UNIVERSITY OF TORONTO

INSTITUTE FOR AEROSPACE STUDIES

4925 Dufferin Street, Toronto, Ontario, Canada, M3H 5T6

B.A.S.S.
Breakbeam Assisted Smart Stacker

prepared by

Team 41 – Tuesday

Frederick Boyd (1004021435)
Daniel Ding (1004015040)
Jerry He (1003979180)

prepared for

Prof. M.R. Emami

A technical report submitted for
AER201 – Engineering Design

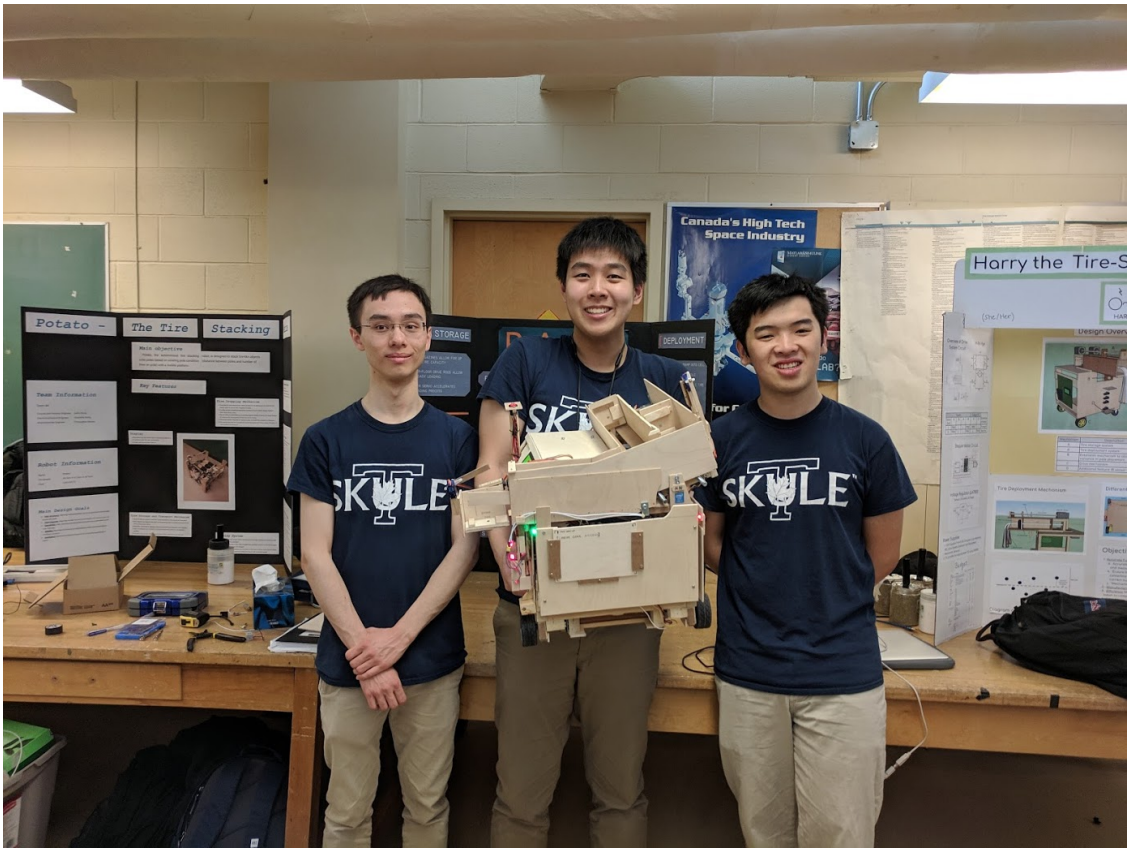
TA: Jieun (Christine) Lee

April 11, 2019

Abstract

The opportunity presented by the Request for Proposal was to design and manufacture a scaled-down, proof-of-concept prototype of a mobile platform that can travel along a row of poles and stack tire-like objects on each pole based on the given instructions. The Breakbeam Assisted Smart Stacker (B.A.S.S.) is a $48\times48\times46\text{cm}^3$, 5.1kg autonomous robot that accomplishes this task through the integration of microcontroller, circuit, and electromechanical subsystems.

Tires are stored in two, long magazines that run the length of the robot. Tires are pushed to the front using a linear actuator consisting of two stepper motors, threaded rods, and push plates. At the front of the magazine, the tires are pushed into a perpendicular ramp, and roll down the ramp when they must be deployed. The ramp empties into an extendable arm with a cell at the end which holds a singular tire. To deploy the tire, the cell clamps the tire, rotates 90 degrees to a horizontal position, and releases the tire onto a pole. This is all mounted on a front-wheel drive mobile platform powered by two DC motors.



Team 41 (from left to right): Frederick Boyd, Daniel Ding, and Jerry He

Contents

Contents **3**

1 Acknowledgements **6**

2 Symbols & Abbreviations **7**

3 Introduction **8**

4 Perspective **9**
4.1 Theory & History 9
4.2 Survey 11

5 Decision Making **11**
5.1 Objectives 11
5.2 Constraints 12
5.3 Acceptance Criteria 12

6 Budget **13**

7 Division of the Problem **14**

8 Circuits Subsystem **17**
8.1 Assessment of the Problem 17
8.2 On-board Power 17
8.2.1 Power Supply Choice 17
8.2.2 Power Supply Changes 19
8.2.3 Brownout Issues 21
8.3 DC Motors 23
8.4 Stepper Motors 25
8.4.1 Unipolar Stepper Motor 25
8.4.2 Bipolar Stepper Motors 25
8.5 Servos 26
8.6 Sensors 27
8.7 Speaker 27
8.8 Suggestions for Improvement of the Subsystem 28

9 Electromechanical Subsystem **31**
9.1 Assessment of the Problem . . . 31
9.2 Solution . . . 33
9.2.1 Structural Frame . . . 33
9.2.2 Drivetrain . . . 35
9.2.3 Tire Storage . . . 36
9.2.4 Extending Arm . . . 39
9.2.5 Sensor Mounts . . . 41
9.2.6 Rotating Cell . . . 42
9.2.7 Miscellaneous Hardware . . . 44
9.3 Supporting Calculations, Computer Programs, Simulation Results . . . 44
9.4 Suggestions for Improvement of the Subsystem . . . 47

10 Microcontroller Subsystem **50**
10.1 Assessment of the Problem . . . 50
10.1.1 Microcontroller selection . . . 50
10.1.2 User Interface . . . 51
10.1.3 Drivetrain . . . 51
10.1.4 Sensor Suite . . . 51
10.1.5 Deployment Mechanism . . . 52
10.1.6 Device Communication . . . 52
10.2 Solution . . . 52
10.3 Main Program Logic . . . 52
10.3.1 Device Communication . . . 56
10.3.2 Pin Assignments . . . 59
10.3.3 User Interface . . . 64
10.3.4 Drivetrain . . . 69
10.3.5 Sensor Suite . . . 72
10.3.6 Deployment Mechanism . . . 77
10.4 Suggestions for Improvement of the Subsystem . . . 83

11 Integration **84**

12 System Improvement Suggestions **87**

13 Initial and Accomplished Schedule **88**

14 Overall Machine Description **94**
14.1 Microcontrollers . . . 96
14.2 Power Supply . . . 96
14.3 Structural Frame . . . 96
14.4 Drivetrain . . . 97
14.5 Tire Storage . . . 98
14.6 Extending Arm . . . 101
14.7 Sensors . . . 102
14.8 Rotating Cell . . . 103

15 Standard Operating Procedure **104**

16 Conclusions **107**

A Full Program Code **109**
A.1 PIC Main Program . . . 109
A.2 Control Uno Main Program . . . 136
A.3 Nano Main Program . . . 141
A.4 Drive Uno Main Program . . . 145
A.5 I2C.h . . . 154
A.6 I2C.c . . . 156
A.7 lcd.h . . . 158
A.8 lcd.c . . . 161
A.9 pitches.h . . . 164

17 References **167**

1 Acknowledgements

We would like to express our gratitude and appreciation to the following people for assisting us throughout the course:

- Professor Emami for his guidance and encouragement all semester long, as well as providing us with the amazing opportunity to experience the challenges of real-world engineering design.
- Christine Lee and all the AER201 TAs for always being willing to help and offer invaluable feedback on our design.
- Colin Harry, the Machine Lab supervisor, for his expertise and assistance throughout the design process. He helped us overcome numerous electromechanical challenges.
- Devansh Ranade, Jerry Bai, and Jason Li for "guiding" our robot to success.
- The Engineering Science 2T1 class, specifically Brian Lam, Jennifer Kong, and Yanisa Khambanonda, for providing moral support, spare parts, and making this experience fun and memorable.
- Dylan Dao, Rikin Gurditta, and Crystal Chen for helping us order parts online using their Amazon Prime accounts.
- Jonathan Spraggett and Joel Kahn for providing us with their plumber's grease at a time of great need.
- Kshitij Shah for assisting with prototyping and testing in the early phases of the project.
- The Myhal Centre Light Fabrication Facility for providing us with the necessary tools and training to help us accomplish our goals.
- Frederick Boyd's roommates for tolerating us through many long and loud nights.
- The city of Shenzhen, for producing high-quality DC motors.
- Jacky 'EternalEnvy' Mao for being the most legendary EngSci.

2 Symbols & Abbreviations

- **B.A.S.S.** Breakbeam Assisted Smart Stacker
- **RFP:** Request for Proposal
- **PWM:** Pulse Width Modulation
- **DC:** Direct Current
- **IR:** Infrared
- **ToF:** Time of Flight
- **GND:** Ground
- **LCD:** Liquid Crystal Display
- **PIC:** Programmable Intelligent Computer (or Peripheral Interface Controller)

3 Introduction

The tire industry, like many other manufacturing fields, has experienced an increased use of technology in recent years. Innovations such as 'smart' sensor integration and advanced tread materials are improving tire reliability and lifespan. Of course, improvements can also be made in the assembly line. Manufacturers are looking to automate more and more aspects of the production process [1]. Tire stacking in warehouse facilities and trucks represents one of the final steps before shipping the product. This has traditionally involved large amounts of manpower and time; workers have turned to hazardous improvised tools such as forklifts and crowbars. By leveraging mechanized and automated solutions, the risk of injuries can be reduced and the efficiency of the operation improved.

This design team has been tasked with implementing a solution to automate tire stacking. A machine is needed to travel along a row of poles, stacking one, two, or no tires depending on specific conditions. The machine must record the distance between adjacent poles, the presence of existing tires, and the total length and operation time of the run, among other factors.

As an answer to the RFP, this report presents a final overview of the B.A.S.S - Breakbeam Assisted Smart Stacker project. The B.A.S.S. is a scaled-down prototype of an autonomous mobile platform that fulfills the requirements outlined above. The B.A.S.S is $48\times48\times46\text{cm}^3$ and weighs 5.1 kg. Dual magazines allow this design to store up to 25 tires. By employing a system of triple breakbeam sensors, along with a stabilized arm, the machine can deploy tires with a high degree of accuracy and reliability. The exterior of the B.A.S.S is composed of sliding panels and many hardware components are easily switchable, allowing for a high degree of modularity.

This report is a culmination of three and a half months of work, consisting of project designing, building, and testing. This document will begin with a high-level description of the challenge, along with a brief analysis of historical projects and existing solutions. Following that, the decision-making section will cover the detailed objectives, constraints, and criteria that will be used to evaluate the robot. The project budget includes a listing of components and their associated prices, along with relevant vendors and information where needed. The division of the problem will be covered, with particular focus on the three main subsystems: circuits, electromechanical and microcontroller. The integration phase of the project will be documented; potential changes and future upgrades will be discussed. A comparison of the planned project timeline and actual accomplishments will follow. The report will conclude with a final machine description and its standard operating procedure.

4 Perspective

Before getting into the details of our solution, it is important to look at existing solutions. There are various industry examples of tire storage and stacking machines on the market. Additionally, there are previous AER201 stacking robots that have achieved successful results.

4.1 Theory & History

Tires are an essential aspect of our modern society. In 1903 P.W. Litchfield of the Goodyear Tire Company patented the first tubeless tire [2]. Over the course of the 20th century, further developments in tire technology have led to the ubiquitous use of rubber tires today.

The tire manufacturing industry is an ever-growing $3 billion dollar juggernaut. It has exhibited continuous growth over the five years leading up to 2018 [3]. This is due to the growth in vehicle sales in North America and increased vehicle travel [3]. With this massive industry and huge demand comes the issue of what to do with the tires once they are worn out. Traditionally car tires have gone straight to landfills once their usable lifespan has passed. However in recent years, there has been an increase in the amount of tires being recycled [4]. Fiber in the tires can be removed and used in cement and utility plants [4]. The rubber in the tire can also be removed and used in a variety of applications such as fuel, playgrounds, parks, and garden borders.

As our society's demand for tires continues to grow, efficient storage of these tires is an important issue that needs to be addressed. Optimizing the storage of tires allows for them to be transported to and from consumers using fewer resources. There are tree main methods of storing tires [5]:

1. Stack tires on their side. See Figure 1 for an example.

2. Stack tires beside each other on their threads. See Figure 2 for an example.

3. Lace tires together using the hollow space inside each tire to store adjacent ones. See Figure 3 for an example.

While the lacing technique is the the most efficient way to store tires, it is relatively complex compared to the other two techniques.

Figure 1: An example of tires stacked on their side [5].

Figure 2: An example of tires stores beside each other on their threads [5].

Figure 3: An example of tires stored using a lacing technique. The hollow space inside each tire is used to store nearby tires. This is the most efficient way to store tires, but is relatively complex to set up [5].

4.2 Survey

5 Decision Making

NOTE: the following subsections are only the ones listed in the textbook - we might want to add/remove/change some to it our needs

5.1 Objectives

The goal of our prototype was to stack an appropriate number of tires on poles lined up along a path, without damaging the poles. Our team's primary goal was to develop a robot that could fulfill the RFP requirements, while being accurate, fast and consistent. The following objectives, first listed in the Design Proposal, were used throughout the development process:

1. **Consistency of tire placement on poles**

   Metric: Percentage of tires launched by the robot that land on the ground after a single operation.

2. **Placing the correct number of tires on the pole.**

   Metric: The number of poles that have the correct number of tires after a single operation, as specified by the RFP.

3. **Completing a single operation as fast as possible.**

   Metric: The time taken in seconds to complete a single operation.

4. **Maximizing the tire capacity of the robot.**

   Metric: The maximum number of tires that the robot can hold and launch without needing to reload.

5.2 Constraints

- The entire prototype shall completely fit within a $50\times50\times50\text{cm}^3$ envelope at all operation times [6].
- The weight of the machine without the tires shall not exceed 8 kg [6].
- The machine must use its own on-board power supply during the operation [6].
- The machine must have an easily-accessible emergency `stop` switch that stops all the mechanical moving parts immediately [6].
- The machine must be fully autonomous, and no interaction with an external PC or remote control is permitted during the operation [6].
- The operation must begin by pressing a `start` button on the keypad [6].
- The operation time shall not exceed 3 minutes [6].
- The time required for setting up the machine must not exceed 2 minutes [6].
- The machine must post no hazard to the operator, and shall not be perceived as hazardous [6].

5.3 Acceptance Criteria

Ontop of the minimum constraints listed above, there are two sets of acceptance criteria that we as a design team have considered. The first is the RFP public demonstration qualification criteria: The machine must detect 3 poles correctly, and deploy at least 2 tires correctly on 2 different poles. This includes the assumption that the machine did not freeze, touch the poles, exceed time limits, etc. This is considered a 'qualified' run by officials, and will score points in the competition. The second criteria is based upon the self-imposed objectives listed earlier. This design team would consider the prototype a success if it is able to score two qualifying runs in the competition, while achieving over 80% drop accuracy. The number

of poles will vary since the course is reset each run, but we would also like to achieve over 80% pole accuracy. Speed will likely be sacrificed for precise tire dropping in our case. As such, any time lower than 3 minutes for a full run will be considered successful. For the tire capacity objective, our team would prefer to aim for 25+ tires, in order to win points for the extendibility bonus feature.

6 Budget

Table 1: Quantity and description of all components used in the robot along with their prices.

Category	*Item*	*Price*	*Quantity*	*Total*
Building Materials	Wood, Fasteners, Nails	$10.00	1	$10.00
	Metal	$5.00	1	$5.00
	Rubber Wheels	$3.50	2	$7.00
	Caster Wheels	$3.29	2	$6.58
Electronics	PIC	$55.00	1	$55.00
	Wires, Solder	$15.00	1	$15.00
	LCD, Keypad	$8.00	1	$8.00
	Power Switch	$0.89	1	$0.89 [7]
	RTC	$4.00	1	$4.00
	Battery Holder	$0.67	1	$0.67 [8]
	H-Bridge Driver Board	$2.00	3	$5.99 [9]
	Encoder	$3.00	1	$3.00
	AA NiMH Batteries	$1.17	13	$15.22 [10]
	Protoboards	$0.50	6	$3.00
	Arduino Nano	$5.00	1	$5.00
	Arduino Uno	$5.99	2	$11.98 [11]
	Green LED strip	$8.00	1	$8.00 [12]
	Speaker	$0.80	1	$0.80 [13]
Motors	High-Torque Servo	$12.00	1	$12.00
	NEMA 17 Steppers	$9.33	2	$18.65
	Small Stepper Motor	$0.30	1	$0.30 [14]
	Micro Servo	$0.49	2	$0.98 [15]
	DC Motor (plastic, angle)	$4.00	2	$8.00
Sensors	10 Laser Pointers	$9.42	1	$9.42 [16]
	Laser Sensor	$4.56	1	$4.56
	Light Receiver	$1.32	3	$3.96 [17]
	Gyroscope	$2.19	1	$2.19 [18]
			TOTAL:	**$225.17**

The final price of our robot was $225.17, which falls below the $230.00 constraint set out

by the RFP. For prices in USD, the exchange rate used was \$1 USD = \$1.3313 CAD, which is the price reported by the Bank of Canada at the end of January 7th, 2019 [19]. A price breakdown of all components used in the robot can be found in Table 1.

Costs of labour involved with the designing and construction of the robot were not included in the budget. Miscellaneous components such as resistors, capacitors, transistors, diodes, and glue were not included in the price as their cost is relatively insignificant compared to the prices of other components used in the robot. For example, a pack of ten 1k resistors rated for 1/4W at 5% can be purchased for \$0.25 at a nearby store [20].

7 Division of the Problem

The task of addressing the challenge proposed by the RFP was split into three distinct roles with corresponding tasks and responsibilities. This was comprised of a Microcontroller member, responsible for the programming of the machine, an Electromechanical member, responsible for physical construction, and a Circuitry member, responsible for connecting all electronic hardware, Of course, team members shared duties when necessary and there was often overlapping of roles. Nonetheless, the following role descriptions provide a good overview of the general delegation of tasks and distribution of work.

Processing and Control (Microcontroller)

The microcontroller member of the team is responsible for developing the combinational and sequential logic required for the algorithms that run the robot. This includes, but is not limited to: polling sensors and encoders, driving DC motors, stepper motors, and servos, interrupts such as the emergency stop, and recording data such as distance during a run. He/she must be prepared to deal with and resolve blocking issues, as many components require constant polling or signalling. In addition to operational code, the microcontroller member must also design and implement a keypad/LCD user interface that provides information about the robot's last (and possibly previous) run(s). In addition to permanent logs, the microcontroller member may integrate other extra design features such as a real-time date/time display and a PC interface.

The microcontroller member must closely communicate with the circuits member to determine the microcontroller's pin assignment. He/she should be able to alleviate issues that arise from pin assignment using methods such as I^2C. The microcontroller member should be able to provide PWM signals to actuators, should they be required. He/she should also be familiar with other hardware requirements, such as the 4-input signal patterns of stepper motors. The microcontroller member should know the limits of the microcontroller

and various components, and should communicate any concerns to his/her team posthaste. He/she should also have basic knowledge and intuition regarding circuitry and construction, and provide assistance to his/her team members should they require it.

The microcontroller member must be comfortable with software and the process of writing code, including flowcharts, pseudocode, version control, and debugging. The debugging process may require extra code or components (such as LEDs): the microcontroller member should strive to make errors obvious so they can be resolved quickly. Bugs in code should be dealt with in a efficient, systematic way. He/she should follow good programming practice, including comments, syntax, and documentation. If issues with the design can be resolved through software, such as filtering sensor noise, the microcontroller member should be able to comply. He/she should also be actively involved in all aspects of the design process, even those that are not directly related to programming. The microcontroller member must also be able react quickly to the needs of his/her team, such as the addition of a new component that requires allocated pins. Lastly, he/she should always look for opportunities to optimize the operation of the robot, such as performing multiple actions simultaneously or reducing the dead time between tasks.

Mechanism and Actuation (Electromechanical)

The Electromechanical member of the team is responsible for planning, design, construction and calibration of the robot's physical structure. At an early stage, this involves generation of conceptual sketches and scale drawings. As the design matures, the Electromechanical member generates CAD models and finalizes aspects such as dimensions, movement of parts and overall layout. Space must be allocated in such a way as to permit free range of motion for moving parts, while keeping the total volume under constraints. Considerations will also be made for ease-of-use for other team members; components must be easily accessible and replaceable.

The bulk of the Electromechanical member's time will be spent in the machine shop. The first major construction elements for this role include the base frame, drive train (mounting of motors and associated circuitry) and supporting structure. Calculations must be performed in order to verify the suitability of actuators and materials. In addition, the main development board will be mounted to the frame. Once the mobile base is completed, work will begin on the upper structure of the machine. This includes elements such as the tire storage mechanism (rack-based magazine), tire deployment mechanism (extending rack-and-pinion arm with rotating tip) and structural supports. All actuators are to be installed as well, including servos, stepper motors and DC motors, along with their mounts and couplings. Finally, once all large-scale structures have been built and verified to be operational, the

Electromechanical member will install any remaining hardware. This includes many items, such as LCD screens, keypads, LEDs and sensors. Once all components are in place and testing begins, this member will need to work with their teammates to calibrate the physical elements of the robot and improve reliability.

The Electromechanical member will have to construct custom solutions for most of the challenges encountered in this competition. A key point of this member's role is to work towards a selection of materials, fabrication techniques and concepts that synergize; ideally, solutions should require minimal time investment, while delivering good results. They should draw from existing successful examples and adapt where necessary. This will require rapid prototyping and aggressive optimization throughout the process.

Instrumentation and Interfacing (Circuit)

The Circuits member of the team is responsible for constructing all digital and analog interfacing electronics to connect sensors, motors, and other components to the microcontroller board. Essentially, the Circuits member acts as the connection between the Electromechanical member and the Microcontroller member. They communicate with each other through the circuits that the Circuits member builds, so it is essential that they are reliable, robust, and safe.

Safety is especially important for the Circuits member because they will be dealing with voltages that could potentially cause serious harm or even death. With this in mind, it is imperative that the Circuits member exercise caution at all times when prototyping, constructing, and testing circuits. When integrating, the Circuits member should inform the other team members of the potential hazards so that work can be conducted in a safe environment.

Prototyping of circuits will primarily be done on breadboards in order to ensure their functionality before soldering them to protoboards and attaching them to the robot. The Circuits member should be constantly communicating with the Microcontroller and Electromechanical members to ensure that the circuits are compatible with the Microcontroller board and that the circuits being prototyped are appropriate for the motors, actuators, and sensors that will be used in the robot. When moving from prototypes on breadboards to protoboards that will be used in the actual robot, the construction of the circuits should be robust as to ensure that contact between components and wires is maintained over time as the robot is operated and moved around. It should be noted that the Circuits member must design and build at least one "open" circuit (i.e. made of basic components, such as transistors, resistors, capacitors, diodes, etc.) for a motor in the final prototype and prove its functionality.

8 Circuits Subsystem

8.1 Assessment of the Problem

The circuits subsystem involves the designing, prototyping, and construction of all electrical systems of the robot. The primary function of this subsystem is to connect the various microcontrollers to the mechanical systems of the robot. The circuit subsystem is also responsible for providing power to all the microcontrollers, sensors, and motors on the robot. This subsystem must also implement an emergency stop button that immediately stops all moving mechanical parts of the robot when pressed.

8.2 On-board Power

8.2.1 Power Supply Choice

One of the RFP's constraints is that the robot must use its own on-board power supply. As such, this on-board power supply should be able to provide a safe and stable source of electricity for the PIC microcontroller, Arduinos, motors, and sensors. Additionally, the on-board power supply should provide enough power to perform a minimum of two consecutive runs while minimizing weight so that the robot does not violate the 8kg weight constraint. The following types of on-board power supplies were considered:

1. AA Batteries (Alkaline)
2. AA Rechargeable Batteries (NiMH)
3. 9V Batteries (Alkaline)
4. Lead-Acid Battery
5. Remote Control Hobby Battery (NiMH)

NiMH AA rechargeable batteries were ultimately chosen. This was due to their ability to provide a relatively large amount of current, low cost, low weight, and compact dimensions. Additionally, the discharge curve for NiMH batteries is flatter compared to Alkaline batteries. A comparison between Alkaline and NiMH battery discharge curves can be seen in Figures 4 and 5. A flatter discharge curve is preferred because there will be less variation in voltage supplied to components as the batteries are drained during operations.

Initially, ten AA NiMH batteries were placed in series to produce a voltage of approximately 12V. As mentioned previously, the discharge curve for NiMH batteries is relatively flat, so any motors requiring 12V were powered directly off the batteries without any voltage

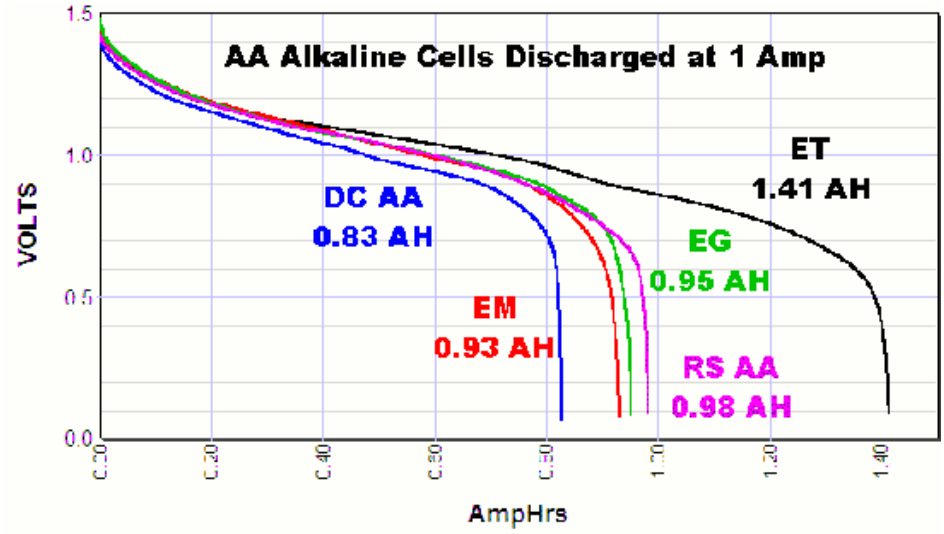


Figure 4: Discharge curve for various brands of AA alkaline batteries at 1A.

regulation. An LM338 adjustable voltage regulator was configured to step the 12V from the batteries down to 5V in order to power the remaining Arduinos, motors, and sensors on our robot. The LM338 was chosen over the LM7805 because of its ability to regulate a continuous current of up to 5A, whereas the LM7805 is only designed to regulate 1A [21] [22]. The equation and calculations used to determine the values of R_1 and R_2 can be found in Equation 1 [22]. Since we did not have a 990Ω resistor for R_2, a 1kΩ resistor was used, which resulted in a V_{out} that was slightly above 5V. Capacitors were used for V_{in} and V_{out} to minimize noise. The 5V regulator's schematic can be found in Figure 6. A heat sink was placed on the voltage regulator to ensure that it did not overheat.

$$V_{out} = V_{ref}\left(1 + \frac{R_2}{R_1}\right) + I_{adj}R_2 \tag{1}$$

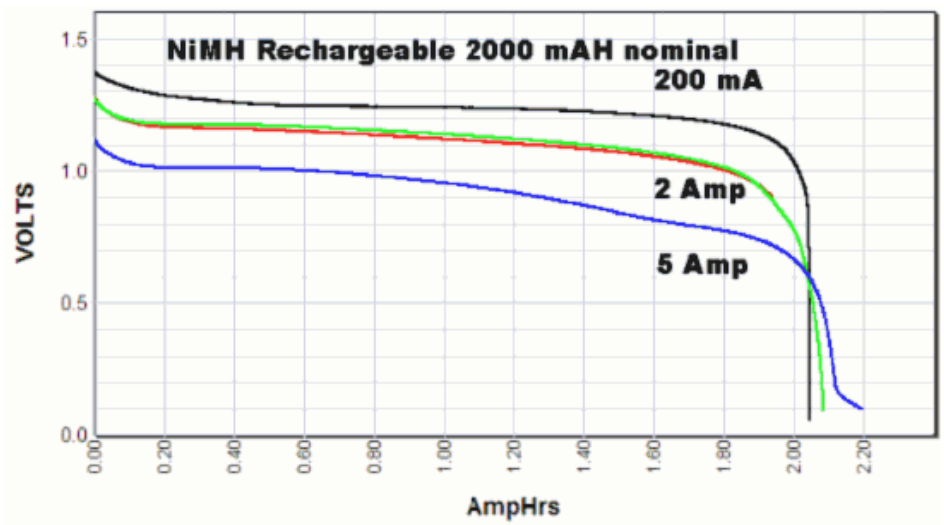


Figure 5: Discharge curve for an AA NiMH battery at various currents.

$$
\begin{aligned}
I_{adj} &\approx 0 \\
\therefore V_{out} &= V_{ref}\left(1 + \frac{R_2}{R_1}\right) \\
\frac{V_{out}}{V_{ref}} &= 1 + \frac{R_2}{R_1} \\
\frac{R_2}{R_1} &= \frac{V_{out}}{V_{ref}} - 1 \\
\text{Assume } R_1 &= 330\Omega \\
V_{out} &= 5V \\
V_{adj} &= 1.25V \\
\therefore R_2 &= 990\Omega
\end{aligned}
$$

8.2.2 Power Supply Changes

During integration, the discharge curve of the batteries made it very difficult to reliably stop the robot in front of the poles. Essentially, the changing voltage throughout operations resulted in the DC motors spinning at different speeds. This made calibrating the stopping of the robot very difficult. This issue is discussed in greater detail in section 8.3. In order to resolve this problem, two options were considered:

1. Add a voltage regulator to regulate the 12V from the batteries down to 9V or less,

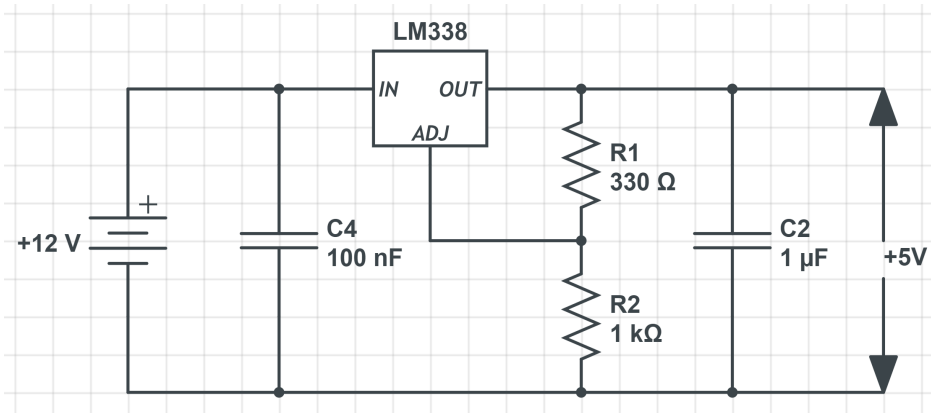


Figure 6: 5V regulator schematic. It uses an LM338 adjustable voltage regulator which is capable of providing up to a continuous 5A of voltage regulation.

which will then be supplied to the DC motors

2. Add additional batteries to produce a higher than necessary voltage, then regulate that voltage down to approximately 12V for the DC motors to use

The first option's advantages were that it was less complex and cheaper. 9V was chosen as the voltage to regulate to because the recommended operating conditions in the LM338's datasheet states that the minimum input-to-output voltage differential should be 3V [22]. However, it would result in the DC motors not being able to run at their maximum 12V rating [23]. This would increase the amount of time it takes to traverse four meters at the end of the operation when moving the robot backwards to stop behind the finish line.

The second option would allow the DC motors to operate at their maximum speed. However, it would require adding additional batteries the robot which would increase the complexity and cost of the robot. Furthermore, adding additional batteries would also increase the usable lifespan of those batteries. In order to save time, we push the magazines forward while driving between poles. From lots of testing, we noticed that the steppers would stop functioning properly when operating simultaneously with the DC motors when the batteries' voltage dropped to just below 12V. By regulating the voltage to slightly above 12V, we will be able to ensure that the voltage stays at a usable voltage until the batteries become more significantly drained.

We decided to go with the second option in order to maximize our robot's speed and reliability, despite the additional cost and complexities. It was decided to add three additional AA batteries to our robot, bringing the batteries' total voltage to approximately 15.6V. The primary factor influencing our decision on how many batteries to add was the minimum recommended input-to-output voltage differential of 3V on the LM338 [22]. Adding three

batteries ensures that the batteries will remain well above this minimum voltage differential for the majority of their lifespan.

The calculations to determine R_1 and R_2 are very similar to the calculations shown in Equation 1. We were limited to using resistors that were available in our project kit, which made it very difficult to design the regulator's circuit to step the voltage down to exactly 12V. Due to the previously mentioned issue of running the DC motors and magazine stepper motors when the voltage dips below 12V, a voltage of slightly above 12V was chosen. A table of possible values for R_1 and R_2 can be found in Table 2. R_1 and R_2 were chosen to be 0.33kΩ and 2.92kΩ respectively. It should be noted that although some of the values for R_1

Table 2: Table of all values for R_1, R_2, the their theoretical V_{out} that were considered when designing the 12V regulator. R_1 and R_2 were eventually chosen to be 0.33kΩ and 2.92kΩ, respectively.

R1 [kΩ]	*R2 [kΩ]*	V_{out} *[V]*
10	88	12.25
3.3	28.6	12.08
100	847	11.84
0.1	0.92	12.75
0.1	0.76	10.75
0.1	0.86	12
0.33	3.22	13.45
0.1	0.88	12.25
0.33	2.9	12.23
0.33	**2.92**	**12.31**

and R_2 in Table 2 seem to work in theory (e.g. 10kΩ and 88kΩ in the first row), this was not the case in practice. It was observed that using resistors with relatively high values did not produce the expected V_{out}. As such, resistors with lower values were used to construct the voltage regulator circuit. It is suspected that the reason for the higher resistor values not producing the expected results is because they caused I_{adj}, which the LM338 uses to regulate its output, to be too small for it to properly use. Capacitors were used for V_{in} and V_{out} to minimize noise. A schematic for the 12V regulator can be found in Figure 7.

8.2.3 Brownout Issues

After adding the 12V regulator, we ran into issues when testing full runs where the robot was supposed to drive up to poles, drop tires, then drive back to the start line. The robot would drive up to poles and drop tires without any major issues, but it would not return back to the start line despite being explicitly told to in the code we were using. However,

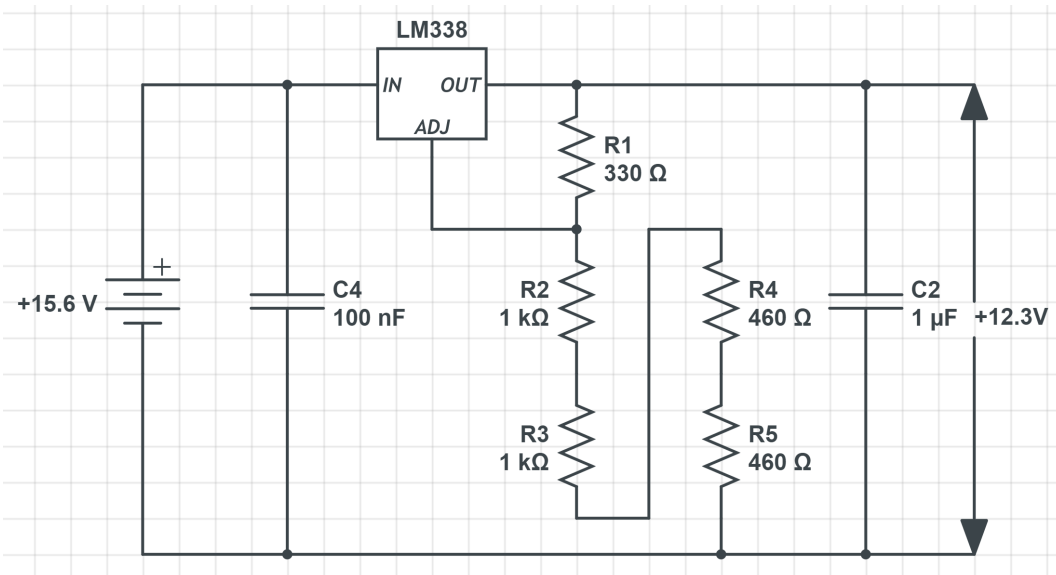
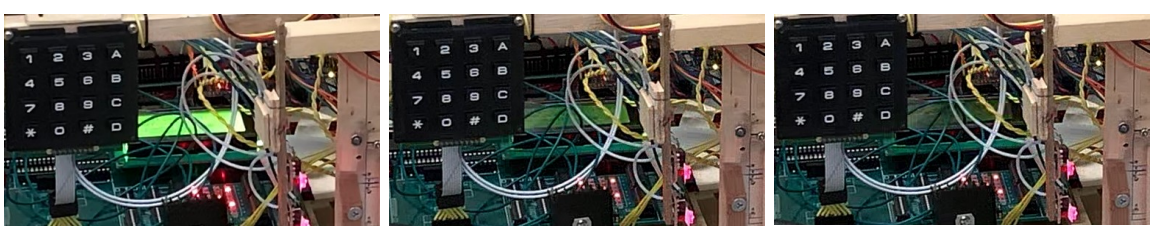


Figure 7: 12V regulator schematic. It consists of a LM338 adjustable voltage regulator which is capable of providing up to a continuous 5A of voltage regulation. Multiple resistors were combined in series because a single resistor with a value of 2.92kΩ was not available.

when we plugged the PIC development board or Arduino Uno into an external power source, this resolved the issue and the robot would drive back to the start line without any issues.

We reviewed footage we took of these runs when a battery bank was plugged into the Arduino Uno's power source and noticed that the LCD on the PIC development board was getting extremely dim when the robot initiated its drive back to the start line. Three frames of the footage that we reviewed where the LCD gets very dim can be found in Figures 8a, 8b, and 8c. The video was recorded at 60FPS, which means that these three consecutive frames occurred within approximately 50 milliseconds.



(a) First frame (b) Second frame (c) Third frame

Figure 8: Video analysis of the LCD dimming as the robot drives back with the Arduino Uno powered by an external power source. Recorded with an iPhone XS at 60FPS.

This footage along with the disappearance of the issue when plugging the PIC development board or Arduino Uno into an external power source led us to suspect that the PIC microcontroller and Arduino Uno were not receiving enough power to operate properly.

After some discussion and testing, we resolved the issue by giving the DC motors more time to accelerate to their full power when driving back. This minimizes the voltage drop experienced by the other sensitive components of the robot.

8.3 DC Motors

The robot is driven by two Shenzhen DC Angled Motors (GP02S-A130 4150-120) connected to the front wheels. They are controlled by a L298 dual motor driver breakout board. It is also capable of PWM control. An image of this board can be seen in Figure 9. A description

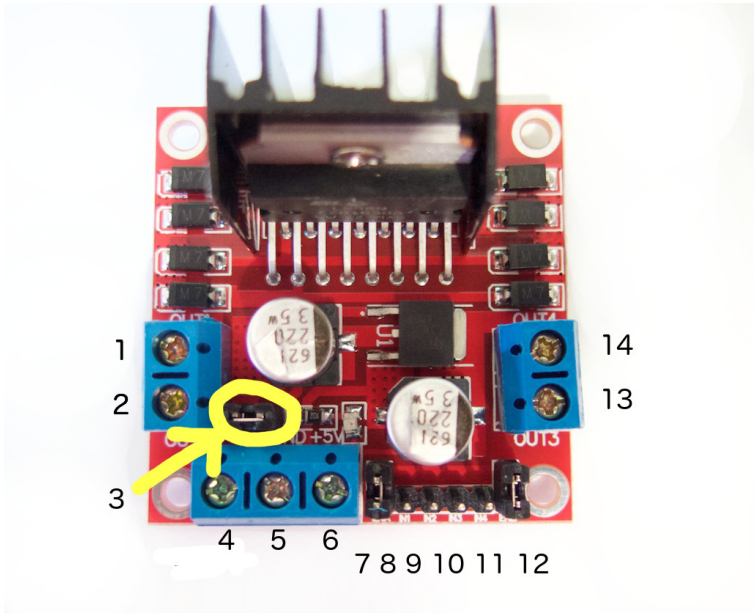


Figure 9: Top view of the L298 dual motor driver breakout board. It is capable of driving two DC motors at 7-24V at a maximum of 2A per motor [24].

of each label in Figure 9 is as listed:

1. DC motor A (+)
2. DC Motor A (-)
3. Jumper that enables power to the on-board 5V regulator
4. Supply voltage (+)
5. GND
6. 5V output (if 5V regulator is enabled)
7. DC Motor A enable jumper (can be connected to a PWM output for DC motor control)

8. IN1
9. IN2
10. IN3
11. IN4
12. DC Motor B enable jumper (can be connected to a PWM output for DC motor control)
13. DC Motor B (+)
14. DC Motor B (-)

The IN1, IN2, IN3, and IN4 pins are used to control the DC motors. IN1 and IN2 control motor A. IN3 and IN4 control motor B. Table 3 provides descriptions on what signals should be provided to these pins in order to move the robot forwards, backwards, and stop.

Table 3: Description on how to control two DC motors using the L298 dual motor driver breakout board.

Action	*IN1*	*IN2*	*IN3*	*IN4*
Forward	HIGH	LOW	HIGH	LOW
Backward	LOW	HIGH	LOW	HIGH
Stop	LOW	LOW	LOW	LOW
Stop (sudden)	HIGH	HIGH	HIGH	HIGH

During integration, we ran into some issues related to stopping our robot when it detects a pole. When the laser sensor detects a pole, there is a fixed delay before a stop command is sent to the DC motors. We had trouble getting the robot to stop precisely in front of the pole. After some debugging, we noticed that the issue became amplified whenever the battery voltage changed significantly (e.g. putting in a fresh set of batteries or when they became completely drained). We determined that the difficulties we were encountering when trying to stop the robot precisely in front of the pole was due to the DC motors operating at varying voltages as the batteries drained. This resulted in the robot's speed being dependant on the battery level, which made it very unpredictable and difficult to calibrate the delay.

In order to resolve this issue, we decided that we needed to change the design of our robot so that DC motors would be minimally affected by the batteries' voltage level. We added three AA batteries to produce a voltage of approximately 15.6V, then use a voltage regulator to step it down to around 12V. This ensures that the DC motors will receive a stable voltage regardless of the batteries' charge levels. The details of this 12V regulator circuit can be found in Section 8.2.2.

8.4 Stepper Motors

A mix of unipolar and bipolar stepper motors were used in the construction of the robot. A unipolar stepper motor was used for the arm extension and two bipolar stepper motors were used to move tires in the magazines.

8.4.1 Unipolar Stepper Motor

A 28BYJ-48 unipolar stepper motor was used for the arm extension. It is controlled by an open circuit consisting of four NPN transistors. The schematic for the open circuit can be found in Figure 10. The circuit consists of four TIP142 NPN transistors which control which side of the coil is powered, as seen in Figure 11. By turning on the four transistors in the correct sequence, the motor can be rotated.

A stepper motor was used for the arm extension because the arm requires precise movements to reliably drop tires on poles. A unipolar stepper motor was chosen over a bipolar one because extending and retracting the arm requires a relatively low amount of force. Bipolar stepper motors tend to provide better performance, but they are more complicated to drive and require two H-bridge circuits. On the other hand, unipolar stepper motors can be driven with just four transistors connected to the ends of each coil.

8.4.2 Bipolar Stepper Motors

Two 42SHD0001-24B NEMA 17 bipolar stepper motors were used to load and push tires in our robot's magazines. The end of each bipolar stepper motor is connected to a threaded rod which controls a pusher plate. The pusher plate moves the tires forward during operations and backwards between operations in order to load tires, as seen in Figure 18.

Loading and deploying tires from the magazines requires fast and precise movements. For this reason, bipolar stepper motors were chosen because they are able to provide the precision of stepper motors while also being relatively fast compared to the unipolar stepper motor used for extending the arm. Speed is especially important for the magazines because the angle of the threads on the rods is quite shallow, so it takes a lot of rotations to move forward the thickness of one tire. As such, it is essential that they move as fast as possible while maintaining enough torque to push the tires forward. The magazines were the bottleneck of our system when it came to performing operations.

Each bipolar stepper motor was controlled using an L298 dual motor driver breakout board, as seen in Figure 9. The red, yellow, grey, and green wires of the motor are connected to the L298 driver breakout board as shown in Figure 12.

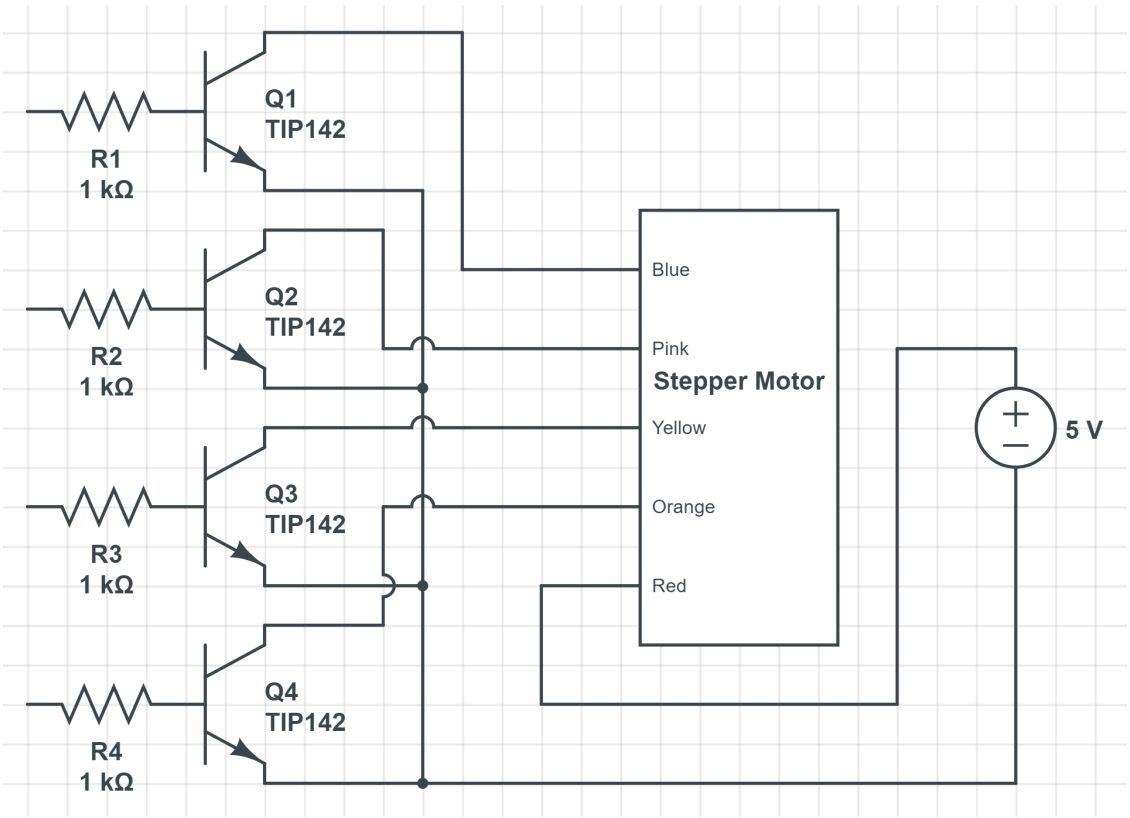


Figure 10: Open circuit for the unipolar stepper motor that controls the robot's arm extension. It consists of four TIP142 NPN transistors. R1 through R4 are connected to a microcontroller where they can be triggered in the correct sequence in order to rotate the motor.

8.5 Servos

There are three servos on the robot: two K-power P0090 micro servos and one MG996R high torque metal gear dual ball bearing servo. The first micro servo is used to grip the tire before the cell is rotated and then release it once it is in the proper position to be dropped. The second micro servo is used as a kicker to propel the tires down the ramp to ensure that they have a much greater chance of reaching the cell. The high torque servo is used to rotate the cell once a tire is inside.

Servos were used in these parts of the robot because they are able to provide very precise movements down to a degree of resolution. Additionally, they have feedback circuits built into them so they are able to sense what position they are currently in. This allows them to not only make precise movements like stepper motors, but also fight to maintain that position. Stepper motors do not have any sense of where they are currently located, and if the motor misses some steps then it is very difficult, if not impossible, to determine the exact position of the motor. For movements like rotating the cell and gripping the tire, we wanted

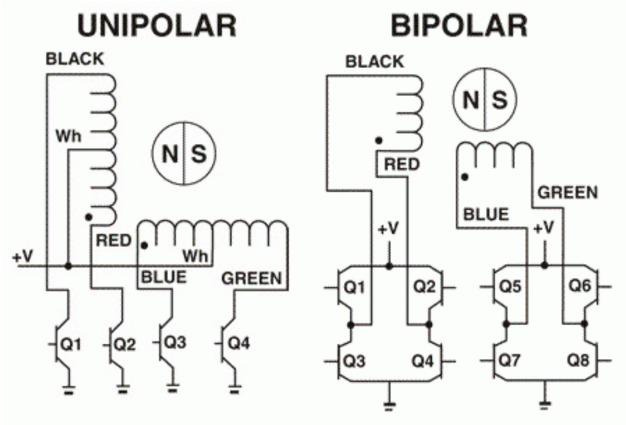


Figure 11: Diagram comparing the interior layout of unipolar and bipolar stepper motors [25].

to ensure that we would always be able to establish a firm grip on the tire, then rotate it precisely 90 degrees. Even if someone bumps into the cell or there is an unexpected external force acting against the servo, it will always fight to go to its programmed position.

Circuits for the servos are relatively simple. They only require three connections:

- +5V
- GND
- PWM control

8.6 Sensors

A variety of sensors were tested throughout the duration of the course. The sensor suite selection process is discussed in more detail in Section 10.3.5. Similar to servos, sensor circuits are relatively simple.

The three break beams consist of lasers and laser receivers. The lasers only require a 5V connection and the laser receivers require 5V and an additional pin that goes to `HIGH` when the laser is blocked.

The VL53L0X laser-ranging module that was used to detect poles and stop the robot only required 5V and I^2C connections. The MPU-6050 gyroscope that assisted with driving straight also only needed 5V and I^2C connections.

8.7 Speaker

The speaker circuit consisted of two sections: an audio amplifier and a transistor to cut power to the speaker when it was not in use. Schematics for the speaker circuit can be seen

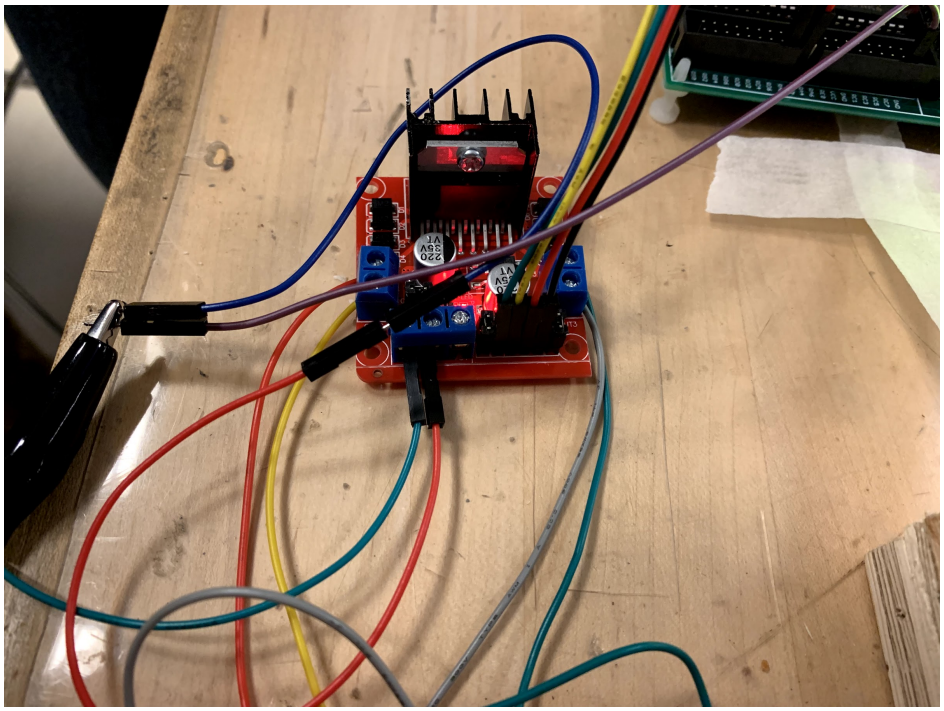


Figure 12: Photo of a bipolar NEMA 17 stepper motor connected to an L298 dual motor driver breakout board. The connections should be made as seen in this photo to ensure that it operates properly.

in Figure 13. A number of modifications were made to the circuit shown in Figure 13 due to the components not being available. A 1μF capacitor was used in place of a 22μF capacitor for C1. A 1kΩ resistor was used in place of a 2.2kΩ resistor for R1. A TIP142 transistor was used in place of a 2N6292 transistor for Q1. These changes were made with careful consideration so that the original functionality of the circuit would be maintained as much as possible.

The other section of the speaker circuit is a single transistor that can cut power to the speaker when not in use. This was added after noticing that the TIP142 amplifying the audio signal to send to the speaker would still get quite hot even when not in use. In order to minimize power losses, an additional transistor was added in series with the speaker to completely cut power to it in order to minimize losses. This change was successful as the audio amplifier transistor no longer got hot when not being used.

8.8 Suggestions for Improvement of the Subsystem

Given more time and resources, there are several improvements that could be made to the circuits subsystem:

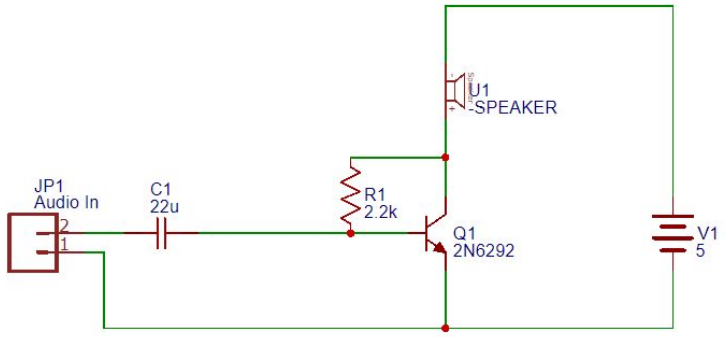


Figure 13: Schematic of the circuit used to amplify audio signals that were sent from the Arduino [26]. A number of changes were made to the circuit due to a lack of available components.

1. **Terminal Blocks:** One protoboard was used as the main power distribution board on the robot. By the end of the project, nearly every hole in the protoboard had a wire coming out of it. This made it quite rigid and difficult to solder wires to. Rather the soldering every wire to a main power distribution board, it would be easier and faster to solder a few terminal blocks so that multiple wires can be connected to a single terminal. Additionally, terminal blocks are much easier to work with when prototyping because once in place, they do not require any additional soldering to securely attach wires.

2. **Independent Voltage Regulator Connections:** The 12V regulator was added to the design once many of the circuits had already been made. As a result, the 5V regulator's V_{in} comes from the 12V regulator's V_{out}. This is not ideal because whenever a device on the 5V rail needs to draw power, it needs to go through both the 12V and 5V regulators. This puts an unnecessary load on the 12V regulator. The 5V and 12V regulators should have been connected in parallel to the batteries so that they are not dependant on each other. In other words, V_{in} for both regulators should come directly from the batteries.

3. **Improved Audio Amplifier Circuit:** The audio amplifier circuit used in the robot was functional but was far from perfect. As mentioned in Section 8.7, the transistor amplifying audio signals from the Arduino got hot even when the speaker was not in

use. An additional transistor was added to cut power to the speaker when not in use, but this was more of a temporary patch rather than addressing the root cause of the issue. If another iteration of the robot were to be made, more time would be spent researching and designing the audio amplification circuit to make it more efficient and ensure that it matches the specifications of the speaker. The schematic that the audio amplification circuit was based off of was for a 10W speaker [26]. However, the speaker on our robot is only rated for 0.5W. We were only using the speaker for short periods of time so this was not an issue, but over time it is likely that the speaker would wear out very quickly.

4. **Durable Ribbon Cable Breakout Board:** One of the protoboards on the robot acts as a breakout board for the ribbon cable so that wires can be soldered to headers as opposed to connected directly to the ribbon cable or the PIC development board. Although it was functional, this protoboard took a lot of time and effort to create. Despite this, it was not very durable and the connections had to be repaired once. Although it was convenient to have a secure connection to the PIC without having to solder directly to the PIC, there are many pre-made breakout boards that provide the same functionality in a durable package and at a low price.

5. **Power Distribution & Cable Management:** Many of the components on the robot had a direct connection to the main power distribution board. Initially this was not an issue, but as the number of components on the robot increased, managing wires quickly became an issue. It was noted that some protoboards beside each other had their own connections to the main power distribution. In retrospect, it would have been neater and a more efficient use of wires to only have one pair going to each major section of the robot and daisy-chaining nearby devices.

9 Electromechanical Subsystem

9.1 Assessment of the Problem

The Electromechanical subsystem of the robot involves the planning, design, construction and calibration of the physical structure. From a physical perspective, the main problem for this particular RFP is controlling the movement of the tires, both within and outside the robot (ie. storage, retrieval from storage, deployment). Several tire properties make this a highly challenging task:

1. **Friction**: The rubber of the tires has a high degree of friction; as a result, they do not roll well on rougher surfaces. If collision forces are applied, the effect of friction is more pronounced; tires can become stuck even on sloped surfaces such as ramps. This makes it harder to transport tires within the robot, if any sort of rolling is used.

2. **Low Mass**: Typically, higher mass would not be a desirable property for cargo. However, in the case of these tires, a higher mass would have allowed them to roll more predictably. With additional momentum, issues such as tires getting stuck in shafts and ramps wouldn't be a problem. Unfortunately the tires used are very light.

3. **Softness**: When observed under slow-motion, it was noted that the tires compress to a high degree when subjected to any sort of stress. They are nowhere close to being rigid, which makes it significantly more difficult to grip predictably. Controlling the path of the tire after a collision or drop is also much harder.

4. **Degradation and Permanent Deformation**: The lack of rigidity in the tires leads to another problem: With enough time they gradually deform and lose even more rigidity. The rubber seems to age quickly and lose strength, after only a few months.

5. **Bounciness**: The effect of compression results in a tendency to bounce when colliding with other materials. This once again makes the path of travel unpredictable, and discourages large drops from heights.

6. **Tire Geometry**: With the diameter of the pole and tire taken into account, there is very limited room for error (2.1 and 4.5 cm, respectively). The accuracy of the robot must be very high if it is to make successful drops. In addition, the tires have a groove on the inner ring, causing them to easily hook onto edges and catch on exposed surfaces. This could be problematic if the tires were to be shuffled along a rod or pole; they may not traverse smoothly (if at all).

Dropping onto the pole can also be problematic; if the aim of the machine is slightly off, the tire can 'hook' onto the top of the pole, denying any further tire deployments. The low surface area of the tire rims may impact sensing from above the pole; sensors will likely have to approach from the side if they are to be reliable. Finally, the tread pattern on the outside of the tires means that they may catch on any surfaces that are not perfectly smooth. This is one of the most troubling issues.

The arrangement of tires within the storage mechanism is also a key issue. The tires are approximately 2.4 centimetres in thickness. Stacked end-to-end, 25 tires would not fit within the space constraints (the stack would be approximately 60 centimetres long, exceeding the 50 cm restriction). Due to this, the storage mechanism will likely need to be split up, into at least two stacks (or creative angles could be used). This introduces additional complexity, since it will likely require more actuators.

7. **Black Colour**: The black colour of the tire means that some sensors (e.g. IR) may not be able to identify them well, unless they are at extremely close range.

8. **Variability in Manufacturing**: It was noted upon examination that even freshly unpackaged tires exhibited a noticeable amount of deviation from each other. This difference in size and shape must be accounted for when constructing any physical elements that contact the tires.

With the above factors taken into account, deploying and storing tires present a major difficulty. Determining a viable selection of materials, actuators, robot geometry/size will be essential to overcoming this. A second major challenge is the detection of poles. The following are a number of key observations:

1. **Placement Variation**: As per the RFP, the poles may be positioned along a line with up to 5 centimetres of variance. The machine must be able to compensate for this deviation, by drivetrain (e.g. omniwheels) and/or extension features (e.g. mobile arms). The machine must have some means of measuring the deviation; sensors with a high degree of accuracy are required. There are also additional variations; if the pole is not fully pushed into its base, it may be higher than intended. This could impact the robot if it does not have sufficient clearance.

2. **Pole Base**: The plastic bases of the poles have a diameter of 11.5 cm. This means that a mobile machine cannot directly drive up and align; there must some sort of overhanging mechanism that can reach beyond the base. This likely takes the form of an extending arm or overarching-frame design.

3. **Pole Colour**: The poles are black in colour, restricting the ability to use IR and similar sensors.

4. **Pole Hollow** The top of the pole is hollow, which means that downwards-firing sensors can have difficulty determining their position. Sensors may 'see' into the hole at the centre of the pole, and return erroneous readings. Therefore sensors will likely need to be mounted from a side-firing position.

With these observations in mind, a successful final design requires a high degree of control and flexibility. There are also additional points beyond these competition requirements: Clearance for moving components must be factored into the initial design. Additional hardware such as LCD screens, keypads, LEDs, sensors, control boards, etc. must be mounted and organized. Cable management will be a continuous process throughout construction. All considerations must be made with ease-of-use, modularity, and compatibility with other subsystems in mind. Aesthetic value will be taken into account, but not at the sacrifice of usability.

9.2 Solution

This section details the various physical components of the B.A.S.S. A basic overview of the specs can be found in Table 4.

Breakbeam Assisted Smart Stacker Spec Sheet	
Property	**Specifications**
Length	48 cm
Width	46 cm
Height	48 cm
Mass	5.1 kg
Materials	Wooden frame, metal fasteners and screws, Epoxy
Drivetrain	Front wheel drive, 6 cm diameter rubber wheels, 2x Shenzhen DC GP02S-A130 4150-120, torsion bar suspension, rear fixed casters
Tire Storage	26 tires at maximum capacity, twin threaded-rod magazines driven by Casun NEMA 17 42SHD0001-24B, tire kicker using K-Power P0090 microservo
Sensor Array	3x laser pointer and receiver, 1x VL53L0X TOF laser distance sensor
Deployment	Extendable arm via rack and pinion (1x 28BYJ-48 Stepper), tire gripper (1x K-Power P0090 microservo), rotating cell (1x Towerpro MG996R High Torque Servo)

Table 4: B.A.S.S. Primary Specifications Table

9.2.1 Structural Frame

The underlying frame of the robot is a basic rectangular prism, with wooden beams forming the major edges. The frame measures 36 cm in length, 29 cm in width, and 18 cm

in height (not including superstructure, protrusions, additional hardware). The frame is screwed together at the vertices. To save weight, thinner wood was used wherever possible. The load-bearing beams forming the bottom rectangle are 2x2 cm in thickness, whereas the lighter superstructure is only 1x1 cm. Twin beams cross the front of the robot, separated by a small gap. These beams provide structural rigidity, but also serve as a track for the linear gear, which will be detailed in a later subsection (9.2.4).

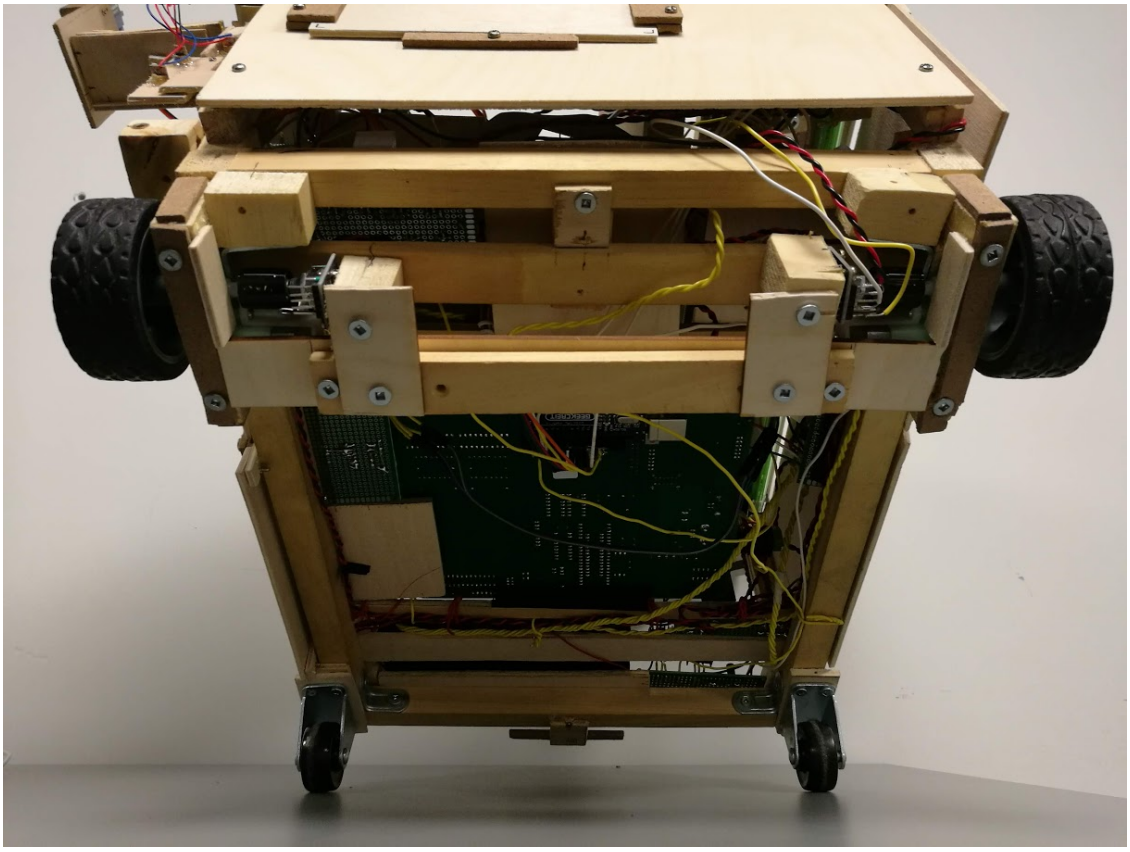


Figure 14: The overall layout of the robot bottom. Note the main rectangular frame, drive wheel mounts, and the suspension bar running across the width of the robot.

Cross beams at the bottom of the robot serve as mounting points for the PIC Development Board, Arduinos, batteries, and other associated circuitry (see Figure 14). Smaller rigid surfaces are mounted throughout the robot, providing attachment points for motor driver boards, sensor circuitry, etc. A pedestal containing key controls rises from the right rear corner of the robot, to an easily-accessible height. Mounting points are also provided for the LCD and keypad. By relocating them from the bottom-mounted PIC, the user experience is greatly enhanced. The LCD is mounted at a raised angle to permit easy viewing, without the user needing to bend down.

The entire frame is covered by exterior plating (Figure 15). These panels are designed to

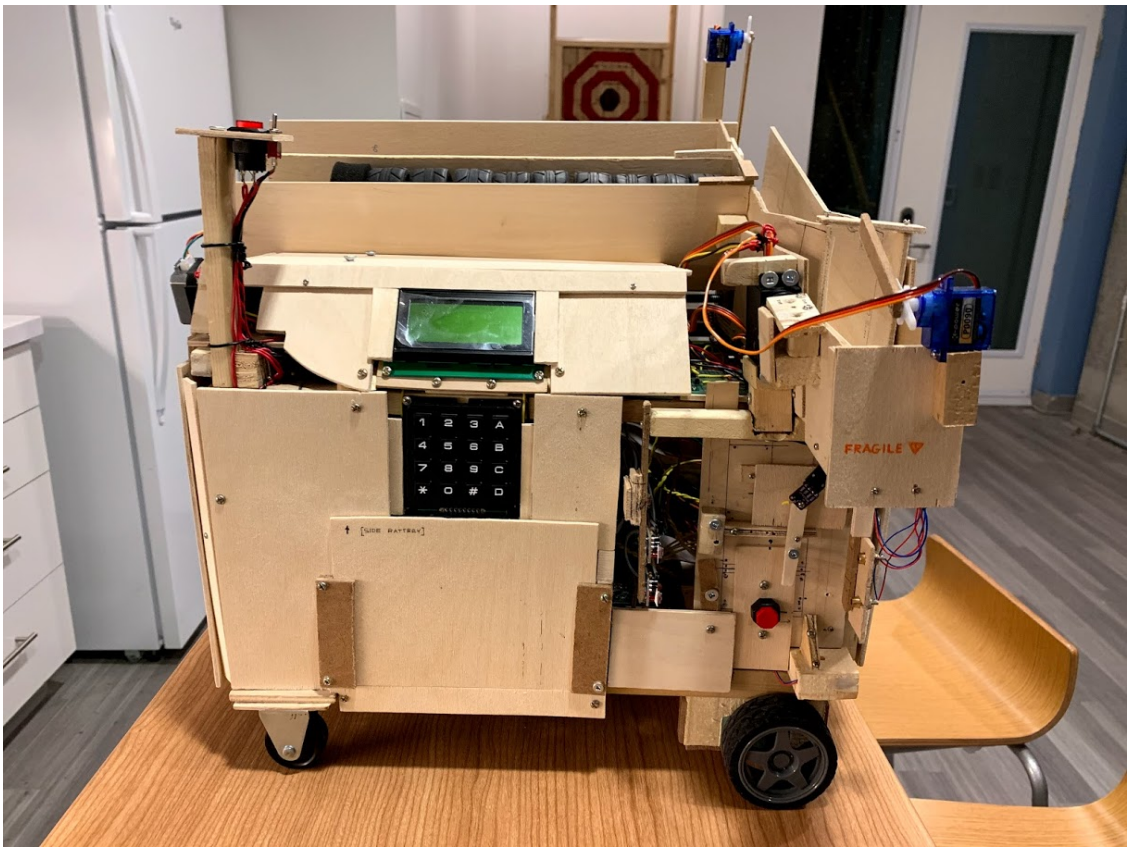


Figure 15: Right side view of the robot, illustrating keypad/LCD mount, exterior playing and arm retraction slot. Also note emergency stop pedestal.

be highly modular, sliding in and out of slots. There are 5 removeable panels: Rear (Main Battery), Right (Side Battery), Front (Linear Gear), Left (I/O) and Top (Stepper Driver Boards). When access to internals is required, a user can remove the panels within seconds. When not in use they are firmly secured.

9.2.2 Drivetrain

The machine has two front drive wheels (6 cm diameter rubber tires) and two rear fixed caster wheels. Two Shenzhen DC Angled Motors (*Model GP02S-A130 4150-120*) are directly connected to the front wheels. They are housed in twin motor mounts that extend beneath the main frame. A major issue noted during the early construction of the robot was the alignment of the wheels. To ensure that the wheels do not sag (excess camber), a torsion bar is secured to both motors. This acts as a suspension, twisting in order to absorb impacts and loads. Aligning the wheels also has the effect of making the robot drive straighter. This is critical for the tire stacking challenge, since the poles are arranged in a relatively straight line. As a further precaution, the robot also uses a gyroscope (MPU-6050) in order to adjust

power to the wheels and correct any drifting.

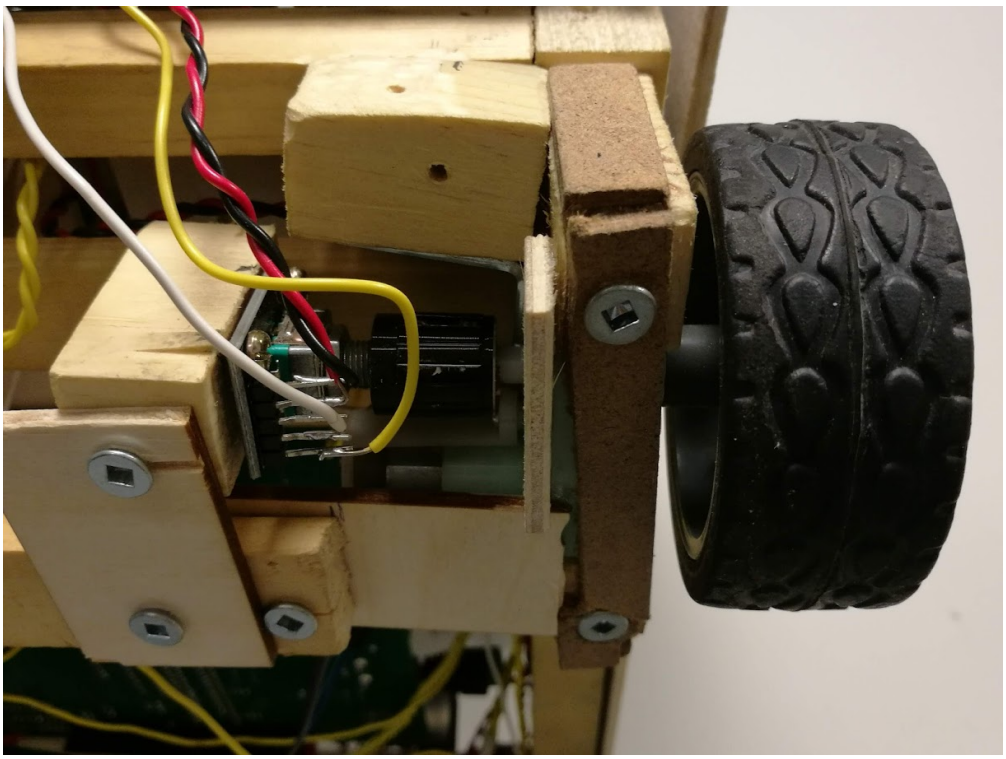


Figure 16: Close-up of the motor mount.

A detailed view of the motor mount is visible in Figure 16. The green portions are the DC motor itself. The torsion bar which minimizes camber is visible on the bottom (the thin flat piece of wood), running across the width of the robot. The MDF strips and wooden block on the right are the main mount for the motor, providing vertical load support. The block on top (with a drill hole) has a metal L-bracket that prevents z-axis rotation of the motor. The block on the left, attached to the torsion bar, is a mounting point for an encoder. The black cylinder in the center is the coupling between the motor axle and the encoder. This mounting style allows the encoder to read directly from axle rotations, while also making it easy to access the pins.

9.2.3 Tire Storage

The B.A.S.S stores up to 26 tires when loaded to maximum capacity. A tire can be pre-loaded into the rotating cell, and another in the deployment ramp itself. The remaining 24 are split between double horizontal magazines. The entire magazine assembly is mounted on the top left of the robot, with the floor tilted down towards the right. This tilt allows the tires to roll out down a ramp (see Figure 17).

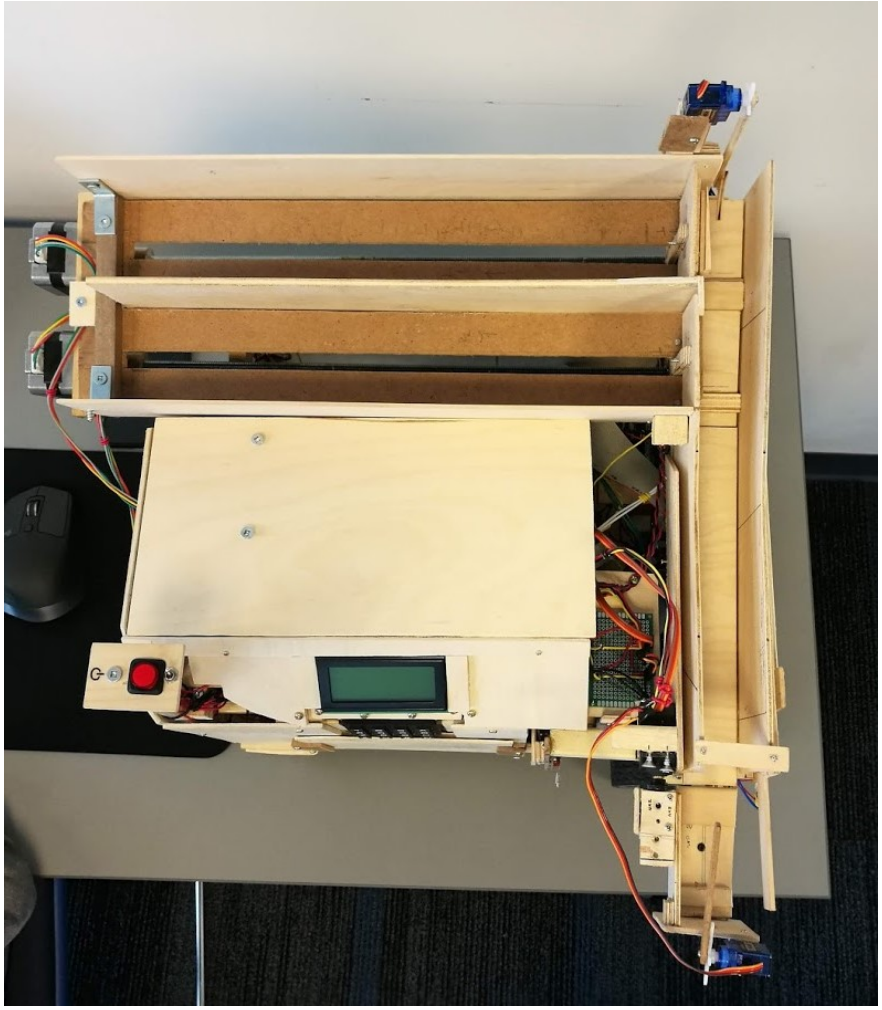


Figure 17: An overhead view of the robot, illustrating the two magazines, ramp and rotating cell.

The tires are moved along using pusher plates. Each plate slides along a track in the floor. Underneath the floor, the plate is mounted to a nut. This nut spins along a threaded rod, driven by a Casun NEMA 17 42SHD0001-24B stepper motor (see Figure 18). The rods are greased for quiet and smooth operation. The floors of the magazines use MDF in order to reduce tire contact friction.

The pusher plates have additional tabs attached to the front. These serve two purposes: First, they ensure that the last tire is always successfully ejected from the magazine. It was noted during testing that sometimes the plates would struggle to push to the end, as they had reached the limit of the floor track. To correct this, the additional tabs extend the reach of the plates forward, allowing them to push cleanly into the ramp. The second purpose of the tabs is to guide the tires away from the push plates and magazine walls. To accomplish this, the tabs have been sanded down until they are sloped.

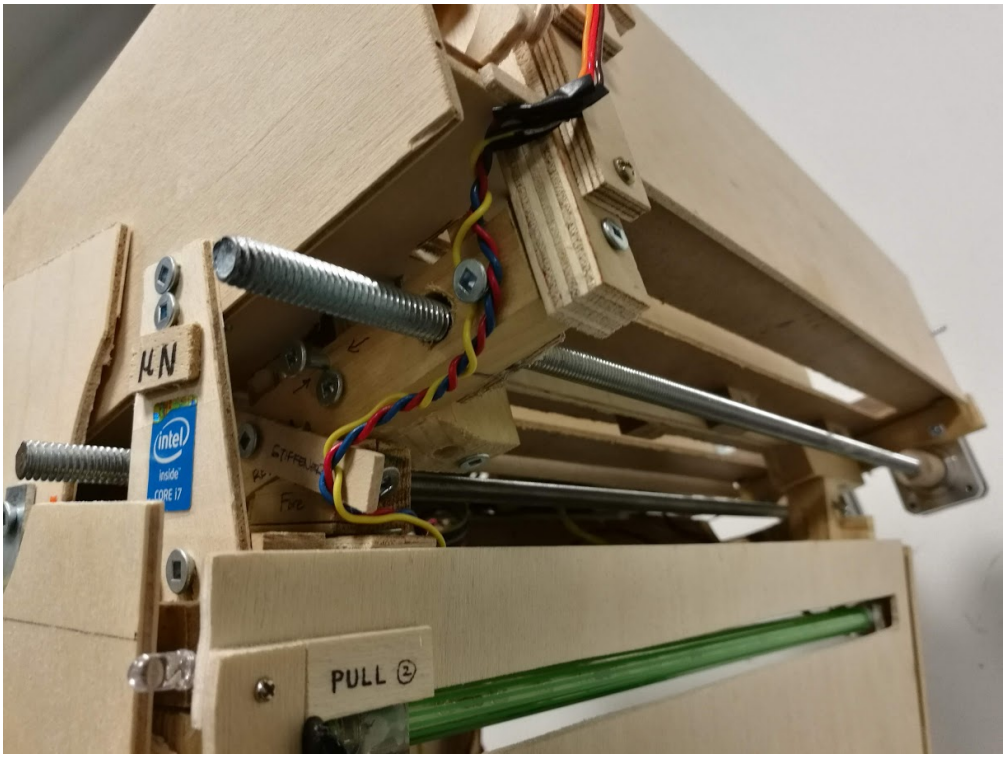


Figure 18: This photo, looking back along the left side of the robot, illustrates the threaded rod mount. The pusher plates are not visible as they are in the full forward position. Note the tilt of the magazines, the MDF floor and its cut-out track, and the NEMA steppers located in the rear. The wiring down the middle connects to the kicker arm above.

Once a tire is fully ejected from the magazine, it arrives at a ramp that leads to the rotating cell (see 9.2.6). However, small 'speedbumps' on the ramp floor prevent the tires from rolling down. A K-Power P0090 microservo fitted with a 'kicker' arm then sweeps the ramp, lifting the tire over the bump and launching it towards the cell. The kicker arm curves to match the shape of the tires; this allows it to perform a clean lifting motion. The kicker was implemented to prevent jamming; it was noted during testing that excessive friction could cause tires to stop rolling down the ramp. This would be catastrophic, since the robot would be unable to reload and the remainder of the run would fail. The kicker arm and pusher plates can be seen in Figure 19.

Because of the lack of exposed moving parts or rods, the B.A.S.S. is simple to load. The user only has to input the number of tires they have, and the plates are automatically brought to the appropriate position. There is no need to place tires into slots or reach in from the sides. To speed up the operation, the machine expends the bottom magazine first, followed by the top.

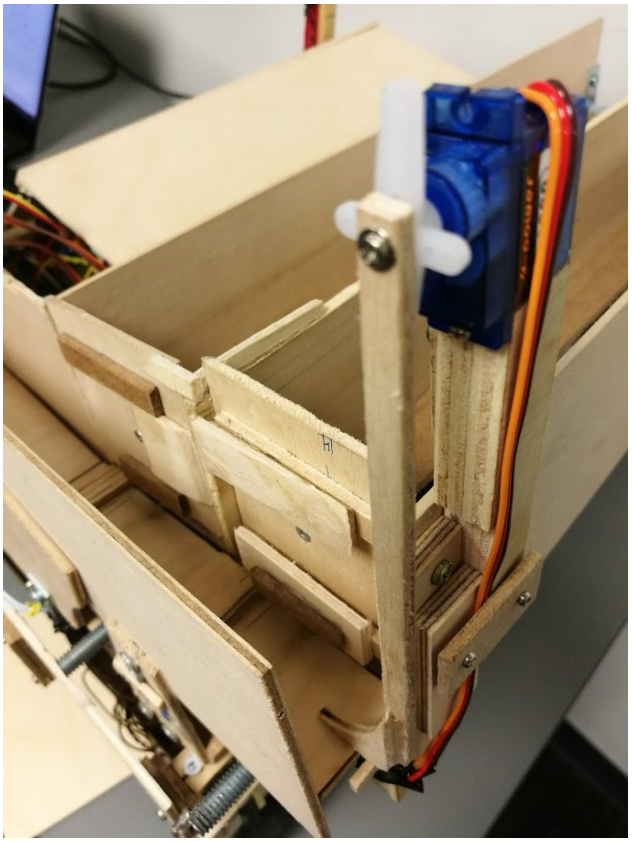


Figure 19: The pusher plates, kicker arm and ramp bumps. Note the slopes on the pusher plate tabs and the curved end of the kicker arm.

9.2.4 Extending Arm

To account for side-to-side deviations in pole placement, the B.A.S.S. is built around an extendable arm. This arm uses a rack and pinion system, and is powered by a small 28BYJ-48 stepper motor. The rack is attached to the underside of the arm, which slides laterally. The stepper and pinion are mounted to the front right support column of the robot.

A major feature of this design is its 'sleeved' walls. Since the arm is a ramp for tires, there must be barriers on both sides to prevent them from escaping. These barriers must be maintained no matter how far the arm is extended. To accomplish this, the extending arm is built to fit around a slightly smaller ramp of the same length. This inner ramp is fixed in place; at maximum extension, the outer walls of the arm continue the inner walls. A cross sectional diagram of this system can be seen in Figure 20. The arm slides on two wooden beams that cross the entirety of the robot. To stabilize the arm and prevent it from leaving its socket, a metal L-bracket grips these beams. The arm also supports several actuators and sensors. The associated cabling is routed through slots in the exterior plating. The cabling is carefully managed so that it retracts cleanly into the robot, while also providing enough

slack to fully extend. When fully extended, the arm is allowed to droop slightly, ensuring that tires can still roll down. A close up of the final arm can be seen in Figure 21.

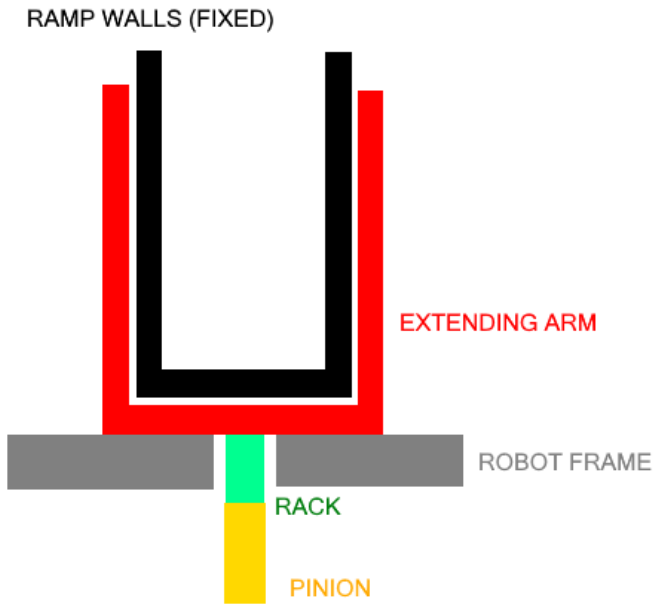


Figure 20: Cross Sectional Drawing of Arm Illustrating Various Layers.

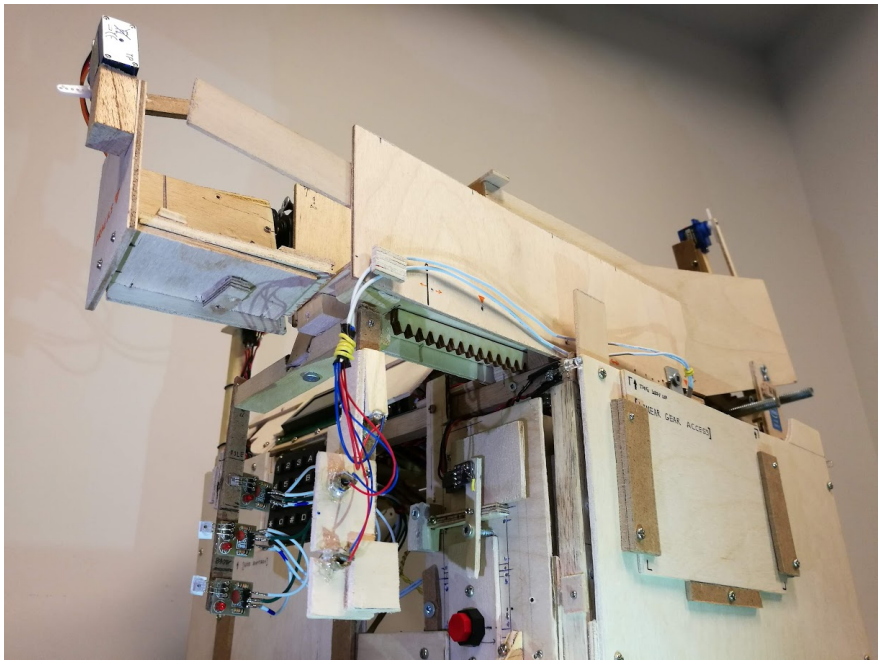


Figure 21: Underside view of the robot arm. Note the inner ramp above, and the larger outer arm walls. A portion of the rack gear is visible, along with the breakbeam array.

9.2.5 Sensor Mounts

Attached to the arm is one of the most important components of the machine: the sensor array. This is mounted beneath the arm, right before the rotating cell. This mounting location allows the B.A.S.S. to make the most of its arm reach. Taking the form of a Π, this structure contains three laser pointers and opposing receivers (breakbeam sensors). The bottom two breakbeams are slightly offset from the top. A single VL53L0X time of flight sensor, mounted on a fixed location on the robot frame, is used for initial pole detection and stopping the driving. Once the stop is complete, the arm-mounted breakbeam array sweeps out across the pole. The stop is timed so that the lasers and receivers should box in the pole. Once the top beam is broken (indicating a pole has been hit), the bottom two are used to scan for tires. Depending on how many bottom beams are broken, the number of tires can be determined.

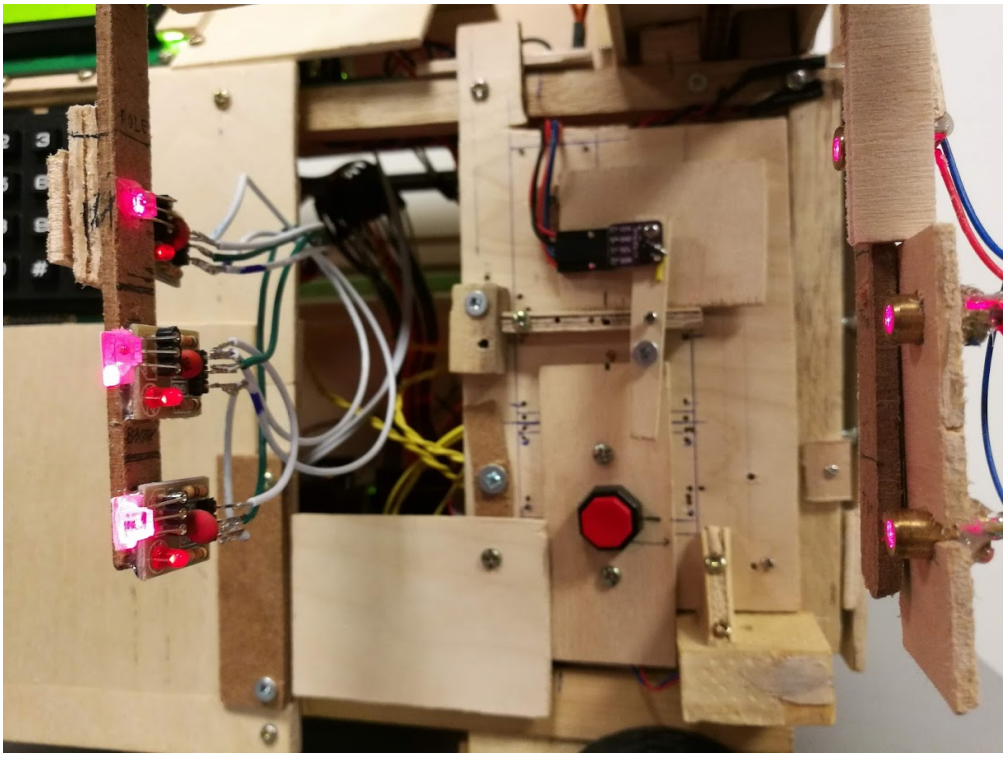


Figure 22: Triple breakbeams mounted underneath extending arm, with fixed VL53L0X TOF sensor in the background. Note the offset lateral positioning of the bottom two breakbeams versus the top.

The alignment of the lasers and receivers is crucial, so these were carefully mounted and adjusted in a dark room to ensure good coverage. MDF was used for the two sensor arms, as it is far stiffer than comparatively sized natural woods. During arm extension, there may be slight vibrations due to the friction between sliding wood panels; the sensor array must stay

rigid in order to get good results. Thicker woods were not used in order to limit the weight of the arm; the overall arm structure is rather light and is not designed to bear large loads (other than tires). The receivers were mounted in such a way that the pins faced inwards, in order to reduce the amount of cabling required. The sensor array can be seen in Figure 22.

The VL53L0X was mounted in a modular fashion. The sensor itself was attached to a thin wooden strip that allowed for ample clearance (the rear of the sensor contains various pins, capacitors and other protrusions). This strip was then attached with two screws to a horizontal bar with a variety of pre-drilled holes. During testing, this allowed for rapid changes in sensor position and mounting angle.

9.2.6 Rotating Cell

At the end of the extending arm is the central element of the B.A.S.S. deployment system, the rotating cell. The purpose of this cell is to receive tires in a rolling configuration, rotate them 90°, and deposit them onto the pole. The primary motivation behind this design was one of reliability and control: Extensive prototyping and testing of pure gravity-based systems (e.g. dropping directly from the ramp) showed clear issues. The difficulty in controlling tire movement (detailed in the Problem Assessment) meant that it was not possible to achieve a high drop success rate (see Figure 23). A more deterministic system was required, where actuators are used to govern tire movement rather than momentum. The cell satisfies this requirement, and is highly controlled. Assuming a correct stop position, the combination of breakbeam-assisted arm extension and the rotating cell ensure nearly perfect, consistent accuracy.

The floor of the rotating cell rests approximately 0.5 cm below the level of the extendable arm. This slight drop is to ensure that the tire cannot rebound off the cell's wall and roll back up the arm. Small wooden strips on the floor of the cell help to ensure that the tire settles in the correct position. Once the tire is inside the cell, a K-Power P0090 micro-servo mounted on the top rotates so that it secures the tire with its lever. This clamps the tire firmly enough that it cannot escape even if the cell moves. The entire cell will then rotate, so that the tire is oriented ready for dropping. This is via a Towerpro MG996R high-torque servo, mounted on the right wall of the arm. The micro-servo will then release its grip, allowing the tire to fall directly onto the pole. The cell will rotate back so that another tire can be reloaded or the machine can move on to the next pole. The drop sequence is fully illustrated in Figure 24.

The cell is akin to a box without its top, left, or rear walls. This absence is necessary, since the 'left' wall of the cell becomes the 'bottom' after rotation is complete. To prevent tires from falling out of the left side, a small fixed panel extends from the arm wall. Several

Figure 23: Early low-fidelity prototypes of gravity-drop systems did not show promising results. Tire hooks like above were common, as were total misses.

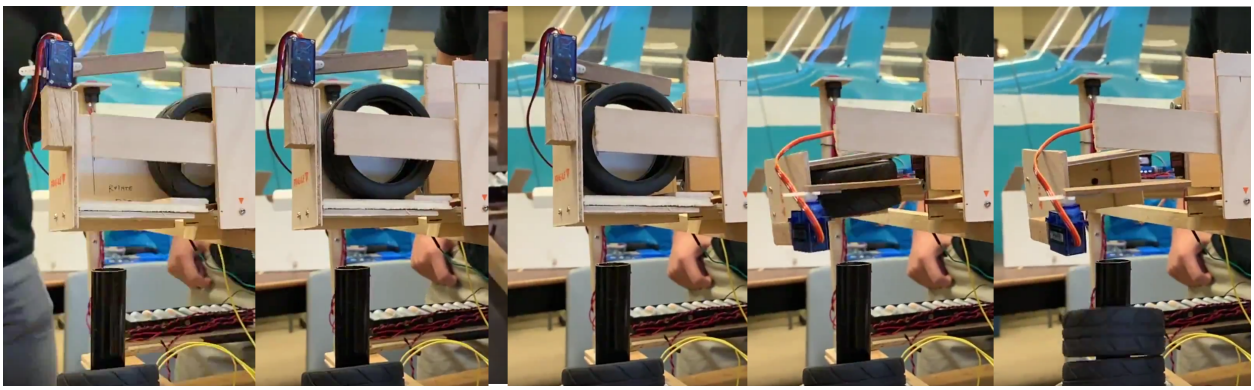


Figure 24: The drop sequence of the rotating cell. From L to R: 1. The tire rolls from the magazine into the cell. 2. The tire settles into the correct position. 3. The microservo clamps down. 4. The cell rotates. 5. The clamp releases and the tire drops onto the pole.

measures were taken to strengthen the cell and its servo connection. A double floor was built to stiffen the structure, and additional tabs were added on joints. The cell-servo connection had significant issues with slipping; accidental bumps or jams would cause the connection to fail and the cell would lose its correct angle. This was fixed by using a servo attachment with multiple perforations; by screwing the cell into this attachment at several points, a very strong connection was formed. The cell is unable to slip and lose its position, unless the

servo mount itself is destroyed. Protruding screws have been grounded down to minimize friction.

9.2.7 Miscellaneous Hardware

The previous sections have covered the major physical components of the robot. In addition to this, there are several miscellaneous features that do not fall under a specific section.

A laser aiming module located above the front-right tire of the robot assists with alignment and setup. It is calibrated to strike the ground exactly 4 metres from the front of the machine. By tracing a path to the laser impact point, the user can easily visualize where the robot will travel and position it for a run.

User-toggled navigation lights are mounted on the front of the robot, with red denoting the port side, and green indicating the starboard side. This was seen as a fitting inclusion, as AER201 is, after all, an aerospace course. The B.A.S.S. may have to navigate treacherous skies (and waters).

An auxiliary cable mounted in the rear allows users to connect their devices and play audio through the robot's internal speaker. When not in use, the cable is tucked away inside its own holding slot.

The μN plate on the front of the robot was installed as a tribute to this team's experience during the 2017 CIV102 Bridge Competition. Team 41's winning bridge, the *μN Club*, sustained a maximum load of 1505 N.

9.3 Supporting Calculations, Computer Programs, Simulation Results

Early in the project process, a CAD model of the machine was created in Fusion 360. This allowed for a relatively accurate simulation of materials and therefore mass. More importantly, the CAD model allowed for the visualization of moving components (fit and clearance). Joints were created and full animations were rigged. This was a key contributing factor to the speed at which the robot was built. Measurements for cuts were able to be taken straight from the CAD model, without the need to calculate and design during lab time. Figure 25 features various renders of the CAD model.

Many calculations were performed in order to verify actuator selection. One of the most critical calculations was the verification of the drive motors. The total mass of the robot was anticipated to be approximately 5.09 kg (the final value was measured to be 5.1 kg). This estimated value was determined through simulations in Fusion 360 CAD software. The robot was modelled in extensive detail and appropriate physical materials were assigned to all the

Figure 25: Fusion 360 CAD model of the robot, various views.

components. This is far more accurate than attempting to mentally estimate geometry or hand sketches. The radius of the wheels is 3 cm, and a typical value for the rolling resistance coefficient of the tires is 0.002 [27] (for bicycle tires on concrete). With these values, the required torque was calculated.

$$\tau = rF_{grip} \sin\theta \tag{2}$$

$$F_{normal} = m_{robot}g$$
$$F_{normal} = 5.09kg \times 9.81m/s$$
$$F_{normal} = 49.9N$$

$$F_{friction} = \mu_{rolling} \times F_{normal}$$
$$F_{friction} = 0.002 \times 49.9$$
$$F_{friction} = 0.0998N$$
$$\tau = r \times F_{friction}$$
$$\tau = 0.03 \times 0.0998$$
$$\tau = 0.003N \cdot m$$

The Shenzhen DC angled motor is rated for $0.035N \cdot m$; as such, it was determined to be sufficient for driving the robot.

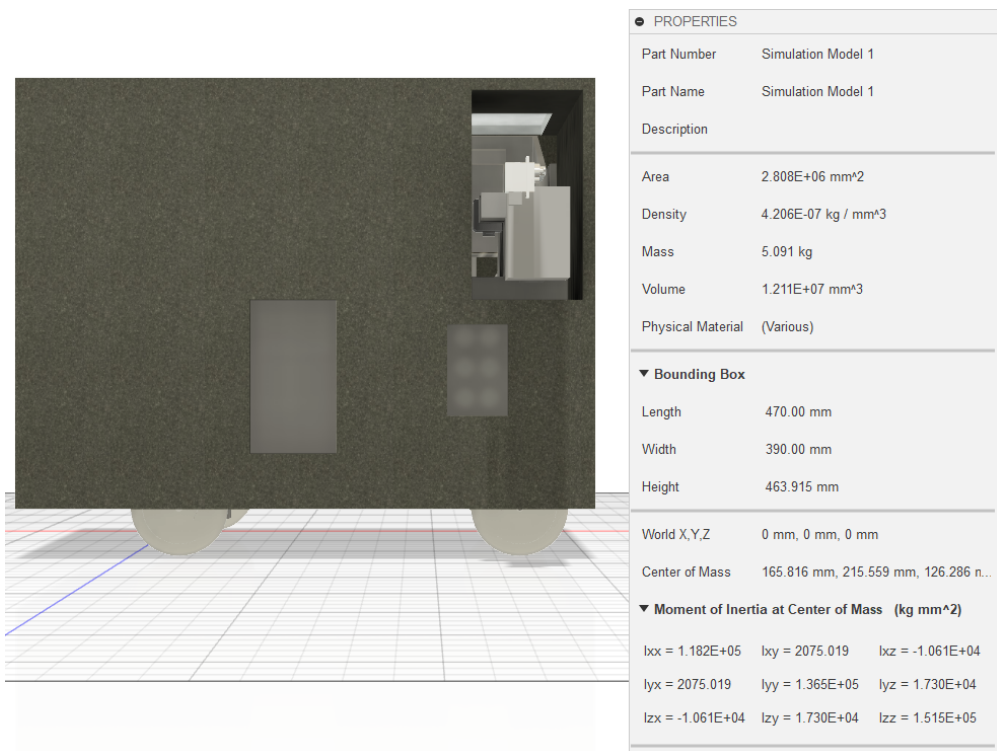


Figure 26: Calculation of robot properties using Fusion 360. Note: Shroud illustrated does not represent actual design, but only serves as a mass approximate.

Centre of mass calculations were also performed using Fusion 360. The calculated centre of mass is located at +16.6 cm in the length direction, +12.6 cm in the width direction, and +21.5 cm in the height direction. This is placed close to the centre of the driving base, ensuring good stability and presenting minimal risk of tipping. The COM, along with other properties can be seen in Figure 26.

Additional calculations were performed for the deployment mechanism. The micro-servo used for clamping the tire is the *K-Power P0090*. This servo provides 1.8 kg-cm (0.018 kg-m) of torque at 6 V. The tire mass is approximately 18 g, and the radius of the arm is 6 cm. The

coefficient of static friction of rubber is 0.95 [28]. In order to verify that the servo was strong enough to grip and hold the tire, calculations were performed using the torque equation:

$$\tau = rF_{grip} \sin\theta \tag{3}$$

$$F_{gravity} = m_{tire}g$$
$$F_{gravity} = 0.018kg \times 9.81m/s$$
$$F_{gravity} = 0.18N$$
$$F_{grip} = \mu_s\tau/r$$
$$F_{grip} = F_{gravity}$$
$$0.95\tau/0.06 = 0.18$$
$$\tau = 0.011N \cdot m$$
$$\tau = 0.0011kg \cdot m$$

Thus, the micro-servo was found to be more than capable of gripping the tire.

Calculations were also performed for verifying the cell rotation mechanism. The high-torque *MG996R* servo was selected for rotating this component. This servo is rated for $11kgf \cdot cm$ or $1.08N \cdot m$ of torque. The moment of inertia for the cell mechanism through the axis of rotation is $219.215kg \cdot mm^2$. This was determined by modelling and simulation in Fusion 360; the value includes a tire in the cell. We will take α to be 2 Hz/s (accelerating from a stop to 60 RPM in 0.5 sec).

$$\tau = I\alpha \tag{4}$$

$$\tau = 0.000219215 \times 2$$
$$\tau = 0.0004N \cdot m$$

Thus, the high torque servo was sufficient for rotating the cell, along with the clamped tire and servo. Good functionality of all actuators was demonstrated in the final physical machine. In particular, driving performance was more than sufficient; a 9 pole run was able to be completed comfortably under the 3 minute time limit.

9.4 Suggestions for Improvement of the Subsystem

If there was additional time left in the project schedule, and perhaps a greater budget, there are several changes that could have improved the performance of the B.A.S.S.:

1. **Improved Motors**: By the final phases of the project, stopping correctly was one of the greatest concerns. The deployment mechanism, arm extension, and sensors were

extremely promising in their performance. Unfortunately, the high stall torque and low control of the Shenzhen DC motors meant that it was not possible to drive the robot in extremely fine movements. This would have been very useful in aligning the robot more precisely; the current system relies on stopping properly in one go, which has no contingency plan in the event of failure. In addition to moving more precisely, better drive motors could have also accelerated the driving phases of the run.

2. **Sweeping Sensor Arm**: The current fixed mounting location of the breakbeam array reduces the effective range of the arm by several centimeters. Ideally, the breakbeams should have been mounted in-line with the centre of the cell in order to maximize reach. However, this would have introduced clearance issues, since even in a fully arm-retracted state the sensors would be dangling in the way of incoming poles. To fix this, the sensor array could have been made mobile, and attached to an additional microservo. This would swing the breakbeams down into place when required, and retract during any driving portions. This would offer the best of both worlds: range and clearance would both be maximized. Unfortunately, there was no time or budget left in the project to implement this. It was decided that the current system was satisfactory enough for the demonstration.

3. **Smoother Ramp Materials**: Friction was a constant nuisance during testing, and prior to the installation of a kicker servo tires would often become stuck in the ramp. It was realized that the ramp material (basswood) was rough and hairy, explaining the massive friction effects felt. Basswood was chosen for its extremely light weight, as the mass limit was an area of concern. Since the ramp and arm were heavily integrated into the structure of the robot, and replacing them would mean tearing everything apart, it was decided to install a kicker servo. This indeed fixed the issue, but for future iterations it would be wise to construct the ramp/arm from different materials. Plastic or MDF would be advisable choices.

4. **Leadscrews**: The twin magazines in the B.A.S.S. use conventional threaded rods. These are meant for construction and fastening of items; their thread pitch is approximately 2 mm, which is significantly on the low end. Leadscrews are designed for motion (often used in 3D printers) and feature a far larger thread pitch. For each revolution of the stepper, a leadscrew would drive the pusher plate much farther than a comparable threaded rod. Unfortunately, this comes at a significant cost. For the size of leadscrew required, it would have been impossible to factor one into our budget, not to mention the need to acquire two. Leadscrews also require custom nuts and couplings, and these are not affordable. A design decision was made early on to sacrifice pusher plate speed

and forgo the leadscrews. Magazine pushing ended up as the biggest bottleneck, time-wise, for the B.A.S.S. If possible, future iterations would benefit greatly from leadscrew use - almost no modifications need to be done, as they work the same as threaded rods. The budget is the only concern.

5. **Laser Cutting, 3D Printing, Metalworking**: Almost no advanced manufacturing was used in this project. Couplings, fasteners and other critical components were all handmade or salvaged from the parts bin. This was done out of a need for rapid prototyping; it was simply easier to hand-build and modify something rather than wait hours for printing or cutting. Countless changes were made on the fly, at a pace not suitable for advanced manufacturing techniques. However, now that the design is proven, it would be ideal to refine the plans and transfer the robot to a reproducible digital medium. This would, of course, improve aesthetic appeal and overall fit/finish. More importantly, the accuracy, reliability and precision of moving components would likely skyrocket. Hand-built components are quick and dirty solutions, and variations in construction can greatly impair functionality. Even if the machine works, it is impossible to determine how long it will hold up.

10 Microcontroller Subsystem

10.1 Assessment of the Problem

The microcontroller member of the team is responsible for writing, testing, and debugging programs that handled the robot's operation and user interface. Further responsibilities of the microcontroller member include selecting which devices to use, pin assignments, and communicating to the circuits and electromech members the capabilities and limitations of the software.

10.1.1 Microcontroller selection

The main microcontroller of choice for this project was the PIC18F4620, for the reasons listed below:

- Large number of I/O pins (33/35) [29]
- Large program memory (64kB) [30]
- Large SRAM data memory (4kB) [30]
- Includes EEPROM data memory (1kB) [30]
- Fast CPU clock speed (40MHz) [30]
- C Compiler Optimized Architecture [30]
- Low cost ($6.94 CAD/unit) [31]
- 100,000 Erase/Write Cycle Enhanced Flash Program Memory Typical [30]
- 1,000,000 Erase/Write Cycle Data EEPROM Memory Typical [30]

The use of the PIC DevBugger Rev 4.2 board allowed the PIC to communicate with the following devices:

- DS1307 Real Time Clock (RTC) [29]
- 4×16 Liquid Crystal Display (LCD) [29]
- 4×4 Matrix Keypad [29]
- Arduino Nano (via I^2C) [29]

- built-in digital and I^2C connections in the board

The Devbugger board also includes an external I^2C module [29], which allowed the PIC to communicate with an Arduino Uno.

10.1.2 User Interface

As specified by the RFP [6], the robot must include a user interface on an LCD screen that displays a termination message and the following information at the end of a run:

- Operation time
- Number of tires deployed on each pole
- Number of tires on each pole after operation
- Total number of supplied tires
- Number of poles
- Distance between each pole's centerline and the start line

Displaying all this information requires the use of multiple pages, and an intuitive menu system to navigate the various pages.

In addition to the features outlined by the RFP, the twin pushing magazines required a menu to control loading and pushing tires prior to operation. Lastly, the use of EEPROM memory logs also required menus to save and load data.

10.1.3 Drivetrain

The lack of landmarks in a straight line along the track means that the robot needs to be able to drive straight independent of its surroundings (the poles cannot be used as references since deviate horizontally). The drivetrain must be able to move the robot from one pole to the next and stop accurately in order for deployment to occur. At the end of the run, it must allow the robot to return behind the start line. Lastly, it must be able to record the distances between each pole's centerline and the start line.

10.1.4 Sensor Suite

The robot must be able to sense the poles accurately in order to stop correctly for deployment. After stopping, sensors must be used to determine how far the arm is to be extended, as well as count the number of tires on the pole before hand. Since there is a only a small margin

of error between the tires' inner circumference and the circumference of the pole, both the stop and arm extension must be very accurate and reliable. The control devices must be able to interpret and use sensor data effectively.

10.1.5 Deployment Mechanism

The robot needs to be able to complete its deployment sequence quickly and precisely. This includes powering all the actuators involved (servos, micro-servos, and stepper motors) correctly and in the right order. There must be ample delays between actuation to ensure synchronization and to account for physical actions (such as a tire rolling down the ramp). If any optimizations can be made, such as performing multiple processes simultaneously, they should be incorporated.

10.1.6 Device Communication

Due to the sheer number of actuators, sensors, and processes that are being used, our robot will require multiple control devices. As such, communication and synchronization of these devices is extremely important. The communication protocol used must be reliable and allow for a wide variety of processes to occur. The devices should also be able to communicate any necessary data to each other promptly.

10.2 Solution

The final design consisted of four microcontrollers (PIC, Arduino Nano, and two Arduino Unos (a "Control" Uno and a "Drive" Uno).

10.3 Main Program Logic

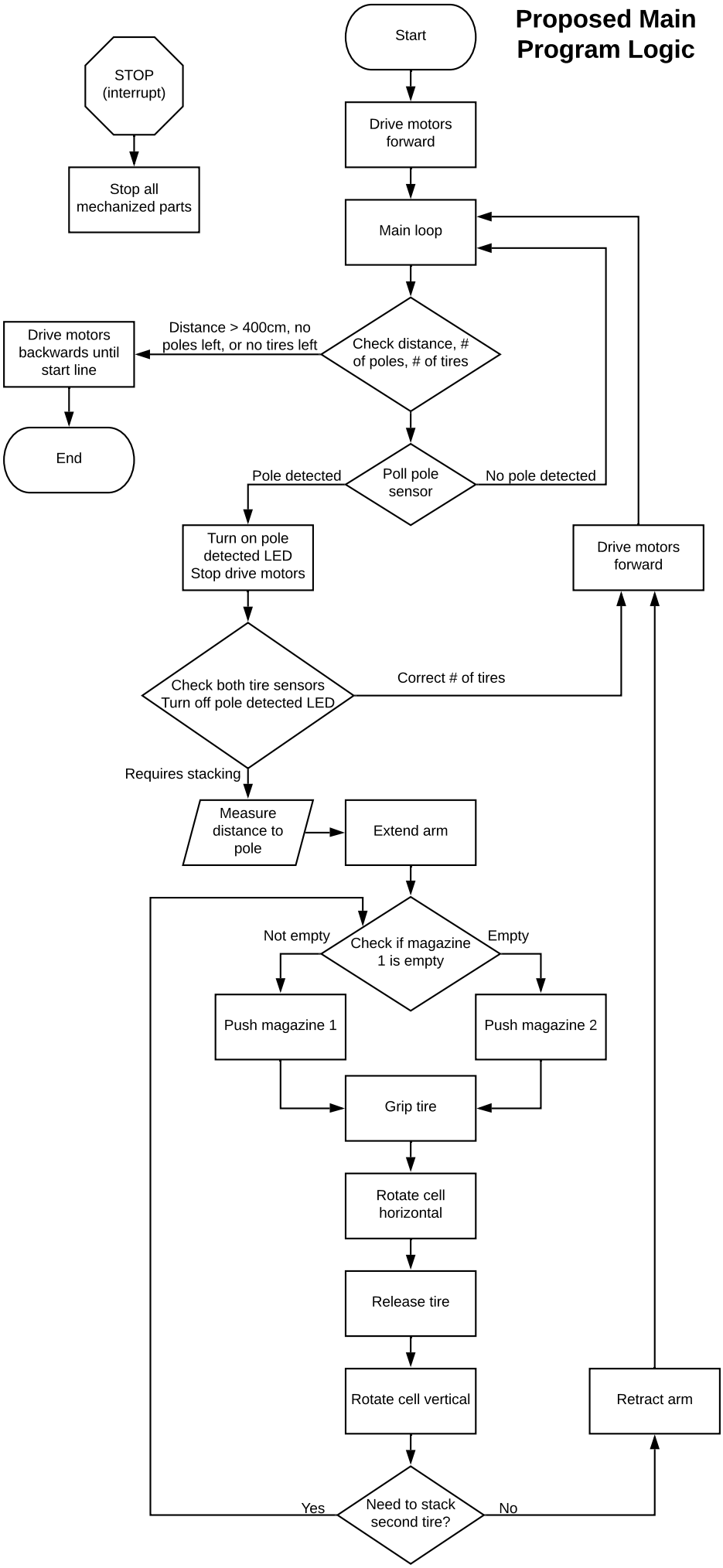
The main program logic has changed since the proposal. An exit condition based on operation time was added: if the run has lasted longer than 2:35, the robot will return to the start line. If the robot has not made it back to the start line by 2:58, the robot will cease moving and display the termination message, finishing the run. Another change is that due to the breakbeam sensor array, the arm must now be extended before tire sensing occurs. The operation of the kicker was also added to the main program logic.

The program will also now push a magazine when the robot is driving if there is no tire in the cell. When the robot detects a pole and stops, the newly pushed magazine will be kicked to the cell. This ensures that there is always a tire in the cell shortly after robot stops. Thus, if tire deployment is required, the robot can immediately operate its cell sequence

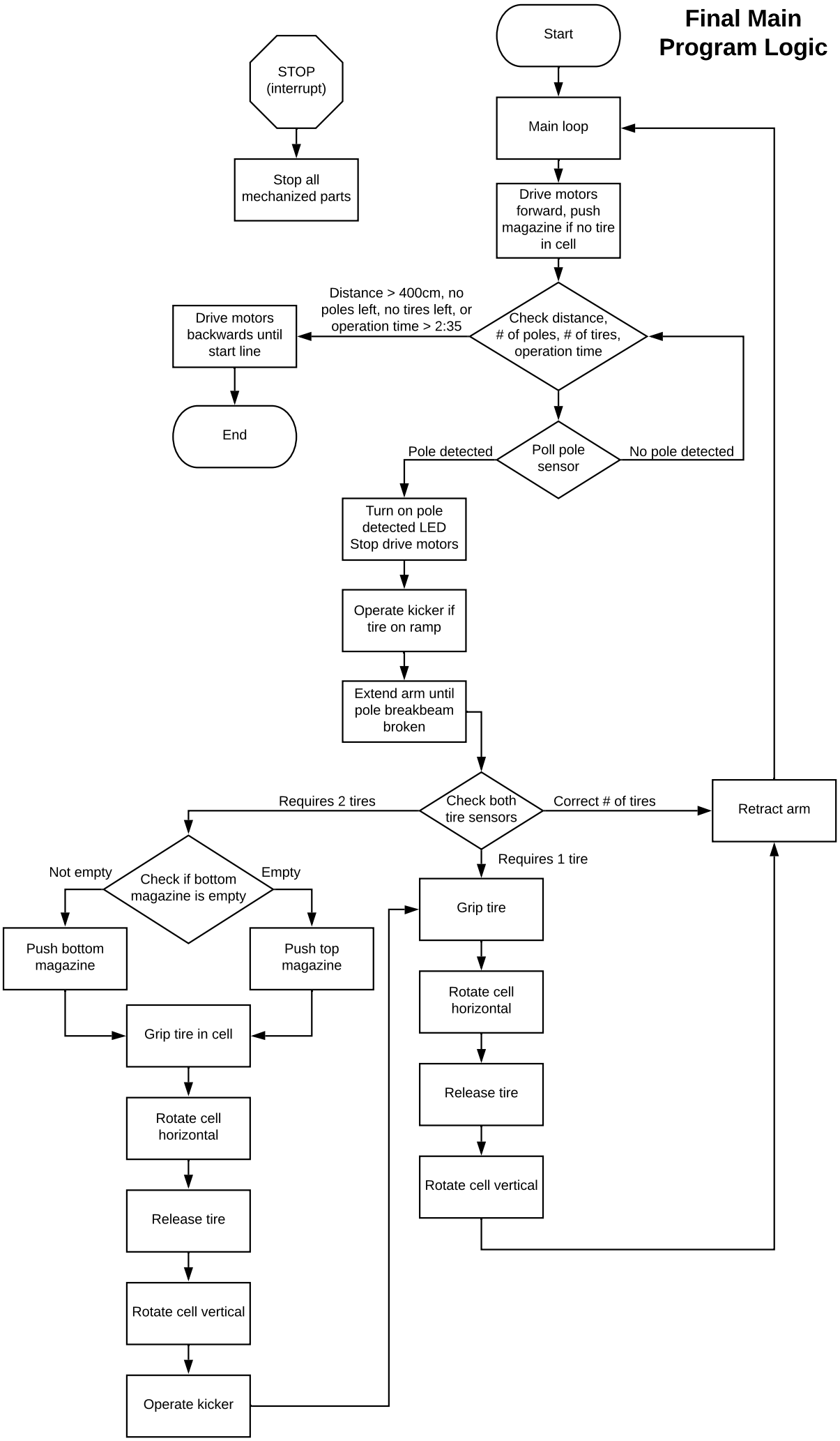
(grip, rotate, release, rotate back). If two tires are required, the robot will start pushing a magazine before the tire in the cell is deployed. The bottom magazine is emptied first: this is because the bottom push plate will become flush with the ramp wall when the magazine is empty, which creates a smooth wall for a top magazine tire to roll past. If a top magazine tire rolls down the ramp before the bottom magazine is emptied, the rolling tire may jam, either due to the friction between the two tires or due to hitting the lower wall of the bottom magazine.

The proposed and final versions of the main program logic can be found on the next two pages. In an actual operation, the operators would place a tire in the cell and a tire on the lower ramp stopper, which requires additional IF statements in the main program. However, this was left out of the flowchart as it would become too convoluted.

Proposed Main Program Logic



Final Main Program Logic



10.3.1 Device Communication

Due to budget constraints, threaded rods had to be used for our magazine pushers instead of lead screws. This meant that pushing tires from our magazines took a long time and thus were a significant bottleneck. To combat this, we used multiple controllers, each controlling different actuators. This allowed us to, for example, push the magazines while driving or while operating the cell. However, operating different devices at the same time required synchronization and well-mapped communication protocols.

The communication between the PIC and the Control Uno was handled using I^2C. I^2C was chosen due to the number of different commands the PIC needed to give the Control Uno, as well as the need for data to be sent back and forth between the two devices. The function below is used by the PIC to send a "command" to the Arduino over the I^2C bus:

```
void writeByteToControlArduino(unsigned char byteToWrite){
    __delay_us(100);                // Allow Control Uno to drive unoBusy HIGH
    while (PORTCbits.RC2){}         // Wait until Control Uno isn't busy
    I2C_Master_Start();             // Start condition
    I2C_Master_Write(0b00010000);   // 7 bit Arduino address + Write
    I2C_Master_Write(byteToWrite);  // Writes command to Control Uno
    I2C_Master_Stop();              // Stop condition
    return;
}
```

unoBusy is used to ensure that the PIC does not send the Control Uno a second command before the first command is finished executing. However, Arduino digital pins take 3 μs to write HIGH; the 100 μs delay ensures that another command will not be sent before unoBusy is written HIGH.

The code below is used by the Control Uno to receive a command from the PIC. receiveEvent() is an interrupt service handler that is called when the master (PIC) writes "Read" to the I^2C bus.

```
void setup() {
    Wire.begin(8); // Join I2C bus with address 8
    // Register callback functions
    Wire.onReceive(receiveEvent); // Called when this slave device receives a
   data transmission from master
    ...
}

void receiveEvent(void){
    digitalWrite(unoBusy, HIGH);  // Writes unoBusy high so the PIC will wait
   to send the next command
```

```
    newCommand = true;              // The Control Uno will only check what command was given if newCommand is true
    command = Wire.read();          // Receive byte
}

void loop(){
    ...
    // Only checks command if newCommand is true
    if (newCommand){
        // Command 1: New run
        if (command == 1){}
            ...
            isRunning = true;                // Operation started
            ...
        // Commands besides 1 are only checked if the robot is running
        } else if (isRunning){
            switch(command){
                // Command 2: Normal speed forward
                case 2:
                    ...
                // Command 4: Extend arm
                case 4:
                    ...
            }
        }
        newCommand = false;          // No command to process
        digitalWrite(unoBusy, LOW); // Control Uno is no longer busy
    }
}
```

As aforementioned, there was also a need to transmit data between the PIC and Control Uno. The following is the code used by the PIC to request the rotary encoder reading from the Control Uno:

```
// Read encoder data
I2C_Master_Start();              // Start condition
I2C_Master_Write(0b00010001);    // 7-bit Arduino slave address + Read
distTravelled = ((I2C_Master_Read(ACK) << 8) & 0xff00) | (I2C_Master_Read(NACK) & 0xff);    // Read distance travelled
I2C_Master_Stop();               // Stop condition
```

The Control Uno then sends the data using

```
void setup() {
    Wire.begin(8); // Join I2C bus with address 8
```

```
    // Register callback functions
    Wire.onRequest(requestEvent); // Called when master requests data from this slave device
    ...
}

void requestEvent(void){
    if (encoderPos == 1){
      Wire.write(0);
      Wire.write(0);
    } else {
      Wire.write(encoderPos>>8);
      Wire.write(encoderPos);
    }
}
```

where requestEvent() is an interrupt service handler that is called when the master (PIC) writes "Write" to the I^2C bus.

The distance is reported as 0 when encoderPos is 1 because the encoder counts 0, 1, 3, 5,... when only the data pin is used (instead of both the data and clock pins). This means that the first tick of the encoder is half the value of the other ticks. To combat this, encoderPos is initialized to 1, which means the encoder counts 1, 2, 4, 6,... If a distance of 0 is sent when the encoder is 1, then the encoder effectively counts 0, 2, 4, 6..., with equal values for each tick.

The remaining communications between the PIC, Nano, Control Uno, and Drive Uno are handled using digital pins. For example, the PIC tells the Nano to operate the kicker servo on the bottom magazine using

```
LATEbits.LATE2 = 1;      // Writes RE2 HIGH (connected to D10 on the Nano)
__delay_ms(10);          // Ensures the Nano reads the signal change
LATEbits.LATE2 = 0;      // Writes RE2 LOW
```

and the Control Uno tells the Drive Uno to start driving using:

```
// Tells the Drive Uno to start running
digitalWrite(startDrive, HIGH);      // Writes startDrive (D5) HIGH (connected to D11 on the Drive Uno)
delay(50);                           // Ensures the Drive Uno reads the signal change
digitalWrite(startDrive, LOW);       // Writes startDrive LOW
```

A longer delay was used here because the Drive Uno polls the gyroscope once in iteration of its main loop, which takes some time. Tables containing the pin assignments of all the

microcontrollers, including communication lines between devices, can be found in the next section.

10.3.2 Pin Assignments

As the length of the main PIC program grew, uploading the code to the PIC took an increasingly inconvenient amount of time (20 seconds). As a result, some actuators that required precise calibrating, such as the kicker servo, were moved to the Arduinos - it only took 5 seconds to program the Arduinos. That way, testing numerous servo positions that only differed by a few degrees would not take excessive amounts of time.

For the pin assignments of all the devices, see Tables 5, 6, 7, and 8.

Table 5: PIC Pin Assignments

Pin	Connected to	Description
GND		
KPD	RC1	Keypad disable
RC6		
RC4	Control Uno A4	I^2C data bus
RC2	Control Uno D11	Driven high if Control Uno busy
RC0	Control Uno D4	Control Uno tells PIC run is finished
RE2	Nano D10	PIC tells Nano to operate kicker on bottom magazine
RE1		Tire 2 breakbeam receiver
RA7		External oscillator
RA5	Nano A3	Nano tells PIC the motors have stopped
RA3	Control Uno D12	Tells Control Uno to stop extending arm
RA1		
RD7		LCD
RD5		LCD
RD3		LCD
RD1	Nano D3	PIC tells Nano to operate kicker on top magazine
RB7		Bottom magazine stepper motor
RB5		Bottom magazine stepper motor
RB3		Bottom magazine stepper motor
RB1		Bottom magazine stepper motor
GND		
RC7		
RC5	Nano D11	PIC tells Nano run is finished
RC3	Control Uno A5	I^2C clock bus
RC1	KPD	Controls keypad disable
GND		
VCC		
RE0		Tire 1 breakbeam receiver
RA6		External oscillator
RA4	Nano A2	PIC tells Nano to start polling laser
RA2		Pole breakbeam receiver
RA0	Nano A0	PIC tells Nano to operate cell and gripper
RD6		LCD
RD4		LCD
RD2		LCD
RD0		Broken pin
RB6		Top magazine stepper motor
RB4		Top magazine stepper motor
RB2		Top magazine stepper motor
RB0		Top magazine stepper motor

Table 6: Control Uno Pin Assignments

Pin	Connected to	Description
A5	PIC RC3	I^2C clock connection to PIC (master)
A4	PIC RC4	I^2C data connection to PIC (master)
A3		Arm stepper motor
A2		Arm stepper motor
A1		Arm stepper motor
A0		Arm stepper motor
D13		
D12	PIC RA3	PIC tells Control Uno to stop extending arm
D11	PIC RC2	Control Uno tells PIC it is busy with an operation
D10		
D9		
D8		
D7	Drive Uno D4	Control Uno tells Drive Uno to start run
D6	Drive Uno D2	Control Uno tells Drive Uno to return to start
D5	Drive Uno D11	Control Uno tells Drive Uno to start driving
D4	PIC RC5	Control Uno tells PIC run is finished
D3		
D2		Rotary encoder data pin
D1		
D0		

Table 7: Nano Pin Assignments

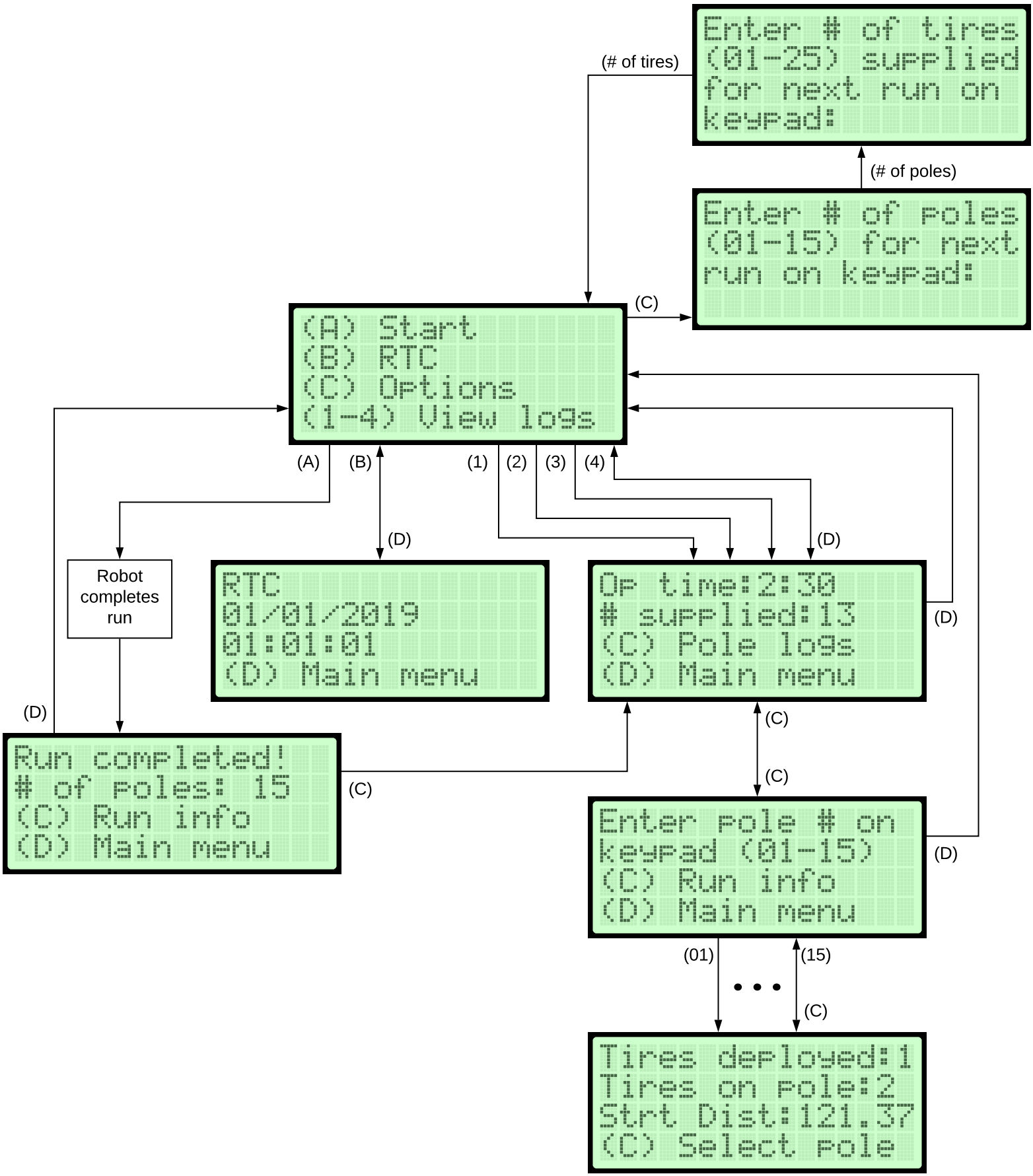
Pin	Connected to	Description
A5	Laser sensor	I^2C clock connection to laser sensor
A4	Laser sensor	I^2C data connection to laser sensor
A3	PIC RA5, Drive Uno D3	Nano tells PIC and Drive Uno to stop motors
A2	PIC RA4	PIC tells Nano to start polling laser
A1		
A0	PIC RA0	PIC tells Nano to start servos
D13		Gripper servo
D12		Cell servo
D11	PIC RC5	PIC tells Nano run is finished
D10	PIC RE2	PIC tells Nano to operate kicker on bottom magazine
D9		Kicker servo
D8		
D7		
D6		
D5		
D4		
D3	PIC RD1	PIC tells Nano to operate kicker on top magazine
D2	PIC RD0	PIC pin broken
D1		
D0		

Table 8: Drive Uno Pin Assignments

Pin	Connected to	Description
A5		I2C clock connection to gyroscope
A4		I2C data connection to gyroscope
A3		
A2		
A1		Speaker signal
A0		Turns speaker on
D13		Pole detected LED
D12		
D11	Control Uno D5	Control Uno tells Drive Uno to start driving
D10		Right DC drive motor
D9		Right DC drive motor
D8		Left DC drive motor
D7		Left DC drive motor
D6		Right DC motor PWM signal
D5		Left DC motor PWM signal
D4	Control Uno D4	Control Uno tells Drive Uno to start run
D3	Nano A3	Nano tells Drive Uno to stop motors
D2	Control Uno D6	Control Uno tells Drive Uno to return to start
D1		
D0		

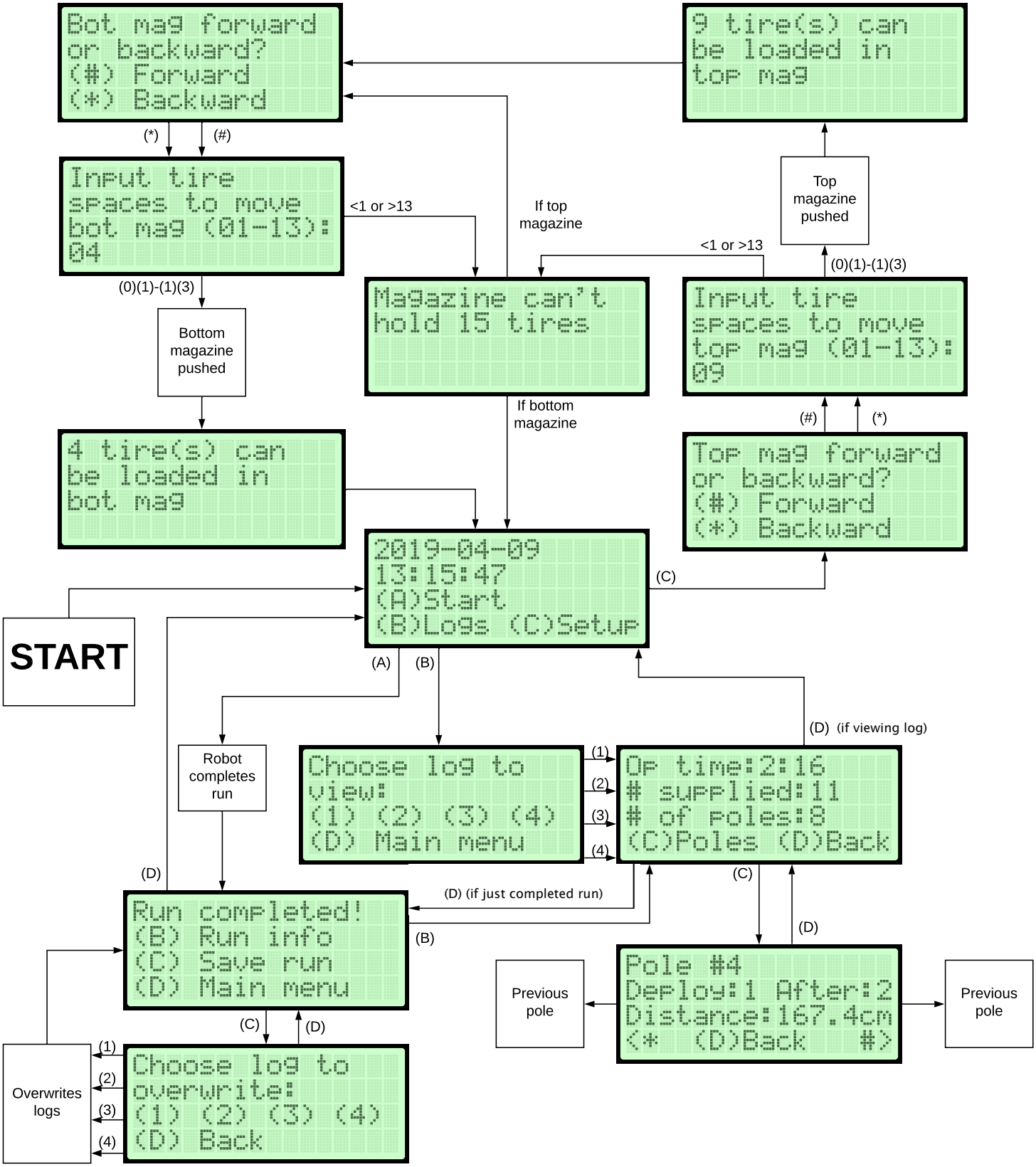
10.3.3 User Interface

The following page contains the proposed Keypad-LCD User Interface.


Enter # of tires (01-25) supplied for next run on keypad:
(# of tires)
(# of poles)
Enter # of poles (01-15) for next run on keypad:
(C)
(A) Start
(B) RTC
(C) Options
(1-4) View logs
(A)
(B)
(1)
(2)
(3)
(4)
(D)
(D)
Robot completes run
RTC
01/01/2019
01:01:01
(D) Main menu
Op time:2:30
supplied:13
(C) Pole logs
(D) Main menu
(D)
(D)
Run completed!
# of poles: 15
(C) Run info
(D) Main menu
(C)
(C)
(C)
Enter pole # on keypad (01-15)
(C) Run info
(D) Main menu
(D)
(01)
(15)
(C)
Tires deployed:1
Tires on pole:2
Strt Dist:121.37
(C) Select pole

The Keypad-LCD User Interface (UI) has undergone some notable changes since the proposed version. The final version can be seen on the following page. The RTC was moved to the Main Menu (standby mode) in accordance with the RFP [6]. The option menu for entering the number of poles on the next run was removed since the number of poles would not be known beforehand. Instead of entering the number of tires supplied, separate pages were created for both magazines and more extensive loading options were provided to improve flexibility (which turned out to be extremely useful and convenient during testing). Lastly, instead of entering pole numbers to view information about poles, a scrolling menu was implemented, which allows information to be viewed faster and more easily.

The following page contains the final Keypad-LCD UI. START is the initial state of the program when the robot is turned on


Bot mag forward
or backward?
(#) Forward
(*) Backward
9 tire(s) can
be loaded in
top mag
(*)
(#)
Input tire
spaces to move
bot mag (01-13):
04
<1 or >13
If top
magazine
Top
magazine
pushed
<1 or >13
(0)(1)-(1)(3)
(0)(1)-(1)(3)
Magazine can't
hold 15 tires
Input tire
spaces to move
top mag (01-13):
09
Bottom
magazine
pushed
If bottom
magazine
(#)
(*)
4 tire(s) can
be loaded in
bot mag
Top mag forward
or backward?
(#) Forward
(*) Backward
2019-04-09
13:15:47
(A)Start
(B)Logs (C)Setup
(C)
START
(A)
(B)
(D) (if viewing log)
Robot
completes
run
Choose log to
view:
(1) (2) (3) (4)
(D) Main menu
(1)
(2)
(3)
(4)
Op time:2:16
supplied:11
of poles:8
(C)Poles (D)Back
(D)
(D) (if just completed run)
(C)
Run completed!
(B) Run info
(C) Save run
(D) Main menu
(B)
(D)
Previous
pole
Pole #4
Deploy:1 After:2
Distance:167.4cm
<* (D)Back #>
Previous
pole
(C)
(D)
Overwrites
logs
(1)
(2)
(3)
(4)
Choose log to
overwrite:
(1) (2) (3) (4)
(D) Back

This UI was created using a changing menu variable. Each page has a corresponding menu value: if the menu variable is equal to that value, then that page is displayed. The PIC code for this is shown below:

```
while(1){
    switch (menu){
        // Main menu
        case 0:
            mainMenu(&menu);
            break;
        // Start operation function
        case 1:
            start(&menu, &currentRun, &topMagTires, &bottomMagTires);
            break;
        // Run finished function
        case 2:
            runFinished(&menu, &currentRun, &polePage, &displayRun, &logs);
            break;
        // Run info menu
        case 3:
            runInfoMenu(&menu, &displayRun, logs);
            break;
        // Pole info page
        // polePage determines which pole's info is shown
        case 4:
            poleInfo(&menu, &displayRun, &polePage);
            break;
        // Choose which EEPROM log to overwrite with new data
        case 5:
            saveRun(&menu, &currentRun);
            break;
        // Choose which EEPROM log to view
        case 6:
            selectLog(&menu, &displayRun);
            break;
        // Termination message/run info menu
        case 7:
            runFinishedMenu(&menu, &logs);
            break;
        // Load tires into magazines
        case 8:
            options(&menu, &topMagTires, &bottomMagTires);
            break;
    }
```

```
}
```

To move between pages, each menu has keypad prompts. When a recognized key is pressed, the menu variable is changed, the program returns to the main loop, and then changes to another page. The following is the keypad prompting in the run finished menu (termination message):

```
// Wait until a key is pressed
while (!PORTBbits.RB1){}
// Bit-shift the data to determine which key was pressed
keypress = (PORTB & 0xF0) >> 4;
// If the key is recognized, set the menu variable
switch (keypress){
    case 0b0111:    // (B)
        *menu = 3;  // Run info menu
        break;
    case 0b1011:    // (C)
        *menu = 5;  // Save run menu
        break;
    case 0b01111:   // (D)
        *logs = true;
        *menu = 0;  // Main menu
        break;
}
// Wait until the key is let go
while (PORTBbits.RB1){}
```

10.3.4 Drivetrain

There were three methods considered to ensure the robot drove straight:

1. Using preset PWM values to account for the differences in the DC motors
2. Using a Proportional Integral Derivative (PID) function to adjust the DC PWM values based on the distance travelled
3. Using a gyroscope to adjust the DC PWM values based on the robot's heading

The preset PWM values were implemented by writing a higher value to the slower motor - one motor drove slower than the other when they were both given the same voltage (likely due to variability in manufacturing). This was by far the simplest and easiest to implement option of the three, but it provided no resilience to drifting off-course. If the robot strayed

from a straight path, there was nothing to turn it back, so it would continue driving off-course. Eventually it would either travel too far from the poles for it to deploy, or worse, run into the poles, which would disqualify it.

The PID option would offer some correction if the robot drifted off-course, but there were downsides to using it. First, it would rely on the accuracy of two encoders. The encoders sometimes skip ticks, so the readings may not be accurate. If the wheels slip, it would also result in false readings. Furthermore, to utilize the data and clock pins of both encoders, one would require four interrupt pins on a microcontroller. However, the Arduinos only possess two interrupts pins. This means that only one pin from each encoder could be used. This is perfectly viable, but it would be impossible to tell if the wheels are moving forward or backward (both would increase the encoder count). If the DC motors do not stop perfectly, for example, if they slip or roll after stopping, this would increase the encoder count and throw off the algorithm. Lastly, PID requires extensive testing in order to set parameter values. If these parameters are not fine-tuned, the robot may not correct its course enough, or it may jerk side to side in an oscillatory motion.

Using a gyroscope would offer course correction without relying on encoder readings. The gyroscope was very accurate, but required 20 seconds to stabilize after turning the robot on. After starting an operation, the yaw (heading) of the robot at the start line would be recorded as a reference yaw. The Drive Uno would constantly poll the gyroscope for an updated yaw using the following code:

```
void loop(){
    // Reset the FIFO buffer
    mpu.resetFIFO();

    // get current FIFO count
    fifoCount = mpu.getFIFOCount();

    // Wait for correct available data length, should be a VERY short wait
    while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

    // Read a packet from FIFO
    mpu.getFIFOBytes(fifoBuffer, packetSize);

    // Get Euler angles in degrees
    mpu.dmpGetQuaternion(&q, fifoBuffer);
    mpu.dmpGetGravity(&gravity, &q);
    mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
}
```

The gyroscope constantly stores data in a FIFO buffer, which is read and converted

to readings in degrees. The current yaw value is stored in ypr[0]. The reference yaw is subtracted from the current yaw, and the function calc(), written by Devansh Ranade, is used to determine a correction factor for the motors:

```
float calc(float d){
  unsigned char amp = 8;
  unsigned char var = 2;
  unsigned char maxVar = 10;
  if (d < (-360.0+maxVar)){
    if (d < (-360.0+var)){
      return amp*sqrt((d+360)/var);
    }
    return amp;
  } else if (d > 360.0-maxVar){
    if (d > (360.0-var)){
      return -amp*sqrt(-(d-360.0)/var);
    }
    return -amp;
  } else  if (d > -maxVar&& d < maxVar) {
    if (d > -var && d < var){
      if (d > 0){
        return -amp*sqrt(d/var);
      } else if (d < 0){
        return amp*sqrt(-d/var);
      }
    }
    if (d > var){
      return -amp;
    } else if (d < -var){
      return amp;
    }
  } else {
    return 0;
  }
  return 0;
}
```

This function takes the differences between the yaw readings and outputs a correction factor proportional to the square root of the difference. amp is the maximum possible output factor, and lower values of val make the correction factor more aggressive. A square root function was chosen because small deviations will be scaled up, which will quickly correct the robot's course even if the heading only change by a little. The calculated factor is then added to the left motor and subtracted from the right motor. The gyroscope option was

chosen for our robot and was very successful in ensuring the robot drove along a straight path.

10.3.5 Sensor Suite

Figure 27: The HC-SR04 Ultrasonic Sensor [32].

Based on preliminary testing (Figure 28), HC-SR04 ultrasonic sensors (Figure 27) seemed to be the best option for both sensing poles and tires. When a pole or tire was placed in front of an ultrasonic sensor, there would be a notable dip in the sensor's measurements, which was ideal (Figure 29). The code to run ultrasonic sensors on an Arduino is shown below:

```
// Clears the trigPin
digitalWrite(trigPin, LOW);
delayMicroseconds(2);
// Sets the trigPin on HIGH state for 10 micro seconds
digitalWrite(trigPin, HIGH);
delayMicroseconds(10);
digitalWrite(trigPin, LOW);
// Reads the echoPin, returns the sound wave travel time in microseconds
duration = pulseIn(echoPin, HIGH);
// Calculating the distance
distance= duration*0.034/2;
// Prints the distance on the Serial Monitor
Serial.println(distance);
```

However, after running stopping tests, it was discovered that the wide cone of the ultrasonic sound wave resulted in inconsistent stopping, from far too early to late stops. Alternative options for stopping would have to be explored.

Black surfaces absorb much of infrared (IR) light, so IR sensors were essentially written off from the beginning. Nevertheless, a Sharp GP2Y0A51SK0F 2-15 cm Analog Distance Sensor (Figure 30) was bought and tested. Sharp does not provide a function to convert

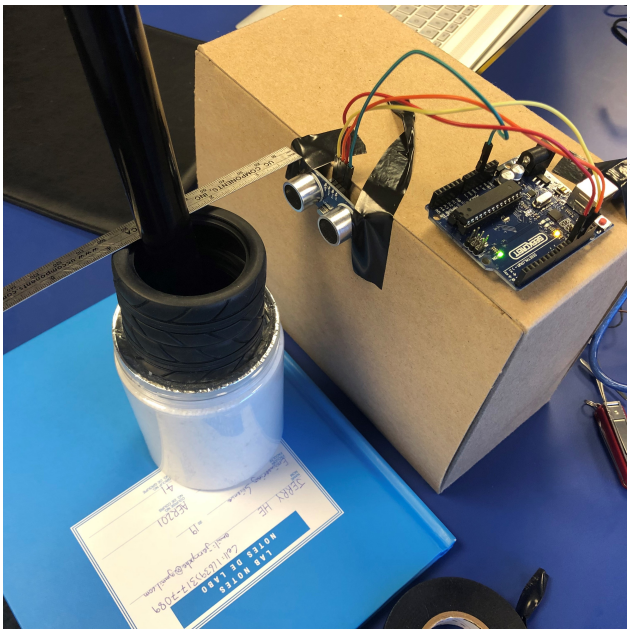


Figure 28: Ultrasonic sensor testing setup.

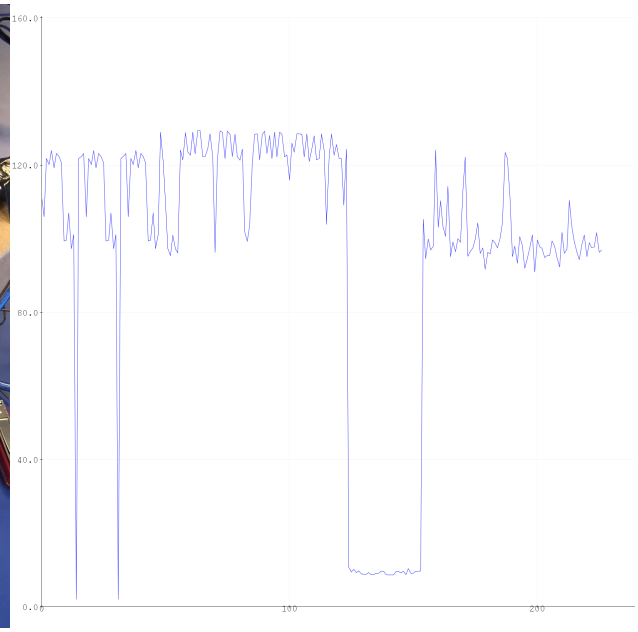


Figure 29: Ultrasonic sensor testing graph.

Figure 30: The Sharp GP2Y0A51SK0F 2-15 cm Analog Distance Sensor [33].

sensor readings into distances, only a graph of analog output vs. distance (Figure 31), so one had to be created from test data (Figure 32).

The function that converts analog outputs to distances, i.e. the equation of the best fit line, is

$$distance = 5.3196 * \frac{1}{analog\,output - 0.04} + 0.0746$$

However, even when sensing non-black objects, the IR sensor seemed to be more inaccurate than ultrasonics or VL53L0X laser sensors. The IR sensor readings would constantly change, even when sensing a non-moving object. Because of its poor performance in tests, the IR sensor was dismissed as an option.

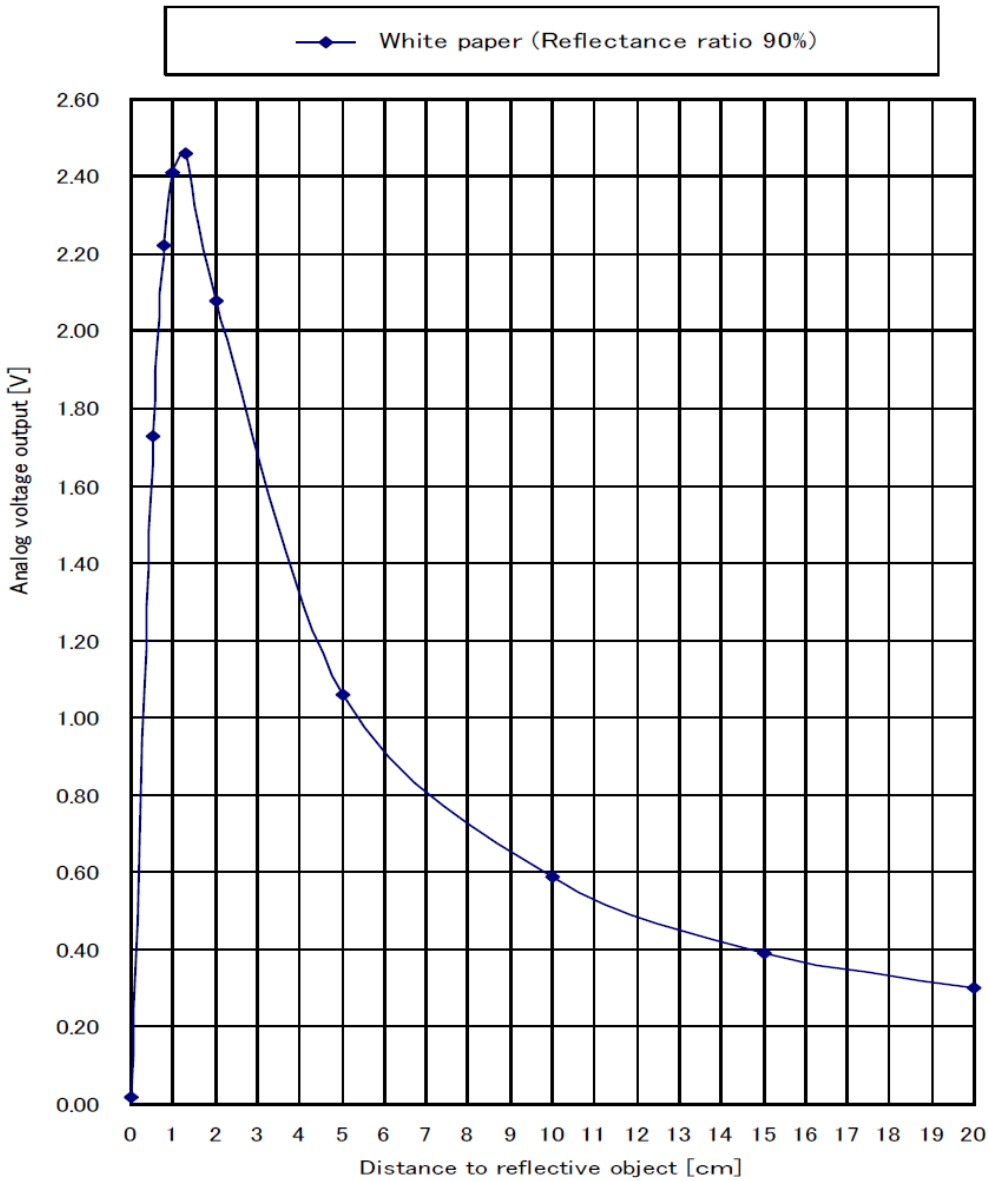


Figure 31: Graph of analog output vs. distance for the Sharp GP2Y0A51SK0F IR Sensor [34].

The aforementioned VL53L0X Time of Flight (ToF) laser sensor (Figure 33) was considered next. The laser sensor is controlled using I^2C, so the Nano had to be used as a master (this project had a very tight time frame and there was little support for the sensor on the PIC; polling the sensor on the Nano also allowed us to upload test code quicker). Preliminary testing seemed to show that the laser sensor stopped the robot more accurately than ultrasonics. However, it seemed to stop sooner at further ranges (due to the cone of the photons), which would cause the cell to drop tires too far behind the pole. To correct for this, a slight delay that scaled based on distance was added before the motors braked.

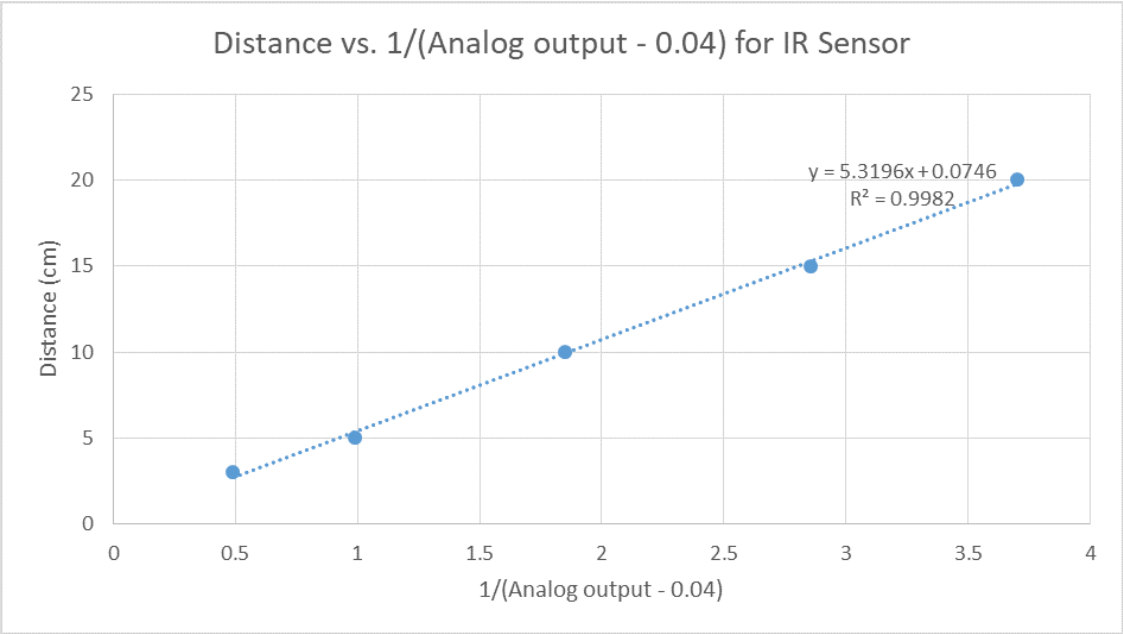


Figure 32: Graph of $\frac{1}{analog\ output - 0.04}$ vs. distance obtained from testing the IR Sensor.

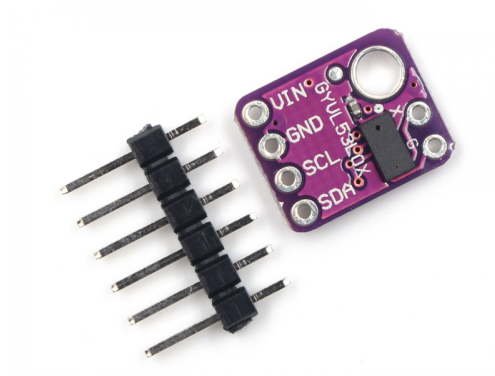


Figure 33: The VL53L0X Time of Flight (ToF) Laser Sensor [35].

The function for this delay, shown below, was again determined by testing.

$$delay = 100 * \frac{laser\ reading - 468}{640 - 468}$$

This function outputs a 0ms delay when the laser reading is 468 and a 100ms delay when the reading is 640. This delay seemed to work well and many successful drops were achieved using it. However, after a sensor panel refit, the laser sensor began stopping the robot correctly at all distances with a constant delay, so this function was removed and replaced with a fixed delay. Sometimes the laser sensor would give false readings and stop the robot when no poles were beside it, so a "double-check" system was used to stop the robot: if the laser sensor read a distance less than 350, it would be polled again, and the robot would only be stopped if the second reading was also less than 350. The code for this is shown below:

```
// Check if laser sensor reads less than 350
if (poleDist <= 350){
    poleSensor.rangingTest(&poleMeasure, false);
    poleDist = poleMeasure.RangeMilliMeter;
    // If the reading is less than 350, poll the sensor again and only stop
   the robot if both readings are less than 350
    if (poleDist <= 350){
        delay(13);
        digitalWrite(stopMotors, HIGH);
    }
}
```

A variety of options were considered in order to determine how far to extend the arm and count the number of tires on the pole. Because of the laser sensor's success in stopping the robot, it was the first candidate to measure distances to the pole and tires. However, it was found that measuring different areas of the pole resulted in readings with variations up to 10 "cm". This error may be due to the significantly shallower angle of attack of the photon beam when shined at the sides of the pole instead of the front. In any case, this inconsistency meant that the arm could not be extended based on laser sensor readings.

The next option was triple ultrasonic sensors (Figure 34), the sensor arrangement presented in our proposal. The top/pole sensor actually proved to be very successful in measuring the distance to the pole, and combined with the laser sensor stopping, the robot was able to successfully deploy numerous tires. Tests were carried out, and a function was determined convert the ultrasonic sensor's reading into the number of steps the arm stepper motor needed to rotate (Figure 35):

$$steps = 107.65 * distance - 684.64$$

When the other two sensors were mounted underneath and used to detect tires, the readings they gave appeared to decrease when tires were placed on the pole (Figure 36).

However, the sensors were only consistent at very close ranges and with the pole in an optimal position. As the pole was moved around, the sensor values became less accurate. Even worse, at extreme distances, the cones of the sensors were so wide that they would detect each other's objects. For example, the pole sensors would start detecting tire 2 and report lower values. Because of this, the ultrasonic sensor array was discarded.

The third candidate was a laser sensor mounted on the cell, pointing downwards at the pole. The sensor would poll as the arm extended: when the sensor passed over tires, it would report a lower reading. Thresholds can then be determined to see if the readings signified one, two, or no tires. Again, early results seemed positive: the laser showed terraced values, clearly differentiating the number of tires (Figure 37).

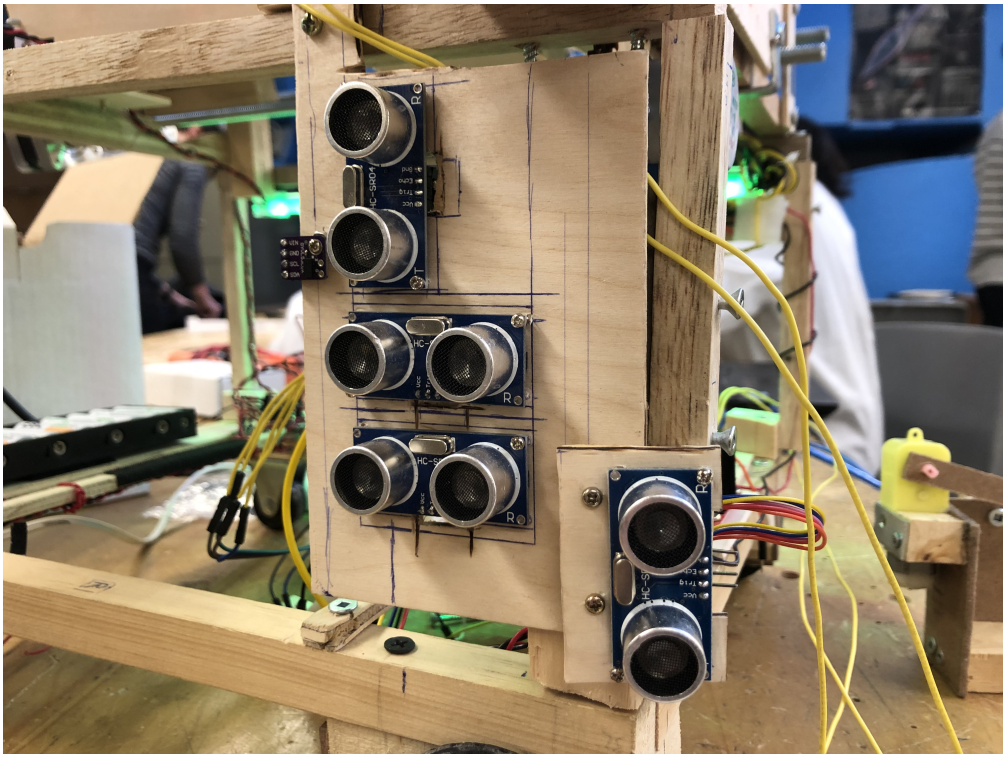


Figure 34: The triple ultrasonic sensor array. Note the bottom-right ultrasonic sensor. This sensor was originally designed to be used as an "early-warning" system: if the sensor detected a pole, the robot would slow to an "approach speed" in order to improve the chances of a successful stop. However, it was determined that the distances between the poles were not large enough to make use of this system, so the sensor was removed.

But, once again, these tests were done under ideal conditions. When the robot stopped by itself and attempted to count tires, the results were inconsistent.

The last option was a sweeping array of laser breakbeams (Figure 22). The arm would extend until the top/pole beam broke. It would then count the number of tires using the bottom two beams. For example, if the bottom beam was broken after the arm finished extending, then there must be one tire on the pole. Because of the wide range of this array, it still functioned correctly even if the stop was not perfect. The pole detection and tire counting were nearly instant and extremely accurate. During the public demonstration, the breakbeams only produced one error over two runs, and the error was likely due to a slanted pole (the pole beam broke "early" since the pole was tilted towards the robot, which caused the bottom beam to detect the bottom of the pole).

10.3.6 Deployment Mechanism

To push the tires from the top magazine, the following function was used:

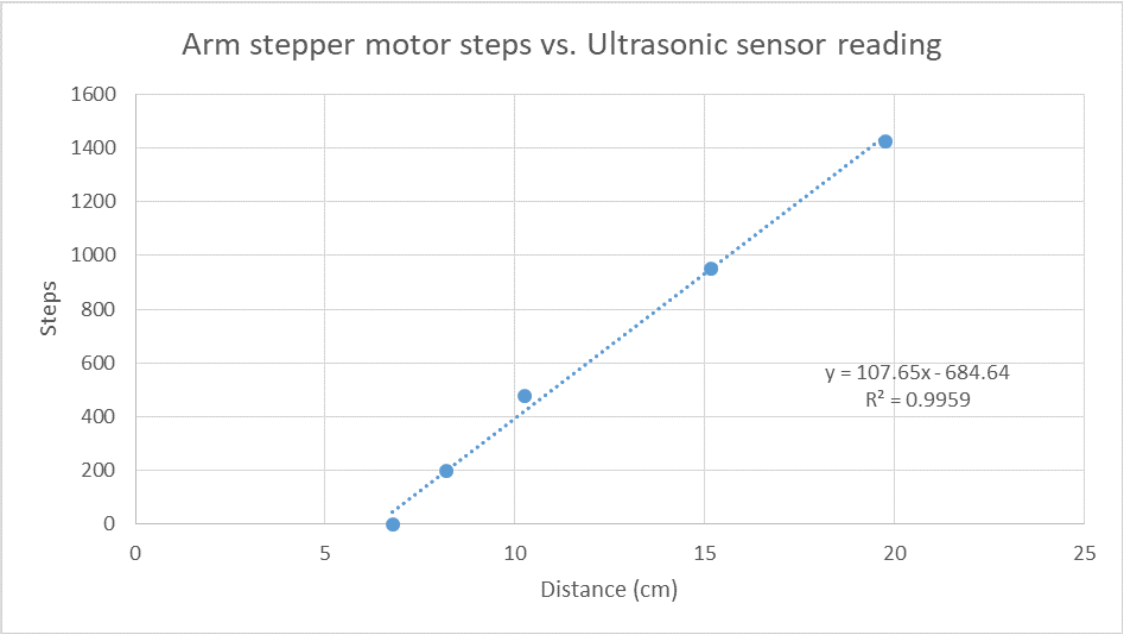


Figure 35: Graph of Arm stepper motor steps vs. Distance obtained from testing the ultrasonic sensor.

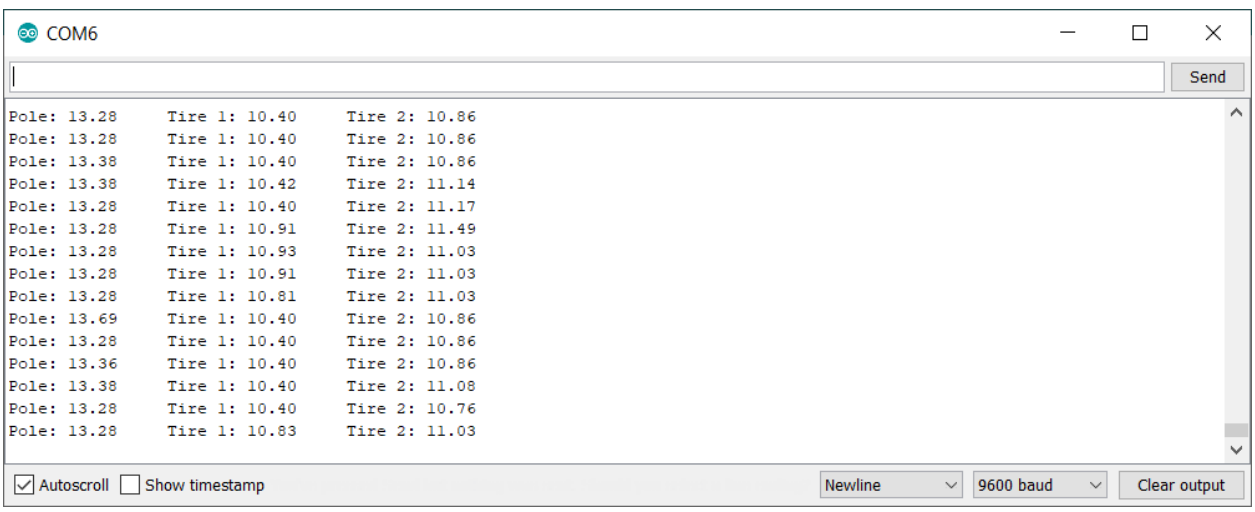


Figure 36: The values read when testing the triple ultrasonic array. Note that both the tire 1 and tire 2 values are significantly lower than the pole value.

```
// steps is the number of steps to rotate the motor
// forward determines the direction the push plate moves (true -> forward,
    false -> backward)
void topMagStepper(unsigned int steps, bool forward){
    unsigned int topMagCurrentStep = 0;

    if (forward){
```

Figure 37: The values read when testing the downward-facing laser sensor. Note the terraced values: the values around 110 signify one tire, the values around 117 signify two tires, and the values around 103 signify no tires.

```
        topMagCurrentStep = 0;          // Start at 0 and count up if forward
    } else {
        topMagCurrentStep = steps;      // Start at steps and count down
    }
    while ((forward && (topMagCurrentStep < steps)) || (!forward && (
topMagCurrentStep > 0))){
```

```
12        // Different coils in the stepper are powered depending on the modulo
    of topMagCurrentSteps
13        if (topMagCurrentStep % 4 == 0){
14            LATBbits.LATB0 = 1;
15            LATBbits.LATB2 = 1;
16            LATBbits.LATB4 = 0;
17            LATBbits.LATB6 = 0;
18            if (forward){
19                // Count up
20                topMagCurrentStep++;
21            } else {
22                // Count down
23                topMagCurrentStep--;
24            }
25        } else if (topMagCurrentStep % 4 == 1){
26            LATBbits.LATB0 = 0;
27            LATBbits.LATB2 = 1;
28            LATBbits.LATB4 = 1;
29            LATBbits.LATB6 = 0;
30            if (forward){
31                topMagCurrentStep++;
32            } else {
33                topMagCurrentStep--;
34            }
35        } else if (topMagCurrentStep % 4 == 2){
36            LATBbits.LATB0 = 0;
37            LATBbits.LATB2 = 0;
38            LATBbits.LATB4 = 1;
39            LATBbits.LATB6 = 1;
40            if (forward){
41                topMagCurrentStep++;
42            } else {
43                topMagCurrentStep--;
44            }
45        } else if (topMagCurrentStep % 4 == 3){
46            LATBbits.LATB0 = 1;
47            LATBbits.LATB2 = 0;
48            LATBbits.LATB4 = 0;
49            LATBbits.LATB6 = 1;
50            if (forward){
51                topMagCurrentStep++;
52            } else {
53                topMagCurrentStep--;
```

```
            }
        }
        __delay_us(magazineStepperDelay);
    }
    // All coils are unpowered so the rod may be rotated by hand
    LATBbits.LATB0 = 0;
    LATBbits.LATB2 = 0;
    LATBbits.LATB4 = 0;
    LATBbits.LATB6 = 0;
}
```

The function rotates the stepper motor a given number of steps, which either moves the magazine push plate forwards or backwards. This function operated the motors in a full-drive pattern, which provides more torque than wave-drive or half-drive patterns. magazineStepperDelay was set to 1300μs, the value which moved the push plates the fastest (found through testing). The bottom magazine was operated using a similar function called botMagStepper, which controlled pins RB1, RB3, RB5, and RB7.

After a tire is pushed from the magazine, the kicker servo is operated:

```
for (i=kickerNeutral;i>59;i--){
    kicker.write(i);
    delay(8);
}
kicker.write(kickerNeutral);
```

where kickerNeutral is the default position of the kicker (127 degrees). The kicker is rotated to 60 degrees in order to kick the tire down the ramp. Both values were found through testing.

Next, the command to rotate the geared stepper motor to extend the arm is sent from the PIC to the Control Uno. The Control Uno extends the arm using the following code:

```
// Command 4: Extend arm
case 4:
    // Extend until PIC drives stopExtending HIGH
    while (!digitalRead(stopExtending)){
        // Step forward 1 step
        armStepper.step(1);
        // Increment stepper position counter
        armStepperCurrentPosition++;
        // If arm is near max extension, stop extending
        if (armStepperCurrentPosition >= 1750){
            break;
        }
    }
```

```
    // Extend 180 steps further to detect tires
    armStepper.step(180);
    // Add 180 to stepper position counter
    armStepperCurrentPosition += 180;
    // All coils are unpowered so the gear may be rotated by hand
    digitalWrite(armStepperIn1, LOW);
    digitalWrite(armStepperIn2, LOW);
    digitalWrite(armStepperIn3, LOW);
    digitalWrite(armStepperIn4, LOW);
    break;
```

The arm extends until the pole breakbeam is broken or until the stepper has rotated 1750 steps, whichever comes first. Next, it rotates another 180 steps in order to detect tires. If the top tire breakbeam is broken, there are two tires on the pole. If the top is unbroken but the bottom is, there is one tire on the pole. If neither are broken, there are no tires on the pole. This is checked by the PIC:

```
// Checks top tire breakbeam
if (!PORTEbits.RE1){
    tiresBefore = 2;        // Top beam broken, 2 tire on pole
// Top beam intact, checks bottom tire breakbeam
} else if (!PORTEbits.RE0){
    tiresBefore = 1;        // Top beam intact, bottom broken, 1 tire on pole
// Both beams intact, no tire
} else {
    tiresBefore = 0;
}
```

The arm is rotated back 950 steps in order to position the cell above the pole. The PIC then sends the cell deployment command to the Nano, which executes the following:

```
// Operate cell
if (digitalRead(startServos)){
    cell.write(cellNeutral+20);
    delay(350);
    gripper.write(gripperNeutral+30);
    delay(100);
    cell.write(cellNeutral-90);
    delay(600);
    gripper.write(gripperNeutral);
    delay(100);
    cell.write(cellNeutral);
}
```

with cellNeutral = 146 and gripperNeutral = 138. The servo angles and delays were deter-

mined through testing. If a pole requires two tires, the magazine starts pushing the second tire while the cell is deploying the first tire. The kicker and cell are then operated again for the second tire. After deployment, a tire is pushed from the magazine while the robot drives to the next pole, thus minimizing the amount of time required for pushing tires.

10.4 Suggestions for Improvement of the Subsystem

One possible improvement of the microcontroller subsystem would be moving the laser sensor onto the PIC I^2C bus for more centralized control. This way, the Nano could be controlled using I^2C instead of digital pins, which would make synchronizing the devices considerable easier.

Another improvement would be using both the encoder pins to record more accurate data. If both pins were attached, the robot would know when it was moving backwards and could decrement the encoder counter instead of both forwards and backwards motion increasing the count. The reason only one pin was used is because the Control Uno originally drove the motors and would be interrupted by the Nano when a pole was detected. Since Arduino Unos only have two interrupt pins, this meant that only one pin of the encoder could be used. After the Drive Uno was added, both pins of the encoder were connected to the Drive Uno, but unfortunately the bugs that arose due to this change could not be solved in time for demo and the change was reverted.

11 Integration

Week 9

- Tried using PID feedback system to drive straight
- Connected and mounted pole detection LED
- Straightened left wheel
- Continued calibrating sensors
  - Adjusted alignment of laser sensors to stop robot in the correct position
  - Calibrated arm extension by reading data from ultrasonic sensor
- Prototyped and tested audio amplification circuit

Week 10

- Tried using ultrasonic and laser sensors to count the number of tires on the pole
- Tried using a downwards-facing laser sensor that polled while the arm was extending to count the number of tires
- Cleaned up temporary wiring for sensor testing and soldered sensor circuits onto protoboard
- Removed temporary breadboard from the robot
- Secured mounting for both laser sensors to reduce wiggle

Week 11

- Made permanent connections for many components, including the magazine stepper motors
- Added plates to upper magazine pusher so tires are pushed all the way out of the magazine
- Stops when pole laser sensor senses something
- Extends arm based on ultrasonic reading
- Pushes tire from magazine when cell is empty

- Altered code so the robot will skip poles that are not in its arm's reach
- Reduced time for cell deployment
- digitalWrite doesn't write pins high instantly, so PIC wasn't synchronized with Uno
  - Added slight delay to allow pin to be driven high
- Attached encoder to the Uno
  - Pole distances can now be recorded
  - Used encoders to return robot to start, returns very close to original start position

Week 12

- Added two lasers and receivers to the arm of the robot to detect poles and upper tire
- Changed main power distribution to provide a more stable 12V rail for the DC motors and NEMA 17s
- Got arm extension and dropping working with the laser sensors
- Can push magazine and run DC motors at full power, but might be risky if batteries drain
  - Step down ∼16V to 12V to remedy this? Need >15V since the regulators need a 3V difference
- Cell sensor isn't very accurate and not much room for error (commonly reports wrong number of tires)
- Pole ultrasonic sensor gives wrong reading when 2 tires are on the pole
- Tried adding two ultrasonics for each tire, but readings between the 3 aren't consistent and the bottom one detects the base
- Tested break beam sensors
  - Works EXTREMELY well (sends signal once the beam is broken)
  - Very fast and accurate
  - Very easy to use (only one output pin)

- Since the cell needs to be over the pole and we can't mount anything on it (it's fragile and rotates), we may need to oversweep to sense the pole and tires, then retract the arm to centre the cell over the pole
- Break beams also give us more freedom in stopping: no longer have to stop within fractions of a centimeter to get correct ultrasonic readings

Week 13

- Continued debugging the robot and ironing out miscellaneous bugs
- Optimized runs to start with one tire in the cell and one in the ramp
- Decided to average two values from the laser sensor that detects the pole and tells the robot when to stop
  - Was having an issue were the robot would randomly stop and extend the arm even if a pole wasn't there
  - Averaging values should reduce any noise that the sensor may be picking up
- Continued trying to get the robot to drive straight
- Attached side panels to robot for aesthetics
- Ironed our issues with getting the operation time correct (switched to the RTC)
- Added laser to front of robot to assist with aligning the robot at the beginning of operations

Week 14

- Implemented gyroscope to correct drive motors if heading changes
- Implemented "subway sound" every time the robot detects a pole
- Bottom magazine was losing steps pushing forward
  - Turned out to be weak jumper cable to header connections on the ribbon cable board
  - Wires were soldered to fix this
- Fixed alignment of kicker (whacker) servo, tires no longer jam in the ramp
- Tried to use two pins on encoder

  - Started observing some weird behaviour, not enough time to correct it before demo, went back to one pin

- Refined the one-pin encoder distance function.
  - Final function: distance = encoder * 0.469 - 8.7

12 System Improvement Suggestions

There are several large-scale changes that we may recommend, but were unable to accomplish either due to limited budget or time. Smaller potential improvements are listed in the independent subsystem sections, but these changes are more major projects.

1. **Steeper Ramp**: During testing, we noticed that tires would often get jammed in the ramp. Much of the jamming can likely be attributed to the challenging tire properties noted in 9. A variety of potential solutions were tried, such as smoothing the ramp and increasing the plate push. Ultimately the issue was remedied by adding the kicker servo and sloping the pusher plate tabs, but occasionally tires would still get jammed. If there was more time, we would have likely adjusted the angle of the ramp itself, since making it steeper would give the tires greater velocity. Unfortunately this would have entailed removing the entire top of the robot, which was not feasible.

2. **Narrowing the Robot**: As a follow-up to increasing the ramp steepness, it would have been beneficial to reduce the width of the robot. This would have allowed for more arm range and reduced the risk of hitting poles. There was considerable unused internal volume in the final B.A.S.S., which likely could have been trimmed given enough time.

3. **Both Encoder Pins**: Using both pins of the encoder would allow us to measure distance travelled forwards and backwards. This would help account for poor stops, slips, etc. (since only using one pin means every tick, no matter which direction, increases the encoder count).

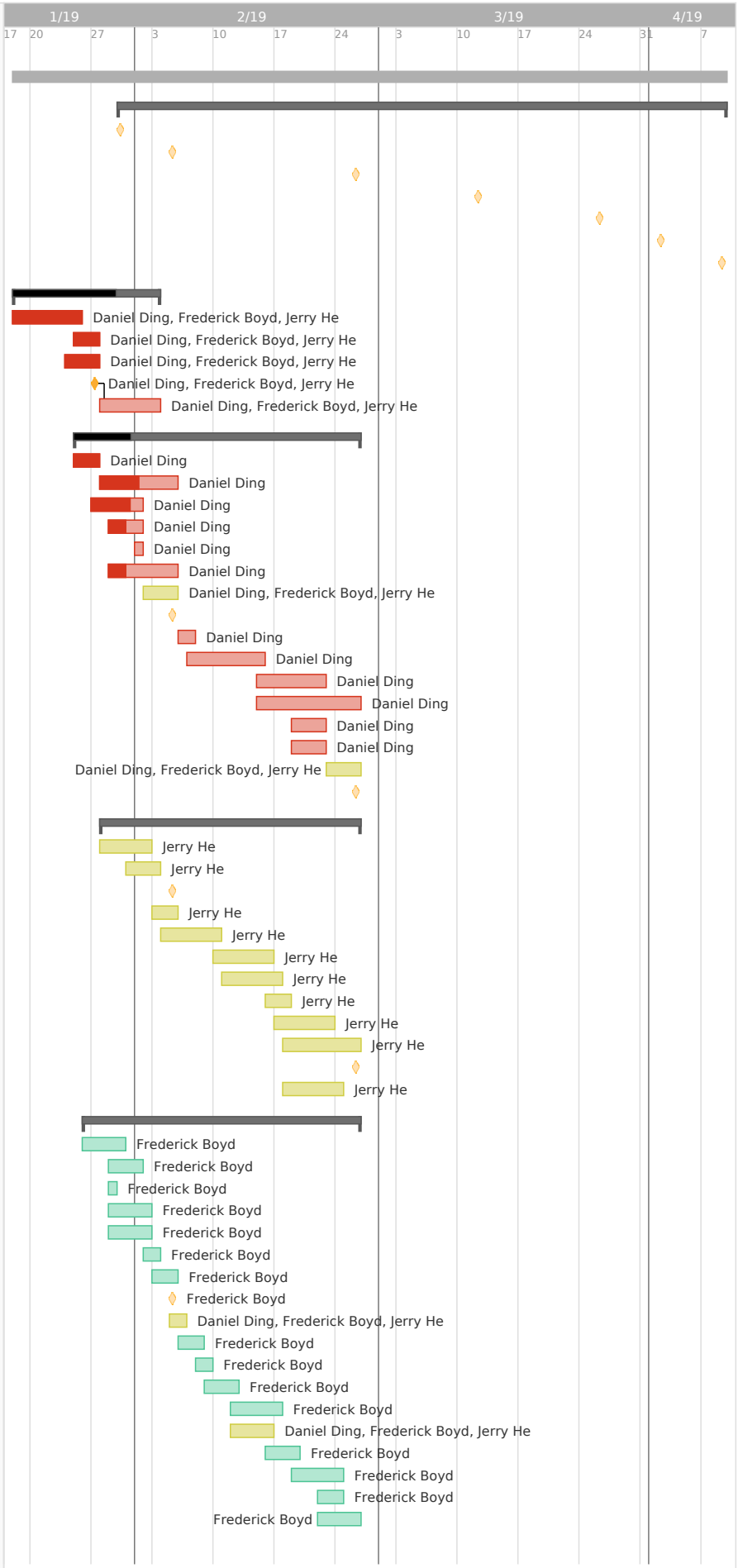
4. **Polling the VL53L0X laser sensor with the PIC**: Polling the laser sensor with the PIC would allow for more centralized control of sensors and make communication and synchronization with the Nano easier since the I^2C bus would be available.

13 Initial and Accomplished Schedule

The following Gantt chart is our schedule as it appears in our proposal (initial schedule).

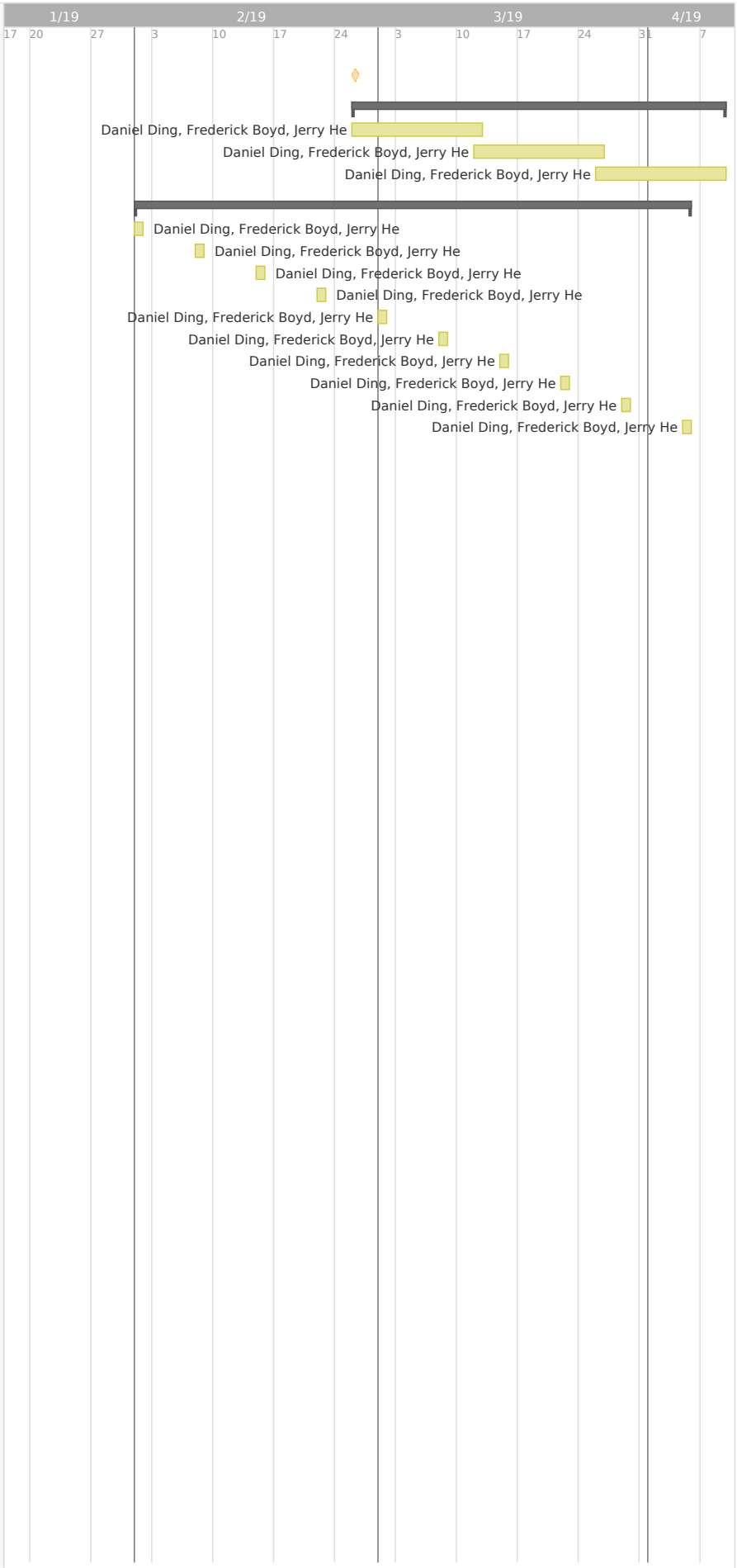
teamgantt
Created with Free Edition

AER201 Schedule	**0h**	**11%**
Assessments/Due Dates	**0h**	**0%**
Proposal Due	0	0%
Individual Evaluation Subsystem Inte...	0	0%
Individual Evaluation Subsystem Fun...	0	0%
Team Evaluation Subsystem Integrat...	0	0%
Team Evaluation Subsystem Functio...	0	0%
Project Review	0	0%
Public Demonstration	0	0%
Design	**0h**	**70%**
Diverging	0	100%
Converging	0	100%
Prototyping	0	100%
Finalize Design	0	100%
Acquire Components	0	0%
Building - Electromech	**0h**	**20%**
CAD Model	0	100%
Acquire Materials and Hardware	0	50%
Weight, Moment Analysis	0	75%
Base Frame	0	50%
PIC Board Tray	0	0%
Wheels, Axles, Motors	0	25%
Integration, Alignment	0	0%
Complete Driving Base	0	0%
Upper Frame, Columns	0	0%
Magazine, Ramp, Reloading	0	0%
Sliding Arm, Linear Gear	0	0%
Rotating Cell, Clamp Servo	0	0%
Sensor Panel	0	0%
LCD, Keypad, Emergency Stop	0	0%
Integration Check	0	0%
Complete Subsystem	0	0%
Microcontroller	**0h**	**0%**
Keypad/LCD UI	0	0%
Pseudo-code of final program	0	0%
Keypad/LCD and Pseudocode done	0	0%
Sensor code	0	0%
Motor code	0	0%
Stepper motor code	0	0%
Servo code	0	0%
Arm and cell code	0	0%
Magazine code	0	0%
Combine into final program	0	0%
Complete final program	0	0%
Driving code	0	0%
Circuits	**0h**	**0%**
Create Schematics	0	0%
Power Requirements	0	0%
Prototype Drive Motors	0	0%
Prototype Servos	0	0%
Prototype Stepper Motors	0	0%
Prototype Sensor Circuits	0	0%
Debugging	0	0%
Complete Circuit & Sensor Prototypes	0	0%
Brief Integration Check	0	0%
Optimize Circuits	0	0%
Drive Motor Circuits	0	0%
Cell Servo Circuit	0	0%
Rotating Arm Servo Circuit	0	0%
Integration Check	0	0%
Tire Magazine Stepper Motor Circuits	0	0%
Arm Extension Circuits	0	0%
Verify Operation of Circuits	0	0%
Debugging	0	0%



teamgantt
Created with Free Edition

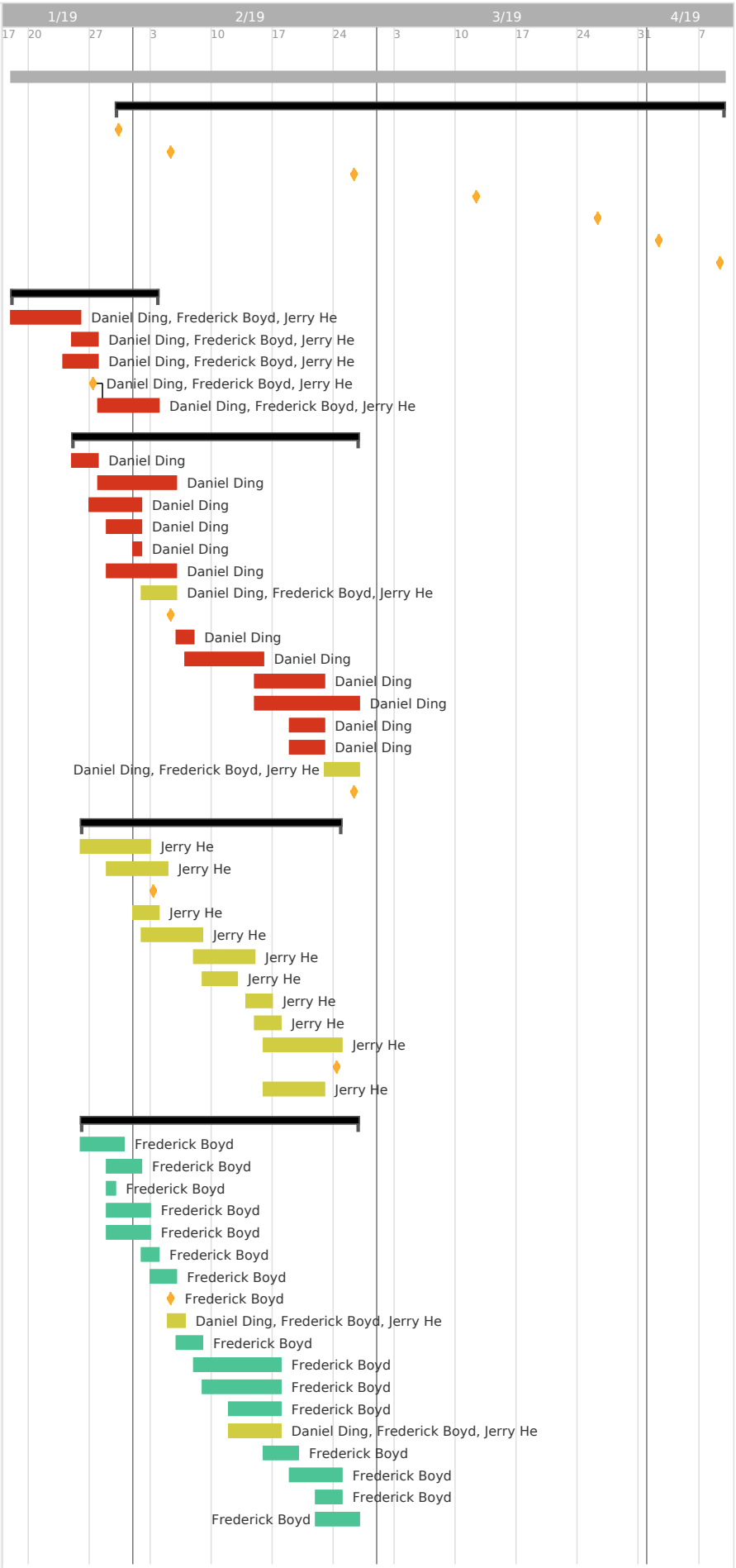
Complete All Circuits	0	0%
Team	**0h**	**0%**
System Integration	0	0%
System Functionality	0	0%
Debugging	0	0%
Meetings	**0h**	**0%**
2/1	0	0%
2/8	0	0%
2/15	0	0%
2/22	0	0%
3/1	0	0%
3/8	0	0%
3/15	0	0%
3/22	0	0%
3/29	0	0%
4/5	0	0%



The following Gantt chart is our completed/accomplished schedule.

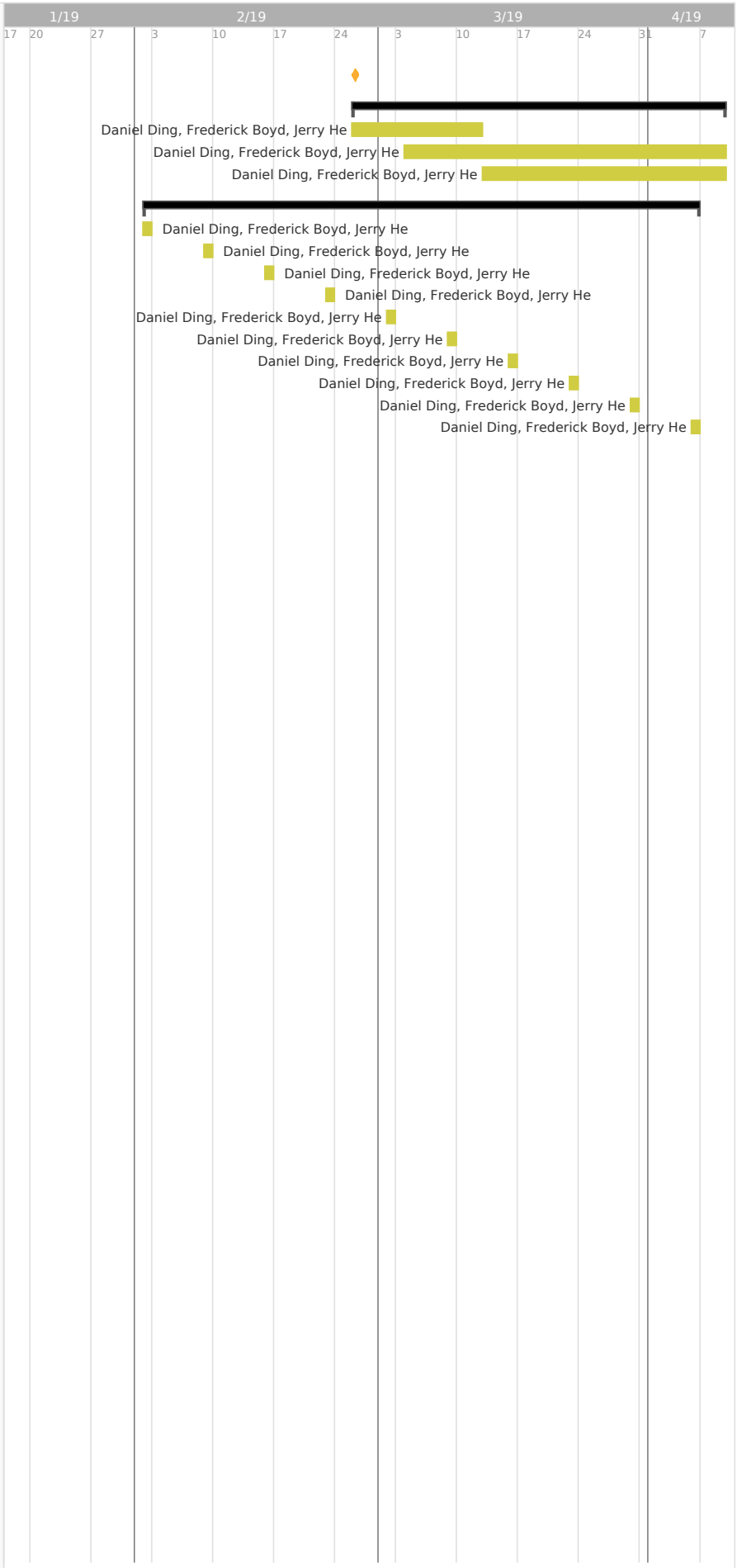
teamgantt
Created with Free Edition

AER201 Schedule	**0h**	**100%**
Assessments/Due Dates	**0h**	**100%**
Proposal Due	0	100%
Individual Evaluation Subsystem Inte...	0	100%
Individual Evaluation Subsystem Fun...	0	100%
Team Evaluation Subsystem Integrat...	0	100%
Team Evaluation Subsystem Functio...	0	100%
Project Review	0	100%
Public Demonstration	0	100%
Design	**0h**	**100%**
Diverging	0	100%
Converging	0	100%
Prototyping	0	100%
Finalize Design	0	100%
Acquire Components	0	100%
Building - Electromech	**0h**	**100%**
CAD Model	0	100%
Acquire Materials and Hardware	0	100%
Weight, Moment Analysis, Calcs	0	100%
Base Frame	0	100%
PIC Board Tray	0	100%
Wheels, Axles, Motors	0	100%
Integration Check 1	0	100%
Complete Driving Base	0	100%
Upper Frame, Columns	0	100%
Magazine, Ramp, Reloading	0	100%
Sliding Arm, Linear Gear	0	100%
Rotating Cell, Clamp Servo	0	100%
Sensor Panel	0	100%
LCD, Keypad, Emergency Stop	0	100%
Integration Check 2	0	100%
Complete Subsystem	0	100%
Microcontroller	**0h**	**100%**
Keypad/LCD UI	0	100%
Pseudo-code of final program	0	100%
Keypad/LCD and Pseudocode done	0	100%
Sensor code	0	100%
Motor code	0	100%
Stepper motor code	0	100%
Servo code	0	100%
Arm and cell code	0	100%
Magazine code	0	100%
Combine into final program	0	100%
Complete final program	0	100%
Driving code	0	100%
Circuits	**0h**	**100%**
Create Schematics	0	100%
Power Requirements	0	100%
Prototype Drive Motors	0	100%
Prototype Servos	0	100%
Prototype Stepper Motors	0	100%
Prototype Sensor Circuits	0	100%
Debugging	0	100%
Complete Circuit & Sensor Prototypes	0	100%
Brief Integration Check	0	100%
Optimize Circuits	0	100%
Drive Motor Circuits	0	100%
Cell Servo Circuit	0	100%
Rotating Arm Servo Circuit	0	100%
Integration Check	0	100%
Tire Magazine Stepper Motor Circuits	0	100%
Arm Extension Circuits	0	100%
Verify Operation of Circuits	0	100%
Debugging	0	100%



teamgantt
Created with Free Edition

Complete All Circuits	0	100%
Team	**0h**	**100%**
System Integration	0	100%
System Functionality	0	100%
Debugging	0	100%
Meetings	**0h**	**100%**
2/2	0	100%
2/9	0	100%
2/16	0	100%
2/23	0	100%
3/2	0	100%
3/9	0	100%
3/16	0	100%
3/23	0	100%
3/30	0	100%
4/6	0	100%



Shortly after proposal, we realized that Saturday was a more convenient time for meetings, so all the Friday meetings were pushed one day later. All of our individual subsystems were completed well before the Week 8 Individual Subsystem Evaluation; we had already begun to integrate our design. By the Week 10 Integration Team Evaluation, we had already achieved a high level of functionality: we were able to deploy tires on poles autonomously. By the Week 12 Functionality Team Evaluation, we had achieved full functionality with occasional bugs (such as tires getting jammed in the ramp). Jammed tires and driving straight were two of our biggest issues in the weeks leading up to the public demonstration, and so much of our time was dedicated to solutions to those problems and debugging said solutions. As a result, functionality and debugging were both ongoing up until demonstration day. Additionally, some aspects of our subsystems that had been completed earlier had to be altered during integration and debugging, such as the microcontroller member's driving program and electromechanical member's sensor panel. Apart from these deviations, the remainder project was carried in accordance with the initial schedule.

14 Overall Machine Description

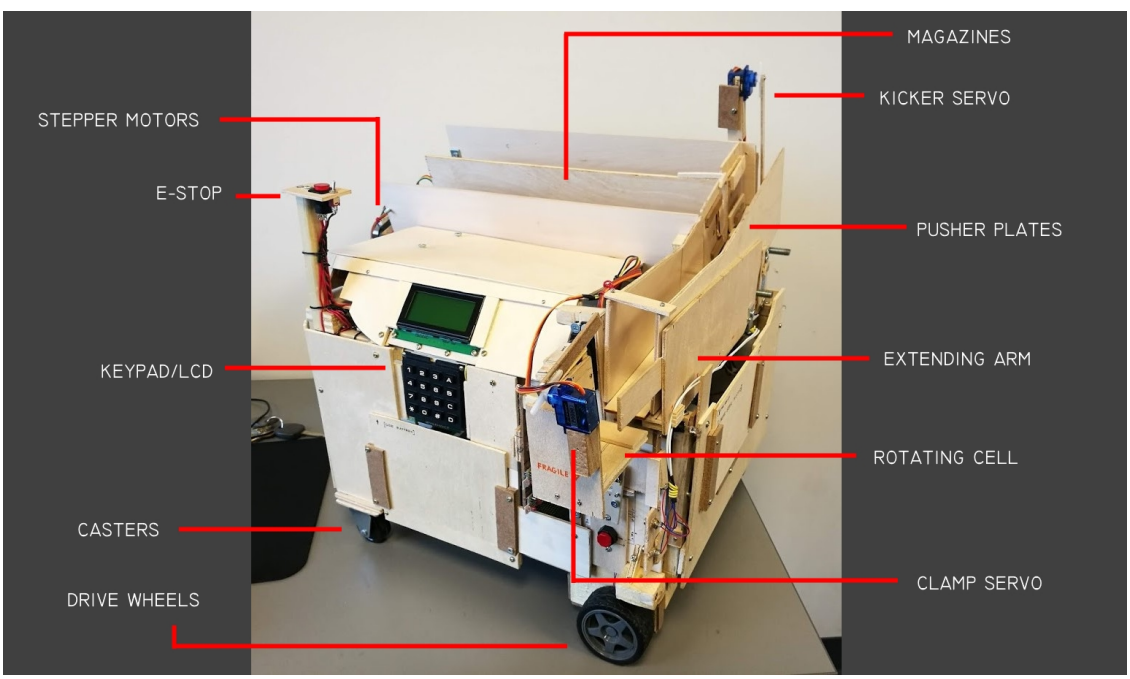


Figure 38: Overall diagram of the robot, labelling major features

The B.A.S.S. is a 48x46x48 cm, 5.1 kg mobile autonomous tire stacking robot. The machine is controlled by a combination of a PIC 18F4620, an Arduino Nano and 2x Arduino

Unos. It is powered by 13 AA NiMH rechargeable batteries. After being loaded with tires and aimed down the left side of a row of poles, the machine operates completely autonomously. The B.A.S.S. senses and stops beside the nearest pole using a time-of-flight laser distance sensor. An arm fitted with laser breakbeam sensors is then extended to sweep across the pole. These breakbeams are able to determine the number of tires present, along with the distance to the pole from the robot. Once ready to deploy, tires are pushed from two threaded-rod driven magazines. A max capacity of up to 26 tires can be accommodated. Pushed tires are hit with a kicker that rolls them down a ramp, through the terminal end of the arm, and into a rotating cell. This cell clamps and flips the tires 90° so that they are ready to deploy; after the rotation, the tires should be in the perfect position to drop onto the pole, with no part of the robot between the tire and the pole. Once deployment is completed, the cell returns to its default state and the arm retracts. The robot then drives forward towards the next pole, and the process is repeated. This continues until the tire load is depleted, or a variety of end conditions are met (time limit approaching 3 minutes, 4 metres travelled, or 10 poles). This describes the bulk of the robot's functionality.

Table 9: Full specifications sheet for the B.A.S.S. robot

Breakbeam Assisted Smart Stacker Full Specification Sheet	
Property	**Specifications**
Length	48 cm
Width	46 cm
Height	48 cm
Mass	5.1 kg
Materials	Wooden frame, metal fasteners and screws, Epoxy
Drivetrain	Front wheel drive, 6 cm diameter rubber wheels, 2x Shenzhen DC GP02S-A130 4150-120, torsion bar suspension, rear fixed casters
Tire Storage	26 tires at maximum capacity, twin threaded-rod magazines driven by Casun NEMA 17 42SHD0001-24B, tire kicker using K-Power P0090 microservo
Sensor Array	3x laser pointer and receiver, 1x VL53L0X TOF laser distance sensor
Deployment	Extendable arm via rack and pinion (1x 28BYJ-48 Stepper), tire gripper (1x K-Power P0090 microservo), rotating cell (1x Towerpro MG958 High Torque Servo)
Microcontrollers	1x PIC 18F4620, 2x Arduino Uno, 1x Arduino Nano
Pins	Total of 95 microcontroller pins usable for sending signals, communication, or data transfer
Memory	Capable of storing up to 4 runs in EEPROM memory
Real Time Clock (RTC)	RTC module capable of keeping time even when robot is powered off
Power Supply	13 AA NiMH Rechargeable Batteries (15.6V)
12V Rail	12V rail capable of providing up to 5A of continuous current regulation
5V Rail	5V rail capable of providing up to 5A of continuous current regulation
Sound System	0.5W mono speaker with AUX cord

The following subsections detail the individual systems that drive this functionality. The machine can be divided into several major physical components: The controllers, power supply, underlying frame, drivetrain, magazines, extending arm, sensors, and rotating cell. The following subsections will provide more detailed information on each aspect. A full specification sheet can be seen on Table 9.

14.1 Microcontrollers

The PIC is responsible for all of the user interface, pushing both magazines, polling the breakbeams, and coordinating/commanding all other devices. The PIC (master) communicates "Control Uno" (slave) via I^2C, which allows for flexibility in terms of commands and data transfer. The Control Uno is responsible for the arm stepper motor, the rotary encoder, and commanding the "Drive Uno" to start and stop using digital pins. The PIC communicates with the Arduino Nano using digital pins. The Nano is responsible for polling the laser sensor/sending a stop request to the Drive Uno and operating the gripper, cell, and kicker servos. Lastly, the Drive Uno is responsible for driving/stopping the DC drive motors, polling the gyroscope/making course adjustments, activating the pole detection LED bar, and operating the speaker to play the pole detected sound. There are pins and delays dedicated to synchronizing the four microcontrollers and ensuring that operations go according to plan.

14.2 Power Supply

The power supply is responsible for providing a reliable and safe source of power to all microcontrollers, sensors, and motors on the robot. It consists of 13 AA NiMH rechargeable batteries arranged in series to produce approximately 15.6V. All components on the robot require either 5V or 12V, so the battery's voltage is regulated down to 12V and 5V using LM338 adjustable voltage regulators. These adjustable voltage regulators are capable of regulating at a continuous current of up to 5A. Capacitors between V_{in}, V_{out}, and GND help to minimize noise. It is critical that these regulators function correctly during operations, so large heat sinks were placed on them to ensure they do not overheat.

14.3 Structural Frame

The underlying frame supports the entire mass of the robot. It is composed of a rectangular prism with wooden beams forming the major edges. The frame measures 36 cm in length, 29 cm in width, and 18 cm in height (not including superstructure, protrusions, additional

hardware). The frame is screwed together at the vertices.

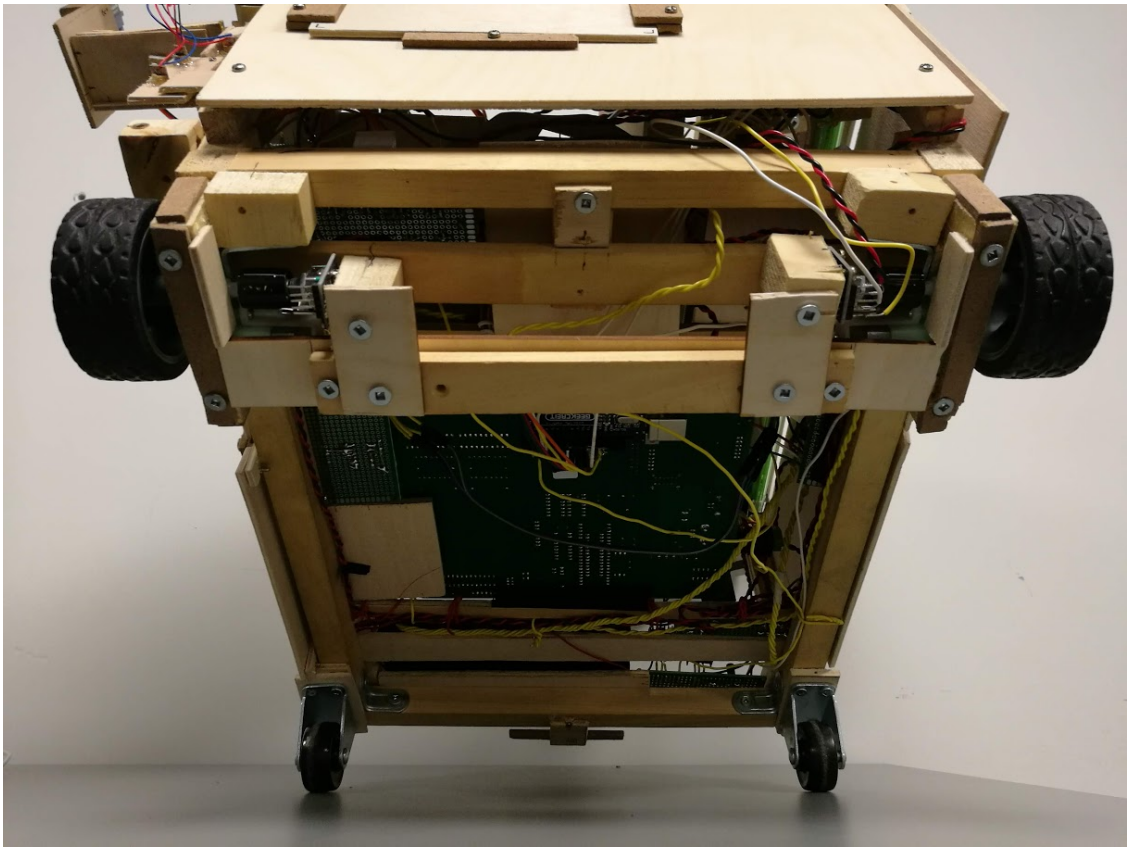


Figure 39: The overall layout of the robot bottom. Note the main rectangular frame, drive wheel mounts, and the suspension bar running across the width of the robot.

Cross beams at the bottom of the robot serve as mounting points for the PIC Development Board, Arduinos, batteries, and other associated circuitry (see Figure 39). Smaller rigid surfaces are mounted throughout the robot, providing attachment points for motor driver boards, sensor circuitry, etc. A pedestal containing key controls rises from the right rear corner of the robot, to an easily-accessible height. Mounting points are also provided for a user LCD and keypad. The entire frame is covered by exterior plating. These panels are designed to be highly modular, sliding in and out of slots. There are 5 removeable panels: Rear (Main Battery), Right (Side Battery), Front (Linear Gear), Left (I/O) and Top (Stepper Driver Boards). When access to internals is required, a user can remove the panels within seconds. When not in use they are firmly secured.

14.4 Drivetrain

The machine has two front drive wheels (6 cm diameter rubber tires) and two rear fixed caster wheels. Two Shenzhen DC Angled Motors (*Model GP02S-A130 4150-120*) are directly

connected to the front wheels. They are housed in twin motor mounts that extend beneath the main frame. A torsion bar keeps the wheels aligned, and a gyroscope (MPU6050) ensures straight driving.

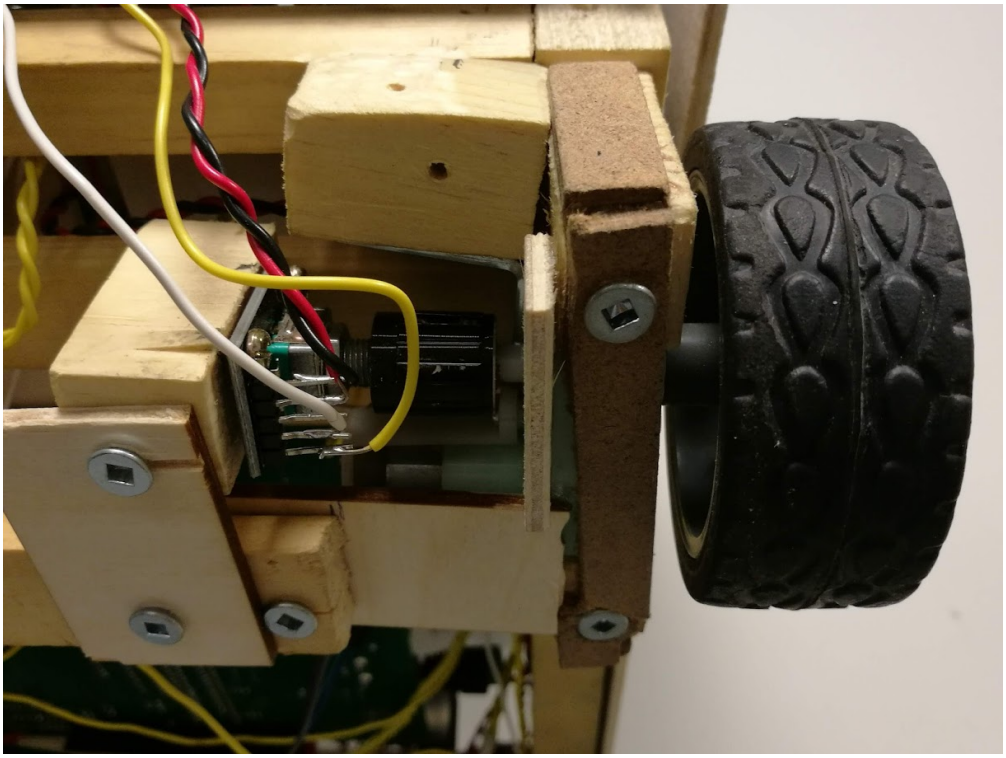


Figure 40: Close-up of the motor mount.

A detailed view of the motor mount is visible in Figure 40. The green portions are the DC motor itself. The torsion bar which minimizes camber is visible on the bottom (the thin flat piece of wood), running across the width of the robot. The MDF strips and wooden block on the right are the main mount for the motor, providing vertical load support. The block on top (with a drill hole) has a metal L-bracket that prevents z-axis rotation of the motor. The block on the left, attached to the torsion bar, is a mounting point for an encoder. The black cylinder in the center is the coupling between the motor axle and the encoder.

14.5 Tire Storage

The B.A.S.S stores up to 26 tires when loaded to maximum capacity. A tire can be pre-loaded into the rotating cell, and another in the deployment ramp itself. The remaining 24 are split between double horizontal magazines. The entire magazine assembly is mounted on the top left of the robot, with the floor tilted down towards the right. This tilt allows the tires to roll out down a ramp (see Figure 41).

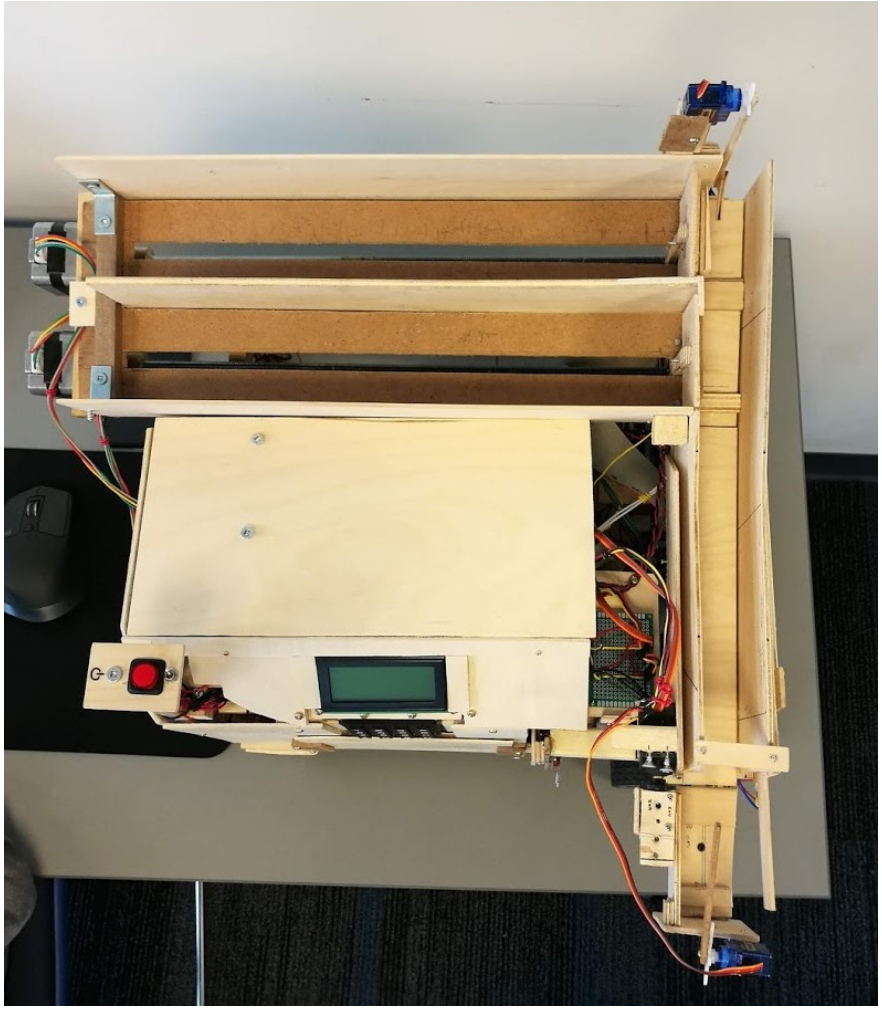


Figure 41: An overhead view of the robot, illustrating the two magazines, ramp and rotating cell.

The tires are moved along using pusher plates. Each plate slides along a track in the floor. Underneath the floor, the plate is mounted to a nut. This nut spins along a threaded rod, driven by a Casun NEMA 17 42SHD0001-24B stepper motor (see Figure 42). The rods are greased for quiet and smooth operation. The floors of the magazines use MDF in order to reduce tire contact friction.

Once a tire is fully ejected from the magazine, it arrives at a ramp that leads to the rotating cell (see 14.8). Small 'speedbumps' on the ramp floor prevent the tires from immediately rolling down. To guarantee tire ejection, a K-Power P0090 microservo fitted with a 'kicker' arm then sweeps the ramp, lifting the tire over the bump and launching it towards the cell. The kicker arm curves to match the shape of the tires; this allows it to perform a clean lifting motion. The kicker arm and pusher plates can be seen in Figure 43.

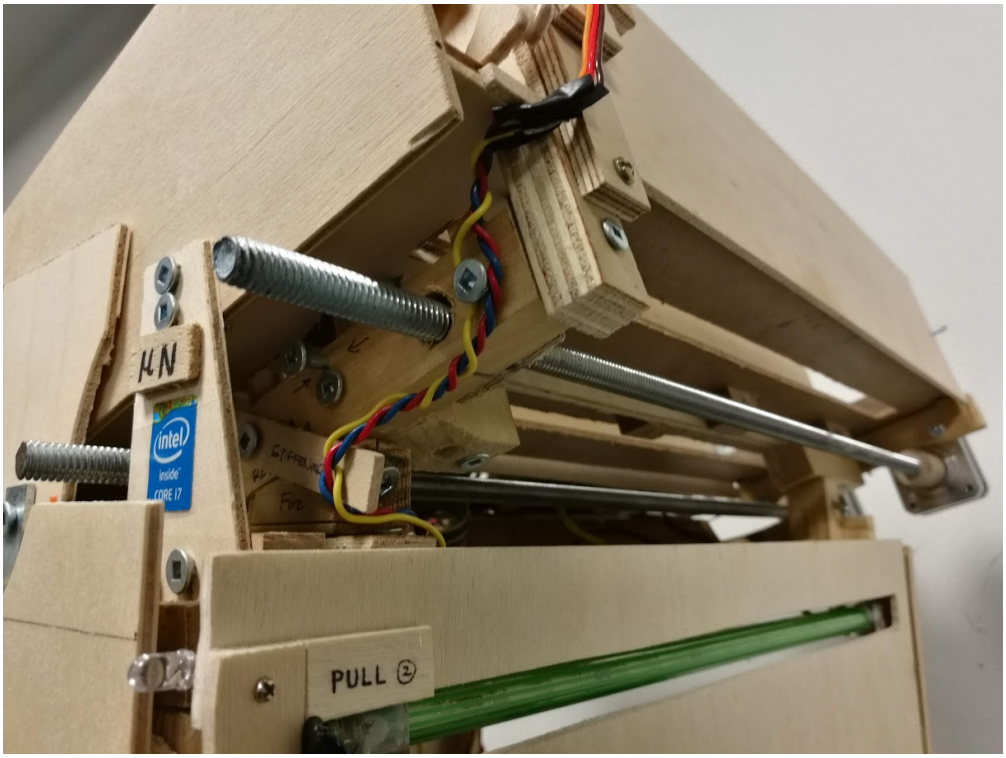


Figure 42: This photo, looking back along the left side of the robot, illustrates the threaded rod mount. The pusher plates are not visible as they are in the full forward position. Note the tilt of the magazines, the MDF floor and its cut-out track, and the NEMA steppers located in the rear. The wiring down the middle connects to the kicker arm above.

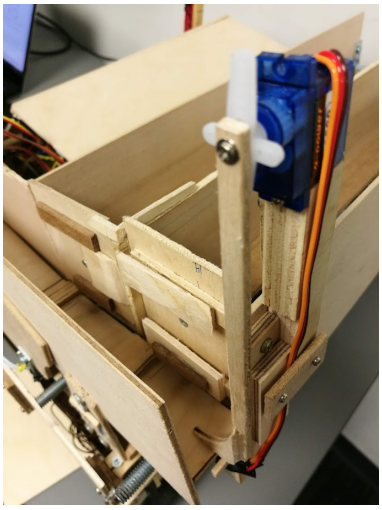


Figure 43: The pusher plates, kicker arm and ramp bumps. Note the slopes on the pusher plate tabs and the curved end of the kicker arm.

14.6 Extending Arm

To account for side-to-side deviations in pole placement, the B.A.S.S. is built around an extendable arm. This arm uses a rack and pinion system, and is powered by a small 28BYJ-48 stepper motor. The rack is attached to the underside of the arm, which slides laterally. The stepper and pinion are mounted to the front right support column of the robot.

The extending arm is built to fit around a slightly smaller ramp of the same length. This inner ramp is fixed in place; at maximum extension, the outer walls of the arm smoothly continue the inner walls. A close up of the final arm can be seen in Figure 44. A cross sectional diagram of this system can be seen in Figure 45. When fully extended, the arm is allowed to droop slightly, ensuring that tires can still roll down.

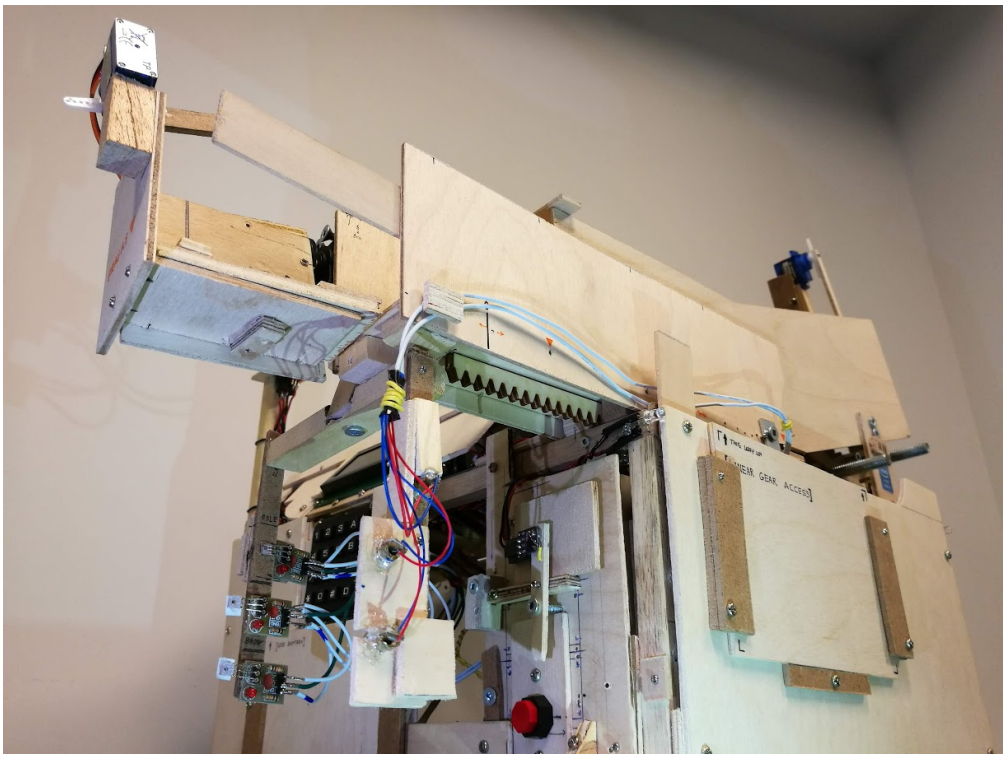


Figure 44: Underside view of the robot arm. Note the inner ramp above, and the larger outer arm walls. A portion of the rack gear is visible, along with the breakbeam array.

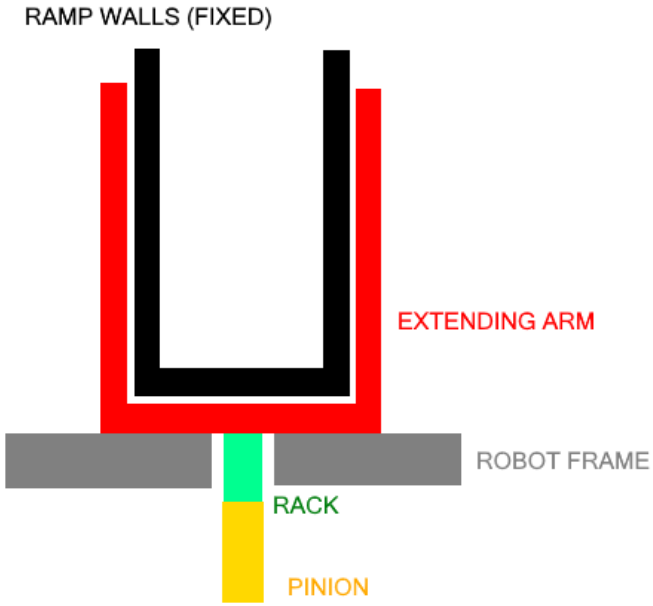


Figure 45: Cross Sectional Drawing of Arm Illustrating Various Layers.

14.7 Sensors

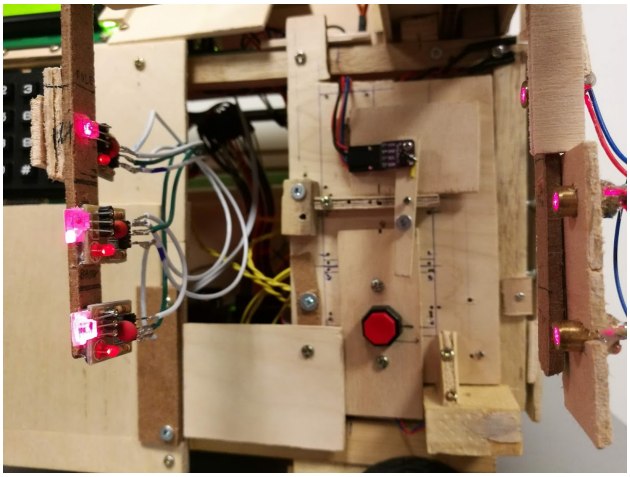


Figure 46: Triple breakbeams mounted underneath extending arm, with fixed VL53L0X TOF sensor in the background. Note the offset lateral positioning of the bottom two breakbeams versus the top.

Attached to the arm is one of the most important components of the machine: the sensor array. This is mounted beneath the arm, right before the rotating cell. Taking the form of a Π, this structure contains three laser pointers and opposing receivers (breakbeam sensors).

The bottom two breakbeams are slightly offset from the top. A single VL53L0X time of flight sensor, mounted on a fixed location on the robot frame, is used for initial pole detection and stopping the driving. Once the stop is complete, the arm-mounted breakbeam array sweeps out across the pole. The stop is timed so that the lasers and receivers should box in the pole. Once the top beam is broken (indicating a pole has been hit), the bottom two are used to scan for tires. Depending on how many bottom beams are broken, the number of tires can be determined. The sensor array can be seen in Figure 46.

14.8 Rotating Cell

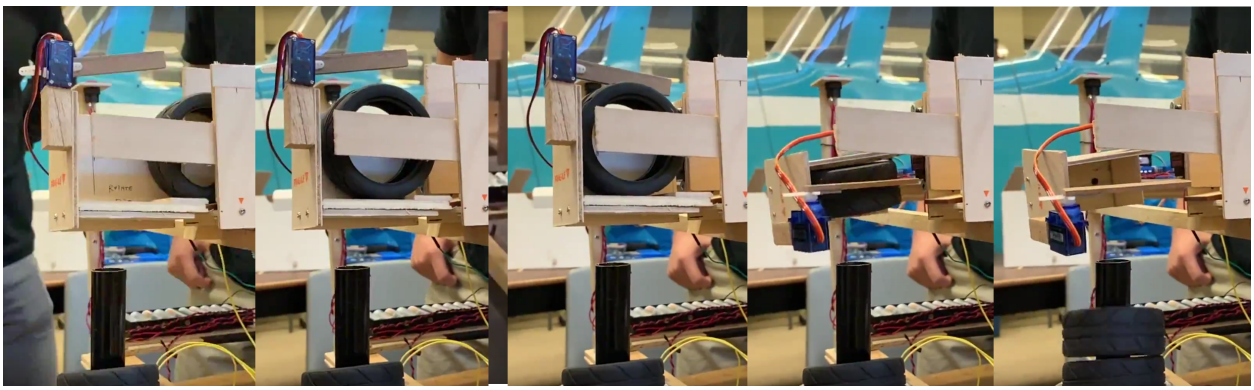


Figure 47: The drop sequence of the rotating cell. From L to R: 1. The tire rolls from the magazine into the cell. 2. The tire settles into the correct position. 3. The microservo clamps down. 4. The cell rotates. 5. The clamp releases and the tire drops onto the pole.

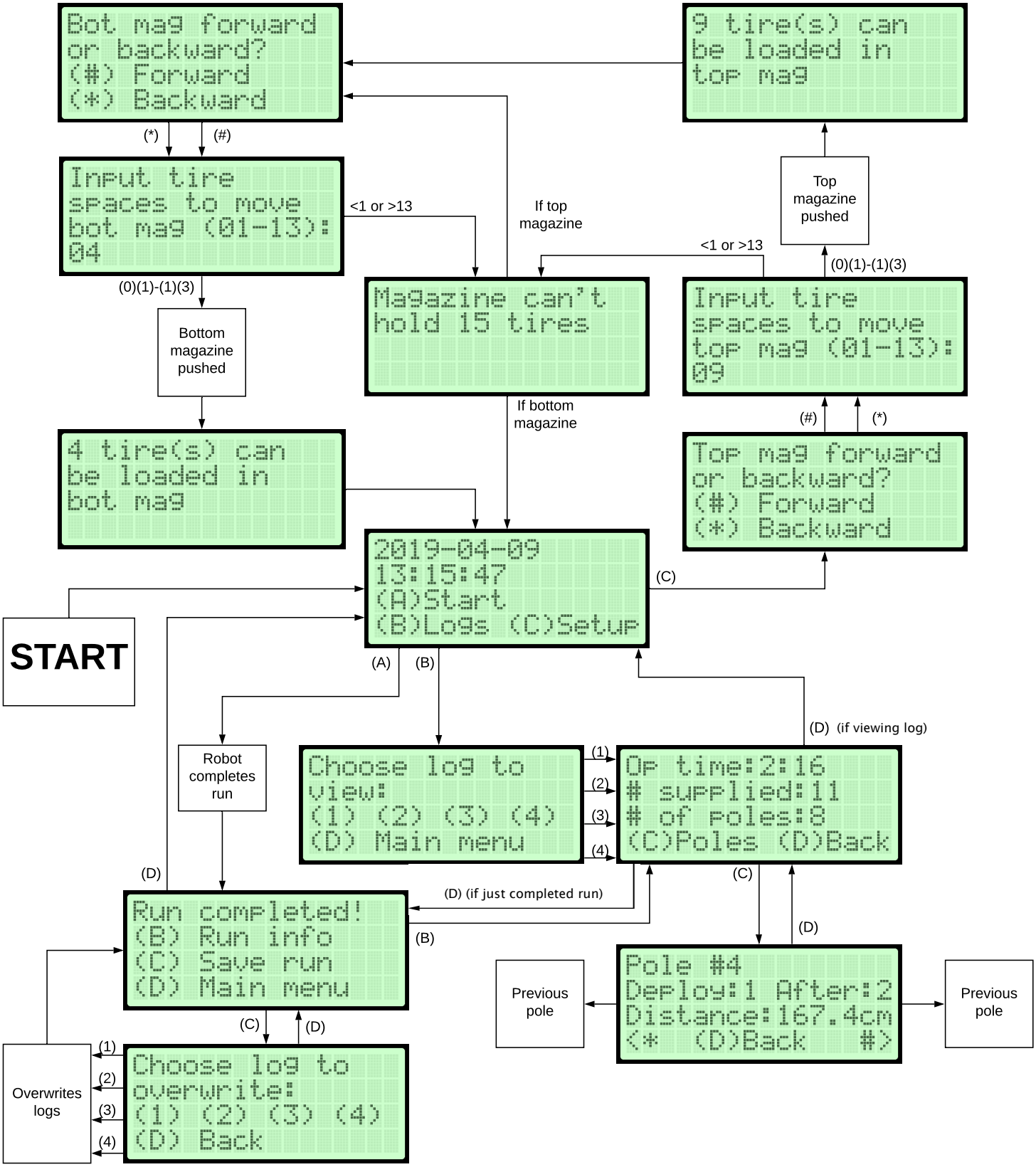
At the end of the extending arm is the central element of the B.A.S.S. deployment system, the rotating cell. The purpose of this cell is to receive tires in a rolling configuration, rotate them 90°, and deposit them onto the pole. This design provides a high degree of control and predictability, due to its extensive use of actuators (see 9.2.6 for more detailed information). Assuming a correct stop position, the combination of breakbeam-assisted arm extension and the rotating cell ensure nearly perfect, consistent accuracy.

The floor of the rotating cell rests approximately 0.5 cm below the level of the extendable arm. This slight drop is to ensure that the tire cannot rebound off the cell's wall and roll back up the arm. Small wooden strips on the floor of the cell help to ensure that the tire settles in the correct position. Once the tire is inside the cell, a K-Power P0090 micro-servo mounted on the top rotates so that it secures the tire with its lever. This clamps the tire firmly enough that it cannot escape even if the cell moves. The entire cell will then rotate, so that the tire is oriented ready for dropping. This is via a Towerpro MG996R high-torque servo, mounted on the right wall of the arm. The micro-servo will then release its grip,

allowing the tire to fall directly onto the pole. The cell will rotate back so that another tire can be reloaded or the machine can move on to the next pole. The drop sequence is fully illustrated in Figure 47.

15 Standard Operating Procedure

The following page contains the final Keypad-LCD UI. START is the initial state of the program when the robot is turned on


Bot mag forward
or backward?
(#) Forward
(*) Backward
9 tire(s) can
be loaded in
top mag
(*)
(#)
Input tire
spaces to move
bot mag (01-13):
04
<1 or >13
If top
magazine
Top
magazine
pushed
<1 or >13
(0)(1)-(1)(3)
(0)(1)-(1)(3)
Magazine can't
hold 15 tires
Input tire
spaces to move
top mag (01-13):
09
Bottom
magazine
pushed
If bottom
magazine
(#)
(*)
4 tire(s) can
be loaded in
bot mag
Top mag forward
or backward?
(#) Forward
(*) Backward
2019-04-09
13:15:47
(A)Start
(B)Logs (C)Setup
(C)
START
(A)
(B)
(D) (if viewing log)
Robot
completes
run
Choose log to
view:
(1) (2) (3) (4)
(D) Main menu
(1)
(2)
(3)
(4)
Op time:2:16
supplied:11
of poles:8
(C)Poles (D)Back
(D)
(D) (if just completed run)
(C)
Run completed!
(B) Run info
(C) Save run
(D) Main menu
(B)
(D)
Previous
pole
Pole #4
Deploy:1 After:2
Distance:167.4cm
<* (D)Back #>
Previous
pole
(C)
(D)
Overwrites
logs
(1)
(2)
(3)
(4)
Choose log to
overwrite:
(1) (2) (3) (4)
(D) Back

1. Turn the robot on by pressing the red LED button on the top of the button pedestal.
2. Press (C) to enter the tire loading menu.
3. Input whether you want to decrease the number of tires that can be loaded (#) or increase the number that can be loaded (*) in the top magazine
4. Input the number of tires you want to increase/decrease the top magazine capacity by. When entering single digit numbers, be sure to press (0) before the desired number.
5. Input whether you want to decrease the number of tires that can be loaded (#) or increase the number that can be loaded (*) in the bottom magazine
6. Input the number of tires you want to increase/decrease the bottom magazine capacity by.
7. To assist with aiming the robot, the octagonal switch on the front right of the robot can be pressed to activate a laser pointer. This laser pointer should point directly down the track and not hit any poles/bases.
8. Once the robot is aimed, let it remain stationary for ∼25 seconds so that the gyroscope may stabilize.
9. Press (A) to begin the operation. The operation is completely autonomous, and will terminate when the robot runs out of tires, if it reaches the desired number of poles, if it reaches 4m, or if 3:00 elapses.
10. To view logs from the run completed/termination message menu, press (B).
11. To view info about individual poles, press (C). If no poles were detected in the previous run, this won't be an option.
12. Use (*) and (#) to scroll through poles. Press (D) to return to the run info menu.
13. Press (D) to return to the run completed/termination message menu.
14. Press (C) to save the run in EEPROM memory.
15. Choose a log to overwrite: (1), (2), (3), or (4). Press (D) to return to the run completed/termination message menu.
16. Press (D) to return to the main menu.
17. To view a saved log, press (B).

18. Choose a log to view: (1), (2), (3), or (4). Press (D) to return to the main menu.

19. Viewing a saved log has the same controls as viewing the info of a just completed run, except pressing (D) at the run info menu will return you to the main menu.

20. To turn the robot off or emergency stop a run, depress the red LED button on the top of the button pedestal.

16 Conclusions

The B.A.S.S. prototype represents the product of a semester's worth of designing, building, and testing. The problem presented by the tire stacking RFP was both challenging and stimulating. Many unique solutions were generated, and it was extremely interesting to see our colleagues' approaches to solving the same problem. Many late nights were had, perhaps later and later as the semester progressed. There were major setbacks when dealing with poor sensors, power supply issues, crashing code, grinding components... It was discouraging to see, but the team recovered from its stumbles and learned more every time. We learned to prepare for uncertainty, troubleshoot methodically, and approach engineering design in a rational way.

Ultimately, a machine was created that satisfied and exceeded the acceptance criteria that we set. On April 9, 2019, the AER201 public demonstration for the tire RFP took place. We were scheduled to go first on the start line. Two runs were permitted. On the first run, the team was rather stressed and frantic during preparation. The aim of the robot was too conservative (aimed away from the poles, to limit risk of disqualification), and unfortunately we drifted away from the last few poles. On our second run, we learned from our mistakes and readjusted our aim, leading to a perfect run of nine poles. This exceeded our expectations by far.

From our observations during the competition, several key factors appeared to greatly affect the performance of the robots. Sensing was likely the most important; many robots had a great deal of mobility due to their use of omniwheels or fast motors, but their sensors failed to account for poles, or gave false readings. Driving straight was another huge factor - many robots drifted and collided with poles, or drove out of range. Of course, even a relatively straight-driving robot can still be undermined by poor aim (as evidenced by our first run). Mechanical failure was the third most common issue; jamming of the deployment or storage mechanisms was usually unrecoverable and impacted the remainder of the run.

Mechanical soundness and accurate sensors were the greatest indicators of success. The combination of straight gyroscope-enabled driving, breakbeam-assisted sensing and the relia-

bility of the rotating cell greatly bolstered the abilities of the B.A.S.S. Of course, any project can always be improved, and this one is no exception. From better motors, to steeper ramps, to enhanced encoders, there are always features to upgrade. We greatly appreciated having the opportunity to work on such a satisfying and involved project. We would like to thank all acknowledged parties again. This has been a truly unique experience and one we will surely look back on with fond memories. "This is, without a doubt, the *finest* engineering program in the English speaking world" [36].

Appendix A Full Program Code

A.1 PIC Main Program

```
#include <xc.h>
#include <stdio.h>
#include <stdbool.h>
#include "configBits.h"
#include "lcd.h"
#include "I2C.h"
#include "I2C.c"

// Struct that contains info for each pole
typedef struct{
    unsigned char tiresDeployed;
    unsigned char tiresAfter;
    unsigned int dist;    // in mm
} pole;

// Struct that contains info for each run
typedef struct{
    unsigned char totalPoleNum;
    unsigned char opTime;   // in seconds
    unsigned char tiresSupplied;
    pole poleArray[10];
} run;

volatile unsigned char i;
volatile unsigned char keypress;
volatile int startTime[7];
volatile int endTime[7];
const unsigned int magazineStepperDelay = 1300;   //1400 microseconds
const unsigned int singleTireSteps = 3925;
const unsigned char maxPoles = 10;

// Used to set RTC
const char inputTime[7] = {
    0x00, // 0 Seconds
    0x47, // 59 Minutes
    0x1, // 24 hour mode, set to 17:00
    0x01, // Sunday
    0x09, // 7th
    0x04, // April
```

```
    0x19  // 2019
};

void writeByteToControlArduino(unsigned char byteToWrite){
  __delay_us(100);          // Allow Control Uno to drive unoBusy HIGH
  while (PORTCbits.RC2){}      // Wait until Control Uno isn't busy
    I2C_Master_Start();        // Start condition
    I2C_Master_Write(0b00010000); // 7 bit Arduino address + Write
    I2C_Master_Write(byteToWrite);  // Writes command to Control Uno
    I2C_Master_Stop();         // Stop condition
    return;
}

void mainMenu(unsigned char *menu){
    lcd_clear();
    lcd_set_ddram_addr(LCD_LINE3_ADDR);
    printf("(A)Start");
    lcd_set_ddram_addr(LCD_LINE4_ADDR);
    printf("(B)Logs (C)Setup");
    unsigned char time[7]; // Create a byte array to hold time read from RTC
  LATCbits.LATC5 = 1; // Tell Nano run finished
    while (!PORTBbits.RB1){ // Prints time until key is pressed
        // Reset RTC memory pointer
        I2C_Master_Start(); // Start condition
        I2C_Master_Write(0b11010000); // 7 bit RTC address + Write
        I2C_Master_Write(0x00); // Set memory pointer to seconds
        I2C_Master_Stop(); // Stop condition

        // Read current time
        I2C_Master_Start(); // Start condition
        I2C_Master_Write(0b11010001); // 7 bit RTC address + Read
        for(unsigned char i = 0; i < 6; i++){
            time[i] = I2C_Master_Read(ACK); // Read with ACK to continue reading
        }
        time[6] = I2C_Master_Read(NACK); // Final Read with NACK
        I2C_Master_Stop(); // Stop condition
        lcd_set_ddram_addr(LCD_LINE1_ADDR);
        printf("20%02x-%02x-%02x", time[6],time[5],time[4]); // Print date in YYYY/MM/DD
        lcd_set_ddram_addr(LCD_LINE2_ADDR);
        printf("%02x:%02x:%02x", time[2],time[1],time[0]); // HH:MM:SS
    if (PORTAbits.RA1){ // Bluetooth start
```

```
            // Reset RTC memory pointer
            I2C_Master_Start(); // Start condition
            I2C_Master_Write(0b11010000); // 7 bit RTC address + Write
            I2C_Master_Write(0x00); // Set memory pointer to seconds
            I2C_Master_Stop(); // Stop condition

            // Read current time
            I2C_Master_Start(); // Start condition
            I2C_Master_Write(0b11010001); // 7 bit RTC address + Read
            for(unsigned char i = 0; i < 6; i++){
                startTime[i] = I2C_Master_Read(ACK); // Read with ACK to continue reading
            }
            startTime[6] = I2C_Master_Read(NACK); // Final Read with NACK
            I2C_Master_Stop(); // Stop condition

        *menu = 1;
        return;
    }
    }
    keypress = (PORTB & 0xF0) >> 4; // Stores keypad info into keypress
    switch (keypress){
        case 0b0011:    // (A)
            // Reset RTC memory pointer
            I2C_Master_Start(); // Start condition
            I2C_Master_Write(0b11010000); // 7 bit RTC address + Write
            I2C_Master_Write(0x00); // Set memory pointer to seconds
            I2C_Master_Stop(); // Stop condition

            // Read current time
            I2C_Master_Start(); // Start condition
            I2C_Master_Write(0b11010001); // 7 bit RTC address + Read
            for(unsigned char i = 0; i < 6; i++){
                startTime[i] = I2C_Master_Read(ACK); // Read with ACK to continue reading
            }
            startTime[6] = I2C_Master_Read(NACK); // Final Read with NACK
            I2C_Master_Stop(); // Stop condition

            *menu = 1;  // Start
            break;
        case 0b0111:    // (B)
            *menu = 6;  // Log menu
```

```
            break;
        case 0b1011:    //(C)
            *menu = 8;  // Options menu
            break;
    }
    while (PORTBbits.RB1){} // Waits until key is released
}

void topMagStepper(unsigned int steps, bool forward){
    unsigned int topMagCurrentStep = 0;

    if (forward){
        topMagCurrentStep = 0;
    } else {
        topMagCurrentStep = steps;
    }
    while ((forward && (topMagCurrentStep < steps)) || (!forward && (
   topMagCurrentStep > 0))){
        if (topMagCurrentStep % 4 == 0){
            LATBbits.LATB0 = 1;
            LATBbits.LATB2 = 1;
            LATBbits.LATB4 = 0;
            LATBbits.LATB6 = 0;
            if (forward){
                topMagCurrentStep++;
            } else {
                topMagCurrentStep--;
            }
        } else if (topMagCurrentStep % 4 == 1){
            LATBbits.LATB0 = 0;
            LATBbits.LATB2 = 1;
            LATBbits.LATB4 = 1;
            LATBbits.LATB6 = 0;
            if (forward){
                topMagCurrentStep++;
            } else {
                topMagCurrentStep--;
            }
        } else if (topMagCurrentStep % 4 == 2){
            LATBbits.LATB0 = 0;
            LATBbits.LATB2 = 0;
            LATBbits.LATB4 = 1;
            LATBbits.LATB6 = 1;
```

```
164             if (forward){
165                 topMagCurrentStep++;
166             } else {
167                 topMagCurrentStep--;
168             }
169         } else if (topMagCurrentStep % 4 == 3){
170             LATBbits.LATB0 = 1;
171             LATBbits.LATB2 = 0;
172             LATBbits.LATB4 = 0;
173             LATBbits.LATB6 = 1;
174             if (forward){
175                 topMagCurrentStep++;
176             } else {
177                 topMagCurrentStep--;
178             }
179         }
180         __delay_us(magazineStepperDelay);
181     }
182     LATBbits.LATB0 = 0;
183     LATBbits.LATB2 = 0;
184     LATBbits.LATB4 = 0;
185     LATBbits.LATB6 = 0;
186 }
187
188
189 void bottomMagStepper(unsigned int steps, bool forward){
190     unsigned int bottomMagCurrentStep = 0;
191
192     if (forward){
193         bottomMagCurrentStep = 0;
194     } else {
195         bottomMagCurrentStep = steps;
196     }
197     while ((forward && (bottomMagCurrentStep < steps)) || (!forward && (
    bottomMagCurrentStep > 0))){
198         if (bottomMagCurrentStep % 4 == 0){
199             LATBbits.LATB1 = 1;
200             LATBbits.LATB3 = 1;
201             LATBbits.LATB5 = 0;
202             LATBbits.LATB7 = 0;
203             if (forward){
204                 bottomMagCurrentStep++;
205             } else {
```

```
                bottomMagCurrentStep--;
            }
        } else if (bottomMagCurrentStep % 4 == 1){
            LATBbits.LATB1 = 0;
            LATBbits.LATB3 = 1;
            LATBbits.LATB5 = 1;
            LATBbits.LATB7 = 0;
            if (forward){
                bottomMagCurrentStep++;
            } else {
                bottomMagCurrentStep--;
            }
        } else if (bottomMagCurrentStep % 4 == 2){
            LATBbits.LATB1 = 0;
            LATBbits.LATB3 = 0;
            LATBbits.LATB5 = 1;
            LATBbits.LATB7 = 1;
            if (forward){
                bottomMagCurrentStep++;
            } else {
                bottomMagCurrentStep--;
            }
        } else if (bottomMagCurrentStep % 4 == 3){
            LATBbits.LATB1 = 1;
            LATBbits.LATB3 = 0;
            LATBbits.LATB5 = 0;
            LATBbits.LATB7 = 1;
            if (forward){
                bottomMagCurrentStep++;
            } else {
                bottomMagCurrentStep--;
            }
        }
        __delay_us(magazineStepperDelay);
    }
    LATBbits.LATB1 = 0;
    LATBbits.LATB3 = 0;
    LATBbits.LATB5 = 0;
    LATBbits.LATB7 = 0;
}

void start(unsigned char *menu, run *currentRun, unsigned char *topMagTires,
    unsigned char *bottomMagTires){
```

```
    unsigned int distTravelled = 0;
    float floatDistTravelled = 0;
    unsigned char tiresBefore = 0;
    unsigned char tiresNeeded = 0;
    unsigned char currentTires = 0;
    bool tireInCell = true;
    unsigned char currentPole = 0;
    bool armReached = true;
  bool bottomMagPushed = true;
  bool tireOnBottomRamp = true;

    currentRun->totalPoleNum = 0;
    LATCbits.LATC1 = 1;      // Disables keypad by setting KPD to HIGH
    TRISB = 0x00;
    LATB = 0x00;
  lcd_clear();
    // Command 1: Reset encoder and timer
    writeByteToDriveArduino(1);
  LATCbits.LATC5 = 0; // Tell Nano run began

    while (1){
        if ((*topMagTires <= 0) && (*bottomMagTires <= 0) && !tireInCell && !
    tireOnBottomRamp){
            *menu = 2;
            return;
        }

//          __delay_us(100);     // Allow Uno to drive runFinished HIGH
        if (PORTCbits.RC0){
            *menu = 2;  // Run finished
            return;
        }

        // Command 2: Normal speed forward
        writeByteToDriveArduino(2);
        __delay_ms(350);     // Slight delay so laser doesn't detect same pole
        // Nano starts polling laser
        LATAbits.LATA4 = 1;
        __delay_ms(5);
        LATAbits.LATA4 = 0;

        if (PORTCbits.RC0){
            LATCbits.LATC5 = 1; // Tell Nano run finished
```

```
            *menu = 2;  // Run finished
            return;
        }

        if (!tireInCell && !tireOnBottomRamp){
            if (*bottomMagTires > 0){
                bottomMagStepper(singleTireSteps, true);
                (*bottomMagTires)--;
        bottomMagPushed = true;
            } else if (*topMagTires > 0){
                topMagStepper(singleTireSteps, true);
                (*topMagTires)--;
        bottomMagPushed = false;
            }
    }

        if (PORTCbits.RC0){
            LATCbits.LATC5 = 1; // Tell Nano run finished
            *menu = 2;  // Run finished
            return;
        }

        // Wait until laser detects a pole
        while (!PORTAbits.RA5){
            if (PORTCbits.RC0){
                LATCbits.LATC5 = 1; // Tell Nano run finished
                *menu = 2;  // Run finished
                return;
            }
        }

    // Inputs run data
    (currentRun->totalPoleNum)++;
    currentPole = currentRun->totalPoleNum - 1;

    // Read encoder data
    I2C_Master_Start();
    I2C_Master_Write(0b00010001); // 7-bit Arduino slave address + Read
    distTravelled = ((I2C_Master_Read(ACK) << 8) & 0xff00) | (I2C_Master_Read(
   NACK) & 0xff); // Read distance travelled
    I2C_Master_Stop(); // Stop condition
    floatDistTravelled = distTravelled * 0.469 - 8.7;
    currentRun->poleArray[currentPole].dist = (unsigned int)(
```

```
floatDistTravelled*10);
 lcd_set_ddram_addr(LCD_LINE1_ADDR);
 printf("Distance: %hu.%hucm", currentRun->poleArray[currentPole].dist/10, currentRun->poleArray[currentPole].dist%10);

 currentRun->poleArray[currentPole].tiresAfter = 0;
 currentRun->poleArray[currentPole].tiresDeployed = 0;

     __delay_us(100);     // Allow Uno to drive runFinished HIGH
     if (PORTCbits.RC0){
         *menu = 2;  // Run finished
         return;
     }

 // Operate kicker
 if (!tireInCell){
   if (tireOnBottomRamp){
     LATEbits.LATE2 = 1;      // Writes RE2 HIGH (connected to D10 on the Nano)
     __delay_ms(10);          // Ensures the Nano reads the signal change
     LATEbits.LATE2 = 0;      // Writes RE2 LOW
     tireOnBottomRamp = false;
   } else if (bottomMagPushed){     // Bottom
     LATEbits.LATE2 = 1;
     __delay_ms(10);
     LATEbits.LATE2 = 0;
   } else {
     LATDbits.LATD1 = 1;
     __delay_ms(10);
     LATDbits.LATD1 = 0;
   }
   __delay_ms(2050);
   tireInCell = true;
 }

     LATAbits.LATA3 = 0; // Drive stopExtending low

     __delay_us(100);     // Allow Uno to drive runFinished HIGH
     if (PORTCbits.RC0){
         *menu = 2;  // Run finished
         return;
     }

```

```
        // Command 4: Extend arm
        writeByteToDriveArduino(4);
        __delay_ms(500);     // Doesn't detect laser sensor

        armReached = true;
        // While pole beam isn't broken
        while (PORTAbits.RA2){
            if (!PORTCbits.RC2){
                armReached = false;
                break;
            }
            if (PORTCbits.RC0){
                *menu = 2;  // Run finished
                return;
            }
        }
        LATAbits.LATA3 = 1; // Stop extending arm

        __delay_us(100);     // Allow Uno to drive runFinished HIGH
        if (PORTCbits.RC0){
            *menu = 2;  // Run finished
            return;
        }

        while (PORTCbits.RC2){} // Wait until arm fully extends

        if (armReached){
            // Checks top tire breakbeam
      if (!PORTEbits.RE1){
        tiresBefore = 2;         // Top beam broken, 2 tire on pole
      // Top beam intact, checks bottom tire breakbeam
      } else if (!PORTEbits.RE0){
        tiresBefore = 1;         // Top beam intact, bottom broken, 1 tire on
    pole
      // Both beams intact, no tire
      } else {
        tiresBefore = 0;
      }
      lcd_clear();
            lcd_set_ddram_addr(LCD_LINE2_ADDR);
            printf("Before: %u", tiresBefore);

      // If first pole
```

```
    if ((currentPole) == 0){
      tiresNeeded = 2;
    } else if ((currentRun->poleArray[currentPole].dist - currentRun->poleArray[currentPole - 1].dist) < 300){
      tiresNeeded = 1;
    } else if ((currentRun->poleArray[currentPole].dist - currentRun->poleArray[currentPole - 1].dist) >= 300){
      tiresNeeded = 2;
    }

          __delay_us(100);    // Allow Uno to drive runFinished HIGH
    if (PORTCbits.RC0){
      *menu = 2;  // Run finished
      return;
    }
          // Command 7: Position cell
          writeByteToDriveArduino(7);

          __delay_us(100);    // Allow Uno to drive unoBusy HIGH

          while (PORTCbits.RC2){} // Wait until cell properly positioned

    __delay_us(100);    // Allow Uno to drive runFinished HIGH
          if (PORTCbits.RC0){
      *menu = 2;  // Run finished
      return;
    }

          if (tiresBefore >= tiresNeeded){
              currentRun->poleArray[currentPole].tiresAfter = tiresBefore;
              currentRun->poleArray[currentPole].tiresDeployed = 0;
          } else {
              currentTires = tiresBefore;
              if ((tiresNeeded - tiresBefore) == 1){
                  // Operate cell
                  LATAbits.LATA0 = 1;
                  __delay_ms(10);
                  LATAbits.LATA0 = 0;
                  __delay_ms(1150);
                  currentTires++;
                  tireInCell = false;
              } else if ((tiresNeeded - tiresBefore) >= 2){
                  // Operate cell
```

```
                    LATAbits.LATA0 = 1;
                    __delay_ms(10);
                    LATAbits.LATA0 = 0;
                    currentTires++;
        tireInCell = false;
        if (PORTCbits.RC0){
          currentRun->poleArray[currentPole].tiresAfter = currentTires;
          currentRun->poleArray[currentPole].tiresDeployed = currentTires - tiresBefore;
                        __delay_ms(1150);
          *menu = 2;  // Run finished
                        return;
                    }
        if (tireOnBottomRamp){
                        __delay_ms(1250);
          LATEbits.LATE2 = 1;
          __delay_ms(10);
          LATEbits.LATE2 = 0;
          tireOnBottomRamp = false;
        } else if (*bottomMagTires > 0){
                        bottomMagStepper(singleTireSteps, true);
          // Operate whacker
          LATEbits.LATE2 = 1;
          (*bottomMagTires)--;
          __delay_ms(10);
          LATEbits.LATE2 = 0;
                    } else if (*topMagTires > 0){
                        topMagStepper(singleTireSteps, true);
          // Operate whacker
          LATDbits.LATD1 = 1;
          (*topMagTires)--;
          __delay_ms(10);
          LATDbits.LATD1 = 0;
                    } else {
                        currentRun->poleArray[currentPole].tiresAfter = currentTires;
                        currentRun->poleArray[currentPole].tiresDeployed = currentTires - tiresBefore;
                        __delay_ms(1150);    // Allows the Nano to operate cell and the Uno to drive unoBusy high
                        *menu = 2;
                        return;
                    }
```

```
494             __delay_ms(2050);
495                         // Operate cell
496                         LATAbits.LATA0 = 1;
497                         __delay_ms(10);
498                         LATAbits.LATA0 = 0;
499                         __delay_ms(1150);
500                         currentTires++;
501                     }
502                     currentRun->poleArray[currentPole].tiresAfter = currentTires;
503                     currentRun->poleArray[currentPole].tiresDeployed =
    currentTires - tiresBefore;
504                 }
505             }
506 
507             // Command 5: Retract arm
508             writeByteToDriveArduino(5);
509 
510         while (PORTCbits.RC2){} // Wait until arm fully retracts
511 
512         __delay_us(100);     // Allow Uno to drive runFinished HIGH
513             if (PORTCbits.RC0){
514                 *menu = 2;  // Run finished
515                 return;
516             }
517 
518             if (currentRun->totalPoleNum >= maxPoles){
519                 *menu = 2;  // Run finished
520                 return;
521             }
522         }
523 }
524 
525 // Converts binary-coded decimal values to decimal values
526 unsigned char bcdToDec(unsigned char val){
527   return( (val/16*10) + (val%16) );
528 }
529 
530 void runFinished(unsigned char *menu, run *currentRun, unsigned char *polePage
    , run *displayRun, bool *logs){
531     int startSec = 0;
532     int endSec = 0;
533   // Command 6: Return to start
534   writeByteToDriveArduino(6);
```

```
    lcd_clear();
    lcd_set_ddram_addr(LCD_LINE1_ADDR);
    printf("Finished");
    // Nano stops polling laser
    LATAbits.LATA4 = 0;
    LATCbits.LATC1 = 0;      // Enables keypad by setting KPD to LOW
    TRISB = 0xff;
    LATB = 0x00;
    currentRun->tiresSupplied = 0;
    for (i=0; i<currentRun->totalPoleNum; i++){
        currentRun->tiresSupplied = currentRun->tiresSupplied + currentRun->poleArray[i].tiresDeployed;
    }
  // Waits for Uno to return to start
    while (PORTCbits.RC2){}
    // Reset RTC memory pointer
    I2C_Master_Start(); // Start condition
    I2C_Master_Write(0b11010000); // 7 bit RTC address + Write
    I2C_Master_Write(0x00); // Set memory pointer to seconds
    I2C_Master_Stop(); // Stop condition

    // Read current time
    I2C_Master_Start(); // Start condition
    I2C_Master_Write(0b11010001); // 7 bit RTC address + Read
    for(unsigned char i = 0; i < 6; i++){
        endTime[i] = I2C_Master_Read(ACK); // Read with ACK to continue reading
    }
    endTime[6] = I2C_Master_Read(NACK); // Final Read with NACK
    I2C_Master_Stop(); // Stop condition

    startSec = bcdToDec(startTime[0]) + bcdToDec(startTime[1])*60 + bcdToDec(startTime[2])*3600;
    endSec = bcdToDec(endTime[0]) + bcdToDec(endTime[1])*60 + bcdToDec(endTime[2])*3600;
    currentRun->opTime = (unsigned char)(endSec - startSec);
    *polePage = 0;  // Sets first pole as default
    *logs = false;
    *displayRun = *currentRun;

    *menu = 7;  // Run finished menu
}

```

```
574 void runFinishedMenu(unsigned char *menu, bool *logs){
575     lcd_clear();
576     lcd_set_ddram_addr(LCD_LINE1_ADDR);
577     printf("Run completed!");
578     lcd_set_ddram_addr(LCD_LINE2_ADDR);
579     printf("(B) Run info");
580     lcd_set_ddram_addr(LCD_LINE3_ADDR);
581     printf("(C) Save run");
582     lcd_set_ddram_addr(LCD_LINE4_ADDR);
583     printf("(D) Main menu");
584 
585   // Wait until a key is pressed
586   while (!PORTBbits.RB1){}
587   // Bit-shift the data to determine which key was pressed
588   keypress = (PORTB & 0xF0) >> 4;
589   // If the key is recognized, set the menu variable
590   switch (keypress){
591     case 0b0111:    // (B)
592       *menu = 3;  // Run info menu
593       break;
594     case 0b1011:    // (C)
595       *menu = 5;  // Save run menu
596       break;
597     case 0b01111:   // (D)
598       *logs = true;
599       *menu = 0;  // Main menu
600       break;
601   }
602   // Wait until the key is let go
603   while (PORTBbits.RB1){}
604 }
605 
606 void runInfoMenu(unsigned char *menu, run *displayRun, bool logs){
607     lcd_clear();
608     lcd_set_ddram_addr(LCD_LINE1_ADDR);
609     printf("Op time:%u:", displayRun->opTime/60);
610     if (displayRun->opTime%60 < 10){
611         printf("0");
612     }
613     printf("%u", displayRun->opTime%60);
614     lcd_set_ddram_addr(LCD_LINE2_ADDR);
615     printf("# supplied:%u", displayRun->tiresSupplied);
616     lcd_set_ddram_addr(LCD_LINE3_ADDR);
```

```
    printf("# of poles:%u", displayRun->totalPoleNum);
    lcd_set_ddram_addr(LCD_LINE4_ADDR);
    if (displayRun->totalPoleNum != 0){
        printf("(C)Poles ");
    }
    printf("(D)Back");
    while (!PORTBbits.RB1){}
    keypress = (PORTB & 0xF0) >> 4;
    switch (keypress){
        case 0b1011:    // (C)
            if (displayRun->totalPoleNum != 0){
                *menu = 4;   // Pole Info
            }
            break;
        case 0b01111:   // (D)
            if (logs){
                *menu = 6;  // Select log menu
            } else {
                *menu = 7;   // Run finished menu
            }
            break;
    }
    while (PORTBbits.RB1){}
}

void poleInfo(unsigned char *menu, run *displayRun, unsigned char *polePage){
    lcd_clear();
    lcd_set_ddram_addr(LCD_LINE1_ADDR);
    printf("Pole #%u", *polePage + 1);
    lcd_set_ddram_addr(LCD_LINE2_ADDR);
    printf("Deploy:%u After:%u", displayRun->poleArray[*polePage].
   tiresDeployed, displayRun->poleArray[*polePage].tiresAfter);
    lcd_set_ddram_addr(LCD_LINE3_ADDR);
    printf("Distance:%hu.%hucm", displayRun->poleArray[*polePage].dist/10,
   displayRun->poleArray[*polePage].dist%10);
    lcd_set_ddram_addr(LCD_LINE4_ADDR);
    printf("<*  (D)Back   #>");
    while (!PORTBbits.RB1){}
    keypress = (PORTB & 0xF0) >> 4;
    switch (keypress){
        case 0b1111:    // (D)
            *menu = 3;   // Run Info
            break;
```

```
        case 0b01100:   // (*)
            if (*polePage == 0){
                *polePage = displayRun->totalPoleNum - 1;   // Pressing
    previous on first pole goes to last pole
            } else {
                *polePage = *polePage - 1;  // Previous Page
            }
            break;
        case 0b01110:   // (#)
            if (*polePage == displayRun->totalPoleNum - 1){ // Pressing next
    on last pole goes to first pole
                *polePage = 0;
            } else{
                *polePage = *polePage + 1;  // Next Page
            }
            break;
    }
    while (PORTBbits.RB1){}
}

void Write_eep( unsigned int badd,unsigned char bdat ){
  char GIE_BIT_VAL = 0;
  EEADRH = (badd >> 8) & 0x03;
  EEADR = (badd & 0x0ff);
  EEDATA = bdat;
    EECON1bits.EEPGD = 0;
  EECON1bits.CFGS = 0;
  EECON1bits.WREN = 1;
  GIE_BIT_VAL = INTCONbits.GIE;
  INTCONbits.GIE = 0;
  EECON2 = 0x55;
  EECON2 = 0xAA;
  EECON1bits.WR = 1;
  while(EECON1bits.WR);        //Wait till the write completion
  INTCONbits.GIE = GIE_BIT_VAL;
  EECON1bits.WREN = 0;
}

unsigned char Read_eep( unsigned int badd ){
  EEADRH = (badd >> 8) & 0x03;
  EEADR = (badd & 0x0ff);
    EECON1bits.CFGS = 0;
  EECON1bits.EEPGD = 0;
```

```
  EECON1bits.RD = 1;
  Nop();                //Nop may be required for latency at high frequencies
  Nop();                //Nop may be required for latency at high frequencies
  return ( EEDATA );               // return with read byte
}

void saveRun(unsigned char *menu, run *currentRun){
    lcd_clear();
    lcd_set_ddram_addr(LCD_LINE1_ADDR);
    printf("Choose log to");
    lcd_set_ddram_addr(LCD_LINE2_ADDR);
    printf("overwrite:");
    lcd_set_ddram_addr(LCD_LINE3_ADDR);
    printf("(1) (2) (3) (4)");
    lcd_set_ddram_addr(LCD_LINE4_ADDR);
    printf("(D) Back");
    while (!PORTBbits.RB1){}
    keypress = (PORTB & 0xF0) >> 4;
    switch (keypress){
        case 0b0000:    // (1)
            for (i=0;i<sizeof(run);i++){
                Write_eep(i, *((char *)currentRun + i));
            }
            *menu = 7;  // Run finished menu
            break;
        case 0b0001:    // (2)
            for (i=0;i<sizeof(run);i++){
                Write_eep(sizeof(run) + i, *((char *)currentRun + i));
            }
            *menu = 7;  // Run finished menu
            break;
        case 0b0010:    // (3)
            for (i=0;i<sizeof(run);i++){
                Write_eep(2*sizeof(run) + i, *((char *)currentRun + i));
            }
            *menu = 7;  // Run finished menu
            break;
        case 0b0100:    // (4)
            for (i=0;i<sizeof(run);i++){
                Write_eep(3*sizeof(run) + i, *((char *)currentRun + i));
            }
            *menu = 7;  // Run finished menu
            break;
```

```
742         case 0b01111:   // (D)
743             *menu = 2;  // Run finished menu
744             break;
745     }
746     while (PORTBbits.RB1){}
747 }
748
749 void selectLog(unsigned char *menu, run *displayRun){
750     lcd_clear();
751     lcd_set_ddram_addr(LCD_LINE1_ADDR);
752     printf("Choose log to");
753     lcd_set_ddram_addr(LCD_LINE2_ADDR);
754     printf("view:");
755     lcd_set_ddram_addr(LCD_LINE3_ADDR);
756     printf("(1) (2) (3) (4)");
757     lcd_set_ddram_addr(LCD_LINE4_ADDR);
758     printf("(D) Main menu");
759     while (!PORTBbits.RB1){}
760     keypress = (PORTB & 0xF0) >> 4;
761     switch (keypress){
762         case 0b0000:    // (1)
763             for (i=0;i<sizeof(run);i++){
764                 *((char *)displayRun + i) = Read_eep(i);
765             }
766             *menu = 3;  // Run info menu
767             break;
768         case 0b0001:    // (2)
769             for (i=0;i<sizeof(run);i++){
770                 *((char *)displayRun + i) = Read_eep(sizeof(run) + i);
771             }
772             *menu = 3;  // Run info menu
773             break;
774         case 0b0010:    // (3)
775             for (i=0;i<sizeof(run);i++){
776                 *((char *)displayRun + i) = Read_eep(2*sizeof(run) + i);
777             }
778             *menu = 3;  // Run info menu
779             break;
780         case 0b0100:    // (4)
781             for (i=0;i<sizeof(run);i++){
782                 *((char *)displayRun + i) = Read_eep(3*sizeof(run) + i);
783             }
784             *menu = 3;  // Run info menu
```

```
            break;
        case 0b01111:   // (D)
            *menu = 0;  // Main menu
            break;
    }
    while (PORTBbits.RB1){}
}

void options(unsigned char *menu, unsigned char *topMagTires, unsigned char *
    bottomMagTires){
    unsigned char tens = 0;
    unsigned char ones = 0;
    unsigned char topMagTireMove;
    char topMagTireCheck;
    unsigned char bottomMagTireMove;
    char bottomMagTireCheck;
    bool forward = false;

    // Magazine 1
    lcd_clear();
    lcd_set_ddram_addr(LCD_LINE1_ADDR);
    printf("Top mag forward");
    lcd_set_ddram_addr(LCD_LINE2_ADDR);
    printf("or backward?");
    lcd_set_ddram_addr(LCD_LINE3_ADDR);
    printf("(#) Forward");
    lcd_set_ddram_addr(LCD_LINE4_ADDR);
    printf("(*) Backward");
    while (!PORTBbits.RB1){}
    keypress = (PORTB & 0xF0) >> 4;
    switch (keypress){
        case 0b1110:    // (#)
            forward = true;
            break;
        case 0b1100:    // (*)
            forward = false;
            break;
    }
    while (PORTBbits.RB1){}
    lcd_clear();
    lcd_set_ddram_addr(LCD_LINE1_ADDR);
    printf("Input tire");
    lcd_set_ddram_addr(LCD_LINE2_ADDR);
```

```
    printf("spaces to move");
    lcd_set_ddram_addr(LCD_LINE3_ADDR);
    printf("top mag (01-13):");
    while (!PORTBbits.RB1){}
    keypress = (PORTB & 0xF0) >> 4;
    switch (keypress){
        case 0b1101:    // (0)
            tens = 0;
            break;
        case 0b0000:    // (1)
            tens = 1;
            break;
    }
    while (PORTBbits.RB1){}
    lcd_set_ddram_addr(LCD_LINE4_ADDR);
    printf("%u", tens);
    while (!PORTBbits.RB1){}
    keypress = (PORTB & 0xF0) >> 4;
    switch (keypress){
        case 0b1101:    // (0)
            ones = 0;
            break;
        case 0b0000:    // (1)
            ones = 1;
            break;
        case 0b0001:    // (2)
            ones = 2;
            break;
        case 0b0010:    // (3)
            ones = 3;
            break;
        case 0b0100:    // (4)
            ones = 4;
            break;
        case 0b0101:    // (5)
            ones = 5;
            break;
        case 0b0110:    // (6)
            ones = 6;
            break;
        case 0b1000:    // (7)
            ones = 7;
            break;
```

```
        case 0b1001:    // (8)
            ones = 8;
            break;
        case 0b1010:    // (9)
            ones = 9;
            break;
    }
    while (PORTBbits.RB1){}
    topMagTireMove = tens*10 + ones;
    if (forward){
        topMagTireCheck = (*topMagTires) - topMagTireMove;
    } else {
        topMagTireCheck = (*topMagTires) + topMagTireMove;
    }
//    if (topMagTireCheck >= 0 && topMagTireCheck <= 12){
    if (true){
        *topMagTires = topMagTireCheck;
        lcd_clear();
        lcd_set_ddram_addr(LCD_LINE1_ADDR);
        printf("%u tire(s) can", *topMagTires);
        lcd_set_ddram_addr(LCD_LINE2_ADDR);
        printf("be loaded in");
        lcd_set_ddram_addr(LCD_LINE3_ADDR);
        printf("top mag");
        LATCbits.LATC1 = 1;     // Disables keypad by setting KPD to HIGH
        TRISB = 0x00;
        LATB = 0x00;
        topMagStepper(singleTireSteps*topMagTireMove, forward);
        LATCbits.LATC1 = 0;     // Enables keypad by setting KPD to LOW
        TRISB = 0xff;
        LATB = 0x00;
        if (topMagTireMove == 0){
            __delay_ms(1000);
        }
    } else {
        lcd_clear();
        lcd_set_ddram_addr(LCD_LINE1_ADDR);
        printf("Magazine can't");
        lcd_set_ddram_addr(LCD_LINE2_ADDR);
        printf("hold %u tires", topMagTireCheck);
        __delay_ms(1000);
    }

```

```
    // Magazine 2
    lcd_clear();
    lcd_set_ddram_addr(LCD_LINE1_ADDR);
    printf("Bot mag forward");
    lcd_set_ddram_addr(LCD_LINE2_ADDR);
    printf("or backward?");
    lcd_set_ddram_addr(LCD_LINE3_ADDR);
    printf("(#) Forward");
    lcd_set_ddram_addr(LCD_LINE4_ADDR);
    printf("(*) Backward");
    while (!PORTBbits.RB1){}
    keypress = (PORTB & 0xF0) >> 4;
    switch (keypress){
        case 0b1110:    // (#)
            forward = true;
            break;
        case 0b1100:    // (*)
            forward = false;
            break;
    }
    while (PORTBbits.RB1){}
    lcd_clear();
    lcd_set_ddram_addr(LCD_LINE1_ADDR);
    printf("Input tire");
    lcd_set_ddram_addr(LCD_LINE2_ADDR);
    printf("spaces to move");
    lcd_set_ddram_addr(LCD_LINE3_ADDR);
    printf("bot mag (01-13):");
    while (!PORTBbits.RB1){}
    keypress = (PORTB & 0xF0) >> 4;
    switch (keypress){
        case 0b1101:    // (0)
            tens = 0;
            break;
        case 0b0000:    // (1)
            tens = 1;
            break;
    }
    while (PORTBbits.RB1){}
    lcd_set_ddram_addr(LCD_LINE4_ADDR);
    printf("%u", tens);
    while (!PORTBbits.RB1){}
    keypress = (PORTB & 0xF0) >> 4;
```

```
    switch (keypress){
        case 0b1101:    // (0)
            ones = 0;
            break;
        case 0b0000:    // (1)
            ones = 1;
            break;
        case 0b0001:    // (2)
            ones = 2;
            break;
        case 0b0010:    // (3)
            ones = 3;
            break;
        case 0b0100:    // (4)
            ones = 4;
            break;
        case 0b0101:    // (5)
            ones = 5;
            break;
        case 0b0110:    // (6)
            ones = 6;
            break;
        case 0b1000:    // (7)
            ones = 7;
            break;
        case 0b1001:    // (8)
            ones = 8;
            break;
        case 0b1010:    // (9)
            ones = 9;
            break;
    }
    while (PORTBbits.RB1){}
    bottomMagTireMove = tens*10 + ones;
    if (forward){
        bottomMagTireCheck = (*bottomMagTires) - bottomMagTireMove;
    } else {
        bottomMagTireCheck = (*bottomMagTires) + bottomMagTireMove;
    }
//    if (bottomMagTireCheck >= 0 && bottomMagTireCheck <= 12){
    if (true){
        *bottomMagTires = bottomMagTireCheck;
        lcd_clear();
```

```
        lcd_set_ddram_addr(LCD_LINE1_ADDR);
        printf("%u tire(s) can", *bottomMagTires);
        lcd_set_ddram_addr(LCD_LINE2_ADDR);
        printf("be loaded in");
        lcd_set_ddram_addr(LCD_LINE3_ADDR);
        printf("bot mag");
        LATCbits.LATC1 = 1;      // Disables keypad by setting KPD to HIGH
        TRISB = 0x00;
        LATB = 0x00;
        bottomMagStepper((singleTireSteps)*bottomMagTireMove, forward);
        LATCbits.LATC1 = 0;      // Enables keypad by setting KPD to LOW
        TRISB = 0xff;
        LATB = 0x00;
        if (bottomMagTireMove == 0){
            __delay_ms(1000);
        }
    } else {
        lcd_clear();
        lcd_set_ddram_addr(LCD_LINE1_ADDR);
        printf("Magazine can't");
        lcd_set_ddram_addr(LCD_LINE2_ADDR);
        printf("hold %u tires", *bottomMagTires);
        __delay_ms(1000);
    }

    *menu = 0;
}

/** @brief Writes the inputTime array to the RTC memory */
void setTime(void){
    I2C_Master_Start(); // Start condition
    I2C_Master_Write(0b11010000); //7 bit RTC address + Write
    I2C_Master_Write(0x00); // Set memory pointer to seconds

    // Write array
    for(char i=0; i < 7; i++){
        I2C_Master_Write(inputTime[i]);
    }

    I2C_Master_Stop(); //Stop condition
}

void main(void){
```

```
    run currentRun;
    run displayRun;
    unsigned char menu = 0;
    unsigned char polePage = 0; // Which pole is displayed in log
    bool logs = true;
    unsigned char topMagTires = 0;
    unsigned char bottomMagTires = 0;

    // RD2 is the character LCD RS
    // RD3 is the character LCD enable (E)
    // RD4–RD7 are character LCD data lines
    LATA = 0x00;
    TRISA = 0x00;
    LATB = 0x00;
    TRISB = 0xff;
    LATD = 0x00;
    TRISD = 0x00;
    LATC = 0x00;
    TRISC = 0x00;
    LATE = 0x00;
    TRISE = 0x00;

    TRISCbits.RC2 = 1;  // Uno busy
    TRISAbits.RA5 = 1;  // Stop motors
    TRISCbits.RC0 = 1;  // Uno run finished
    TRISAbits.RA2 = 1;  // Pole break beam
    TRISEbits.RE0 = 1;  // Tire 1 break beam
    TRISEbits.RE1 = 1;  // Tire 2 break beam

    // Set all A/D ports to digital (pg. 222)
    ADCON1 = 0b00001111;

    // Initialize LCD
    initLCD();

    // Initialize I2C Master with 100 kHz clock
    I2C_Master_Init(100000);

    // Set the time in the RTC. To see the RTC keep time, comment this line
   out
    // after programming the PIC directly before with this line included
//    setTime();

```

```
1084    lcd_display_control(true, false, false);
1085
1086
1087    while(1){
1088    switch (menu){
1089      // Main menu
1090      case 0:
1091        mainMenu(&menu);
1092        break;
1093      // Start operation function
1094      case 1:
1095        start(&menu, &currentRun, &topMagTires, &bottomMagTires);
1096        break;
1097      // Run finished function
1098      case 2:
1099        runFinished(&menu, &currentRun, &polePage, &displayRun, &logs);
1100        break;
1101      // Run info menu
1102      case 3:
1103        runInfoMenu(&menu, &displayRun, logs);
1104        break;
1105      // Pole info page
1106      // polePage determines which pole's info is shown
1107      case 4:
1108        poleInfo(&menu, &displayRun, &polePage);
1109        break;
1110      // Choose which EEPROM log to overwrite with new data
1111      case 5:
1112        saveRun(&menu, &currentRun);
1113        break;
1114      // Choose which EEPROM log to view
1115      case 6:
1116        selectLog(&menu, &displayRun);
1117        break;
1118      // Termination message/run info menu
1119      case 7:
1120        runFinishedMenu(&menu, &logs);
1121        break;
1122      // Load tires into magazines
1123      case 8:
1124        options(&menu, &topMagTires, &bottomMagTires);
1125        break;
1126    }
```

```
  }
}
```

A.2 Control Uno Main Program

```
#include <Wire.h>
#include <elapsedMillis.h>
#include <Stepper.h>

#define encoderPin 2
#define runFinished 4
#define startDrive 5
#define returnToStartCommand 6
#define startRun 7
//#define enL 5
//#define enR 6
//#define in1 7
//#define in2 8
//#define in3 9
//#define in4 10
#define unoBusy 11
#define stopExtending 12
#define armStepperIn1 14
#define armStepperIn2 15
#define armStepperIn3 16
#define armStepperIn4 17

elapsedMillis timer;
unsigned int poleDuration = 0;
float poleDist = 0;
volatile unsigned int distTravelled = 0;
volatile boolean isRunning = false;

volatile unsigned char command = 0;      // Command from PIC
volatile bool newCommand = false;        // Used to check if command has
    changed

volatile unsigned long encoderPos = 0;  // a counter for the dial
volatile boolean rotating = false;      // debounce management
volatile boolean set = false;

const int stepsPerRevolution = 500;
volatile int armStepperCurrentPosition = 0;
volatile int armStepperNewPosition = 0;
```

```
Stepper armStepper(stepsPerRevolution, armStepperIn1, armStepperIn2,
    armStepperIn3, armStepperIn4);

unsigned long i = 0;

void setup() {
  pinMode(runFinished, OUTPUT);
  digitalWrite(runFinished, LOW);

  Wire.begin(8); // Join I2C bus with address 8
  // Register callback functions
  Wire.onReceive(receiveEvent); // Called when this slave device receives a
    data transmission from master
  Wire.onRequest(requestEvent); // Called when master requests data from this
    slave device

  pinMode(encoderPin, INPUT);
  // encoder pin on interrupt 0 (pin 2)
  attachInterrupt(digitalPinToInterrupt(encoderPin), doEncoder, CHANGE);

  pinMode(startDrive, OUTPUT);
  digitalWrite(startDrive, LOW);

  pinMode(returnToStartCommand, OUTPUT);
  digitalWrite(returnToStartCommand, LOW);

  pinMode(startRun, OUTPUT);
  digitalWrite(startRun, LOW);

  pinMode(unoBusy, OUTPUT);
  digitalWrite(unoBusy, LOW);

  pinMode(stopExtending, INPUT);

  armStepper.setSpeed(60);
  pinMode(armStepperIn1, OUTPUT);
  pinMode(armStepperIn2, OUTPUT);
  pinMode(armStepperIn3, OUTPUT);
  pinMode(armStepperIn4, OUTPUT);
  digitalWrite(armStepperIn1, LOW);
  digitalWrite(armStepperIn2, LOW);
  digitalWrite(armStepperIn3, LOW);
  digitalWrite(armStepperIn4, LOW);
```

```
  Serial.begin(9600);
}

void returnToStart(){
//  Serial.println("returning");
  digitalWrite(runFinished, HIGH);
  isRunning = false;
  distTravelled = encoderPos;
  encoderPos = 1;
  armStepper.step(-armStepperCurrentPosition-20);
  armStepperCurrentPosition = 0;
  digitalWrite(armStepperIn1, LOW);
  digitalWrite(armStepperIn2, LOW);
  digitalWrite(armStepperIn3, LOW);
  digitalWrite(armStepperIn4, LOW);
  digitalWrite(returnToStartCommand, HIGH);
//  Serial.println("start driven high");
//  Serial.println("encoder");
//  Serial.println(distTravelled);
//  Serial.println(encoderPos);
  while ((encoderPos < distTravelled + 140) && (timer < 175000)){
    rotating = true;
//    Serial.println(encoderPos);
  }
  digitalWrite(returnToStartCommand, LOW);
  digitalWrite(runFinished, LOW);
}

void loop() {
//  Serial.println(timer);
  rotating = true;
//  Serial.println(encoderPos);
  // Reached 4 metres or 2:30 has elapsed
  if (isRunning && (((encoderPos) > 865) || (timer > 153000))){
//    Serial.println("2:35");
    digitalWrite(runFinished, HIGH);
  }

  // Only checks command if newCommand is true
  if (newCommand){
    // Command 1: New run
    if (command == 1){}
```

```
    encoderPos = 1;                    // Resets encoder
    timer = 0;                         // Resets timer
    isRunning = true;                  // Operation started
    armStepperCurrentPosition = 0;     // Resets arm stepper position
    // Tells the Drive Uno to get ready for a run
    digitalWrite(startRun, HIGH);
    delay(50);
    digitalWrite(startRun, LOW);
// Commands besides 1 are only checked if the robot is running
} else if (isRunning){
  switch(command){
    // Command 2: Normal speed forward
    case 2:
      // Tells the Drive Uno to start running
      digitalWrite(startDrive, HIGH);
      delay(50);
      digitalWrite(startDrive, LOW);
      break;
    // Command 4: Extend arm
    case 4:
      // Extend until PIC drives stopExtending HIGH
      while (!digitalRead(stopExtending)){
        // Step forward 1 step
        armStepper.step(1);
        // Increment stepper position counter
        armStepperCurrentPosition++;
        // If arm is near max extension, stop extending
        if (armStepperCurrentPosition >= 1750){
          break;
        }
      }
      // Extend 180 steps further to detect tires
      armStepper.step(180);
      // Add 180 to stepper position counter
      armStepperCurrentPosition += 180;
      // All coils are unpowered so the gear may be rotated by hand
      digitalWrite(armStepperIn1, LOW);
      digitalWrite(armStepperIn2, LOW);
      digitalWrite(armStepperIn3, LOW);
      digitalWrite(armStepperIn4, LOW);
      break;

    // Retract arm
```

```
      case 5:
//      Serial.print("Stepping backward ");
//      Serial.println(armStepperCurrentPosition);
        armStepper.step(-armStepperCurrentPosition-20);
        armStepperCurrentPosition = 0;
        digitalWrite(armStepperIn1, LOW);
        digitalWrite(armStepperIn2, LOW);
        digitalWrite(armStepperIn3, LOW);
        digitalWrite(armStepperIn4, LOW);
        break;

      // Reached max poles or out of tires
      case 6:
//      Serial.println("returning");
        returnToStart();
        break;

      // Position cell
      case 7:
        if (armStepperCurrentPosition >= 950){
          armStepper.step(-950);
          armStepperCurrentPosition -= 950;
        } else {
          armStepper.step(-armStepperCurrentPosition-20);
          armStepperCurrentPosition = 0;
        }
        digitalWrite(armStepperIn1, LOW);
        digitalWrite(armStepperIn2, LOW);
        digitalWrite(armStepperIn3, LOW);
        digitalWrite(armStepperIn4, LOW);
//        Serial.print("Position cell: ");
//        Serial.println(armStepperCurrentPosition);
        break;
      }
    }
        newCommand = false;          // No command to process
        digitalWrite(unoBusy, LOW); // Control Uno is no longer busy
  }
}

// Interrupt on encoder changing state
void doEncoder() {
  // debounce
```

```
  if ( rotating ) delay (1);  // wait a little until the bouncing is done

  // Test transition, did things really change?
  if ( digitalRead(encoderPin) != set ) { // debounce once more
  set = !set;

  encoderPos += 1;
//  Serial.println(encoderPos);

  rotating = false;  // no more debouncing until loop() hits again
  }
}

/** @brief Callback for when the master transmits data */
void receiveEvent(void){
  digitalWrite(unoBusy, HIGH);  // Writes unoBusy high so the PIC will wait to
    send the next command
  newCommand = true;            // The Control Uno will only check what
   command was given if newCommand is true
  command = Wire.read();        // Receive byte
}

/** @brief Callback for when the master requests data */
void requestEvent(void){
  if (isRunning){
    if (encoderPos == 1){
      Wire.write(0);
      Wire.write(0);
    } else {
      Wire.write(encoderPos>>8);
      Wire.write(encoderPos);
    }
  } else {
//    Serial.print("timer: ");
//    Serial.println(timer);
    Wire.write(timer/1000);
  }
}
```

A.3 Nano Main Program

```
#include "Adafruit_VL53L0X.h"
#include <Servo.h>

```

```
#define startKickerBottom 10
#define startKickerTop 3
#define kickerPin 9
#define runFinished 11
#define cellPin 12
#define gripperPin 13
#define startServos 14
#define bluetoothStart 15
#define startPollingLaser 16
#define stopMotors 17

#define gripperNeutral 138
#define cellNeutral 146
#define kickerNeutral 127

Adafruit_VL53L0X poleSensor = Adafruit_VL53L0X();

VL53L0X_RangingMeasurementData_t poleMeasure;

volatile unsigned int poleDist = 10000;

Servo gripper;
Servo cell;
Servo kicker;

volatile unsigned int i = 0;
volatile char startByte = 0;

void setup() {
  pinMode(stopMotors, OUTPUT);
  digitalWrite(stopMotors, LOW);

  Serial.begin(9600);

  // wait until serial port opens for native USB devices
  while (! Serial) {
    delay(1);
  }

  Serial.println("Adafruit VL53L0X test");
  if (!poleSensor.begin()) {
    Serial.println(F("Failed to boot VL53L0X"));
    while(1);
```

```
47    }
48
49    pinMode(startKickerBottom, INPUT);
50
51    pinMode(startKickerTop, INPUT);
52
53    pinMode(startServos, INPUT);
54
55    pinMode(startPollingLaser, INPUT);
56
57    pinMode(bluetoothStart, OUTPUT);
58    digitalWrite(bluetoothStart, LOW);
59
60    pinMode(runFinished, INPUT);
61
62    kicker.attach(kickerPin);
63    kicker.write(kickerNeutral);
64
65    gripper.attach(gripperPin);
66    gripper.write(gripperNeutral);
67
68    cell.attach(cellPin);
69    cell.write(cellNeutral);
70  }
71
72  void loop() {
73    if (digitalRead(runFinished) && Serial.available() > 0){
74      startByte = Serial.read();
75      if (startByte == '1'){
76        digitalWrite(bluetoothStart, HIGH);
77        startByte = 0;
78        delay(50);
79        digitalWrite(bluetoothStart, LOW);
80      }
81    }
82
83    if (digitalRead(startPollingLaser)){
84      digitalWrite(stopMotors, LOW);
85      poleDist = 10000;
86      while (poleDist > 350 && !digitalRead(runFinished)){
87        poleSensor.rangingTest(&poleMeasure, false);
88        poleDist = poleMeasure.RangeMilliMeter;
89  //        Serial.println(poleDist);
```

```
    // Check if laser sensor reads less than 350
    if (poleDist <= 350){
      poleSensor.rangingTest(&poleMeasure, false);
      poleDist = poleMeasure.RangeMilliMeter;
      // If the reading is less than 350, poll the sensor again and only stop
    the robot if both readings are less than 350
      if (poleDist <= 350){
        delay(13);
        digitalWrite(stopMotors, HIGH);
      }
    }
  }

  // Operate cell
  if (digitalRead(startServos)){
    cell.write(cellNeutral+20);
    delay(350);
    gripper.write(gripperNeutral+30);
    delay(100);
    cell.write(cellNeutral-90);
    delay(600);
    gripper.write(gripperNeutral);
    delay(100);
    cell.write(cellNeutral);
  }

  if (digitalRead(startKickerTop)){
    for (i=kickerNeutral;i>59;i--){
      kicker.write(i);
      delay(8);
    }
    kicker.write(kickerNeutral);
  }

  if (digitalRead(startKickerBottom)){
    for (i=kickerNeutral;i>59;i--){
      kicker.write(i);
      delay(8);
    }
    kicker.write(kickerNeutral);
  }
}
```

A.4 Drive Uno Main Program

```
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"
#include <Wire.h>
#include "pitches.h"

#define returnToStart 2
#define nanoStop 3
#define startRun 4
#define enL 5
#define enR 6
#define in1 7
#define in2 8
#define in3 9
#define in4 10
#define startDrive 11
#define poleLED 13
#define speakerOn 14

#define sixteenth 90

MPU6050 mpu;
// MPU control/status vars
bool dmpReady = false;  // set true if DMP init was successful
uint8_t mpuIntStatus;   // holds actual interrupt status byte from MPU
uint8_t devStatus;      // return status after each device operation (0 =
    success, !0 = error)
uint16_t packetSize;    // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount;     // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars
Quaternion q;           // [w, x, y, z]         quaternion container
VectorFloat gravity;    // [x, y, z]            gravity vector
float ypr[3];           // [yaw, pitch, roll]   yaw/pitch/roll container and
    gravity vector


volatile unsigned char leftPWM = 63;
volatile unsigned char rightPWM = 53;
volatile bool driving = false;
volatile bool runFinished = false;
volatile float factor = 0;
```

```
float yawRef = 0.0;
volatile bool sound = false;

void setup() {
  pinMode(poleLED, OUTPUT);
  digitalWrite(poleLED, LOW);

  pinMode(speakerOn, OUTPUT);
  digitalWrite(speakerOn, LOW);

  pinMode(returnToStart, INPUT);
  // runFinished pin on interrupt 1 (pin 2)
  attachInterrupt(digitalPinToInterrupt(returnToStart), doReturnToStart,
    RISING);

  pinMode(nanoStop, INPUT);
  // nano pin on interrupt 1 (pin 3)
  attachInterrupt(digitalPinToInterrupt(nanoStop), doNanoStop, RISING);

  pinMode(startRun, INPUT);
  pinMode(startDrive, INPUT);

  pinMode(enL, OUTPUT);
  pinMode(enR, OUTPUT);
  pinMode(in1, OUTPUT);
  pinMode(in2, OUTPUT);
  pinMode(in3, OUTPUT);
  pinMode(in4, OUTPUT);
  digitalWrite(enL, LOW);
  digitalWrite(enR, LOW);
  digitalWrite(in1, LOW);
  digitalWrite(in2, LOW);
  digitalWrite(in3, LOW);
  digitalWrite(in4, LOW);

  // join I2C bus (I2Cdev library doesn't do this automatically)
  #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
    Wire.begin();
    Wire.setClock(400000); // 400kHz I2C clock. Comment this line if having
    compilation difficulties
  #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
    Fastwire::setup(400, true);
  #endif
```

```
Serial.begin(115200);
while (!Serial); // wait for Leonardo enumeration, others continue immediately

// initialize device
Serial.println(F("Initializing I2C devices..."));
mpu.initialize();

// verify connection
Serial.println(F("Testing device connections..."));
Serial.println(mpu.testConnection() ? F("MPU6050 connection successful") : F("MPU6050 connection failed"));

// ready when buffer empty
while (Serial.available() && Serial.read()); // empty buffer

// load and configure the DMP
Serial.println(F("Initializing DMP..."));
devStatus = mpu.dmpInitialize();
Serial.println("hi");

// supply your own gyro offsets here, scaled for min sensitivity
mpu.setXGyroOffset(220);
mpu.setYGyroOffset(76);
mpu.setZGyroOffset(-85);
mpu.setZAccelOffset(1788); // 1688 factory default for my test chip

// make sure it worked (returns 0 if so)
if (devStatus == 0) {
  // turn on the DMP, now that it's ready
  Serial.println(F("Enabling DMP..."));
  mpu.setDMPEnabled(true);

  // set our DMP Ready flag so the main loop() function knows it's okay to use it
  Serial.println(F("DMP ready!"));
  dmpReady = true;

  // get expected DMP packet size for later comparison
  packetSize = mpu.dmpGetFIFOPacketSize();
} else {
  // ERROR!
```

```
    // 1 = initial memory load failed
    // 2 = DMP configuration updates failed
    // (if it's going to break, usually the code will be 1)
    Serial.print(F("DMP Initialization failed (code "));
    Serial.print(devStatus);
    Serial.println(F(")"));
  }
}

void loop() {
  factor = 0.0;
  leftPWM = 63;
  rightPWM = 53;

  // Reset the FIFO buffer
  mpu.resetFIFO();

  // get current FIFO count
  fifoCount = mpu.getFIFOCount();

  // wait for correct available data length, should be a VERY short wait
  while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

  // read a packet from FIFO
  mpu.getFIFOBytes(fifoBuffer, packetSize);

  // track FIFO count here in case there is > 1 packet available
  // (this lets us immediately read more without waiting for an interrupt)
  fifoCount -= packetSize;

  // display Euler angles in degrees
  mpu.dmpGetQuaternion(&q, fifoBuffer);
  mpu.dmpGetGravity(&gravity, &q);
  mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
  Serial.print("ypr\t");
  Serial.println(ypr[0] * 180/M_PI);
//  Serial.print("\t");
//  Serial.print(ypr[1] * 180/M_PI);
//  Serial.print("\t");
//  Serial.println(ypr[2] * 180/M_PI);

  if (runFinished){
    digitalWrite(poleLED, LOW);
```

```
    digitalWrite(in1, HIGH);
    digitalWrite(in3, HIGH);
    digitalWrite(in2, LOW);
    digitalWrite(in4, LOW);
    analogWrite(enL, 70);
    analogWrite(enR, 85);
    delay(1000);
    analogWrite(enL, 160);
    analogWrite(enR, 170);
    delay(1000);
    analogWrite(enL, 250);
    analogWrite(enR, 255);
    while (digitalRead(returnToStart)){
//      Serial.println("stuck");
    }
    digitalWrite(in2, HIGH);
    digitalWrite(in4, HIGH);
    runFinished = false;
    driving = false;
  }
  if (digitalRead(startRun)){
    yawRef = (ypr[0] * 180/M_PI);
  }
  if (digitalRead(startDrive)){
    digitalWrite(poleLED, LOW);
    driving = true;
    factor = (ypr[0] * 180/M_PI) - yawRef;
    factor = calc(factor);
//    Serial.print("factor:");Serial.println(factor);
    analogWrite(enL, leftPWM + factor);
    analogWrite(enR, rightPWM - factor);
    digitalWrite(in1, LOW);
    digitalWrite(in3, LOW);
    digitalWrite(in4, HIGH);
    digitalWrite(in2, HIGH);
  }
  if (driving){
    factor = (ypr[0] * 180/M_PI) - yawRef;
    factor = calc(factor);
//    Serial.print("factor:");Serial.println(factor);
    analogWrite(enL, leftPWM + factor);
    analogWrite(enR, rightPWM - factor);
//    Serial.print("left:");Serial.println(leftPWM + factor);
```

```
//      Serial.print("right:");Serial.println(rightPWM - factor);
  }
  if (sound){
    digitalWrite(speakerOn, HIGH);
    tone(15, NOTE_G5, 250*1.3);
    delay(250*1.3);
    noTone(15);
    tone(15, NOTE_E5, 250*1.3);
    delay(250*1.3);
    noTone(15);
    tone(15, NOTE_C5, 250*1.3);
    delay(250*1.3);
    noTone(15);
    digitalWrite(speakerOn, LOW);
    sound = false;
  }
}

void doReturnToStart(){
  runFinished = true;
}

void doNanoStop(){
//  Serial.print("Nanostop");
  digitalWrite(in1, HIGH);
  digitalWrite(in3, HIGH);
  analogWrite(enL, 255);
  analogWrite(enR, 255);
  digitalWrite(in2, HIGH);
  digitalWrite(in4, HIGH);
  digitalWrite(poleLED, HIGH);
  driving = false;
  sound = true;
}

float calc(float d){
  unsigned char amp = 8;
  unsigned char var = 2;
  unsigned char maxVar = 10;
  if (d < (-360.0+maxVar)){
    if (d < (-360.0+var)){
      //return 0;
      return amp*sqrt((d+360)/var);
```

```
    }
    return amp;
    //return map(d,-360.0,-360.0+maxVar,0.0,-amp);
  } else if (d > 360.0-maxVar){
    if (d > (360.0-var)){
      //return 0;
      return -amp*sqrt(-(d-360.0)/var);
    }
    return -amp;
    //return map(d,360.0-maxVar,360.0,amp,0.0);
  } else  if (d > -maxVar&& d < maxVar) {
    if (d > -var && d < var){
      if (d > 0){
        //return 0;
        return -amp*sqrt(d/var);
      } else if (d < 0){
        //return 0;
        return amp*sqrt(-d/var);
      }
    }
    if (d > var){
      return -amp;
    } else if (d < -var){
      return amp;
    }
  } else {
    return 0;
  }
  return 0;
}
\subsection{configBits.h}
\begin{lstlisting}[language=C]
/**
 * @file
 * @author Tyler Gamvrelis
 *
 * Created on July 10, 2017, 10:54 AM
 *
 * @ingroup Config_18F4620
 */

#ifndef CONFIG_BITS_H
#define CONFIG_BITS_H
```

```
294
295 // CONFIG1H
296 #pragma config OSC = HS          // Oscillator Selection bits (HS oscillator)
297 #pragma config FCMEN = OFF       // Fail-Safe Clock Monitor Enable bit (Fail-
    Safe Clock Monitor disabled)
298 #pragma config IESO = OFF        // Internal/External Oscillator Switchover bit
     (Oscillator Switchover mode disabled)
299
300 // CONFIG2L
301 #pragma config PWRT = OFF        // Power-up Timer Enable bit (PWRT disabled)
302 #pragma config BOREN = SBORDIS   // Brown-out Reset Enable bits (Brown-out
    Reset enabled in hardware only (SBOREN is disabled))
303 #pragma config BORV = 3          // Brown Out Reset Voltage bits (Minimum
    setting)
304
305 // CONFIG2H
306 #pragma config WDT = OFF         // Watchdog Timer Enable bit (WDT disabled (
    control is placed on the SWDTEN bit))
307 #pragma config WDTPS = 32768     // Watchdog Timer Postscale Select bits
    (1:32768)
308
309 // CONFIG3H
310 #pragma config CCP2MX = PORTC    // CCP2 MUX bit (CCP2 input/output is
    multiplexed with RC1)
311 #pragma config PBADEN = ON       // PORTB A/D Enable bit (PORTB<4:0> pins are
    configured as analog input channels on Reset)
312 #pragma config LPT1OSC = OFF     // Low-Power Timer1 Oscillator Enable bit (
    Timer1 configured for higher power operation)
313 #pragma config MCLRE = ON        // MCLR Pin Enable bit (MCLR pin enabled; RE3
    input pin disabled)
314
315 // CONFIG4L
316 #pragma config STVREN = ON       // Stack Full/Underflow Reset Enable bit (
    Stack full/underflow will cause Reset)
317 #pragma config LVP = OFF         // Single-Supply ICSP Enable bit (Single-
    Supply ICSP disabled)
318 #pragma config XINST = OFF       // Extended Instruction Set Enable bit (
    Instruction set extension and Indexed Addressing mode disabled (Legacy
    mode))
319
320 // CONFIG5L
321 #pragma config CP0 = OFF         // Code Protection bit (Block 0 (000800-003
    FFFh) not code-protected)
```

```
#pragma config CP1 = OFF        // Code Protection bit (Block 1 (004000-007FFFh) not code-protected)
#pragma config CP2 = OFF        // Code Protection bit (Block 2 (008000-00BFFFh) not code-protected)
#pragma config CP3 = OFF        // Code Protection bit (Block 3 (00C000-00FFFFh) not code-protected)

// CONFIG5H
#pragma config CPB = OFF        // Boot Block Code Protection bit (Boot block (000000-0007FFh) not code-protected)
#pragma config CPD = ON         // Data EEPROM Code Protection bit (Data EEPROM code-protected)

// CONFIG6L
#pragma config WRT0 = OFF       // Write Protection bit (Block 0 (000800-003FFFh) not write-protected)
#pragma config WRT1 = OFF       // Write Protection bit (Block 1 (004000-007FFFh) not write-protected)
#pragma config WRT2 = OFF       // Write Protection bit (Block 2 (008000-00BFFFh) not write-protected)
#pragma config WRT3 = OFF       // Write Protection bit (Block 3 (00C000-00FFFFh) not write-protected)

// CONFIG6H
#pragma config WRTC = OFF       // Configuration Register Write Protection bit (Configuration registers (300000-3000FFh) not write-protected)
#pragma config WRTB = OFF       // Boot Block Write Protection bit (Boot Block (000000-0007FFh) not write-protected)
#pragma config WRTD = OFF       // Data EEPROM Write Protection bit (Data EEPROM not write-protected)

// CONFIG7L
#pragma config EBTR0 = OFF      // Table Read Protection bit (Block 0 (000800-003FFFh) not protected from table reads executed in other blocks)
#pragma config EBTR1 = OFF      // Table Read Protection bit (Block 1 (004000-007FFFh) not protected from table reads executed in other blocks)
#pragma config EBTR2 = OFF      // Table Read Protection bit (Block 2 (008000-00BFFFh) not protected from table reads executed in other blocks)
#pragma config EBTR3 = OFF      // Table Read Protection bit (Block 3 (00C000-00FFFFh) not protected from table reads executed in other blocks)

// CONFIG7H
#pragma config EBTRB = OFF      // Boot Block Table Read Protection bit (Boot
```

```
    Block (000000-0007FFh) not protected from table reads executed in other
    blocks)

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.

#include <xc.h>

#define _XTAL_FREQ 10000000     // Define osc freq for use in delay macros

#endif /* CONFIG_BITS_H */
```

A.5 I2C.h

```
/**
 * @file
 * @author Michael Ding
 * @author Tyler Gamvrelis
 *
 * Created summer 2016
 *
 * @defgroup I2C
 * @brief I2C driver
 * @{
 */

#ifndef I2C_H
#define I2C_H

/******************************** Includes ********************************
    */
#include <xc.h>
#include <configBits.h>

/******************************** Macros ********************************
    */
// These mean different things depending on the context, see "Understanding
    the
// I2C bus" by Texas Instruments for more details
#define ACK  0 /**< Acknowledge     */
#define NACK 1 /**< Not acknowledge */

/********************** Public Function Prototypes **********************
    */
```

```
/**
 * @brief Initializes the MSSP module for I2C mode. All configuration register
 *        bits are written to because operating in SPI mode could change them
 * @param clockFreq The frequency at which data is to be transferred via the
 *        I2C bus
 * @note The argument is used to generate the baud rate according to the
 *       formula clock = FOSC / (4 * (SSPADD + 1)). Because the argument
 *       sets the 7 bits of control signals in the SSPADD register, the
 *       following are the limitations on the value of clockFreq for
 *       FOSC = 40 MHz: Minimum: 78125, Maximum: 10000000
 */
void I2C_Master_Init(const unsigned long clockFreq);

/**
 * @brief Initiates Start condition on SDA and SCL pins. Automatically cleared
 *        by hardware
 */
void I2C_Master_Start(void);

/**
 * @brief Initiates Repeated Start condition on SDA and SCL pins. Automatically
 *        cleared by hardware
 */
void I2C_Master_RepeatedStart(void);

/**
 * @brief Initiates Stop condition on SDA and SCL pins. Automatically cleared
 *        by hardware
 */
void I2C_Master_Stop(void);

/** @brief Writes a byte to the slave device currently being addressed */
void I2C_Master_Write(unsigned byteToWrite);

/**
 * @brief Reads a byte from the slave device currently being addressed
 * @param ackBit The acknowledge bit
 *        -# ackBit == 0 --> acknowledge bit sent; ready for next bit
 *        -# ackBit == 1 --> no acknowledge bit (NACK); done reading data
 * @return The byte received
 */
unsigned char I2C_Master_Read(unsigned char ackBit);
```

```

/**
 * @}
 */

#endif /* I2C_H */
```

A.6 I2C.c

```
/**
 * @file
 * @author Michael Ding
 * @author Tyler Gamvrelis
 *
 * Created on August 4, 2016, 3:22 PM
 *
 * @ingroup I2C
 */

/******************************** Includes ********************************
    */
#include "I2C.h"

/*************************** Private Functions ***************************
    */
/**
 * @brief Private function used to poll the MSSP module status. This function
 *        exits when the I2C module is idle.
 * @details The static keyword makes it so that files besides I2C.c cannot
 *          "see" this function
 */
static inline void I2C_Master_Wait(){
    // Wait while:
    //   1. A transmit is in progress (SSPSTAT & 0x04)
    //   2. A Start/Repeated Start/Stop/Acknowledge sequence has not yet been
    //      cleared by hardware
    while ((SSPSTAT & 0x04) || (SSPCON2 & 0x1F)){
        continue;
    }
}

/**************************** Public Functions ****************************
    */
void I2C_Master_Init(const unsigned long clockFreq){
```

```
    // Disable the MSSP module
    SSPCON1bits.SSPEN = 0;

    // Force data and clock pin data directions
    TRISCbits.TRISC3 = 1; // SCL (clock) pin
    TRISCbits.TRISC4 = 1; // SDA (data) pin

    // See section 17.4.6 in the PIC18F4620 datasheet for master mode details.
    // Below, the baud rate is configured by writing to the SSPADD<6:0>
    // according to the formula given on page 172
    SSPADD = (_XTAL_FREQ / (4 * clockFreq)) - 1;

    // See PIC18F4620 datasheet, section 17.4 for I2C configuration
    SSPSTAT = 0b10000000; // Disable slew rate control for cleaner signals

    // Clear errors & enable the serial port in master mode
    SSPCON1 = 0b00101000;

    // Set entire I2C operation to idle
    SSPCON2 = 0b00000000;
}

void I2C_Master_Start(void){
    I2C_Master_Wait(); // Ensure I2C module is idle
    SSPCON2bits.SEN = 1; // Initiate Start condition
}

void I2C_Master_RepeatedStart(void){
    I2C_Master_Wait(); // Ensure I2C module is idle
    SSPCON2bits.RSEN = 1; // Initiate Repeated Start condition
}

void I2C_Master_Stop(void){
    I2C_Master_Wait(); // Ensure I2C module is idle
    SSPCON2bits.PEN = 1; // Initiate Stop condition
}

void I2C_Master_Write(unsigned byteToWrite){
    I2C_Master_Wait(); // Ensure I2C module is idle

    // Write byte to the serial port buffer for transmission
    SSPBUF = byteToWrite;
}
```

```

unsigned char I2C_Master_Read(unsigned char ackBit){
    I2C_Master_Wait(); // Ensure I2C module is idle
    SSPCON2bits.RCEN = 1; // Enable receive mode for I2C module

    I2C_Master_Wait(); // Wait until receive buffer is full

    // Read received byte from the serial port buffer
    unsigned char receivedByte = SSPBUF;

    I2C_Master_Wait(); // Ensure I2C module is idle
    SSPCON2bits.ACKDT = ackBit; // Acknowledge data bit
    SSPCON2bits.ACKEN = 1; // Initiate acknowledge bit transmission sequence

    return receivedByte;
}
```

A.7 lcd.h

```
/**
 * @file
 * @author Michael Ding
 * @author Tyler Gamvrelis
 *
 * Created on August 12, 2016, 4:24 PM
 *
 * @defgroup CharacterLCD
 * @brief Driver for Hitachi HD44780-based character LCD
 * @{
 */

#ifndef LCD_H
#define LCD_H

/******************************** Includes ********************************
    */
#include <xc.h>
#include <stdio.h>
#include <stdbool.h>
#include <configBits.h>

/********************************* Macros *********************************
    */
#define RS LATDbits.LATD2
```

```
24 #define E  LATDbits.LATD3
25
26 /** @brief Clears both LCD lines */
27 #define lcd_clear(){\
28   lcdInst(0x01);\
29   __delay_ms(5);\
30 }
31
32 /** @brief Sets cursor position to start of first line */
33 #define lcd_home(){\
34   lcdInst(0x80);\
35   __delay_ms(2);\
36 }
37
38 /**
39  * @brief Sets the cursor's position to a specific display data RAM (DDRAM)
40  *        address
41  * @param addr The DDRAM address to move the cursor to (min: 0, max: 127)
42  * @note The cursor will not be visible at all addresses
43  */
44 #define lcd_set_ddram_addr(addr){\
45     lcdInst(0x80 | addr);\
46 }
47
48 /**
49  * @brief Backlight and cursor control
50  * @param display_on Turns on the backlight if true, otherwise turns it off
51  * @param cursor_on Turns on cursor if true, otherwise turns it off
52  * @param blink_cursor Blinks the cursor if true, otherwise cursor is static
53  */
54 #define lcd_display_control(\
55   display_on,\
56   cursor_on,\
57   blink_cursor\
58 )\
59 {\
60   lcdInst(\
61         (unsigned char)(8 | (display_on << 2) | (cursor_on << 1) |
    blink_cursor)\
62     );\
63 }
64
65 /****************************** Constants ******************************
```

```
   */
// Display dimensions as seen in the real world (before you use these in your
// code, double-check that they match the size of your LCD)
extern const unsigned char LCD_SIZE_HORZ; /**< Number of visible columns */
extern const unsigned char LCD_SIZE_VERT; /**< Number of visible rows    */

extern const unsigned char LCD_LINE1_ADDR; /**< Address of first line */
extern const unsigned char LCD_LINE2_ADDR; /**< Address of 2nd line   */
extern const unsigned char LCD_LINE3_ADDR; /**< Address of 3rd line   */
extern const unsigned char LCD_LINE4_ADDR; /**< Address of 4th line   */

/********************************* Types *********************************
   */
/** @brief The directions the display contents and cursor can be shifted */
typedef enum{
  LCD_SHIFT_LEFT = 0, /**< Left shift  */
  LCD_SHIFT_RIGHT = 1 /**< Right shift */
}lcd_direction_e;

/*********************** Public Function Prototypes ***********************
   */
/**
 * @brief Sends a command to a display control register
 * @param data The command byte for the Hitachi controller
 */
void lcdInst(char data);

/** @brief Performs the initial setup of the LCD */
void initLCD(void);

/**
 * @brief Moves the cursor in a given direction by numChars characters
 * @param numChars The number of character positions by which the cursor is to
 *        be moved (min: 0, max: 127)
 * @param direction The direction for which the shift is to occur
 */
void lcd_shift_cursor(unsigned char numChars, lcd_direction_e direction);

/**
 * @brief Shifts the display in a given direction by numChars characters
 * @param numChars The number of character positions by which the display
 *        contents are to be shifted (min: 0, max: 127)
 * @param direction The direction for which the shift is to occur
```

```
 */
void lcd_shift_display(unsigned char numChars, lcd_direction_e direction);

/**
 * @brief Sends a character to the display for printing
 * @details The familiar C function printf internally calls a function named
 *          "putch" (put character) whenever a character is to be written to
 *          the screen. Here we have chosen to implement putch so that it
 *          sends the character to the LCD, but you can choose to implement it
 *          however you'd like (e.g. send the character over UART, etc.)
 * @param data The character (byte) to be displayed
 */
void putch(char data);

/**
 * @}
 */

#endif  /* LCD_H */
```

A.8 lcd.c

```
/**
 * @file
 * @author Michael Ding
 * @author Tyler Gamvrelis
 *
 * Created on July 18, 2016, 12:11 PM
 * @ingroup CharacterLCD
 */

/****************************** Includes ******************************
    */
#include "lcd.h"

/****************************** Constants ******************************
    */
const unsigned char LCD_SIZE_HORZ = 16;
const unsigned char LCD_SIZE_VERT = 4;

const unsigned char LCD_LINE1_ADDR = 0;
const unsigned char LCD_LINE2_ADDR = 64;
const unsigned char LCD_LINE3_ADDR = 16;
const unsigned char LCD_LINE4_ADDR = 80;
```

```
/**************************** Private Functions ***************************
    */
/**
 * @brief Pulses the LCD register enable signal, which causes the LCD to latch
 *        the data on LATD. Interrupts are disabled during this pulse to
 *        guarantee that the timing requirements of the LCD's protocol are met
 */
static inline void pulse_e(void){
    unsigned char interruptState = INTCONbits.GIE;
    di();
    E = 1;
    // This first delay only needs to be 1 microsecond in theory, but 25 was
    // selected experimentally to be safe
    __delay_us(25);
    E = 0;
    __delay_us(100);
    INTCONbits.GIE = interruptState;
}

/**
 * @brief Low-level function to send 4 bits to the display
 * @param data The byte whose 4 least-significant bits are to be sent to the
    LCD
 */
static void send_nibble(unsigned char data){
    // Send the 4 least-significant bits
    LATD = (unsigned char)(LATD & 0x0F); // Clear LATD[7:4]
    LATD = (unsigned char)((data << 4) | LATD); // Write data[3:0] to LATD
    [7:4]
    pulse_e();
}

/**
 * @brief Low-level function to send a byte to the display
 * @param data The byte to be sent
 */
static void send_byte(unsigned char data){
    // Send the 4 most-significant bits
    send_nibble(data >> 4);

    // Send the 4 least-significant bits
    send_nibble(data);
```

```
}

/*************************** Public Functions ***************************
    */
void lcdInst(char data){
    RS = 0;
    send_byte(data);
}

void initLCD(void){
    __delay_ms(15);

    RS = 0;
    // Set interface length to 4 bits wide
    send_nibble(0b0011);
    __delay_ms(5);
    send_nibble(0b0011);
    __delay_us(150);
    send_byte(0b00110010);

    send_byte(0b00101000); // Set N = number of lines (1 or 2) and F = font
    send_byte(0b00001000); // Display off
    send_byte(0b00000001); // Display clear
    __delay_ms(5);
    send_byte(0b00000110); // Entry mode set

    // Enforce on: display, cursor, and cursor blinking
    lcd_display_control(true, true, true);
}

void lcd_shift_cursor(unsigned char numChars, lcd_direction_e direction){
    for(unsigned char n = numChars; n > 0; n--){
        lcdInst((unsigned char)(0x10 | (direction << 2)));
    }
}

void lcd_shift_display(unsigned char numChars, lcd_direction_e direction){
    for(unsigned char n = numChars; n > 0; n--){
        lcdInst((unsigned char)(0x18 | (direction << 2)));
    }
}

void putch(char data){
```

```
    RS = 1;
    send_byte((unsigned char)data);
}
```

A.9 pitches.h

```
/**********************************************
 * Public Constants
 **********************************************/

#define NOTE_B0  31
#define NOTE_C1  33
#define NOTE_CS1 35
#define NOTE_D1  37
#define NOTE_DS1 39
#define NOTE_E1  41
#define NOTE_F1  44
#define NOTE_FS1 46
#define NOTE_G1  49
#define NOTE_GS1 52
#define NOTE_A1  55
#define NOTE_AS1 58
#define NOTE_B1  62
#define NOTE_C2  65
#define NOTE_CS2 69
#define NOTE_D2  73
#define NOTE_DS2 78
#define NOTE_E2  82
#define NOTE_F2  87
#define NOTE_FS2 93
#define NOTE_G2  98
#define NOTE_GS2 104
#define NOTE_A2  110
#define NOTE_AS2 117
#define NOTE_B2  123
#define NOTE_C3  131
#define NOTE_CS3 139
#define NOTE_D3  147
#define NOTE_DS3 156
#define NOTE_E3  165
#define NOTE_F3  175
#define NOTE_FS3 185
#define NOTE_G3  196
#define NOTE_GS3 208
```

```
#define NOTE_A3   220
#define NOTE_AS3  233
#define NOTE_B3   247
#define NOTE_C4   262
#define NOTE_CS4  277
#define NOTE_D4   294
#define NOTE_DS4  311
#define NOTE_E4   330
#define NOTE_F4   349
#define NOTE_FS4  370
#define NOTE_G4   392
#define NOTE_GS4  415
#define NOTE_A4   440
#define NOTE_AS4  466
#define NOTE_B4   494
#define NOTE_C5   523
#define NOTE_CS5  554
#define NOTE_D5   587
#define NOTE_DS5  622
#define NOTE_E5   659
#define NOTE_F5   698
#define NOTE_FS5  740
#define NOTE_G5   784
#define NOTE_GS5  831
#define NOTE_A5   880
#define NOTE_AS5  932
#define NOTE_B5   988
#define NOTE_C6   1047
#define NOTE_CS6  1109
#define NOTE_D6   1175
#define NOTE_DS6  1245
#define NOTE_E6   1319
#define NOTE_F6   1397
#define NOTE_FS6  1480
#define NOTE_G6   1568
#define NOTE_GS6  1661
#define NOTE_A6   1760
#define NOTE_AS6  1865
#define NOTE_B6   1976
#define NOTE_C7   2093
#define NOTE_CS7  2217
#define NOTE_D7   2349
#define NOTE_DS7  2489
```

```
#define NOTE_E7   2637
#define NOTE_F7   2794
#define NOTE_FS7  2960
#define NOTE_G7   3136
#define NOTE_GS7  3322
#define NOTE_A7   3520
#define NOTE_AS7  3729
#define NOTE_B7   3951
#define NOTE_C8   4186
#define NOTE_CS8  4435
#define NOTE_D8   4699
#define NOTE_DS8  4978
```

17 References

[1] Smithers Rapra. (2016) Tire industry in boom cycle. [Online]. Available: https://www.smithersrapra.com/resources/2017/april/tire-industry-in-boom-cycle

[2] F. Rader. The history of tires. [Online]. Available: https://www.tirerecappers.com/tire-recappers-news/the-history-of-tires/

[3] IBISWorld. Tire manufacturing - canada market research report. [Online]. Available: https://www.ibisworld.ca/industry-trends/market-research-reports/manufacturing/tire-manufacturing.html

[4] Vulcana. History of tire recycling. [Online]. Available: https://www.vulcana.net/tire-recycling.htm

[5] Martins Industries. What is the best way to store your tires? [Online]. Available: https://martinsindustries.com/en/?microsite=what-is-the-best-way-to-store-your-tires

[6] R. Emami, "Request for proposal #2: The tire stacking machine," University of Toronto, RFP, 2019.

[7] Shenzhen Xuanyao Electronics Co., Ltd. 16mm latching on/off switch illuminated pushbutton with colorful led. [Online]. Available: https://www.alibaba.com/product-detail/16mm-Latching-On-Off-Switch-Illuminated_60793916262.html?spm=a2700.7724838.2017115.333.64073495AZsJhZ

[8] Cixi Sinco Machinery Factory. Bh309-gr 10 aa battery holder. [Online]. Available: https://www.alibaba.com/product-detail/BH309-GR-10-aa-battery-holder_60424596407.html?spm=a2700.7724838.2017115.23.523d146c4kg5Ik

[9] Shenzhen Xin Yuan Electronic Technology Co., Ltd. China l298n motor driver board l298 module for stepper motor smart car robot. [Online]. Available: https://www.alibaba.com/product-detail/China-L298N-motor-driver-board-L298_60734910820.html?spm=a2700.7724838.2017115.27.32714baaBO9Owt

[10] General Electronics Technology (Shenzhen) Co., Ltd. Get hot selling nimh aa 1600mah 1.2v rechargeable battery. [Online]. Available: https://www.alibaba.com/product-detail/GET-hot-selling-nimh-aa-1600mah_60832625793.html?spm=a2700.7724838.2017115.25.3ed25adcJPaVZw&s=p

[11] Pandas Tech. Arduino uno r3 development board microcontroller with usb cable atmega328p suitable for win mac. [Online]. Available: https://www.newegg.ca/Product/Product.aspx?Item=9SIA8Y63G31810&ignorebbr=1&source=region&nm_mc=KNC-GoogleAdwordsCAMKPL-PC&cm_mmc=KNC-GoogleAdwordsCAMKPL-PC-_-pla-Pandas+Tech-_-Gadgets-_-9SIA8Y63G31810&gclid=Cj0KCQjwkIzlBRDzARIsABgXqV_BMD0Gei78iMtsXQL67i4CHG8ZNICFXZ0Z7yEmldYzwXZ9oJP6OJoaAn9WEALw_wcB

[12] Shenzhen Green New Source Lights Technology Co., Ltd. 24-26lm/smd brightness super bright waterproof ip67 outdoor rigid aluminum led strip light bar 12v 24v green blue red yellow. [Online]. Available: https://www.alibaba.com/product-detail/24-26LM-SMD-Brightness-Super-Bright_60716436593.html?spm=a2700.7724838.2017115.131.388d25d0d3E2Sw

[13] Changzhou Wujin Lianhua Electronics Co., Ltd. High quality 50mm 0.5w mylar speaker 8ohm waterproof speaker. [Online]. Available: https://www.alibaba.com/product-detail/High-quality-50mm-0-5w-mylar_60207008896.html?spm=a2700.galleryofferlist.normalList.11.429c60e8NQLUmW&s=p

[14] Shenzhen Rainbowsemi Electronics Co., Ltd. Stepper motor 28byj-48. [Online]. Available: https://www.alibaba.com/product-detail/Stepper-Motor-28BYJ-48_60558319276.html?spm=a2700.7724857.normalList.38.524525adcniu2r

[15] 1pcs 9g micro servo for airplane aeroplane 6ch rc helcopter kds esky align helicopter sg90. [Online]. Available: https://ca.dhgate.com/product/1pcs-9g-micro-servo-for-airplane-aeroplane/440869456.html

[16] Huimai. 10pcs 5v 650nm 5mw adjustable laser dot diode module red sight copper head mini laser pointer. [Online]. Available: https://www.amazon.com/10PCS-Adjustable-Module-Copper-Pointer/dp/B07G6KHBFW

[17] TxHang Electronic. Laser sensor module non-modulator tube laser receiver new module. [Online]. Available: https://www.ebay.com/itm/Laser-Sensor-Module-non-modulator-Tube-Laser-Receiver-New-Module-/401160430957

[18] Gy - 521 mpu - 6050 triaxial acceleration gyroscope 6 dof module. [Online]. Available: https://www.gearbest.com/sensors/pp_009636731429.html?wid=1433363&

currency=CAD&vip=4452582&gclid=Cj0KCQjw4qvlBRDiARIsAHme6ouNCXa_KaWD1szohVsK8Aqwdmc08IOqfQZFjU5zoV6MjuRDykcAjRYaAh0vEALw_wcB

[19] Bank of Canada. Daily exchange rate. [Online]. Available: https://www.bankofcanada.ca/rates/exchange/daily-exchange-rates/

[20] Creatron Inc. 1/4w 5% resistor (10 pack). [Online]. Available: https://www.creatroninc.com/product/14w-5-resistor-10-pack/?search_query=1k&results=40#/73-1_4w_5_resistor-1kohm

[21] F. Semiconductor, "Lm78xx/lm78xxa 3-terminal 1a positive voltage regulator," 2006.

[22] T. Instruments, "Lm138 and lm338 5-amp adjustable regulators," 2016.

[23] "Shenzhen dc motor angle."

[24] STMicroelectronics, "L298 full bridge driver," 2000.

[25] George. Unipolar stepper motor vs bipolar stepper motors. [Online]. Available: https://www.circuitspecialists.com/blog/unipolar-stepper-motor-vs-bipolar-stepper-motors/

[26] JunezRiyaz. Simple audio amplifier using single transistor. [Online]. Available: https://www.instructables.com/id/Simple-Audio-Amplifier-Using-Single-Transistor/

[27] T. E. ToolBox. Rolling resistance. [Online]. Available: https://www.engineeringtoolbox.com/rolling-friction-resistance-d_1303.html

[28] Friction coefficients between different wheel/tire materials and concrete. [Online]. Available: http://www2.mae.ufl.edu/designlab/Class%20Projects/Background%20Information/Friction%20coefficients.htm

[29] P. M. Lab, *Personal Mechatronics Lab DevBugger User Manual*.

[30] Microchip, "Pic18f2525/2620/4525/4620 data sheet."

[31] Pic18f4620 - microcontrollers and processors. [Online]. Available: https://www.microchip.com/wwwproducts/en/PIC18F4620

[32] D. Dallos. Android things basics: Measure distance with ultrasonic sensor. [Online]. Available: https://hackernoon.com/android-things-basics-measure-distance-with-ultrasonic-sensor-3196fe5d7d7c

[33] Pololu - sharp gp2y0a51sk0f analog distance sensor 2-15cm. [Online]. Available: https://www.pololu.com/product/2450

[34] Sharp, "Gp2y0a51sk0f data sheet."

[35] Vl53l0x laser ranger module. [Online]. Available: https://makerfabs.com/VL53L0X-Laser-Ranger-Module.html

[36] SkuleTube. Professor michael p. collins: What is engineering science? [Online]. Available: https://www.youtube.com/watch?v=YYaJ_pGklE0