IEEE SoutheastCon 2017 Hardware Competition

Final Report

Brian Roskuszka, Cameron McSweeney, Daniel Hofstetter, Kendall Knapp

Advised by Dr. Jing Wang, Dr. Yufeng Lu, and Dr. In Soo Ahn

2 May 2017

Abstract

The 2017 IEEE SoutheastCon Student Hardware Competition requires participants to build a robot that autonomously navigates through a Star Wars themed arena. The robot dimensions are constrained by a 12-inch cube. It must complete four tasks within four minutes.

In Task 1, the robot must approach a stage with six copper pads on it. Five of the pads are connected to five different electronic components and the sixth is common to all of the components. The robot must determine the position of each of the components. In Task 2, the robot must detect the magnetic field present in the arena when it is turned on. The robot must strike a lightsaber when the magnetic field is on. In Task 3, the robot must turn a knob in alternating directions governed by the component position code determined in Task 1. In Task 4, the robot must deposit up to three Nerf darts into a hole on the far side of the arena.

Based on those competition guidelines, the robot was constructed with four modules to perform four tasks. Two microcontrollers were programmed for the basic functions of robot navigation, components measurements, magnetic field detection, servo motor control for the gripper and swing arm, and dart firing. The robot is equipped with infrared sensors, ultrasonic distance sensors, contact sensors, and wheel encoders for navigation. Each module was implemented individually and all were integrated on a robotic platform. The robot was thoroughly tested to ensure it would meet the requirements of the IEEE SoutheastCon Hardware Challenge.

Acknowledgements

The team's advisors, Dr. Jing Wang, Dr. Yufeng Lu, and Dr. In Soo Ahn, supported the team, providing valuable suggestions and keeping the project on task. Several other Bradley University faculty also assisted with the robot's development.

A special thanks to Nick Schmidt, the assistant lab director, for providing the idea and first prototype for a spring loaded dart launching mechanism that was later used to design the final mechanism that the robot had in competition.

A special thanks also to Christopher Mattus, the lab director, for providing the team with access to a large collection of motors, breadboards, H-bridges, and testing materials that were used in the process of making the final robot.

None of this would have been possible without the help of Terry McCowan, the engineering shop director. He helped the team make the physical chassis for the robot and provided invaluable insights into how to design and fabricate such a modular robot.

Table of Contents

Abstract **1**

Acknowledgements **2**

Table of Contents **3**

1. Introduction **6**

1.1 Problem Statement 6

1.2 Motivation 6

1.3 Review of Literature and Prior Work 7

2. Problem Formulation **8**

2.1 Task Specifications 8

2.2 Project Goals 10

3. System Design and Analysis **11**

3.1 System Level Inputs 11

3.2 System Level Outputs 11

3.3 Modes of Operation 12

3.4 System Level Block Diagram 12

3.5 Navigation Subsystem 13

3.5.1 Design Considerations 13

3.5.2 Navigation Algorithm 14

3.5.3 Motor Encoders 15

3.5.4 Infrared Sensors 15

3.5.5 Ultrasonic Distance Sensors 16

3.5.6 Touch Sensors 17

3.5.7 Navigation Flowchart 18

3.5.8 Results 20

3.6 Component Identification - Task 1 Subsystem 20

3.6.1 Preliminary Work 20

3.6.2 Actions Performed 21

3.7 Lightsaber Duel - Task 2 Subsystem 25

3.7.1 Overview of Task 2 and Problem Analysis 25

3.7.2 Magnetic Field Detection 26

3.7.3 Lightsaber Actuator 29

3.7.4 System Operation 30

3.7.5 Testing and Results 30

3.8 Lock Turning - Task 3 Subsystem 31

3.8.1 Attaching to Task 3 31
3.8.2 Rotating Task 3 33
3.8.3 System Functionality 34
3.9 Dart Firing - Task 4 Subsystem 38

4. System Integration, Testing and Evaluation 40
4.1 System Integration - Hardware 40
4.1.1 First Robot Chassis 40
4.1.2 Final Robot Chassis 40
4.2 System Integration - Software 44
4.3 Competition Results 47
4.3.1 Competition Overview 47
4.3.2 Robot Functionality 47
4.3.3 Changes Made During the Competition 47
4.3.4 Results 49

5. Division of Labor 50

6. Possible Improvements 51

7. Conclusion 52

8. References 53

A1. Appendix A - Parts List 54

A2. Appendix B - Navigation and System Control Code 56
A2.1 Main Navigation 56
A2.2 Motor Control 65
A2.3 I2C Library 67
A2.4 Motor Encoder Library 68
A2.5 Ultrasonic Sensors - Timers 70
A2.6 Ultrasonic Sensors - External Interrupts 72
A2.7 IR Sensors - ADC 74
A2.8 IR Sensors - Line Adjustment 75
A2.9 Task 1 Status Signal 78

A3. Appendix C - Task 1 Code 79
A3.1 Task 1 Control 79
A3.2 ADC Library 82
A3.3 Component ID Algorithm 84

A4. Appendix D - Task 2 Code 89
A4.1 Task 2 Algorithm 89

A4.2 Magnetic Sensor Driver 93
A4.3 Servo Control 96

A5. Appendix E - Task 3 Code 98
A5.1 Task 3 Control 98
A5.2 Servo Control 99

A6. Appendix F - Competition Rules and Specifications 101

1. Introduction

Every year, the IEEE Southeast Conference hosts a student hardware challenge, where students compete by designing a robot to perform a set of predefined tasks. The Bradley University team designed a robot for the competition and travelled to the IEEE SoutheastCon 2017 to compete. During the project, the team gained valuable real-world experience and demonstrated a viable design for an autonomous robot meeting the specifications of the competition.

1.1 Problem Statement

Rules are provided that outline the specifications for the competition [1]. The robot attempts to complete four tasks within a four minute time limit. It receives inputs for location, navigation, and other task-specific applications, uses those inputs to make decisions, and then moves itself around the arena and performs task-specific actions. The system must be completely autonomous once the start button is pressed. In addition, the robot must fit within a 12 inch cube until the competition begins. After that, it can expand to any size as long as it does not extend beyond the arena walls by more than 3 inches. The set-up for the competition is shown in Figure 1. There are four identical arenas.

Figure 1: Picture of the Competition Area

1.2 Motivation

The competition is a complex challenge involving the integration of many components. The autonomous requirement further adds to the difficulty. A successful robot requires efficient hardware construction and software design. Because teams are given specifications and asked to solve the problem in the best way possible, the competition simulates a real-world design

process. The team's experience provides insights into designing an autonomous robot that will be useful for other design work.

1.3 Review of Literature and Prior Work

Only the official competition rules and specifications provided by the IEEE SoutheastCon officials were consulted for literature specifically related to this project. From this, the rules and requirements were obtained and designs for the robot were created to accommodate those rules.

Other literature was consulted to learn how particular components of the robot worked to interface those components with the central microcontroller; datasheets were primarily consulted for these components.

As this competition was designed specifically for the upcoming IEEE SoutheastCon Conference, no prior work is useful to consult regarding this project.

2. Problem Formulation

The robot's design is determined by the competition rules. An outline of the problems the robot needs to solve is necessary before work can begin. The four tasks, described in detail below, are the primary consideration.

2.1 Task Specifications

In Task 1, there are six copper pads arranged such that five pads form a pentagon surrounding a sixth, central pad. The arrangement is shown in Figure 2. Each of the five outer pads are connected to exactly one of the following components: a wire, a resistor, a capacitor, an inductor, and a diode. One end of each component is attached to the center pad, which acts as a common reference point for all the components. The robot identifies the location of each component and saves the location information for Task 3.

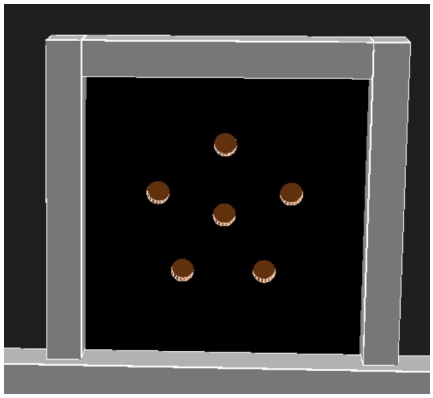

Figure 2: Rendering of Arena Task 1

For Task 2, there is a rod sticking up above the arena representing a light saber. A magnetic field is randomly generated five times for two-second intervals within a thirty-second time frame. The robot must hit the rod with a robotic arm while the magnetic field is on, and is penalized for hitting the rod when no field is generated. Figure 3 displays the rendering for Task 2.

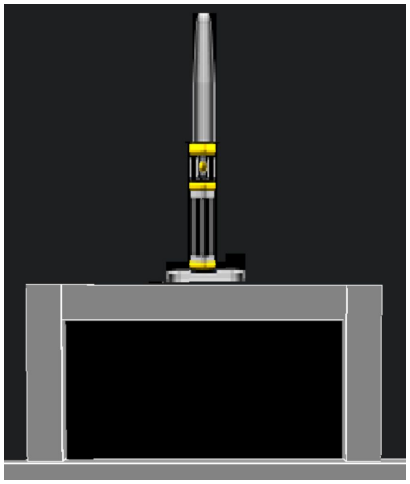

Figure 3: Rendering of Arena Task 2

At Task 3, the robot recalls the order of the components from Task 1. Each component has a number associated with it (1-5), and the robot turns a knob, at the center of Figure 4, multiples of 360° in the same order the components were discovered in Task 1. For example, if the code were 3-2-5-1-4, the robot turns the knob 3 revolutions clockwise, then 2 revolutions counter clockwise, then 5 revolution clockwise, then 1 revolution counter-clockwise, followed by 4 final clockwise revolutions.

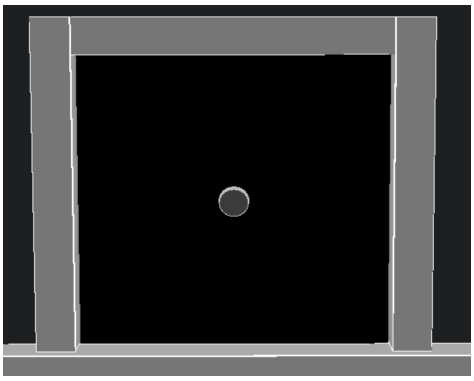

Figure 4: Rendering of the Arena Task 3

To finish up the competition, Stage 4 requires the robot to attempt to get up to 3 nerf darts into a box at the far end of the arena. The robot may shoot the nerf darts or drive to the box and simply drop the darts in, but the arena has increasingly tall steps leading to the box, encouraging competitors to shoot the darts from below. Figure 5 shows the opening in the arena wall.

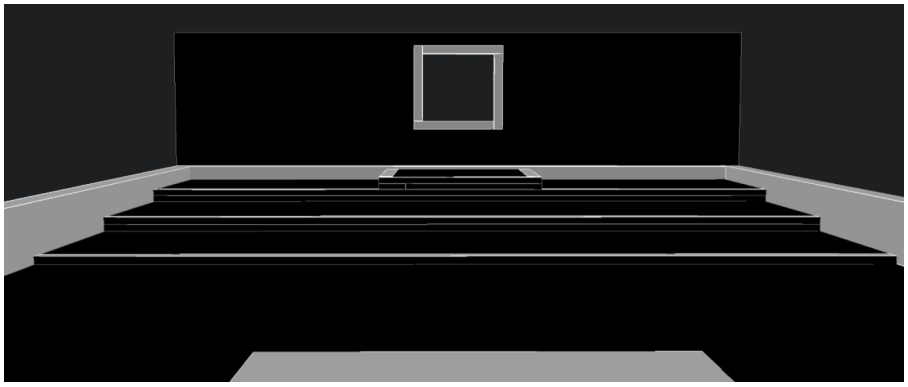

Figure 5: Rendering of the Arena Task 4

2.2 Project Goals

The primary goal of the project is to build a system that will perform well at the IEEE SoutheastCon Hardware Challenge. The team wanted to represent Bradley University at an IEEE event. Furthermore, it will apply knowledge gained from classroom theory in a simulation of the real world. The design process will enhance the team's understanding of autonomous robotics and assist future endeavors in this exciting field.

3. System Design and Analysis

The next step in designing the robot is to identify the inputs and outputs to the system. From these, an appropriate division of the subsystems can be developed.

3.1 System Level Inputs

Power: The robot has a simple on-off power switch. In the off position, no power is provided to the robot and the circuit board. In the on position, power is supplied.

Distance from Walls: The robot has distance sensors located on the front edge of the robot to sense the distance between the robot and the walls.

Contact with Walls: Touch sensors signal when the robot has made contact with the arena wall.

Position on Line: IR sensors receive data about the location of the navigation strip leading to Task 1.

Component Voltage: For Task 1, a 5V step is applied to a circuit in which a resistor and the component are in series. The transient voltage response of the component is measured and the pattern of the response determines which component is being tested. The analog to digital converter is implemented to read these voltages.

Magnetic Field: The robot senses a magnetic field and uses the readings to complete Task 2.

3.2 System Level Outputs

Move Robot: The robot moves around the arena to each stage.

Move Robotic Arms: The arms of the robot need to be moved slightly to correctly complete each task, even when the robot is correctly located in front of each task.

Component ID Step Output: In Task 1 the robot excites the components to observe their voltage characteristics and identify each component. This is a simple 5V step output.

Numeric Code from Task 1: The robot displays the order of the components identified in Task 1 to gain points for correctly identifying the components in the event that Task 3 is not completed correctly.

Hit Lightsaber: The robot uses an arm to hit the stand-in lightsaber on the arena at the prescribed time.

Turn Knob: Upon receiving a signal of how many turns to execute, a clamp rotates the knob the correct number of turns before reversing direction to execute the next number of turns.

Shoot Nerf N-Strike Dart: To complete Task 4, three Nerf N-Strike darts must be shot into a hole at the far end of the arena.

3.3 Modes of Operation

Operating: After powering on, the robot executes the functions of navigation and each of the four stages autonomously.

Mission Complete: The robot enters this mode upon completion of all the tasks and simply stays idle with no movement. This allows the power switch to be flipped and robot to be picked up safely.

3.4 System Level Block Diagram

The system block diagram is given in Figure 6. It shows the inputs going into the system from the outside environment and the system's outputs. The interactions between the individual subsystems is also shown.

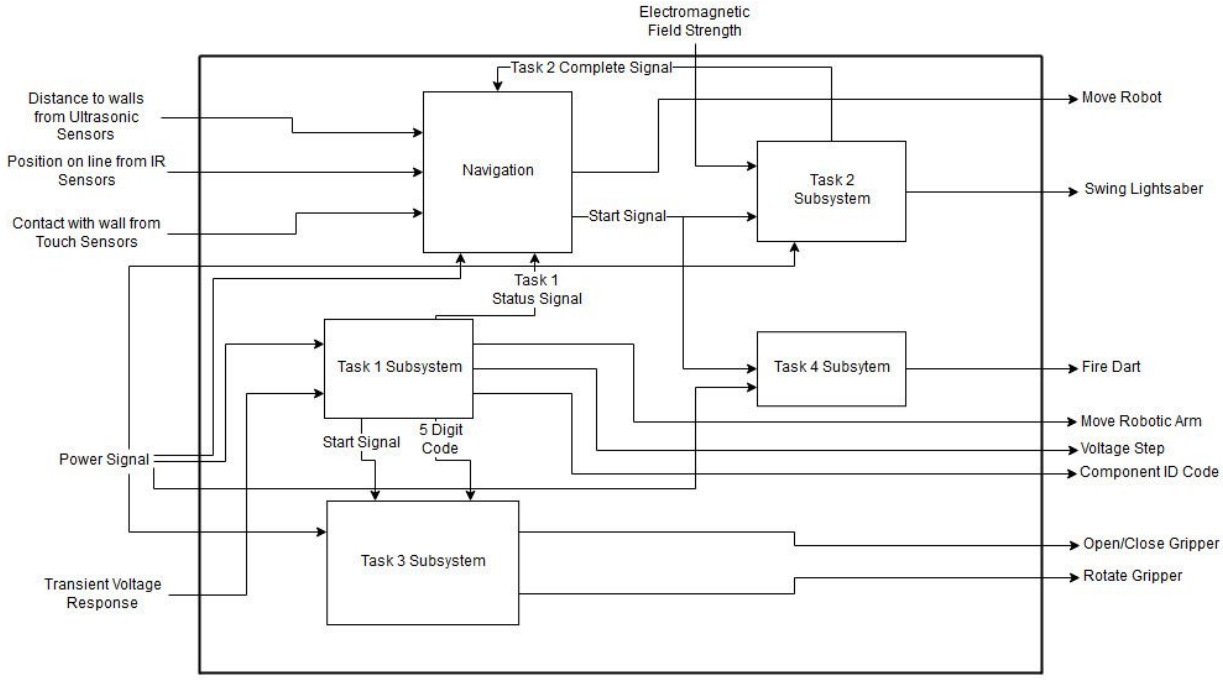

Figure 6. System Level Block Diagram

3.5 Navigation Subsystem

The navigation subsystem is in charge of accurately moving the robot to each task. The subsystem keeps an internal record of where the robot should be located in the arena. Rotary encoders are the primary method of navigation, allowing the robot to move a set distance. However, wheel slippage introduces error, which requires external sensors to provide feedback and increase accuracy. Infrared sensors located beneath the robot detect the white line leading from the starting square. Two ultrasonic sensors on the front of the robot measure the distance to the arena wall. Finally, two bumpers send a signal whenever the robot touches the wall. The inputs and outputs are shown in Figure 7 below.

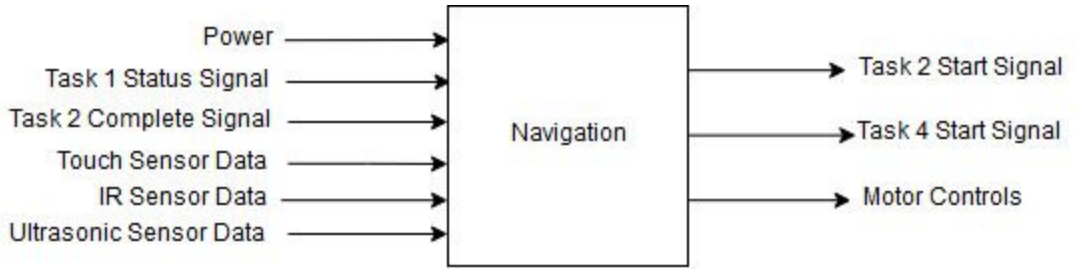

Figure 7. Navigation Subsystem Block Diagram

3.5.1 Design Considerations

The primary concern with the navigation subsystem is accuracy. Each task has different accuracy requirements. Determining these requirements, shown in Table 1, was the first step in developing the system. The values given are the maximum amount of horizontal offset from the center of the stage that can be tolerated.

Table 1. Navigation Specifications

Task	Accuracy Requirement
1	0.25 in
2	~ 2 in
3	< 0.2 in
4	3 in

Tasks 1 and 3 are the most demanding. Task 3 is especially difficult because the gripper needs to be as centered as possible. A slight variation will cause it to lock and fail to turn the knob because it does not have the necessary torque. In contrast, tasks 2 and 4 have low accuracy requirements. As a result, navigation is most focused on improving the accuracy of the first two tasks.

An additional focus of the navigation subsystem is precision. It needs to deliver reliable results. If a certain task is always off by the same amount, hardware can be adjusted to make the system accurate. This was the method used during testing.

In addition to horizontal accuracy, the navigation needs to adjust the robot's angle. Ideally, the robot should always be perpendicular to the arena wall. Any angle will affect the horizontal accuracy. For example, if the robot is within 3" of center for task 4, but is angled to one side, the darts will still miss. Sensors were added to attempt to remedy this issue.

3.5.2 Navigation Algorithm

The navigation process is designed to be as simple as possible, utilizing a known starting position to maximum advantage and making the least possible number of turns. This strategy minimizes error and reduces the complexity of sensor feedback. The robot is placed facing Task 1. It drives forward, performs Task 1, then drives backward to Task 3. Next it drives forward again making a 90° turn to Task 2. Finally, it drives backwards to the bottom of the steps and fires the darts for Task 4. Figure 8 displays the navigation sequence. The entire process only requires one turn. The primary drawback is that tasks 1 and 2 need to be on the front of the robot, while tasks 3 and 4 are on the back. The team was able to overcome the space difficulties and take advantage of a simple navigation system.

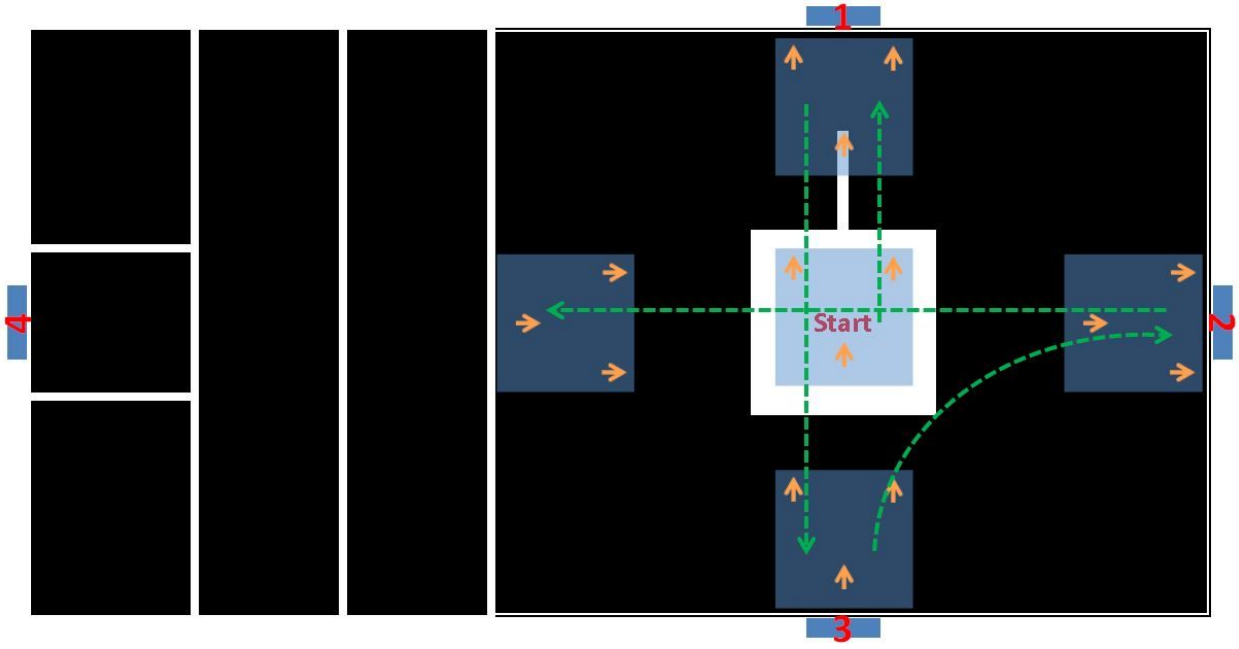

Figure 8. Navigation Path

3.5.3 Motor Encoders

Two Vex Robotics integrated motor encoders are used to provide feedback on the distance and direction travelled. They have a resolution of 627.2 ticks per revolution. Determining the revolution in inches is essential to knowing how accurate the encoders are. Equation 1 gives the calculation. The wheel diameter is 4in.

$$\frac{4\pi\ in}{627.2\ ticks} = 0.02\ in\ /\ tick \quad (1)$$

A 0.02in resolution is more than enough. Unfortunately, the wheels are not able to rotate with the same accuracy. The wheels can move several degrees without turning the motor shaft, due to the shape of the shaft. This property could not be changed and significantly degrades the accuracy of navigation.

An additional issue occurs due to the robot's momentum. After the navigation system sends a stop signal, the robot keeps moving for a fraction of a second. As a result, the stop signal needs to be sent before the encoders record a full revolution. Through empirical testing, it was found that the final revolution needs 535 ticks, a 14.7% decrease. This amount will vary based on the speed of the robot. Consequently, the necessary encoder value to move the robot a certain distance can never be calculated precisely; it must be fine-tuned using an iterative approach.

3.5.4 Infrared Sensors

Three VEX Robotics IR sensors are located on the bottom of the robot. They are spaced evenly with the leftmost and rightmost sensors being approximately 1in apart. The spacing is fixed by pre-made holes in the Vex Robotics chassis. The white line they are trying to detect also has 1" width, meaning the left and right sensors should be on the edge between white and black, while the center sensor should always see white.

The IR sensors send an analog signal whose amplitude is determined by the reflectivity of the surface. This signal is read using a 10-bit ADC. Table 2 shows the thresholds used to determine the color of the surface below the IR sensor. The values were determined empirically.

Table 2. IR sensor thresholds

Color	Range
White	0-99
Edge	100-799
Black	800-1024

In order for the IR sensors to have useful accuracy, they must be no more than 0.5in from the surface. 0.125in is optimal. The robot's sensors were 0.25in above the arena, which allowed for the full range of values to be used.

3.5.5 Ultrasonic Distance Sensors

Two Kuman HC-SR04 ultrasonic distance sensors are mounted on the front of the robot. They send out an output pulse and wait for the return echo. The distance to the object is calculated based on the time the echo takes to arrive. Equation 2 shows how to derive the distance *d* to the nearest object given a wait time *t*.

$$d = \frac{v_{sound} * t}{2} \tag{2}$$

Each sensor uses an external interrupt to determine when the echo returns. Instead of using a separate timer for each sensor, one 16-bit timer measured the time for both sensors. An overflow interrupt detected rollover since the last measurement. Figure 9, from the datasheet, shows the input and output signals [2].

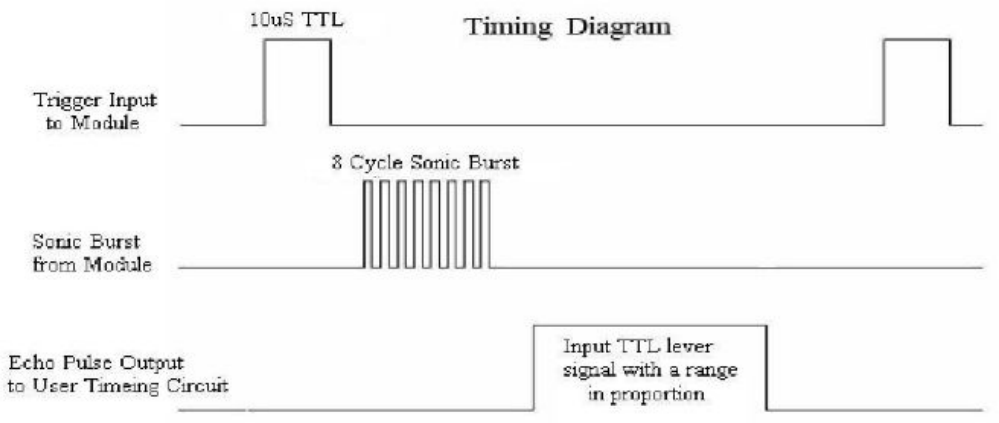

Figure 9. Ultrasonic sensor timing diagram

The code below shows how one sensor is operated using an external interrupt. The full code is in Appendix B, section A2.6.

```
ISR(INT0_vect)
{
        PORTB ^=0x10;
        if (change_to_falling_INT0==1)
        {
        EICRA |=  1<<ISC01;      //falling edge
        EICRA &= ~(1<<ISC00);
        change_to_falling_INT0=0;
        Time0_old = TCNT1;
        Time0_overflow_old = getOverflow();
        }
```

```
        else
        {
        if (Time0_overflow_old != getOverflow())
        {
                Time0 = 65536 - Time0_old + TCNT1;
        }
        else
        {
                Time0 = TCNT1 - Time0_old;
        }
        d0 = Time0/1.85;            //.1 in
        EICRA |= 1<<ISC01 | 1<<ISC00;             //rising edge
        change_to_falling_INT0 = 1;
        }
}
```

The original intent of including two distance sensors was to measure the angle of the robot relative to the wall. The two sensors are mounted 6in apart. It was thought that the difference in distance measurements between the two sensors would give information about the robot's angle. A difference of zero means that the robot is perpendicular to the wall. Unfortunately, this method did not work as planned. A 0.1in resolution is not good enough when the difference created by a slight angle is negligible. Large angle changes could be detected easily, but the system was unreliable at small angles, which are the most important.

The line-following algorithm was developed due to the failure of angle measurement. The distance sensors were still useful in measuring the distance of the robot from the wall, helping to correct errors caused by the encoders and wheel slippage.

3.5.6 Touch Sensors

Two VEX Robotics bumpers are located on the front of the robot. They act as simple buttons, signalling logic low when they are pressed. They allow the robot to determine with certainty when it has reached the wall. Their primary function is to force the robot to be perpendicular with the wall. To do this, the robot needs more force, so the distance sensors signal when the robot is very close to the wall. When this happens, the robot speeds up and the touch sensors are activated. The program waits until at least one bumper is pressed before proceeding to the next step. In Figure 10, the bumpers are located beneath the ultrasonic sensors, on the front of the robot.

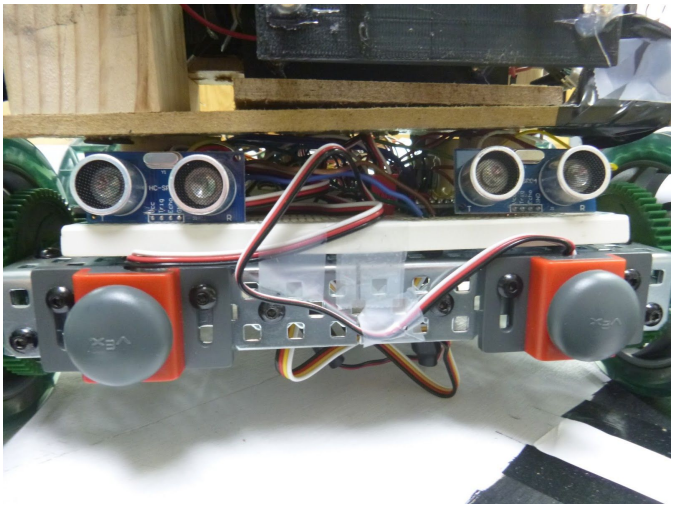

Figure 10. Mounted ultrasonic and touch sensors

3.5.7 Navigation Flowchart

The navigation program is complex. Not only does it move the robot, it also sends start signals to tasks 2 and 4, and receives a status signal from tasks 1 and 2. The control process is given in Figure 11.

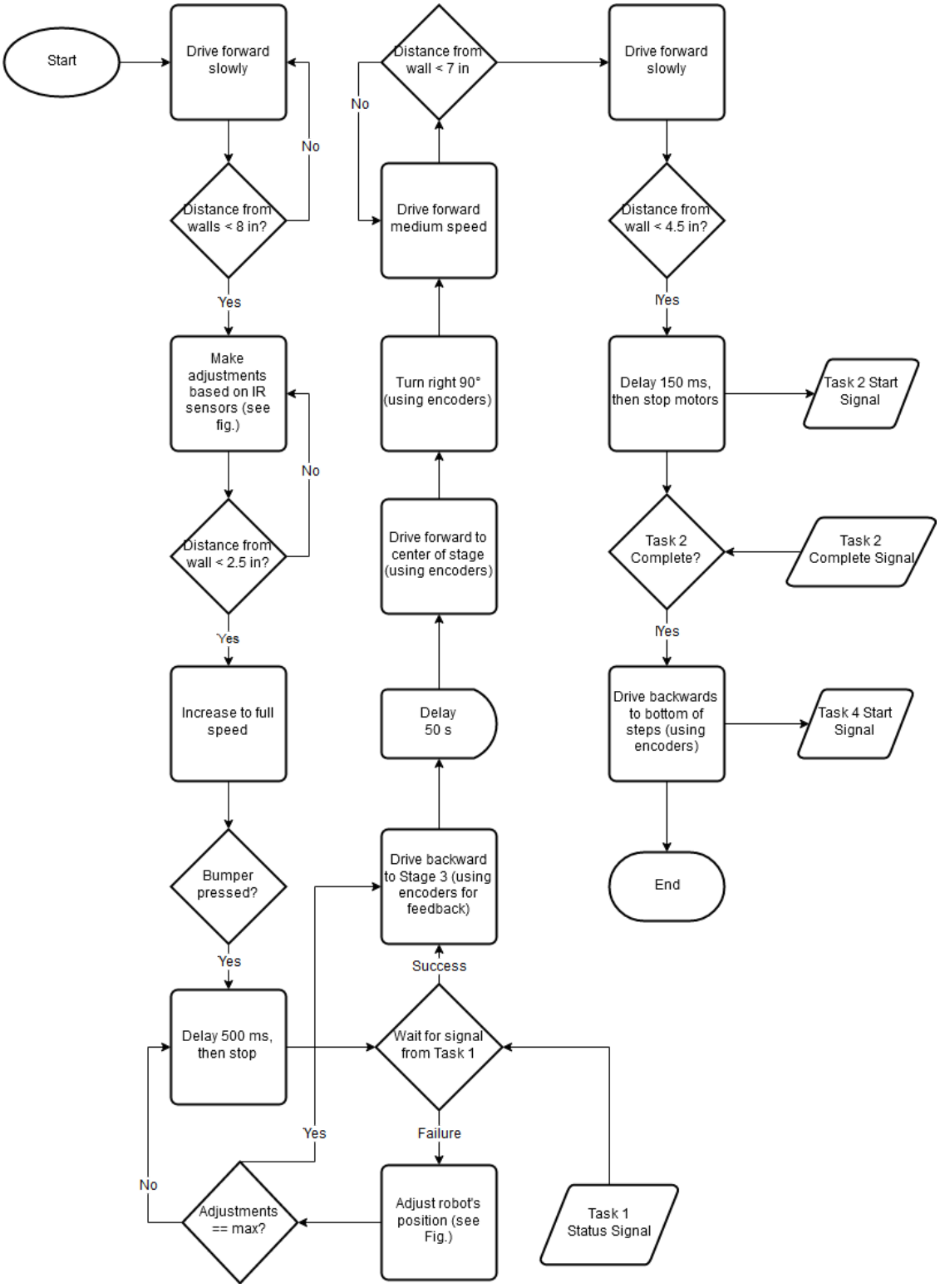

Figure 11. Navigation Flowchart. See Appendix B, Section A2.1 for relevant code

3.5.8 Results

At the competition, navigation worked well. It could successfully drive to all four tasks. Three of the tasks were aligned consistently, while Task 3 caused problems. Sometimes it was close enough for the gripper to attach to the knob, but offset enough that the gripper did not have enough torque to rotate it. Other times, navigation missed by several inches. Due to these problems, Task 3 never got any points.

The primary cause of this inaccuracy is the wheel hardware. The wheels have significant backlash. When the wheels stop, there is uncertainty as to where the robot will come to rest. In addition, the robot had no way of correcting a poor starting angle after finishing with Task 1. Making additional adjustments to align with Task 1 seemed to greatly decrease Task 3 accuracy. However, the ultrasonic distance sensors corrected the robot and allowed proper alignment with Task 2.

Navigation could be improved by changing the chassis design. Wheels providing side-to-side movement would have allowed accurate small adjustments to be made, particularly to align properly with Task 1. Also, a new method for angle measurement needs to be developed.

3.6 Component Identification - Task 1 Subsystem

The sections below describe how Task 1: Bring Down the Shields was implemented on the robot.

3.6.1 Preliminary Work

The first goal regarding Task 1 was to correctly differentiate the components. There are different methods to identify each of the components, from measuring DC resistance to AC impedance to changing voltage polarity, the latter primarily to identify the diode. The first idea that was examined involved testing each component’s voltage response to a step input; that is, how would each component react to a sudden increase in voltage? The voltage reaction would be measured between a resistor and the component, where the resistor is in series with the power source and the component. The resistor, herein referred to as the protection resistor, would prevent a short circuit when the wire connected to the voltage source, as well as attempt to force the capacitor and inductor to have similar time constants

In general, the components could be expected to react in specific ways: the wire would have a constant voltage of 0V, because the test point would be connected to ground; the resistor component would reveal a constant voltage that could be predicted using voltage division principles with the protection resistor; the capacitor would have an increasing voltage from 0V to the step voltage level; the inductor would have a decreasing voltage from the step voltage level to a level predictable by voltage division with the protection resistor and the inductor’s

resistance; the diode would either reveal a constant voltage level matching the step voltage or a constant voltage equal to its forward bias voltage, depending on its orientation.

As stated earlier, the protection resistor value could be chosen to induce a similar time constant between the capacitor and inductor. Both time constant equations can be found below in Equation 2.

$$Tc(s) = R(\Omega) * C(F) \qquad Tl(s) = \frac{L(H)}{R(\Omega)} \tag{2}$$

Because the time constants were to be equivalent, the right sides of each equation were set equivalent and the resistance R was solved for. The result is shown in Equation 3.

$$R(\Omega) = \sqrt{\frac{L(H)}{C(F)}} \tag{3}$$

Substituting the values for the inductor and capacitor into the equation, a resistor with a value of $2236\,\Omega$ would cause each component to have the same time constant. For this reason a standard 5% tolerance $2.2\text{k}\,\Omega$ resistor was chosen to be the protection diode. The time constants predicted by the above equations is .22ms, meaning the voltage responses should reach steady state after 1.1ms. The circuit thus designed is shown in Figure 12.

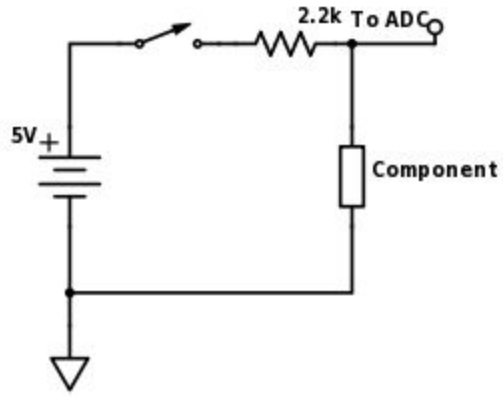

Figure 12. Preliminary Circuit

3.6.2 Actions Performed

The next task was to read to voltage trend for the attached component. To simplify the algorithm, two sets of samples would only be taken immediately following the excitation of the circuit and after steady state voltage had been reached. The algorithm tested whether or not a large increase or large decrease between samples was detected, signifying that the capacitor or inductor were connected, respectively. Following that the algorithm used experimentally derived

thresholds to determine the remaining components, first checking if the values signified a wire, then the forward biased diode, then the resistor, and ending by assuming that if those thresholds were not met that the component must be a reverse biased diode.

To experimentally obtain the threshold values the ADC results were displayed for each component for ten trials. A threshold value was then chosen to include those observed values as well as some readings that may contain some noise.

While contacting only one component pad and the common pad at a time would have been simpler in software, it was decided to contact all six pads at the same time to simplify the hardware aspect of the task. After the microcontroller reliably identified each component when only one component was connected, the next step was to reliably identify the positions of each component when all five were connected to the circuit. Each component was isolated using logic level MOSFET switches that could be easily driven by the microcontroller directly. The circuit designed is shown on the next page. A flyback protection diode is shown to protect the ADC when the step function goes low when the inductor is connected.The 10kΩ resistors connected to the gate of each FET are pull-down resistors used to reliably ensure the circuits would default to the open state. The designed circuit is given in Figure 13.

After the circuit was designed and tested in a breadboard it was soldered into a perfboard circuit to save space on the robot.

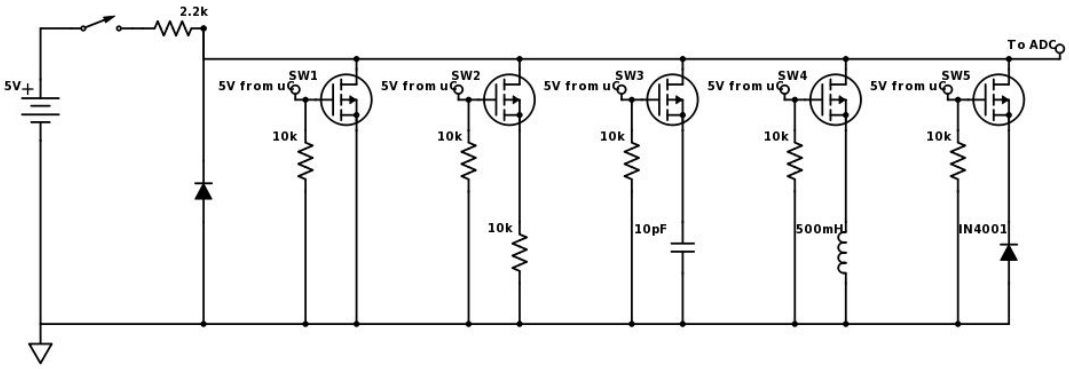

Figure 13. Component ID Circuit

With all the software and component identification work completed the final steps were to build the hardware that would be used by the robot to make contact with the arena. The initial idea that held through testing was to attach springs to a board that would be advanced to make contact with the arena stage. Springs were the probes prefered to others because they allowed for some error in the vertical plane to be corrected with ease. A plate of sorts was created with a 3D printer that had knobs positioned to correspond with the copper pad positioning on the arena. Springs with wires attached were glued to the knobs, and the wires were sent through a hole in the plate to the MOSFET switch array described above.

The plate was attached to a stepper motor driven linear actuator. The specific linear actuator chosen because of its long travel distance of about three inches; most other actuators in the price range only advanced half an inch. The actuator was mounted on a right-angle bracket that was then attached to the robot. The distance the actuator pushed the plate was initially arbitrarily set for one hundred rotations of the stepper motor, but the distance of approximately two inches was ideal for the placement of the motor on the robot.

If contact with the copper pads was not made, the robot would back up and adjust its position to attempt to line itself up correctly. Once contact had been made and the component locations were identified, the robot displayed the component codes on the LCD display on the top of the robot. The LCD was used to show the judges where the robot judged each component to be. This was necessary to accommodate the possibility that the robot would not perform Task 3 correctly and show the component locations in that way.

The subsystem block diagram for Task 1 is shown in Figure 14.

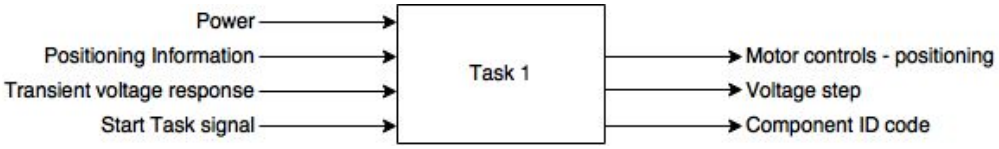

Figure 14. Task 1 Block Diagram

Figure 15 shows the flowchart describing the algorithm followed to complete the task.

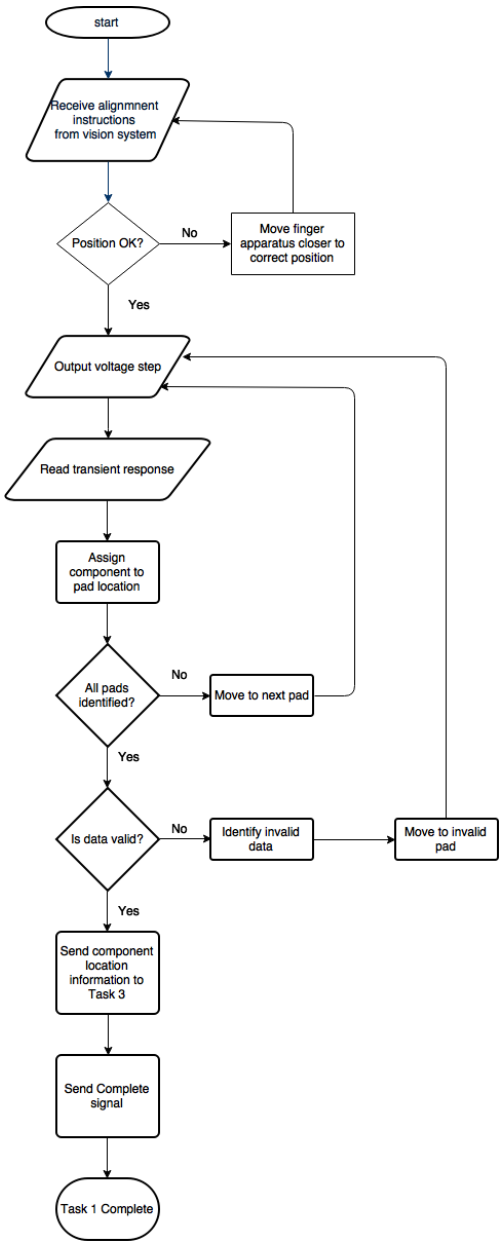

Figure 15. Task 1 Flowchart. See Appendix C for relevant code.

3.7 Lightsaber Duel - Task 2 Subsystem

Task 2 is a difficult subsystem to implement. It involves detecting a small change in magnetic field and acting on that information. Its design and implementation is described below.

3.7.1 Overview of Task 2 and Problem Analysis

The overall goal of Task 2 is for the robot to win a “lightsaber duel” with the arena’s lightsaber. The arena has a plastic 3D printed lightsaber mounted on the Task 2 frame. The lightsaber has a vibration sensor mounted inside it to detect when it is struck. An electromagnet is mounted beneath the lightsaber, behind the arena wall. During the duel, the magnet turns on and off at random intervals. Each time the magnet turns on, it stays on for two seconds. During that time, the robot must detect that the magnet is on and strike the lightsaber to gain points. If the lightsaber is struck when the magnet is off, points are taken away. Before the duel begins, the magnet is on. The robot must strike the lightsaber once to initiate the match, after which the thirty-second duel commences.

To solve this problem, several things must be taken into consideration. In particular, the robot must be able to detect and analyze the magnetic field. The robot must have an algorithm that can detect when the magnet is turned on, regardless of the direction of the current and therefore the direction of the electromagnetic field. The robot must also be equipped with an actuator or manipulator of some type that can be activated quickly to strike the lightsaber with enough force to activate the vibration sensor inside of it. The system inputs and outputs are relatively simple; they are shown in Figure 16 below. However, the key is to choose the right components and the right approach to solve the problem using them.

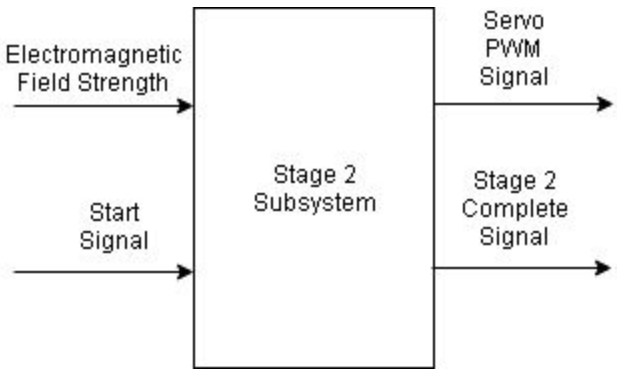

Figure 16. Task 2 Block Diagram

3.7.2 Magnetic Field Detection

First of all, if any points are to be gained, the robot must be able to detect and analyze the magnetic fields present. The robot must be equipped with some sort of component that can detect the magnetic fields. At first, a simple inductor coil was considered to convert the magnetic field into a voltage. However, because the electromagnet uses one amp direct current, this would at the least have been very difficult if not impossible. A voltage is only induced in a coil if the magnetic field present is changing. Because the magnet is powered using direct current, the only time that a voltage would be induced in a pickup inductor coil would be the short instant of time when the magnet is turning on or off. Another device would be necessary, in particular one that is very sensitive. According to calculations performed, an electromagnet with one amp going through 40 turns around a half-inch diameter core will produce a field strength of about 7.5 microTeslas two inches away from itself. (Two inches is about the distance between the inside edge of the arena wall and the outside edge where the magnet is located.) For perspective, the earth's magnetic field as around 50 microTeslas at the surface. To meet the task's requirements, a digital sensor orientation sensor was chosen. The MPU-9250 is a nine-axis sensor with a 3D accelerometer, 3D gyroscope, and 3D magnetometer on-board. Communication is handled using the I^2C serial protocol. The magnetometer, or digital compass, is separate from the other two sensors, and is known as the AK8963. When in 16-bit mode, the AK8963 has a sensitivity of 0.6 uT, so this should be sufficient to allow room for error when detecting a 7.5 uT magnetic field change. The magnetic readings from this sensor are signed, so the program can detect the direction of the magnetic field and subsequent changes. This sensor was purchased on a breakout board from Amazon. Its diagram is shown in Figure 17.

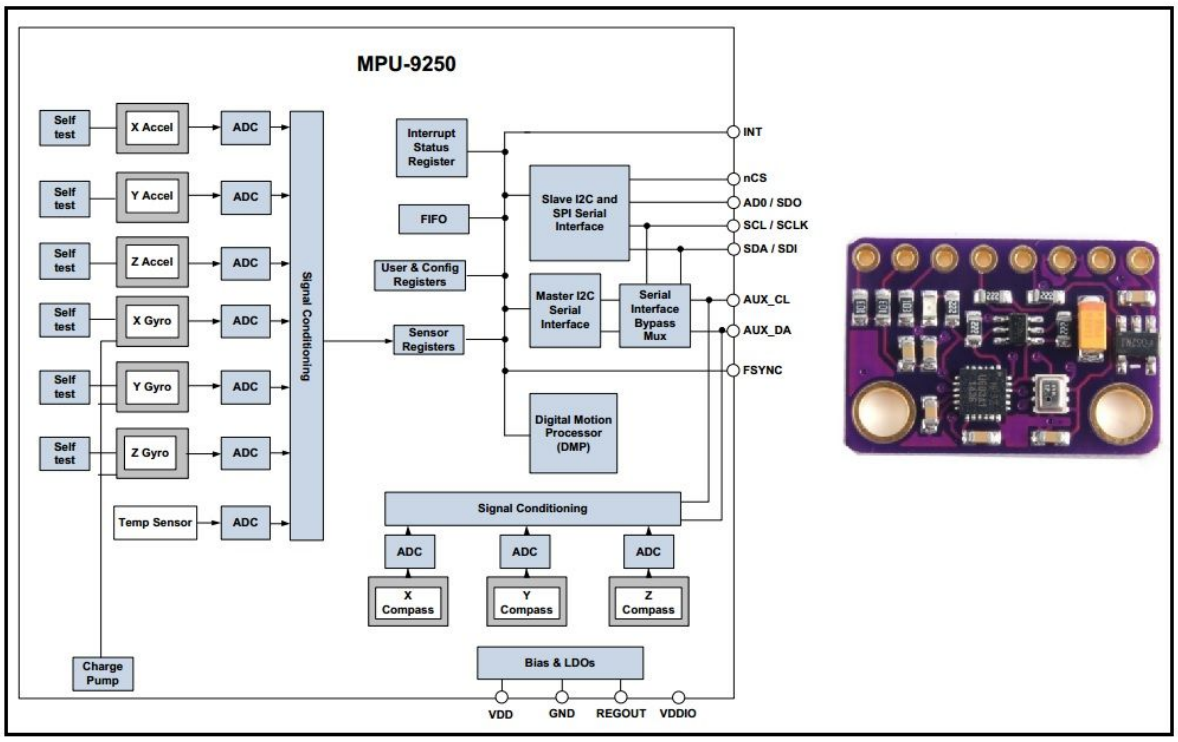

Figure 17. Electromagnetic Sensor

To use the MPU-9250, a connection must be created to it from the microcontroller over an I^2C bus. This proved to be both a challenge and a good learning experience about I^2C. A pre-existing piece of basic I^2C code that utilizes the Atmel TWI module was used as a starting point. The ATmega TWI (Two Wire Interface) module is a very effective processor resource that can be used for I^2C and SPI communications with minimal processor overhead. I^2C typically operates in either Master Transmitter mode, Master Receiver mode, Slave Transmitter mode, or Slave Receiver mode. For this application, an ATmega164A processor is the master and the sensor is the slave, so the Master Transmitter and Master Receiver modes were used. These are both modes in which all transmission is initiated by the master (the microcontroller) and data is either sent to or requested from the slave (the sensor). The pre-existing code library sets up a simple 15.2KHz connection with functions for Master Transmission and Master Reception. This was used as a wrapper library to create a library specifically for the AK8963. The I2C/TWI code library is shown in Appendix A2.3.

One of the challenges that was faced in creating code for the AK8963 magnetic sensor was in understanding how to properly address the sensor. Because the AK8963 is on a separate physical die within the same integrated circuit package, it is addressed separately from the other sensors and functionality in the MPU-9250. Looking at the block diagram in Figure 17, the compass sensors and their ADCs are shown on the bottom right of the diagram. These are connected to the auxiliary bus, which is typically used to allow the MPU-9250 to communicate, as a master, to other sensors including the AK8963 magnetometer. It seemed that communicating with the AK8963 through the slave interface of the MPU-9250 would add extra

complication, but there was an alternative available to us. There is a bypass multiplexer (shown on the right-hand side of the diagram) that can be used to connect the master microcontroller directly to the devices on the auxiliary bus. This multiplexer is accessed through a register in the MPU-9250. Setting this multiplexer to bypass the MPU-9250 master I^2C interface for the auxiliary bus allows the microcontroller to directly interface with the AK8963 from the main I^2C bus.

However, this approach did not achieve immediate success. As a communications test, an attempt was made to read the WHO_AM_I register. Because this register is on the MPU-9250, it can be accessed without setting up the bypass multiplexer. If communication is working properly, a read operation performed on this register should return a constant 8-bit value that is specified in the datasheet. However, when this was tried, the loop in the read function would simply hang, meaning that the slave device never responded to the read request by the master. Several approaches were used for solving this problem. First of all, the TWI library was double-checked with the datasheet. No problems were apparent. To check the hardware connections, the ATmega164A datasheet was consulted to find what size of pull-up resistor to use. This is shown below in Figure 18.

Rp	Value of Pull-up resistor	$f_{SCL} \leq 100kHz$	$\frac{V_{CC}-0.4V}{3mA}$	$\frac{1000ns}{C_b}$	Ω
		$f_{SCL} > 100kHz$	$\frac{V_{CC}-0.4V}{3mA}$	$\frac{300ns}{C_b}$	

Figure 18. I^2C Pull-up Resistor

The above formula was used to calculate a replacement for the 1K resistors that had been in use. Because a clock speed of 15kHz is used, the top formula is applicable. The suggested pull-up resistor value was 2.2K. However, switching these around did not solve the problem. After finding some Arduino examples for the MPU-9250 to pattern the code after, it was determined that the device was being addressed incorrectly. Whenever addressing a device on an I^2C bus, the device address makes up the highest 7 bits of the first byte after the start condition is issued. The LSB of this first byte is the R/W bit. If it is zero, then a write operation is being performed; if it is 1, then a read operation is being performed. The mistake was that the device address and the R/W bit were simply OR'd. The code was changed so that the address was shifted to the left by one before it was OR'd with the R/W bit. This solved the problem; the correct WHO_AM_I register value was retrieved. The bypass multiplexer was also able to be set, and so the AK8963 could be directly accessed. Magnetic field readings from specific axes could be read as well. After some more testing, it was determined that it was necessary to soft-reset the AK8963 after setting the bypass multiplexer and before configuring the AK8963. This allowed the microcontroller to effectively configure the measurement mode for continuous 8 or 100 Hz measurement and for 16-bit resolution.

The final version of the magnetic sensor AK8963 driver firmware contains an initialization function that enables the MPU-9250 bypass multiplexer, resets the AK8963, and configures the

measurement mode and bit resolution. It also contains functions that read and write from both the MPU-9250 and the AK8963. A header file contains many of the register value definitions making it very easy to modify it or use it in a variety of ways. This code for the Magnetic Field Sensor is shown in Appendix A4.2.

Using the magnetometer interface library, the magnetic field readings must be retrieved and processed so that the robot knows when the magnet turns on and off. The 16-bit signed values read from the sensor must be analyzed using simple sampling and signal processing techniques to adjust for environmental changes in electromagnetic interference and noise. The system must also be able to work regardless of the direction of the current in the electromagnet (and therefore the direction of the magnetic field.) A plan was developed that involved averaging and standard deviation calculations to generate thresholds before the match starts. The robot starts by driving gently up to Task 2 without activating the vibration sensor in the lightsaber. Because the electromagnet is on from the beginning of the competition round, the robot can take a measurement of the electromagnetic field strength in preparation for the duel itself. Once these measurements are taken, the lightsaber actuator strikes the arena's lightsaber and the match is started. All the robot has to do during the match is compare the incoming magnetic field readings to the threshold. When the real-time magnetic field crosses the threshold, the actuator strikes the lightsaber.

The algorithm used to calculate the thresholds is summarized in the formulae below. To use these formulae, the robot, before the start of the match, takes 16 samples of the magnetic field reading when the magnet is on. These numbers, which are 16-bit signed integers, are averaged and a standard deviation is calculated. Because of the possibility for arithmetic overflow when doing these math calculations on 16-bit numbers, intermediate 32-bit numbers are used. The author recognizes that this is very poor technique when using an 8-bit processor; however, any inefficiency in this process is unimportant. There is no other process waiting for the calculations to finish, they are not run in an interrupt, and they are only used once before the timed lightsaber duel starts;; therefore speed is not important. Equations #4 is used if the measured on average is numerically larger than the measured off average. If the off average is larger than the on average, than Equation #5 is used. The use of two separate formulae, given in Equation 4 and Equation 5, allows for the task 2 subsystem to operate regardless of the direction of the current in the electromagnet. Basically, the threshold generated will catch any magnetic field that is within the value of the standard deviation around the average of the magnetic field of the magnet when it is on. This approach uses numerical techniques to account for any variations in magnetic field direction and will also detect when the magnet is on regardless of field strengths used and electromagnetic noise present. These algorithms and the numerical functions used for them are all in Appendix A4.1.

$$ON_{thr} = \frac{ON_{avg} - OFF_{avg}}{2} + OFF_{avg} + ON_{stddev} \quad (4)$$

$$ON_{thr} = \frac{OFF_{avg} - ON_{avg}}{2} + ON_{avg} - ON_{stddev} \quad (5)$$

3.7.3 Lightsaber Actuator

Several options were available for use in striking the lightsaber and activating the vibration sensor inside of it. The need was for an option that provides fast effective response and does not take up much space on the robot. A linear actuator was definitely an option; it could provide effective force to activate the sensor, and a shield could be constructed to attach to the linear actuator that would have enough surface area to effectively reach the lightsaber regardless of minor navigation errors. However, this would have taken up a lot of space on the robot, and a more elegant solution was desired. The option that was chosen was a simple servo motor with an arm attached that could be swung into the side of the lightsaber. This solution would have sufficient speed and also the coverage needed to account for any minor navigational errors. A servo does not take up much space on the robot, and a swinging arm looks more appropriate for a lightsaber duel should than a linear actuator. Another option available to the team, which was eventually needed during the competition due to problems with the strength of and mounting of the servo, was that of ramming the wall beneath the lightsaber with the robot itself. This would have the same result of activating the vibration sensor.

3.7.4 System Operation

As was described in section 3.7.2, the operation of the system relied on the readings from the magnetic field sensor. The flowchart in Figure 19 outlines the overall system operation. One point to note is that after the robot strikes the lightsaber to initiate the match, there is a loop that waits for a “significant” change in the magnetic field. Testing in the lab showed that any change that is, in magnitude, 3x the standard deviation calculated for the magnet being on indicated that the magnet had turned off. The factor was reduced slightly in the competition because the ramming technique was used, as will be discussed in 3.7.5.

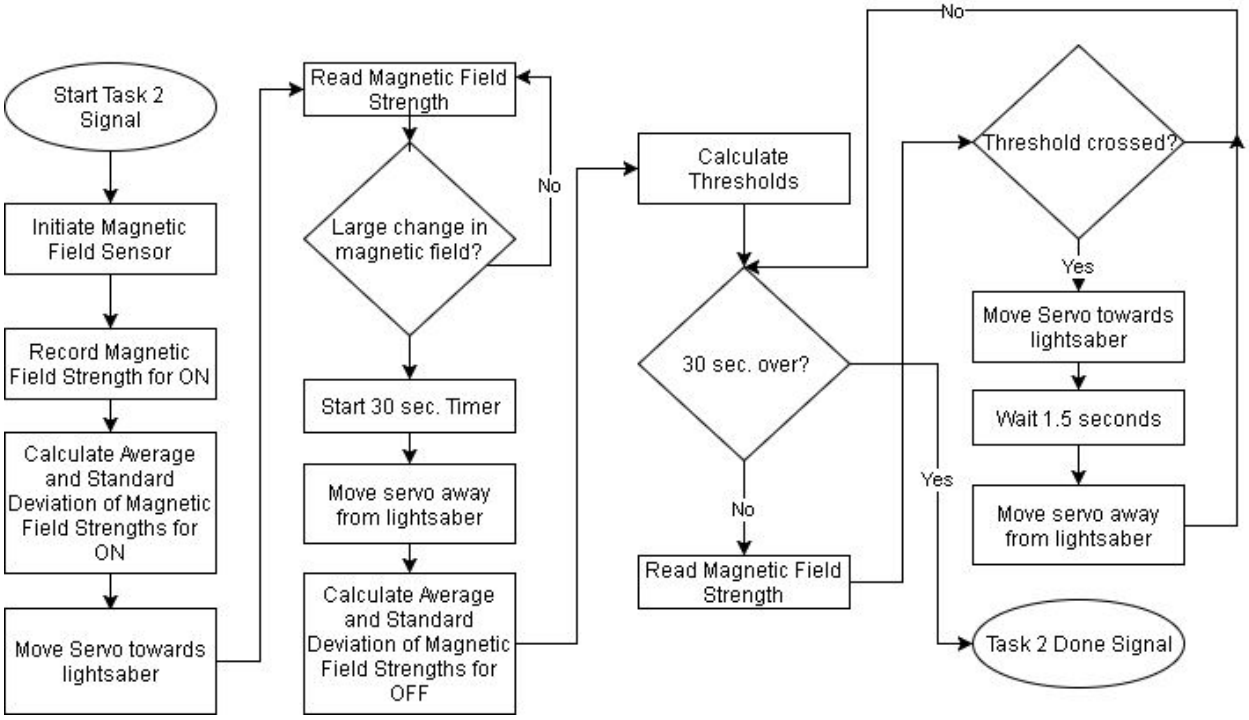

Figure 19. Task 2 Flowchart

3.7.5 Testing and Results

The Task 2 subsystem was tested using an arena controller that was built and programmed by Daniel. The purpose was to interface with the vibration sensor and the electromagnet so that the magnet could be controlled and the score counted based on the vibration sensor response. The system provided one amp for the electromagnet and also had a display to show the score. A simplified schematic of it is shown in Figure 20. The testing proved that the adaptive algorithm with threshold calculation allowed for accurate system responses with the magnet located at multiple different distances from the sensor. According to lab testing, the robot could have stayed as far away as 4 inches from the wall of the arena and the detection algorithm would still have worked properly. For the vibration sensor setup that was constructed in the lab, the sensor was fastened inside of a 3D printed lightsaber hilt using tape. While the servo motor setup provided enough force to activate the sensor using this lab setup, it was not enough for the sensor at the competition. During testing on the competition arenas the night before the competition, it was shown that the servo did not have enough power to strike the lightsaber with enough force to activate the vibration sensor. Also, the servo mounting on the robot (using Velcro) was not rigid enough and some of the force from the servo was twisting itself on the mounting piece instead of being put into the lightsaber. The code was changed the night before the competition to use the ramming technique instead of the servo. Because the detection algorithm was written using proper programming techniques, a few simple changes changed the sensitivity to adapt for the added distance that the robot had to be from the wall to ram into it. These changes allowed the Task 2 subsystem to gain up to 500 points cumulative from all three rounds. This put the team well on the way to second place in the open division.

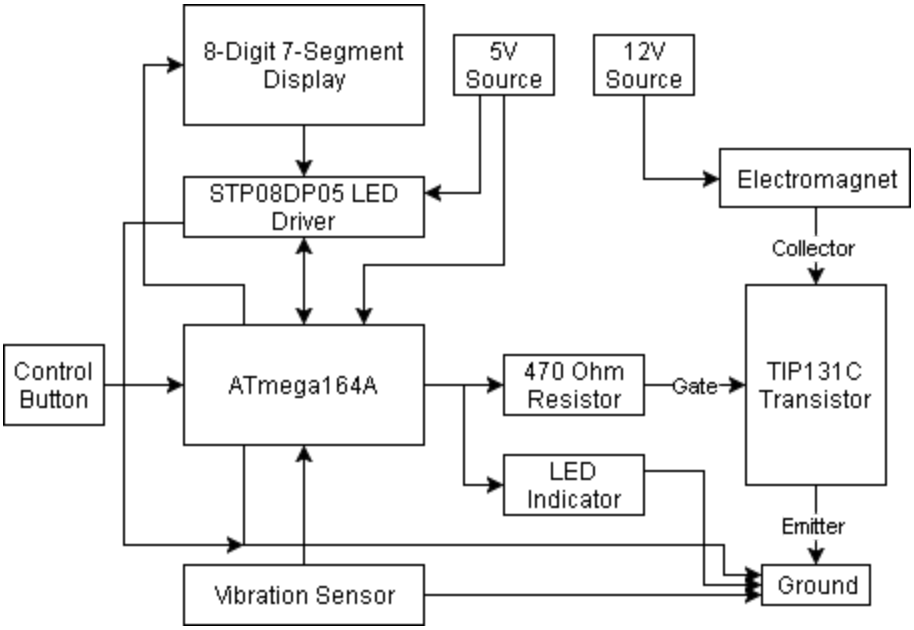

Figure 20: Arena Testing System Schematic

3.8 Lock Turning - Task 3 Subsystem

The lock turning subsystem takes the code data collected from the component identification subsystem in task 1 and uses it to turn a stepper motor the correct number of turns. The physical output of the system is quite simple; the robot must grab onto the arena knob and then rotate it according to the code collected in Task 1. Each number from Task 1 correlates to the number of turns in a given direction before reversing direction and turning again.

There are two main functions required for the Task 3 subsystem to be successful at its job. The first is a way to grab onto and release the knob on Task 3. The other function is to be able to precisely rotate the apparatus which is grabbing onto the knob. In the following two sections, the selection process for each function will be outlined.

3.8.1 Attaching to Task 3

There were three main options brainstormed in order to grab onto the knob in Task 3. Each of the options is outlined in Table 3 below. The paragraphs to follow describe the tests and conclusions of the options and justify the decision that was reached.

Table 3. Attachment Options. Servo-powered gripper was chosen.

	Grip Strength	Can Release	Wire Tangling?	Difficulty to fabricate
Plate with Sticky	Low/Mid	No	None	Easy

Surface				
Spring Gripper with release	Mid/High	No	None	Difficult
Servo Powered Gripper	**High**	**Yes**	**Maybe**	**Easy**

Using a sticky material was the simplest option, as all it requires is a piece of sticky tape and a surface to be rotated. This option is simple and would not have any extra cords that could get tangled in rotation. Since it was the easiest of options to fabricate, it made sense to test this proposed solution and only look to others if it didn't meet the required functionality. Upon upon testing, the sticky surface allowed for a good connection between the robot and the knob. However, there had to be a constant amount of pressure between the robot and the knob in order for the knob to rotate without slippage. During testing, it was determined that the navigation distance from the wall was not accurate enough to generate the desired result. In some instances, the robot would not drive close enough to create a strong enough connection. In other instances, the robot would drive half an inch further and as a result would put so much force on the knob that it would prevent the stepper motor from being able to rotate properly. It is for these reasons, and the fact that the sticky surface method would not allow for small adjustments in the knob position relative to the axis of rotation. If the robot was slightly off and the gripper arm was in use, the robot would correct itself slightly in terms of alignment allowing for a cleaner rotation that would cause less chance of slipping in the rotation motor. An example of a generic gripper is shown in FIgure 21.

Figure 21. Gripper Example

A spring loaded gripper, like that of a c-clamp that would be used in a machine shop, would be useful. With the given options available online, it proved difficult to find one where an apparatus could be used to hold the gripper open and then release it at a specific time. It sounds good in theory as it prevents a wire from getting tangled in the turning, but in practice would prove quite difficult as even the release mechanism would need to be rotated. Also, the holding force would need to be low enough to allow for the robot to still be able to drive away yet high enough to prevent any slippage. Because of the precise calibration required and the difficulty of assembly, this option was dismissed as an option unless all other options failed in testing.

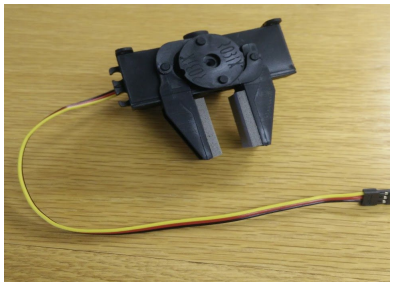

Figure 22. Servo-powered gripper

The servo powered gripper arm proved to be the best alternative to choose as it allowed for a centering of alignment and the ability to grab firmly onto and release the knob so the robot could cleanly accomplish task 3 before moving onto task 2. The actual gripper used in the competition is shown in Figure 22 above.

3.8.2 Rotating Task 3

Upon researching different types of motors that could rotate more than 360 degrees in both directions, the design selection was narrowed down to 3 distinct types of rotation motor. Table 4 shows each type of motor as well as its performance relative to the parameters that the group determined to be most important.

Table 4. Motor Options. Stepper motor was chosen.

	Precision	Size	Power
Continuous Rotation Servo Motor	Low	Small	Medium
DC Motor with Rotary Encoder	High	Large	Large
Stepper Motor	**High**	**Medium**	**Medium**

Figure 23. Motor options from left to right: Servo, DC, Stepper

The group determined that the parameters in terms of importance were precision, space, then power. The design parameters quickly dismissed a continuous rotation servo as a viable option because when a servo is configured to rotate in 360 degrees, it loses its ability to have position feedback. This meant that the servo would rotate different amounts given the same input depending on how hard it would be to rotate the knob, a parameter which could easily change between competition runs.

This left the decision to be made between the stepper motor and the DC motor with rotary encoder. Both have the required precision while one is slightly stronger than the other. At this point in the robot construction, the component for Task 1 was already constructed and the dimensions were known. This meant the team knew exactly how much space was left for the gripper arm and rotation motor in terms of component length. There were precisely 6 inches left for the Task 3 rotation apparatus. When measuring out the space taken up by both of the options, it was determined that there was just enough space for the use of the stepper motor and not enough space to use a DC motor with a rotary encoder. As a result of this insight, the stepper motor was selected as the best option.

3.8.3 System Functionality

The servo gripper required a PWM signal to switch from an open position to a closed position with enough holding torque to allow the arm to rotate without losing a grip on the knob. A functional PWM code was created for the servo and was calibrated to grab onto the knob with a force strong enough to firmly grip the knob without damaging the motor.

The stepper motor required a dual H-bridge to power the coils since it would require a 0.5 amp current draw at 4 volts for each coil. To make the stepper motor rotate, the circuit would have to apply voltage steps to the 4 input terminals in the configuration shown below. In order to change direction, all that was required was to reverse the sequence of the pulse signals. The control signals are shown in Figure 24.

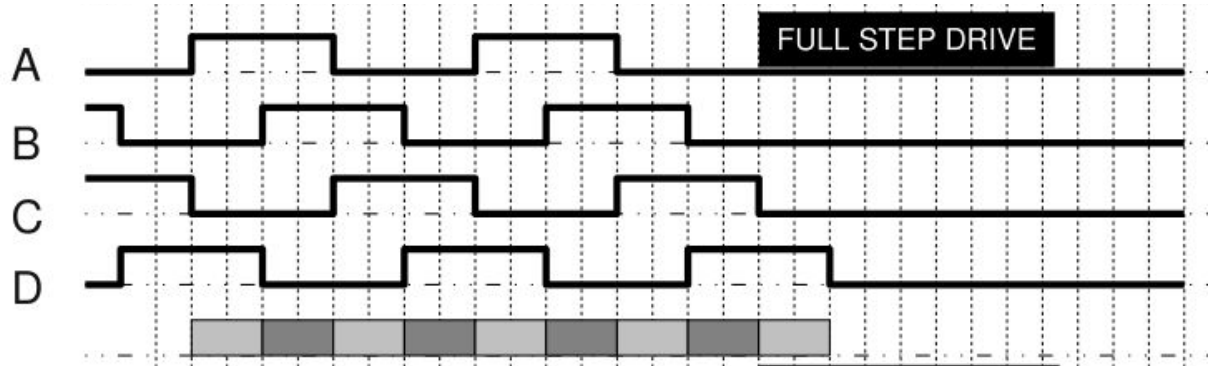

Figure 24. Stepper motor control

Upon successfully implementing a driving function for both the clockwise and counterclockwise direction, the team created a rotation function for each direction. This rotation function would

take the input of the number of steps required to do a full 360 degree turn as the input and as a result rotate the shaft 360 degrees. These two functions are shown below in the code as the functions motor_in_cw() and motor_out_ccw().

```
void task_3_process(void)
{
	OCR0B = 30;		//close gripper
	_delay_ms(2000);

	//Convert 16 bit input (component_code) to rotation numbers
	//t1=number of rotations in the first turn
	//t2=number of rotations in the second turn...
	uint16_t t5, t4, t3, t2, t1;
	t5 = 0x7 & component_code;
	component_code = component_code >>3;
	t4 = 0x7 & component_code;
	component_code = component_code >>3;
	t3 = 0x7 & component_code;
	component_code = component_code >>3;
	t2 = 0x7 & component_code;
	component_code = component_code >>3;
	t1 = 0x7 & component_code;

	//Turning Routine 20 seconds
	enable_task3();
	for (uint8_t i=0;i<t1;i++){motor_in_cw(TASK_3_ROT_STEPS);}  _delay_ms(200);
	for (uint8_t i=0;i<t2;i++){motor_out_ccw(TASK_3_ROT_STEPS);} _delay_ms(200);
	for (uint8_t i=0;i<t3;i++){motor_in_cw(TASK_3_ROT_STEPS);}  _delay_ms(200);
	for (uint8_t i=0;i<t4;i++){motor_out_ccw(TASK_3_ROT_STEPS);} _delay_ms(200);
	for (uint8_t i=0;i<t5;i++){motor_in_cw(TASK_3_ROT_STEPS);}  _delay_ms(200);
	disable_task3();

	_delay_ms(200);
	OCR0B = 18;
	_delay_ms(1000);
	Servo_Off();

}
```

The full stage 3 code implemented above would first close the gripper, then convert the 16 bit signal from stage 1 into the 5 digit code. Then, use those numbers t1-t5 as integer values for the number of times to implement the rotation function before waiting. Lastly, it will open up the gripper arm up after the task 3 rotation function is finished running. The full code used for stage 3 which includes the function declarations for motor_in_cw() and motor_out_ccw() can be found in Appendix E.

The following block diagram in Figure 25 is a simplified version of what the code for component 3 is designed to do.

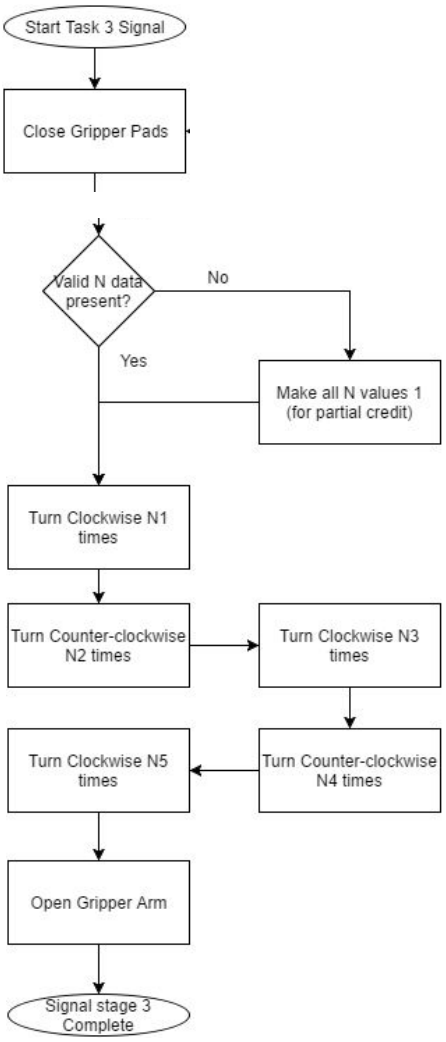

Figure 25. Task 3 Flow Chart

Once the two systems were functioning independently, the two systems were fixed together with a wooden mount and slip washer as shown below. The wooden mount was custom made to be able to be mounted on to the gripper arm because it would allow the shaft to affix to the gripper without drilling any holes in the gripper or using any glue to affix it. Figure 26 shows the completed mount.

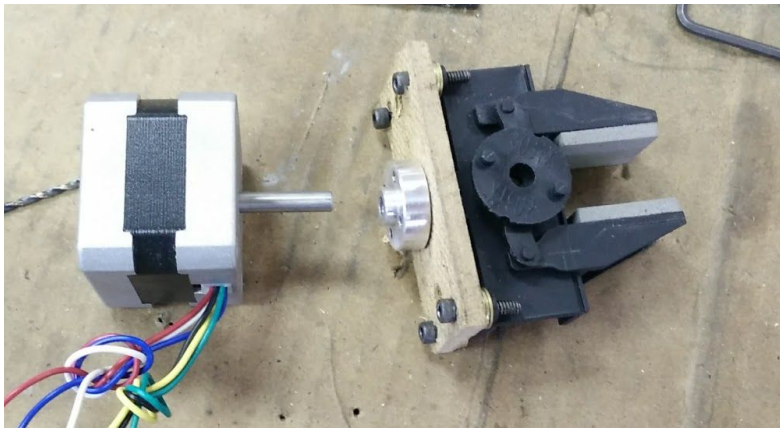

Figure 26. Task 3 Assembly

The system was tested with various sets of input data and functioned properly. It was mounted on the upper platform of the chassis at the appropriate stage height and was still able to maintain full functionality.

Upon testing on the robot, it became clear that the wire powering the servo gripper could get tangled up during the course of executing the code. As a result, a protective shielding was designed in order to prevent the wire from getting caught through the series of rotations.Figure 27 demonstrates the functionality of the protective shielding.

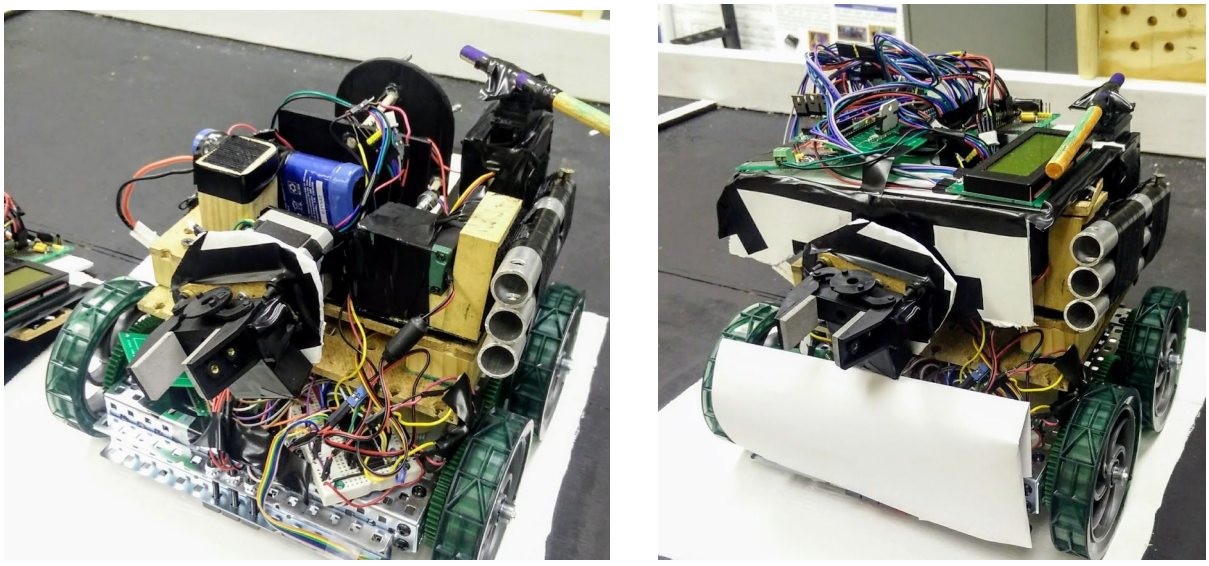

Figure 27. Robot Task 3 Before and After Wire Shielding

Figure 28 shows the I/O structure for the whole subsystem. The overall signal flow was quite simple. Once the component receives the start signal, it takes the 5 digit code from Task 1 and

uses that to output the appropriate gripper open/close and arm turning functions. The final output is the done signal back to the navigation so the robot can continue on to the next stage.

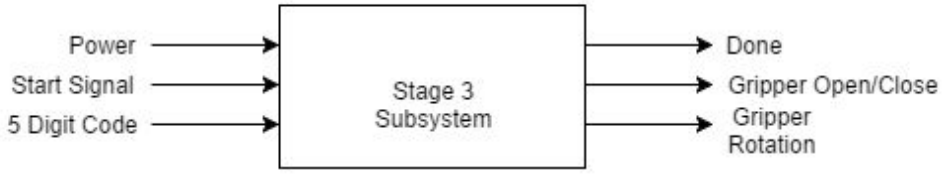

Figure 28. Task 3 Block Diagram

3.9 Dart Firing - Task 4 Subsystem

The dart firing subsystem takes a signal from the navigation system as an input. This input tells the dart to fire when the robot has been lined up with the target. The only output is the dart firing after aiming is complete. The match ends once the last dart is fired. Figure 29 demonstrates the simplicity of the subsystem.

Figure 29. Task 4 Block Diagram

The initial proposal was to simply attach three Nerf pistols to the robot and pull the triggers using a servo motor. One of the guns is shown in Figure 30. This option was soon discarded because three Nerf pistols simply occupied too much space. Additionally, the triggers for each pistol took more force than could be expected from small electric motors that could fit on the robot. Upon testing in the machine shop it was determined that over 3 pounds of linear force was required.

Figure 30. Task 4 Discarded Nerf Gun Idea

Once this idea was discarded, progress on Task 4 stopped. It was decided that any more effort would be better spent on other tasks where more points could be scored. Development of Task 4 continued after the three previous tasks had been completed and time was still available to dedicate to Task 4.

The final design was a more compact and powerful version of a prototype developed by Mr. Schmidt. Using his design for liberal inspiration in the final product, shorter and stiffer springs were used to conserve space. The tube used to fire each dart was also made shorter to conserve space. Finishing nails were used as pins to hold the springs back. The nails were tied with fishing line to a DC motor that, when on, wound the tethers and pulled the pins, releasing the springs and firing the darts. Figure 31 shows the final design for the component for Task 4.

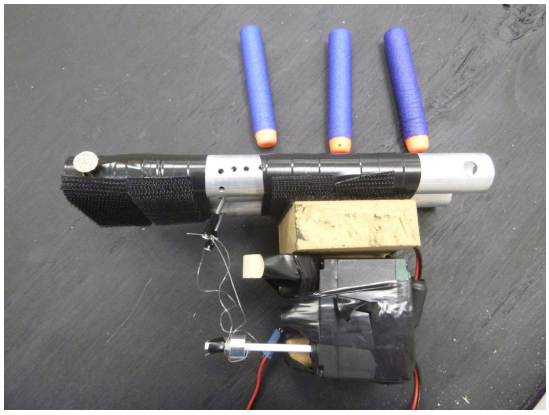

Figure 31. Task 4 Final Dart Launching Mechanism

The robot is designed to stop at the first step in a position that allows the darts to be accurately fired without any further aiming. This design required much less force than the nerf gun trigger while also saving space by being able to stack the barrels on top of each other. Figure 32 below shows how the component for Task 4 was able to fit onto the robot and be adjusted by removing and attaching the barrels with velcro. It also shows the final navigation waypoint of the robot before sending the command to launch all three darts.

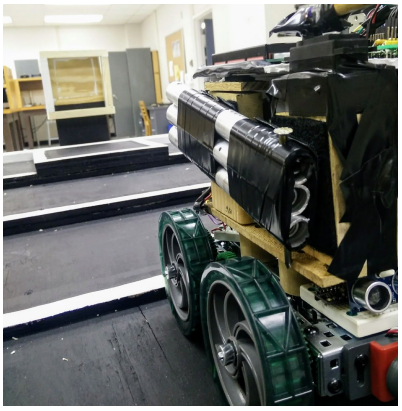

Figure 32. Task 4 Final Dart Launching Mechanism Attached to Robot

4. System Integration, Testing and Evaluation

Each subsystem was built independently of the others. Once the subsystems functioned well on their own, they were combined into one complete system. A stable chassis was needed for accurate operation of each task and navigation. Also, a design that allowed for the components to be easily moved and fixed at the competition was desired.

4.1 System Integration - Hardware

The robot chassis went through a few design iterations. What was initially thought of as advantageous ended up making the robot difficult to function as discussed below.

4.1.1 First Robot Chassis

The first robot chassis that was created included a 10” square base platform with the base board being 4.5” off of the ground. On top of the first base platform was another 10” square platform that could rotate freely allowing for any stage on the top to be rotated to the front of the robot.

The rotating platform above the base is another 10” square upon which the components of the robot can be attached. The rotation is done with a lazy susan bearing rotator, which allowed for a simple and smooth rotation relative to the base. The constructed robot is shown in Figure 33.

Figure 33. First Robot Chassis

Through testing, it was discovered that the navigation of this robot was not very precise and that the rotation of the top platform did not allow for wires to travel easily from above the base to below the base. Also, as the development of the stages continued, it was discovered that the electronics would end up taking much more space than previously expected. For these reasons, a new chassis was designed to take up less space and allow for more precision.

4.1.2 Final Robot Chassis

The new robot chassis was assembled from a robotics starter kit and then built upon with a new platform fabricated in the university machine shop. It was based on the idea of modular design

where each component could be moved around upon the chassis and function independently of each other. This base is shown in Figure 34.

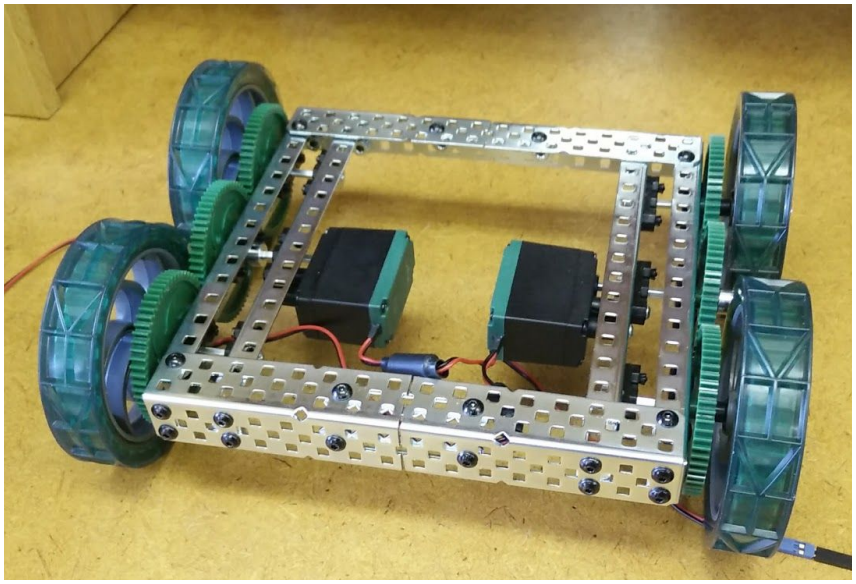

Figure 34. Robot Base. Made from a Vex Robotics kit.

The new design allowed for each component to be easily detached and moved around. As shown in Figure 35, each element was made to detach from the robot simply so it could be worked on independent of the others.

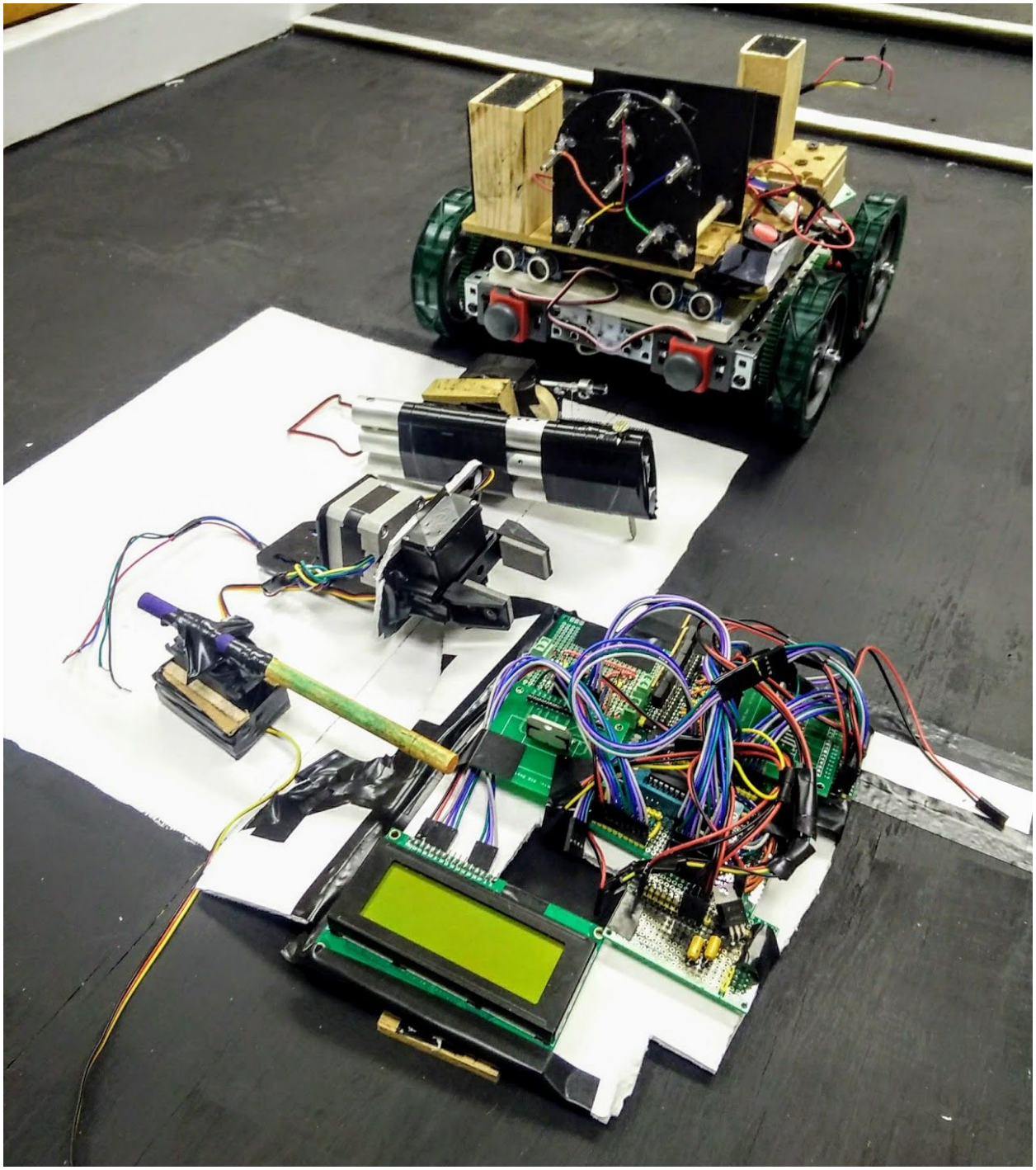

Figure 35. Disassembled robot chassis

The base chassis, shown below in Figure 36, has a split level platform for the components to mount upon. The lower level was constructed at the height for Task 1 to be mounted on and the second level was constructed at the height for Task 1 to be mounted on. The following two figures show the base mounting platform with and without the first 3 tasks mounted on top of it.

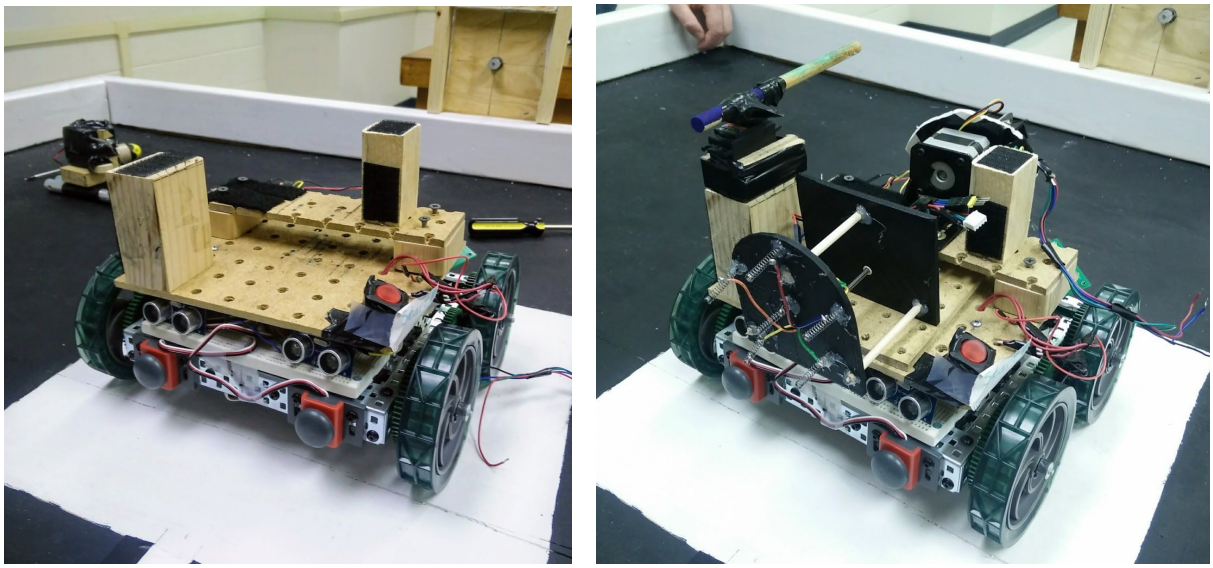

Figure 36: Robot Chassis With and Without the First 3 Components Attached

Description of mounted components:
Task 1 was made to be mounted to the frame with a DC motor mounting bracket.
Task 2 lightsaber was made to be attached via velcro on a pillar to the left of Task 1.
Task 3 gripper was made to be mounted to the upper platform.
Task 4 was made to be mounted to the side of the Task 2 pillar and the velcro on the top of the upper platform.
The battery was able to mount on the velcro of the supporting pillar next to stage 3.
The electronic components on top were laid out to fit precisely onto a piece of posterboard that could mount to the top of the support pillars for stage 2 and the battery.

With the way the robot was designed, it could be completely taken apart or put back together in the course of 5 minutes. Figure 37 displays the robot with all components put together: tasks 1-4, the electronics top platform and the Task 3 shielding.

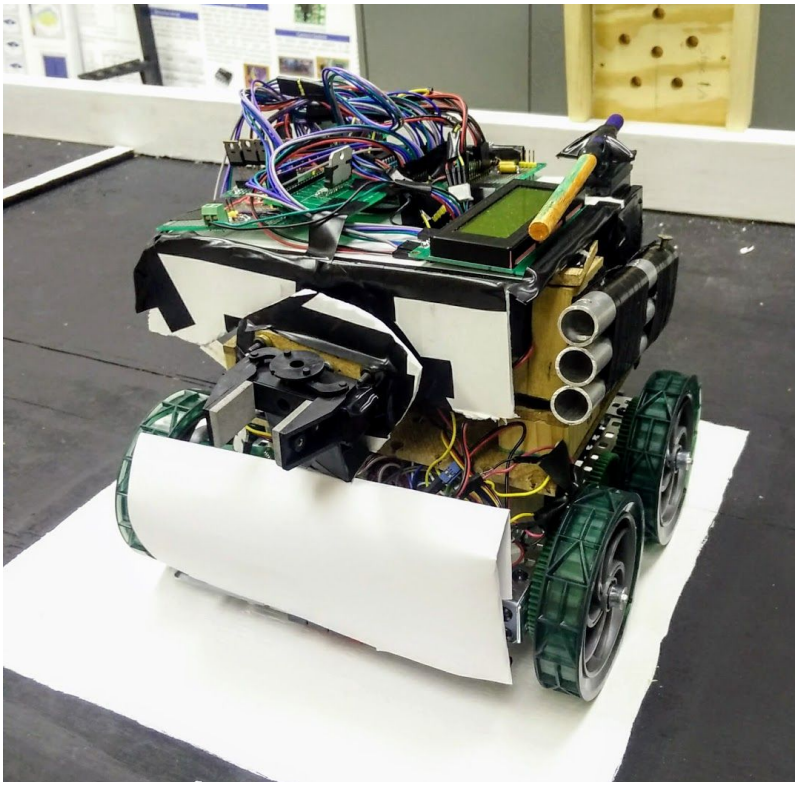

Figure 37. Fully Assembled Robot Used During the Final Competition

4.2 System Integration - Software

In addition to combining the subsystems physically, their code needed to be merged. Ideally, all the subsystems would be controlled by one microcontroller. However, limited resources prevented this. ATmega microcontrollers are used due to their low cost and the team's previous class experience. The largest DIP Atmel chip, the ATmega164A, has 44 pins. This is not enough for the requirements of the integrated robot. As a result, an ATmega328P, with 28 pins, is used in addition to an ATmega164A. Using two microcontrollers also makes programming easier because of the additional timers and external interrupts.

Navigation, Task 2, and Task 4 are combined on the ATmega164A. Task 1 and Task 3 are controlled with the ATmega 328P. There is little communication between the two microcontrollers. A one-way signal from Task 1 to Navigation communicates the success or failure of the probing. Navigation waits until it receives this signal via an external interrupt. If successful, the robot continues to Task 3. If it receives the failure signal, Navigation moves the robot to the side. Task 1 has a delay of 5s, then tries again. Task 3 is also timed using a delay. Tasks 2 and 4 are on the same microcontroller as the navigation, so they receive a start signal. This system is not the most elegant, but results in reduced complexity. The robot easily finished within the four minute time limit. Figure 38 shows the division of the subsystems and the simple communication between microcontrollers.

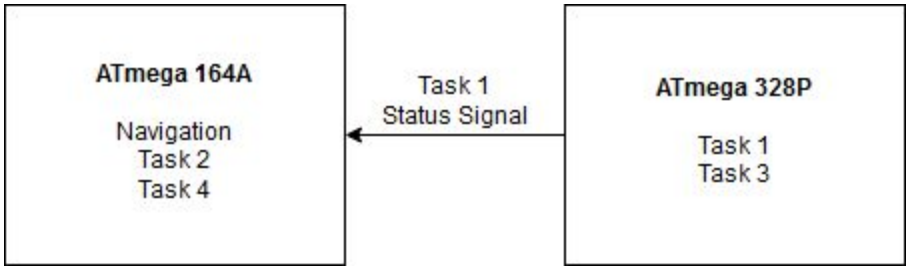

Figure 38. System Integration Block Diagram

The code snippet below shows how Navigation uses the Task 1 status signal. See the full code in Appendix B.

```
else if (step == 2)           //Wait for task1
        {
                if (((PINB | 0xFD) == 0xFD) || ((PINB | 0xFE) == 0xFE) || ((PINB | 0xFC) == 0xFC))
                {
                _delay_ms(500);
                Set_Motor('R','S');
                Set_Motor('L','S');
                }

                if (IL_getTask1Status() > 0)          //task 1 status signal received
                {
                _delay_ms(5);
                if (IL_getTask1Status()==1) //success
                {
                        IL_reset();
                        zero(addressLeft<<1);     //reset encoders
                        zero(addressRight<<1);
                        _delay_ms(2000);          //allow time for Task 1 arm to retract
                        Set_Duty_CycleL(speed_ToStage3L);
                        Set_Duty_CycleR(speed_ToStage3R);
                        Set_Motor('L','B');                     //move toward stage 3
                        Set_Motor('R','B');
                        step=3;
                }
                else      //failure
                {
                        IL_reset();
                        step = 99;
                }
                }
        }
else if (step == 99) //adjustment
        {
                if (rightAdjustments<3) {
                nav_Task1AdjustRight();
                rightAdjustments++;
                step = 2;
```

```
}
else if (leftAdjustments<4) {
nav_Task1AdjustLeft();
leftAdjustments++;
step = 2;
}
else {
zero(addressLeft<<1);      //reset encoders
zero(addressRight<<1);
_delay_ms(2000);
Set_Duty_CycleL(speed_ToStage3L);
Set_Duty_CycleR(speed_ToStage3R);
Set_Motor('L','B');
Set_Motor('R','B');
step=3;
```

4.3 Competition Results

Below is a description of the results of the robot at the competition.

4.3.1 Competition Overview

The final competition was on Saturday April 1st in Charlotte North Carolina. There were 51 teams that were competing and 8 teams in the open division that the Bradley team was in. The competition was scored by the total of points collected after going through 3 qualifying runs.

The group traveled out the day before to problem solve any last issues on their practice arenas before the final competition. During the three qualifying runs, the robot progressively increased in score each time as modifications were made to the robot.

4.3.2 Robot Functionality

The robot was able to appropriately line up with and gather most of the information from Task 1 on all three runs. This was a difficult task which many of the robots were not able to complete.

Upon backing up to Task 3, the robot had some problems lining up properly and was not able to rotate the knob with the code attained in Task 1 for any of the three runs. Even though Task 3 was not lined up properly, the code from Task 1 was still executed by the robot, just not while holding Task 3.

Despite the fact that Task 3 was not lined up properly, the robot was able to turn and drive up to Task 2 for all three of the runs allowing us to gain many points on each run. After adjusting the thresholds for detecting the magnetic field at Task 2, the robot progressed from receiving partial credit on the first two runs to achieving full credit for the task on the final run.

The robot was able to fire two out of the three nerf darts through the portal upon driving up to the first step about 4 feet away from the target on the last two runs.

4.3.3 Changes Made During the Competition

Upon arriving at the competition the team came across many unexpected difficulties that had to be addressed in order for the robot to function fully. This section details a brief timeline of the events that went on before and during the competition.

Before the Competition Rounds
The first issue encountered was that the height of Task 1 was not the same on the test arena at the competition as it was at the test arena at Bradley. In order to address this issue, the team had to raise the component for Task 1 by an eighth of an inch. Because the robot was designed to be modular, it was a simple fix to add an extra piece of pegboard on the place where the

Task 1 component was mounted. In fact, the team came prepared with such a piece in case it was needed.

Another concern was that the robot was not consistently lining up with Task 1. It turned out that the infrared sensors did a good job of determining what side of the line the robot was on, however, the robot was not able to move precisely enough to make the right corrections. To address the issue, the team took away the programming for the infrared sensors and made extra sure to line up the robot appropriately with Task 1. The realignment algorithm ensured that the robot would correct itself if Task 1 did not align the first time.

The vibration sensor on Task 2 was not as sensitive as the vibration sensor was on the testing arena the team used at Bradley. When the robot's lightsaber struck the arena lightsaber there was not enough power for the arena to register a strike. As a result, the team had to modify the lightsaber. The team first tried adding weight to the lightsaber to give it more momentum upon swinging. Upon testing it was found that it still did not have enough force. Another option was to ram the wall as opposed to using a lightsaber. This required some programming to change the way the robot worked, but proved to work well during testing. As a result, in each of the three competition rounds the robot used the wall-ramming algorithm that was created at the competition.

During the Competition Rounds

In the first run, the robot did not align successfully with Task 4 and also was too sensitive during the lightsaber duel. The team had 30 minutes to work with these issues and come up with a solution. During this time, the team changed the algorithm used for the threshold magnetic field detection for Task 2. The team also used the test arena to make adjustments to the navigation so Task 4 would line up properly. This allowed for a greater score of points in the second round.

During the second round the robot successfully aligned with Task 1, then backed up to Task 3. However, it was not center-aligned and as a result could not rotate the knob more than a few degrees in either direction. Navigation was robust enough for the robot to still line up properly with Task 2 and do well on a lightsaber duel before backing up to Task 4 and firing two out of the three darts successfully.

Task 2 was still too sensitive, so the team adjusted the thresholds again and tried to get Task 3 to align better. The algorithm for Task 2 worked better and the team scored all the points available for Task 2 and still lined up Task 4 well enough to get 2 out of 3 darts through the portal. Unfortunately, Task 3 never aligned properly during the competition.

4.3.4 Results

Overall, the team came in 9th place out of the 51 teams and 2nd in the group of 8 teams in the open division. The scores of the Bradley team's runs are posted in Figure 39, as well as a picture of the team receiving the award for 2nd place in the open division in Figure 40.

Rank	Team #	University	O/M	R1	R2	R3	Total
1	27	Tech Memphis	open	560	510	420	1490
2	3	Bradley University	open	280	450	500	1230
3	29	Citadel	open	95	245	190	530
4	42	UK Kids	open	70	185	80	335
5	16	UNC Charlotte	open	60	180	75	315
6	50	UNC Charlotte	open	75	50	75	200
7	43	Georgia Institute Of Technology	open	40	50	110	200
8	55	University of West Indies	open	90	100		190
9	35	South Alabama	open	0	0		0

Figure 39. Open division competition results

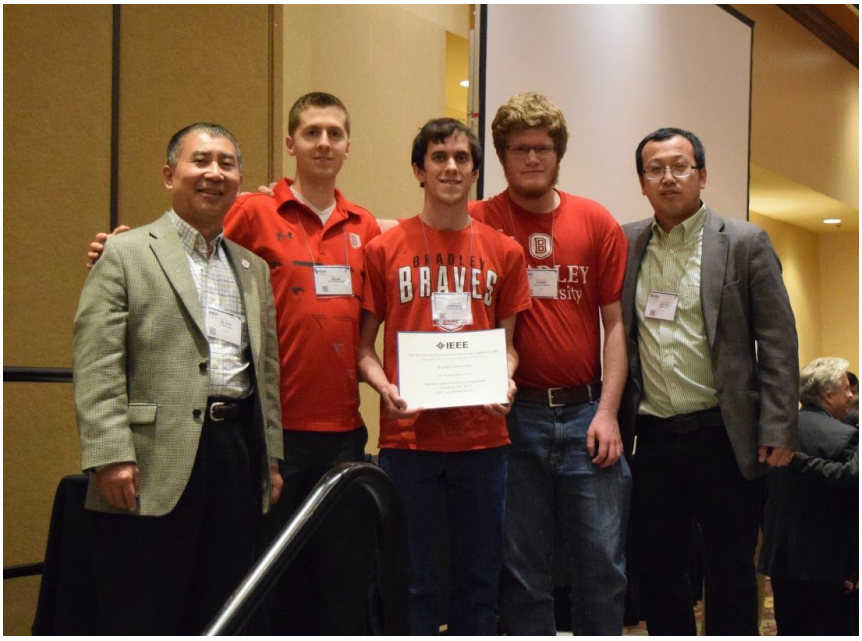

Figure 40. Bradley IEEE Competition Team. Dr. Wang and Cameron were not present at the competition.

5. Division of Labor

Each team member was initially responsible for one task, designing the components necessary to complete it. The division of labor was later changed according to the workload of each member. Table 5 describes how the tasks were divided in practice.

Table 5. Division of Labor

Team Member	Functionality	Task
Cameron	Protoboard circuit construction	1: Component ID 4: Launching Dart
Daniel		2: Lightsaber Swing
Brian	Robot Chassis Layout/Design	3: Turning Knob
Kendall	Navigation, Software Integration	

As the competition moved closer the team began to collaborate more and more to help with any tasks that were falling behind schedule. There was also many team meetings working to fit all of the components together on the chassis. There was also much collaboration in order to get all of the components of the robot interfaced together on the two microcontrollers.

6. Possible Improvements

Based on the team's experience at the competition, there are several suggestions that can be made to improve the accuracy of the robot and therefore improve the scoring if it were to rerun the competition. These improvements would mainly have to do with task 3 and with the navigation.

Despite the excellent work that was done to create a reliable and precise navigation system, there are many variables in the competition arena that can affect the responses of the sensors and therefore their reliability. For example, despite the high precision of the encoders, there is still some play in the wheels and in the gearing that could lend itself to errors. In particular, the biggest concerns are with lining up properly with tasks 1 and 3. While the line sensors were helpful with lining up for task 1, it is possible that the resolution was too low, especially when combined with the limited short-range motion constraints on the robot's motors and wheels. Three line sensors were used, which gives only a rough idea of where the robot is. Perhaps a vision system would be more useful, whether used to identify the location of the task 1 pentagon layout or to view a higher resolution image of the line leading up to task 1.

Task 3 was a source of significant struggle for the robot during the competition. While the gripper and stepper assembly was designed well, it required a precise alignment with the knob to work properly. The gripper had to grab the knob straight on or the stepper would simply torque to the left and right without turning the knob. Other solutions that might work would be using some sort of sticky surface to grip the knob, but this is not very reliable either. The best way to improve task 3 would be to use a vision system to help line the robot up with the knob. Also, more ultrasonic sensors on the sides or back of the robot could be used. This would improve the accuracy far more than using the encoders to simply drive straight backwards from task 1.

7. Conclusion

Overall, this project was successful in that all the goals of the project were completed. All team members learned how to apply lessons learned in class to physical products, even if, as was the case with the vision system, the work was not used on the final competition version of the robot. The team members also learned about different technologies, the areas where those technologies excelled and their limitations, such as ultrasonic distance sensors, IR sensors, stepper motors, and H bridges. Team members also learned how to work under the pressure of an imminent deadline to finish the most vital tasks before refining and improving other aspects of the project.

Other important lessons learned involved the communication of progress made to superiors. The team had to demonstrate weekly the progress made on the project. Additionally, all members learned how to effectively share the ideas central to the project via reports and presentations. Oftentimes the presentation of the work completed to superiors is more important to a career than the work itself.

Arguably more important than learning about technology or communication of progress, the team members learned how to work with other engineers toward a common goal. Each person developed skills necessary to share any frustrations with other team members in a constructive way that prevented the project from grinding to a halt. All of these lessons will undoubtedly help each person advance in his career to be the best engineer he can be.

8. References

[1] *SoutheastCon 2017 Hardware Competition Rules* [Online]. Available: http://sites.ieee.org/southeastcon2017/student-program/student-hardware-competition/

[2] Kuman, “Ultrasonic Ranging Module HC - SR04,” HC - SR04 datasheet, n.d.

[3] Atmel, “Atmel 8-bit microcontroller with 4/8/16/32Kbytes in-system programmable flash,” ATMega328P datasheet, Dec. 2009 [Revised Nov. 2015]

[4] Atmel, “8-bit Atmel Microcontroller with 16/32/64/128K Bytes In-System Programmable Flash,” ATMega164A datasheet, Jan. 2010 [Revised Jan. 2015]

[5] Atmel, “8-bit Atmel Microcontroller with 4/8/16K Bytes In-System Programmable Flash,” ATMega168 datasheet, Jan. 2004 [Revised Apr. 2011]

[6] InvenSense, “MPU-9250 Product Specification Revision 1.0,” MPU-9250 datasheet, Jan. 2014

[7] InvenSense, “MPU-9250 Register Map and Descriptions Revision 1.4,” MPU-9250 datasheet, Sep. 2013

[8] B. Wilkins, “MPU-9250 Hookup Guide,” sparkfun.com, 2016. [Online]. Available: https://learn.sparkfun.com/tutorials/mpu-9250-hookup-guide. [Accessed Jan. 19, 2017].

[9] K. Winer, "Arduino sketch for MPU-9250 9DoF with AHRS sensor fusion,” github.com, Apr. 1, 2014 [Revised Jan. 10 2017 by B. Wilkins]. [Online]. Available: https://github.com/sparkfun/SparkFun_MPU-9250_Breakout_Arduino_Library. [Accessed Jan. 28, 2017].

A1. Appendix A - Parts List

A list of parts that have been purchased during design work or that are used on the robot is shown below. The current total of $332 is well within the budget of $1000.

Description	Quantity	Cost (if known)
Arena		
Wood		$40
Paint	2 Quarts	$20
Medium Vibration Sensor	1	$0.95
Clear Plastic Knob	1	$0.95
Rotary Encoder - Illuminated	1	$3.95
Task 1		
Linear Actuator	1	$20
Logic-Level FETs	5	
Low forward voltage diode: < .3V	2	On hand
LCD	1	
Task 2		
MPU-9250 Magnetic Field Sensor	1	$15.99
Standard Servo Motor	1	On hand
ATmega168A, DIP	1	On hand
Hall Effect Current Sensor	1	$4.50
Task 3		
Lynxmotion Little Grip Kit	1	$15.99
HS-422 Servo	1	$9.69
Task 4		
Nerf Jolt Guns	3	$18.97

Chassis		
Parallax Ultrasonic Sensor	4	$29.99
Auto EC 68 mm wheels x4	1	$9.99
4'x8' MDF Sheet	1	$19.17
6" Lazy Susan Everbilt	1	4.48
#3 screws x10	2	$6.28
1" Zinc Coated Caster Ball Vestil	2	$6.50
Pololu Wheel 42x19 mm	2	$13.96
12V DC Motor W/rotary encoder	3	On hand
Right Angle DC Motor mount 37mm	3	$26.97
Total:		**$332**

A2. Appendix B - Navigation and System Control Code

This code was implemented on an ATMega164A microcontroller.

A2.1 Main Navigation

```
/*
 * main.c
 *
 * Date: 4/1/2017
 * Author : Kendall Knapp
 * Purpose: Main navigation code that controls the robot and moves it between tasks. Also controls Tasks 2 &
4.
 */

#define F_CPU 8000000L
#include <util/delay.h>

#include <avr/io.h>
#include <avr/interrupt.h>
#include "bios_timer_int.h"
#include "interrupt_ext.h"
#include "Motor_PWM.h"
#include "I2C.h"
#include "Encoder.h"
#include "ADC.h"
#include "Servo_PWM.h"
#include "Magnetic_Sensor_Driver.h"
#include "Task2.h"

void nav_LineAdjustRight();
void nav_LineAdjustLeft();
void nav_Task1AdjustRight();
void nav_Task1AdjustLeft();

uint8_t angle_direction_old = 0;
uint8_t angle_counter=0;
uint8_t stop=0;
uint16_t distance, distanceL, distanceR;
int16_t dutyR, dutyL;
int16_t angle;
uint8_t step = 1;
int32_t positionR = 0, positionL=0;
uint8_t addressLeft = I2CENCODER_STARTING_ADDRESS;
uint8_t addressRight = I2CENCODER_STARTING_ADDRESS+2;
uint16_t counter_duty = 0;
```

```
uint16_t counter_ADC = 0;
uint16_t IRValueL, IRValueC, IRValueR;

uint8_t right_stopped = 0, left_stopped = 0;
uint8_t going_forward = 0, turning_left = 0, turning_right = 0;
uint8_t leftAdjustments=0,rightAdjustments=0;

/***Navigation Parameters***/
uint8_t speed_ToStage3R = 110;
uint8_t speed_ToStage3L = 110;


int main(void)
{

        MCUCR |= 1<<JTD;        //disable JTAG to enable PORTC
        MCUCR |= 1<<JTD;

        //Initialization
        ADC_init();
        initialize_timer();
        interrupt_init();
        Initialize_PWM_Interrupt();

        ADC_start();
        twi_init();
        encoder_initL();
        encoder_initR();

        //Initialize Motor Control
        DDRC |= 0xFC;
        PORTC &= 0x03;

        //Initialize Bumpers
        DDRB &= ~(0x03);
        PORTB |= 0x03;
        PINB &= ~(0x03);

        //Initialize Interlock
        IL_init();

        //Initialize dart launcher
        DDRB |= (1<<3);
        PORTB &= ~(1<<3);

        sei();      //Enable interrupts

        //Initialize ultrasonic sensors and send first pulse
        DDRD |= 0x30;   //3&4 input, 5&6 output
        PORTD &= ~(0x30);
```

```
	_delay_us(2);
	PORTD |= 0x30;
	_delay_us(10);		//output pulse
	PORTD &= ~(0x30);

	_delay_ms(1000);	//delayed start to allow time to get out of the way

	Set_Duty_CycleR(80); //initial speed (slow)
	Set_Duty_CycleL(80); //Duty cycle = 80/255 = 31%

	Set_Motor('R','S');
	Set_Motor('L','S');
	_delay_ms(1);
	Set_Motor('R','F');
	Set_Motor('L','F');

	while(1)
	{
	//Get ultrasonic distance sensor data every time
	distanceL = getDistanceL();
	distanceR = getDistanceR();
	if (distanceL > distanceR) {		//use closer sensor reading
		distance = distanceR;
	}
	else {
		distance = distanceL;
	}
	angle = getDistanceL() - getDistanceR();
	if (step == 1)	//Go forward to Task 1 until wall is hit
	{
		if (distance > 2 && distance < 25 && OCR2A < 250)		//Ram wall when close
		{
		Set_Duty_CycleL(250);
		Set_Duty_CycleR(250);
		}

		//If at least one bumper is hit, wait .5s, then stop. This avoids infinite loop from partially
pressed bumper.
		if (((PINB | 0xFD) == 0xFD) || ((PINB | 0xFE) == 0xFE) || ((PINB | 0xFC) == 0xFC))
		{
		_delay_ms(500);
		Set_Motor('L','S');
		Set_Motor('R','S');
		IL_reset();
		step = 2;
		}
	}
	else if (step == 2)		//Wait for task1
	{
		if (((PINB | 0xFD) == 0xFD) || ((PINB | 0xFE) == 0xFE) || ((PINB | 0xFC) == 0xFC))
```

```
		{
		_delay_ms(500);
		Set_Motor('R','S');
		Set_Motor('L','S');
		}

		if (IL_getTask1Status() > 0)		//task 1 status signal received
		{
		_delay_ms(5);
		if (IL_getTask1Status()==1) //success
		{
			IL_reset();
			zero(addressLeft<<1);	//reset encoders
			zero(addressRight<<1);
			_delay_ms(2000);		//allow time for Task 1 arm to retract
			Set_Duty_CycleL(speed_ToStage3L);
			Set_Duty_CycleR(speed_ToStage3R);
			Set_Motor('L','B');			//move toward stage 3
			Set_Motor('R','B');
			step=3;
		}
		else	//failure
		{
			IL_reset();
			step = 99;
		}
		}
}
//Make 3 adjustments to right, then 4 to left until Task 1 succeeds at least partially.
//If no success after 7 adjustments, the robot will go backwards toward stage 3 anyway.
else if (step == 99) //adjustment
{
		if (rightAdjustments<3) {
		nav_Task1AdjustRight();
		rightAdjustments++;
		step = 2;
		}
		else if (leftAdjustments<4) {
		nav_Task1AdjustLeft();
		leftAdjustments++;
		step = 2;
		}
		else {
		zero(addressLeft<<1);	//reset encoders
		zero(addressRight<<1);
		_delay_ms(2000);
		Set_Duty_CycleL(speed_ToStage3L);
		Set_Duty_CycleR(speed_ToStage3R);
		Set_Motor('L','B');
		Set_Motor('R','B');
```

```
            step=3;
            }
}
else if (step == 3)           //Move backwards toward stage 3 using encoders as feedback
{
            positionL = encoderGetRawPositionL();
            positionR = encoderGetRawPositionR();

            if (positionR<-1727 && right_stopped == 0) {
            Set_Motor('R','S');
            right_stopped=1;
            }
            if (positionL < -1727 && left_stopped == 0) {
            Set_Motor('L','S');
            left_stopped=1;
            }
            if (positionL<-1727 && positionR<-1727) {
            step=4;
            }
}

else if (step == 4)  //Arrival at Task 3
{
            Set_Motor('L','S');
            Set_Motor('R','S');
            _delay_ms(10);
            Set_Duty_CycleL(20); //Lock wheels to prevent sliding while turning knob
            Set_Duty_CycleR(20);
            Set_Motor('L','B');
            Set_Motor('R','B');
            _delay_ms(50000);         //Wait for Task 3. Controlled by ATmega328P
            Set_Motor('R','S');
            Set_Motor('L','S');
            _delay_ms(10);
            Set_Duty_CycleL(90);
            Set_Duty_CycleR(90);
            Set_Motor('L','F');
            Set_Motor('R','F');
            zero(addressLeft<<1);     //reset encoders
            zero(addressRight<<1);
            step = 5;
}
else if (step == 5)  //Drive forward to center
{
            positionL = encoderGetRawPositionL();
            positionR = encoderGetRawPositionR();
            if (positionR>650 && positionL>650)
            {
            Set_Motor('L','S');
            Set_Motor('R','S');
```

```
                _delay_ms(1);
                Set_Duty_CycleL(180);    //Turn right 90 degrees
                Set_Duty_CycleR(40);
                Set_Motor('L','F');
                Set_Motor('R','B');

                zero(addressLeft<<1);
                while(positionL<970) //~1.75 rotations of left wheel
                {
                        positionL = encoderGetRawPositionL();
                }

                Set_Motor('L','S');
                Set_Motor('R','S');
                _delay_ms(1);
                Set_Duty_CycleR(90);
                Set_Duty_CycleL(130);
                Set_Motor('L','F');
                Set_Motor('R','F');
                step = 6;
                }
        }
        else if (step == 6)          //Drive forward toward Task 2 until wall is hit
        {
                if (distance > 40 && distance < 70)          // Slow down to avoid hitting wall and activating
touch sensor
                {
                Set_Duty_CycleL(80);
                Set_Duty_CycleR(80);
                }

                if (distance > 2 && distance < 45)           // Stop far enough away from wall
                {
                Set_Motor('L','S');
                Set_Motor('R','S');
                task2Process();
                Initialize_PWM_Interrupt();
                _delay_ms(2000);
                Set_Duty_CycleL(105);
                Set_Duty_CycleR(95);
                zero(addressLeft<<1);
                zero(addressRight<<1);
                positionL=0;
                positionR=0;
                right_stopped=0;
                left_stopped=0;
                Set_Motor('R','B');
                Set_Motor('L','B');
                step = 7;
                }
```

```
		}
		else if (step==7)		//Drive backward toward Stage 4
		{
			positionR=encoderGetRawPositionR();
			positionL=encoderGetRawPositionL();
			if (positionR < -2175 && right_stopped == 0) {
			Set_Motor('R','S');
			right_stopped=1;
			}
			if (positionL < -2175 && left_stopped == 0) {
			Set_Motor('L','S');
			left_stopped=1;
			}
			if (positionR<-2175 && positionL<-2175)
			{
			_delay_ms(1000);
			PORTB |= (1<<3);		//Fire darts!
			_delay_ms(10000);		//Delay to allow motor to pull pins
			PORTB &= ~(1<<3);		//Stop motor
			step=8;
			}
		}

		}
}

//Move robot to the right to reattempt Task 1
void nav_Task1AdjustRight()
{
		Set_Motor('L','S');
		Set_Motor('R','S');
		_delay_ms(100);
		zero(addressRight<<1);
		zero(addressLeft<<1);
		positionL=0;
		positionR=0;
		Set_Motor('R','B');
		Set_Motor('L','B');
		while (positionL > -80 && positionR > -80)
		{
		positionL=encoderGetRawPositionL();
		positionR=encoderGetRawPositionR();
		}
		Set_Motor('R','S');
		Set_Motor('L','S');
		Set_Duty_CycleL(200);
		Set_Duty_CycleR(30);
		_delay_ms(100);
		zero(addressRight<<1);
		zero(addressLeft<<1);
```

```
	positionL=0;
	positionR=0;
	Set_Motor('L','F');
	Set_Motor('R','B');
	while (positionL<100)
	{
	positionL=encoderGetRawPositionL();
	if (((PINB | 0xFD) == 0xFD) || ((PINB | 0xFE) == 0xFE) || ((PINB | 0xFC) == 0xFC)) {
		_delay_ms(500);
		break;
	}
	}
	Set_Motor('L','S');
	Set_Motor('R','S');
	Set_Duty_CycleR(200);
	Set_Duty_CycleL(30);
	_delay_ms(100);
	zero(addressRight<<1);
	zero(addressLeft<<1);
	positionL=0;
	positionR=0;
	Set_Motor('R','F');
	Set_Motor('L','B');
	while (positionR<190)
	{
	positionR=encoderGetRawPositionR();
	if (((PINB | 0xFD) == 0xFD) || ((PINB | 0xFE) == 0xFE) || ((PINB | 0xFC) == 0xFC)) {
		_delay_ms(500);
		break;
	}
	}
	Set_Motor('L','S');
	Set_Motor('R','S');
	_delay_ms(1000);
	Set_Duty_CycleL(250);
	Set_Duty_CycleR(250);
	Set_Motor('L','F');
	Set_Motor('R','F');
}

//Move robot to the left to reattempt Task 1
void nav_Task1AdjustLeft()
{
	Set_Motor('L','S');
	Set_Motor('R','S');
	_delay_ms(100);
	zero(addressRight<<1);
	zero(addressLeft<<1);
	positionL=0;
	positionR=0;
```

```
Set_Motor('R','B');
Set_Motor('L','B');
while (positionL > -75 && positionR > -75)
{
positionL=encoderGetRawPositionL();
positionR=encoderGetRawPositionR();
}
Set_Motor('R','S');
Set_Motor('L','S');
Set_Duty_CycleR(200);
Set_Duty_CycleL(30);
_delay_ms(100);
zero(addressRight<<1);
zero(addressLeft<<1);
positionL=0;
positionR=0;
Set_Motor('R','F');
Set_Motor('L','B');
while (positionR<120)
{
positionR=encoderGetRawPositionR();
if (((PINB | 0xFD) == 0xFD) || ((PINB | 0xFE) == 0xFE) || ((PINB | 0xFC) == 0xFC)) {
            _delay_ms(500);
            break;
}
}
Set_Motor('L','S');
Set_Motor('R','S');
Set_Duty_CycleL(200);
Set_Duty_CycleR(30);
_delay_ms(100);
zero(addressRight<<1);
zero(addressLeft<<1);
positionL=0;
positionR=0;
Set_Motor('L','F');
Set_Motor('R','B');
while (positionL<185)
{
positionL=encoderGetRawPositionL();
if (((PINB | 0xFD) == 0xFD) || ((PINB | 0xFE) == 0xFE) || ((PINB | 0xFC) == 0xFC)) {
            _delay_ms(500);
            break;
}
}
Set_Motor('R','S');
Set_Motor('L','S');
_delay_ms(1000);
Set_Duty_CycleL(250);
Set_Duty_CycleR(250);
```

```
        Set_Motor('L','F');
        Set_Motor('R','F');
}
```

A2.2 Motor Control

```
/*
* Motor_PWM.c
*
* Created: 11/8/2016 11:43:31 PM
* Author: Kendall and Daniel
* IEEE Robotic Competition Code
*/

 #define F_CPU 8000000L
 #include <util/delay.h>

#include "Motor_PWM.h"

uint16_t enable_pinL = 0x00;
uint16_t enable_pinR = 0x00;
uint8_t motor_pins = 0x00;
uint8_t duty_cycle_oldL;
uint8_t duty_cycle_oldR;


void Initialize_PWM_Interrupt() {

        // Setup Hardware
//      MOTOR_DDR = MOTOR_DDR|(MOTOR_BITS); // Output
//      MOTOR_PORT = MOTOR_PORT&(~MOTOR_BITS); // Set to LOW

        OCR2A = 0; //  PWM duty cycle
        OCR2B = 0;

        // Phase Correct PWM mode, TOP set by OCRA
        TCCR2A = 1<<WGM20;
        TCCR2B = 0<<WGM22 | 1<<CS22;
        TIMSK2 = 1<<OCIE2A | 1<<OCIE2B;                 //compare mode
        // OC0A is cleared on compare match when counting up, set on compare match when counting down

        // Phase Correct PWM Mode
        // Clock Divider of 8 - Assuming 8.0 MHz clock on Phase Correct PWM Mode, this means that the
PWM will operate at a frequency of 1953 Hz
        //TCCR1B = (0<<FOC1A)|(0<<FOC1B)|(0<<WGM12)|(0<<CS12)|(1<<CS11)|(0<<CS10);



        //
        //TIMSK0 = (0<<TOIE0)|(1<<OCIE0A);
```

```
//
        //sei();
}

void Set_Duty_CycleL(uint8_t duty_cycleL) {
        OCR2A = duty_cycleL;
 }

void Set_Duty_CycleR(uint8_t duty_cycleR)
{
        OCR2B = duty_cycleR;
}

//Function to set either wheel to forward, backward, or stopped. Uses ASCII characters for easy reading
void Set_Motor(uint8_t motor, uint8_t direction) {
        duty_cycle_oldL = OCR2A;
        duty_cycle_oldR = OCR2B;
        OCR2A=0x00;
        OCR2B=0x00;
        TCCR2B=0x00;
        TIMSK2=0x00;
        TCCR2A=0x00;
        if (motor == 'L')               //Left wheel
        {
        if (direction == 'F') {         //Forward
                MOTOR_PORT &= ~(0x20);
                MOTOR_PORT |= 0x10;
                enable_pinL |= 0x40;
                MOTOR_PORT &= ~(0x40);
        }
        else if (direction == 'B') {  //Backward
                MOTOR_PORT &= ~(0x10);
                MOTOR_PORT |= 0x20;
                enable_pinL |= 0x40;
                MOTOR_PORT &= ~(0x40);
        }
        else if (direction == 'S') {  //Stop
                enable_pinL &= ~(0x40);
                MOTOR_PORT &= ~(0x40);
        }
        }
        else if (motor == 'R')        //Right wheel
        {
        if (direction == 'F') {         //Forward
                MOTOR_PORT &= ~(0x08);
                MOTOR_PORT |= 0x04;
                enable_pinR |= 0x80;
                MOTOR_PORT &= ~(0x80);
        }
        else if (direction == 'B') {  //Backward
```

```
                MOTOR_PORT &= ~(0x04);
                MOTOR_PORT |= 0x08;
                enable_pinR |= 0x80;
                MOTOR_PORT &= ~(0x80);
        }
        else if (direction == 'S') {  //Stop
                enable_pinR &= ~(0x80);
                MOTOR_PORT &= ~(0x80);
        }
        }

        Initialize_PWM_Interrupt();
        OCR2A=duty_cycle_oldL;
        OCR2B=duty_cycle_oldR;
}

 ISR(TIMER2_COMPA_vect) {
        MOTOR_PORT ^= enable_pinL;
 }

 ISR(TIMER2_COMPB_vect) {
        MOTOR_PORT ^= enable_pinR;
 }
```

A2.3 I2C Library

```
//I2C communication
//Cameron McSweeney
//9/25/2016

#include <util/delay.h>
#include <string.h>
#include <avr/interrupt.h>
#include <avr/io.h>
/*
void twi_init(void);
void twi_start(void);
void twi_stop(void);
void twi_write(uint8_t);
uint8_t twi_read_ack(void);
uint8_t twi_get_status(void);
*/

void twi_init(void){
        TWSR = 0x00;
        TWBR = 0xff;
        TWCR = (1<<TWEN);
}
```

```
void twi_start(void){
        TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
        while((TWCR & (1<<TWINT)) == 0);
}

void twi_stop(void){
        TWCR = (1<<TWINT) | (1<<TWSTO) | (1<<TWEN);
}

void twi_write(uint8_t data){
        TWDR = data;
        TWCR = (1<<TWINT) | (1<<TWEN);
        while((TWCR & (1<<TWINT)) == 0);
}

uint8_t twi_read_ack(void){
        TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);
        while((TWCR & (1<<TWINT)) == 0){
        }
        _delay_us(2);
        return TWDR;
}

uint8_t twi_read_nack(void){
        TWCR = (1<<TWINT) | (1<<TWEN);
        while((TWCR & (1<<TWINT)) == 0){
        }

        _delay_us(2);
        return TWDR;
}

uint8_t twi_get_status(void){
        uint8_t status;
        status = TWSR & 0xF8;
        return status;
}
```

A2.4 Motor Encoder Library

```
/*
 * Encoder.c
 *
 * Created: 2/23/2017 10:15:56 AM
 * Author: Kendall Knapp
 * Purpose: Read data from integrated motor encoders
 */
#define F_CPU 8000000L
#include <util/delay.h>
```

```
#include <avr/io.h>
#include "Encoder.h"
#include "I2C.h"

uint8_t addressL = I2CENCODER_STARTING_ADDRESS;            //this address must be initialized first
uint8_t addressR = I2CENCODER_STARTING_ADDRESS+2;
float rotation_factor = MOTOR_393_TORQUE_ROTATIONS;
float time_delta = MOTOR_269_TIME_DELTA;
int ticks = TICKS;

//Initialize encoder for left wheel - must be first because it is connected directly to microcontroller
void encoder_initL()
{



write_byte(I2CENCODER_DEFAULT_ADDRESS,I2CENCODER_ADDRESS_REGISTER,addressL<<1);

        twi_start();
        twi_write(addressL<<1);
        twi_write(I2CENCODER_UNTERMINATE_REGISTER);
        twi_stop();

        zero(addressL<<1);
}

//Initialize right wheel's encoder
void encoder_initR()
{

write_byte(I2CENCODER_DEFAULT_ADDRESS,I2CENCODER_ADDRESS_REGISTER,addressR<<1);

        zero(addressR<<1);
}

//Distance traveled in ticks
int32_t encoderGetRawPositionL()
{
        // TODO: Deal with the two extra bytes
        int32_t position = read_4bytes(addressL<<1, I2CENCODER_POSITION_REGISTER);

        return -position;  //Left encoder is reversed
}

int32_t encoderGetRawPositionR()
{
        // TODO: Deal with the two extra bytes
        int32_t position = read_4bytes(addressR<<1, I2CENCODER_POSITION_REGISTER);

        return position;
```

```
}

//Reset encoder
void zero(uint8_t device_address) {
//          write_byte(I2CENCODER_STARTING_ADDRESS, address, I2CENCODER_ZERO_REGISTER);
            twi_start();
            twi_write(device_address); //device address
            twi_write(I2CENCODER_ZERO_REGISTER);        //data address
            twi_stop();
}

void write_byte(uint8_t device_address, uint8_t register_address, uint8_t data)
{
            twi_start();
            twi_write(device_address); //device address
            twi_write(register_address);          //data address
            twi_write(data);   //data to be sent
            twi_stop();
}

int32_t read_4bytes(uint8_t device_address, uint8_t register_address)
{
            twi_start();
            twi_write(device_address);          //device address
            twi_write(register_address);        //data address
            twi_stop();

            twi_start();
            twi_write(device_address+1);                        //read from the RTC
            int32_t data = 0;
            data |= twi_read_ack() << 8;
            data |= twi_read_ack();
            data |= twi_read_ack() << 24;
            data |= twi_read_nack() << 16;
            twi_stop();

            return data;
}
```

A2.5 Ultrasonic Sensors - Timers

```
/*
 * bios_timer_int.c
 *
 * Date: 4/1/2017
 * Author: Kendall Knapp
 * Purpose: Read data from integrated motor encoders
 */
```

```
#define F_CPU 8000000
#include <util/delay.h>

#include "bios_timer_int.h"
#include "interrupt_ext.h"
#include <avr/io.h>
#include <avr/interrupt.h>

#define xtal 8000000L
uint8_t counter = 0;
uint16_t distance0 = 0;
uint16_t distance1 = 0;
uint8_t startup = 1;
int16_t angle = 0;
uint16_t overflow = 0;
uint8_t negative = 0;


void initialize_timer()
{
	//Timer0 - Send output pulse for ultrasonic sensors
	TCNT0 = 0;
	TCCR0A = 0x00;
	TCCR0B = 1<<CS00 | 1<<CS01 | 0<<CS02;		//64 prescaler
	TIMSK0 = (1<<OCIE0A)|(1<<TOIE0);

	//Timer1 - Measure time between ultrasonic output and echo
	TCNT1 = 0;
	TCCR1A = 0x00;
	TCCR1B = 1<<CS11 | 1<<CS10; //64 prescalar
	TIMSK1 = 1<<TOIE1;   //enable overflow interrupt
}

ISR (TIMER0_COMPA_vect)
{
	if (counter>50)    // 250 = .25s
	{
	counter=0;

	PORTD |= 0x30;
	_delay_us(10);		//output pulse
	PORTD &= ~(0x30);
	}
	else
	{
	counter++;
	}

	EIFR = 0x0F;  //Clear any interrupt requests
```

```
}

ISR (TIMER1_OVF_vect)
{
	overflow++;
}

uint16_t getOverflow()
{
	return overflow;
}
```

A2.6 Ultrasonic Sensors - External Interrupts

```
/*
 * interrupt_ext.c
 *
 * Created: 9/3/2015 4:07:39 PM
 *  Author: krknapp
 */
#define F_CPU 8000000L

#include "interrupt_ext.h"
#include "bios_timer_int.h"
#include <avr/interrupt.h>
#include <util/delay.h>

uint8_t change_to_falling_INT0 = 1;
uint8_t change_to_falling_INT1 = 1;
uint16_t d0=0,d1=0;
uint16_t Time0 = 0;
uint16_t Time1 = 0;
uint16_t Time0_old = 0;
uint16_t Time1_old = 0;
uint16_t Time0_overflow_old = 0;
uint16_t Time1_overflow_old = 0;
uint8_t task1_status=0;


void interrupt_init()
{
	EIMSK |= 1<<INT0 | 1<<INT1;
	EICRA |= 1<<ISC01 | 1<<ISC00 | 1<<ISC11 | 1<<ISC10;		//rising edge
}

uint16_t getDistanceL()
{
	return d0;
}
```

```
uint16_t getDistanceR()
{
	return d1;
}


ISR(INT0_vect)
{
	PORTB ^=0x10;
	if (change_to_falling_INT0==1)
	{
	EICRA |=  1<<ISC01;	//falling edge
	EICRA &= ~(1<<ISC00);
	change_to_falling_INT0=0;
	Time0_old = TCNT1;
	Time0_overflow_old = getOverflow();
	}
	else
	{
	if (Time0_overflow_old != getOverflow())
	{
		Time0 = 65536 - Time0_old + TCNT1;
	}
	else
	{
		Time0 = TCNT1 - Time0_old;
	}
	d0 = Time0/1.85;		//.1 in
	EICRA |= 1<<ISC01 | 1<<ISC00;		//rising edge
	change_to_falling_INT0 = 1;
	}

//	EIFR = 0x0F;  //Clear any interrupt requests
}

ISR(INT1_vect)
{
	if (change_to_falling_INT1==1)
	{
	EICRA |=  1<<ISC11;	//falling edge
	EICRA &= ~(1<<ISC10);
	change_to_falling_INT1=0;
	Time1_old = TCNT1;
	Time1_overflow_old = getOverflow();
	}
	else
	{
	if (Time1_overflow_old != getOverflow())
	{
```

```
                Time1 = 65536 - Time1_old + TCNT1;
        }
        else
        {
                Time1 = TCNT1 - Time1_old;
        }
        d1 = Time1/1.85;            //.1 in
        EICRA |= 1<<ISC11 | 1<<ISC10;              //rising edge
        change_to_falling_INT1 = 1;
        }

//      EIFR = 0x0F;  //Clear any interrupt requests
}

ISR(INT2_vect){
        task1_status++;
}
```

A2.7 IR Sensors - ADC

```
/*
 * ADC.c
 *
 * Created: 3/2/2017 1:59:17 PM
 * Author: Kendall Knapp
 * Purpose: Convert analog IR sensor values to digital
 */


#include "ADC.h"

uint16_t IRValue0,IRValue1,IRValue2;
uint8_t ADC_pin=0;

void ADC_init()
{
        ADMUX = 1<<REFS0;
        ADCSRA = 1<<ADEN | 1<<ADIE| 1<<ADPS0 | 1<<ADPS1 | 1<<ADPS2;
}

void ADC_start()
{
        ADCSRA |= 1<<ADSC;
}

uint16_t getIRValue0()
{
        return IRValue0;
}
```

```
uint16_t getIRValue1()
{
	return IRValue1;
}

uint16_t getIRValue2()
{
	return IRValue2;
}

ISR(ADC_vect)
{
	if (ADC_pin==0)
	{
	IRValue0 = ADCL;
	IRValue0 |= ADCH<<8;
	ADMUX = 1<<REFS0 | 1<<MUX0;
	ADC_pin = 1;
	}
	else if (ADC_pin==1)
	{
	IRValue1 = ADCL;
	IRValue1 |= ADCH<<8;
	ADMUX = 1<<REFS0 | 1<<MUX1;
	ADC_pin = 2;
	}
	else if (ADC_pin==2)
	{
	IRValue2 = ADCL;
	IRValue2 |= ADCH<<8;
	ADMUX = 1<<REFS0;
	ADC_pin = 0;
	}

	ADC_start();
}
```

A2.8 IR Sensors - Line Adjustment

```
//Move robot horizontally to the right to align with navigation stripe
void nav_LineAdjustRight()
{
	Set_Motor('L','S');
	Set_Motor('R','S');
	_delay_ms(100);
	zero(addressRight<<1);
	zero(addressLeft<<1);
	positionL=0;
```

```
positionR=0;
Set_Motor('R','B');
Set_Motor('L','B');
while (positionL > -40 && positionR > -40)
{
positionL=encoderGetRawPositionL();
positionR=encoderGetRawPositionR();
}
Set_Motor('R','S');
Set_Motor('L','S');
Set_Duty_CycleL(200);
Set_Duty_CycleR(50);
_delay_ms(100);
zero(addressRight<<1);
zero(addressLeft<<1);
positionL=0;
positionR=0;
Set_Motor('L','F');
Set_Motor('R','B');
while (positionL<120)
{
positionL=encoderGetRawPositionL();
if (((PINB | 0xFD) == 0xFD) || ((PINB | 0xFE) == 0xFE) || ((PINB | 0xFC) == 0xFC)) {
            _delay_ms(500);
            break;
}
}
Set_Motor('L','S');
Set_Motor('R','S');
Set_Duty_CycleR(200);
Set_Duty_CycleL(50);
_delay_ms(100);
zero(addressRight<<1);
zero(addressLeft<<1);
positionL=0;
positionR=0;
Set_Motor('R','F');
Set_Motor('L','B');
while (positionR<160)
{
positionR=encoderGetRawPositionR();
if (((PINB | 0xFD) == 0xFD) || ((PINB | 0xFE) == 0xFE) || ((PINB | 0xFC) == 0xFC)) {
            _delay_ms(500);
            break;
}
}
Set_Motor('L','S');
Set_Motor('R','S');
_delay_ms(1000);
Set_Duty_CycleL(80);
```

```
	Set_Duty_CycleR(80);
	Set_Motor('L','F');
	Set_Motor('R','F');
}

//Move robot horizontally to the left to align with navigation stripe
void nav_LineAdjustLeft()
{
	Set_Motor('L','S');
	Set_Motor('R','S');
	_delay_ms(100);
	zero(addressRight<<1);
	zero(addressLeft<<1);
	positionL=0;
	positionR=0;
	Set_Motor('R','B');
	Set_Motor('L','B');
	while (positionL > -40 && positionR > -40)
	{
	positionL=encoderGetRawPositionL();
	positionR=encoderGetRawPositionR();
	}
	Set_Motor('R','S');
	Set_Motor('L','S');
	Set_Duty_CycleL(50);
	Set_Duty_CycleR(200);
	_delay_ms(100);
	zero(addressRight<<1);
	zero(addressLeft<<1);
	Set_Motor('R','F');
	Set_Motor('L','B');
	while (positionR<100)
	{
	positionR=encoderGetRawPositionR();
	if (((PINB | 0xFD) == 0xFD) || ((PINB | 0xFE) == 0xFE) || ((PINB | 0xFC) == 0xFC)) {
		_delay_ms(500);
		break;
	}
	}
	Set_Motor('L','S');
	Set_Motor('R','S');
	Set_Duty_CycleL(200);
	Set_Duty_CycleR(50);
	_delay_ms(100);
	zero(addressRight<<1);
	zero(addressLeft<<1);
	positionL=0;
	positionR=0;
	Set_Motor('L','F');
	Set_Motor('R','B');
```

```
        while (positionL<125)
        {
        positionL=encoderGetRawPositionL();
        if (((PINB | 0xFD) == 0xFD) || ((PINB | 0xFE) == 0xFE) || ((PINB | 0xFC) == 0xFC)) {
                _delay_ms(500);
                break;
        }
        }
        Set_Motor('L','S');
        Set_Motor('R','S');
        _delay_ms(1000);
        Set_Duty_CycleL(80);
        Set_Duty_CycleR(80);
        Set_Motor('L','F');
        Set_Motor('R','F');
}
```

A2.9 Task 1 Status Signal

```
void IL_reset()
{
        task1_status=0;
}

void IL_init(void){
        //set up INT2 to receive IL info from other uC
        DDRB &= ~(1<<2);
        PORTB |= ~(1<<2);

        EIMSK |= (1<<INT2);
        EICRA |= (1<<ISC21);
        EICRA |= (1<<ISC20); //trigger interrupt on rising edge of INT2
}
```

A3. Appendix C - Task 1 Code

This code was implemented on an ATMega328P microcontroller.

A3.1 Task 1 Control

```
/*
 * ComponentID.c
 *
 * Created: 11/1/2016 10:46:16 AM
 * Author : cmcsweeney
 */
#define F_CPU 1000000UL

#include <stdint.h>
#include <avr/io.h>
#include <util/delay.h>
#include <string.h>
#include <avr/interrupt.h>
#include "LCD.h"
#include "adc.h"
//#include "stepper.h"
#include "componentID.h"
#include "Servo_PWM.h"


/****************** Port Setup ******************/
/*
#define ADC_PORT        PORTC  //port setup for ADC
#define ADC_DDR                 DDRC
#define ADC_PIN                 PINC
#define ADC_PIN_NUM             0
#define ADC_LEFT        1       //left adjust result T/F
#define ADC_CHAN        0       //ADC Channel used
#define ADC_REF                 1       //Reference voltage source - 0: AREF; 1:AVCC; 3: internal 2.56V
*/
/*
#define STEP_PORT       PORTC  //port setup for Step output
#define STEP_DDR        DDRC
#define STEP_PIN_NUM 1

#define SWITCH_DDR              DDRD   //port setup for Switch circuit. Assumes use of bottom 5 pins (pins
0-4)
#define SWITCH_PORT             PORTD
*/
/****************** Constants ******************/
#define MOTOR_EN_REG            PORTD
#define TASK1_EN                7
```

```
#define TASK3_EN                6
#define MOTOR_STEPS             240     //number of steps to advance motor
#define MOTOR_DELAY             5       //number of ms wait between each step
#define TASK_3_ROT_STEPS        210

#define PWM_PIN_NUM             5



/****************** Function Statements ******************/
void motor_init(void);          //initializes the stepper motor ports
void motor_step(void);          //steps motor one full cycle. Does not control direction
void motor_out_ccw(uint8_t);            //moves motor given number of steps outward
void motor_in_cw(uint8_t);      //moves motor given number of steps inward
void motor_idle(void);          //sets motor outputs to 0 to conserve power
void enable_task1(void);
void disable_task1(void);
void enable_task3(void);
void disable_task3(void);
void task_3_process(void);


/****************** Global Variables ******************/

extern uint8_t adc_cycles;
extern uint8_t adc_avg;
extern uint8_t adc_avg1_flag;
extern uint8_t adc_avg1[5];
extern uint8_t adc_avg2[5];

extern uint8_t wait_for_go;

uint8_t task1_attempts = 0;
//extern int16_t diffs[5];
//extern int16_t mid_avg[5];
//extern uint16_t component_code;

int main(void)
{

        sei();
        lcd_init();

        DDRD |= (1<<PWM_PIN_NUM) | (1<<TASK1_EN) | (1<<TASK3_EN);       //set PWM and motor
Enable pins as output
        PORTD &= ~(1<<PWM_PIN_NUM);

        adc_init();
        step_init();
```

```
        motor_init();       //initializations

                                //set PWM output high

//      PCICR |= (1<<PCIE0); //enable IL interrupt
//      PCMSK0 = (1<<PCINT7);
        DDRB |= (1<<7);                 //IL initialization
        PORTB &= ~(1<<7);

        _delay_ms(15000);

        while(1){           //while there is no contact
        enable_task1();
        motor_out_ccw(MOTOR_STEPS);
        disable_task1();                //push motor out
        _delay_ms(100);
        perform_test_noLCD();

        if(component_code==0x5B6D){     //if ADC pin is still high (no contact)
                component_code=0;
                enable_task1();
                motor_in_cw(MOTOR_STEPS);
                disable_task1();                //pull motor back in
                task1_attempts += 1;            //increment task attempts

                PORTB |= 1<<7;
                _delay_ms(1);                   //send ADJUST signal (triggers interrupt twice)
                PORTB &= ~(1<<7);
                _delay_ms(1);
                PORTB |= 1<<7;
                _delay_ms(1);
                PORTB &= ~(1<<7);
                _delay_ms(5000);
        }
        else {
                break;
        }

        }

        perform_test();

        enable_task1();
        motor_in_cw(MOTOR_STEPS);
        disable_task1();

        PORTB |= 1<<7;
        _delay_ms(1);                   //send TASK_COMPLETE signal (triggers interrupt once)
        PORTB &= ~(1<<7);
}
```

```
void motor_init(void){
	DDRC |= (1<<PINC2) | (1<<PINC3) | (1<<PINC4) | (1<<PINC5);
	PORTC |= (1<<PINC2) | (1<<PINC3) | (1<<PINC4) | (1<<PINC5);
}

void motor_step(void){
	PORTC ^= 0x0C;
	_delay_ms(MOTOR_DELAY);
	PORTC ^= 0x30;
	_delay_ms(MOTOR_DELAY);
}

void motor_out_ccw(uint8_t steps){
	PORTC |= 0x18;
	PORTC &= ~(0x24);

	for(uint8_t i = 0; i<steps; i++){
	motor_step();
	}
//	motor_idle();
}

void motor_in_cw(uint8_t steps){
	PORTC |= 0x28;
	PORTC &= ~(0x14);

	for(uint8_t i = 0; i<=steps; i++){
	motor_step();
	}
//	motor_idle();
}

void motor_idle(void){
	PORTC &= ~((1<<PINC2) | (1<<PINC3));
	PORTC &= ~((1<<PINC4) | (1<<PINC5));
}

void enable_task1(void){
	MOTOR_EN_REG |= 1<<TASK1_EN;
}

void disable_task1(void){
	MOTOR_EN_REG &= ~(1<<TASK1_EN);
}
```

A3.2 ADC Library

```
/*
```

```
* adc.c
*
* Created: 3/7/2017 10:28:09 AM
*  Author: cmcsweeney
*/

#include "adc.h"
#include "componentID.h"

uint8_t adc_cycles = 8;
uint8_t adc_avg = 0;
uint8_t adc_avg1_flag = 0;
uint8_t adc_avg1[5] = {0};
uint8_t adc_avg2[5] = {0};

int16_t diffs[5] = {0};
int16_t mid_avg[5] = {0};
uint16_t component_code = 0;

void adc_init(void){
	ADC_DDR &= ~(1<<ADC_PIN_NUM);		//setup ADC pin for input
	ADC_PORT &= ~(1<<ADC_PIN_NUM);
	ADMUX = (ADC_LEFT << ADLAR) | (ADC_REF << REFS0) | (ADC_CHAN << MUX0);		//set up
MUX register
	ADCSRA = (1<<ADATE) | (1<<ADIE);		//set up control register for Free Run, do not start
	ADCSRB = 0;
}

void adc_start(void){
	ADCSRA |= (1<<ADEN);		//enable ADC and start conversions
	ADCSRA |= (1<<ADSC);
}

void adc_stop(void){
	ADCSRA &= ~(1<<ADEN);				//disable ADC
}

ISR(ADC_vect){
	if(adc_cycles > 0){		//avg is incomplete
	adc_avg += (ADCH >> 3);		//update the running average
	adc_cycles--;
	}
	else{				//avg is complete
	adc_stop();			//stop ADC

	if(adc_avg1_flag){		//if the first average has already been taken, record second average
		//			adc_avg2 = adc_avg;
		step_low();			//test is over, turn off step
	}
	else{
```

```
                //              adc_avg1 = adc_avg;     //test is halfway complete
        }

        //      adc_avg = 0;            //prepare for another average reading
        adc_cycles = 8;
        }
}
```

A3.3 Component ID Algorithm

```
/*
 * componentID.c
 *
 * Created: 3/7/2017 10:47:36 AM
 *  Author: cmcsweeney
 */

#include "componentID.h"

#define IL_PORT             PORTB
#define IL_PIN              PINB
#define IL_DDR                      DDRB
#define IL_PIN_NUM          7

extern uint8_t adc_cycles;
extern uint8_t adc_avg;
extern uint8_t adc_avg1_flag;
extern uint8_t adc_avg1[5];
extern uint8_t adc_avg2[5];

uint8_t wait_for_go = 0;

void find_component_readings(uint8_t index){
        adc_avg1_flag = 0;

        step_high();
        adc_start();                //start ADC before step input - allows for more accurate/consistent
readings
//      step_high();                //excite the component

        _delay_ms(RESPONSE_DELAY_MS);   //wait for transient response to settle
        adc_avg1[index] = adc_avg;
        adc_avg1_flag = 1;
        adc_avg = 0;

        adc_start();                //read steady-state value

        _delay_ms(RESPONSE_DELAY_MS);
        adc_avg2[index] = adc_avg;
```

```
		adc_avg1_flag = 0;
		adc_avg = 0;


}

void do_avg_math(void){
		for(uint8_t i=0; i<5; i++){
		diffs[i] = adc_avg2[i] - adc_avg1[i];
		mid_avg[i] = (adc_avg1[i] + adc_avg2[i])/2;
		writeIntegerToLCD(mid_avg[i]);
		writeStringToLCD("-");
		writeIntegerToLCD(diffs[i]);
		writeStringToLCD(";");
		}
}

//setup Step pin for output, keep low
void step_init(void){
		STEP_DDR |= (1<<STEP_PIN_NUM);
		step_low();
}

void step_high(void){
		STEP_PORT |= (1<<STEP_PIN_NUM);
}

void step_low(void){
		STEP_PORT &= ~(1<<STEP_PIN_NUM);
}

void IDComponent(uint8_t index){
		if (adc_avg2[index] > (adc_avg1[index] + CAP_CHANGE_THRESH)){		//big increase
		component_code = component_code << 3;
		component_code += 3;
		writeIntegerToLCD(3);
//		writeStringToLCD("Capacitor");
		}
		else if ((adc_avg2[index] + IND_CHANGE_THRESH) < adc_avg1[index]){	//big decrease
		component_code = component_code << 3;
		component_code += 4;
		writeIntegerToLCD(4);
//		writeStringToLCD("Inductor");
		}
		else if (adc_avg2[index] < WIRE_THRESH){					//wire has lowest value
		component_code = component_code << 3;
		component_code += 1;
		writeIntegerToLCD(1);
//		writeStringToLCD("Wire");
		}
```

```
        else if (adc_avg2[index] < DIODE_F_THRESH){                         //forward biased
diode has 2nd lowest value
        component_code = component_code << 3;
        component_code += 5;
        writeIntegerToLCD(5);
//      writeStringToLCD("Diode Forward");
        }
        else if (adc_avg2[index] < RES_THRESH){                             //10k resistor has 3rd
lowest value
        component_code = component_code << 3;
        component_code += 2;
        writeIntegerToLCD(2);
//      writeStringToLCD("Resistor");
        }
        else{                                                       //reverse biased diode has highest value
        component_code = component_code << 3;
        component_code += 5;
        writeIntegerToLCD(5);
//      writeStringToLCD("Diode Reverse");
        }
}

void IDComponent_noLCD(uint8_t index){
        if (adc_avg2[index] > (adc_avg1[index] + CAP_CHANGE_THRESH)){       //big increase
        component_code = component_code << 3;
        component_code += 3;
//      writeIntegerToLCD(3);
        //      writeStringToLCD("Capacitor");
        }
        else if ((adc_avg2[index] + IND_CHANGE_THRESH) < adc_avg1[index]){  //big decrease
        component_code = component_code << 3;
        component_code += 4;
//      writeIntegerToLCD(4);
        //      writeStringToLCD("Inductor");
        }
        else if (adc_avg2[index] < WIRE_THRESH){                            //wire has lowest value
        component_code = component_code << 3;
        component_code += 1;
//      writeIntegerToLCD(1);
        //      writeStringToLCD("Wire");
        }
        else if (adc_avg2[index] < DIODE_F_THRESH){                         //forward biased
diode has 2nd lowest value
        component_code = component_code << 3;
        component_code += 5;
//      writeIntegerToLCD(5);
        //      writeStringToLCD("Diode Forward");
        }
        else if (adc_avg2[index] < RES_THRESH){                             //10k resistor has 3rd
lowest value
```

```
		component_code = component_code << 3;
		component_code += 2;
//		writeIntegerToLCD(2);
		//		writeStringToLCD("Resistor");
		}
		else{                                                    //reverse biased diode has highest value
		component_code = component_code << 3;
		component_code += 5;
//		writeIntegerToLCD(5);
		//		writeStringToLCD("Diode Reverse");
		}
}

void perform_test(void){
		SWITCH_DDR |= 0x1F;

		for (uint8_t i = 0;i<5;i++){
		SWITCH_PORT = (1<<(i));
		find_component_readings(i);
		_delay_ms(RESPONSE_DELAY_MS);
		IDComponent(i);
		_delay_ms(RESPONSE_DELAY_MS);
		}
}

void perform_test_noLCD(void){
		SWITCH_DDR |= 0x1F;

		for (uint8_t i = 0;i<5;i++){
		SWITCH_PORT = (1<<(i));
		find_component_readings(i);
		_delay_ms(RESPONSE_DELAY_MS);
		IDComponent_noLCD(i);
		_delay_ms(RESPONSE_DELAY_MS);
		}
}

void publish_avg_vals(void){
		_delay_ms(RESPONSE_DELAY_MS*5);

		moveCursorToRowColumn(2,0);
		writeIntegerToLCD(adc_avg1[0]);
		writeStringToLCD("-");
		writeIntegerToLCD(adc_avg2[0]);
		writeStringToLCD(";");

		moveCursorToRowColumn(2,8);
		writeIntegerToLCD(adc_avg1[1]);
		writeStringToLCD("-");
		writeIntegerToLCD(adc_avg2[1]);
```

```
		writeStringToLCD(";");

		moveCursorToRowColumn(3,0);
		writeIntegerToLCD(adc_avg1[2]);
		writeStringToLCD("-");
		writeIntegerToLCD(adc_avg2[2]);
		writeStringToLCD(";");

		moveCursorToRowColumn(3,8);
		writeIntegerToLCD(adc_avg1[3]);
		writeStringToLCD("-");
		writeIntegerToLCD(adc_avg2[3]);
		writeStringToLCD(";");

		moveCursorToRowColumn(4,0);
		writeIntegerToLCD(adc_avg1[4]);
		writeStringToLCD("-");
		writeIntegerToLCD(adc_avg2[4]);
		writeStringToLCD(";");
//		new_line();
/*
		for(uint8_t k = 0;k<5;k++){
		writeIntegerToLCD(adc_avg1[k]);
		writeStringToLCD("-");
		writeIntegerToLCD(adc_avg2[k]);
		writeStringToLCD("; ");
		if(k==1){new_line();}
		}
*/
}

/*ISR(PCINT0_vect){
		if(((IL_PIN & (1<<IL_PIN_NUM)) == 0) && ((IL_DDR & (1<<IL_PIN_NUM)) == 0)){	//if IL pin is an
input AND low
		wait_for_go += 1;
		}
}*/
```

A4. Appendix D - Task 2 Code

This code was implemented on an ATMega164A microcontroller.

A4.1 Task 2 Algorithm

```
/*
 * Task2.c
 *
 * Created: 3/27/2017 1:38:55 PM
 * Author: Daniel Hofstetter
 * Purpose: Detects when magnet turns on and rams wall
 */
#include "Servo_PWM.h"
#include "Magnetic_Sensor_Driver.h"
#include "Task2.h"
#include "Motor_PWM.h"

uint8_t match_status = 0; // Status 0 means match is not started.

int16_t on_range_avg = -32761;
int16_t on_range_std_dev = -32761;
int16_t on_high_range = 0;
int16_t on_low_range = 0;
int16_t off_range_avg = -32761;
int16_t off_range_std_dev = -32761;
int16_t off_high_range = 0;
int16_t off_low_range = 0;
int16_t on_threshold;
int16_t off_threshold;
uint16_t timer_count = 0;
uint16_t match_time = 0;
uint16_t timer_mark0 = 0;
uint16_t timer_mark1 = 0;

uint8_t mpu_who = 0;
uint8_t ak_id = 0;
uint8_t duty = SERVO_MIN_DUTY;
//int16_t data[40]; // 5 seconds worth of data samples @ 8Hz
int16_t sample = 0;
uint8_t waiting = 0;
uint16_t timeout = 0;
/***Task 2***/

void task2Process()
{
//          Initialize_Servo_PWM();
            Magnetic_Sensor_Init();
```

```
//          TIMSK2 = (1<<TOIE2); // Used for timing, 61 Hz overflow interrupt

//          Set_Duty_Cycle(SERVO_MAX_DUTY, 0); // make sure servo is in starting position
            _delay_ms(1000);
            CalculateOnRange();  // Read magnetic field when magnet is on
//          Set_Duty_Cycle(SERVO_MIN_DUTY, 0); // Swings toward lightsaber to start match
            Set_Duty_CycleR(250);
            Set_Duty_CycleL(250);
            _delay_ms(10);
            Set_Motor('R','F');
            Set_Motor('L','F');
            _delay_ms(750);
            match_time = 0;
            match_status = 1;
            Set_Duty_CycleR(100);
            Set_Duty_CycleL(100);
            Set_Motor('R','B');
            Set_Motor('L','B');
            _delay_ms(200);
            Set_Motor('R','S');
            Set_Motor('L','S');
            // Wait till magnet turns off
            /*on_high_range = on_range_avg + 3*on_range_std_dev;
            on_low_range = on_range_avg - 3*on_range_std_dev;
            waiting = 1;
            while(waiting&&(timeout<80)) {
            sample = ReadMagneticSensor();
            if ((sample<on_low_range)|(sample>on_high_range)) {
                        waiting = 0;
            }
            timeout++;
            }*/
            // Start Timer
            //match_time = 0;
            //match_status = 1;
//          PORTD|=0x02;
            // Return servo
//          Set_Duty_Cycle(SERVO_MAX_DUTY, 0);
            // Calculate magnetic field range when magnet is off
            CalculateOffRange();

            // Find appropriate threshold based on comparing the std. deviation with the spread between the low
and high averages
            // Find high and low thresholds at 50% between the high and low average, adjusting for std. dev.
            if (on_range_avg>off_range_avg) {
            on_threshold = (on_range_avg-off_range_avg)/2 + off_range_avg + on_range_std_dev;
            //on_threshold = (on_range_avg-off_range_avg)/2 + off_range_avg;
            //off_threshold = (on_range_avg-off_range_avg)/2 + off_range_avg -off_range_std_dev;
            } else {
```

```
            on_threshold = (off_range_avg-on_range_avg)/2 + on_range_avg - on_range_std_dev;
            //on_threshold = (off_range_avg-on_range_avg)/2 + on_range_avg;
            //off_threshold = (off_range_avg-on_range_avg)/2 + on_range_avg + off_range_std_dev;
            }

            while (match_time<14706) {

            sample = ReadMagneticSensor(); // Gather data
            // Assuming that the magnet is presently off (because it will be when the while loop is entered for the
first time), process the magnetic data
            // and respond accordingly
            // Look for on threshold to be crossed (must compensate for either a + or - slope threshold
            if
(((on_range_avg>off_range_avg)&(sample>on_threshold))|((on_range_avg<off_range_avg)&(sample<on_thres
hold))) {
//                      Set_Duty_Cycle(SERVO_MIN_DUTY, 0);
                        //waiting = 1;
                        //while((waiting)&(match_time<1830)) {
                        //sample = ReadMagneticSensor(); // Gather data
                        //if
(((on_range_avg>off_range_avg)&(sample<off_threshold))|((on_range_avg<off_range_avg)&(sample>off_thres
hold))) {
                        //waiting = 0;
                        //Set_Duty_Cycle(SERVO_MIN_DUTY, 0);
                        //}
                        //}
//                      _delay_ms(1500);
//                      Set_Duty_Cycle(SERVO_MAX_DUTY, 0);
                        Set_Duty_CycleR(250);
                        Set_Duty_CycleL(250);
                        Set_Motor('R','F');
                        Set_Motor('L','F');
                        _delay_ms(750);
                        Set_Duty_CycleR(100);
                        Set_Duty_CycleL(100);
                        Set_Motor('R','B');
                        Set_Motor('L','B');
                        _delay_ms(200);
                        Set_Motor('R','S');
                        Set_Motor('L','S');
            }
            }
 }

 /***Task 2***/
 int16_t ReadMagneticSensor() {
             uint8_t d0 = 0xAF;
             uint8_t d1 = 0x27;
             uint8_t status = 0;
             int16_t result = INVALID_MAGNETIC_READING;
```

```
        while ((status&AK_DRDY)==0) {
        Read_AK8963(AK_ST1, &status);
        _delay_ms(1);
        }

        //Read_AK8963_Array(AK_HXL, data, 6);
        Read_AK8963(AK_HYL, &d0);
        Read_AK8963(AK_HYL+1, &d1);
        Read_AK8963(AK_ST2, &status);

        if ((status&AK_HOFL)==0) {
        result = (d1<<8)|d0;
        }
        return result;
}

void CalculateOnRange() {
        int16_t samples[16];
        int16_t sample;
        uint8_t index = 0;
        while (index<16) {
        sample = ReadMagneticSensor();
        if (sample!=INVALID_MAGNETIC_READING) {
                //              SignedNumberToText(sample, &display[0], &display[1], &display[2],
&display[3], &display[4], &display[5]);
                samples[index] = sample;
                index++;
        }
        }
        on_range_avg = Mean(samples, 16);
        on_range_std_dev = StdDev(samples, 16, on_range_avg);
}

void CalculateOffRange() {
        int16_t samples[4];
        int16_t sample;
        uint8_t index = 0;
        while (index<4) {
        sample = ReadMagneticSensor();
        if (sample!=INVALID_MAGNETIC_READING) {
                //              SignedNumberToText(sample, &display[0], &display[1], &display[2],
&display[3], &display[4], &display[5]);
                samples[index] = sample;
                index++;
        }
        }
        off_range_avg = Mean(samples, 4);
        off_range_std_dev = StdDev(samples, 4, off_range_avg);
```

```
}

int16_t Mean(int16_t *data, uint8_t length) {
        int16_t result = 0;
        int32_t temp = 0;
        for (uint8_t i = 0; i<length; i++) {
        int32_t d = data[i];
        temp += d;
        }
        temp = temp/length;
        result = temp;
        return result;
}

int16_t StdDev(int16_t *data, uint8_t length, int16_t mean) {
        int32_t var = 0;
        int32_t temp = 0;
        int16_t result = 0;
        for (uint8_t i = 0; i<length; i++) {
        int32_t d = data[i];
        int32_t m = mean;
        var = var + (d-m)*(d-m);

        }
        var = var/length;

        while ((temp*temp)<var) {
        temp++;
        }
        result = temp;
        return result;
}

ISR(TIMER0_OVF_vect) {
        //timer_count++;
        if (match_status>0) {
        match_time++;
        }
}
```

A4.2 Magnetic Sensor Driver

```
/*
 * Magnetic_Sensor_Driver.c
 *
 * Created: 2/7/2017 4:03:55 PM
 *  Author: Daniel
 */
```

```
#include "Magnetic_Sensor_Driver.h"
#include "I2C.h"

uint8_t Magnetic_Sensor_Init() {

	uint8_t response = 0; // returned at the end of the function
	twi_init(); // initialize TWI Hardware

	//// Allow bypass of the master I2C interface so that the AK8963 magnetometer can be interfaced
directly to the microcontroller
	Write_MPU9250(MPU_INT_PIN_CFG, MPU_EN_BYPASS);

	//// Reset the AK8963
	Write_AK8963(AK_CNTL2, AK_SRST);
	_delay_ms(10);

	//// Set AK8963 Mode and Bit Output: cont. measurement mode and 16-bit
	Write_AK8963(AK_CNTL1, AK_16BIT_OUTPUT|AK_MODE_CONT_8HZ);
	_delay_ms(10);


	return response;
}

uint8_t Write_MPU9250(uint8_t address, uint8_t data) {
	twi_start();
	twi_write((MPU9250_ADDRESS<<1)|ADDRESS_W);
	twi_write(address);
	twi_write(data);
	twi_stop();
	return 0;
}

uint8_t Write_AK8963(uint8_t address, uint8_t data) {
	twi_start();
	twi_write((AK8963_ADDRESS<<1)|ADDRESS_W);
	twi_write(address);
	twi_write(data);
	twi_stop();
	return 0;
}

uint8_t Read_MPU9250(uint8_t address, uint8_t *data) {
	// transmit address
	twi_start();
	twi_write((MPU9250_ADDRESS<<1)|ADDRESS_W);
	twi_write(address);
	twi_stop();

	// read back from device
```

```
	twi_start();
	twi_write((MPU9250_ADDRESS<<1)|ADDRESS_R);
	*data = twi_read_nack();
	//*data = 0xA5;
	twi_stop();

	return 0;
}

uint8_t Read_AK8963(uint8_t address, uint8_t *data) {
	// transmit address
	twi_start();
	twi_write((AK8963_ADDRESS<<1)|ADDRESS_W);
	twi_write(address);
	twi_stop();

	// read back from device
	twi_start();
	twi_write((AK8963_ADDRESS<<1)|ADDRESS_R);
	*data = twi_read_nack();
	twi_stop();
	return 0;
}

uint8_t Read_AK8963_Array(uint8_t address, uint8_t *data[], uint8_t length) {
	// transmit address
	twi_start();
	twi_write((AK8963_ADDRESS<<1)|ADDRESS_W);
	twi_write(address);
	twi_stop();

	// read back from device
	uint8_t i;
	for (i=0;i<(length-1);i++) {
	twi_start();
	twi_write((AK8963_ADDRESS<<1)|ADDRESS_R);
	*data[i] = twi_read_ack();
	}
	twi_start();
	twi_write((AK8963_ADDRESS<<1)|ADDRESS_R);
	*data[i] = twi_read_nack();
	twi_stop();
	return 0;


}
```

A4.3 Servo Control

```
/*
 * Motor_PWM.c
 *
 * Created: 11/17/2016
 *  Author: Daniel
 *
 *          IEEE Robotic Competition Code
 *
 */

#include "Servo_PWM.h"

 void Initialize_Servo_PWM() {

            // Setup Hardware
            SERVO_DDR = SERVO_DDR|(SERVO_BITS); // Output
            SERVO_PORT = SERVO_PORT&(~SERVO_BITS); // Set to LOW

            OCR2A = 0; //  PWM duty cycle
            OCR2B = 0; // PWM duty cycle

            //
            //TCCR0A = (1<<WGM01)|(1<<WGM00);
            TCCR2A = (0<<WGM21)|(1<<WGM20);
            // OC0A is cleared on compare match when counting up, set on compare match when counting down

            //// The output compare pins are controlled via header file definitions for easy configuration changes
            // Inverting mode is used so that we can have an absolute zero to disable the servo
            if ((OCR2_PIN&OCR2A_PIN)==OCR2A_PIN) {
            TCCR2A |= (1<<COM2A1)|(0<<COM2A0);
            }
            if ((OCR2_PIN&OCR2B_PIN)==OCR2B_PIN) {
            TCCR2A |= (1<<COM2B1)|(0<<COM2B0);
            }
            // Phase Correct PWM Mode
            // Clock Divider of 256 - Assuming 8.0 MHz clock on Fast PWM, the PWM signal will have a frequency
of 8MHz/256/512 = 61 Hz.
            // This means that the resolution of the PWM is (1/61Hz/256) = 64 microSeconds./
            TCCR2B = (0<<FOC2A)|(0<<FOC2B)|(0<<WGM22)|(1<<CS22)|(1<<CS21)|(0<<CS20);


            //
            //TIMSK0 = (0<<TOIE0)|(1<<OCIE0A);
            //
            //sei();
 }
```

```
void Set_Duty_Cycle(uint8_t duty_cycleA, uint8_t duty_cycleB) {
        if ((SERVO_MIN_DUTY<=duty_cycleA)&(duty_cycleA<=SERVO_MAX_DUTY)) {
        OCR2A = duty_cycleA;
        } else if (SERVO_MIN_DUTY>duty_cycleA){
        OCR2A = SERVO_MIN_DUTY;
        } else if (SERVO_MAX_DUTY<duty_cycleA){
        OCR2A = SERVO_MAX_DUTY;
        }

        if ((SERVO_MIN_DUTY<=duty_cycleB)&(duty_cycleB<=SERVO_MAX_DUTY)) {
        OCR2B = duty_cycleB;
        } else if (SERVO_MIN_DUTY>duty_cycleB){
        OCR2B = SERVO_MIN_DUTY;
        } else if (SERVO_MAX_DUTY<duty_cycleB){
        OCR2B = SERVO_MAX_DUTY;
        }
}

void ServoA_Full_Sweep() {
        uint8_t sweep_duty = SERVO_MIN_DUTY+1;
        // Sweep up
        while (sweep_duty<SERVO_MAX_DUTY) {
        Set_Duty_Cycle(sweep_duty, OCR0B);
        _delay_ms(SERVO_SWEEP_TIME_STEP);
        sweep_duty = sweep_duty+SERVO_SWEEP_INCREMENT;
        }
        // Sweep down
        while (sweep_duty>SERVO_MIN_DUTY) {
        Set_Duty_Cycle(sweep_duty, OCR0B);
        _delay_ms(SERVO_SWEEP_TIME_STEP);
        sweep_duty = sweep_duty-SERVO_SWEEP_INCREMENT;
        }
        Servo_Off();
}

 void Servo_Off() {
          OCR0A = 0;
          OCR0B = 0;
 }


 //ISR(TIMER0_OVF_vect) {
        //PORTD = (PORTD^0x10);
 //}
//
 //ISR(TIMER0_COMPA_vect) {
        //PORTD = (PORTD^0x20);
 //}
```

A5. Appendix E - Task 3 Code

This code was implemented on an ATMega328P microcontroller.

A5.1 Task 3 Control

```
enable_task1();
motor_in_cw(MOTOR_STEPS);
disable_task1();

PORTB |= 1<<7;
_delay_ms(1);                          //send TASK_COMPLETE signal (triggers interrupt once)
PORTB &= ~(1<<7);

Initialize_Servo_PWM();
_delay_ms(200);
OCR0B = 18;      //Open gripper
enable_task3();
PORTC |= 0x28;
PORTC &= ~(0x14);

_delay_ms(10000);

task_3_process();

while (1)
{
}

void enable_task3(void){
        MOTOR_EN_REG |= 1<<TASK3_EN;
}

void disable_task3(void){
        MOTOR_EN_REG &= ~(1<<TASK3_EN);
}

void task_3_process(void)
{
        OCR0B = 30;                //close gripper
        _delay_ms(2000);

        //Convert 16 bit input (component_code) to rotation numbers
        //t1=number of rotations in the first turn
        //t2=number of rotations in the second turn...
        uint16_t t5, t4, t3, t2, t1;
        t5 = 0x7 & component_code;
        component_code = component_code >>3;
```

```
		t4 = 0x7 & component_code;
		component_code = component_code >>3;
		t3 = 0x7 & component_code;
		component_code = component_code >>3;
		t2 = 0x7 & component_code;
		component_code = component_code >>3;
		t1 = 0x7 & component_code;

		//Turning Routine 20 seconds
		enable_task3();
		for (uint8_t i=0;i<t1;i++){motor_in_cw(TASK_3_ROT_STEPS);}  _delay_ms(200);
		for (uint8_t i=0;i<t2;i++){motor_out_ccw(TASK_3_ROT_STEPS);} _delay_ms(200);
		for (uint8_t i=0;i<t3;i++){motor_in_cw(TASK_3_ROT_STEPS);}  _delay_ms(200);
		for (uint8_t i=0;i<t4;i++){motor_out_ccw(TASK_3_ROT_STEPS);} _delay_ms(200);
		for (uint8_t i=0;i<t5;i++){motor_in_cw(TASK_3_ROT_STEPS);}  _delay_ms(200);
		disable_task3();

		_delay_ms(200);
		OCR0B = 18;
		_delay_ms(1000);
		Servo_Off();

}
```

A5.2 Servo Control

```
/*
 * Motor_PWM.c
 *
 * Created: 11/17/2016
 *  Author: Daniel
  *
  *		IEEE Robotic Competition Code
  *
 */


#include "Servo_PWM.h"

void Initialize_Servo_PWM() {
		OCR0B = 0; //  PWM duty cycle
		//// The output compare pins are controlled via header file definitions for easy configuration changes
		// Inverting mode is used so that we can have an absolute zero to disable the servo
		TCCR0A = (1<<COM0B1) | (1<<WGM01) | (1<<WGM00);
		TCCR0B = (1<<CS01) | (1<<CS00);
		// Clock Divider of 256 - Assuming 8.0 MHz clock on Fast PWM, the PWM signal will have a frequency
of 8MHz/256/512 = 61 Hz.
		// This means that the resolution of the PWM is (1/61Hz/256) = 64 microSeconds./
//		TCCR0B = (0<<FOC0A)|(0<<FOC0B)|(0<<WGM02)|(1<<CS02)|(0<<CS01)|(0<<CS00);
```

```
}

 void Set_Duty_Cycle(uint8_t duty_cycle) {

          if ((SERVO_MIN_DUTY<=duty_cycle)&(duty_cycle<=SERVO_MAX_DUTY)) {
          OCR0B = duty_cycle;
          } else if (SERVO_MIN_DUTY>duty_cycle){
          OCR0B = SERVO_MIN_DUTY;
          } else if (SERVO_MAX_DUTY<duty_cycle){
          OCR0B = SERVO_MAX_DUTY;
          }

 }

 void Open_Gripper() {
          Set_Duty_Cycle(SERVO_MID_DUTY-20); //smaller number means more open
 }

 void Close_Gripper() {
          Set_Duty_Cycle(SERVO_MID_DUTY-5); //Larger number means more closed
 }

 void Servo_Center() {
          Set_Duty_Cycle(SERVO_MID_DUTY);
 }

 void Servo_Off() {
          TCCR0B &= ~((1<<CS02) | (1<<CS01) | (1<<CS00));
 }
```

A6. Appendix F - Competition Rules and Specifications

Below is the text from the competition rules and specifications.

Episode MMXVII: The Engineering Force Awakens

A long time ago, in a galaxy far, far away a lot of stuff happened that pitted good against evil in some strange universe. Unfortunately that has nothing to do with this challenge.

Instead, in this robot competition you will build a robot that must discover the unknown, use the Force in a lightsaber duel, bring down the energy shield protecting the enemy base, and then launch a proton torpedo to defeat the enemy.

And your robot must complete these tasks in under four minutes!

The robot

The robot must be completely autonomous and must fit entirely within a 12”x12”x12” cube at the start of each match. The robot must at all times be wholly contained within the playing surface and cannot reach more than 3” beyond the edge of any the arena walls. There is no weight restriction on the robot. Aerial or flying and/or launched robots are not allowed.

The robot may expand beyond the initial size to any size, and may even split into multiple independent robots during the competition. If multiple independent robots are used, they must all start within the same 12”x12”x12” space and split after the competition begins.

In addition, the independent robots must communicate over a wired link – wireless communication between the various robots is not allowed - this includes radio, light, sound – any non-wired communication channel. This is a safety issue to avoid interference with, and to, other robots that are running at the same time.

You may not tether or externally control the robot in any way including wired or wireless tethers, two way data transmission, or one way data transmission to or from the robot.

The robot cannot contain any explosives or flammable liquids or gases. Compressed gas is allowed on the robot as long as the pressure is limited to no more than 30 pounds per square inch at any time. Gasses other than air are permitted as long as they do not pose a safety threat if accidentally released.

2017 Southeastcon Hardware Challenge Rules (October 9, 2017) Page 1

While multiple switches can exist on the robot for powering up and controlling the various subsystems on the robot, there must be a single clearly visible and labeled start switch. This switch can be either a pushbutton or a toggle switch and will be actuated by the robot team once the judge indicates the start of the match. It is recommended that the robot have an easily reachable emergency cut off switch to allow the robot team to disable the robot if necessary (this is to avoid damage to both the robot and the arena in case a sudden stop is required). If the robot splits into multiple independent robots, the single emergency cut off switch should stop motion of all the robots.

The robot may not present any danger to the judges, spectators, the playing arena or neighboring area around the arena. If at any time the judges deem the robot is causing, or is likely to cause harm, the judge may terminate the match immediately and will have the discretion whether any points are awarded for the match, and if the robot is allowed to complete any remaining matches.

Arena layout

The arena is a single 4' x 8' sheet of smooth sanded ½" (nominal thickness is approximately 15/32") BC grade plywood (B side up) that is surrounded by standard 2x4 'stud' lumber (nominal size is approximately 1.5" x 3.5") walls that form a frame on top of the plywood sheet (the arena inner dimensions will thus be approximately 93" x 45").

(Note that all ½" plywood referenced in the construction of the arena and stages will have a nominal thickness of approximately 15/32" and all 2x4 'stud' lumber will have a nominal size of approximately 1.5" x 3.5", and all 1x2 lumber will have a nominal size of approximately 0.75" x 1.5". Standard building material tolerances are expected).

The arena is divided into five levels, each at a different height. The robot begins in a starting square in the middle of the lowest level. This layer is the largest area, and comprises over half of the playing surface (45" wide by 57" long), and contains the robot starting square and the locations of stages 1, 2 and 3. The starting square is 15"x15" and is located in the center of this space (15" from the inside of the long walls of the arena, and 21" from the inside of the short wall). A 1" wide and 8" long white navigation stripe leads from the middle of the starting area square toward stage 1.

The rest of the arena contains a four tier stepped area consisting of three full-width steps and one smaller step. Each of the full-width steps are 12" deep and 45" wide. The first step is approximately 0.5" high (one thickness of ½" plywood) above the ground/starting floor, followed by a 1" step (two thicknesses of ½" plywood) to the second step and a 1" step (two thicknesses of ½" plywood) to the third step. The fourth step is 1" high (two thicknesses of ½" plywood), or approximately 3.3" above the ground floor) and is only 12" wide and 12" deep and is centered over the third step. Each of the steps will have a 1" gloss

white ‘warning’ stripe painted on the top edge of the step at the drop-off to the next lowest step.

Figure 1 - Arena Diagram

The three staging areas for the competition are located on the three walls surrounding the starting area. They are fixed in position and the center of each staging area aligns with the center of the starting square (the center of stage 1 and stage 3 is 28.5” from the inside wall containing stage 2, and the center of stage 2 is located 22.5” from each of the inside walls containing stages 1 and 3).

Each stage represents a scene from a Star Wars movie.

• Stage 1 – Uncovering the Unknown This stage represents the scene where R2D2 plugs into the data port of the Death Star to shut down the trash compactor to save Princess Leia. For this contest, your robot must decode the secrets required to shut down the shield that is protecting the enemy base.

• Stage 2 – Lightsaber Duel This stage represents one of many scenes where good must battle against evil in a lightsaber duel. Your robot will be required to “use the Force” by sensing an electromagnetic force field to know when to strike the arena’s lightsaber with your own lightsaber.

• Stage 3 – Bring Down the Shields Using the secret information you learned in stage 1, your robot must now enter the correct combination code to bring down the energy shield that is protecting the enemy base.

• Final stage – Launch a Proton Torpedo Now that the shields are down, you must launch three proton torpedoes through a hole at far end of the arena.

The stages may be completed in any order, with the following exceptions – the information from stage 1 is needed for stage 3, and the firing of the last torpedo toward stage 4 ends the match (you are able to fire the first two at any time during the match).

All the plywood surfaces, including each of the layers of the arena and stages 1-4, are painted flat black. All the non-plywood surfaces, including the 2x4 stud wall frames, and the 1x2 frames of each stage are painted gloss white. In addition, gloss white is used in the 15”x15” starting square, the 1” line leading from the starting square toward stage 1, the 1x2 thick frame around the portal hole for the final stage, and the 1” warning line for each step. Note that we will not be providing exact paint vendor names and manufacturing codes for these paints, so your robot should be able to discern between flat black and gloss white without looking for an exact match.

(Note: As past competitors have discovered, colors never look the same under different lighting conditions, so it is never advisable to look for exact colors in a competition. Thus we chose this high contrast paint option as a benefit to you, not as an additional challenge).

Stage 1 Discovering the Unknown

In stage 1, the robot must decode the hidden code required to bring down the shields in stage 3.

This stage consists of six flat conductive copper pads arranged as shown in the diagram below. Each pad is attached to the back of the ½” plywood and is accessible via a 0.5” diameter hole. The pads on the perimeter are arranged in a circle with a 1.5” radius around the center pad (the 1.5” radius circle passes through the center of each of the 0.5” diameter holes). The pads are only numbered in the following description; no number actually exists on the pad or on the stage). Pads begin with 1 at the top 0° position, with the next 4 counting off sequentially in a clockwise manner at 72°, 144°, 216°, and 288°.

Figure 2: Stage 1 (Discover the Unknown)

Between the common (center) pad and each of the surrounding pads are one of five components arranged in an order that is unknown to the robot when the match begins. Each component will exist once, and only once, but their order will be chosen randomly at the start of each match. The five components are a wire, resistor, capacitor, inductor, and a diode. The message is encoded with each component representing a unique digit code value as shown in the table below.

Code

Component type

Suggested vendor / part number 1 Wire N/A N/A 2 Resistor 10KΩ, 1/2W, 10% tolerance Any 3 Capacitor 0.1uF, non polarized, ceramic Any 4 Inductor 500mH, 30mA, 736Ω Digikey, M10176-ND

5 Diode

Component value

IN4001–cathode/anode can be oriented in either direction

Any

Note: This inductor is now listed as obsolete and no longer available from Digikey. However, it is possible to obtain it from other vendors, or via Amazon. Also check on the Facebook page for the competition for an offer to obtain a free copy of this inductor (while supplies last).

The robot can access the pads in any order, and with any configuration and/or number of simultaneous connections. For example, the robot can measure only pad 1 to common, and then pad 2 to common, etc, or it can simultaneously connect to all six pads, or any subset number of pads.

The robot should not scratch, dent or modify the copper pads in any way, or leave any visible signs of damage to the pads or the plywood frame in front of the pads. If any damages are visible by the judges, the robot will be disqualified and given 0 points for the match.

The measurement of the component values should not exceed a safety voltage limit of 12v, or a current of 25mA. A single 100mA slow blow fuse will exist between the center common pad and all the components. This fuse will be automatically checked at the beginning and end of each match and if a robot blows a fuse, it will be disqualified and will result in a match score of zero points.

Stage 1 rises above the wall and consists of a 6” tall by 6” wide 1⁄2” plywood backboard inside a frame of 1x2 lumber, creating an approximately 1” deep shadow box. The center (common) pad will be located in the middle of the 6”x6” plywood backboard. The stage will be attached by a single bolt mid height on the wall so that it can be taken down for easier storage and transportation, or replacement during the competition if a problem is discovered. The bolt head will be countersunk so that it does not protrude beyond the wall edge.

Stage 2 Lightsaber Duel

In stage 2, the robot must use the (electromagnetic) force to sense when a lightsaber is activated. The robot must battle using its own lightsaber to hit the stage 2 lightsaber each time the electromagnetic force field is activated.

The electromagnetic force will be generated by an electromagnet located in the middle of the wall and is located directly behind the stage 2 wall mounting bolt. An immobile lightsaber rises approximately 7” above the stage 2 frame. The lightsaber will consist of a hilt, approximately 3” tall attached directly to the wall and a 4” lightsaber blade. The lightsaber will be lit by an eight element RGB neopixel (LED) array and will also contain a vibration sensor. The robot should only strike the lightsaber hilt, and NOT the blade directly, to avoid damaging the lightsaber blade and RGB light array.

Each time the magnetic field is energized, the robot should strike the vertical (non-moving) lightsaber. Each time the lightsaber is hit, it will briefly flash - either blue to indicate it was struck while the magnetic force field was activated and points are awarded, or red to indicate the lightsaber was not activated and a penalty will be deducted. Striking the lightsaber will be detected by the vibration sensor and automatically counted via the arena controller.

Figure 3: Stage 2 (Lightsaber Duel)

The LED array will be visible along the length of the lightsaber and the vibration sensor will be attached to the lightsaber. The electromagnet will consist of 40 turns of #20 stranded, insulated, copper wire wound around a 0.5” diameter bobbin and will be energized by one amp. The wire should be wrapped clockwise around the bobbin, as viewed from the robot inside the arena.

Both the lightsaber (mounting bracket and hilt, inner RGB array support and saber blade) and the bobbin form will be a 3D printed PLA part, with the design available on the competition rule website.

2017 Southeastcon Hardware Challenge Rules (October 9, 2017) Page 7

The vibration sensor will ignore any vibrations/motion in the first five seconds of the match to avoid any false triggers while the team is starting their robot and stepping away from the arena. Once the five second timeout period is over, the magnetic force field will be energized.

Part description Suggested vendor / part number

Lightsaber LED array

Adafruit neopixel stick (https://www.adafruit.com/products/1426)

Vibration sensor

Adafruit medium vibration sensor switch (https://www.adafruit.com/products/2384)

Once the robot strikes the lightsaber the first time, the force field is disabled and a 30- second battle timer begins that indicates the total length of time for the remainder of the lightsaber duel. During the next 30 seconds, the force field will be activated and deactivated a total of four more times.

The starting time for the four additional times will vary from match to match, with the following rules:

• The lightsaber is activated only when the electromagnetic force is on.

• The force field is activated each time for two seconds, or until the lightsaber is hit, whichever occurs first.

• Each deactivation will vary in time, with a minimum set time of 1 second.

• The final (fourth) activation will occur at exactly 28 seconds into the duel. This is to guarantee that each duel can be completed in 30 seconds to ensure fairness from robot to robot so that the final activation does not occur earlier in some matches.

• At most only one hit is counted per active state — the rest will be ignored until the next activation.

• At most only one negative hit penalty is deducted per inactive state — the rest will be ignored until the next deactivation.

• Hits that occur during the first 0.5 seconds of the deactivation do not count for or against the robot – they are neutral points – to avoid assigning a penalty to a hit that arrived just after the field was deactivated.

The lightsaber LEDs will also be used as the countdown timer at the start of the match. The initial state of the lightsaber will be all red before the contest begins. Once the team is ready, the judge will give the team a 3 second warning, and the lightsaber will countdown to indicate each second. Once the 3 second warning is over, the lightsaber will light green

indicating the match has started. Except for the 30 second duel with the lightsaber, it will remain green during the match, and will turn red once the match is over.

Once the first vibration is sensed, the lightsaber will flash red and then prepare for ‘battle mode’ by turning all the LEDs off. Then, on each hit, the lightsaber will either flash blue (for a hit during the time the magnet field is energized) or red (for a hit when the magnet field is not energized). Each hit will result in the blue or red flash, but only the first hit of each on or off period will result in points. Once the 30 second duel is over, the lightsaber will again be lit green. If the four minute match ends during the duel, the duel immediately ends and the lightsaber is lit in red.

Stage 2 rises above the wall and consists of a 3” tall x 6” wide plywood backboard inside a frame of 1x2 lumber, creating a 1” deep shadow box. The lightsaber rises about 7” above the top of the frame. The stage will be attached by a single bolt mid height on the wall so that it can be taken down for easier storage and transportation, or replacement during the competition if a problem is discovered. The bolt head will be countersunk so that it does not protrude beyond the wall edge, but the bolt head will be within 1⁄8” of the inside wall surface. In addition, this bolt will pass through the magnetic coil bobbin that is used to generate the force field.

Stage 3 Bring Down the Shields

In stage 3, the robot uses the codes from stage 1 to unlock and bring down the force fields by dialing in each digit on a combination lock. The stage 3 combination lock is a quadrature encoder with a built-in RGB LED.

The sequence begins by the robot first turning the combination lock knob a full circle clockwise 'N' number of times for the code value derived from pad 1 (top) of stage 1. The robot then turns the knob counterclockwise 'N' number of complete turns for the code derived from pad 2, and then back clockwise ‘N’ complete turns for the code value derived from pad 3, counter-clockwise ‘N’ turns for the code value derived from pad 4, and finally completing with ‘N’ clockwise turns for the code value derived from pad 5.

A turn is considered a complete turn if it is +/- 15° from a complete 360° turn (ie: 345° to 375° is considered one turn, 705° to 735° is considered two turns, etc). If the robot stops and reverses direction when not within this range, it is not considered a valid digit entry.

If more than five digits are entered (via the alternation of clockwise and counterclockwise turns), only the last five will be counted. This allows a robot to “abort” a sequence by simply starting over at the beginning again. The robot can use this feature as a way to account for any accidental rotation of the knob during initial robot alignment.

Part description Suggested vendor / part number

Quadrature encoder Sparkfun rotary encoder – illuminated (RGB)

(https://www.sparkfun.com/products/10982) Quadrature encoder knob Sparkfun clear plastic knob

(https://www.sparkfun.com/products/10597)

The initial state of the stage 3 RGB LED is a 5Hz white flashing beacon. Once the robot engages with the encoder and rotates it more than ±15°, the RGB LED will stop flashing. The RGB LED will then be lit red while the shaft is rotated clockwise, blue while rotated counterclockwise and white when within the ±15° of a complete turn (when moving either direction).

(Note: The control knob LED colors are more for visual verification by the judge that the knob is moving and is not recommended as a signal to the robot, but this is not disallowed and is up to the robot team designer)

Figure 4: Stage 3 (Bring Down the Shields)

Stage 3 rises above the wall and consists of a 6” tall by 6” high 1⁄2” plywood backboard inside a frame of 1x2 lumber, creating a 1” deep shadow box. The quadrature encoder will be located in the middle of the 6”x6” plywood backboard. The stage will be attached by a single bolt mid height on the wall so that it can be taken down for easier storage and transportation, or replacement during the competition if a problem is discovered. The bolt head will be countersunk so that it does not protrude beyond the wall edge.

Final stage (Fire the Proton Torpedo)

In the final stage, the robot must launch three missiles (a Nerf N-Strike Elite dart – these are the regular tips, not suction tips) into the portal at the far end of the arena (farthest from stage 2). The portal is a 6”x6” square opening that is located in the backboard. The backboard is constructed of 1⁄2” plywood that extends a minimum of 12” above the wall and is 48” wide, extending the full width of the arena. The hole is framed with the same 1x2 gloss white frame as used in stages 1-3. The bottom of the frame is 3.5” from the top step, and it is located midway between the long walls on which stages 1 and 3 are located. The backboard is 48” wide, extending the full width of the arena, and is bolted to the back of the 2x4” wall.

The robot can launch the Nerf missiles from any location within the arena. This means the robot can try for the longer shot from the lower ground stage area, or it can run up directly to the portal and ‘dunk’ (drop) it in the hole if it chooses to navigate the ever increasing higher platforms leading up the portal.

The robot gets points if at least one Nerf missile is fired (whether it makes its target or not), and points for each Nerf missile that goes through the portal. The match is over once the last Nerf missile is fired and either clears the portal (or is declared a miss). A simple net capture system will exist behind the hole to capture the Nerf missiles for counting.

While an occasional errant missile fire is expected, if the judges suspect that a missile is intentionally being fired in a direction other than the portal, the robot will be immediately disqualified and removed from the competition.

Figure 5: Stage 4 Target Area (and the steps leading to it)

Running the Competition

There is no limit on team size for the participating team, but each team member should be in the same local student chapter, and they must all be IEEE Region 3 student members. Teams that do not fit this qualification (either other regions, robots not associated with the local student chapter, hobby groups, non-students) will compete in the Open category, but all team members should be IEEE members.

The competition will consist of two preliminary runs, with an optional third preliminary run if time permits. The total points for each team will be the sum of their match scores for all preliminary runs.

The top four highest scoring teams in the preliminary runs will compete in the final run to be held during the awards banquet. The final competition placement of the four teams will be determined solely on the points in the final run and not dependent on any points earned in the preliminary runs.

Each match will last for at most four minutes and the end of the time will be indicated by all 8 LEDs on the arena lightsaber turning on red. The match for a single robot is over when either the four-minute time limit expires, or the last missile is fired. In addition, the robot team can signal to the judge at any time that they are finished with their match, or the judge can stop the match if the robot is acting in a manner that can cause injury to anyone nearby or damage to the arena or itself.

At the start of each run, the judges will require that all robots be sequestered in a special staging area. Once in the staging area, the robots must remain off and cannot be touched by

any students until they are called for their match to begin. The robots also cannot be charging during this time.

For each match, the judges will call the names of the teams to run in that match. Once called, teams will have at most two minutes to retrieve their robot from the sequestration area, place it into the arena, and be ready to start at the judge's command.

When a match is ready to begin, the arena judge will start the time on their arena and the stage 2 LED bar will count down to zero (over a three second period) and will turn green to indicate the match has started. Each team will be responsible for starting their own robot (using the clearly provided and labeled start button), then must back out of the way of the arena. Once the match is over, the robot will be returned to the sequestration area until all the robots in the match have completed. Once that is over, the teams will be instructed to retrieve their robot. Teams will be guaranteed at least 30 minutes between the end of one match and the start of the next match for any changes to the robot necessary for the next match.

While there will be multiple arenas running at the same time, the relative times for the robots within a single run will not matter as the robot times for all of their matches will be tallied and sorted against all other robots.

Constructing the arena

The arena was designed so that it can be constructed from two sheets of 4'x8' 1⁄2" BC grade plywood. The actual thickness of the plywood will vary and a more accurate description of the thickness is approximately 15/32", but robots should be built to allow slight variations without affecting their performance.

In addition to the two sheets of plywood, the arena will require three 8' long 2x4 pine studs, and one 8' long 1x2 standard pine lumber.

As stated earlier, the paint colors are flat black and gloss white. Exact vendor and manufacturing numbers for the paint will not be provided.

A separate build document is being prepared that shows photos and build hints from building the initial arena prototype.

Scoring

Points are awarded as follows:

• Starting

o 10 points are awarded if the robot shows any sign of motion.

o 30 points are awarded if any part of the robot crosses over the edge of the

starting square. o The maximum points possible for starting are 40.

• Stage 1: Decoding the Unknown

o 10 points will be awarded for the robot touching any part of stage 1. o 15 points will be awarded for correctly decoding 1 pad, 35 for decoding 2 pads, 60 for decoding 3 pads, 90 for decoding 4 pads, and 125 for decoding all five pads. o Points for this stage will be awarded by properly decoding stage 3 or by presenting to the judges a display that shows that the stage 1 results were properly decoded. The display must be an electronic display (LED or LCD) on the robot that clearly is visible by the judges at the end of the match that indicates the code values determined for each of the five pads on stage 1. The displayed code should be a five digit number, read left to right, to coincide with stage 1 pad numbers 1..5. Note that the display must be visible on the robot by the judges, and ssh'ing to the robot, or providing the judges a piece of paper with the numbers on it will not count as a valid display. Additionally, the display must be read by the judges before the robot enters the sequestration area, or the points will not qualify. o The maximum point total possible for stage 1 is 135.

• Stage 2: Lightsaber Duel

o 10 points will be awarded for the robot touching any part of stage 2. o 30 points will be awarded for the first hit (to start the duel, as indicated by a

the lightsaber flashing blue). o 65 points will be awarded if only one additional hit is registered during the magnetic field activation, 110 if two are registered, 170 if three are registered, and 250 if all four are registered. Note that only one hit per magnetic field activation will be counted, so to get more than 2 hits, each must occur during a different magnetic field activation. o 50 points will be deducted for a hit that occurs when the force field is not activated (not counting the 0.5 second grace period after the field is deactivated). Note that only one penalty per magnetic field deactivation will be counted (maximum loss is 200). o The maximum point total possible for stage 2 is 290 and cannot be negative.

• Stage 3: Bring Down the Shields

o 10 points will be awarded for the robot touching any part of stage 3.

o 45 points will be awarded if one digit is dialed in at the correct location in the sequence, 95 if two are dialed in correctly, 155 if three are dialed in correctly, 230 if four are dialed in correctly and 325 if all five are dialed in correctly (for example, if the code is 12345, but 12435 is entered then the 1, 2 and 5 are correct so 155 points are awarded). o The maximum point total possible for stage 3 is 325. o Since failure at this stage also means no points for stage 1, the team should provide a very clear display on the robot that will show the decoded digits from stage 1. If all the digits are entered correctly at stage 3, it is assumed all digits were decoded correctly from stage 1 correctly. If one or more digits were incorrectly entered at stage 3, and the team can show that the values were decoded successfully at stage 1, they get the full points for the correct digits at stage 1, plus any points for correctly entered digits at stage 3.

• Final stage: Fire the Proton Torpedoes

o 10 points will be awarded if at least one Nerf missile is fired (regardless of

whether it goes through the portal). o 50 points bonus will be awarded if only one missile passes through the portal. 120 if two pass through the portal, and 200 if all three pass through the portal. o It is not necessary to successfully complete either stages 1, 2 or 3 to be awarded points for the final stage, but once the last Nerf proton torpedo is fired (and either clears the portal or is considered a miss), the match is over and the robot is not allowed to engage with any of the other stages. o The maximum point total possible for the final stage is 210.

• Maximum points for a competition:

o Starting: 40 maximum points o Stage 1: 135 maximum points o Stage 2: 290 maximum points o Stage 3: 325 maximum points o Final stage: 210 maximum points o Total maximum points: 1000

FAQ

• Where do I find out more about Southeastcon 2017? http://sites.ieee.org/southeastcon2017/

• Where is the 2017 Southeastcon Facebook page?
https://www.facebook.com/groups/SouthEastCon2017HardwareCompetition/

• Where do I find the latest official copy of the rules? The latest (and only official copy) will always be posted to the following location:
http://sites.ieee.org/southeastcon2017/student-program/

• I see the terms match, run and competition used – what does each mean?

A match refers to a single robot running a single time, with the result of a single score. There is one match per run per robot. A run is the collection of matches for all teams to run once. The first two runs (or optionally three, if time allows a third run before the banquet) are called preliminary runs. The purpose of the preliminary runs is to find the top four teams to compete in the final run. The final run will occur during the awards banquet on Saturday night. During the final run, the only points used to define the final competition placement of the top four teams are the scores from the match in the single final run – no points from the preliminary runs are carried forward.

• If I have a question, can I post it to the 2017 Southeastcon Hardware Competition page?

Yes, you can post the question there and perhaps someone will help you with your answer. However, all official answers must be answered by the 2017 Southeastcon Hardware Rules document and not by a comment on the Facebook page. The page is designed mainly for students to exchange ideas and information, and for quick notices about changes in the official documentation. If a discrepancy exists between what is stated on either of the 2017 Southeastcon Facebook pages and the official website and official rules on the website, the official page and rules will be deemed more correct.

Please read and follow the Facebook etiquette rules in the Facebook group.

• Where do I send my questions/comments about the 2017 hardware challenge? Please contact Rodney Radford, the 2017 Southeastcon hardware rules chair at rodney.radford.us@ieee.org.

• I really liked that opening Star Wars scroll – where can I see it again? Click “Begin” at the link below to see the current scroll, or to edit/create your own scroll (and make sure you have the sound turned on for the full effect!) http://www.starwars.com/games-apps/star-wars-crawl-creator/?cid=5700879ee4b03db91956f207

Additional photographs of the arena

This section contains additional diagrams of the arena. These were all created from the official OpenSCAD code files available in the files section of the Southeastcon 2017 Facebook page.

Figure 6: Arena Area (as seen from above)

Figure 7: Arena as seen from behind stage 1

Figure 8: Lower Level and Stages (as seen from the stage 4 steps)

2017 Southeastcon Hardware Challenge Rules (October 9, 2017) Page 19

Important dates for the 2017 hardware challenge:

Initially an aggressive schedule was proposed for releasing the schematic diagrams and Arduino code for the arena electronics. While this schedule below has relaxed the deadline of the arena electronics a bit, this should not impact the robot design, building, or testing. Each team can build and test their robot by visually inspecting the robot meets the requirements (as they have done in years past), and the release of the arena electronics is mainly to help debug or uncover any issues in the arena electronics.

The build guide will provide hints on how the stages can be tested without building the full arena electronics – both saving time and money to the students.

The new dates are:

• April 3

o Deliver the basic set of rules at the Sunday Southeastcon 2016 meeting. o 2017 Southeastcon hardware rules Facebook page goes live.

• May 15

o Release of the OpenSCAD source files used to generate the 3D diagrams.

• July 4

o Next major release of the rules. The July 4 release included the following

changes and updates:

▪ Wording clarifications based on questions received before July 4.

▪ Correction of some of the arena sizes (to match the OpenSCAD file).

▪ Release of the OpenSCAD files for the stage 2 lightsaber and coil spool.

▪ Component vendors and part numbers for stage 1 parts.

▪ Changed the number of turns (to 40) and increase of current (to 1A) in the coil for stage 2 (this was based on results from the initial prototype build).

▪ Changed the colors of the stage 3 RGB quadrature to red/blue to match the theme in the lightsaber.

▪ Added the requirement that all team members must be IEEE student members and in the same student chapter. This requirement has always existed for Southeastcon, but listed it specifically in the rules to avoid any confusion.

▪ Increased the time between matches from 1 minute to 2 minutes.

▪ Updated the dates/schedule for next updates.

• July 11

o Minor update of the rules that included the following changes:

▪ Updated vendor information for unknown parts in stage 1 – only the inductor is considered unusual enough that it requires a specific vendor and part number.

▪ Changed the max voltage, current and fuse for stage 1 based on the inductor selected for stage 1.

▪ Clarified the maximum distance from the bolt surface to the inside of the wall for stage 2.

• October 9

o Final minor update to the rules that included the following changes:

▪ Added information about the Open category for teams that are not associated with a Region 3 student group.

▪ Clarified that no aerial vehicles are allowed in the competition.

▪ Updated information on the inductor availability.

▪ Major changes to the lightsaber LED lighting patterns and sequences. This was done to make the interaction more exciting and the changes were based on observations with interacting with the prototype lightsaber. While the lighting changes were extensive, this should not impact any team.

▪ Updated stage 2 drawing to include the light saber.

▪ Clarification of the types of Nerf darts (non-suction tips)

▪ Minor wording changes for clarity (no functional changes).

▪ Updated the document footer date (this was not updated on previous two releases so still showed the original April date). o Rules update posted to the IEEE website.

• October 31

o All rules and documents are moved to the IEEE site (and removed from

Facebook). o After this date, only major issues will allow any further update to the rules and they will be in a separate document. This additional document will contain any Errata or FAQ to show any changes or clarifications of this document. o Release the schematic diagrams and first draft of the code for the arena electronics. It is expected that both the schematic and the code will continue to evolve and improve over the next few months based on test results. o First release of the external build document with photographs of the

prototype arena, building and assembly guide for the stages, and build hints.

• November and December

o Updates to the arena control hardware and software.

• January 1

o Expected release PCB designs for the arena electronics.

• March 1

o Release of the final version of the Arduino control code for the arena

electronics.

• March 30 – April 2

o Southeastcon 2017 in Charlotte, NC o Full schedule of each of the competitions, and updates on the times for each competition will be posted in advance on both the 2017 Southeastcon Hardware Facebook page and to the 2017 Southeastcon official page