Robot Project Final Report

Engineering 1282.01H

Spring 2018

Team E4

Zach Boledovic

Mason Cobb

Luke Koury

Dennis Sweeney

M. Parke 12:40

Date of Submission: 04/23/18

Executive Summary

In late January 2018, the team was tasked with developing a robot to serve as a pit crew assistant at the Formula EH Grand Prix. When a signal light activated, the robot would have to autonomously navigate a course to turn a crank 180° in a specified direction, move a wrench from a pedestal to a garage receptacle, push a lever on a car jack, and push the correct button as indicated by the color of a light on the course. The team was provided with a Proteus controller to direct the actions of the robot, and the course was equipped with a Robot Positioning System (RPS). Additionally, the robot was also required to fit inside a 9” x 9” box, be no taller than 12”, cost under $160 to construct, and be finished by its final assessment on April 7, 2018.

The team began solving this problem by brainstorming possible design solutions, and predicted that a robot with four omni wheels, a forklift mechanism to transport the wrench, and an independent 3D-printed crank-turning mechanism would perform the best on the course. Once an initial mockup of this design had been made, the team began construction. First, the team constructed the basic chassis and drivetrain and used this base to complete the first required performance test. For each of the three subsequent performance tests, the team implemented and tested the corresponding feature of the robot, so that by the last performance test, the robot completed the entire course.

This strategy resulted in a final design similar to the initial mockup - each of the robot’s tasks had a corresponding specialized appendage, and the robot used the RPS and omni wheel navigation. This was consistently successful in performing complete runs of the course. However, the robot failed once during the final evaluation due to a failure of the RPS, indicating that future similar design problems would be best solved with reduced reliance on RPS.

Table of Contents

List of Figures

List of Tables

1. Introduction

The Formula EH Grand Prix is a large event where teams from around the nation show off their Formula One cars. Several team managers and race executives reached out to the Ohio State Research and Development team (OSURED) to develop an autonomous robot that can perform the tasks associated with pit preparations as quickly and reliably as possible. OSURED constructed a model of the pit and asked companies to submit prototype solutions. When prompted by a starting light, the prototype would have to accomplish the pit crew tasks of flipping a car jack lever, determining the color of a light and pushing a corresponding diagnostic button, picking up a wrench and depositing it into a garage receptacle, turning a crank 180° in a specified direction, and pushing a large final button. Team E4 - consisting of Zach Boledovic, Mason Cobb, Luke Koury, and Dennis Sweeney - set out to design a prototype that met OSURED's specifications and accomplished all of these tasks as quickly and as reliably as possible. The team began initial designs on January 31, 2018 and was required to submit their prototype for final assessment on April 7, 2018.

This report illustrates the processes and conclusions from the team's design, testing, and improvement of the robot prototype. Section 2 highlights the original brainstorming process the team used and shows some of the results of this brainstorming. Section 3 discusses how the prototype was refined, including some tests of the robot's progress and investigations of how to improve. The section also shows the transition from the early stages of the robot towards its eventual final design. Section 4 highlights the first major competition point where the team's robot design was tested on its full abilities and compared to other robots. Section 5 highlights the final design of the robot's construction and behavior, and discusses how the robot met the design requirements. Section 6 covers discusses the robot's performance in its final assessment. Section

7 summarizes the results of the design project and discusses improvements for future robotics work.

2. Preliminary Concepts

This section defines the design problem and its constraints, then discusses the brainstorming process used to approach the problem is and the ideas it produced, including the initial rendering and mockup of the robot. Also included is a discussion of the initial thought approach to the robot's programming.

2.1. Requirements and Constraints

The following sections will cover the requirements and constraints that were imposed by OSURED, including both the tasks required to complete the course and the physical constraints placed on the robot.

2.1.1. The Course

OSURED designed a model course to mirror the pit where a final robot would work. The course was had two-level design with a grass ramp on one side and a concrete ramp on the other. The tasks that needed to be completed were scattered around the course. First, the robot was required to begin its tasks only once a starting light had been activated. Located on the first level was a panel of buttons. The robot was assigned to identify the red or blue color of a light on the floor of the course then press the button with the same color. The robot was also assigned to flip a lever on a car jack to release a car. Additionally, there was a wrench located on the bottom level that had to be transported into a garage located on the top level. Also on the top level was a fuel crank that needed to be turned 180^{o} in a direction based on a fuel type assigned for the car at each test. After each of these tasks were completed, the robot was to hit a final button located

where the robot started its run. Certain actions on the course could also detract from OSURED's assessment of the robot. If the robot pressed the wrong button on the button board, points would be deducted. If the crank was turned with an incorrect angle or direction, points were also deducted. The course featured a Robot Positioning System, referred to as RPS, which relayed helpful information to the robot such as position, heading, and what kind of fuel the car needed. The second level of the track featured an RPS dead zone where the robot's access to such information was deactivated, but which could be reactivated if the robot held the center white button on the button board for 5 seconds.

2.1.2. Other Constraints

Although the project was broadly open-ended, certain constraints were specified on the robot itself. The team was allowed $160 to build their entire robot. The robot had to be autonomous - no one could control the robot from a distance, and every move and decision the robot made would have to be preprogrammed. Additionally, the robot had to fit within a 9" x 9" x 12" box. If the robot exceeded the size requirement, it could not participate in the Final Competition to be considered by OSURED. In order for the robot to communicate with the RPS system, a team specific QR code had to be mounted on the robot.

2.2. Brainstorming

The team began to brainstorm solutions by individually generating different design ideas for each task that needed to be accomplished. This brainstorming continued at the first team meeting, where the members discussed their ideas. Before full robot designs could be drafted, the team discussed the criteria by which they wanted to judge potential robot designs. The three main concerns they had were survivability, consistency, and durability. The first and foremost concern was survivability. The team wanted to build a robot that could complete the course

through a broad range of unforeseen scenarios and recover from unpredictable failures; if the robot hit a wall and got off course, the team wanted a way for it to right itself and continue completing the tasks. This led to the second main concern: consistency. The team wanted a robot that could behave predictably and perform a perfect run on each and every course because it was known that a slow perfect run would be assessed higher than a fast but non-perfect run. Finally, the team knew they were going to do extensive testing on the robot so they wanted to build one that could stand up to their rigorous testing.

Some secondary concerns in the robot's design were speed, cost, and simplicity of construction. As mentioned above, speed was not a primary concern because completion outweighed speed, but completion time was an assessment factor. The team planned to make the robot faster only after it could consistently complete the course. Cost was considered secondary because performance was prioritized. Simplicity of construction was also not a primary concern because the team members were dedicated to put in the time necessary to make their robot successful, but simpler designs would make this easier.

2.3. Descriptions of Preliminary Concepts

Before the first team meeting, each team member had drafted different solutions to the tasks of the course. These ideas were discussed and compiled into three more detailed design concepts. The first robot concept, labeled R01, featured a drivetrain of four motors and omni-directional wheels and a dedicated mechanism for each task. Two appendages were mounted on one side so that two buttons could be pushed simultaneously. A rotational forklift piece with a servo would be constructed to transport the wrench. The fuel crank would be turned by a servo mounted to a piece that fit on the outside of the crank. These ideas can be seen below in Figure 1.

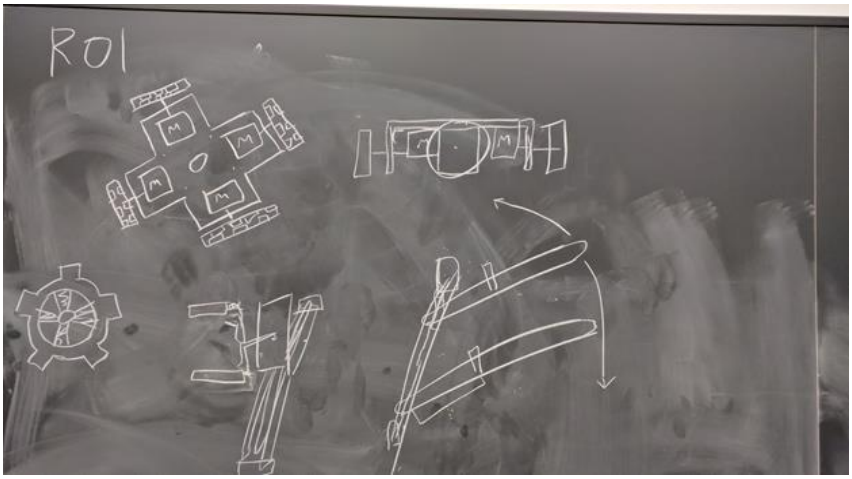


Figure 1: The First Design Idea

The second idea featured a simpler drivetrain with two wheels and a skid. The main mechanism on this design was an arm that actuated up and down and could grasp objects using a sliding gripper. The robot would use this appendage's tip to press buttons, its upward actuation to flip the jack, and its gripping ability to move the wrench. This idea also featured a rubber pad attached to a servo for turning the crank with friction. It was predicted that this friction pad would allow a larger margin of error than a piece that fit into the crank. A sketch of this design is shown below in Figure 2.

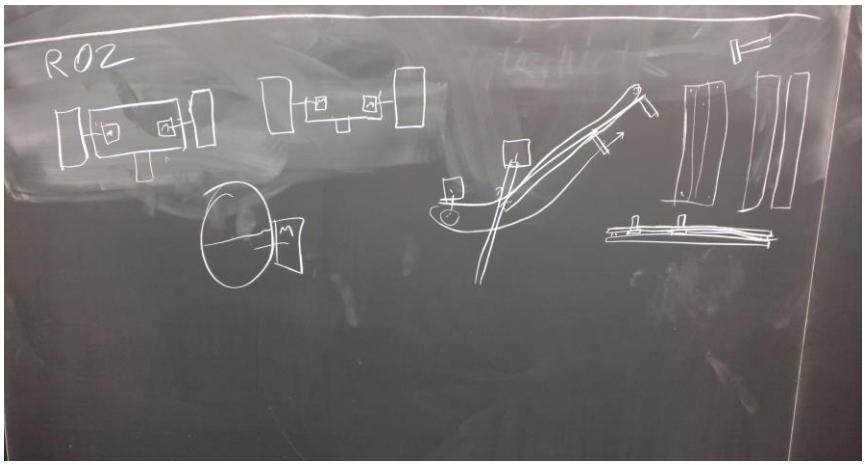


Figure 2: The Second Design Idea

The third idea was the team's most imaginative and risky. The design featured a four-motor drivetrain where each wheel was attached to a servo and could swivel 90° in place. This feature would allow for four directions of movement without turning. The button pushing method and wrench delivery system was a pair of J-shaped arms that would hook around the wrench and secure it into place. The method for turning the crank was similar to the other two ideas, except the piece that was mounted to the servo was a set of spring-loaded pins that would assume the shape of the crank. A sketch of this design is shown below in Figure 3.

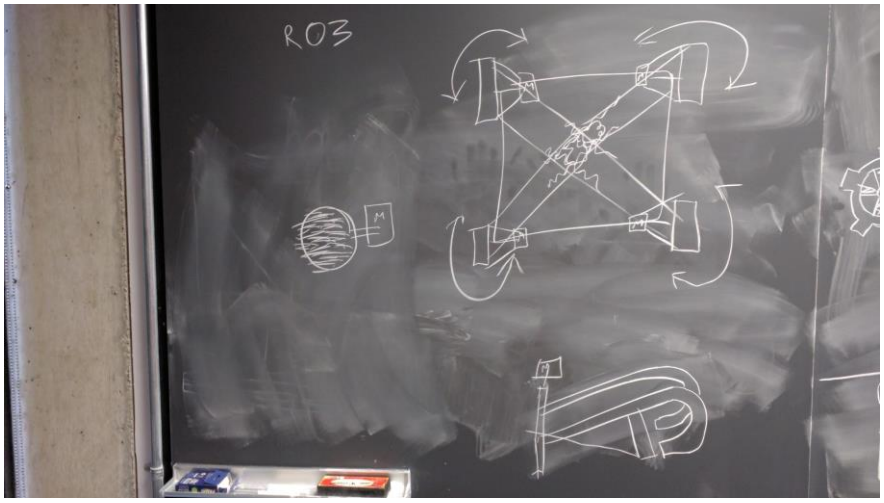


Figure 3: The 3rd Robot Design Idea

2.4. Mockup and Initial SolidWorks Model

After a consensus was reached, the team decided to move forward with the first design concept. The only feature that was changed from the initial brainstorming session was the shape of the chassis. Originally, it was thought that a cross-shaped design would save the team money because it would use less material, but after building the original prototype, it was found that an octagonal design gave more workable space, so much so that it outweighed the cost of the extra material. The original mockups and renderings can be seen in Figures 4 and 5 on the next page.

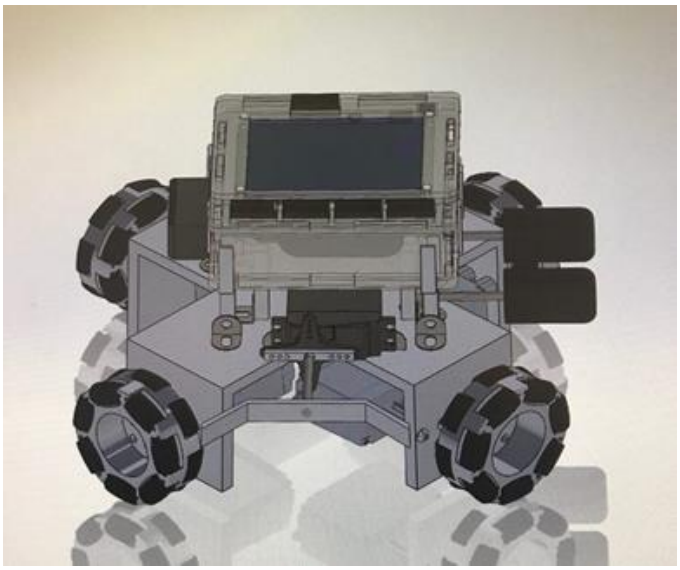


Figure 4: Initial SolidWorks of the Design

Figure 5: Initial Physical Mockup

2.5. Initial Programming Strategies

As one of our initial brainstorming steps, the team brainstormed the beginnings of how the robot's code would be organized and what algorithms it would use for navigation. To establish a common vocabulary, the team decided to declare the cardinal directions on each course with "North" as the direction toward the top of the course. It was also decided that the

"front" would be the side of the robot with the crank mechanism. Once these were defined, the team brainstormed the possible strategies of motion with omni wheels. Two possible schemes were found: either (1) driving the robot in only the eight cardinal and diagonal directions, or (2) driving the robot in a straight line toward any given point on the course. While the first option would have been much simpler to implement, it would not guarantee the same reliability to precisely take the robot to a destination on the course, and so the team chose the second scheme. The team then brainstormed a simple breakdown of the tasks assigned to each function, deciding that one function could set the robot in a given direction and another function could use this function to drive the robot to a certain pair of coordinates. From this, the team created a preliminary version of the functional dependence chart in Figure 15 in this report's section 5.5.3.

3. Analysis, Testing, and Refinements

The sections below highlight some of the tests and analyses conducted while refining the robot into the final design. The sections cover the calculations used to determine what motors to use, the testing done to determine what filter to put on the CdS cell, and the four performance tasks that were done throughout the building process that tested the robot's ability to perform the individual course tasks.

3.1. Drivetrain Calculations

In order to figure out what motors were suitable for the robot's drivetrain, the team needed to do several calculations to ensure the motors had enough power for a sustainable drivetrain. The main issue was making sure the motors had enough power to drive the robot up the course's ramps. The calculations done found the necessary torque and speed so that the robot could easily make it up the ramp. These calculations can be seen in Appendix D. The results of

the calculations yielded a point that could be plotted on a torque vs speed graph, as seen in Figure 6 below. If the point fell under a motor's line, then that motor could deliver sufficient power. In this graph, the requirement point is well under the line for every available motor, indicating that the team could use any motor they chose. The team decided to use VEX motors due to their compatibility with the omni wheel design.

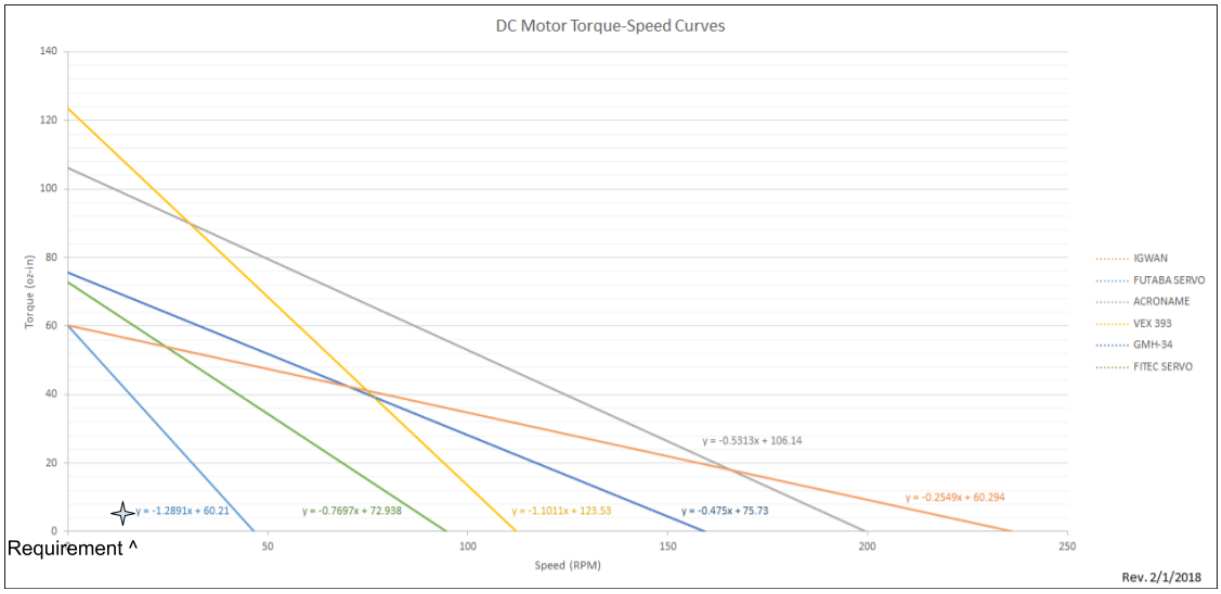


Figure 6: A graph of the available motors' torque-speed performance, with a point representing the calculated requirements for torque and speed [2].

After the motors were eventually installed, more tests needed to be done to determine the maximum speed of the robot. This value was important because it was used when the robot internally calculated the distance it had traveled using its speed and time. The team set the robot to drive in a straight line for 5 seconds using only two wheels at maximum speed, and measured the distance it traveled. After several tests, the team found a speed constant of 13.9 inches per second. It was also discovered that because the two perpendicular diagonal components of motion add vectorially, the maximum speed using all four wheels was 19.7 inches per second.

3.2. Performance Test 1

The first performance test was completed on February 21, 2018. The team was required to have their robot start with the starting light, navigate to the lever, flip the lever, and as a secondary task, drive up either of the ramps to reach the top section. Shipping logistics dictated that the team did not receive its four omni wheels until two days before the performance test, but the day the wheels arrived, the team was able to complete the test within two hours. The robot at this point did not utilize RPS, so all coordinates for movement were directly hard-coded. The team took several attempts of trial and error to find accurate values to use, but eventually found values that worked. On the team's second official attempt, the robot completed every goal in the performance test, including the stretch goal of touching the button board.

3.3. CdS Cell Filter Testing

In order to properly find the color of the light on the course floor, the team used two CdS photoresistor cells. These CdS cells decrease in resistance in light environments. By creating a voltage divider between 0V and 3.3V on the Proteus controller using a CdS cell and a pull-up resistor, the signal between the components would range between 0V and 3.3V, depending on the light delivered to the cell, with lower values corresponding to greater light. To accurately determine the color of the light, the team needed to find a way to differentiate the behaviors of the CdS cells between blue and red light. The team accomplished this by using plastic light filters. Placing a filter over the CdS cell allowed only certain wavelengths of light to pass through and affect the cell's reading. In order to determine what filter did this the best, the team measured the drop in voltage caused by the blue light and the red light for four filter colors. The results of this testing can be seen in Figure 7 on the next page.

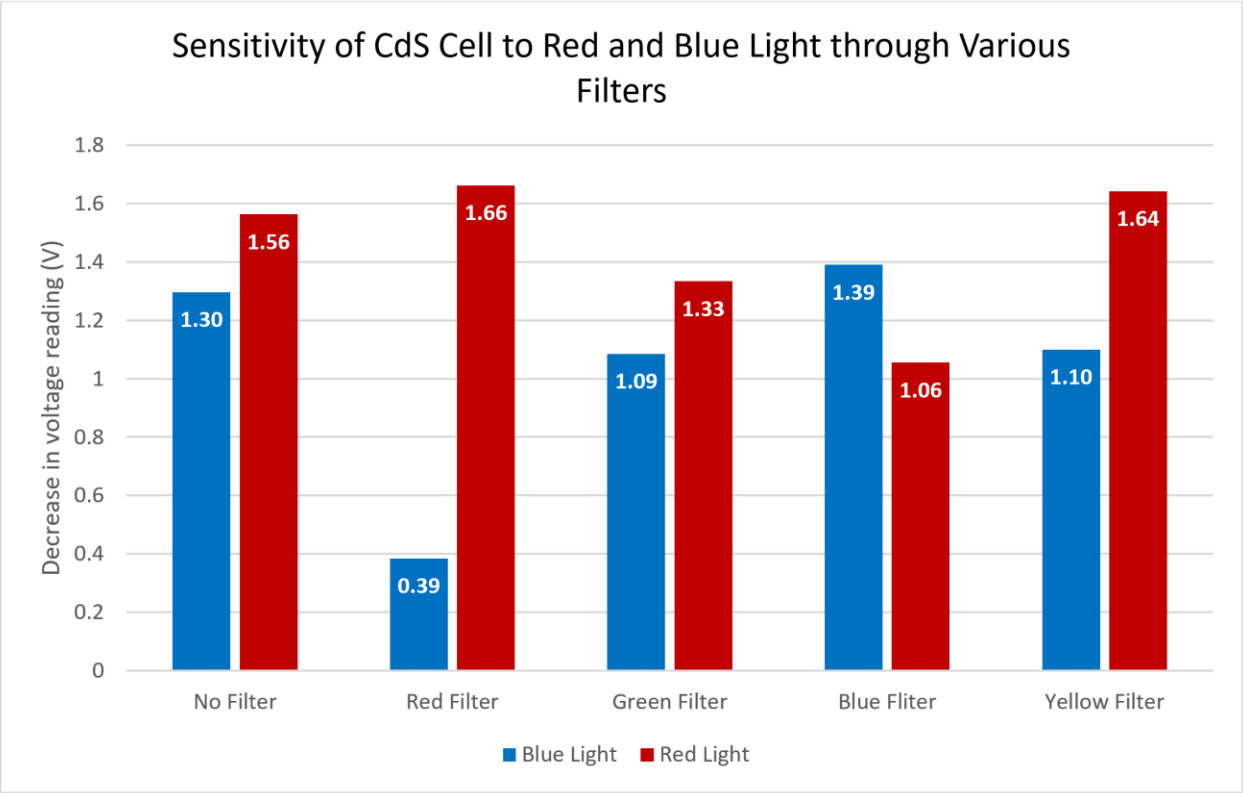


Figure 7: Graph of values given by the CdS cell with different filters

The graph shows that the largest difference in behavior between red and blue light values was when a red filter was used. The team selected this filter and used this on their CdS cell to accurately measure the difference in colors. Another CdS cell with no filter was also mounted on the robot so that the robot could compare the drop in the red cell's voltage to the drop in the bare cell's voltage. By judging these voltage drops proportionally, the robot could distinguish between a dim blue light and a red light or vice versa – it did not matter what the absolute decrease in voltage on the red cell was; it only mattered if it was 80% of the bare cell's voltage decrease. This 80% figure was found through trial and error.

3.4. Performance Test 2

The second performance test was completed on February 26, 2018. The robot was required to start with the starting light, navigate to the light on the floor in front of the button

board, determine the light's color, push the corresponding button depending on that color, and as a secondary task, touch the wrench. The team initially ran into difficulty in determining the light's color. However, once the team adopted the system of proportional judgement of two CdS cells as described in the previous section, the robot was able to reliably distinguish between the two colors. The robot also occasionally had issues with bumping into the button board on its way to touch the wrench, but this was not a major concern because the robot was still only using hard-coded coordinates that would be fixed later when the team implemented the use of RPS. The robot was able to complete all the tasks by its second official run.

3.5. Performance Test 3

The third performance test was completed on March 5, 2018. The robot was required to start with the starting light, navigate to the wrench, pick up the wrench, drive up to the top section of the ramp, deposit the wrench in the upper garage, and as a secondary objective, touch the crank. This performance test was the first time the team fully utilized RPS in navigation. However, the team realized that in order to have RPS enabled in the top section of the course, the robot would need to hold down the white button for 5 seconds, so the team added this step to the program procedure. After building the forklift mechanism, the main issue encountered when preparing for the test was correctly picking up the wrench. The coordinates used in movement had to be precise enough that the forklift did not miss the wrench's holes. After measurement, trial, and adjustment, the team eventually found precise values to use to make the robot complete the test. The robot met all the requirements on their second official performance test run.

3.6. Performance Test 4

The fourth and final performance test was completed on March 7, 2018. The robot was required to navigate to the top section of the course, detect the correct direction to turn the crank from RPS, turn the crank by 180° in the correct direction, and as a secondary task, return back down to the lower level. As with the previous performance test, the team decided to allow the robot to use RPS on the top level, so the robot's program ensured that the white button had been held for 5 seconds. The course's tolerance associated with determining that a robot had tried to turn the crank in the wrong direction was lower than initially expected, so a major issue encountered while preparing for this test was accidentally bumping the crank a small amount when the robot first approached. This would turn the crank the incorrect direction and cause failure. The team refined the robot's code so that the robot more carefully approached the crank without bumping before attempting to rotate it in the correct direction. After this, the team was able to perform the test and complete all the requirements on their first official run.

4. Individual Competition

The first subsection below discusses the Individual Competition, describing how it was carried out and what rules and specifications for assessment and scoring were given. The next subsection will discuss Team E4's approach to the competition and how the robot planned to navigate the course and complete all the required tasks. The following subsection covers the results of the competition and also offers some analysis of the results. The final section covers changes the team wanted to make in the final week leading up to the Final Competition.

4.1. Description of Competition

The Individual Competition served as a preliminary assessment by OSURED and took place on Friday, March 30th at 12:40pm on a randomly assigned test course. During this competition, the team was called into a separate room for three scored runs of the robot. On each of the three runs, the team had the possibility of earning a total of 100 points by accomplishing different tasks. The tasks are split up into 75 possible primary points and 25 secondary points. Table 1 on the following page shows the breakdown of the possible points.

Table 1: A Table of possible points in the Individual Competition [1]

Primary Tasks	**Points**
Initiate on start light (Notes 1 & 2)	9
Touch wrench (Notes 3 & 5)	8
Control wrench (Notes 3 & 5)	12
Toggle jack lever (Note 6)	10
Rotate fuel pump crank (Note 7)	8
Press an electrical test button (Note 8)	7
Press the correct electrical test button (Note 9)	12
Press the final charging button (Note 10)	9
Possible Primary Task Points	**75**
Secondary Tasks	
Deposit wrench (Note 11)	8
Rotate fuel pump crank in correct direction (Note 12)	10
Rotate fuel pump crank 180° (Note 13)	7
Possible Secondary Task Points	**25**
Total Possible Task Points	**100**
Penalties	
Interfering with a competitor's robot (Note 14)	DQ

Seeding for the Final Competition and assessment was determined by the performance in the Individual Competition. Seeding was determined by taking the average of the total points scored in all three runs. If multiple teams had the same average score, the tiebreaker was the best time of the highest scoring run.

4.2. Strategy

Team E4's strategy towards the competition was to focus on the robot's consistency more than the robot's potential speed. The robot was fully capable of achieving perfect 100-point runs, and the team sought to achieve perfect scores on 3 runs, regardless of speed. An outline for the general strategy of the robot in each run can be seen in the cyan path in Figure 8 below.

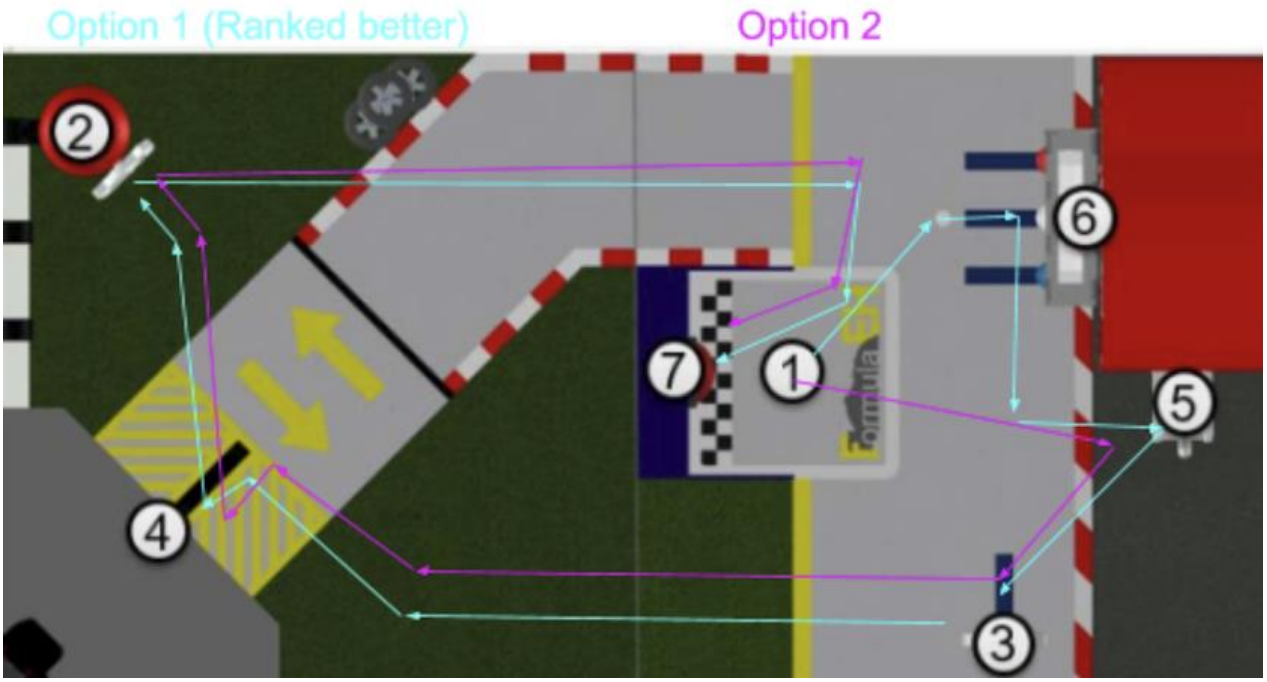


Figure 8: Path of course navigation and order of tasks completed

First, the robot would wait for the starting light at position 1 in the image above to be turned on. Once the light turned on, the robot would then immediately move towards the light in front of the button board, located at position 6 in Figure 8 above. This light was read using the CdS cells to determine whether the blue or red button was to be pressed. After this determination, this button and the white button would be pushed and held until the RPS zone at the top was enabled. After this was completed, the robot then moved to the corner of the course with the lever and then drove toward the lever located at position 5, pushing it using a cardboard square on its body. After the lever was pushed, the robot then would travel in front of the wrench

at position 3 and would carefully lower the forklift arms and pick up the wrench. The robot would then drive up the grass ramp to reach the arrows in front of the garage located at position 4. After aligning with the garage, the wrench was then deposited in the garage and the robot moved towards the crank at position 2. The crank was then turned in the correct direction that was indicated through RPS. Finally the robot returned to the starting area and pushed the final button at position 7.

4.3. Robot Performance and Analysis

The first run was performed on test course H, with the blue button and clockwise crank rotation assigned. The robot ran through the entire course with no issues and was able to achieve a perfect run in 48.73 seconds. The second run was performed on course G, with the red button and counterclockwise crank rotation randomly assigned. The robot accomplished all the tasks again with no issues and scored a perfect 100 with a time of 43.72 seconds. In the final run, the selected course was G, the button color was red and the crank direction was counterclockwise. In this run, the team was able to select course G, a red button, and counterclockwise crank-turning. The robot once again achieved a score of 100 with a time of 49.40 seconds. The average score of all the three runs was 100 points with an average time of 47.28 seconds. The performance resulted in the team getting a number one seed going into the Final Competition.

4.4. Suggested Modification

Since the robot performed very well throughout the duration of the Individual Competition, only subtle modifications were needed. With a week still left before the Final Competition however, the team decided to work throughout that week to reduce any extra position checks or anything else that caused extra time in order to shave down some time before

the Final Competition. The team wanted to focus on maintaining their consistency while lowering the course completion time as much as they could.

5. Final Design

This section details all of the final major components and mechanisms used the robot. Explanations as to construction and operation for all components are given below, as well as pictures showing the completed design. SolidWorks drawings of the robot can be found in Appendix A. A picture of the completed robot can be seen below in Figure 9. The section also details the code and algorithms used in the completion of the course. Finally, this section also details the team's overall budget and how it was spent.

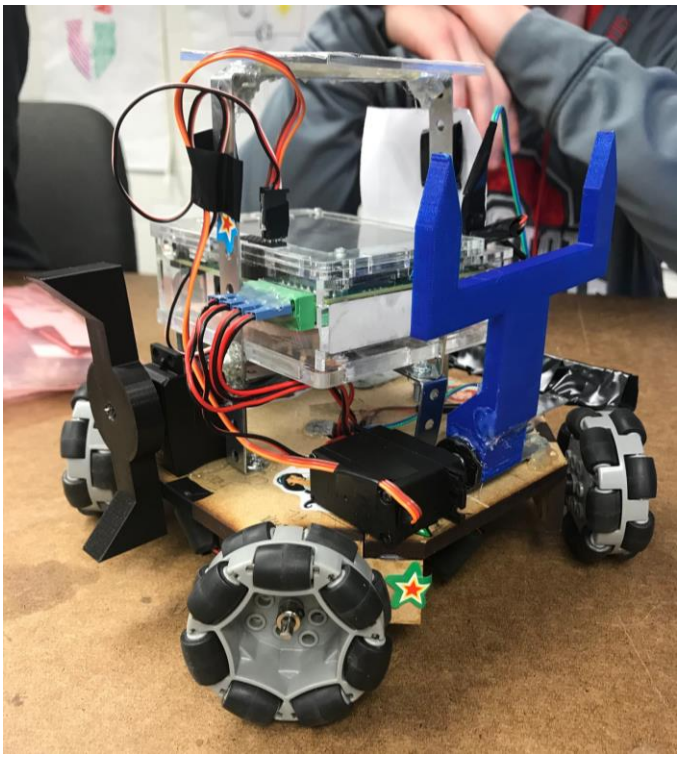


Figure 9: Picture of Final Robot

5.1. Chassis and Drivetrain

The final chassis for the robot was constructed out of medium density fiberboard (MDF). This material was ideal because it was inexpensive, yet durable. While budget was not a primary concern, durability certainly was, given the amount of testing that would be needed to test and refine the robot. This made MDF the ideal material for the chassis. This material was laser-cut into the shapes necessary for construction as to have the most possible precision and consistency between parts. It was especially important that these parts be accurate, as any irregularities in the chassis could affect mounting of other components and hinder accuracy in movement and other tasks. The holes for axles and mounting points were later drilled into the laser cut pieces, as these holes did not require consistency between parts in order to fit together and function accurately.

The base of the chassis was cut into an octagonal shape with a hole cut in the middle. The hole would allow for wires to be run to the motors, as well as provide a mounting place for the CdS cells. The shape itself allowed there to be four faces on the perimeter angled 45° from the front, back, left, and right faces. This allowed the omni wheels to be placed at 45° for maximum power in the front, back, left, and right directions. Notches were cut in these angled portions to allow the base and the wheels supports to fasten together securely. The wheel supports were cut into rectangles with a tab to fit in the notches on the base, allowing for secure alignment of the drivetrain.

Vex motors were then mounted to the wheel supports using the provided screws. This assembly was then bonded to the base of the chassis with hot glue. The aforementioned notch in the wheel supports ensured the 45°-orientation while mounting. Using a 2” Vex axle, a plastic spacer, and two shaft lock collars for each, all four omnidirectional wheels were connected to the Vex motors through the wheel supports. Wires for the motors were fed through the hole in the

chassis base so they could be connected to the Proteus via plastic terminal blocks. This assembly provided the central structure and movement component for the robot. It was also made to be perhaps the most durable component of the robot because these parts would see the most wear during testing. Figure 10 below shows pictures of the completed chassis and drivetrain.

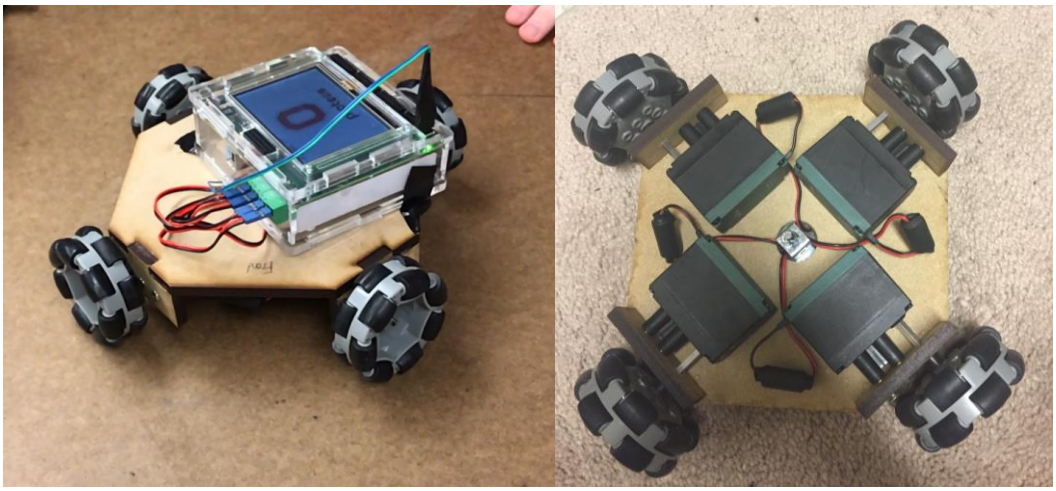


Figure 10: Top (left) and Bottom (right) views of chassis and drivetrain

5.2. Button and Lever Pushing

Originally, the chassis base of the robot was to be used to accomplish the lever-pushing task. While this would have worked, the chassis base was slightly too low to be completely practical. To solve this issue, a square of cardboard from the box of a VEX motor was used instead. This component was mounted on the side of the robot that was to push the lever. It was affixed to the chassis and the Proteus mounting hardware so that a slight upward angle was achieved. This gave an ideal surface for pushing the lever and removed stress from the chassis by now having a dedicated component for this task.

To push the buttons, a series of erector set pieces were used. Original designs included two independent button pushers, but, this provided a small margin of error in aiming for the buttons. Instead, an erector set angle girder connected the two appendages, creating a bar

between the two pushing points that extended slightly past each individual pusher. This provided a much greater surface area to push the buttons with and thus a greater margin of error. To further refine this system, electrical tape was added to the button bar, and later hot glue. These were both included to prevent slipping between the button pusher and the button board to provide better consistency. Both the lever and button pushing mechanisms required the drivetrain to supply all of the force, with the hardware being simply a medium to apply the force in the way that was needed. A picture of the button pushing mechanism can be seen below in Figure 11.

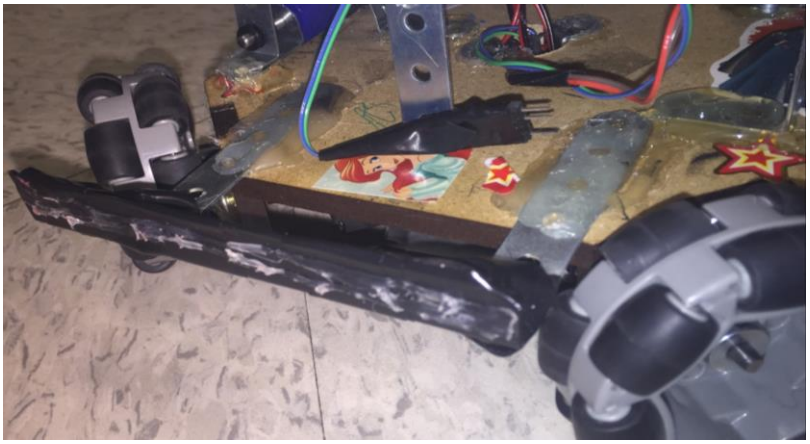


Figure 11: Picture of button-pusher

5.3. Forklift Mechanism

The forklift mechanism needed to be specially made. This part was designed in SolidWorks to have the exact dimensions to fit into the wrench. The chassis base was above the wrench, so the arm had to come down slightly from the chassis in order to pick up the wrench. Given the unique constraints needed for this task, the arm itself was 3D printed. This part was printed out of PLA plastic, which was light but still somewhat sturdy. While this part was broken in testing, hot glue was used to fix and strengthen the broken area. This part was attached to a Fitec high-torque servo via the provided hardware, which was mounted onto the base of the

chassis with hot glue. Angle brackets, a one-inch erector set axle, and hot glue were used on the side of the part opposite the servo motor to create a solid axis of rotation for the arm.

This arm was able to actuate up and down via the servo. While servos provide for 180° of rotation, only 90° of the range were needed for this mechanism. This would allow the arm to be parallel to the floor for picking up the wrench, and point upward for when the wrench was picked up and being carried. As designed, once on the forklift arm, the wrench would sit below the lip of the garage when the arm was down. However, the arm itself would not below the lip. This situation was ideal for depositing the wrench with this mechanism. The garage was approached with the arm up, then the arm was lowered inside the garage. This allowed the wrench but not the forklift to get caught on the lip, leaving the wrench inside the garage when the robot backed out. A picture of this entire mechanism can be seen below in Figure 12.

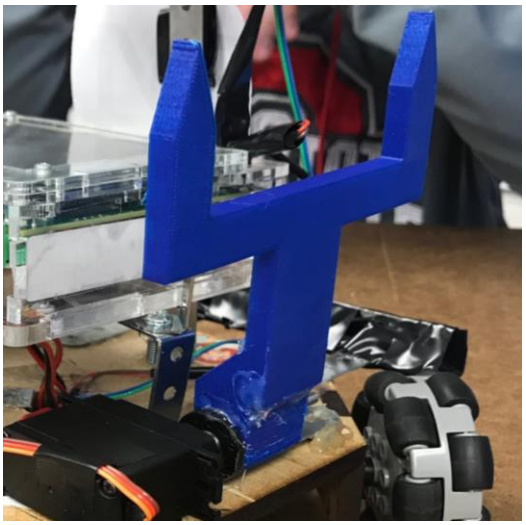


Figure 12: Picture of forklift mechanism

5.4. Crank-turning Mechanism

The Crank-turning component was also a part that needed to be specially made. The part used to grip the fuel crank needed to be specifically made to fit around the crank to be turned. Using SolidWorks, a custom part was designed to grip the outside of the crank and still easily

connect to the provided hardware. This part was also 3D printed out of PLA plastic. Using the hardware provided with the Futaba servo and hot glue, the crank-turning piece could be mounted directly to the servo. The axis about which the crank-turning piece rotated needed to be at the same height as the axis about which the crank itself rotated. This would ensure smooth and accurate rotation for the completion of this task. The servo was able to be mounted directly to the chassis to accomplish this, as the axle in the servo was at the same height as the one in the crank. The servo could be mounted on the side away from the wheel so that the crank would not hit the wheel.

When mounting the crank-turning piece, careful attention was paid so that the custom piece was oriented vertically when the servo was at 90° out of its 180° range of motion. This is due to the fact that the crank needed to be turned 180° in potentially both directions. Mounting the custom part in this way allowed for the range of motion required. Being vertical at 90° allowed for the piece to rotate 90° right in order to move 180° left and vice versa. This allowed the crank to be rotated 180° in either direction. To use this mechanism, the crank-turning piece was rotated 90° opposite the desired direction, then the mechanism was positioned around the crank. The servo could then be rotated 180° in the desired direction to turn the crank. A picture of the crank-turning mechanism can be seen on the next page in Figure 13.

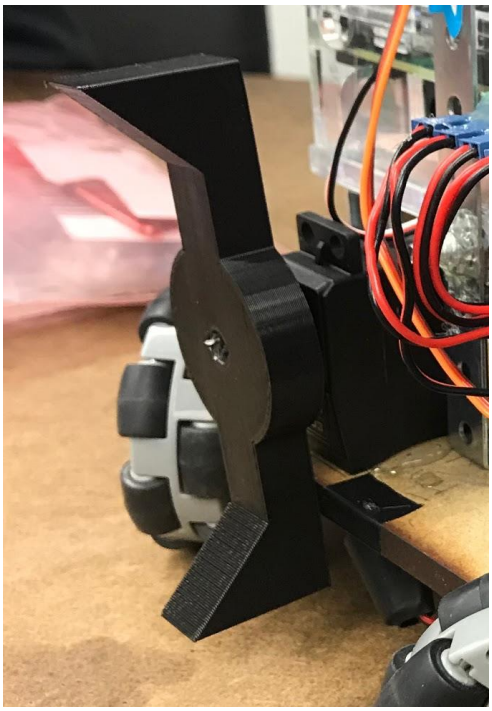


Figure 13: Picture of crank-turning mechanism

5.5. Code

The following sections cover the code for the robot that allowed it to accomplish each of the course's tasks. The first section covers the general strategies that were used when designing the code. The next section covers the code used in the starting up and calibration aspect of the code. The final section covers the navigation algorithm the robot. The code for the entire robot program is attached in Appendix B, along with a pseudocode representation.

5.5.1. Coding Strategies

The team's first goal in implementing the robot's program was to modularize the program. To this end, the program's 1011 lines of code were divided into 37 unique functions. This allowed different tasks in the program to be tested and debugged independently from one another. The program included functions for each of the robot's actions, such as raising and lowering the forklift or rotating the crank-turning piece to a particular orientation.

The program also included a simple logging function that would write a given string and up to four numbers to the Proteus LCD screen and at the same time to a log file on its micro SD card with a time stamp, so that it could be apparent what functions the robot was calling and what data it had stored. The calling of this function also served as a comment within the code, so any log file could be directly compared to the current code and the program's failures could be easily isolated and debugged. As a helper function proved itself to be reliable, its logging function was commented out to reduce clutter in the log files, so that only the highest-level comments were logged. A sample log file is attached in Appendix C.

5.5.2. Robot Startup and RPS Calibration

At the startup of every run, the robot called a function that encapsulated all of the necessary startup actions for that run. This function set all of the servo motors' upper and lower limits and initialized their angles, initialized and calibrated the RPS, calibrated the CdS cell benchmark values, and waited for a final starting touchscreen tap and for the CdS cell to detect the starting light. Most of these tasks were encapsulated into their own functions.

To calibrate the RPS, the team designed the graphical user interface for the Proteus depicted on the next page in Figure 14. The team would move the robot to three waypoints on the course - the crank, the wrench, and the starting light - and hit the corresponding buttons on the screen. This would store the RPS position and heading at those locations and would allow the robot to return precisely to these locations during the run. The calibrated coordinates were displayed on the screen with a font color that was dependent on their proximity to hard-coded default values for each of the three waypoints: grey indicated that that waypoint had not yet been calibrated, red indicated that the calibration was three or more inches away from the expected value, green indicated that the calibration was within one inch of the expected defaults, and

yellow indicated that the calibration was between one and three inches away from the expected defaults. Thus, the team could actively estimate the legitimacy of the waypoint calibrations. If a waypoint was incorrectly calibrated, then the team could press another button to revert that value to its defaults. Also included were a button to test all six motors and toggle the servos between their calibration positions (extended forklift and horizontal crank-turning piece) and their start positions (upright forklift and vertical crank-turning piece), and a final starting button.

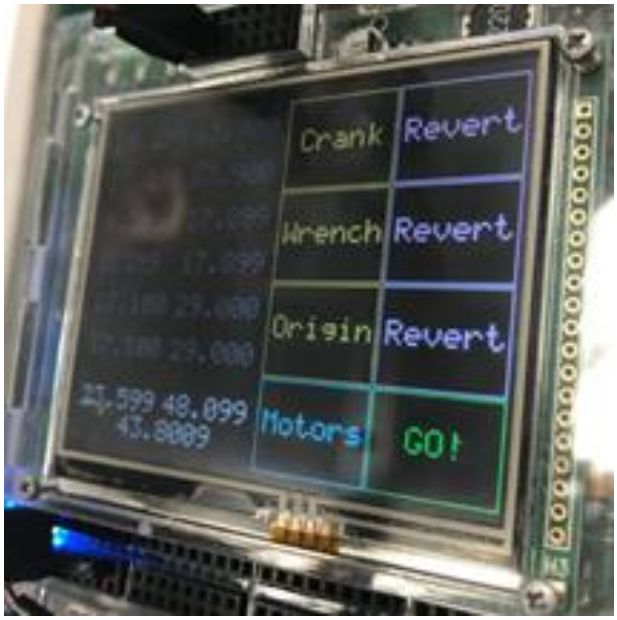


Figure 14: The RPS calibration GUI

5.5.3. Navigation Algorithm

Beyond modularization, the major strategy for the robot's motion and navigation code was to build up a hierarchy of abstraction in its functionality. For example, each individual wheel's motion did not need to be considered for every move. Therefore, beginning with the most primitive function of setting the speeds and directions of the four wheels of the drivetrain, the robot used layers of helper functions to successively add functionality. These functions are structured according to Figure 15 on the next page.

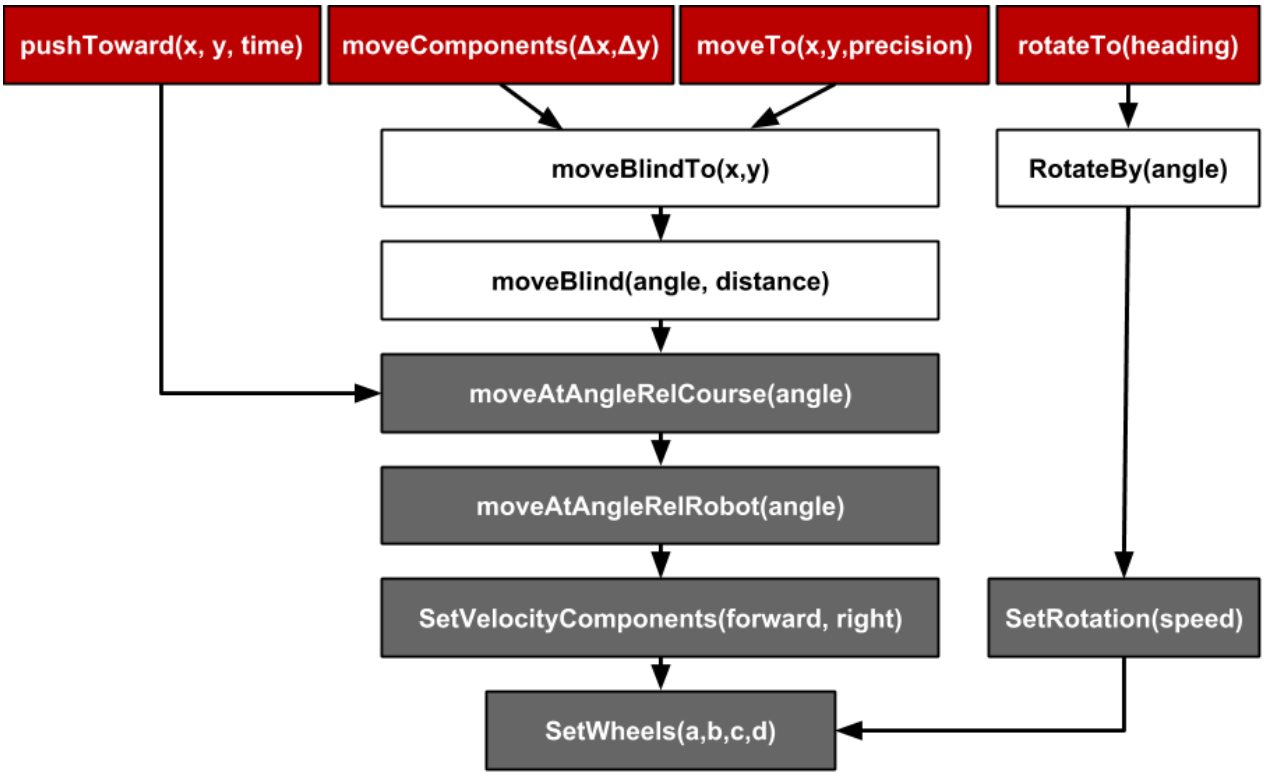


Figure 15: A diagram of the dependencies of motion functions. The arguments listed are approximate.

In Figure 15 above, arrows represent a function calling another function. The four scarlet functions at the top of the figure are the highest-level functions that comprise the majority of the main top-level procedure. The grey functions toward the bottom serve to instantaneously set the robot in a particular motion, as compared to the scarlet and white functions, which deliberately take time to run.

At the most primitive end of the hierarchy, the *setWheels* function set the four wheels of the drivetrain rotating at specified rotational velocities. The *setVelocityComponents* function was then able to call this function and set the rotation of the wheels so that the robot would move at a vector with the specified components. This was accomplished by projecting the given vector onto the axes of the wheels' functional directions, then proportionally scaling all four wheel speeds so that there was no overflow.

Using simple unit-circle trigonometry, the *moveAtAngleRelRobot* function called the *setVelocityComponents* function to set the robot moving in a particular direction relative to its own front/back and left/right axes. By subtracting the current heading of the robot, the *moveAtAngleRelCourse* function was then able to call the *moveAtAngleRelRobot* function and set the robot moving in a particular direction relative to the course's "North/South" and "East/West" directions. This function was then called by the *moveBlind* function, which set the robot moving in the specified direction, calculated the speed at which it would move, and then moved the specified distance by halting the robot's motion only after the correct time had elapsed, as determined by dividing the distance by the speed. The *moveBlind* function also incremented the robot's stored position values based on the trigonometry of the angle and distance. The *moveBlindTo* function could then take in a pair of coordinates, calculate the angle and distance between them, and then call the *moveBlind* function to move by that distance at that angle.

Two top-level functions depended on the *moveBlindTo* function: *moveComponents* and *moveTo.* The *moveComponents* function simply called *moveBlindTo* for a point that differed from the current robot position by a specified vector. The *moveTo* function took in a pair of destination coordinates and a maximum allowable error distance, then called *moveBlindTo* function for those coordinates. It then halted and waited to update its stored position with the RPS. Once updated, if the error was within the limit, the function ended, but if the error was outside of the limit, *moveTo* would be called again recursively. A global variable was also used as a failsafe for specific motions to set a maximum on the time that the *moveTo* function would iterate.

To facilitate pushing both the buttons and the lever on the course, the program included a *pushToward* function that would call *moveAtAnglRelCourse* to push the robot toward a

destination and continue pushing for the specified time, using RPS to update the correct direction every 100 milliseconds and not incrementing the stored position.

Compared to the linear motion scheme described above, the robot's rotational motion algorithm was similar but simpler. The *setRotation* function called the *setWheels* function to set the robot rotating clockwise or counterclockwise at a specified speed. Then, the *rotateBy* function could rotate the robot by a specified angle by calling *setRotation*, then halting the wheels after the correct time, as determined by dividing the angle difference by the rotational speed. The *rotateBy* function also incremented the robot's stored heading accordingly. Finally, the *rotateTo* function found the distance between the robot's current heading and a destination heading and then - in the same fashion as the *moveTo* function discussed before - called the *rotateBy* function, paused until the RPS gave current values, then called itself recursively only if the heading was outside of a specified margin of error.

To ensure that the robot always had access to a valid and accurate estimate for its position, the *updatePosition* function, which was called whenever the RPS was accessed, checked for the RPS values to be valid before storing them in a position global variable. This is the position variable that was accessed or incremented by all of the movement functions. The calls to this *updatePosition* function all occurred while the robot was stationary and had paused for 800 milliseconds so that the robot would have had time to halt its movement, stabilize its position, and wait for the RPS to return accurate values. Two additional functions - *updateOnlyHeading* and *updateOnlyXY* - allowed the robot to also check its heading during all movements and check its coordinates during all rotations. These additional checks allowed for alternating sequences of motion and rotation in quick succession without pauses in between.

5.6. Budget

Throughout the project, team E4 had a budget of $160 to spend on the entirety of the robot. Overall, the team spent $141.62, with $18.38 left over. A total of $88.43 was spent on the drive train, $13.04 was spent on the crank, $15.84 was spent on the forklift, $7.09 was spent on the chassis, $0.90 was spent on Fasteners and Electric components, $1.34 was spent on CdS cells, $2.78 was spent on the button pusher, and $13.34 was spent on the mount for the Proteus and QR code. A full breakdown of all the parts ordered can be seen below in Table 2. A graphic showing percentages of the total budget can be seen in Figure 16 on the next page.

Table 2: A list of how the budget was spent

Part	Quantity	Price
Omni Wheels	x4	$20.00
Vex Motors	x4	$60.00
Futaba Servo	x1	$10.00
Fitec Servo	x1	$12.00
Laser-cut MDF	x1	$7.09
3D-printed Forklift	x1	$3.40
Vex shaft lock collars	x4	$2.00
Erector shaft lock collars	x5	$1.75
Terminal Blocks	x4	$0.20
Vex Axels	x8	$3.60
Plastic Spacer	x4	$0.88
Angle brackets	x8	$4.64
Screws and Nuts	10 pack, x2	$0.40
Double bent strip	x1	$0.79
CdS Cell	x1	$0.45
Angle girder	x1	$1.58
3-inch Erector strip	x2	$1.20
Red plastic light filter	x1	$0.10
Mounting for Proteus and QR Code	7 pieces	$8.70
3D-printed crank turner	x1	$3.04
Long Header Pins	x1	$0.50
1-inch Axle Rod	x1	$0.44
Store Retruns		-$2.39

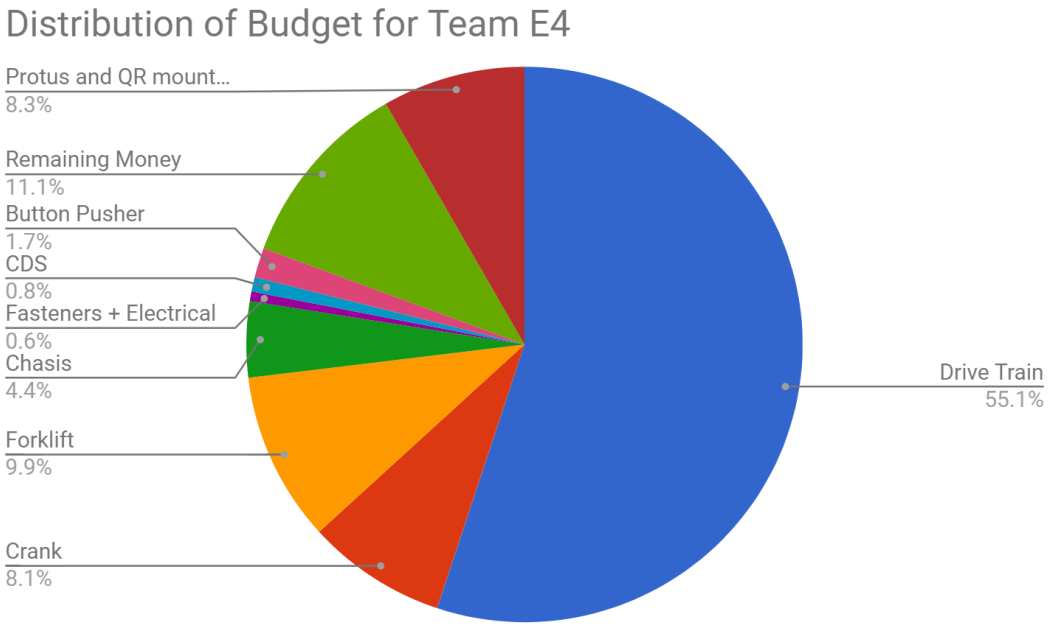


Figure 16: A pie chart of how the budget was spent

6. Final Competition

The first section below gives a description of the Final Competition, including its location, the organization of events, and how the scoring was determined. This competition served as a final assessment by OSURED. The next section covers the strategy taken by team E4 throughout the competition. The final section covers the results of each run and the overall results of the Final Competition. That section also covers the possible modifications the team would make if they continued working on the robot.

6.1. Description of the Competition

The Final Competition took place on Saturday, April 7th at 12:00 pm at the Tom W. Davis Multipurpose Gym in The Ohio State University's RPAC. The Final Competition featured two main events: a Round Robin and a Head-to-Head competition. The Round Robin was executed very similarly to the Individual Competition - each robot ran the course 3 times and

achieved scores from 0 to 100 based on how many tasks were completed. The scores were then averaged and the quickest time of the highest scoring runs was used to break potential ties.

After the completion of the Round Robin, 63 robots were entered into a single-elimination tournament based on the results of the Final Competition. In this Head-to-Head event, four robots would compete at the same time and the robot that achieved the highest score moved on. If two robots achieved the same score, then the faster of the two times would be used to determine who the winner was. The scoring of points for the Round Robin and Head-to-Head was identical to the scoring of the Individual Competition.

6.2. Team Strategy

Team E4's strategy was very similar to their strategy during the Individual Competition in that the focus was on consistency. The robot had the ability to achieve scores of 100 in every run, so the team was less concerned with speed. The robot was one of the most consistent contenders, so the team wanted to fully utilize that advantage.

The team's strategy for course navigation was the same general path as the Individual Competition, but with slight changes in the robot's code save time. The team eliminated several redundant or unnecessarily precise checks throughout the run, which ended up reducing run times by 5 to 6 seconds on average from the Individual Competition.

6.3. Results and Analysis of the Competition

In the first run of the Round Robin Competition, the team's robot ran on course E with a blue light and a clockwise crank direction. The robot ran through the course without encountering any issues and completed a 100 run in 42.33 seconds. The second run was done on course B with a red light and a counterclockwise crank direction. The robot once again ran through the course without encountering any issues and completed the run in 41.60 seconds. In

the final Round Robin run, the robot ran on course C with a red light and a clockwise crank direction. The robot again did not encounter any issues and finished with a perfect run in 40.17 seconds. The team finished the Round Robin with an average score of 100 points and an average time of 41.36 seconds netting them 3rd place in the Round Robin competition.

The team then went on to compete in the Head-to-Head competition. In the first run of the Head-to-Head competition, the robot ran on course C with a blue light and clockwise crank direction. During the run the robot lost connection to RPS in front of the garage before depositing the wrench. The disconnection occurred during a rotational movement which caused the robot to quickly rotate back and forward. This occurred for approximately 15 seconds until the robot was able to reconnect to RPS, after which it was able to successfully navigate the entire course. The robot finished with a 100 score and a time of 57.68 seconds. In the next round of Head-to-Head, the robot ran on course G with a blue light and a counterclockwise crank direction. During this run, the robot lost connection to RPS before picking up the wrench. In this run, the robot lost connection during a movement causing the robot to drive wildly to a position that was not even on the course. The robot crashed into the lever knocking it off and eliminating the possibility of a perfect run. The team was forced to kill the run after the robot's RPS values and calculations were inconsistent with its physical position. The robot finished with a score of 58 and a time 72.07 seconds. The robot was eliminated from the competition based on this run.

The team received 3rd place for the most consistent award due to the robot's performance in the Round Robin. After examining the robot's final run, the team determined that if work were to continue work on the robot, additional mechanisms should be included for navigation independent of RPS. Since the team was heavily reliant on RPS when the faulty values were read, the robot did not have any possibility of still completing the course successfully.

7. Summary and Conclusions

The first section below gives a summary of the entire report thus far and covers the major points of interest found throughout the report. The next section draws conclusions on the final performance of the robot and also determines some possible changes that can be implemented to show what the team would do in future work.

7.1. Summary

Team E4 was tasked to build an autonomous robot that could accomplish several tasks necessary for cleaning up and resetting a garage model after a pit at the FEH grand prix. The team first started by brainstorming several possible solutions to the problem and then used decision matrices to select the best approach. The team selected a four omni-wheel design that had specialized mechanisms for accomplishing each task. The team then solidified their initial design with a mockup. After the mockup was completed and evaluated, the team began building the robot. Several Tests were performed during the construction of the robot including tests to see which motors were necessary as well as tests to see which colored filter should be put on the CdS cell. The team also performed four performance tests designed to see how well the robot could accomplish individual tasks.

The final robot had several key features that allowed for its success. The final design featured a four omni-wheel design on a laser cut MDF chassis. The design also featured forklift arm used to pick up and deposit the wrench, a crank turner used to turn the crank, and a button pusher used to push the buttons. The final design also featured advanced navigation code that allowed it to move to a specific point with relative ease.

The robot was able to get 3 perfect runs with an average time of 47.28 seconds in the Individual Competition. This performance gave the team the number 1 seed for the Final

Competition. In the Final Competition, the robot performed well in the first half of the competition, the Round Robin. The robot achieved 3 perfect runs with an average time of 41.36 seconds. The robot however ran into several issues with RPS throughout the Head-to-Head competition and the robot was eliminated in the second round. The team received award for its 3rd place finish in the Round Robin.

7.2. Conclusions

In general, the robot was able to reach the full potential of how it was designed. The main focus of the robot's design was consistency and this goal was fully met. The robot had a streak of 2 weeks without failing a run, with failure meaning a non-perfect run. The robot actually never failed a run with its final version of code when RPS was fully functional. Every mechanism on the design worked reliably as it was intended, and the team was able to stay true to the original design principles and strategies set fourth at the start of the design process.

Even though the robot achieved success, there are still several improvements the team would make for future work on the project. The team decided that the inclusion of sensors like bump switches could be used to allow stronger navigation without RPS. The main issue with the final rounds of the competition is that when RPS connection was lost the robot had no way to possibly still navigate the course due to the heavy reliance on RPS. A second more drastic approach to future work would be a complete redesign of the robot code. The code was programmed with a consistency-first and speed-second approach, and it was for this reason that the robot was very consistent but they could never achieve any times lower than 39 seconds. If a speed-first approach was taken, the team could have potentially gotten much faster times and from there they could have gotten more consistent.

References

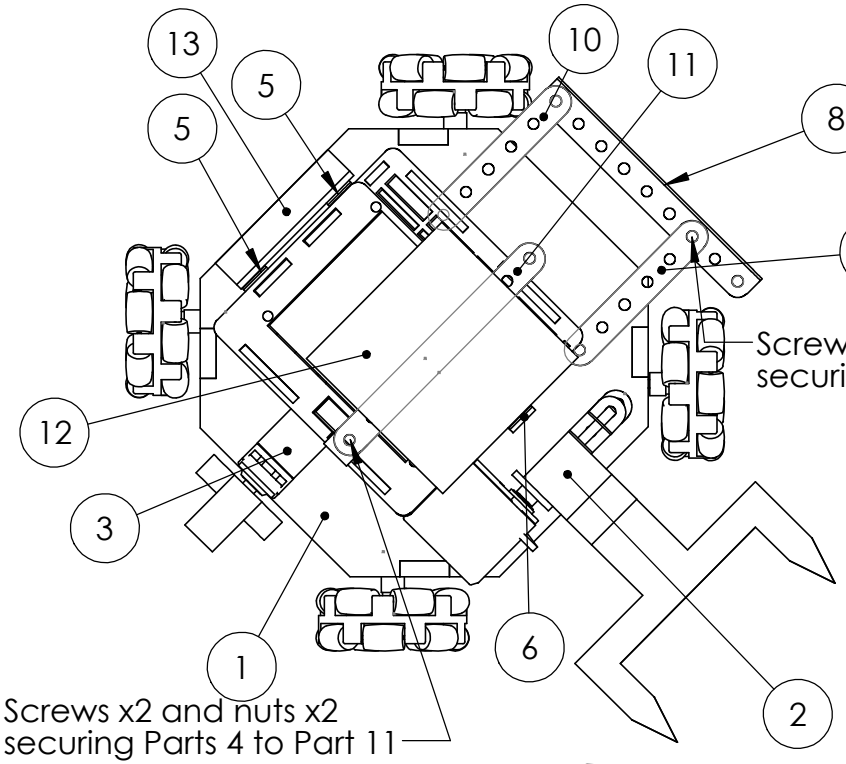
[1] Robot Scenario: Design Problem Statement and Specifications. 2018, February 2. https://www.carmen.osu.edu.

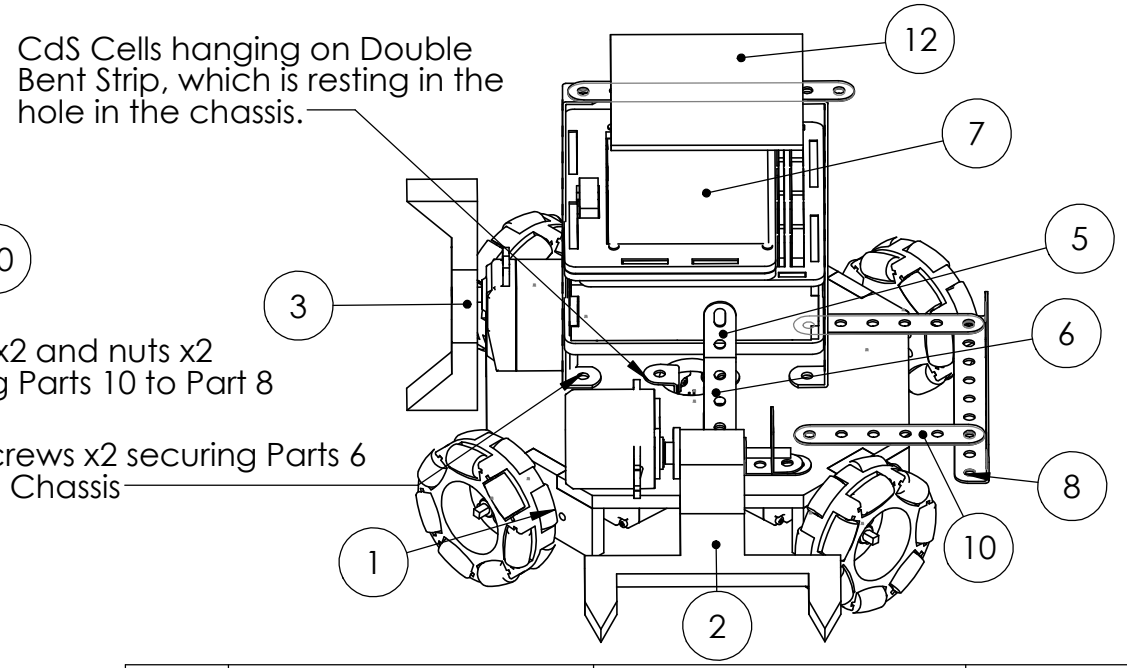
[2] FEH Motors Graph. 2018, February 2. https://www.carmen.osu.edu

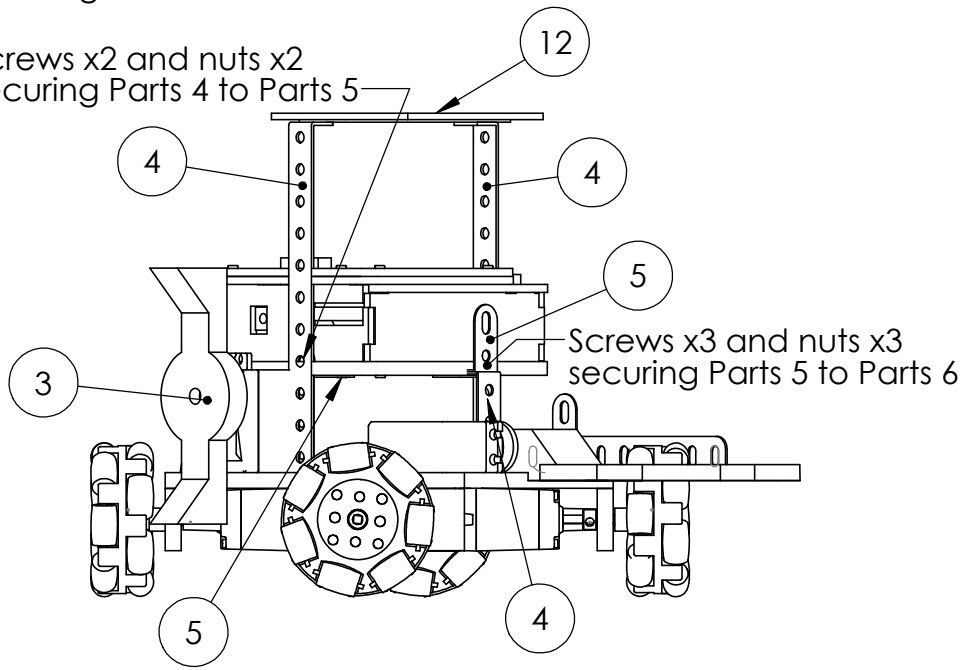
[3] FEH Proteus Guide. 2018, March 20. https://u.osu.edu/fehproteus

APPENDIX A

SolidWorks Drawings

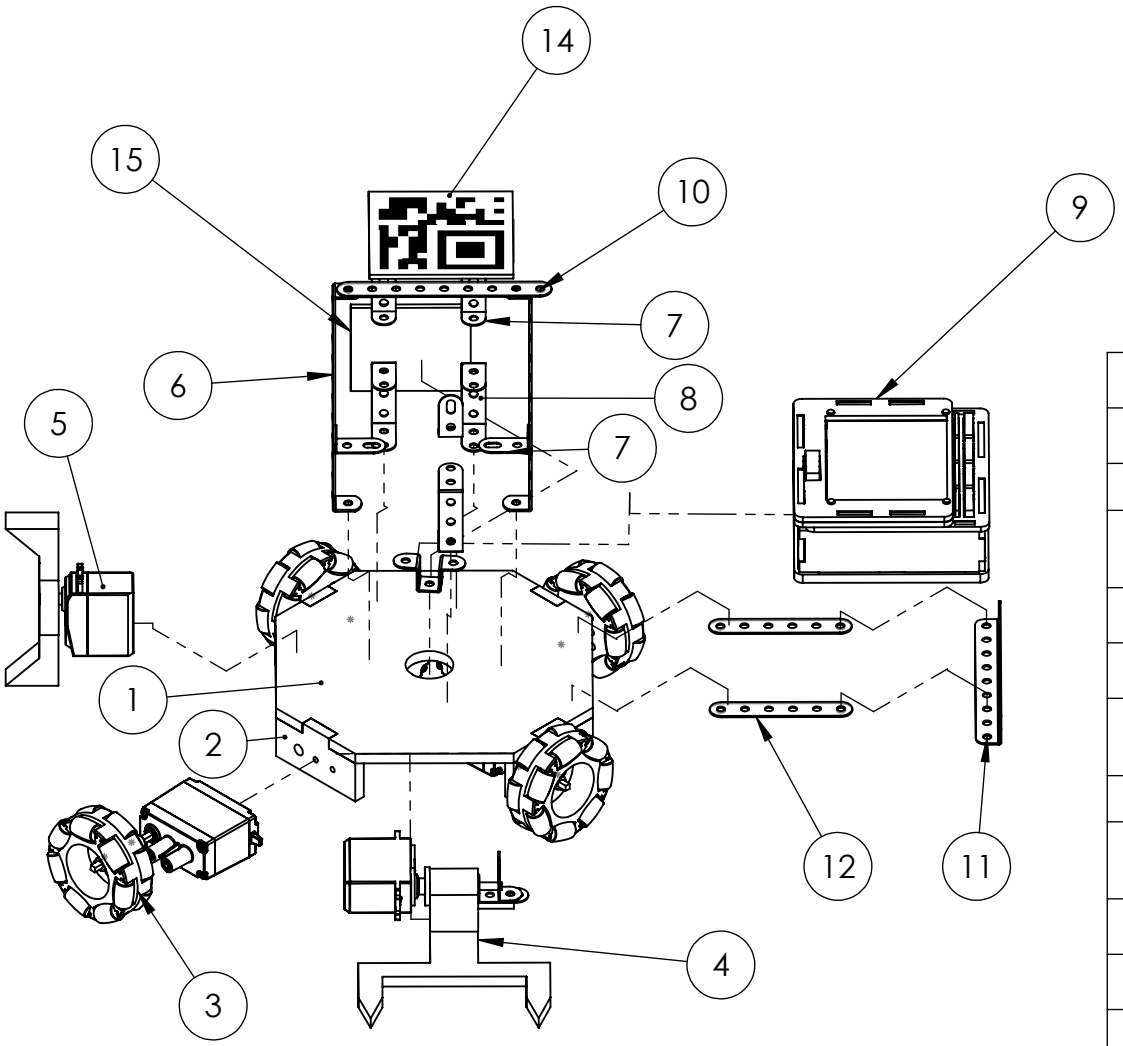






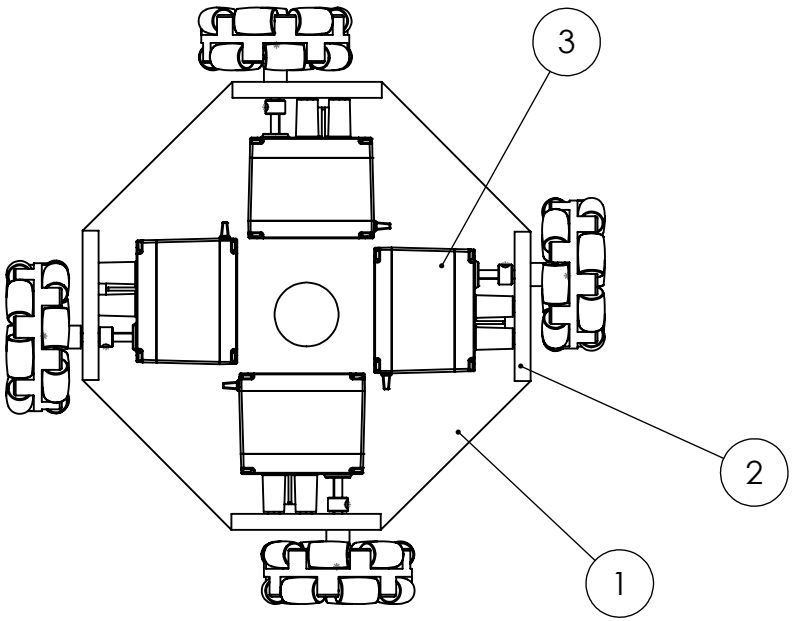
ITEM NO.	PART NUMBER	DESCRIPTION	QTY.
1	Chassis Subassembly		1
2	Forklift Subassembly		1
3	Crank Subassembly		1
4	Double Angle Strip	4.5"X.5" 1X9X1 Hole	2
5	Angle Bracket	1X.5" 2X1 Hole	5
6	Double Angle Strip	1.5"X1" 2X3X2 Hole	3
7	Proteus Controller		1
8	Angle Girder	4.5" 9 Hole	1
9	Double Bent Strip		1
10	Erector Strip	2" 5 Hole	2
11	Erector Strip	4.5" 9 Hole	1
12	Qr Code Plate	3"X3" Acrylic	1
13	Lever Pusher	2.5"X2.5" Cardboard	1
14	CdS Cell		2
19	Screw	#8 Thread, 0.5" Length	11
20	Nut	#8 Thread, Square	9

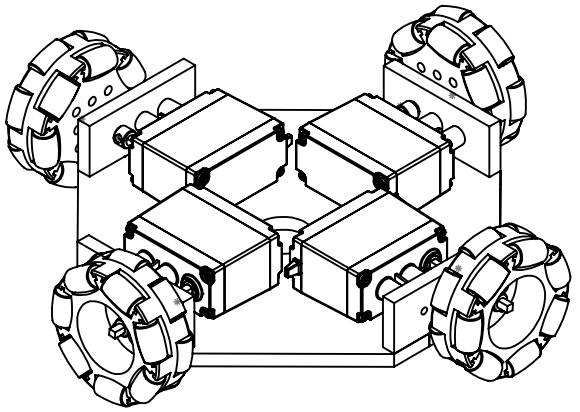
The Ohio State University First Year Engineering	Dwg. Title: FULL ROBOT ASSEMBLY	Inst.: PARKE	Scale: 1:3	Dwg. No.: 01
	Drawn By: TEAM E4	Hour: 12:40	Units: INCH	Date: 4/15/2018

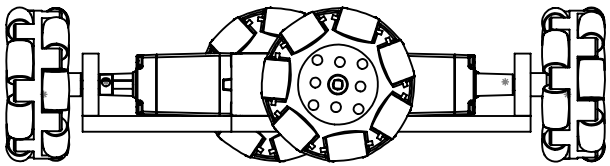


ITEM NO.	PART NUMBER	Material	QTY.
1	Chasis Base	Lazer Cut - MDF	1
2	Wheel Support	Lazer Cut - MDF	4
3	Wheel and Motor Assembly	Subassembly	4
4	Forklift Assembly	Subassembly	1
5	Crank Assembly	Subassembly	1
6	1X9X1 Double Angle	Alluminum	2
7	Angle Bracket 2X1	Alluminum	5
8	Double Angle Strip 2X3X2	Alluminum	3
9	Proteus	Subassembly	1
10	Metal Strip 9X1	Alluminum	1
11	Button Bar	Alluminum	1
12	Button Bar Support	Alluminum	2
13	CDS Mount	Alluminum	1
14	Qr Code Plate	Acrylic	1
15	Lever Pusher	Cardboard	1

The Ohio State University First Year Engineering	Dwg. Title: EXPLODED ASSEMBLY	Inst.: Parke	Scale: 1:4	Dwg. No.: 02
	Drawn By: Team E4	Hour: 12:40	Units: Inches	Date: 4/15/18

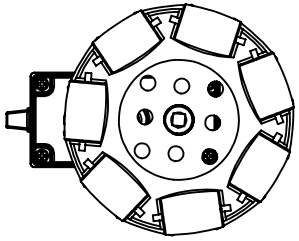


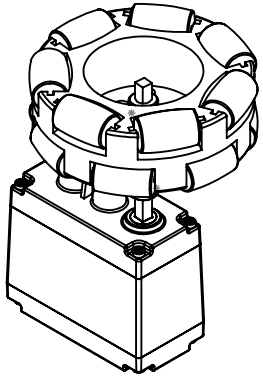


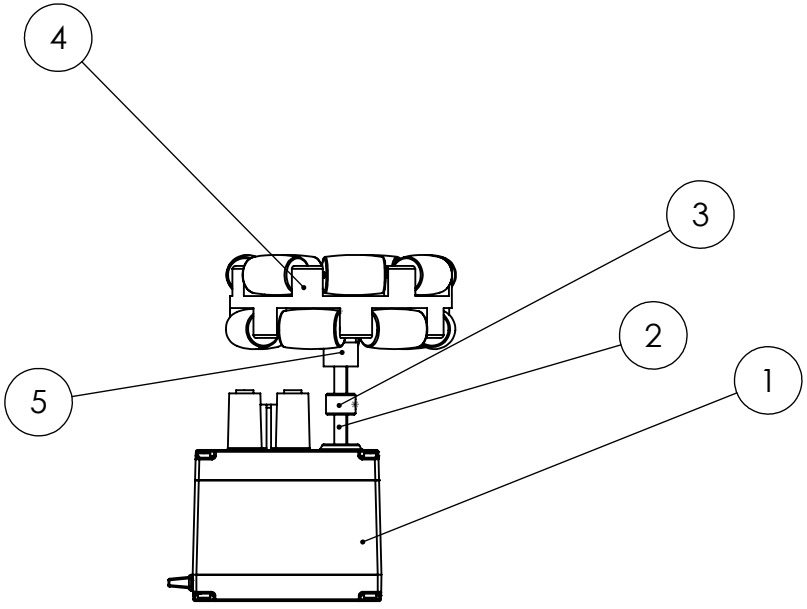


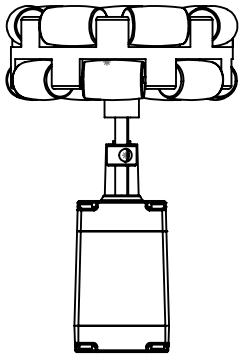
ITEM NO.	PART NUMBER	DESCRIPTION	QTY.
1	Chasis Lazer Cut	MDF	1
2	Wheel Support Lazer Cut	MDF	4
3	A Wheel and Motor Assembly	Various	4

The Ohio State University First Year Engineering	Dwg. Title: CHASSIS AND DRIVETRAIN	Inst.: PARKE	Scale: 1:3	Dwg. No.: 03
	Drawn By: E4	Hour: 12:40	Units: INCHES	Date: 4/15/2018



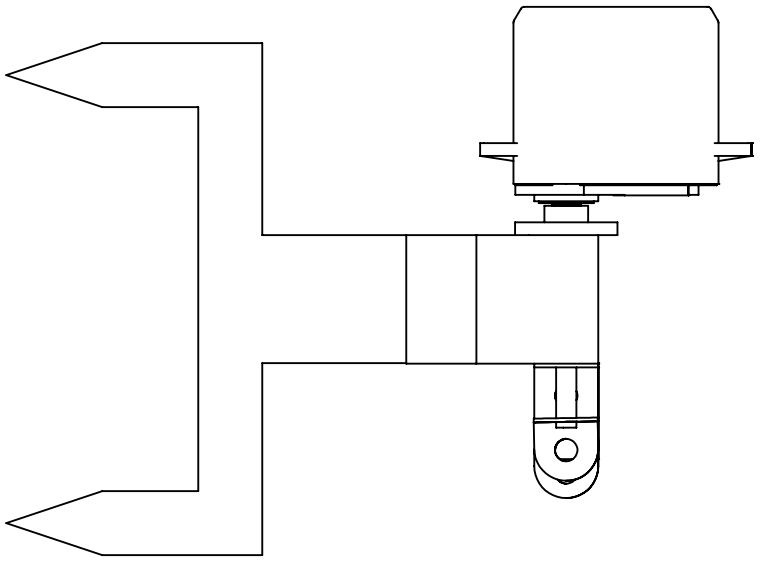


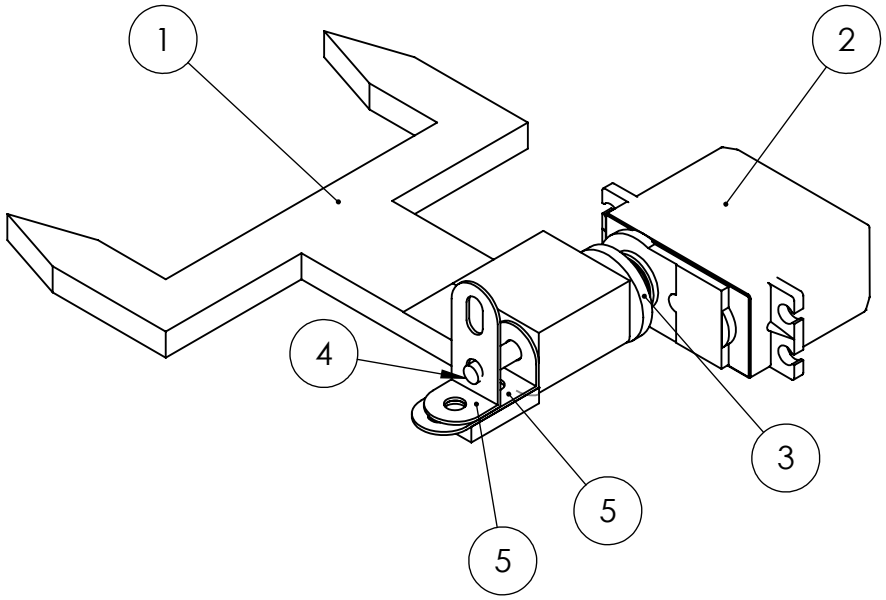


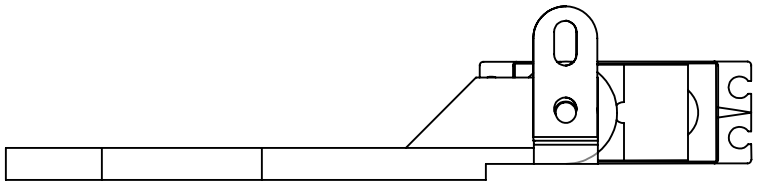


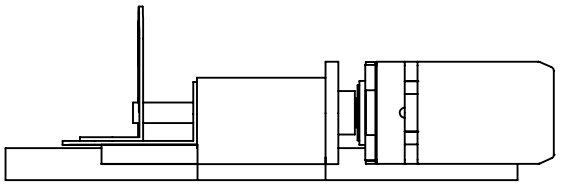
ITEM NO.	PART NUMBER	DESCRIPTION	QTY.
1	VEX_Motor (1)	various	1
2	Vex Axle 2 inch	steel	1
3	Shaft_Lock_Collar	steel	2
4	Omni Assembly	various	1
5	Spacer	plastic	1

The Ohio State University First Year Engineering	Dwg. Title: WHEEL ASSEMBLY	Inst.: PARKE	Scale: 1:2	Dwg. No.: 04
	Drawn By: LUKE KOURY	Hour: 12:40	Units: INCHES	Date: 4/15/2018



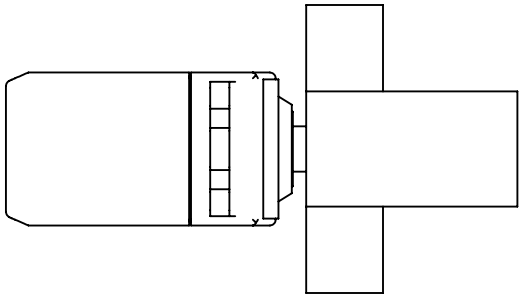


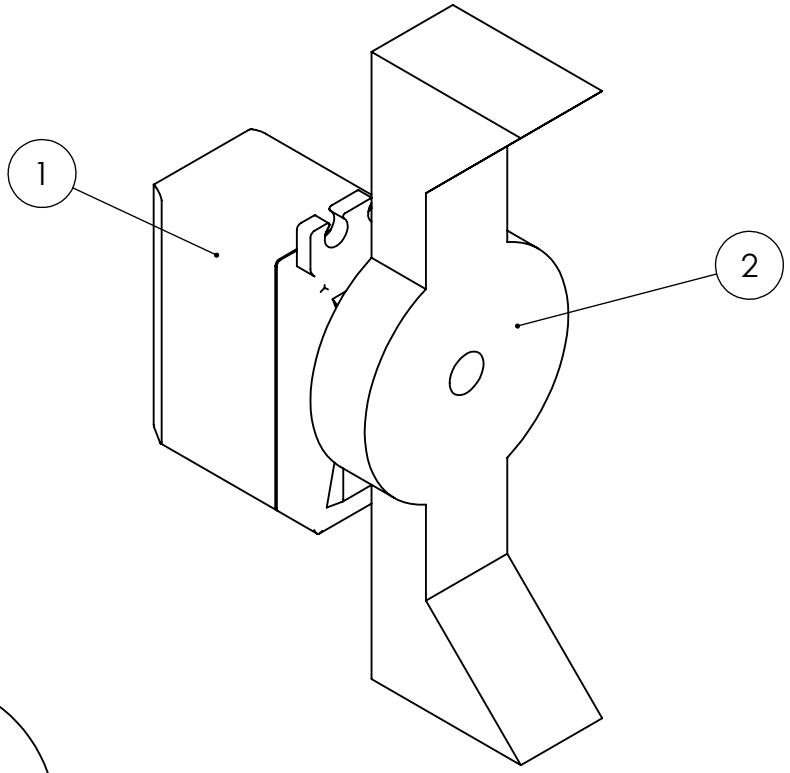


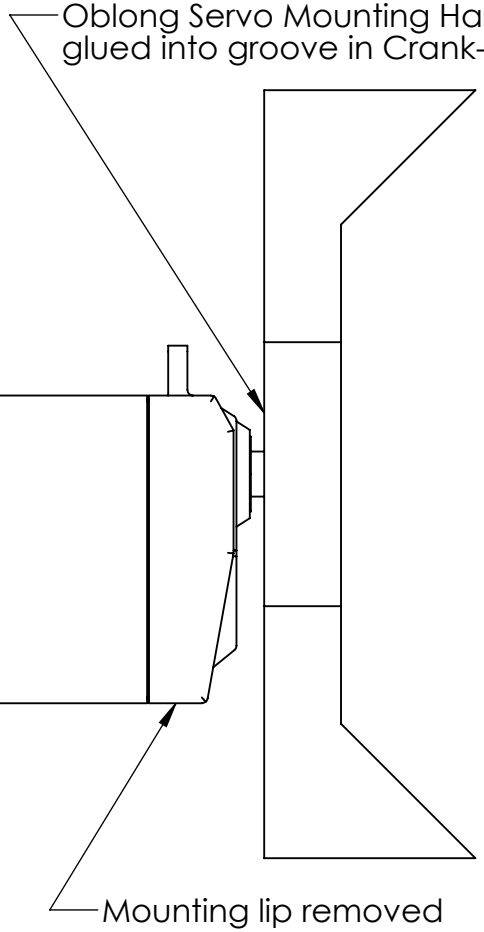


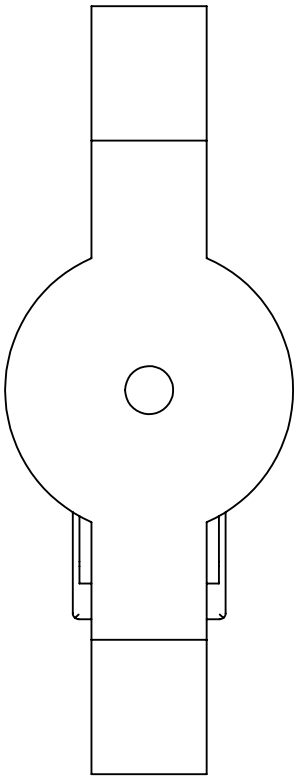
ITEM NO.	PART NUMBER	DESCRIPTION	QTY.
1	Forklift Piece	3D printed in PLA plastic	1
2	FITEC Servo	Standard Part	1
3	FITEC Circular Mounting Hardware	Standard Part	1
4	1-Inch Axle Rod	Standard Part	1
5	Angle Bracket	Standard; 1X.5"; 2X1 Holes	2
6	Forklift Axis Support		1

The Ohio State University First Year Engineering	Dwg. Title: FORKLIFT SUBASSEMBLY	Inst.: PARKE	Scale: 2:3	Dwg. No.: 05
	Drawn By: TEAM E4	Hour: 12:40	Units: INCH	Date: 4/15/2018



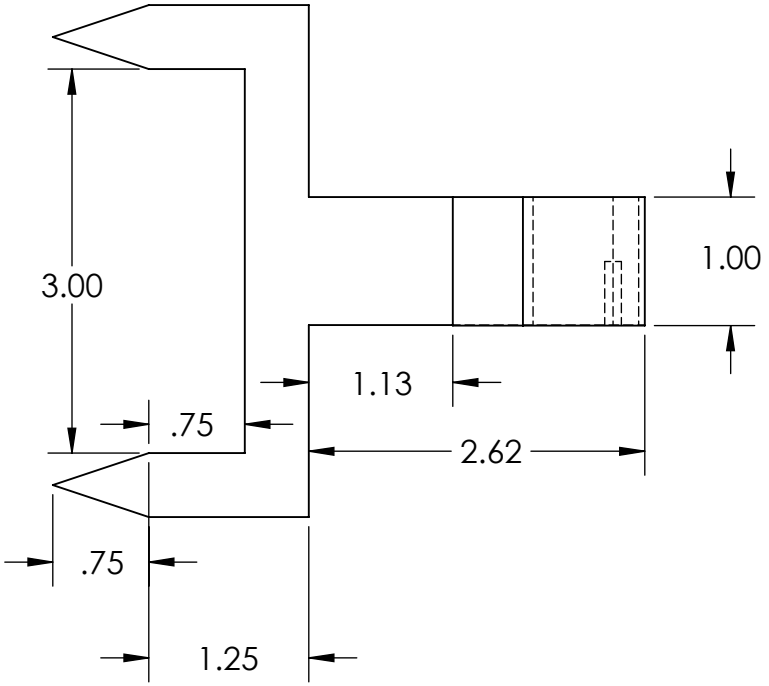


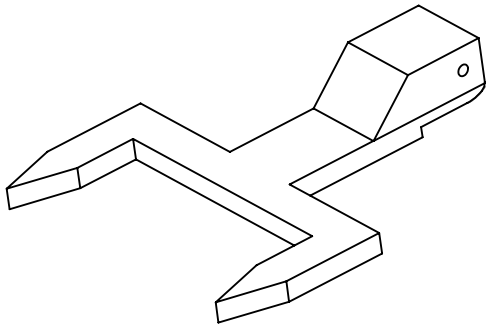


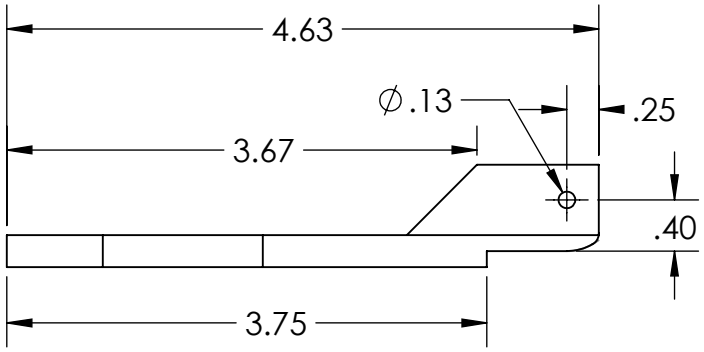


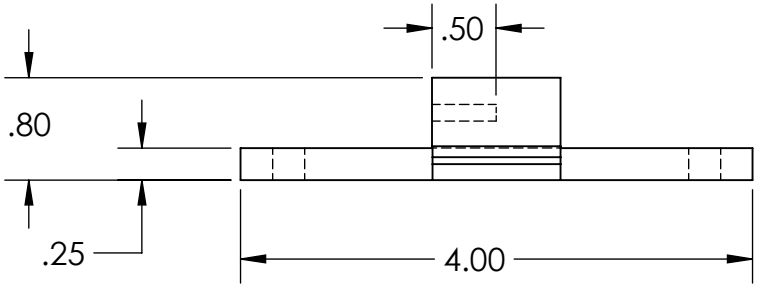
ITEM NO.	PART NUMBER	DESCRIPTION	QTY.
1	Futaba Servo	Mounting lip removed	1
2	Crank-Gripping Piece	3D printed in PLA plastic	1

The Ohio State University First Year Engineering	Dwg. Title: CRANK SUBASSEMBLY	Inst.: PARKE	Scale: 1:1	Dwg. No.: 06
	Drawn By: TEAM E4	Hour: 12:40	Units: INCH	Date: 4/15/2018



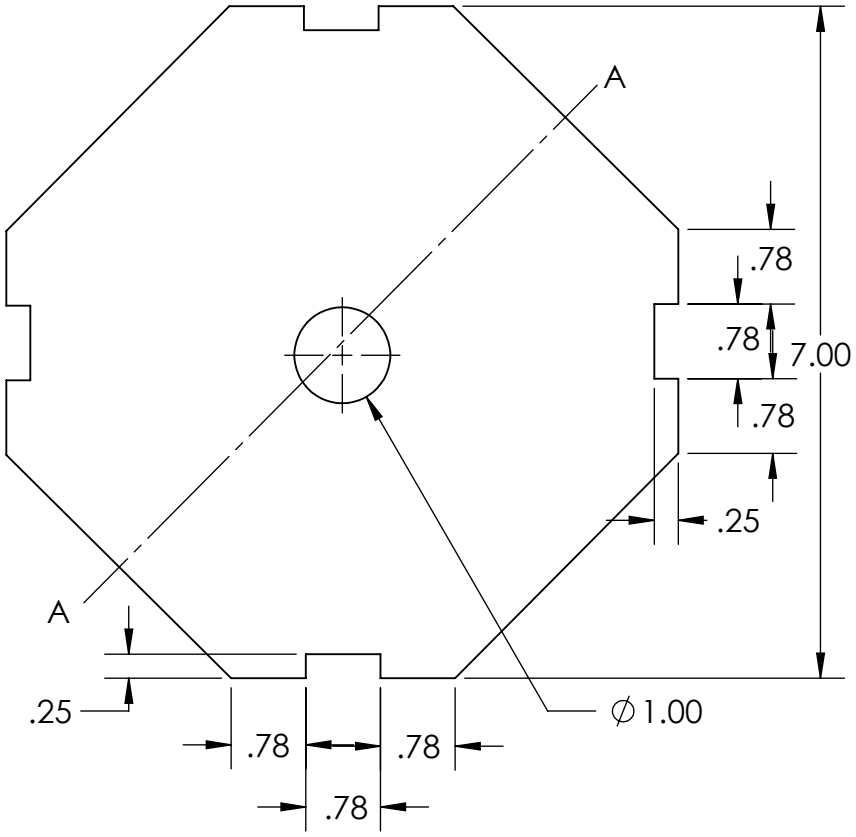


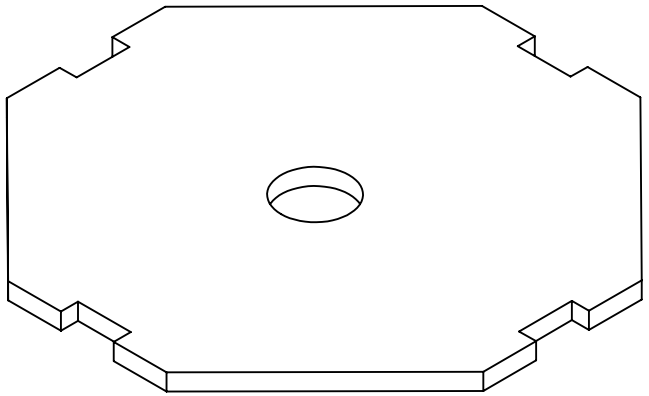


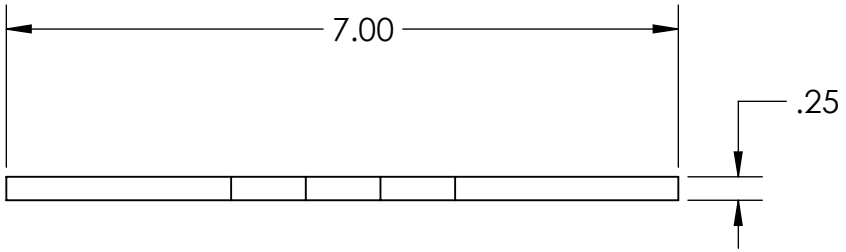


SOLIDWORKS Educational Product. For Instructional Use Only

The Ohio State University First Year Engineering	Dwg. Title: FORKLIFT ARM	Inst.: Parke	Scale: 2:3	Dwg. No.: 07
	Drawn By: Team E4	Hour: 12:40	Units: Inches	Date: 4/15/18

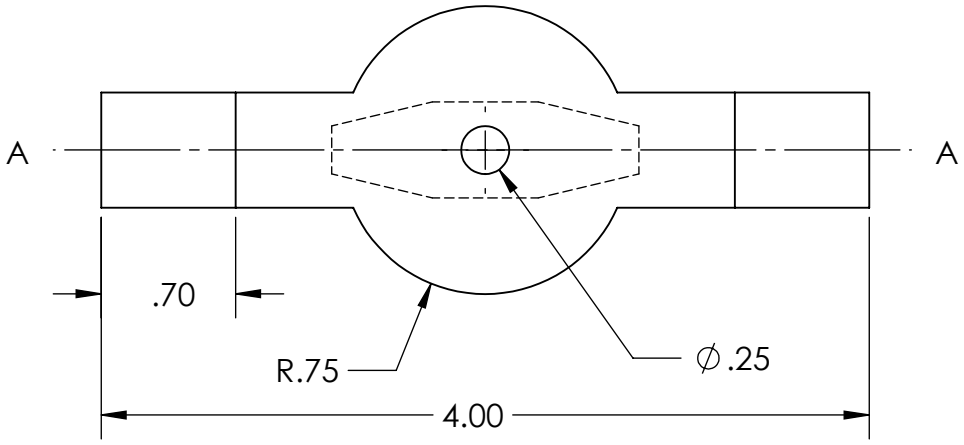


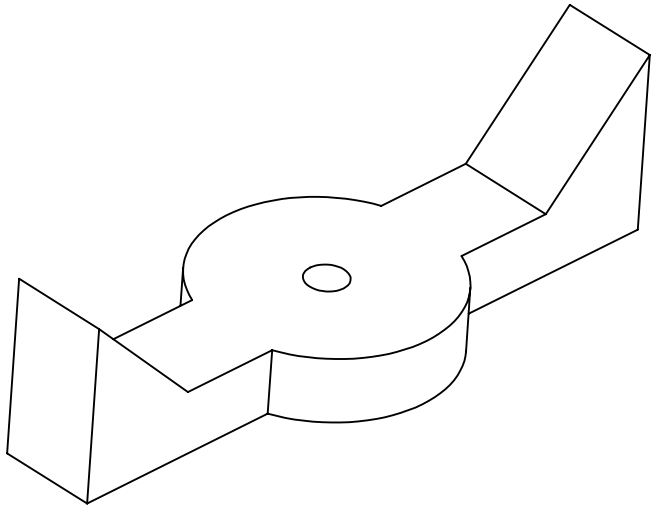


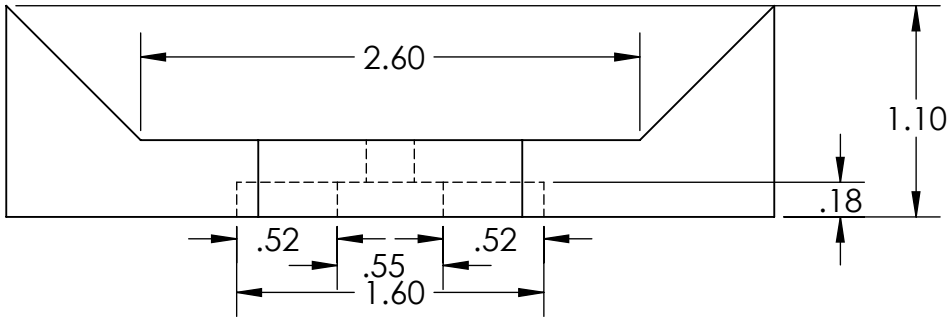


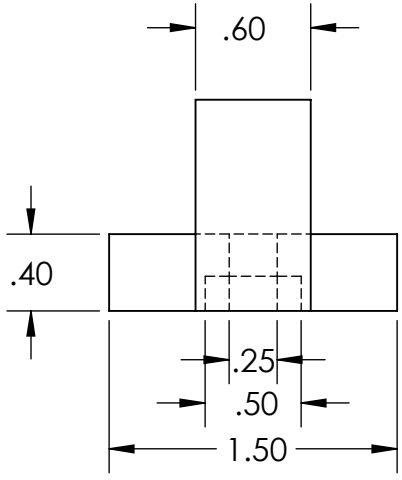
Part is symetric about the A-A axis

The Ohio State University First Year Engineering	Dwg. Title: CHASIS BASE	Inst.: Parke	Scale: 1:2	Dwg. No.: 09
	Drawn By: Team E4	Hour: 12:40	Units: Inches	Date: 4/15/18



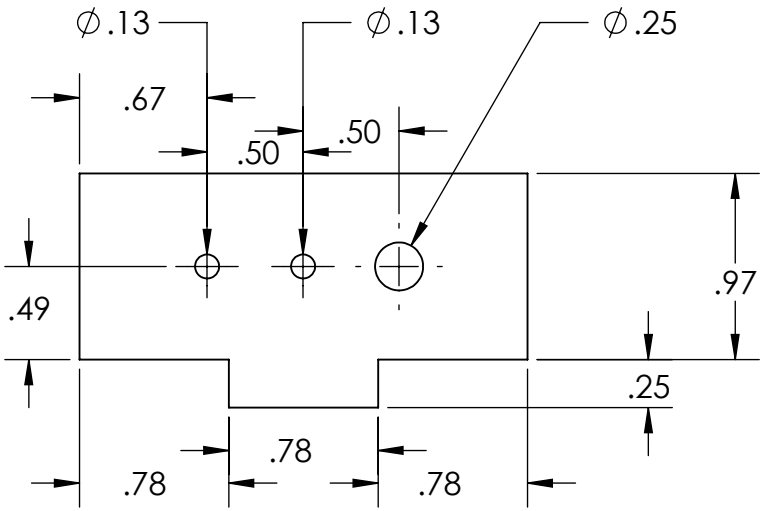


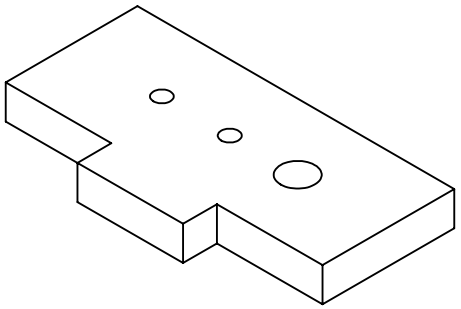


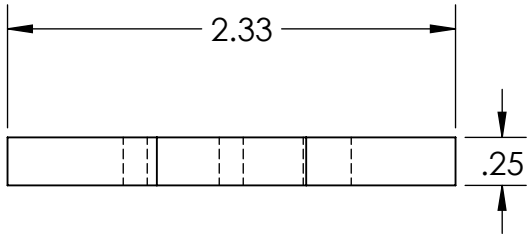


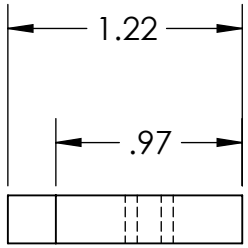
Part is symmetric about the A-A axis

The Ohio State University First Year Engineering	Dwg. Title: CRANK-GRIPPING PIECE	Inst.: Parke	Scale: 1:1	Dwg. No.: 08
	Drawn By: Team E4	Hour: 12:40	Units: Inches	Date: 4/15/18









The Ohio State University First Year Engineering	Dwg. Title: WHEEL SUPPORT	Inst.: Parke	Scale: 1:1	Dwg. No.: 10
	Drawn By: Team E4	Hour: 12:40	Units: Inches	Date: 4/15/18

APPENDIX B

Pseudocode Representation and C++ Code

```
define main:
        setup
        move South 8 inches
        while color not detected:
                move to light position
                detect light color
        if red light:
                move over to red light
                push toward red light
        if blue light:
                move over to blue light
                push blue light

        while white button not touched:
                push toward white button
                break if touched
                move north 2 inches
        while white button not activated:
                push toward white button
                break if pushed for longer that 30 seconds

        move north 4 inches
        move west 16 inches
        move to line up east/west with lever
        move to line up north/south with lever
        push toward lever
        move west 3 inches away from lever
        move north 5 inches

        move 2 inches to the right of the wrench
        lower the forklift
        rotate to wrench heading
        drive west to insert forklift into wrench
        raise forklift

        move 3 inches east way from wrench pedastal
        move to base of west ramp
        move north 12 inches
        move to a point on top of course
        move to center of top of course

        until not in dead zone:
                while in dead zone:
                        move southeast 1.4 inches;
                move to center of top of course

        rotate to heading: -45 degrees
        move blind toward garage
        lower forklift
        move 4.2 inches southeast
        raise forklift
        move to 8 inches south and west of crank
        if clockwise:
                start = 0 degrees
                end = 180 degrees
        if counterclockwise:
                stert = 180 degrees
                end = 0 degrees
        set crank-turner to start
```

```
        move to 2.5 inches south and west of crank
        rotate to crank heading
        move to crank
        set crank-turner to end

        back up 8 inches south and west
        set crank to vertival
        rotate to heading: 0 degrees
        move to top of east ramp
        move to base of east ramp

        loop until program stopped:
                move to center of bottom of course
                move to center of top of course
                move south 3 inches


define setup:
        initialize servos
        initialize RPS
        set servos to RPS calibration angles
        calibrate RPS
        set servos to start angles
        calibrate CdS cells
        wait for cds to read light

define calibrateRPS:
        draw buttons on screen
        until final start button is pressed:
                if a calibrate button is pushed:
                        calibrate that waypoint with current RPS values
                if a revert button is pushed:
                        revert that waypoint to default values
                if motor button pressed:
                        test motors and toggle servos between start and calibration positions
                display calibrations on screen
                display current RPS values on screen
```

```
#include <math.h>
#include <string.h>
#include <FEHIO.h>
#include <FEHLCD.h>
#include <FEHUtility.h>
#include <FEHRPS.h>
#include <FEHSD.h>
#include <FEHMotor.h>
#include <FEHServo.h>
#include <FEHBuzzer.h>
#include <FEHBattery.h>

#define PI_180 0.0174532925
#define SPEEDCONSTANT 13.9 //inches per second
#define ROTATIONCONSTANT 180 //degrees per second
#define FRMT "%.2f "

#define OFF 0
#define REDLIGHT 1
#define BLUELIGHT 2

#define NEWRPS 1
#define NORPS 0
#define OLDRPS 2

#define OCTANE 1
#define NITRO 2

#define SLEEPTIME 0.8 //Time delay for robot to stably halt and RPS to catch up

using namespace std;

// GLOBAL VARIABLES ############################################################

struct POS{
    //store usable values for the robot's current position
    float x;
    float y;
    float heading;
    float timestamp; //when this was last updated
} Robot;

float cdsControl, redControl; //baseline voltages for comparison
float startX, startY, startHeading; //the "tare" that will make (0,0) be the starting
position.
float crankX, crankY, crankHeading; //Results of Calibration waypoints
float wrenchX, wrenchY, wrenchHeading;

int fuelType;
float crankPosition; //current crank orientation in degrees
float runBeginTime=0; //when the run begins, so that the log file has times beginning then.
float movementEndTime=9999999999; //if it gets to this time and a movement hasn't worked, stop
trying to move.
float addToFLWheel=0; //workaround to get more torque when pushing the white button

//Drivetrain
FEHMotor motorFL(FEHMotor::Motor0,7.2);
FEHMotor motorFR(FEHMotor::Motor1,7.2);
FEHMotor motorBL(FEHMotor::Motor2,7.2);
FEHMotor motorBR(FEHMotor::Motor3,7.2);
```

```
//Sensors
AnalogInputPin cds(FEHIO::P0_1);
AnalogInputPin cdsRed(FEHIO::P0_0);

//Servos
FEHServo crankyBoi(FEHServo::Servo0);
FEHServo forkLift(FEHServo::Servo1);

// FUNCTION PROTOTYPES ##################################################

//documentation for log file
void doc(const char *text);
void doc(const char *text, float a);
void doc(const char *text, float a, float b);
void doc(const char *text, float a, float b, float c);
void doc(const char *text, float a, float b, float c, float d);

//setup & sensors
void setupRun();
void calibrateRPS();
bool startWithCds();
void calibrateCds();
int updatePosition();
int updateOnlyXY();
int updateOnlyHeading();
int getColor();
void waitForTouch();
void motorTest();
void reverseMotorTest();

//actions
void raiseForkLift();
void lowerForkLift();
void setCrank(float pos);

//Instantaneous motion commands - they set wheels and take no time.
void setWheels(float fl, float fr, float bl, float br);
void halt();
float setVelocityComponents(float right, float forward, float speedPercent);
float moveAtAngleRelRobot(float heading, float speedPercent);
float moveAtAngleRelCourse(float heading, float speedPercent);
void setRotation(float direction);

//Motion commands that take time
void moveBlind(float angle, float distance, float speedPercent);
void moveBlindTo(float x, float y, float speedPercent);
void moveComponents(float x, float y, float speedPercent);
void rotateBy(float angle, float speedPercent);
void rotateTo(float heading, float precision);
void moveTo(float x, float y, float precision);
void pushToward(float x, float y, float speedPercent, float duration);

//Math functions without side effects
float principal(float x);
float arg(float x1,float y1,float x2,float y2);
float pythag(float x1,float y1,float x2,float y2);
float deltaAngle(float from, float to);

// ######################################################################
// OVERALL PROCEDURE ####################################################
// ######################################################################
```

```
int main(){
    setupRun();

    //############################################################
    //# THE MAIN PROGRAM ########################################
    //############################################################


    doc("MOVING DOWN AND GETTING COLOR");
        moveBlindTo(0,-8,1);
        int color = OFF;
        while(color==OFF){
            moveTo(8.75,-8.5,1);
            color=getColor();
            if(color!=OFF) break;
            Sleep(SLEEPTIME);
            updatePosition();
        }

    doc("GOT COLOR, PUSHING BUTTONS", 0);
        float startTime=TimeNow();
        //Pusher centered 0.5 inches to the left of robot center
        //Buttons at 8.75 +/- 3.25,
        //so buttons are centered on pusher at x=(6 | 9.25 | 11.5)
        float awayFromCenter=3.25*0.5;
        switch(color){
            //slide over to the button, then push toward it for a bit.
            case REDLIGHT:
                LCD.Clear(RED);
                moveBlindTo(9.25-awayFromCenter,-8.5,.5);
                pushToward(9.25-awayFromCenter,-15,.8,0.6);
            break;
            case BLUELIGHT:
                LCD.Clear(BLUE);
                moveBlindTo(9.25+awayFromCenter,-8.5,.5);
                pushToward(9.25+awayFromCenter,-15,.8,0.6);\
            break;
        }

    doc("AIMING FOR WHITE ", startTime-TimeNow());
        while(RPS.IsDeadzoneActive()==0){ //until pushed
            switch(color){
            //hold the white button.
            //Case distinction in case we want to aim toward the safe side
                case REDLIGHT:
                    pushToward(9.25-awayFromCenter/4.0,-14,0.5,0.2);
                break;
                case BLUELIGHT:
                    addToFLWheel = .5;
                    pushToward(9.25,-14,0.5,0.2);
                    addToFLWheel = 0;
                break;
            }
            if(RPS.IsDeadzoneActive()!=0) break;
            moveComponents(0,2,1);
        }
        doc("TOUCHED WHITE", startTime-TimeNow());
        while(RPS.IsDeadzoneActive()!=2 && startTime-TimeNow()<30){ //until activated
            switch(color){
            //hold the white button.
```

```
        //Case distinction in case we want to aim toward the safe side
            case REDLIGHT:
                pushToward(9.25-awayFromCenter/4.0,-14,0.5,0.2);
            break;
            case BLUELIGHT:
                addToFLWheel = 0.5;
                pushToward(9.25,-14,0.5,0.2);
                addToFLWheel = 0;
            break;
        }
        if(TimeNow()-startTime>30) break;
        updatePosition();
    }
    doc("PUSHED FOR", TimeNow()-startTime);

doc("AWAY FROM BOARD");
    moveComponents(0,4,1);      //away from button board
    moveBlindTo(9.25-16,Robot.y,1);
doc("LINE UP FOR LEVER");
    movementEndTime=TimeNow()+8;
        moveTo(wrenchX+1,wrenchY,3);  //line up East-West for lever
doc("LINE UP N/S");
    moveTo(wrenchX-3.0,wrenchY-10,2); //up to lever TODO: consider blind
doc("PUSHING LEVER");
    pushToward(-1.79,-22.1,1,0.8); //push lever
    halt();
doc("AWAY FROM LEVER");
    moveComponents(-3,0,1); //back up
    moveComponents(0,5,1);  //move up to not run into lever again

doc("LINING UP W/WRENCH");
    moveTo(wrenchX+2,wrenchY,0.5); //Move over to wrench
    float forkliftEndTime=TimeNow()+0.8;
    forkLift.SetDegree(180);
    rotateTo(wrenchHeading, 2);
    while(TimeNow()<forkliftEndTime){
        Sleep(1);
    }
doc("INSERTING FORKLIFT");
    moveTo(wrenchX,wrenchY,0.5); // insert into wrench
    raiseForkLift();

if(RPS.IsDeadzoneActive()==2){ //if RPS on top, didn't time out pushing

doc("MOVING TO BASE OF RAMP");
    moveComponents(0,3,1); //avoid running into wrench place
    moveBlindTo(-12,-4.5,1); //base of ramp
doc("MOVING UP RAMP");
    moveComponents(0,12,1);     //up ramp
    updatePosition();
    movementEndTime = TimeNow()+8;
    moveTo(-13,15,4); // make sure up ramp

doc("MOVING TO CENTER");
    moveTo(0,27.5,0.5); //to center of top TODO: bigger tolerance?
    while(updatePosition()==NORPS){
        //in case we overshoot and land in dead zone
        while(updatePosition()==NORPS){
            //inch back out of dead zone
            moveComponents(1,-1,0.5);
        }
```

```
            moveTo(0,27.5,0.5); //to center of top
        }

    doc("LINING UP TO DEPOSIT");
        rotateTo(-45,2); //line up to deposit wrench
        moveBlindTo(-9+0.2, crankY+0.2, 1.0); //up to garage
    doc("DEPOSITING");
        lowerForkLift();
    doc("LEAVING GARAGE");
        moveComponents(3,-3,1); // out of garage
        forkLift.SetDegree(90); // return to vertical
        moveTo(crankX-8,crankY-8,2); //to center(ish) of top

    doc("MOVING TO CRANK");
        float startAngle, endAngle;
        switch(fuelType){
            case OCTANE:
                doc("OCTANE");
                startAngle=0;
                endAngle=180;
            break;
            case NITRO:
                doc("NITRO");
                startAngle=180;
                endAngle=0;
            break;
        }
        crankyBoi.SetDegree(startAngle);
        crankPosition=startAngle;
        moveTo(crankX-2.5,crankY-2.5,0.5); //move to crank TODO:delete?
        rotateTo(crankHeading, 3);
        movementEndTime=TimeNow()+8;
            moveTo(crankX,crankY,0.4); //move to crank
        setCrank(endAngle);
        Sleep(SLEEPTIME);

    doc("LEAVING CRANK");
        moveComponents(-1,-1,.5);
        moveComponents(-7,-7,1.0);
        crankyBoi.SetDegree(90);

    doc("GOING TO RAMP");
        rotateBy(deltaAngle(Robot.heading,0),0.4);    //get level TODO: consider faster
        movementEndTime=TimeNow()+8;
            moveBlindTo(12,14,1.0);    //approach ramp TODO: CONSIDER BLIND
        //rotateBy(deltaAngle(Robot.heading,0),0.4);    //get level TODO: consider faster
        moveTo(12,-5,3);    //descend ramp
        for(int i=0; i<10; i++){
            //please let RPS respond so we don't go the wrong direction
            if(updatePosition()!=NORPS) break;
            Sleep(20);
        }

    } //end if RPS on top

    doc("GOING TO END BUTTON");
        while(true){
            moveBlindTo(0,-5.5,1);     //get to center
            movementEndTime=TimeNow()+4;
            moveTo(0,9,1);      //smack the button
            //PROGRAM SHOULD END HERE
```

```
            LCD.Clear(YELLOW);
            moveComponents(0,-3,1);
        }


    //###################################################################
    //###################################################################
    //###################################################################

    //inaccessible
    doc("Program finished.");
    SD.CloseLog();
    return 0;
}

// ###############################################################################
// STARTUP ########################################################################
// ###############################################################################

void setupRun(){
    /*
     * All pre-run tasks
     */
    //forkLift.TouchCalibrate();
    //crankyBoi.TouchCalibrate();
    forkLift.SetMin(690); forkLift.SetMax(2215);
    crankyBoi.SetMin(515); crankyBoi.SetMax(2300);

    RPS.InitializeTouchMenu();
    //SD.OpenLog();
    doc("Voltage: ", Battery.Voltage());

    forkLift.SetDegree(180);
    crankyBoi.SetDegree(180);
    calibrateRPS();
    movementEndTime=9999999999;

    LCD.Clear();
    forkLift.SetDegree(90);
    crankyBoi.SetDegree(90);
    crankPosition=90;
    calibrateCds();
    LCD.Clear(BLACK);
    //waitForTouch();
    LCD.Clear(0x99ff99);
    doc("Prepare for Domination.");
    doc("Waiting for CdS.");
    if(updatePosition()==NORPS){
        Robot.x=0; Robot.y=0;
        Robot.heading=0;
    }
    startWithCds();
    fuelType=RPS.FuelType();
}
void motorTest(){
    /*
     * Test all six motors; leave them in starting position.
     */
    forkLift.SetDegree(180);
    crankyBoi.SetDegree(180);
    setWheels(1,0,0,0); Sleep(250);
```

```
    crankyBoi.SetDegree(0);
    setWheels(0,1,0,0); Sleep(250);
    forkLift.SetDegree(90);
    setWheels(0,0,1,0); Sleep(250);
    crankyBoi.SetDegree(90);
    setWheels(0,0,0,1); Sleep(250);
    halt();
}
void reverseMotorTest(){
    /*
     * Move the servos to calibration positions.
     */
    forkLift.SetDegree(180);
    crankyBoi.SetDegree(180);
    Sleep(250);
}

void waitForTouch(){
    /*
     *  Wait until the screen is touched.
     *  Beep when touched.
     */
    float x,y;
    while(LCD.Touch(&x,&y)) Sleep(1); //until untouched
    while(!LCD.Touch(&x,&y)) Sleep(1); //until pressed
    while(LCD.Touch(&x,&y)) Sleep(1); //until released
    Buzzer.Beep();
}

// ###########################################################################
// CdS CELL SENSORS ##########################################################
// ###########################################################################

void calibrateCds(){
    /*
     * Reads the CdS cell average, and sets control to it.
     * Takes 1.4 seconds in total.
     */
    LCD.WriteLine("Touch to calibrate CdS.");
    //waitForTouch();
    Sleep(1.0); //Wait a bit to avoid interfenece of user
    float sum=0, redsum=0;
    int numCalibrations=100;
    for(int i=0; i<numCalibrations; i++){
        sum+=cds.Value();
        redsum+=cdsRed.Value();
        Sleep(4);
    }
    cdsControl = sum / numCalibrations;
    redControl = redsum / numCalibrations;
    doc("CdS baseline:", cdsControl);
    doc("Red CdS baseline:", redControl);
}
bool startWithCds(){
    /*
     *  Wait until the CdS sensor consistently reads
     *  bright (low) values. Return true if successful.
     */
    int numreadings = 50;
    float readings[numreadings];
    for(int i=0;i<numreadings;i++)readings[i]=4;
```

```
    float sum=4*numreadings;
    float m=cdsControl*0.8; //threshhold in volts (20% brightness)
    int panicTime = TimeNow() + 40; //after 40 seconds, go anyway.
    while(sum/numreadings>m){
        if(TimeNow()>panicTime){
            doc("Going without CdS.");
            return false;
        }
        //move only after average of 20 under threshold.
        sum-=readings[0];
        for(int i=0;i<numreadings-1;i++)
            readings[i]=readings[i+1];
        readings[numreadings-1]=cds.Value();
        sum+=readings[numreadings-1];
        Sleep(2);
    }
    runBeginTime=TimeNow();
    doc("Going with CdS.", sum/numreadings, m);
    return true;
}
int getColor(){
    /*
     * Reads the CdS cells and determines the light color code.
     */
    float avg=0, redavg=0;
    int numCalibrations=50;
    for(int i=0; i<numCalibrations; i++){
        avg+=cds.Value();
        redavg+=cdsRed.Value();
        Sleep(2);
    }
    avg/=numCalibrations;
    redavg/=numCalibrations;

    float brightness = (cdsControl-avg)/cdsControl;
    float redness = (redControl-redavg)/redControl;

    int color;

    if(brightness<.3){
        color=OFF;
    }else if(redness>.8*brightness){
        color=REDLIGHT;
    }else{
        color=BLUELIGHT;
    }

    doc("Bright/red/color:", brightness, redness, color);
    return color;
}

// ###############################################################################
// ACTIONS ########################################################################
// ###############################################################################

void raiseForkLift(){
    for(int i=180; i>95; i--){
        forkLift.SetDegree(i);
        Sleep(8);
    }
}
```

```
void lowerForkLift(){
    float endTime = TimeNow()+0.4;
    forkLift.SetDegree(180);
    while(TimeNow()<endTime);
}

void setCrank(float pos){
    for(float i=crankPosition; i<=pos; i+=(pos-crankPosition)/60){
        crankyBoi.SetDegree(i);
        Sleep(0.5/60);
    }
    crankyBoi.SetDegree(pos);
    crankPosition=pos;
}

// ###########################################################################
// RPS #######################################################################
// ###########################################################################

int updatePosition(){
    /*
     *  Reads the RPS data into the 'Robot' global variable if valid and new
     *  (The "Timestamp" functionality is not currently used.)
     */
    float x = RPS.X()-startX;
    float y = RPS.Y()-startY;
    float heading = RPS.Heading();
    if(heading>-1){
        heading=principal(heading-startHeading);
        if(Robot.x == x && Robot.y == y && Robot.heading == heading){
            //Do not update timestamp if RPS hasn't been updated
            //doc("RPS Stayed the same");
            return OLDRPS;
        } else {
            Robot.x = x;
            Robot.y = y;
            Robot.heading = heading;
            Robot.timestamp=TimeNow();
            //doc("RPS: ", x, y, heading);
            return NEWRPS;
        }
    }
    //doc("RPS Update Failed: ",x,y,heading);
    return NORPS;
}
int updateOnlyXY(){
    /*
     *  Reads the RPS data into the 'Robot' global variable if valid and new
     *  (The "Timestamp" functionality is not currently used.)
     */
    float x = RPS.X()-startX;
    float y = RPS.Y()-startY;
    float heading = RPS.Heading();
    if(heading>-1){
        heading=principal(heading-startHeading);
        if(Robot.x == x && Robot.y == y && Robot.heading == heading){
            //Do not update timestamp if RPS hasn't been updated
            //doc("XY Stayed the same");
            return OLDRPS;
        } else {
            Robot.x = x;
```

```
            Robot.y = y;
            //Robot.heading = heading; DON'T UPDATE
            Robot.timestamp=TimeNow();
            doc("XY:  ", x, y, heading);
            return NEWRPS;
        }
    }
    //doc("XY Update Failed: ",x,y,heading);
    return NORPS;
}
int updateOnlyHeading(){
    /*
     *  Reads the RPS data into the 'Robot' global variable if valid and new
     *  (The "Timestamp" functionality is not currently used.)
     */
    float x = RPS.X()-startX;
    float y = RPS.Y()-startY;
    float heading = RPS.Heading();
    if(heading>-1){
        heading=principal(heading-startHeading);
        if(Robot.x == x && Robot.y == y && Robot.heading == heading){
            //Do not update timestamp if RPS hasn't been updated
            //doc("Heading Stayed the same");
            return OLDRPS;
        } else {
            //Robot.x = x; DON'T UPDATE
            //Robot.y = y;
            Robot.heading = heading;
            Robot.timestamp=TimeNow();
            //doc("Heading: ", x, y, heading);
            return NEWRPS;
        }
    }
    //doc("Heading Update Failed: ",x,y,heading);
    return NORPS;
}


class waypoint{
    //points to caibrate before run
public:
    //fill constructor
    waypoint(char* name, float x, float y, float head, float screenLeft, float screenTop,
float w, float h);
    //defaults in case we don't have time to calibrate
    float myDefaultX;
    float myDefaultY;
    float myDefaultHeading;
    //current values
    float myX;
    float myY;
    float myHeading;
    //GUI buttons
    FEHIcon::Icon calButton;
    FEHIcon::Icon revertButton;
    void calibrateHere();
    void revert(); //to defaults
    bool calibrated; //changed from defaults
    unsigned int labelColor(); //color dependent on accuracy
};
```

```
    waypoint::waypoint(char* name, float x, float y, float head, float screenLeft, float
screenTop, float w, float h){
        waypoint::myDefaultX=x; waypoint::myDefaultY=y; waypoint::myDefaultHeading=head;
        waypoint::myX=myDefaultX; waypoint::myY=myDefaultY;
waypoint::myHeading=myDefaultHeading;
        waypoint::calibrated=false;
        waypoint::calButton.SetProperties(name,screenLeft+1, screenTop+1,w/2-2,h-2,GOLD,GOLD);
        waypoint::revertButton.SetProperties("Revert",screenLeft+w/2+1, screenTop+1,w/
2-2,h-2,PINK,PINK);
        waypoint::calButton.Draw();
        waypoint::revertButton.Draw();
    }
    void waypoint::calibrateHere(){
        Buzzer.Beep();
        Robot.x=-1; Robot.y=-1;
        float endTime=TimeNow()+2;
        while((Robot.x<0 || Robot.y<0) && TimeNow()<endTime){
            Robot.x=RPS.X();
            Robot.y=RPS.Y();
            Robot.heading=RPS.Heading();
            if( !(Robot.x<0 || Robot.y<0) ){
                calibrated=true;
            }
        }
        waypoint::myX=Robot.x;
        waypoint::myY=Robot.y;
        waypoint::myHeading=Robot.heading;

    }
    void waypoint::revert(){
        //revert x,y to defaults
        Buzzer.Beep();
        waypoint::myX=waypoint::myDefaultX;
        waypoint::myY=waypoint::myDefaultY;
        waypoint::myHeading=waypoint::myDefaultHeading;
        waypoint::calibrated=false;
    }
    unsigned int waypoint::labelColor(){
        //color depends on how close calibration is to expected value
        if(waypoint::calibrated){
            float d=0.05*fabs(deltaAngle(
                        waypoint::myHeading,
                        waypoint::myDefaultHeading)
                    )
                    +pythag(
                        waypoint::myX,waypoint::myY,
                        waypoint::myDefaultX,waypoint::myDefaultY
                    );
            if(d<1){
                return 0x00FF00;
            }else if(d<3){
                return YELLOW;
            }else{
                return RED;
            }
        }
        return GRAY;
    }

void calibrateRPS(){
    /*
```

```
     * Allows us to re-calibrate exact cooordinates before each run.
     *
     * Creates a GUI that will be used as follows:
     *  1. Move Robot to Crank and press "Crank"
     *  2. Move Robot to Wrench and press "Wrench"
     *  3. Press "Motors" To test all 6 Motors and move them to starting positions
     *  4. Move Robot to Origin (starting light) and press "Origin"
     *  5. Press "GO!" to begin.
     *
     *  -If any of these steps is not followed, we should be fine because of the defaults.
     *  -The GUI includes the default values (GRAY) next to the current waypoint values.
     *  -The current waypoint values are colored by their distance from the default
     *      -Red=more than 3 inches away; Green=closer than 1in; Yellow=between
     *      -Gray=hasn't been changed from default
     *  -If one of the values is incorrect, Recalibrate OR press "Revert" to reset to default.
     *  -Pressing "Motors" again will return motors to original state for continuing
calibration.
     */

    LCD.Clear();
    waypoint crank("Crank",     25.4,63.9,315   ,160,0,160,60);
    waypoint wrench("Wrench",    8.4,17.9,0     ,160,60,160,60);
    waypoint origin("Origin",   17.1,29.0,0     ,160,120,160,60);
    FEHIcon::Icon motorButton, goButton;
    bool motorsTested=false;
    motorButton.SetProperties("Motors",161,181,78,56,CYAN,CYAN);
    goButton.SetProperties("GO!",241,181,78,56,0x00FF00,0x00FF00); //green
    goButton.Draw();
    motorButton.Draw();

    LCD.SetFontColor(GRAY); //Display Defaults
    LCD.WriteAt(crank.myDefaultX, 0,10);     LCD.WriteAt(crank.myDefaultY, 80,10);
    LCD.WriteAt(wrench.myDefaultX, 0,70);   LCD.WriteAt(wrench.myDefaultY, 80,70);
    LCD.WriteAt(origin.myDefaultX, 0,130);  LCD.WriteAt(origin.myDefaultY, 80,130);

    float x,y; //screen position touched
    while(true){
        LCD.Touch(&x,&y); //get the point that was touched on thet screen.
        if(goButton.Pressed(x,y,1)) break;
        if(crank.calButton.Pressed(x,y,1))      crank.calibrateHere();
        if(wrench.calButton.Pressed(x,y,1))     wrench.calibrateHere();
        if(origin.calButton.Pressed(x,y,1))     origin.calibrateHere();
        if(crank.revertButton.Pressed(x,y,1))   crank.revert();
        if(wrench.revertButton.Pressed(x,y,1))  wrench.revert();
        if(origin.revertButton.Pressed(x,y,1))  origin.revert();
        if(motorButton.Pressed(x,y,1)){
            if(!motorsTested){
                motorTest();
                motorsTested=true;
            } else {
                reverseMotorTest();
                motorsTested=false;
            }
        }
        //Display calibrations
        LCD.SetFontColor(crank.labelColor());
            LCD.WriteAt(crank.myX, 0,40);   LCD.WriteAt(crank.myY, 80,40);
        LCD.SetFontColor(wrench.labelColor());
            LCD.WriteAt(wrench.myX, 0,100); LCD.WriteAt(wrench.myY, 80,100);
        LCD.SetFontColor(origin.labelColor());
            LCD.WriteAt(origin.myX, 0,160); LCD.WriteAt(origin.myY, 80,160);
```

```
        LCD.SetFontColor(WHITE); //Display Current position
        LCD.WriteAt(RPS.X(), 0,197); LCD.WriteAt(RPS.Y(), 80,197);
        LCD.WriteAt(RPS.Heading(), 40,215);
        Sleep(1);
    }

    doc("Crank RPS", crank.myX, crank.myY, crank.myHeading);
    doc("Wrench RPS", wrench.myX, wrench.myY, wrench.myHeading);
    startX=origin.myX;
    startY=origin.myY;
    startHeading=origin.myHeading;
    doc("RPS Tare:",startX,startY,startHeading);
    crankX=crank.myX-startX;
    crankY=crank.myY-startY;
    crankHeading=principal(crank.myHeading-startHeading);
    doc("Crank Position:",crankX, crankY, crankHeading);
    wrenchX=wrench.myX-startX;
    wrenchY=wrench.myY-startY;
    wrenchHeading=principal(wrench.myHeading-startHeading);
    doc("Wrench Position:",wrenchX, wrenchY, wrenchHeading);

    updatePosition();
}

// ###########################################################################
// MOVEMENT ##################################################################
// ###########################################################################
//      roughly in order of dependence, most primitive first.

void setWheels(float fl, float fr, float bl, float br){
    /*
     *  Sets wheels to given speeds.
     *  Positive is counterclockwise from perspective of robot;
     *  Ex: setWheels(1,1,1,1);
     *      will make the robot spin counterclockwise real fast.
     */
    motorFL.SetPercent(100.0*fl);
    motorFR.SetPercent(100.0*fr);
    motorBL.SetPercent(100.0*bl);
    motorBR.SetPercent(100.0*br);
}
void halt(){
    //doc("Halting");
    setWheels(0,0,0,0);
}
float setVelocityComponents(float right, float forward, float speedPercent){
    /*
     *  Sets the x- and y-velocities at speedPercent of the maximum.
     *  arguments between -1 and 1.
     *
     *  Ex: setVelocityComponents(-1, 1, 0.75); will set Robot going diagonally
     *       forward and to the left at 75% Speed
     *
     *  returns speed in inches/second
     */

    //Wheel values based on orientation of wheels:
    float fl=-forward-right + addToFLWheel;
    float fr=+forward-right;
    float bl=-forward+right;
```

```
    float br=+forward+right;

    //Scale back values so speed is maximum and no values greater than 1:
    float m=fmax( fmax( fabs(fl),fabs(fr) ), fmax( fabs(bl),fabs(br) ) );
    if(fabs(m)>.001){
        fl/=m; fr/=m; br/=m; bl/=m;
    }
    fl*=speedPercent; fr*=speedPercent; br*=speedPercent; bl*=speedPercent;
    setWheels(fl,fr,bl,br);

    //pythagorean theorem between two components of motion to return speed
    float speed = SPEEDCONSTANT * sqrt(fl*fl+fr*fr); //in inches/second
    //doc("Wheels set: ",fl,fr);
    //doc("Velocity:", right, forward, speed);
    return speed;
}
float moveAtAngleRelRobot(float heading, float speedPercent){
    /*
     *  Sets Robot moving at an angle (0=right, 90=forward)
     */
    //doc("RelRobot: ", heading, speedPercent);
    return setVelocityComponents(
                cos(PI_180*heading),
                sin(PI_180*heading),
                speedPercent);
}
float moveAtAngleRelCourse(float heading, float speedPercent){
    /*
     *  Sets Robot moving at an angle (0=East, 90=North)
     */
    //doc("RelCourse: ", heading, speedPercent);
    return moveAtAngleRelRobot(principal(heading-Robot.heading), speedPercent);
}
void moveBlind(float angle, float distance, float speedPercent){
    /*
     * Moves robot at angle (RelCourse), by distance, at speedPercent,
     * without RPS.
     */
    float speed = moveAtAngleRelCourse(angle, speedPercent);
    float endTime = TimeNow()+distance/speed;
    while(TimeNow()<endTime){
        updateOnlyHeading(); //Shouldn't be changing, so consistent enough to record.
        Sleep(2);
    }
    halt();

    //This is the one place where we assume that the robot's
    //  intended motion has actually happened.
    //  (Along with wherever this function is called)
    Robot.x += distance*cos(PI_180*angle);
    Robot.y += distance*sin(PI_180*angle);

    doc("blindPosition", Robot.x, Robot.y);

}
void moveBlindTo(float x, float y, float speedPercent){
    /*
     * Moves the robot to (x,y) without help of RPS.
     */
    float angle = arg(Robot.x, Robot.y, x, y);
    float distance = pythag(Robot.x, Robot.y, x, y);
```

```
    moveBlind(angle, distance, speedPercent);
}
void moveComponents(float dx, float dy, float speedPercent){
    /*
     * Moves robot by <x,y> on course without the help of RPS.
     * Note: we don't do this directly with setVelocityComponents
     *       because the robot could be tilted
     */
    moveBlindTo(Robot.x+dx, Robot.y+dy, speedPercent);
}
void moveTo(float x, float y, float precision){
    /*
     *  Moves the robot to within precision of (x,y).
     *  Is the most frequently called function in the main program!
     */
    doc("moveTo", x, y, precision);
    float distance = pythag(Robot.x,Robot.y, x, y);
    if(distance>precision && TimeNow()<=movementEndTime){
        float speedPercent=1;
        if(distance<6) speedPercent=.4; //slower if closer TODO: Make faster?
        moveBlindTo(x,y,speedPercent);
        Sleep(SLEEPTIME); //wait for RPS to catch up
        updatePosition();
        moveTo(x, y, precision); //Recursion is always good and cool, right?
    } else {
        //Finish moving
        movementEndTime=99999999999;  //so as not to affect the next movement
    }
}

void pushToward(float x, float y, float speedPercent, float duration){
    /*
     * Pushes toward a particular point for some duration.
     * Updates position and adjusts, so more stable than simple pushAgainst.
     * Used to push buttons.
     */
    float endTime=TimeNow()+duration;
    while(TimeNow()<endTime){
        float angle = arg(Robot.x, Robot.y, x, y);
        moveAtAngleRelCourse(angle, speedPercent);
        Sleep(100); //update motion every 100 ms
        updatePosition();
    }
}

void setRotation(float direction){
    /*
     *  Sets Robot Rotating at given speed/direction
     *  Ex. setRotation(.5); makes robot start spinning counterclockwise
     *      at 50% speed.
     */
    if(fabs(direction)>1) direction=(direction>0? 1:-1); //ensure no overflow
    //doc("Setting Rotation:", direction);
    setWheels(direction, direction, direction, direction);
}
void rotateBy(float angle, float speedPercent){
    /*
     *  Rotates robot by given angle and %speed.
     */
    //doc("Rotating by:", angle, speedPercent);
    setRotation(speedPercent*(angle>0? 1:-1));
```

```
    float timeEnd=TimeNow()+fabs(angle)/(speedPercent*ROTATIONCONSTANT);
    while(TimeNow()<timeEnd){
        updateOnlyXY();
        Sleep(2);
    }
    halt();
    Robot.heading=principal(Robot.heading+angle);
    doc("Rotation Finished.");
}
void rotateTo(float heading, float precision){
    /*
     *  Rotates the robot to face the specified heading.
     *  (Within "precision" degrees)
     */
    updatePosition();
    float rotationAngle=deltaAngle(Robot.heading, heading);
    if(fabs(rotationAngle)>precision){
        doc("Rot from/to/by:", Robot.heading, heading, rotationAngle);
        float angleSpeed=.6;
        if(fabs(rotationAngle)<90) angleSpeed=.4;
        rotateBy(rotationAngle, angleSpeed);
        Sleep(SLEEPTIME);
        rotateTo(heading, precision);
    }

}

// ###########################################################################
// MATH FUNCTIONS WITHOUT SIDE EFFECTS #######################################
// ###########################################################################
//      note: angles measured in degrees counteclockwise from +x axis

float principal(float x){
    //returns x's coterminal angle in [0,360).
    while(x>=360) x-=360;
    while(x<0) x+=360;
    return x;
}
float arg(float x1, float y1, float x2, float y2){
    //returns the angle from (x1,y1) to (x2,y2)
    return atan2(y2-y1,x2-x1)/(PI_180);
}
float pythag(float x1, float y1, float x2, float y2){
    //returns the distance from (x1,y1) to (x2,y2)
    return sqrt(pow(y2-y1,2)+pow(x2-x1,2));
}
float deltaAngle(float from, float to){
    //returns the angle from "from" to "to", between -180 and +180
    //Much much simpler algorithm than weird if statements.
    float angle = to-from;
    while(angle>180) angle-=360;
    while(angle<=-180) angle+=360;
    return angle;
}


// ###########################################################################
// DATA LOGGING ##############################################################
// ###########################################################################
// Functions for displaying stuff on screen and logging to the SD card.
```

```
// Accept a string and 0-4 floating point numbers to document.

void doc(const char *text){
    SD.Printf(FRMT, TimeNow()-runBeginTime);
    SD.Printf(text);        LCD.Write(text); LCD.Write(" ");
    SD.Printf("\r\n");      LCD.Write('\n');
    //Fun fact: Windows doesn't recognize "\n" as a line break,
    //  but it does recognize "\r\n",
    //  so you can open these files straight in Notepad.
    //If you're an FEH TA/Instructor reading this, tell future students.
}
void doc(const char *text, float a){
    SD.Printf(FRMT, TimeNow()-runBeginTime);
    SD.Printf(text);        LCD.Write(text); LCD.Write(" ");
    SD.Printf(FRMT, a);     LCD.Write(a); LCD.Write(" ");
    SD.Printf("\r\n");      LCD.Write('\n');
}
void doc(const char *text, float a, float b){
    SD.Printf(FRMT, TimeNow()-runBeginTime);
    SD.Printf(text);        LCD.Write(text); LCD.Write(" ");
    SD.Printf(FRMT, a);     LCD.Write(a); LCD.Write(" ");
    SD.Printf(FRMT, b);     LCD.Write(b); LCD.Write(" ");
    SD.Printf("\r\n");      LCD.Write('\n');
}
void doc(const char *text, float a, float b, float c){
    SD.Printf(FRMT, TimeNow()-runBeginTime);
    SD.Printf(text);        LCD.Write(text); LCD.Write(" ");
    SD.Printf(FRMT, a);     LCD.Write(a); LCD.Write(" ");
    SD.Printf(FRMT, b);     LCD.Write(b); LCD.Write(" ");
    SD.Printf(FRMT, c);     LCD.Write(c); LCD.Write(" ");
    SD.Printf("\r\n");      LCD.Write('\n'); LCD.Write(" ");
}
void doc(const char *text, float a, float b, float c, float d){
    SD.Printf(FRMT, TimeNow()-runBeginTime);
    SD.Printf(text);        LCD.Write(text); LCD.Write(" ");
    SD.Printf(FRMT, a);     LCD.Write(a); LCD.Write(" ");
    SD.Printf(FRMT, b);     LCD.Write(b); LCD.Write(" ");
    SD.Printf(FRMT, c);     LCD.Write(c); LCD.Write(" ");
    SD.Printf(FRMT, d);     LCD.Write(d); LCD.Write(" ");
    SD.Printf("\r\n");      LCD.Write('\n');
}
```

APPENDIX C

Example Testing Log

```
14.76 Voltage: 10.93
21.50 CdS baseline:2.27
21.51 Red CdS baseline:2.65
21.52 RPS: .10 -.89 360.00
21.53 Touch to dominate.
23.59 Waiting for CdS.
26.01 Going with CdS.1.79 1.81
26.39 Halting
26.40 blindPosition-.00 -8.00
26.44 moveTo8.75 -8.50 1.00
26.93 Halting
26.93 blindPosition8.74 -8.50
27.74 RPS: 6.79 -8.39 2.30
27.75 moveTo8.75 -8.50 1.00
28.13 Halting
28.14 blindPosition8.75 -8.50
28.95 RPS: 10.40 -8.60 3.40
28.99 moveTo8.75 -8.50 1.00
29.32 Halting
29.33 blindPosition8.75 -8.50
30.14 RPS: 8.90 -8.60 3.69
30.15 moveTo8.75 -8.50 1.00
30.27 Bright/red/color:.76 .49 2.00
30.43 RPS Not Updated
30.53 RPS Not Updated
30.64 RPS: 8.90 -8.60 3.50
30.75 RPS: 8.90 -8.60 3.59
30.86 RPS Not Updated
30.97 RPS Not Updated
31.07 RPS: 9.10 -8.70 2.69
31.18 RPS: 9.79 -9.49 3.30
31.29 RPS: 10.29 -9.89 .59
31.41 RPS: 10.60 -9.89 358.30
31.45 Halting
31.62 Halting
31.63 blindPosition10.60 -6.89
31.74 RPS: 10.69 -9.79 357.69
31.86 RPS Not Updated
31.96 RPS: 10.69 -9.79 357.59
32.08 RPS: 10.69 -9.79 357.80
32.19 RPS Not Updated
32.30 RPS: 10.69 -9.79 357.59
32.44 RPS: 11.19 -9.29 357.19
32.56 RPS: 11.19 -9.39 357.30
32.67 RPS: 10.50 -9.79 .19
32.78 RPS: 10.19 -9.79 1.09
32.89 RPS: 10.19 -9.79 .90
33.01 RPS: 10.19 -9.79 .69
33.12 RPS: 10.29 -9.79 .69
33.26 RPS: 10.19 -9.79 1.09
33.38 RPS: 10.19 -9.79 .69
33.49 RPS: 10.00 -9.79 .80
33.60 RPS: 9.90 -9.79 .59
33.71 RPS: 9.90 -9.79 .19
33.82 RPS: 10.00 -9.79 .30
33.94 RPS: 9.90 -9.79 .40
34.09 RPS: 9.90 -9.79 .09
34.20 RPS: 10.00 -9.79 .00
34.31 RPS: 10.10 -9.79 359.30
34.42 RPS Not Updated
34.53 RPS: 10.10 -9.79 358.59
34.64 RPS: 10.10 -9.79 359.19
34.75 RPS: 10.10 -9.89 359.09
34.87 RPS: 10.00 -9.79 .59
35.01 RPS: 10.00 -9.79 359.90
35.13 RPS: 10.00 -9.79 359.40
35.24 RPS: 10.00 -9.79 359.09
35.35 RPS: 10.00 -9.79 359.59
35.46 RPS: 10.00 -9.79 360.00
35.58 RPS: 10.00 -9.79 .00
35.69 RPS: 9.90 -9.79 1.00
35.83 RPS: 9.90 -9.79 .80
35.94 RPS: 9.90 -9.79 .40
36.06 RPS: 9.79 -9.79 .69
36.17 RPS: 9.69 -9.79 1.30
36.28 RPS Not Updated
36.39 RPS: 9.79 -9.79 .80
36.50 RPS: 9.90 -9.79 .19
36.61 RPS: 9.90 -9.79 .40
36.73 RPS: 9.60 -9.79 1.80
36.87 RPS: 9.29 -9.70 2.09
36.98 RPS: 9.10 -9.89 2.00
37.10 RPS: 9.29 -9.79 1.19
37.21 RPS: 9.29 -9.79 1.50
37.32 RPS: 9.50 -9.79 1.09
37.43 RPS: 9.40 -9.79 1.90
37.54 RPS: 9.40 -9.79 2.59
37.69 RPS: 9.40 -9.79 1.19
37.70 Halting
37.87 Halting
37.87 blindPosition9.40 -6.79
37.99 Halting
38.00 blindPosition9.40 -4.79
38.43 Halting
38.44 blindPosition1.40 -4.79
38.45 moveTo-5.50 -12.00 2.00
39.21 Halting
39.22 blindPosition-5.49 -12.00
40.03 Position Update Failed: -18.39 -
30.29 -1.00
40.05 moveTo-5.50 -12.00 2.00
40.06 moveTo-14.00 -22.10 1.00
41.02 Halting
41.03 blindPosition-13.99 -22.10
41.84 RPS: -11.99 -21.79 8.30
41.89 moveTo-14.00 -22.10 1.00
42.25 Halting
42.26 blindPosition-14.00 -22.10
43.07 RPS: -13.20 -21.70 10.40
43.08 moveTo-14.00 -22.10 1.00
```

```
43.20 RPS: -13.20 -21.60 10.50
43.31 RPS: -13.20 -21.60 10.40
43.46 RPS Not Updated
43.57 RPS: -13.20 -21.60 10.50
43.68 RPS: -13.20 -21.70 10.50
43.79 RPS: -12.30 -21.49 11.19
43.91 RPS: -11.30 -21.49 11.59
44.02 RPS: -9.20 -21.70 12.30
44.13 RPS: -8.20 -21.70 11.19
44.14 Halting
44.39 Halting
44.43 blindPosition-12.20 -21.70
44.45 moveTo-5.50 -11.69 .50
45.33 Halting
45.33 blindPosition-5.50 -11.69
46.14 RPS: -5.89 -16.60 334.19
46.15 moveTo-5.50 -11.69 .50
47.25 Halting
47.25 blindPosition-5.50 -11.69
48.10 RPS: -5.70 -12.60 332.59
48.11 moveTo-5.50 -11.69 .50
48.32 Halting
48.33 blindPosition-5.50 -11.69
49.14 RPS: -5.59 -11.89 332.19
49.15 moveTo-5.50 -11.69 .50
50.24 RPS: -5.59 -11.89 332.09
50.29 Rot from/to/by:332.09 .00 27.90
50.69 Halting
50.70 Rotation Finished.
51.51 RPS: -5.59 -11.99 .19
52.32 moveTo-9.00 -11.69 .50
52.96 Halting
52.97 blindPosition-9.00 -11.69
53.78 RPS: -8.59 -11.70 1.09
53.79 moveTo-9.00 -11.69 .50
55.08 Halting
55.08 blindPosition-8.59 -8.70
55.49 Halting
55.50 blindPosition-12.00 -4.50
56.13 Halting
56.14 blindPosition-12.00 7.49
56.15 RPS: -11.39 -5.99 3.90
56.16 moveTo-13.00 15.00 4.00
57.29 Halting
57.30 blindPosition-13.00 14.99
58.11 RPS: -13.59 12.10 6.80
58.12 moveTo-13.00 15.00 4.00
58.13 moveTo.00 27.50 1.00
59.62 Halting
59.63 blindPosition-.00 27.50
60.44 Position Update Failed: -18.39 -
      30.29 -1.00
60.49 moveTo.00 27.50 1.00
60.51 Position Update Failed: -18.39 -
      30.29 -1.00
60.52 Rot from/to/by:6.80 -45.00 -51.80
61.27 Halting
61.27 Rotation Finished.
62.08 RPS: -2.30 29.39 321.09
62.09 Rot from/to/by:321.09 -45.00 -6.09
62.20 Halting
62.20 Rotation Finished.
63.01 RPS: -2.30 29.39 315.90
63.06 moveTo-10.25 36.00 1.00
63.64 Halting
63.64 blindPosition-10.25 36.00
64.45 Position Update Failed: -18.39 -
30.29 -1.00
64.47 moveTo-10.25 36.00 1.00
66.29 Halting
66.29 blindPosition-7.25 33.00
66.31 moveTo.00 27.50 2.00
66.87 Halting
66.88 blindPosition.00 27.50
67.69 Position Update Failed: -18.39 -
      30.29 -1.00
67.71 moveTo.00 27.50 2.00
67.72 Position Update Failed: -18.39 -
      30.29 -1.00
67.74 moveTo6.40 33.29 .50
68.22 Halting
68.23 blindPosition6.39 33.29
69.07 RPS: 8.29 34.00 324.09
69.09 moveTo6.40 33.29 .50
69.58 Halting
69.58 blindPosition6.40 33.29
70.39 RPS: 6.60 33.79 324.90
70.41 moveTo6.40 33.29 .50
70.53 Halting
70.54 blindPosition6.40 33.29
71.38 RPS: 6.60 33.39 324.30
71.40 moveTo6.40 33.29 .50
71.41 moveTo7.90 34.79 .50
71.78 Halting
71.79 blindPosition7.90 34.79
72.60 RPS: 7.40 34.10 323.09
72.61 moveTo7.90 34.79 .50
72.81 Halting
72.82 blindPosition7.90 34.79
73.63 RPS: 7.90 34.29 320.69
73.64 moveTo7.90 34.79 .50
74.93 Halting
74.93 blindPosition4.90 31.29
75.74 moveTo.00 27.50 2.00
76.14 Halting
76.14 blindPosition-.00 27.50
76.99 RPS: 1.40 29.10 323.80
77.00 moveTo.00 27.50 2.00
77.40 Halting
77.41 blindPosition-.00 27.50
```

```
78.22 RPS: -.49 27.39 322.69
78.23 moveTo.00 27.50 2.00
78.25 moveTo12.00 14.00 2.00
79.37 Halting
79.38 blindPosition12.00 14.00
80.19 Position Update Failed: -18.39 -
      30.29 -1.00
80.21 moveTo12.00 14.00 2.00
80.22 Position Update Failed: -18.39 -
      30.29 -1.00
80.24 Rot from/to/by:322.69 90.00 127.30
81.44 Halting
81.45 Rotation Finished.
82.25 RPS: 11.60 14.10 99.40
82.30 Rot from/to/by:99.40 90.00 -9.40
82.45 Halting
82.46 Rotation Finished.
83.26 RPS: 11.50 14.20 88.00
83.27 moveTo12.00 -5.00 1.00
84.32 Halting
84.33 blindPosition12.00 -4.99
85.14 Position Update Failed: -18.39 -
      30.29 -1.00
85.20 moveTo12.00 -5.00 1.00
85.21 moveTo.00 -5.00 1.00
85.86 Halting
85.86 blindPosition.00 -5.00
86.67 Position Update Failed: -18.39 -
      30.29 -1.00
86.69 moveTo.00 -5.00 1.00
86.70 moveTo.00 9.00 1.00
87.45 Halting
87.46 blindPosition
```

APPENDIX D

Drivetrain Calculations

Part I: Speed

Estimated driving distance based on Cyan path in Figure 8 on page 15: 188.75 in.

Maximum allowable driving time assuming 20 seconds of actions: 120 s – 20 s = 100 s

$$Linear\ speed = \frac{Driving\ Distance}{Driving\ Time} = \frac{(188.75\ in)}{120s-20s} = 1.89\ in/s$$

Wheel radius for omni wheels: 1.185 in.

$$Rotational\ Speed = \frac{Linear\ Speed}{Radius} = \frac{(1.89\ in/s)(60\ s/min)}{(1.185\ inches\ per\ radius)(2\pi\ radii\ per\ revolution)} = 15.2\ rpm$$

Part II: Torque

Estimate of total weight: 29.8 oz

Ramp Dimensions: 9 inches run by 3 inches rise

Ramp Angle of elevation = arctan(3/9)=18°

Max Force:

Force of motors = weight * sin(18) + estimate for force of friction

= (29.8 oz)(0.309) + (8 oz)

= 17.2 oz (or 4.78 N)

Max Torque:

$$\tau = r \times F = (1.185\ in)(17.2\ oz) = 20.38 \text{ ounce-inches of torque}$$

Max Torque per motor:

Torque per wheel = 20.38 /4 = 5.09 ounce-inches per wheel