Austin

The competition rules specify that the operator must drive the robot over the internet from a location of at least 50 miles away. Each team is allowed a maximum of three communication ports, each of which is forwarded on the competition router. Each team is also allowed one IP camera. On the day of the competition, the team connects their robot to competition router, and the operator connects to the static IP of the competition router over the internet.

To qualify for first place, the robot must pass a loss-of-signal test. The robot must demonstrate that when connection is lost with the client, the robot halts operation and waits for the client to reconnect. The robot must have some type of external feedback to indicate that the connection is lost (an LED, a buzzer, etc.), and the robot must be able to reconnect and resume operation without any intervention by the robot handler. The test is performed as follows:

1. The operator is asked to connect to the robot and begin driving it.
2. The competition router is unplugged and powered down. At this point, the robot has three minutes to stop and indicate that connection is lost.

The competition router is plugged back in and powered on. The operator has 5 minutes to demonstrate that they can reconnect to the robot and resume operation. [1]

2.3 Competition Rules and Scoring

Written By: Kamran Coulter

The competition manual does not detail many design parameters, and the rules of the competition are primarily related to safety during handling and operation of the team's robot. In general, a team's robot must be able to maximize score while not posing a threat to persons or property at the competition location. The only rules specified in the competition documents are as follows:

1. The robot must fit into an 18x18x12 inch cube at the beginning of the run but can expand beyond those dimensions once the run has begun.
2. All Lithium batteries must include proper charging systems as well as low voltage cut-off circuits.
3. All robots must be operated from a minimum distance of 50 miles away from the competition location.
4. Components that could cause damage to persons or property are not permitted.

As stated previously, the goal of the competition is ultimately to maximize the team's score in order to win the competition. The theoretical maximum score in the competition is 261.25, which corresponds to a perfect run with a zero second sprint. The following section describes the scoring procedures of the competition by stating the scoring algorithm and describing all possible score penalties.

$$\boldsymbol{Score = (T + P + S - 15 * SP + SS + D - DP) * M - 5 * WC - 10 * RP} \quad \textbf{(1)}$$

Table 1: Scoring Equation Variable Description [1]

Description	Variable	Scoring
Tunnel	T	Clean Run = 30; One Wall Touch = 15; Otherwise = 0
Pickup Bolt	P	Securing = 30; Otherwise = 0
Slalom	S	No Pylon Contacts = 50
Slalom Penalty	SP	SP = Pylon Contacts * -15
See Saw	SS	Crosses With Bolt = 40; Crosses Without Bolt = 20
Deposit	D	½" Hole = 40; 1" Hole = 30; 2" Hole = 20; Missed = 0
Deposit Penalty	DP	Contact With Drop Off Zone = -5
Sprint Multiplier	M	M = 1.375 - 0.0075*Sprint Time
Wall Contact	WC	WC = Wall Contacts * -5
Reset Penalty	RP	RP = # of Resets * -10

2.4 Competition Results

Written By: Kamran Coulter

The competition was held on Saturday April 22, 2017 in the NRC of Oklahoma State University. The team's robot was present at the competition, completed the loss-of-signal test, and completed the course with a score of 180 points to finish third overall of twenty competing robots. The score is reflective of a near perfect run with only two deductions: 5 points for 1 wall contact and 40 points for bypassing the seesaw. During testing, it was determined that the seesaw could potentially cause damage to the robot due to traction issues while climbing the incline, so the team decided to incur the point deduction instead of risking a no-score.

In addition, the robot was able to complete the sprint section with a competitive time of 18 seconds. The two robots that outscored the team's robot were both from Columbia, each able to go over the seesaw. In addition to placing third in the competition, the team also received the best video award for the robot demonstration video. The team's robot and awards are shown in Figure 2.

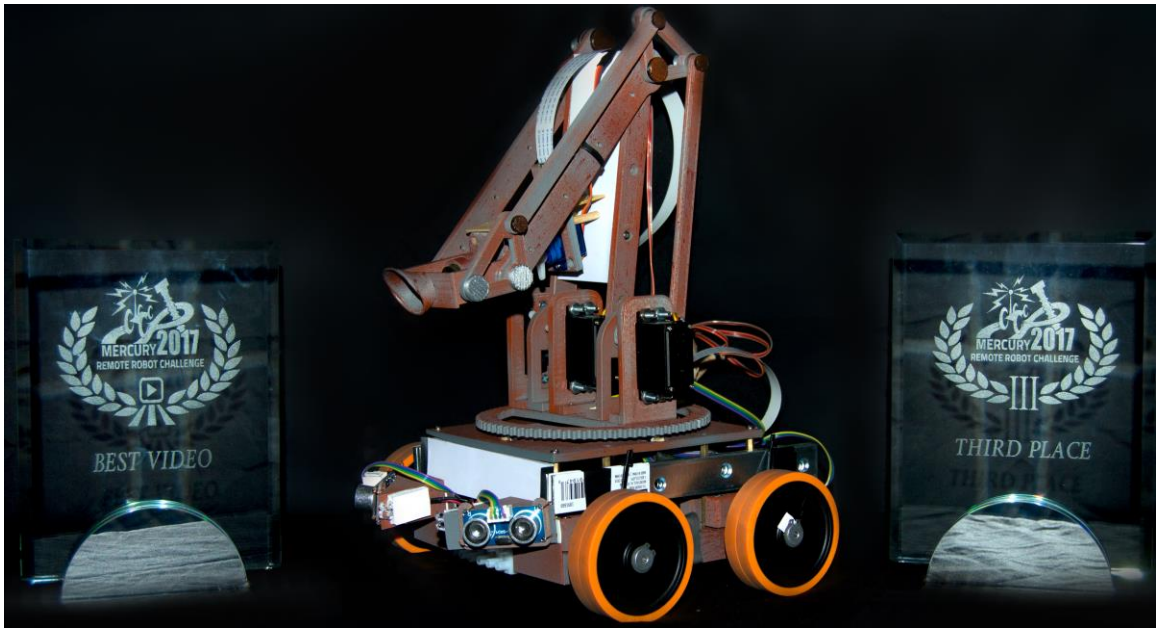


Figure 2: Competition Robot and Awards

3 Mechanical Design

3.1 Mechanical Systems Overview

Written By: Jonathan Ballew

The robot's mechanical design is divided into two primary systems: the motion platform and the manipulator arm. The motion platform serves as the mounting point for the main drive motors, collision detection sensors, battery, and the primary control and communication electronics. The motion platform is designed for high stability and maneuverability at moderate speeds. It has a low ground clearance for maintaining a low center-of-gravity, and it incorporates wire management for easy motor access and access to the Raspberry Pi's USB and Ethernet ports. The wheels are a high friction rubber for high traction when traversing the seesaw obstacle.

The manipulator arm is responsible for collecting and depositing the bolt. It also serves as the mounting point for the drive camera. The manipulator is designed for minimum weight and maximum versatility. The end effector is a screw-driven retractable high-strength magnet and bolt guide. The camera is positioned above the end effector to provide the best view possible for bolt manipulation. The manipulator is powered by 5 servos for 5 degrees of freedom: 4 rotational degrees (3 in the arm elbows and 1 for rotation of the entire arm left and right) and 1 translational degree (for the retraction of the magnet).

3.2 Motion Platform

Written By: Jonathan Ballew

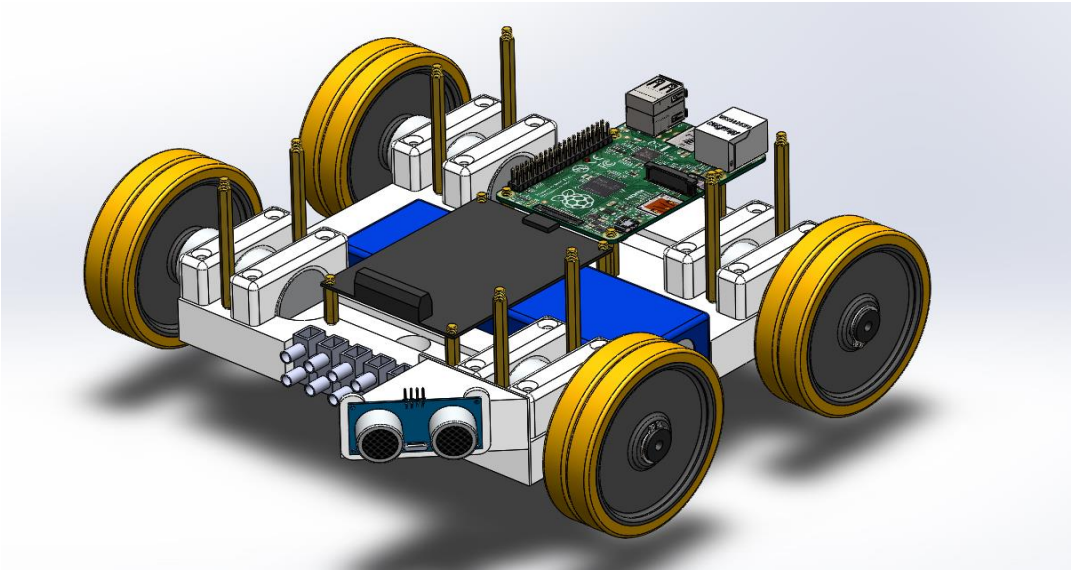


Figure 3: Motion Platform Base Rendering

All motion platform components (excluding fastening and standoff hardware) are custom modeled by Jonathan Ballew. The models are 3D printed in ABS plastic by

Kamran Coulter on his personal printer. The design achieves high stability by the reducing the height of the center of mass by placing the heaviest components -- the high-capacity lithium-polymer battery and the arm assembly -- at the bottom of the robot. The base is also designed with low a ground clearance to further lower the center-of-gravity. The arm platform attaches to 8 50mm standoffs mounted at the corners of the bot. These provide only enough clearance to connect all wires to the Raspberry Pi HAT, further lowering center of mass. The final base dimensions are: 9.75"w x 10.75"l x 3.25"h.

High top speed and torque are accomplished with 4 Actobotics 970 RPM Econ Gear motors. The four motors are fixed to the base and do not support any form of suspension. Each pair of motors is wired in parallel on separate left and right channels. This provides a slip-steering/tank-drive control system with steering accomplished by running each side at different speeds and/or directions. Each of the four corners of the robot house ultrasonic sensors mounted at 45 degrees outward from the robot. These sensors provide real-time distance and obstacle detection to the robot operator. Through testing, the ultrasonic sensors proved to be accurate only in a small cone centered on perpendicular surfaces. The 45 degree angle prioritizes accurate readings in the situations where the robot is turning towards an obstacle as opposed to driving alongside it.

The Raspberry Pi 3, Cytron motor control board, custom Raspberry Pi HAT, ultrasonic sensors, and battery are arranged to minimize unused space. The Pi is oriented to expose the onboard USB and Ethernet ports for any necessary debugging or future expansion of functionality. The motor wires run in troughs perpendicular to the back of the motor to reduce the strain on the solder joints and reduce the chance of breakage. Connecting wires run through holes to the bottom of the robot where they are bound together and brought through to the wire block. This simplifies connection to the motor board and provides simple disconnection and debugging access.

The team's robot attained 3rd place in the competition, but the robot could not successfully complete the seesaw obstacle. Due to the lack of wheel suspension, the design did not allow for maintaining uniform surface traction given variations in ground surface height. When attempting to climb the seesaw, the lack of uniform traction caused the wheels to be unevenly loaded, and the robot would spin out towards the seesaw edge or slide back down to the bottom without ever achieving enough traction to climb the incline. Prototype base designs did not exhibit this behavior, but only when the manipulator arm was not attached. It was determined that the added weight of the arm amplified the uneven loading and traction to critical amounts.

The incorporation of wheel suspension in the design could promote more even distribution of weight across all wheels on uneven terrain. This would ensure that

no wheel is unevenly loaded, preventing the robot from torqueing to the left or right when ascending the incline. This modification would likely require a complete redesign of the motion platform, but no changes would need to be made to the electrical and control systems.

Future designs could also include some form automatic traction control by using motors with built-in encoders. When a wheel slips, it spins faster than the other wheels on the robot. The feedback provided by wheel encoders could be used to determine when a wheel slips by periodically comparing the speeds of each motor to each other. Increasing or decreasing the power provided to each motor could compensate for the difference and more evenly distribute the load. However, employing this method would require that all motors be on four separate control channels instead of the two currently employed.

The location and positioning of the Raspberry Pi also proved to be non-ideal. This design does not allow for access to the HDMI, audio, and USB power jacks that were used constantly for debugging and development. To make used of these ports, the bot had to be partially disassembled. The power switches and debug LEDs on the custom Raspberry Pi HAT are also not conveniently accessed, though they can still be accessed when the bot is fully assembled. The LED headlights also proved to be barely powerful enough to illuminate the tunnel. Possible improvements include more LEDs and/or fixing LEDs to the arm near the camera. Lastly, the battery could not be removed from the bot without first removing a wheel. Removing any of the wheels requires a specific tool, making it inconvenient to remove and charge the battery.

3.3 Manipulator Arm

Written By: Jonathan Ballew

All manipulator components (excluding fastening and electrical hardware) are custom designed and modeled by Jonathan Ballew. The models are 3D printed by Kamran Coulter in ABS plastic on his personal printer. The design focuses on low weight and high versatility. It minimizes weight using minimally sized structures and lower torque servos. To accommodate the lower power servos, all servo motors are mounted at the base with small linkage arms connecting them to joint they control. Placing the servos at the base as opposed to at the joints they control minimizes the load on each servo. All critical joints also use bearings to reduce friction and servo loading. The bottom platform has cutouts to reduce weight and allow access to the power switches on the Raspberry Pi HAT, and the base of the arm connects to the standoffs located on the motion platform.

The arm is powered by 5 servos total. 3 Hitec HS-311 digital positional servos power the three main arm elbow joints, each providing 51 oz-in of torque. The base rotation is powered by a Parallax Continuous Rotation Digital Servo, which provides 38 oz-in of torque. The end effector screw mechanism is powered by a 9g lightweight servo. The custom Raspberry Pi HAT drives all servos at the same PWM frequency and standard pulse timing (0.75ms - 2.25ms pulse width).

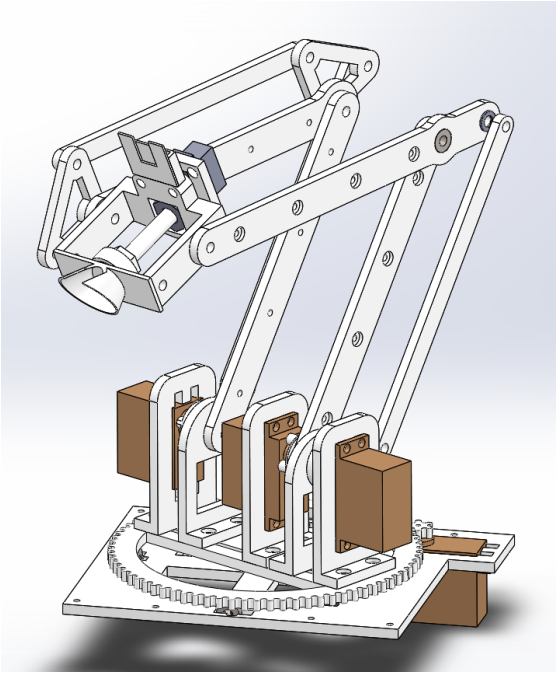


Figure 4: Manipulator Arm Rendering

Versatility is an important aspect of the arm's design. The arm is able to place the bolt in a variety of positions beyond the three outlined in the competition specifications, and can maneuver to many different angles to provide different viewpoints for the operator. The arm is divided into three individually controlled segments and a rotating base. The end effector includes the bolt collection assembly and the camera. The camera is mounted directly along the bolt to maximize the operator's ability to accurately position it while depositing the bolt in the drop off zone. Bolt collection is performed using a high-strength permanent magnet, able to extend and retract via a screw mechanism. The mechanical advantage of the screw allows the servo powering the bolt collection to be much smaller and lighter than other designs that translate a permanent magnet away from the effector to release the bolt. The high strength permanent magnets guarantee the bolt remains secure during operation regardless of the robot's speed or orientation.

The largest possible improvement for the arm would be incorporating higher power servos. Stronger servos would allow stronger construction and more accurate positioning. Stronger servos would also reduce the effect of servo vibration on the camera view, making driving easier for the operator. Incorporating higher-torque servos would also allow a redesign of the manipulator to have a greater range of motion.

Another significant improvement to the manipulator would be the addition of a second camera to take the place of the current camera during driving. By adding a second camera on the base, the driver would have a more stable viewpoint with a full view of the wheelbase to prevent accidental collision.

3.4 Paint

Written By: Kyle Edwards

To personalize the robot, the team painted all of the 3D printed components on the robot. Inspired by the Borderlands video game series, the team painted the robot to give it a damaged, rusted appearance. The paintwork was done with a red oxide primer and an orange acrylic on top of a grey primer. The bottom and top platforms were painted with the grey primer tinted to a darker tone, and the manipulator was painted with the original lighter grey in a spray booth. Following the application of primer, each part was lightly sprayed with the orange acrylic and splattered with the red oxide at certain points to give the robot dark, worn, and rusted characteristics.

4 Hardware Design

Written By: Kamran Coulter

The following section outlines all of the electrical hardware design related to the team's robot. An overview will be presented alongside a system block diagram. The remained of the sections will describe in detail each subsystem and its overall function in the final robotic system.

4.1 Electrical System Overview

Written By: Jonathan Ballew

A high-level block diagram of the robot's hardware and software systems is shown below in Figure 5.

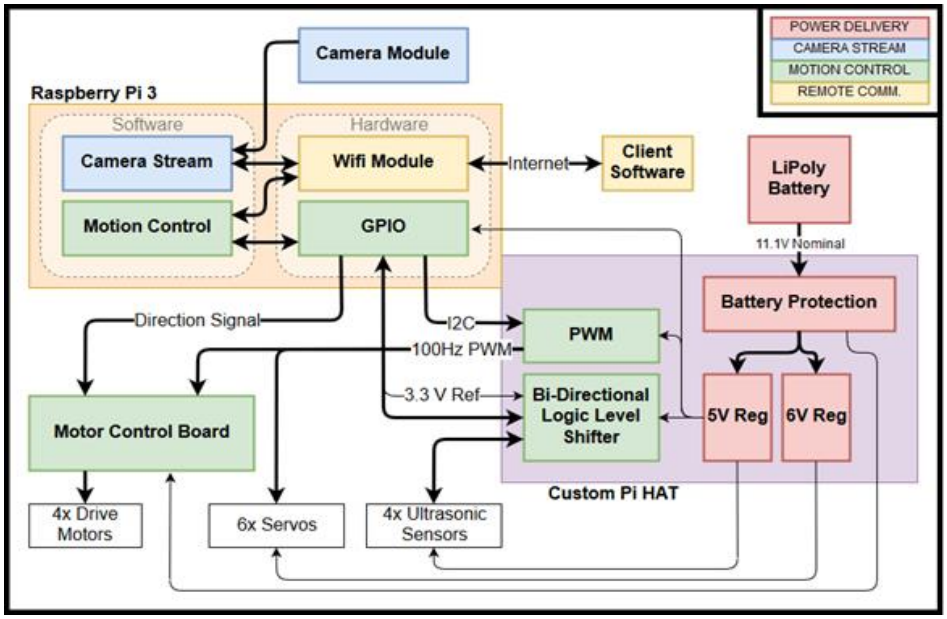


Figure 5: Functional Hardware and Software Block Diagram

The Raspberry Pi is supplemented with custom designed hardware attached on top (HAT) that connects and simplifies various electronics hardware and interconnections on the robot. In addition, the HAT serves as a power supply for all other subsystems in the design (excluding the motors, which draw power directly from the battery). All systems source power from a single 11.1V nominal lithium polymer battery, regulated to lower voltages for each separate subsystem via switching regulators. The robot has four brushed DC motors controlled by a 2-channel commercial motor control board with a 10A per channel maximum output current. The manipulator arm has five servos which are connected directly to the HAT. Finally, the HAT interfaces the four ultrasonic sensors that are used for collision detection with the Raspberry Pi controller.

4.2 Main Computer

Written By: Jonathan Ballew

The Raspberry Pi 3 Model B serves as the only controller for the robot. The Raspberry Pi 3 is an inexpensive and high performance computer capable of handling all control and communications systems necessary for operating the robot. The Raspberry Pi 3 is loaded with the NOOBS installation of Raspbian Jessie, an arm derivative of Debian Linux. This allows for simple user access to all the hardware available on the Pi. In particular, NOOBS provides access to the high performance processor, wifi module, CSI camera port, and GPIO without the need to install for extra drivers or hardware. Additionally, the full Linux operating system gives access to hundreds of open-source software libraries and languages, expediting development tremendously.

The only potential improvement available to the main controller system would be swapping out the Raspberry Pi 3 controller for a Raspberry Pi Zero, released shortly after the team began work on the project. While the Raspberry Pi Zero is significantly smaller than the Raspberry Pi 3 and can be purchased for as low as $5, using a Raspberry Pi Zero compared to a Raspberry Pi 3 reduces the computation resources available for possible future design improvements.

Raspberry Pi 3 Model B Specifications [2]

- 1.2 Ghz 64-bit quad-core ARMv8 CPU
- 802.11n Wireless LAN
- Bluetooth 4.1 with BLE
- 1 GB RAM
- 4 USB 2.0 Ports
- Ethernet Port
- Audio/Composite Video Jack
- 40 GPIO Pins
- HDMI
- CSI (Camera Serial Interface)
- DSI (Display Serial Interface)
- MicroSD Card Slot
- Power Requirements: 5V @ up to 2.5A, 200mA typ.

4.3 Custom Designed Raspberry Pi HAT

Written By: Kamran Coulter

The following section describes in detail the design of the previously mentioned custom designed Raspberry PI HAT. The HAT was designed in accordance with the published Raspberry PI HAT standards. [3] For the board to be called a HAT, it must have an identification EEPROM for automatic hardware identification by the Raspberry Pi. However, the team decided not to develop the EEPROM functionality because this board will not be sold as a commercial product and is therefore unnecessary. The board contains the proper surface mount footprints for implementing EEPROM for device identification, so the board will still be referred to as a HAT for simplicity.

4.3.1 Overview

Written By: Kamran Coulter

The main controller for the robot is the Raspberry Pi 3 controller. Although the Raspberry Pi is simple to use for a wide range of communication systems, it is not well suited for the PWM signaling applications required in mobile robotics due to the threading implications of the computer's full Linux operating system. The team decided to design a custom Raspberry Pi HAT to solve the PWM signaling issues. In addition to PWM signaling hardware, the HAT hosts various connectors, sensor level shifting circuitry, two switching power supplies, and power protection circuitry. Figure 6 shows a high-level signal diagram for the HAT.

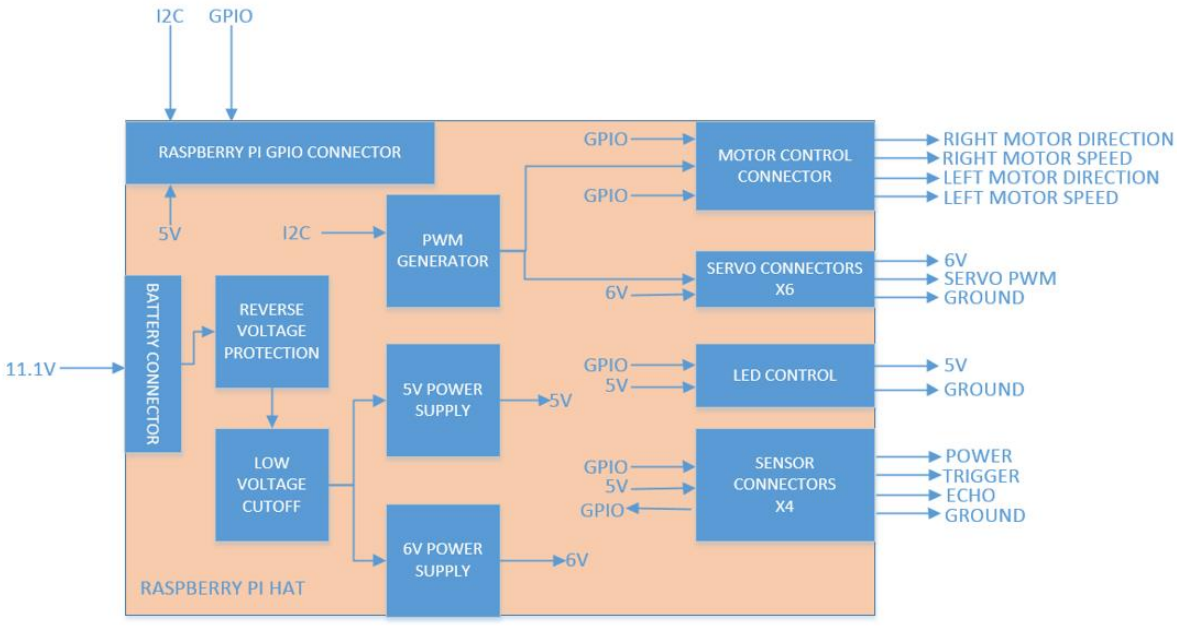


Figure 6: Raspberry Pi HAT Signal Diagram

4.3.2 PWM Generation Hardware

Written By: Kamran Coulter, Jonathan Ballew

The hat uses the PCA9685, a 16-channel 12-bit PWM LED driver IC, to implement PWM signaling. The IC is optimized for RGBA backlighting applications, but the 12-bit resolution and programmable frequency ranging from 24 Hz to 1526 Hz makes the IC ideal for the PWM applications of the team's robot. The Raspberry Pi communicates with the IC via an I2C interface across the Raspberry Pi's GPIO pins. The I2C interface first sets the operating frequency of 100 Hz across all channels on the device. The IC shares the overall PWM frequency across all channels, and it cannot be changed on a per-channel basis. Individual channels can then be set to a particular duty cycle by writing I2C values to the device in software. The corresponding PWM frequency and the 12 bit resolution allow the IC to function for both servo and motor PWM signaling. The final design utilizes eight channels of the device, where two channels are used for motor speed control, and 6 channels are used for servo angle/speed control. [4]

4.3.3 Level Shifting Sensor Hardware

Written By: Kamran Coulter

The hat also host a TXB0108, an 8-bit bidirectional voltage level translator, in order to facilitate communication between the Raspberry Pi and ultrasonic sensors. The IC does automatic direction sensing and is configurable for different voltage levels. The device translates between the 5V rail towards the sensors and the 3.3V rail towards the Raspberry Pi. In addition, the device must be enabled via a GPIO signal connected to its output-enable pin, as it maintains a high impedance state at startup. [5]

4.3.4 Power Distribution on the HAT

Written By: Kamran Coulter

Additionally, the HAT hosts two switching power supplies designed around the TPS5454 IC from Texas Instruments. A simplified application schematic is shown in Figure 7. A detailed schematic with selected component values can be found in the appendix. Each regulator has an adjustable output voltage and can supply 5A of current, which is far greater than the robot's maximum power

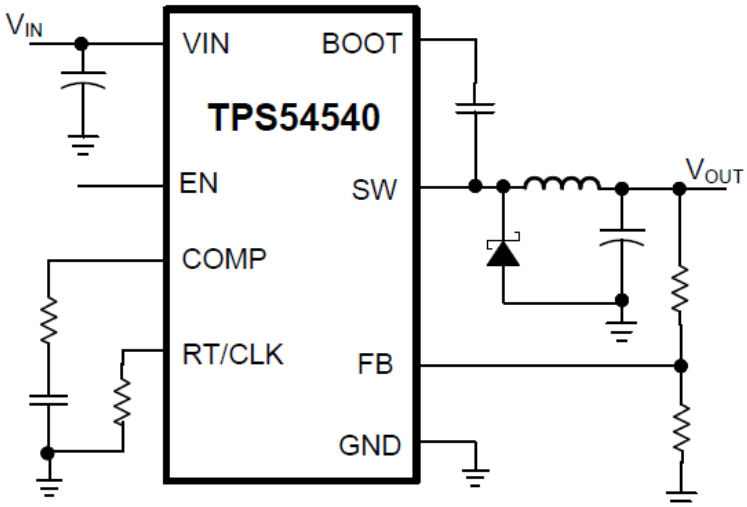


Figure 7: Simplified Application Schematic TPS5454

requirements. One regulator is configured to output 6V to the servos, and the other is configured to supply 5V to the other hardware including the Raspberry Pi. The HAT also implements a low voltage cutoff circuit via the EN pin of the device, which powers down the entire system if the battery voltage falls below 3.3V per cell. The supporting hardware surrounding the regulators was selected with assistance from Texas Instruments WEBENCH ™ Designer software. The software outputs were then verified against the provided equations in the data sheet. [6]

4.3.5 PCB Layout

Written By: Kamran Coulter

The team designed the PCB with 4 layers total using NI Ultiboard/Multisim, and it was constructed by a commercial manufacturer. The team initially attempted to design the PCB using only two layers due to budget constraints, but it was clear that routing of power and ground signals would be impossible after all other signals had been routed. The physical dimensions of the board are 65 x 56 mm, defined by the Raspberry Pi Hat Standards. [3] As such, the team was unable to expand the board's physical footprint to make a two layer design possible. The final layout uses four layers, with internal layers containing unbroken power and ground planes. Images of the signal layer layouts are included below in Figure 8 and Figure 9.

TOP

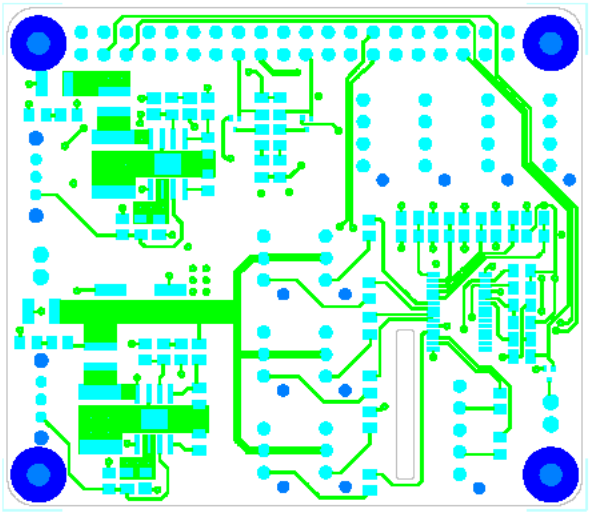


Figure 8: Top Layer HAT Routing

BOTTOM

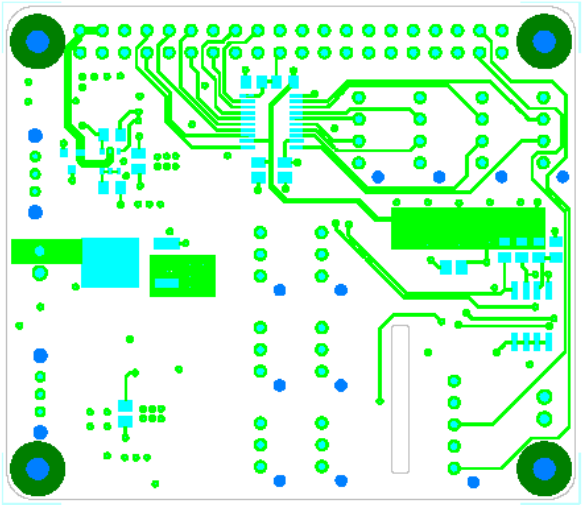


Figure 9: Bottom Layer HAT Routing

4.4 Power Systems

Written By: Kyle Edwards

The power source for the entire design is a single Venom 11.1V lithium-polymer battery with a 4000 milliamp-hour capacity. The battery is divided between three main subsystems: the motor driver and two switching regulators. The motor driver provides the voltage potential to the left pair and right pair of motors, each pair in parallel. The two switching regulators output two different voltages: one at 6 volts and the other at 5 volts. The 6-volt regulator supplies power to the servos in the manipulator and end effector, while the 5-volt regulator supplies power to the Raspberry Pi and ultrasonic sensors. Power is routed through two forms of voltage protection before reaching the two regulators.

4.4.1 Reverse Voltage Protection

Written By: Kyle Edwards

The first means of voltage protection is a PMOS transistor to prevent against reverse voltage between the battery and the load. The reverse voltage protection uses a P-channel transistor acting as a diode in series with the battery and the load. When the battery is connected properly, the gate voltage is taken low and the channel of the transistor shorts to allow current through. The selected PMOS transistor has a drain to source voltage of -35V and a gate source voltage +25,-25, which is above the required 11.1 volts. It also only has a static drain-to-source on-resistance of around 9.6 to 11.6 milliohms, resulting in only a slight drop in voltage. Using a PMOS transistor is one of a few common methods to create a reverse voltage protection circuit, with the others using a similar circuit with diodes and PNP transistors. While other methods to protect against reverse voltage exist, using a single PMOS transistor had the least significant voltage drop among the options explored.

4.4.2 Low Voltage Cut-Off

Written By: Kyle Edwards

The second means of voltage protection is a low-voltage cutoff circuit that disconnects the battery from the power supply and switches on an LED once the battery falls below 9.9 volts.

The low voltage cut-off uses an adjustable under voltage lockout with TPS54540 switching regulators. The regulator the chip turns on when the input voltage rises above 4.3 volts, and turns off when the input falls below 4 volts. For dealing with higher cut-off, the technical document for the regulator gives the method for finding the two external resistor values R1 and R2. The solved resistor values were 147 kilo ohms for R1 and 19.1 kilo ohms for R2 with the equation below:

$$R_1 = \frac{V_{Start} - V_{Stop}}{I_{Hys}}$$

$$R_2 = \frac{V_{ENA}}{\frac{V_{Start} - V_{Ena}}{R_1} + I_1}$$

Figure 10: Voltage Cutoff Equations [6]

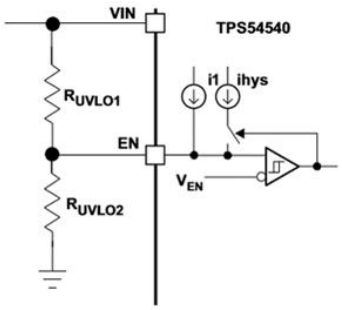


Figure 11: Voltage Cutoff Functional Circuit [6]

$V_{Start} = initial\ voltage\ of\ source$

$V_{Stop} = desired\ voltage\ cut - off$

$V_{ENA} = EN\ terminal\ voltage\ enabled\ threshold$

$I_{Hys} = hysteresis\ current\ of\ 3.4\ \mu A$

$I_1 = internal\ pull - up\ current\ of\ 1.2\ \mu A$

4.4.3 Switching vs Linear

Written By: Kyle Edwards

When choosing the regulators to use for the build we went with two switching regulators instead of linear. Linear chips are usually cheap and easy to use although are inefficient when dealing with high-powered devices. They take the difference between the input and output, and burn up the difference as wasted heat. Therefore, with the large difference, we have with the 11.1V input and 5V output on one of them with a load current of 1.26 amps for the pi and sensors would be wasting about 7.6 watts of power as shown in the equation below:

$Power\ wasted = (input - output\ voltage) * load\ current$

$(11.1 - 5) * 1.26 = 7.6\ watts$

The switching regulators only take small amounts of energy from input and transferring them to the output by a switch and controller used to regulate the amount of energy transferred. So the amount of energy lost from this is smaller compared to linear that relies on the amount of the input voltage. They can be a more complex circuit to design but the payoff of the efficiency gain from the lower power losses is ideal with our system. [7]

4.5 Motor and Wheel Selection

Written By: Kamran Coulter

The motors and wheels for the robot needed to be selected and sized appropriately to ensure the robot would be able to complete all course obstacles. The two major obstacles related to the motor and wheel subsystem were the seesaw and the sprint sections. The robot needed to be able to produce the required torque to maintain a constant velocity up a 30-degree incline while also allowing for a competitive sprint time. In addition, the motor eventually selected needed to be a DC motor due to the robot being battery powered. Brushed DC gear motors were selected for their cost, simply control structure, and relative availability at the hobbyist level.

4.5.1 Robot Sprint Speed Calculations

Written By: Kamran Coulter

It was determined through evaluation of the scoring algorithm and feedback from the team's project mentor that the motor revolutions per minute (RPM) and corresponding wheel size would provide an unloaded ground speed of 10 ft/second if the robot were to be competitive in the sprint section. The underlying equation for robot speed, assuming zero slippage, is as follows:

$$S = 2\pi * R * RPM * \frac{1}{12 * 60} \quad (2)$$

S, Speed (ft/sec)
R, Wheel Radius (in)
RPM, Motor RPM (rev/min)

Starting with a wheel with a radius of 1 7/16", it was found that a motor rated at approximately 800 RPM would provide the desired speed.

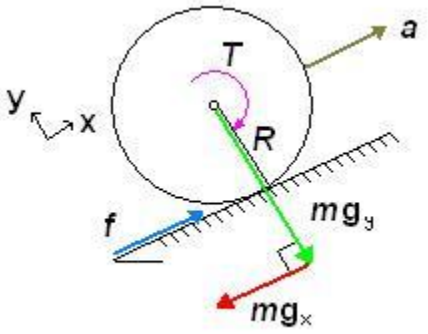
4.5.2 Required Torque Calculations

Written By: Kamran Coulter

The required torque to adequately carry the robot over the seesaw required more assumptions. A table of those assumptions is provided below in table 2. The torque analysis used a standard power balance approach that assumed perfect physics in terms of friction. A 65% gear efficiency was assumed to overcome any deficiencies in friction or gear imperfections.

Table 2: Torque Calculation Assumptions

Assumption	Value Assumed
Total Mass	7.5 (lbs.)
Number of Drive Motors	4
Acceleration	1 (ft/s^2)
Gear Efficiency	65 (%)
Wheel Radius	1 7/16 (in)
Incline	30 degrees



m: {mass}

a: {acceleration}

T: {torque}

g: {gravitational constant}

R: {radius}

Figure 12: Wheel Cross Section [8]

Figure 12 shows a two-dimensional cross section of a wheel accelerating up an incline. The required torque can be calculated by performing a standard force balance in the x direction as follows:

$$\boldsymbol{\Sigma F = ma = f - mgx} \quad \textbf{(3)}$$

$$\boldsymbol{Ma = T/R - mg * sin(\theta)} \quad \textbf{(4)}$$

$$\boldsymbol{T = Rm(a + g * sin(\theta))} \quad \textbf{(5)}$$

Equation 5 will give the required torque of the entire robot at 100% efficiency. To calculate the required torque per motor at 65% efficiency, a correction factor must be applied as follows:

$$\boldsymbol{Tm = T * \frac{0.65}{4}} \quad \textbf{(6)}$$

By using the above assumptions and equation 6 it can be found that at a 65% gear efficiency each motor needs to be able to produce approximately 35 ozf-in of torque in order to accelerate up a 30 degree incline.

4.5.3 Motor and Wheel Issues

Written By: Kamran Coulter

The actual motor selected for the design was the Actobotics 970 RPM Econ Gear motor, which has a rated torque of approximately 53 ozf-in and a stall current of 3.8A. This motor exceeded both the torque and speed specifications set above, and was selected due to its low cost and availability. It was assumed that since the motor specifications greatly exceeded the required torque, the robot would have no issues completing the seesaw. Unfortunately, the notion that the potential torque of the motors is directly related to the friction between the wheels and the seesaw surface was overlooked. It was found in testing that the wheel material did not provide sufficient traction while on the smooth surface of the seesaw. Random slips in the wheels would cause the robot to spin and become uncontrollable on the smooth

surface. Future improvements in the design would incorporate different wheel materials to facilitate better traction on smooth surfaces.

4.6 Motor Control Hardware

Written By: Kamran Coulter

The motors are controlled with a MDD10A solid-state motor driver from Cytron Technologies pictured in Figure 13. The board hosts a standard solid-state H-Bridge circuit for switching the motors direction and speed. The interface between the motors and the main controller consists of two motor direction GPIO signals and two PWM speed control signals. A commercial motor control board was selected primarily due to budgetary and time constraints. Designing and manufacturing two separate printed circuit boards (PCB) could not be accomplished within the team's budget and project timeline. The motor control board hosts the following features:

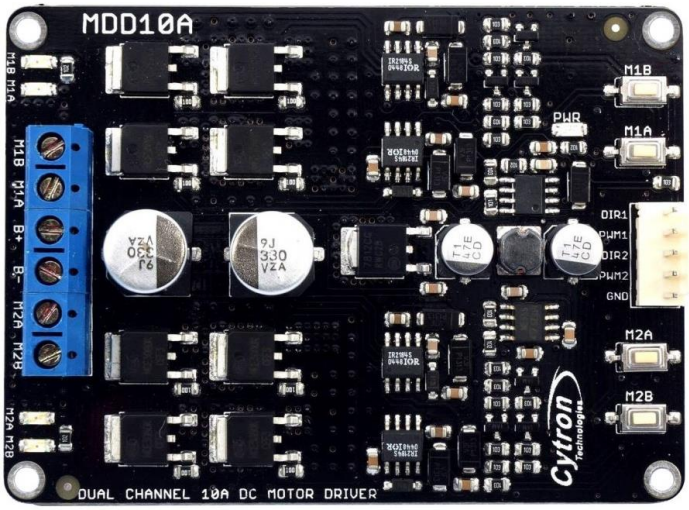


Figure 13: Cytron MDD10A DC Motor Controller

- Bi-directional control for 2 brushed DC motors
- Supports motor voltages from 5-25V
- 10A max current per channel
- NMOS H-Bridge for improved efficiency
- PWM frequency up to 20 KHz
- Locked-antiphase and sign-magnitude PWM operations supported

4.7 Ultrasonic Sensor System

Written By: Jonathan Ballew

The robot's collision detection system is comprised of four ultrasonic sonar sensors, one at each corner of the robot. These sensors operate at 5V supplied from the same voltage regulator on the Raspberry Pi HAT that supplies power to the Raspberry Pi. A request for a distance read begins by sending a 10 us 5V pulse to the sensor's "TRIG" pin, triggering the transmitter to send a 40 kHz audio signal. At this time, the "ECHO" pin raises to 5V and remains high until the receiver detects the reflected audio signal. The Raspberry Pi measures the length of time that the ECHO pin is high and returns that raw value in microseconds. This value can be converted into measurement distance. Each sensor is read serially with a sensor being triggered only after a value has been read by the previous or after the previous

sensor times out. If the sensor does not detect a reflected signal after 30 ms, the ECHO pin resets to 0V and another read can be attempted. The sensors are rated with an accuracy of 0.3 cm and an average current draw of 15 mA.

The Raspberry Pi's GPIO operates at 3.3 V, making the GPIO pins of the Raspberry Pi not directly compatible with the 5V logic of the ultrasonic sensors. Connecting the sensors to the Raspberry Pi required a logic level shifter with eight channels, two for each sensor. The logic level shifter requires a negligible 10ns to switch between 3.3V and 5V, having no noticeable impact on the sensor accuracy. [5]

4.8 Video Feedback System

Written By: Jonathan Ballew

The team implemented a camera feedback system using the camera module v2 for Raspberry Pi. This was chosen primarily for its native compatibility with the Raspberry Pi and high resolution and framerate capabilities for its price. Its low weight also allows for mounting on the end of the manipulator without much impact on the reliability of the manipulator. The camera module connects to the Raspberry Pi's dedicated camera-serial-interface (CSI) port with an 18" ribbon cable. The CSI port handles all power and signals for the camera module. Camera image quality was not considered to be of high importance as the operator only needed to be able to traverse a well-defined track.

Camera Module V2 Specifications: [9]

- 8MP Sony IMX219 Image Sensor
- Common Supported Resolutions: 1080p30, 720p60, 480p90
- Raspberry Pi CSI Port compatibility
- Weight: 3.4 g
- Dimensions: 25mm x 23mm x 9mm

5 Software Design

5.1 Software Systems Overview

Written By: Joseph Austin

In order to interface between the operator's control input, the Raspberry Pi controller, and the robot's hardware, the design needed a software system on both the operator's machine and the Raspberry Pi. The operator's machine needed to process input from a control peripheral and pass control data to the Raspberry Pi over a socket. Moreover, the operator's machine needed to receive feedback from the robot during operation and display crucial information to the operator. The robot needed software to receive control data over a socket, interpret the data, and exert control over the robot's servos, motors, and LED's. The robot also needed to be able to read sonar data and convert it into positional information. To meet these needs, software design was broken down into the following key subsystems:

- The operator's control and feedback interface
- Software to decode received control data
- Motor speed and direction control software
- Servo position/speed control software
- Software to read and convert positional data from four sonar sensors
- Software to stream video data to the operator's machine

The following sections detail and justify the design and implementation of each of these subsystems.

5.2 Operator Interface

Written By: Joseph Austin

The control scheme was designed around operating the robot using a Microsoft Xbox 360 controller as the control peripheral. The Xbox 360 controller provides a familiar and approachable control layout, and it is easy to incorporate into control software on the operator's machine. Moreover, its range of both digital and analog control inputs would easily accommodate controlling the numerous hardware systems on the robot. The operator's control interface is written in the Python language using the PyGame open-source library. The software is written using the PyGame library for its familiarity and ease of both reading data from a control peripheral and displaying a graphical-user-interface (GUI) to the operator with feedback data from the robot.

Software on the operator's machine polls input data from the Xbox 360 controller every 0.1 seconds. This time interval was selected empirically; reading slower than this interval lead to degraded control response from the robot, while reading faster than this interval showed no noticeable improvement in control response. The state

of every button, analog stick, and trigger is read simultaneously, and the values are saved to an object. The control interface compares the values saved to the object to values stored in the previous iteration of polling control input. If a value differs, a button has been pressed or released, or some stick or trigger has changed position. Once the program determines a state change of control input, it constructs a command string to send over a non-blocking socket to the robot.

Each command is a 7-byte string, with the button, stick or trigger indicated in the first two bytes of the string and an analog value, if applicable, in the remaining 5 bytes. This allows analog values to encode to integers ranging from -9999 to 9999, providing a wide range of control input values while simplifying the message decoding process on the Raspberry Pi's software interface. Each string message was padded with spaces to 7 bytes to ensure every 7 bytes read by the Raspberry Pi was only one command. To prevent flooding the communication socket with commands and simplify the control scheme, the program sends only one of each type of button press, stick movement along the X-axis, stick movement along the Y-axis, and trigger press per input poll. The following are examples of command strings the operator's machine would send to the Raspberry Pi:

- "LX-5493" – indicates the Left Stick in the X direction was reading a value 54.93% of completely to the left.
- "A "– indicates the A button was pressed down.
- "RY102" – indicates the Right Stick in the Y direction was reading a value of 10.2% of completely up.

The control scheme is divided into the three modes: motor mode, manipulator mode, and routine mode. Pressing the start button on the controller cycles between each of the three modes. This was done to allow each mode to have the full range of buttons on the Xbox 360 controller, if needed. Motor mode allows the operator control over the motors and headlights, as well as positioning the base of the arm and the tip of the arm to reposition the camera while driving. Manipulator mode allows control of each of the servos on the manipulator arm separately. Lastly, routine mode allows the operator to run pre-programmed routines for different portions of the course. These routines included folding the manipulator arm to fit in the tunnel, reaching the manipulator down to secure the bolt, raising the manipulator into a position to deliver the bolt, and moving the effector between fully retracted or fully extended. Pictured below in Figure 14 is the complete control layout.

Figure 14: Xbox Control Layout

After sending each group of commands, the program will poll the socket buffer for any strings received from the robot. This is done to allow the operator's machine to display feedback information from the Raspberry Pi to the operator. The interface prints string messages from the robot and renders sonar positional data to the screen in a GUI. Pictured below in Figure 15 is the GUI the operator sees during robot operation.

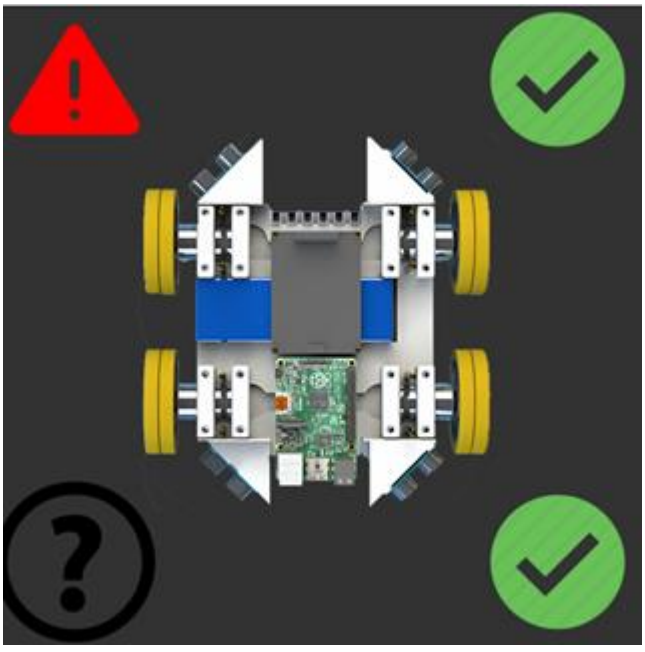


Figure 15: Sensor Data GUI

The GUI displays the status of each of the four sonars, and it is only updated if a sonar's state changes to prevent unnecessary processor load when rendering the frame. A warning symbol indicates a sonar is reading "too close" (less than 10 centimeters), a check mark indicates the sonar is reading a value that is "ok" (between 10 and 100 centimeters), and a question mark indicates the sonar reading may be unreliable (greater than 100 centimeters). Moreover, any time the GUI draws a warning symbol to the screen, it plays a beep to warn the operator that the robot is dangerously close to collision.

The largest fault of the operator's interface is its complicated design. Because the robot needed to be able to perform a wide range of operations, the control scheme became increasingly more complicated as additional features were implemented. A better solution would design a more simplified control scheme using a controller with a greater number of inputs, such as a Steam controller. The modes of operation could be redesigned to require fewer types of button presses and stick values to function, or the three operational modes could be simplified to the extent of being removed entirely.

Second, the GUI displayed to the operator is overly simplistic and its separation from the video stream did not prove to be an ideal means of relaying feedback from the robot to the operator. The ideal solution would be to integrate the feedback from the Raspberry Pi directly into the camera stream interface to remove the need for looking between multiple windows for operational data.

5.3 Communication Software

Written By: Joseph Austin

Once the operator's machine sends a command over the socket, the robot needed to be able to receive, decode and execute the command. Moreover, the Raspberry Pi needed to be able to send operational information, sonar data, and video data to the operator while they were manipulating the robot. Lastly, software on the robot's end needed to be able to determine if the connection with the operator had been unexpectedly lost, and, if it had, and restart the socket.

To accomplish this, the Raspberry Pi first opens a non-blocking socket, acting as a server. The socket is nonblocking to allow the Raspberry Pi to continuously read and transmit data from the sonars even if no command is received. If the socket were blocking, the program could only transmit sonar data as often as it receives commands or the socket times out. The socket will only block until a client connects. Afterward, it will time out immediately if the socket buffer is empty.

The communication interface processes commands from the buffer one at a time. The program determines the type of control input from the first two bytes of the 7-byte command and will read the remaining 5 bytes as an analog integer value if

applicable. Analog values are used to control motor speed and direction, as well as servo positions and speeds. Following each poll of the socket buffer, the Raspberry Pi will poll each sonar sensor sequentially and determine which, if any, sonar states have changed (transitioned from reading values in one range to reading values in a different range). The Raspberry Pi will then construct a string with the number of the sensor that underwent a state change (0 being front-right, 1 being front-left, 2 being back-right, and 3 being back-left) followed by either "c" for close, "o" for okay, or "f" for far.

To determine if communication with the operator is errant or has unexpectedly closed, the program keeps track of the time at which it received the most recent command. For every one second a command is not received, the robot sends the message "<3" to the operator's machine, requesting a response. If the operator's machine is still connected, it will read this message in the buffer and immediately send "<3" back to the Raspberry Pi to indicate that the operator is still connected. If the robot goes two seconds without receiving any response from the operator, the Raspberry Pi stops the motors and the servos powered off. If three seconds pass without a response, loss-of-signal is declared. In the case of loss-of-signal, the robot closes the socket connection and reboots the server, blocking until the client reconnects.

5.4 Servo Interfacing Software

Written By: Joseph Austin

Control of both the servos and motors is performed using the PWM signal driver on the custom Raspberry Pi HAT. To interface with the PWM driver, the control software on the Raspberry Pi writes I2C values to the PWM IC using the Pi's GPIO pins and the pigpio open-source library. The I2C value written indicates which channel to write to and what PWM value to set the channel to. The PWM value written determines either the speed of a motor or continuous-rotation servo or the angle of a positional servo depending on the channel the value was written to.

The positional servos accept a PWM value between X and Y that indicates an angle to rotate to. To control the servos, the control software first converts values from the left stick, right stick, and trigger commands into percentages. The percentages are then multiplied by a servo speed value, which the operator can change with the D-Pad and ranges from 0 to 100. The servo speed integer indicates how much to add or subtract at a time from the current PWM value on any channel. The percentage times the servo speed is then added or subtracted from the corresponding PWM value on the channel of whichever servo the operator is controlling. The operator controls the continuous rotation servos with only digital inputs from the controller (the left and right bumpers for the base, and the D-Pad for the effector). In the case that the operator is driving the base or the effector servos, the program only writes either the maximum speed value in one direction or the stop value to the channel.

5.5 Motor Control Software

Written By: Joseph Austin

The motors accept a PWM value ranging from 0 to 4095, with 0 being stopped and 4095 being max throttle. 4096 is a special integer that entirely powers off the channel. Two direction pins on the motor controller board are set or reset to change the direction of either pair of motors.

The left stick in the X direction is used to steer the robot. The right and left triggers are used for throttle and reverse. However, because the robot's motion platform uses slip steering, the implementation of this control scheme in software is somewhat complicated.

First, the Raspberry Pi reads the value from the left stick (from -9999 to 9999) and converts it to a percentage out of 10,000. If the value is negative, the robot needs to turn left. The left motors would need to be driven slower than the right motors or backward. Likewise, if the value is positive, the robot needs to turn right, and the right motors are driven slower than the left or backward. The percentage value read from the left stick is multiplied by the current maximum steering value (which the operator can manipulate with the D-Pad and ranges from 0 to 4000). Then, the Raspberry Pi reads the value from either the left trigger or the right trigger (with -9999 indicating left being fully pressed and 9999 indicating right being fully pressed) and converts this to a percentage out of 10,000 as well. This percentage is multiplied by the current maximum throttle value (which the operator can also manipulate with the D-Pad and ranges from 0 to 4000). The software then adds the throttle value to the steering value. If the value on the left motors is negative, the left motor direction pin for the left motors is driven low and the value is made positive before it is asserted. Likewise, if the value on the right motors is negative, the right motor direction pin is driven low and the value is made positive. Each value is then written to the corresponding PWM channel to drive each motor. The result is an intuitive system that accommodates both turning in place and turning while driving forward or backward.

5.6 Camera Feedback Software

Written By: Joseph Austin

In order to display video data from the Raspberry Pi camera module to the operator, we made use of an open-source web-streaming interface known as the "Rpi-Cam-Web-Interface." This quickly and seamlessly allows incorporation video streaming into the design, as well as allow the operator to change the stream size and quality on the fly, as they needed. The software simply required initialization at boot via an auto-start script, and video data from the Raspberry Pi camera streams to a web interface over port 4620 automatically. The operator could then connect to the IP of the Raspberry Pi across port 4620 using any modern web browser and log in to the

password-protected interface. From there, the operator has access to the camera stream and video size and quality.

5.7 Ultrasonic Sensor Software

Written By: Joseph Austin

To allow the operator a greater positional awareness during operation, four sonar sensors continuously read distance data and report changes to the operator. Each sonar sensor sends an acoustic pulse once its trigger pin is driven high. The sonar's echo pin is driven high until the sensor receives the acoustic pulse back, at which time the sonar drives its echo pin low. The Raspberry Pi measures the time it takes from driving the trigger pin high to reading a low voltage from the echo pin with a timeout of one second. The sonar interface converts this time reading into a centimeter distance by multiplying by the time reading by a conversion factor. If a sonar reads a state change (i.e. transitions from reading "close" to reading to reading "ok," or "ok" to "far," etc.), the program will construct a string with the number of the sonar followed by "c," "o," or "f," and will send the string to the operator's machine.

6 Specifications

6.1 Critical Specifications

Written By: Joseph Austin

This section defines specifications critical to completing the Mercury Robotics Competition track and all competition objectives.

The competition robot must:

- Be mobile.
- Be able to be controlled over the internet from a location of at least 50 miles away.
- Be able to relay sensor read information back to the operator over the internet during the competition.
- Be able to connect to an 802.11b/g/n Wi-Fi router with an ESSID that is not broadcast and has no security protocol.
- Be able to navigate a dark tunnel.
- Be able to complete the track in under 10 minutes.
- Be able to secure a quarter-inch, steel lag bolt.
- Be able to ascend a 30-degree see-saw incline and descend without dropping the lag bolt.
- Be able to deposit the lag bolt in one of three drop-off points.
- Have a width and length no greater than 18 inches by 18 inches, and have a height no greater than 12 inches.
- Be able to detect loss of signal in under 5 seconds.
- Be able to demonstrate a loss-of-signal routine and a reconnection routine in the event that internet connection is lost. The robot must not move during

the loss-of-signal event. The robot must demonstrate that the operator can control it once connection is restored.
- Use no more than three ports total for internet communication.
- Demonstrate safe and proper usage of lithium polymer batteries.
- Be able to operate for at least 15 minutes continuously.
- Be able to drive at a maximum speed of least at 7 ft/s
- Must weigh less than 10 pounds.

6.2 Desirable Specifications

Written By: Joseph Austin

This section defines specifications that are desirable for the robot to meet.

The competition robot should:

- Be easy to control.
- Keep latency between operator input and robot control over the internet less than 500ms.
- Keep latency of the transmission of sensor data over the internet less than 500ms.
- Achieve a speed of at least 10 feet per second during the sprint section.
- Prevent collision with the track wall and obstacles entirely.
- Complete the track in less than 4 minutes to allow for an additional run if desired.
- Be able to deposit the payload in the highest-scoring drop-off point.
- Have a maximum nominal communication latency no greater than 1000ms.

7 Works Cited

[1] Mercury Robotics, 21 August 2016. [Online]. Available: mercury.okstate.edu. [Accessed 1 January 2017].

[2] C. Bensen, "Drive Motor Sizing Tutorial," Robot Shop, 1 Feburary 2012. [Online]. Available: http://www.robotshop.com/blog/en/drive-motor-sizing-tutorial-3661. [Accessed 1 January 2017].

[3] Cytron Technologies, "User Manual," 1 December 2013. [Online]. Available: http://www.robotshop.com/media/files/pdf/user-manual-mdd10a.pdf. [Accessed 1 January 2017].

[4] Raspberry Pi, "raspberrypi/hats," Raspberry Pi, 3 March 2016. [Online]. Available: https://github.com/raspberrypi/hats. [Accessed 1 January 2017].

[5] Demension Engineering, "A beginners guide to switching regulators," Demension Engineering, [Online]. Available: https://www.dimensionengineering.com/info/switching-regulators. [Accessed 1 January 2017].

[6] Raspberry Pi, "Raspberry Pi 3 Model B," Raspberry Pi, [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. [Accessed 1 January 2017].

[7] Raspberry Pi, "Camera Module," Raspberry Pi, [Online]. Available: https://www.raspberrypi.org/documentation/hardware/camera/. [Accessed 1 January 2017].

[8] NXP, "PCA9685," NXP, 2015.

[9] Texas Intruments, "TXB0108," Texas Intruments, 2014.

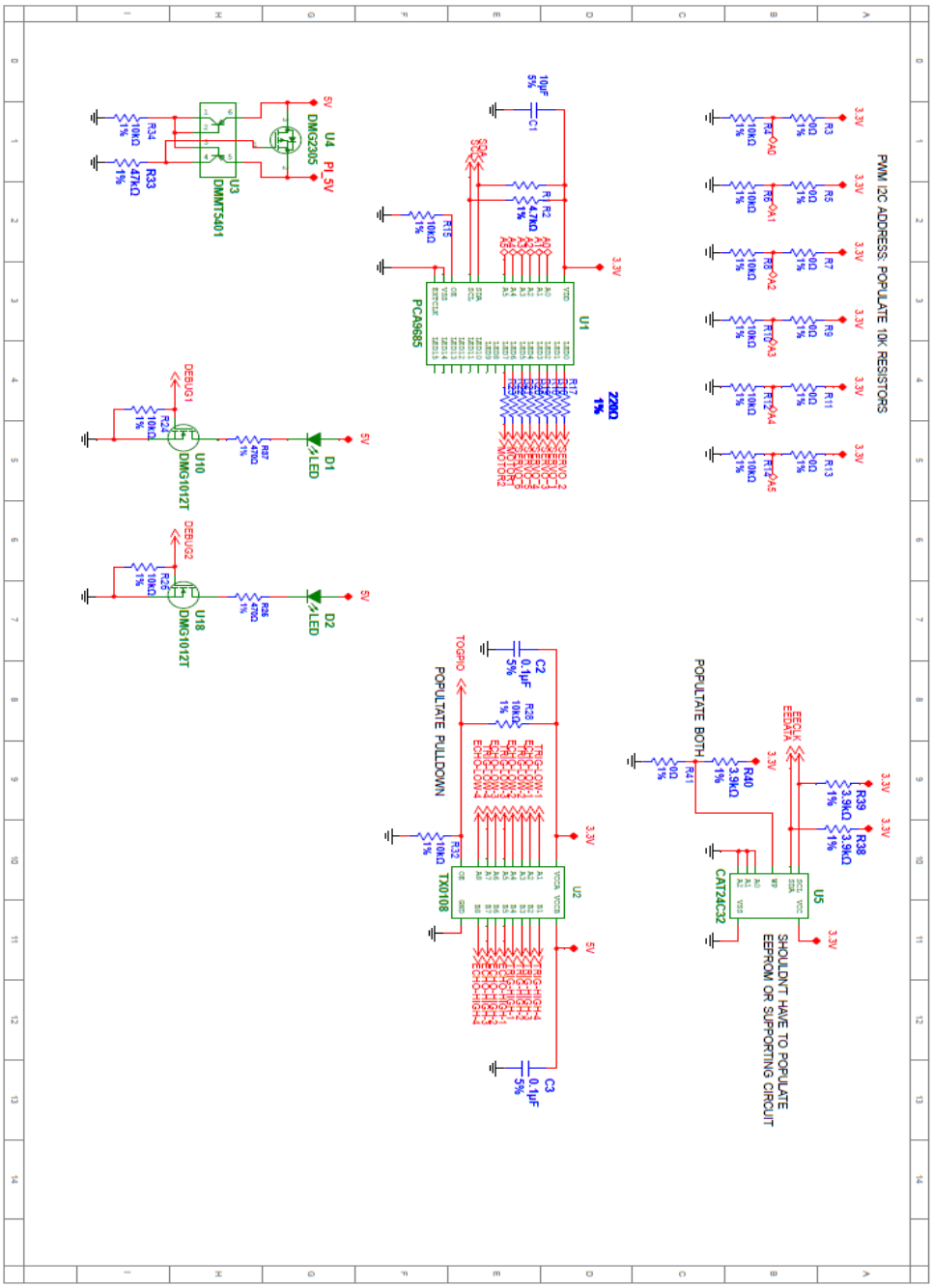
[10] Texas Intruments, "TPS54540," Texas Instruments, 2017.

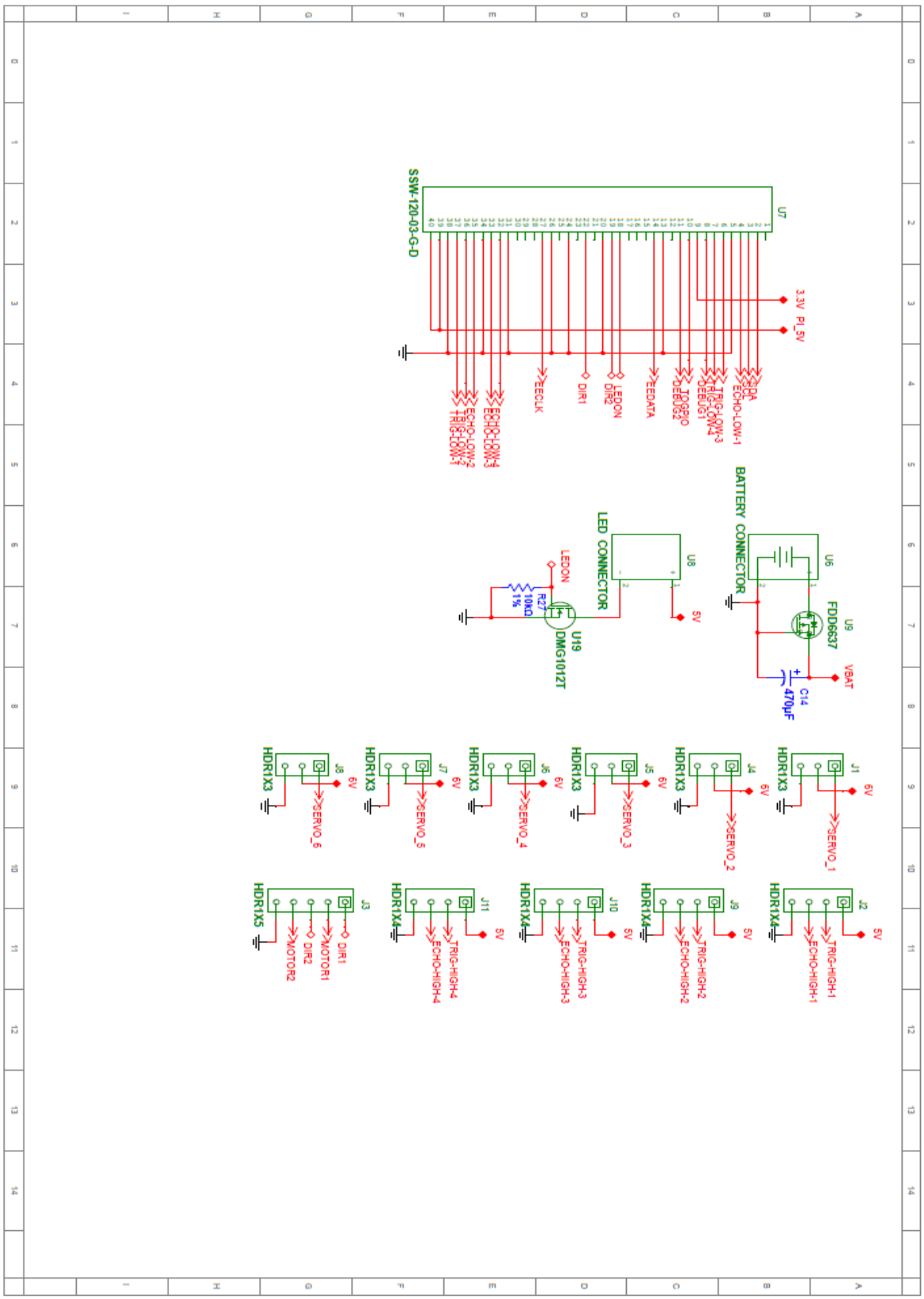
[11] On Semiconductor, "FDD6637," On Semiconductor, 2015.

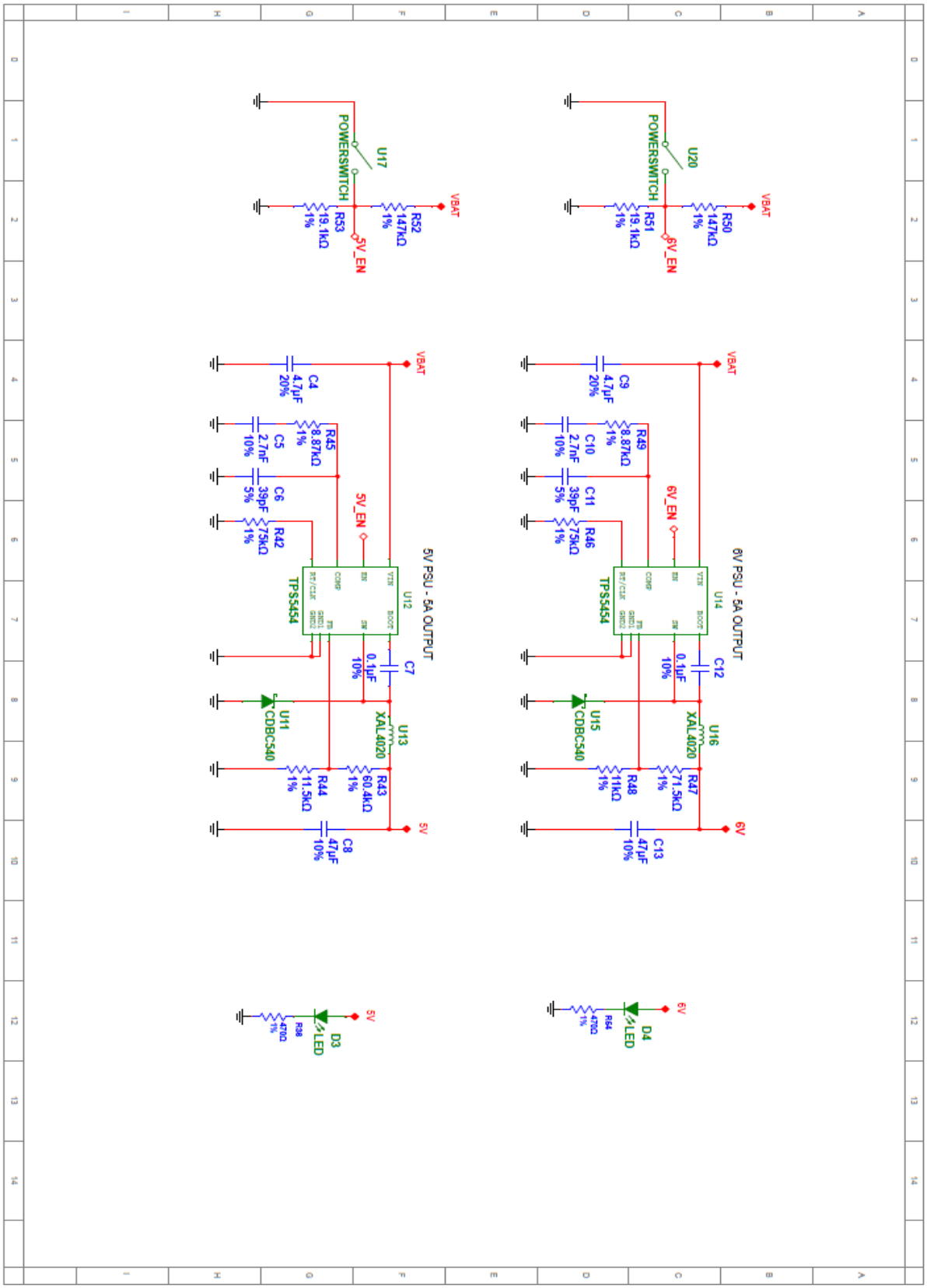
8 APPENDIX

8.1 APPENDIX I - SCHEMATICS

Written By: Kamran Coulter




SSW-120-03-G-D
U7
3.3V
PI_5V
SDA
SCL
ECHO-LOW-1
TRIG-LOW-3
DEBUG1
DEBUG2
EEDATA
LEDON
DIR2
DIR1
EECLK
ECHO-LOW-3
ECHO-LOW-4
ECHO-LOW-2
TRIG-LOW-1
BATTERY CONNECTOR
U6
FDD6637
U9
VBAT
C14
470uF
LED CONNECTOR
U8
5V
LEDON
R27
10MΩ
1%
U19
DMG1012T
HDR1X3
J1
6V
>SERVO_1
HDR1X4
J2
5V
TRIG-HIGH-1
ECHO-HIGH-1
HDR1X3
J4
6V
>SERVO_2
HDR1X4
J9
5V
TRIG-HIGH-2
ECHO-HIGH-2
HDR1X3
J5
6V
>SERVO_3
HDR1X4
J10
5V
TRIG-HIGH-3
ECHO-HIGH-3
HDR1X3
J6
6V
>SERVO_4
HDR1X4
J11
5V
TRIG-HIGH-4
ECHO-HIGH-4
HDR1X3
J7
6V
>SERVO_5
HDR1X3
J8
6V
>SERVO_6
HDR1X5
J3
DIR1
>MOTOR1
>MOTOR2
DIR2


VBAT
U20
POWERSWITCH
R51
19.1kΩ
1%
R50
147kΩ
1%
6V_EN
U17
POWERSWITCH
R53
19.1kΩ
1%
R52
147kΩ
1%
5V_EN
6V PSU - 5A OUTPUT
C9
4.7µF
20%
C10
2.7nF
10%
R49
8.87kΩ
1%
C11
39pF
5%
R46
75kΩ
1%
U14
TPS5454
C12
0.1µF
10%
U15
CDBC540
U16
XAL4020
R48
11kΩ
1%
R47
71.5kΩ
1%
C13
47µF
10%
6V
D4
LED
R64
470Ω
1%
5V PSU - 5A OUTPUT
C4
4.7µF
20%
C5
2.7nF
10%
R45
8.87kΩ
1%
C6
39pF
5%
R42
75kΩ
1%
U12
TPS5454
C7
0.1µF
10%
U11
CDBC540
U13
XAL4020
R44
11.5kΩ
1%
R43
60.4kΩ
1%
C8
47µF
10%
5V
D3
LED
R38
470Ω
1%

8.2 Appendix III – Bill of Materials

Written By: Kamran Coulter, Jonathan Ballew

Description	Qty	Unit Price
MOSFET P-Ch ENH FET -20V 52mOhm -5.0V	2	$0.38
Bipolar Transistors - BJT MATCHED PNP SM SIGNAL TRANS	2	$0.43
Translation - Voltage Levels 8-Bit Bi-directional V-Level Translator	2	$1.53
Standard LEDs - SMD Super Green, 565nm 2.2V, 15mcd	3	$0.17
Standard LEDs - SMD Hyper Red, 645nm 15mcd, 20mA	3	$0.27
LED Display Drivers I2C Bus LED Controller 28-Pin	2	$2.24
MOSFET 35V PCH PowerTrench MOSFET	2	$0.90
Thick Film Resistors - SMD 1/8watt 75Kohms 1% 100ppm	10	$0.049
Switching Voltage Regulators 42V,5A,SD DC-DC Converter	3	$4.17
Thick Film Resistors - SMD 1/8watt 71.5Kohms 1% 100ppm	10	$0.049
Thick Film Resistors - SMD 1/8watt 60.4Kohms 1% 100ppm	10	$0.049
Thick Film Resistors - SMD 1/8watt 11Kohms 1% 100ppm	10	$0.049
Thick Film Resistors - SMD 1/8watt 11.5Kohms 1% 100ppm	10	$0.049
Thick Film Resistors - SMD 1/8watt 8.87Kohms 1% 100ppm	10	$0.049
Fixed Inductors XAL4020 High Current 2.2 uH 20 % 5.5 A	3	$1.80
Schottky Diodes & Rectifiers SCHOTTKY DIODE 5A, 40V (Green)	3	$0.68
Multilayer Ceramic Capacitors MLCC - SMD/SMT 47uF 16Volts 10%	3	$1.27
Multilayer Ceramic Capacitors MLCC - SMD/SMT 4.7uF 25Volts 20%	10	$0.064
Multilayer Ceramic Capacitors MLCC - SMD/SMT 0805 0.1uF 16volts X7R 10%	10	$0.051
Multilayer Ceramic Capacitors MLCC - SMD/SMT 2.7nF 50V X7R 10%	10	$0.037
Multilayer Ceramic Capacitors MLCC - SMD/SMT 39pF 50V NPO 5%	10	$0.048
Thick Film Resistors - SMD 1/8watt 147Kohms 1% 100ppm	10	$0.049
Slide Switches SPDT On-On	3	$0.39
Aluminum Electrolytic Capacitors - SMD 470uF 16 Volts 0.2	2	$0.68
Headers & Wire Housings 5 CKT. 2.5MM ASSY VERTICAL 250V 3A	3	$0.68
Headers & Wire Housings 2.5 TO BOARD HOUSING	10	$0.181
Headers & Wire Housings 2.50MM 4P VERT HDR FRCTN POS LOCK	10	$0.472
Headers & Wire Housings 5 CKT 2.5MM HSNG	3	$0.28
Headers & Wire Housings MiniLock 2.5mm Hdr Vrt 3Ckt Fric&PosLck	10	$0.385
Headers & Wire Housings 2.50MM HSG 01X03P POS LOCK	10	$0.143
Headers & Wire Housings MN-LK TERM 22-28G F Cut Strip of 100	100	$0.055
Thick Film Resistors - SMD 1/8watt 3.9Kohms 1%	10	$0.049
Thick Film Resistors - SMD 1/8watt 47Kohms 1% 100ppm	10	$0.049
Thick Film Resistors - SMD 1/8watt 220ohms 1% 100ppm	20	$0.049
Thick Film Resistors - SMD 1/8watt ZEROohm Jumper	20	$0.042
Thick Film Resistors - SMD 1/8watt 4.7Kohms 1%	10	$0.048

Description	Qty	Unit Price
Multilayer Ceramic Capacitors MLCC - SMD/SMT 0805 10uF 6.3volts X5R 10%	10	$0.083
Thick Film Resistors - SMD 1/8watt 10Kohms 1% 100ppm	30	$0.049
MOSFET MOSFET N-CHANNEL SOT-523	10	$0.198
Fixed Terminal Blocks 2P SIDE ENTRY 2.54mm	3	$0.52
Thick Film Resistors - SMD 1/8watt 470ohms 1% 100ppm	10	$0.049
Thick Film Resistors - SMD 1/8watt 19.1Kohms 1%	10	$0.049
Raspberry Pi 3 Model B	1	$43.75
Raspberry Pi Camera Module V2	1	$29.89
18” Ribbon Cable	1	$6.91
HC-SR04 Ultrasonic Sensors (4 Pack)	1	$9.79
Hitec HS-311 Servo	3	$11.95
Parallax Continuous Rotation Servo	1	$12.09
Unbranded 9g Servo	1	$4.95
32 GB Micro SD Card	1	$9.99
10A Dual Channel DC Motor Driver	1	$23.49
Neodymium Magnets 18mmD x 3mm (3 Pack)	1	$5.49
BaneBots Wheel	8	$3.00
BaneBots Hex Wheel Hub	4	$4.50
Venom 20C 3S 4000mAh 11.1V LiPo Battery	1	$41.99
970 RPM Econ Gearmotor	4	$14.99
5mm x 11mm x5mm Ball Bearings (8 Pack)	1	$7.99
Nylon Screw and Nut Set (2 Pack)	1	$2.95
5mm x 4’ Wooden Dowel	1	$0.75
1.75mm ABS 3D Printing Filament	2	$19.89
*M3 6mm Pan Head Screw 60pcs	1	$6.59
*M3 50mm Brass Standoff Male to Female 30	1	$11.69
*M2.5 Brass Standoff Kit 120pcs	1	$11.99
*Machine Screw Kit	2	$3.49
*Steel Washer Kit	1	$3.49
*Wire Screw Terminal Block (5 Pack)	1	$10.99
*5mm Round White LEDs (100pcs)	1	$3.08

*Starred materials were purchased in large quantities despite the robot only requiring a small number of each. As such their listed prices do not accurately describe the value of the materials used on the robot itself.

8.3 Appendix IV – Advisor Meeting Summary

DATE	SUMMARY
1/19/17	Initial meeting, Meeting times established, Joint meetings agreed upon, General subsystems discussed, Housekeeping task given.
1/26/17	Member roles defined, Initial design discussed, part order procedures established, Told to have design finalized by week 4.
2/09/17	Proposal presentation and specifications discussed, Parts ordered began to arrive, Motors needed to be tested ASAP.
2/16/17	Arm believed to be behind schedule, Servos ordered for arm, Design to be finalized by next week.
2/23/17	Definitive task list from each member requested, Concern over manipulator arm degrees of freedom, PCBs to be ordered within 10 days
3/02/17	Concern over progress on BOT, Specification document requested, Discussion on teams perception of progress.
3/09/17	PCBs ordered, Commercial prototype demo
3/16/17	Fully functional proto demo including sensors, ramp tested, Arm Redesign being finalized.
3/23/17	PCBs arrived and assembled, functional testing through next week, final integration within one week, Bogotá not happening.
3/30/17	Dr. Latino Gone
4/06/17	Dr. Latino Gone
4/13/17	Final competition preparation, little to update, need for more practice
4/20/17	Competition week, no meeting required
4/27/17	No meeting required
5/04/17	No meeting required

8.4 Appendix II – Code

```
# RunThis.py
# Top level module on the Raspberry Pi for operating the robot
# Instantiates other supporting modules for robot operation:
# MessageReceiver, MessageDecoder, and SonarReader.
import sys
from time import sleep
import time
sys.path.append("/home/pi/Desktop/SeniorDesign2/Control/Supporting")
sys.path.append("/home/pi/Desktop/SeniorDesign2/Control/Adafruit")
from MessageReceiver import Receiver
r = Receiver()
from MessageDecoder import Decoder
d = Decoder()
from SonarReader import Sonars
s = Sonars()
msglength = 7
while True:
    try:
        #RECEIVE
        msg = r.receive() #Heartbeat after 1 second of not
                          #receiving anything
        #DECODE, EXECUTE
        #Send a message to operator if there is one
        msgToOperator = d.decode(msg)
        if msg == "STOP!": #An exception occurred
            print "Rebooting Bot."
            r.close()
            r = None
            d.close()
            d = None
            s.close()
            s = None
            sleep(3)
            r = Receiver()
            d = Decoder()
            s = Sonars()
            continue
        elif msg == "Q".ljust(msglength):
            print "Quitting."
            r.close()
            r = None
            d.close()
            d = None
            s.close()
            s = None
            sleep(3)
            r = Receiver()
            d = Decoder()
            s = Sonars()
            continue
        else:
            #Send message from decoder
            if msgToOperator:
                r.send(msgToOperator)

            #Send message from sonars
            msgToOperator = s.readAll()
            if msgToOperator:
                r.send(msgToOperator)
    except Exception as inst:
        print inst
        print "Error, Rebooting Bot."
        sleep(1)
        r.close()
        r = None
        d.close()
        d = None
        s.close()
        s = None
        sleep(3)
        r = Receiver()
        d = Decoder()
        s = Sonars()
        continue
```

```
#MessageReceiver.py
#Hosts socket for communicating with client
#Receives messages and passes message to RunThis.py
#Handles heartbeat and loss of signal
from Server import Server
import RPi.GPIO as GPIO
import socket
import time
from time import sleep
TIMES_TO_TRY = 3
msglength = 7
greenLED = 9
redLED = 22
class Receiver:
    def __init__(self):
        #sleep(30) #Wait for wlan to connect on autostart
        GPIO.setmode(GPIO.BCM)
        GPIO.setwarnings(False)
        GPIO.setup(redLED, GPIO.OUT)
        GPIO.setup(greenLED, GPIO.OUT)
        #Client has not connected yet
        GPIO.output(redLED, GPIO.HIGH)
        GPIO.output(greenLED, GPIO.LOW)
        self.msglength = 7 #Character length of message is 7, longest message is LX-1000
        self.server = Server()
        self.retries = 0
        self.nextCheckTime = time.time() + 1.0
        #Client connected
        GPIO.output(redLED, GPIO.LOW)
        GPIO.output(greenLED, GPIO.HIGH)
    def receive(self):
        msg = ""
        try: #Try to receive a message, except a timeout exception
            msg = self.server.c.recv(msglength)
            #If we don't time out waiting for a message, set retries to 0
            self.retries = 0
            if msg == "Q".ljust(msglength):
                self.server.sock.close()
            return msg #Return to main module to be decoded
        except (socket.timeout, socket.error): #Timed out receiving a message, do heartbeat
            #If the client doesn't respond after the first heartbeat, stop the motors.

            if(time.time() > self.nextCheckTime):
                if(self.retries > 0):
                    msg = "STOP" #Tell the decoder to stop everything
                self.server.c.send("<3") #ask the client if they're still there
                self.retries+=1 #Number of heartbeats sent to client without a response
                msg = "<3"
                self.nextCheckTime = time.time() + 1.0
                if(self.retries >= TIMES_TO_TRY): #Connection lost
                    print "</3"
                    print "Loss of signal."
                    print "Waiting for client to reconnect..."
                    self.retries = 0
                    GPIO.output(greenLED, GPIO.LOW)
                    GPIO.output(redLED, GPIO.HIGH)
                    msg = "STOP!" #Stop the bot if an exception occurs
                    return msg
                else: #Haven't exceeded retry limit, see if they have responded yet.
                    print "<3" #A heartbeat signal was sent
                    return msg #msg is empty or STOP
                    pass
        except Exception as inst: #Except any other exception
            print "An unexpected error occurred in MessageReceiver.py:"
            print inst
            msg = "STOP!" #Stop the bot if an exception occurs
            return msg
    def send(self, msg): self.server.c.send(msg)
    def close(self):
        self.server.sock.close()
        self.server.sock = None
        self.server = None
        #Client will be disconnected
        GPIO.output(redLED, GPIO.HIGH)
        GPIO.output(greenLED, GPIO.LOW)
```

```
#Server.py
#Opens a socket for communication
#with the operator. Handles restarting
#server after loss of signal.
import sys, socket, fcntl, struct
from time import sleep
import SocketServer
import smtplib
SocketServer.TCPServer.allow_reuse_address = True
SocketServer.UDPServer.allow_reuse_address = True
DEFAULT_TIMEOUT = 0 #heartbeat every 1 second
class Server:
    def waitForClient(self):
        self.sock.close()
        sleep(1)
        self.sock = socket.socket()
        self.host = self.get_ip_address("wlan0")
        port = 12345 #Doesn't seem to really matter
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.sock.bind((self.host, port))
        self.sock.settimeout(None)
        self.c.settimeout(None)
        self.sock.listen(1)
        self.c, self.addr = self.sock.accept() #Sit here and wait for a client.
        print "Client connected from", self.addr
        self.c.send('Reconnected to server.')
        #Set the timeout for the heartbeat signal
        self.sock.settimeout(DEFAULT_TIMEOUT)
        self.c.settimeout(DEFAULT_TIMEOUT)
    #Automatically determine the IP address from the Pi's WLAN
    def get_ip_address(self, ifname):
        self.s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        return socket.inet_ntoa(fcntl.ioctl(
            self.s.fileno(),
            0x8915,  # SIOCGIFADDR
            struct.pack('256s', ifname[:15])
        )[20:24])
    def sendEmail(self): #Send email with IP to connect to
        server = smtplib.SMTP_SSL('smtp.gmail.com', 465)
        server.login("rpi.capstone4b@gmail.com", "capstone4b")
        msg = "Catch a ride at {} !".format(self.host)
        recepients = ["josephvcaustin@gmail.com"]
        server.sendmail("rpi.capstone4b@gmail.com", recepients, msg)
        server.quit()
    def start(self):
        #Call start() again if an exception occurs.
        #Close anything that was opened.
        if self.sock is not None:
            self.sock.close()
            self.sock = None
        if self.s is not None:
            self.s.close()
            self.s = None
        if self.c is not None:
            self.c.close()
            self.c = None
        try:
            self.sock = socket.socket()
            self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            self.host = self.get_ip_address("wlan0")
            self.s.close() #s is only used to get the IP
            port = 12345 #Doesn't seem to really matter
            self.sock.bind((self.host, 12345))
            self.sock.settimeout(None)
        except Exception as inst:
            print "Failed to bind socket: {}".format(inst)
            sleep(1)
            self.start()
##        try: self.sendEmail()
##        except Exception as inst:
##            print "Failed to send email: {}".format(inst)
##            sleep(1)
```

```
        a = False
        while(not a):
            try:
                self.sock.listen(5) #Only accept one client
                print "Starting server at", self.host, ": 12345"
                self.c, self.addr = self.sock.accept() #Sit here and wait for a client.
                a = True
                break
            except Exception as inst:
                print "Failed listening for client: {}".format(inst)
                err = "{}".format(inst)
                if "[Errno 11]" in err:
                    print "Passing errno 11."
                    a = True
                    pass
                else:
                    sleep(5)
                    continue
        print "Client connected from", self.addr
        self.c.send('Successfully connected to server.')
        #Set the timeout for the heartbeat signal
        self.sock.settimeout(DEFAULT_TIMEOUT)
        self.c.settimeout(DEFAULT_TIMEOUT)
        return
    def __init__(self):
        SocketServer.TCPServer.allow_reuse_address = True
        SocketServer.UDPServer.allow_reuse_address = True
        self.s = None
        self.sock = None
        self.host = None
        self.c = None
        self.addr = None
        self.start()
#MessageDecoder.py
#Decodes 7-byte string commands
#and controlls motors, servos, and LEDs
from MotorController import MotorController
from ServoController import ServoController
from time import sleep
MOTOR_MODE = -1
SERVO_MODE = 0
ROUTINE_MODE = 1
PRECISION = 1000 #Encoded messages send values from -precision to precision read from analog
msglength = 7
MIN_SERVO_SPEED = 10 #At least increment servo value by 10
MAX_SERVO_SPEED = 100 #At most increment servo value by 100
MAX_MOTOR_SPEED = 4000
STEER_DIRECTION = 1 #Negate this value if steering is backward.
class Decoder:
    def __init__(self):
        self.listening = True #Whether or not the bot should respond to a command
        self.stopped = False
        self.controllingServos = False #Whether or not to control servos this iteration
        self.controllingMotors = False #Whether or not to control motors this iteration
        self.sprint = False
        self.mode = MOTOR_MODE
        self.steeringRange = 2000
        self.motorRange = 2000
        self.servoSpeed = MIN_SERVO_SPEED
        self.steering = 0
        self.throttle = 0
        self.bot = 0
        self.mid = 0
        self.top = 0
        self.sc = ServoController()
        self.mc = MotorController()
        self.mc.stopMotors()
        self.sc.resetServos()
        sleep(0.5)
        self.sc.stopServos()
```

```
    #Snap all servos back to middle position.
    def resetServos(self):
        self.sc.resetServos()
        self.bot = 0
        self.mid = 0
        self.top = 0
    #Write STOP to the motors
    def stopMotors(self):
        self.mc.stopMotors()
        self.steering = 0
        self.throttle = 0
    #Write STOP to the servos (stop powering them)
    def stopServos(self):
        self.sc.stopServos()
        self.bot = 0
        self.mid = 0
        self.top = 0
############## MESSAGE DECODER ##############
    def decode(self, msg):
        msgToOperator = ""
        if not msg: return msgToOperator #If the string is empty, ignore
        #First assume we aren't going to control anything, change if we are
        self.controllingMotors = False
        self.controllingServos = False
        #B button will stop everything and stop listening, or resume listening
        if msg == "B".ljust(msglength):
            if self.listening: #If listening
                self.stopServos()
                self.stopMotors()
                self.listening = False
                msgToOperator = "NOT LISTENING"
            else: #If not listening
                self.listening = True
                msgToOperator = "LISTENING"
        elif msg == "STOP" or msg == "STOP!": #Stop all motors and servos
            self.stopMotors()
            self.stopServos()
            print "Forcing bot to stop."
            return
        #Back button quits
        elif msg == "Q".ljust(msglength):
            self.stopMotors()
            self.stopServos()
            self.mc.lightsOff()
            self.listening = False
            return
        elif not self.listening: return #If not listening, do nothing.
        #A button will turn on headlights
        elif msg == "A".ljust(msglength):
            if (self.mode == MOTOR_MODE):
                msgToOperator = "LIGHTS"
                self.mc.toggleHeadlights()
            elif(self.mode == ROUTINE_MODE):
                self.sc.reachArm()
        #X button will snap servos back to SERVO_MID
        elif msg == "X".ljust(msglength):
            msgToOperator = "SERVO RESET"
            self.resetServos()
        #Y button will power off servos
        elif msg == "Y".ljust(msglength):
            msgToOperator = "SERVO STOP"
            self.stopServos()
        #Start will change the mode of operation
        elif msg == "START".ljust(msglength):
            self.mode += 1
            if(self.mode > 1): self.mode = -1
            if(self.mode == MOTOR_MODE):
                msgToOperator = "DRIVE MODE"
                self.stopMotors()
                self.sc.rotateBase(0)
                self.sc.rotateEff(0)
```

```
    elif (self.mode == SERVO_MODE):
        msgToOperator = "MANIPULATOR MODE"
        self.stopMotors()
    else:
        msgToOperator = "ROUTINE MODE"
        self.stopMotors()
#Rotate the base left or right, stop rotating when a bumper is released
elif msg == "LP".ljust(msglength):
    if(self.mode == ROUTINE_MODE): self.sc.lowerForTunnel()
    else: self.sc.rotateBase(-1) #Rotate base left
elif msg == "LR".ljust(msglength):
    self.sc.rotateBase(0) #Stop rotating
elif msg == "RP".ljust(msglength):
    if(self.mode == ROUTINE_MODE): self.sc.raiseForDelivery()
    else: self.sc.rotateBase(1) #Rotate base right
elif msg == "RR".ljust(msglength):
    self.sc.rotateBase(0) #Stop rotating
#Left Stick X used for steering
elif msg[:2] == "LX":
    lxVal = int(msg[2:]) #Left stick X val from -1000 to 1000
    if (self.mode == MOTOR_MODE):
        self.steering = STEER_DIRECTION * int(float(lxVal)/PRECISION * self.steeringR
        self.controllingMotors=True
#Left Stick Y used to control the middle servo
elif msg[:2] == "LY":
    val = int(msg[2:]) #Left stick Y val from -1000 to 1000
    if (self.mode == SERVO_MODE):
        self.controllingServos=True
        valPercent = float(val)/float(PRECISION)
        self.mid = int(valPercent * self.servoSpeed)*(-1)
elif msg[:2] == "RX":
    val = int(msg[2:]) #Right stick X val from -1000 to 1000
#Right Stick Y Value used to control the top servo
elif msg[:2] == "RY":
    val = int(msg[2:]) #Left stick Y val from -1000 to 1000
    if (self.mode != ROUTINE_MODE):
        self.controllingServos=True
        valPercent = float(val)/float(PRECISION)
        self.top = int(valPercent * self.servoSpeed)
#Right Trigger used to control throttle and the bottom servo
elif msg[:2] == "RT":
    #Multiply val by -1 to flip trigger direction
    val = int(msg[2:]) #trigger val from 0 to 1000
    if (self.mode == MOTOR_MODE):
        self.throttle = int(float(val)/float(PRECISION) * float(self.motorRange)) #Fr
        self.controllingMotors=True
    elif (self.mode == SERVO_MODE):
        self.controllingServos=True
        valPercent = float(val)/float(PRECISION)
        self.bot = int(valPercent * self.servoSpeed)*(-1)
#DPAD X used to increase/decrease steering speed or servo movement speed
elif msg[:2] == "DX":
    val = int(msg[2:]) #Dpad X val (-1 = left, 0 = center, 1 = right)
    if (self.mode == MOTOR_MODE):
        self.steeringRange += 100*val #Increment or decrement the steering speed
        if self.steeringRange > MAX_MOTOR_SPEED: self.steeringRange = MAX_MOTOR_SPEED
        elif self.steeringRange < 0: self.steeringRange = 0
        msgToOperator = "Steer: {}".format(self.steeringRange)
    elif (self.mode == SERVO_MODE):
        self.servoSpeed += 10*val #increment or decrement the servo speed by 10
        if self.servoSpeed < MIN_SERVO_SPEED: self.servoSpeed = MIN_SERVO_SPEED
        elif self.servoSpeed > MAX_SERVO_SPEED: self.servoSpeed = MAX_SERVO_SPEED
        msgToOperator = "Servo: {}".format(self.servoSpeed)
```

```
            else:
                if(val > 0):
                    self.steeringRange = 500
                    self.motorRange = 4000
                    msgToOperator = "SPRINT"
                    self.sprint = True
                elif(val < 0):
                    self.steeringRange = 2000
                    self.motorRange = 2000
                    msgToOperator = "NORMAL"
                    self.sprint = False
        #DPAD Y used to increase/decrease max throttle or to extend/retract the effector
        elif msg[:2] == "DY":
            val = int(msg[2:]) #Dpad Y val (-1 = down, 0 = center, 1 = up)
            if (self.mode == MOTOR_MODE):
                self.motorRange += 100*val #Increment or decrement the max speed
                if self.motorRange > MAX_MOTOR_SPEED: self.motorRange = MAX_MOTOR_SPEED
                elif self.motorRange < 0: self.motorRange = 0
                msgToOperator = "Throttle: {}".format(self.motorRange)
            elif (self.mode == SERVO_MODE): self.sc.rotateEff(val)
            else:
                if(self.sprint):
                    if val < 0: self.mc.trimVal -= 10
                    elif val > 0: self.mc.trimVal += 10
                    msgToOperator = "Trim {}".format(self.mc.trimVal)
                else:
                    if (val < 0): self.sc.retractEff()
                    elif (val > 0): self.sc.extendEff()
        #If need to update throttle/steering on motors, tell motorcontroller to update it
        if (self.controllingMotors): self.mc.controlMotors(self.steering, self.throttle)
        #If need to update servo positions, tell servocontroller to update them
        if (self.controllingServos): self.sc.controlServos(self.bot, self.mid, self.top)
        return msgToOperator
    def close(self):
        self.stopMotors()
        self.stopServos()
        self.mc = None
        self.sc = None

#MotorController.py
#Receives arguments from MessageDecoder.py
#and exerts control over the motors and LEDs.
from Adafruit_PWM_Servo_Driver_PIGPIO import PWM
import RPi.GPIO as GPIO
import warnings
warnings.filterwarnings("ignore") #Ignore GPIO warnings
### Motor Pins ###
LEFT_MOTOR_CHANNEL = 6
RIGHT_MOTOR_CHANNEL = 7
LEFT_DIRECTION_PIN = 20
RIGHT_DIRECTION_PIN = 26
##################
HEADLIGHTS = 19
### Booleans to indicate motor direction ###
RIGHT_MOTOR_FORWARD = False #Change to True if motors are backward
RIGHT_MOTOR_BACKWARD = not RIGHT_MOTOR_FORWARD #Opposite of forward
LEFT_MOTOR_FORWARD = RIGHT_MOTOR_FORWARD #Opposite of right
LEFT_MOTOR_BACKWARD = RIGHT_MOTOR_BACKWARD
############################################
MAX_SPEED = 4090 #Max Speed
STOP = 4096
THRESHOLD = 300 #Speed threshold for considering to be stopped to prevent buzzing.
FREQ = 100 #Motor PWM frequency
HAT_ADDR = 0x40 #Pi hat address
class MotorController:
    def __init__(self):
        self.pwm = PWM(HAT_ADDR)
        self.pwm.setPWMFreq(FREQ)
```

```
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(RIGHT_DIRECTION_PIN, GPIO.OUT)
        GPIO.setup(LEFT_DIRECTION_PIN, GPIO.OUT)
        GPIO.setup(HEADLIGHTS, GPIO.OUT)
        GPIO.output(RIGHT_DIRECTION_PIN, RIGHT_MOTOR_FORWARD)
        GPIO.output(LEFT_DIRECTION_PIN, LEFT_MOTOR_FORWARD)
        GPIO.output(HEADLIGHTS, GPIO.LOW)
        self.pwm.setPWM(RIGHT_MOTOR_CHANNEL, 0, STOP)
        self.pwm.setPWM(LEFT_MOTOR_CHANNEL, 0, STOP)
        self.lightsOn = False
        self.trimVal = 150
    def controlMotors(self, steering, throttle):
        ##################### Calculate motor values ########################
        #Motors will turn the bot based on the LX steering value plus
        #the value of throttle to allow both turning in place and
        #turning while going forward/back.
        newRightVal = 0
        newLeftVal = 0
        if (steering == 0): #Both motors going straight
            newRightVal = throttle
            newLeftVal = throttle
        else:
            newRightVal = (((-1)*steering) + throttle) #Right motor speed is always opposite
            newLeftVal = steering+throttle
        #Write "stop" to prevent motors buzzing at low speeds.
        if abs(newRightVal) < THRESHOLD: newRightVal = STOP
        elif(newRightVal < 0): #Right motors are running backward.
            GPIO.output(RIGHT_DIRECTION_PIN, RIGHT_MOTOR_BACKWARD)
            newRightVal*=-1 #newRightVal is negative, so make it positive before asserting.
        else: GPIO.output(RIGHT_DIRECTION_PIN, RIGHT_MOTOR_FORWARD)
        #Write "stop" to prevent motors buzzing at low speeds.
        if abs(newLeftVal) < THRESHOLD: newLeftVal = STOP
        elif(newLeftVal < 0): #Left motors are running backward.
            GPIO.output(LEFT_DIRECTION_PIN, LEFT_MOTOR_BACKWARD)
            newLeftVal*=-1 #newLeftVal is negative, so make it positive before asserting.
        else: GPIO.output(LEFT_DIRECTION_PIN, LEFT_MOTOR_FORWARD)
        ######################################################################
        if newRightVal != STOP: newRightVal -= self.trimVal
        if newRightVal < THRESHOLD: newRightVal = STOP
        #Assert control on right and left motors
        self.pwm.setPWM(RIGHT_MOTOR_CHANNEL, 0, newRightVal)
        self.pwm.setPWM(LEFT_MOTOR_CHANNEL, 0, newLeftVal)
        #print "Right = {}".format(newRightVal)
        #print "Left = {}".format(newLeftVal)
    def stopMotors(self):
        self.controlMotors(0, STOP)
    def lightsOff(self):
        GPIO.output(HEADLIGHTS, GPIO.LOW)
    def toggleHeadlights(self):
        self.lightsOn = not self.lightsOn
        if self.lightsOn: GPIO.output(HEADLIGHTS, GPIO.HIGH)
        else: GPIO.output(HEADLIGHTS, GPIO.LOW)
```

```
#ServoController.py
#Receives arguments from MessageDecoder.py
#and exerts control over the servos. Also runs
#programmed routines on manipulator arm.
from Adafruit_PWM_Servo_Driver_PIGPIO import PWM
from time import sleep
#Max and min assertable PWM values given freq=100
#SERVO_MIN = 307
#SERVO_MID_VAL = 635
#SERVO_MAX = 921
SERVO_MIN = 250
SERVO_MID_VAL = 635
SERVO_MAX = 1050
#Servo pin values
SERVO_EFF = 0
SERVO_TOP = 1
SERVO_MID = 2
SERVO_BOT = 3
SERVO_BASE = 4
```

```
FREQ = 100 #PWM freq
HAT_ADDR = 0x40 #Pi hat address
class ServoController:
    def __init__(self):
        self.servoTopVal = STOP #Value to assert on servo
        self.servoBotVal = STOP
        self.servoMidVal = STOP
        self.pwm = PWM(HAT_ADDR)
        self.pwm.setPWMFreq(FREQ)
        #setPWM(pin, channel, value)
        self.pwm.setPWM(SERVO_BOT, 0, self.servoBotVal)
        self.pwm.setPWM(SERVO_MID, 0, self.servoMidVal)
        self.pwm.setPWM(SERVO_TOP, 0, self.servoTopVal)
    #Write a specific value to all the servos
    def setServos(self, bot, mid, top):
        self.servoBotVal = bot
        self.servoMidVal = mid
        self.servoTopVal = top
        if(self.servoBotVal > SERVO_MAX) and not (self.servoBotVal == STOP): self.servoBotVal = SERVO_MAX
        elif(self.servoBotVal < SERVO_MIN): servoBotVal = SERVO_MIN
        if(self.servoMidVal > SERVO_MAX) and not (self.servoMidVal == STOP): self.servoMidVal = SERVO_MAX
        elif(self.servoMidVal < SERVO_MIN): self.servoMidVal = SERVO_MIN
        if(self.servoTopVal > SERVO_MAX) and not (self.servoTopVal == STOP): self.servoTopVal = SERVO_MAX
        elif(self.servoTopVal < SERVO_MIN): self.servoTopVal = SERVO_MIN
        self.pwm.setPWM(SERVO_BOT, 0, self.servoBotVal)
        self.pwm.setPWM(SERVO_MID, 0, self.servoMidVal)
        self.pwm.setPWM(SERVO_TOP, 0, self.servoTopVal)
    #Assert a new value on the servos by adding the value passed in to the existing PWM value
    def controlServos(self, bot, mid, top):
        self.servoBotVal += bot
        self.servoMidVal += mid
        self.servoTopVal += top
        if(self.servoBotVal > SERVO_MAX) and not (self.servoBotVal == STOP): self.servoBotVal = SERVO_MAX
        elif(self.servoBotVal < SERVO_MIN): servoBotVal = SERVO_MIN
        if(self.servoMidVal > SERVO_MAX) and not (self.servoMidVal == STOP): self.servoMidVal = SERVO_MAX
        elif(self.servoMidVal < SERVO_MIN): self.servoMidVal = SERVO_MIN
        if(self.servoTopVal > SERVO_MAX) and not (self.servoTopVal == STOP): self.servoTopVal = SERVO_MAX
        elif(self.servoTopVal < SERVO_MIN): self.servoTopVal = SERVO_MIN
        self.pwm.setPWM(SERVO_BOT, 0, self.servoBotVal)
        self.pwm.setPWM(SERVO_MID, 0, self.servoMidVal)
        self.pwm.setPWM(SERVO_TOP, 0, self.servoTopVal)
    def rotateBase(self, direction):
        #-1 = CCW, 0 = Stop, 1 = CW
        #Use a fixed speed that isn't the max speed
        if(direction < 0): self.pwm.setPWM(SERVO_BASE, 0, SERVO_MIN+300)
        elif(direction > 0): self.pwm.setPWM(SERVO_BASE, 0, SERVO_MAX-300)
        else: self.pwm.setPWM(SERVO_BASE, 0, STOP)
    def rotateEff(self, direction):
        #-1 = Retract, 0 = Stop, 1 = Extend
        #Just use the max speed
        if(direction < 0): self.pwm.setPWM(SERVO_EFF, 0, SERVO_MIN)
        elif(direction > 0): self.pwm.setPWM(SERVO_EFF, 0, SERVO_MAX)
        else: self.pwm.setPWM(SERVO_EFF, 0, STOP)
    def resetServos(self):
        self.setServos(679, 698, 722)
        self.rotateBase(0)
        self.rotateEff(0)
    def stopServos(self):
        self.setServos(STOP, STOP, STOP)
        self.servoBotVal = SERVO_MID_VAL
        self.servoMidVal = SERVO_MID_VAL
        self.servoTopVal = SERVO_MID_VAL
        self.rotateBase(0)
        self.rotateEff(0)
    def reachArm(self):
        self.resetServos()
        while(self.servoBotVal < 800):
            self.controlServos(20, 0, 0)
            sleep(0.1)
        while(self.servoTopVal < 840):
            self.controlServos(0, 0, 20)
            sleep(0.1)
        while(self.servoBotVal < 970):
            self.controlServos(20, 0, 0)
            sleep(0.1)
        while(self.servoMidVal < 550):
            self.controlServos(0,20,0)
            sleep(0.1)
```

```
    def raiseForDelivery(self):
        self.resetServos()
        sleep(0.1)
        while(self.servoMidVal < 1050):
            self.controlServos(0, 30, 0)
            sleep(0.1)
        while(self.servoTopVal < 870):
            self.controlServos(0, 0, 30)
            sleep(0.1)
    def lowerForTunnel(self):
        self.resetServos()
        sleep(0.1)
        while(self.servoMidVal < 920):
            self.controlServos(0, 10, 0)
            sleep(0.1)
        while(self.servoBotVal > 450):
            self.controlServos(-10, 0, 0)
            sleep(0.1)
        while(self.servoBotVal < 500):
            self.controlServos(10, 0, 0)
            sleep(0.1)
    def retractEff(self):
        self.rotateEff(-1)
        sleep(5.2)
        self.rotateEff(0)
    def extendEff(self):
        self.rotateEff(1)
        sleep(5.0)
        self.rotateEff(0)
```

```
#SonarReader.py
#Polls all sonar sensors
#and constructs a string to send
#to the operator if any sonar states change.
import SonarTriggerEcho as sensor
import pigpio
import time
class Sonars:
    def __init__(self):
        RF_TRIG = 24
        RF_ECHO = 14
        LF_TRIG = 27
        LF_ECHO = 4
        RB_TRIG = 23
        RB_ECHO = 15
        LB_TRIG = 17
        LB_ECHO = 18
        TOGPIO = 10
        #list = [rf, lf, rb, lb]
        self.pi = pigpio.pi()
        self.pi.write(TOGPIO, 1)
        self.sensors = (sensor.ranger(self.pi, RF_TRIG, RF_ECHO),sensor.ranger(pigpio.pi(), L
        #self.sensors = (sensor.ranger(self.pi, RF_TRIG, RF_ECHO),sensor.ranger(pigpio.pi(),
        self.CLOSE = -1
        self.OK = 0
        self.FAR = 1
        self.states = [self.FAR, self.FAR, self.FAR, self.FAR]
        self.distances = [0.0, 0.0, 0.0, 0.0]
    def readAll(self):
        msg = ""
        # .read() returns raw micros value. cm = micros/1000000.0*34030/2
        CLOSE = 30 #Less than CLOSE (in cm) = getting too close
        FAR = 100 #Greater than 100cm = too far for good reading
        ERROR_LOW = 6 #Sensors erroneously produce reads between ERROR_LOW and ERROR_HIGH
        ERROR_HIGH = 7
        for i in range(len(self.sensors)): #Read each sensor sequentially
            self.distances[i] = self.sensors[i].read()/1000000.0*34030/2
            #print(self.distances[i])
            if self.distances[i] > ERROR_LOW and self.distances[i] < ERROR_HIGH: continue #Ba
            elif self.distances[i] < CLOSE: #If this sensor reads too close
                if self.states[i] != self.CLOSE:
                    self.states[i] = self.CLOSE
                    msg += "{}c".format(i)
                    #Send "i is too close"
```

```
            elif self.distances[i] < FAR: #Between CLOSE and FAR
                if self.states[i] != self.OK:
                    self.states[i] = self.OK
                    msg += "{}o".format(i)
                    #Send "i is ok"
            else: #We're too far for consistent readings
                if self.states[i] != self.FAR:
                    self.states[i] = self.FAR
                    msg += "{}f".format(i)
                    #Send "i is too far"
        return msg
    def close(self):
        for i in range(len(self.sensors)): self.sensors[i].cancel()
        self.pi.stop()
```

```
#!/usr/bin/env python
#SonarTriggerEcho.py
#Open-source library for reading
#distances from sonars and timing out
#if a sonar doesn't respond.
import time
import pigpio
class ranger:
   """
   This class encapsulates a type of acoustic ranger.  In particular
   the type of ranger with separate trigger and echo pins.

   A pulse on the trigger initiates the sonar ping and shortly
   afterwards a sonar pulse is transmitted and the echo pin
   goes high.  The echo pins stays high until a sonar echo is
   received (or the response times-out).  The time between
   the high and low edges indicates the sonar round trip time.
   """
   def __init__(self, pi, trigger, echo):
      """
      The class is instantiated with the Pi to use and the
      gpios connected to the trigger and echo pins.
      """
      self.pi    = pi
      self._trig = trigger
      self._echo = echo
      self._ping = False
      self._high = None
      self._time = None
      self._triggered = False
      self._trig_mode = pi.get_mode(self._trig)
      self._echo_mode = pi.get_mode(self._echo)
      pi.set_mode(self._trig, pigpio.OUTPUT)
      pi.set_mode(self._echo, pigpio.INPUT)
      self._cb = pi.callback(self._trig, pigpio.EITHER_EDGE, self._cbf)
      self._cb = pi.callback(self._echo, pigpio.EITHER_EDGE, self._cbf)
      self._inited = True
   def _cbf(self, gpio, level, tick):
      if gpio == self._trig:
         if level == 0: # trigger sent
            self._triggered = True
            self._high = None
      else:
         if self._triggered:
            if level == 1:
               self._high = tick
            else:
               if self._high is not None:
                  self._time = tick - self._high
                  self._high = None
                  self._ping = True
```

```
   def read(self):
      """
      Triggers a reading.  The returned reading is the number
      of microseconds for the sonar round-trip.
      round trip cms = round trip time / 1000000.0 * 34030
      """
      if self._inited:
         self._ping = False
         self.pi.gpio_trigger(self._trig)
         start = time.time()
         while not self._ping:
            if (time.time()-start) > 0.01:
               return 20000
            time.sleep(0.001)
         return self._time
      else:
         return None

   def cancel(self):
      """
      Cancels the ranger and returns the gpios to their
      original mode.
      """
      if self._inited:
         self._inited = False
         self._cb.cancel()
         self.pi.set_mode(self._trig, self._trig_mode)
         self.pi.set_mode(self._echo, self._echo_mode)
#RunThis.py
#Top level module on operator's machine for driving
#the robot over the internet. Sends 7-byte string
#commands to robot to be decoded and executed.
#Renders Sonar data to operator using a GUI.
import sys
from time import sleep
from XboxController import XboxController
from Client import Client
import socket
from SonarGUI import SonarGUI
controller = XboxController()
gui = SonarGUI(controller.screen)
sleep(0.5) #Delay between setting up controller and communicating with bot
client = Client()
connected = True
client.sock.setblocking(0) #Non-blocking, time out immediately if buffer is empty
msglength = 7 #character length of each instruction string
              #length of 7 based on maxval being 1000, longest instruction
              #is LX-1000 (7 characters)
#Values used to only send if they change
lsx_prev = 0 #Left Stick X
lsy_prev = 0 #Left Stick Y
rsx_prev = 0 #Right Stick X
rsy_prev = 0 #Right Stick Y
trigger_prev = 0 #Right Trigger and left triggers
dpad_x_prev = 0
dpad_y_prev = 0
A_prev = False
B_prev = False
X_prev = False
Y_prev = False
LB_prev = False
RB_prev = False
BACK_prev = False
START_prev = False
GUIDE_prev = False
polltime = 0.1 #Time between controller input reads and packet sending
maxval = 1000 #Encode stick and trigger input as values from -max to max
retries = 0
TIMES_TO_TRY = 30 #Times to check if something is in the buffer before determining LOS
#Sonar Constants
OK = 0
CLOSE = 1
FAR = 2
```

```
#Poll input, encode, and send
while True:
    #Only send 4 messages per poll
    buttonMsg = ""
    stickMsgX = ""
    stickMsgY = ""
    triggerMsg = ""
    sleep(polltime) #Sleep to prevent flooding the buffer with commands
    controller.getInput() #read all values from controller
    if controller.quitting: #Quitting handled by XboxController object
        client.send("Q".ljust(msglength))
        client.sock.close()
        sys.exit()
    #If the operator's side boots and can't connect,
    #Try again until connection is established.
    if not connected:
        if not client.reconnect(): continue
        else: connected = True
    #Only send "A", "B", "X", "Y", "BACK", or "START"
    #one time when the button is pressed down.
    #Suppress multiple button input (not needed)
    if (A_prev == False) and (controller.A == True):
        A_prev = True
        buttonMsg ="A".ljust(msglength)
    elif (controller.A == False):
        A_prev = False
    if (B_prev == False) and (controller.B == True):
        B_prev = True
        buttonMsg = "B".ljust(msglength)
    elif (controller.B == False):
        B_prev = False
    if (X_prev == False) and (controller.X == True):
        X_prev = True
        buttonMsg = "X".ljust(msglength)
    elif (controller.X == False):
        X_prev = False
    if (Y_prev == False) and (controller.Y == True):
        Y_prev = True
        buttonMsg = "Y".ljust(msglength)
    elif (controller.Y == False):
        Y_prev = False
    if (BACK_prev == False) and (controller.BACK == True):
        BACK_prev = True
        buttonMsg = "BACK".ljust(msglength)
    elif (controller.BACK == False):
        BACK_prev = False
    if (START_prev == False) and (controller.START == True):
        START_prev = True
        buttonMsg = "START".ljust(msglength)
    elif (controller.START == False):
        START_prev = False
    #Send "LP" when LB is pressed, send "LR" when it is released.
    #Used to indicate the bumper is being held.
    if (LB_prev == False) and (controller.LB == True):
        LB_prev = True
        buttonMsg = "LP".ljust(msglength)
    elif (LB_prev == True) and (controller.LB == False):
        LB_prev = False
        buttonMsg = "LR".ljust(msglength)

    #Send "RP" when RB is pressed, send "RR" when it is released.
    #Used to indicate the bumper is being held.
    if (RB_prev == False) and (controller.RB == True):
        RB_prev = True
        buttonMsg = "RP".ljust(msglength)
    elif (RB_prev == True) and (controller.RB == False):
        RB_prev = False
        buttonMsg = "RR".ljust(msglength)
```

```
if (lsx_prev != controller.lsx_val):
    lsx_prev = controller.lsx_val
    val = int(maxval*controller.lsx_val)
    stickMsgX = "LX{}".format(val).ljust(msglength)
if (lsy_prev != controller.lsy_val):
    lsy_prev = controller.lsy_val
    val = int(maxval*controller.lsy_val)
    stickMsgY = "LY{}".format(val).ljust(msglength)
if (rsx_prev != controller.rsx_val):
    rsx_prev = controller.rsx_val
    val = int(maxval*controller.rsx_val)
    stickMsgX = "RX{}".format(val).ljust(msglength)
if (rsy_prev != controller.rsy_val):
    rsy_prev = controller.rsy_val
    val = int(maxval*controller.rsy_val)
    stickMsgY = "RY{}".format(val).ljust(msglength)
if (trigger_prev != controller.trigger_val):
    trigger_prev = controller.trigger_val
    val = int(maxval*controller.trigger_val)
    triggerMsg = "RT{}".format(val).ljust(msglength)
if (dpad_x_prev != controller.dpad_x):
    dpad_x_prev = controller.dpad_x
    buttonMsg = "DX{}".format(dpad_x_prev).ljust(msglength)
if (dpad_y_prev != controller.dpad_y):
    dpad_y_prev = controller.dpad_y
    buttonMsg = "DY{}".format(dpad_y_prev).ljust(msglength)
#Only send one type of each message at a time
if buttonMsg: client.send(buttonMsg)
if triggerMsg: client.send(triggerMsg)
if stickMsgX: client.send(stickMsgX)
if stickMsgY: client.send(stickMsgY)
#Check and see if there's anything in the buffer.
try:
    msg = client.sock.recv(1024)
    #Bot probably sent back some sonar info
    sonars = [-1, -1, -1, -1]
    for i in range(len(sonars)):
        if "%sc"%(i) in msg:
            sonars[i] = CLOSE
            strn = "{}c".format(i)
            msg = msg.replace(strn, "")
        if "%so"%(i) in msg:
            sonars[i] = OK
            strn = "{}o".format(i)
            msg = msg.replace(strn, "")
        if "%sf"%(i) in msg:
            sonars[i] = FAR
            strn = "{}f".format(i)
            msg = msg.replace(strn, "")
    gui.updateSonars(sonars)
    if msg: print msg
    client.send("h".ljust(msglength)) #Tell server "I heard you"
    retries = 0
except socket.timeout as inst: #Except the timeout exception if buffer is empty.
    print retries
    retries+=1
    if(retries > TIMES_TO_TRY):
            retries = 0
            print "Lost connection to server."
            connected = False
    continue
except Exception as inst:
    #[Errno 10035] A non-blocking socket operation could not be completed immediately
    #Will usually occur from trying to read from the pipe
    continue
```

```
#XboxController.py
#Xbox Controller object used for reading input from
#a Microsoft Xbox 360 controller. Saves values read
#from controller to variables.
import sys, pygame
from time import sleep
from SonarGUI import SonarGUI
###Button Constants###
A_BUTTON = 0
B_BUTTON = 1
X_BUTTON = 2
Y_BUTTON = 3
LBUMPER = 4
RBUMPER = 5
BACK = 6
START = 7
GUIDE = 8
########################
###Trigger Constants###
LEFTSTICK_X = 0
LEFTSTICK_Y = 1
RIGHTSTICK_X = 4
RIGHTSTICK_Y = 3
TRIGGER = 2
DEADZONE = 0.2 #Ignore analog values from sticks less
                #than 0.2 on either side for better control.
                #A reading of 0 indicated perfectly centered.
class XboxController:
    def __init__(self):
        ###Draw the frame on the screen###
        pygame.init()
        size = 520, 520
        self.screen = pygame.display.set_mode(size)
        pygame.event.set_allowed(pygame.QUIT) #Make event polling faster by only checking fo
        ###################################
        self.xbox = None
        pygame.joystick.init()
        try:
            self.xbox = pygame.joystick.Joystick(0) #assume only Xbox controller is plugged
            self.xbox.init() #start the joystick
        except:
            print "No Xbox controller is connected."
            pygame.display.quit()
            sys.exit()
        ###Current Values###
        self.lsx_val = 0 #Left Stick X
        self.lsy_val = 0 #Left Stick Y
        self.rsx_val = 0 #Right Stick X
        self.rsy_val = 0 #Right Stick Y
        self.trigger_val = 0 #Right and left triggers
        self.A = False
        self.B = False
        self.X = False
        self.Y = False
        self.LB = False
        self.RB = False
        self.BACK = False
        self.START = False
        self.GUIDE = False #Exclusively for quitting robot operation entirely
        self.dpad_x = 0
        self.dpad_y = 0
        self.quitting = False #Used for closing the frame
    #Call to read all controller input values
    #Should be called in a loop to poll for controller input
    #Loop must check if the quitting variable is set to create a breakpoint
    def getInput(self):
        self.quitting = False
        event = pygame.event.poll() #Read all values on the controller
        if event.type == pygame.QUIT: #Used for closing the frame cleanly.
            self.quitting = True
```

```
if(abs(self.xbox.get_axis(LEFTSTICK_X)) > DEADZONE):
    self.lsx_val = self.xbox.get_axis(LEFTSTICK_X)
else: self.lsx_val = 0
if(abs(self.xbox.get_axis(LEFTSTICK_Y)) > DEADZONE):
    self.lsy_val = self.xbox.get_axis(LEFTSTICK_Y)
else: self.lsy_val = 0
if(abs(self.xbox.get_axis(RIGHTSTICK_X)) > DEADZONE):
    self.rsx_val = self.xbox.get_axis(RIGHTSTICK_X)
else: self.rsx_val = 0
if(abs(self.xbox.get_axis(RIGHTSTICK_Y)) > DEADZONE):
    self.rsy_val = self.xbox.get_axis(RIGHTSTICK_Y)
else: self.rsy_val = 0
self.trigger_val = self.xbox.get_axis(TRIGGER) #Z axis value, -1 if RT fully pressed
self.dpad_x, self.dpad_y = self.xbox.get_hat(0) #Returns an (x, y) tuple
if(self.xbox.get_button(A_BUTTON)):
    #print "A"
    self.A = True
else: self.A = False
if(self.xbox.get_button(B_BUTTON)):
    #print "B"
    self.B = True
else: self.B = False
if(self.xbox.get_button(X_BUTTON)):
    #print "X"
    self.X = True
else: self.X = False
if(self.xbox.get_button(Y_BUTTON)):
    #print "Y"
    self.Y = True
else: self.Y = False
if(self.xbox.get_button(LBUMPER)):
    #print "LB"
    self.LB = True
else: self.LB = False
if(self.xbox.get_button(RBUMPER)):
    #print "RB"
    self.RB = True
else: self.RB = False
if(self.xbox.get_button(BACK)):
    #print "BACK"
    self.BACK = True
    self.quitting = True
else: self.BACK = False
if(self.xbox.get_button(START)):
    #print "START"
    self.START = True
else: self.START = False
if(self.xbox.get_button(GUIDE)):
    #print "GUIDE"
    self.GUIDE = True
    self.quitting = True
if self.quitting:
    print "Quitting."
    pygame.display.quit()
```

```
#SonarGUI.py
#Draws a GUI to the screen with
#symbols to indicate sonar readings.
import sys, pygame
from time import sleep
class SonarGUI:
    def __init__(self, scr):

        ###Draw the frame on the screen###
        self.screen = scr
        self.GREY = 50, 50, 50
        self.screen.fill(self.GREY)
        pygame.display.set_caption('Robot Control Interface')
        botImg = pygame.image.load("bot.png")
        self.warnImg = pygame.image.load("warning.png")
        self.okImg = pygame.image.load("ok.png")
        self.qImg = pygame.image.load("q.png")
        pygame.display.flip()
        #0  1  2  3
        #RF LF RB LB
        self.imgPositions = [(384, 0), (0, 0), (384, 384), (0, 384)]
        self.screen.blit(botImg, (107, 107))
        pygame.event.set_allowed(pygame.QUIT)
        pygame.mixer.music.load("buzzer.mp3")
        ###################################
        self.OK = 0
        self.CLOSE = 1
        self.FAR = 2
        self.sonars = [self.FAR, self.FAR, self.FAR, self.FAR]
        for i in range(len(self.sonars)):
            self.screen.blit(self.qImg, self.imgPositions[i])
        pygame.display.flip()
    def updateSonars(self, readings):
        pygame.mixer.music.rewind()
        for i in range(len(self.sonars)):
            if self.sonars[i] != readings[i]: #Only update if it changes
                if readings[i] is self.OK:
                    pygame.draw.rect(self.screen, self.GREY,
                                     pygame.Rect(self.imgPositions[i], (128, 128))) #Draw a r
                    self.screen.blit(self.okImg, self.imgPositions[i])
                    self.sonars[i] = self.OK
                elif readings[i] is self.CLOSE:
                    pygame.draw.rect(self.screen, self.GREY,
                                     pygame.Rect(self.imgPositions[i], (128, 128))) #Draw a r
                    self.screen.blit(self.warnImg, self.imgPositions[i])
                    self.sonars[i] = self.CLOSE
                    pygame.mixer.music.play() #Play warning sound
                elif readings[i] is self.FAR:
                    pygame.draw.rect(self.screen, self.GREY,
                                     pygame.Rect(self.imgPositions[i], (128, 128))) #Draw a r
                    self.screen.blit(self.qImg, self.imgPositions[i])
                    self.sonars[i] = self.FAR
        pygame.display.flip()
```