Honors Research Project

ROObockey: Remote Controlled, Aim-Assisted Street Hockey Robot

The goal of the ROObockey project is to design and construct a floor hockey robot that can competitively shoot a puck. The robot design quickly locates a specific beacon through the use of image processing and uses a pneumatic shooting mechanism to send a puck to a specified target. The beacons act as possible player or goal positions in a hockey game. The robot also utilizes a wireless controller device to allow a user to maneuver the robot across a hockey field.

Deboshri Sadhukhan, Electrical Engineering, is the Project Manager overseeing all team members and design progress. This includes keeping an up-to-date schedule, budget, and task list for the project. She is also responsible for the design of the power distribution system and battery implementation as well as wire management and integration. This includes the schematic and PCB layout for the power distribution, selection of batteries, wiring the beacons and assistance with wire harnessing and integration points of the whole system.

Troy W. Bowers, Electrical Engineering, is the Archivist overseeing compilation of the final design report and documentation during the process. He is also responsible for the development and implementation of the wireless controller and communications system and supporting software and electrical hardware implementation. This includes interfacing the wireless controller with the main processor through software design and implementing embedded systems communications to control the shooting mechanism and the motor controller to properly drive the robot.

Keith R. Martin, Electrical Engineering, is the Software Manager in charge of image processing and motor control software design and implementation. This includes developing all software for image processing and beacon detection systems, software for autonomous maneuvering based on beacon detection, and compiling all the software used in implementation into a single project. He is also responsible for supporting electrical hardware specification and implementation.

Final Design Report

Design Team E
"ROObockey"

Deboshri Sadhukhan
John Supel
Keith Martin
Troy Bowers

Faculty Advisor: Dr. Kye-Shin Lee

Date Submitted: 04/22/2016

Contents

Design Responsibilities 8

Abstract 8

1. Introduction 8

2. Problem Statement 9

2.1. Need 9

2.2. Objective 9

2.3. Background (Research Survey) 9

2.3.1. Overview 9

2.3.2. Relevant Technologies 9

2.4. Objective Tree 10

3. Design Requirements Specification 11

3.1. Engineering and Marketing Requirements 11

4. Technical Design 12

4.1. Level 0 12

4.2. Level 1 13

4.2.1. Hardware 13

4.2.2. Software 15

4.3. Level 2 16

4.3.1. System Overview 16

4.3.2. Sensor Network [KM] 17

4.3.3. Wireless Controlled Communications [TB] 20

4.3.4. Shooting Mechanism [JS] 24

4.3.5. Main Processing Unit [KM] 28

4.3.6. Motor Control System [JS] 34

4.3.7. Power Distribution [DS] 38

4.3.8. Mechanical Design [JS] 46

4.3.9. Sensor Beacons [DS & KM] 47

4.3.10. Software Implementation [KM] 50

5. Operation Instructions 53

5.1. Robot Startup 53

5.2. Driving 53
5.3. Shooting and Targeting 53
6. Design Team Information 54
7. Conclusions and Recommendations 54
References 55
Appendix 58
A. 5V DC-DC Convertor Calculations [DS] 58
B. 24V DC-DC Convertor Calculations [DS] 59
C. Wiring Diagrams [DS] 61
D. Software Code included in the project for the Raspberry Pi 2: 64
i. Main 64
ii. Object Tracking 67
iii. Xbox 360 Controller 88
iv. GPIO and UART Communication 93
v. Beacon Detection 101
vi. Variable Definitions 105
E. Beacon Design 110
i. Beacon Software 110
F. Parts Request 112

Table of Tables

Table 1: Engineering and Marketing Requirements....11
Table 2: Level 0 Input-Outputs....12
Table 3: Sensor Network Level 1 Input-Outputs....14
Table 4: Wireless Controlled Communications Level 1 Input-Outputs....14
Table 5: Main Processing Unit Level 1 Input-Outputs....14
Table 6: Shooting Mechanism Level 1 Input-Outputs....14
Table 7: Motor Controller Level 1 Input-Outputs....15
Table 8: Power Distribution Level 1 Input-Outputs....15
Table 9: Camera Input Level 2 Input-Outputs....15
Table 10: Shape Recognition Level 2 Input-Outputs....16
Table 11: Robot Motor Position Algorithm Level 2 Input-Outputs....16
Table 12: Image Processing Level 2 Input-Outputs....18
Table 13: Motor Control Algorithm Using PWM Module....19
Table 14: Software Image Shape Search Algorithm....19
Table 15: 2.4GHz Xbox Controller / Dongle Level 2 Input-Outputs....21
Table 16: Xbox Controller Driver....22
Table 17: Motor Drive Algorithm Input-Outputs....22
Table 18: Shooting Mechanism Level 2 Input-Outputs....26
Table 19: 9oz CO2 Tank Specifications....26
Table 20: Paintball Tank to Pneumatic Prop Regulator Specifications....27
Table 21: 4-way 5-port valve with 1/4 inch ports Specifications....27
Table 22: ¾ inch bore double-acting universal mount cylinder Specifications....27
Table 23: Software-Assisted Motor Control Level 2 Input-Outputs....32
Table 24: Input Motor Direction from Wireless Controller....34
Table 25: Input USB Camera Feed....34
Table 26: DC Motor Level 2 Input-Outputs....35
Table 27: 3202 Pololu Gear Motor Specifications....36
Table 28: Motor Controller Level 2 Input-Outputs....37
Table 29: Motor Controller Specifications....37
Table 30: Power Distribution Level 2 Input-Outputs....39
Table 31: Mechanical Specifications....46
Table 32: GPIO Pin Assignments....51

Table of Figures

Figure 1: Objective Tree 10
Figure 2: Level 0 Block Diagram 12
Figure 3: Hardware Level 1 Block Diagram 13
Figure 4: Software Level 1 Block Diagram 15
Figure 5: System Overview Level 2 Block Diagram 17
Figure 6: Image Processing Level 2 Hardware Block Diagram 18
Figure 7: 5MP USB Camera with 120 Degree Lens (PN: Genius Widecam F100) 20
Figure 8: Wireless Controlled Communication Level 2 Diagram 21
Figure 9: Wireless Controlled Communication Level 3 Diagram 22
Figure 10: Controller Inputs in Raspbian Terminal 23
Figure 11: Shooting Mechanism CAD Model 25
Figure 12: Pneumatic constants [28] 25
Figure 13: Shooting Mechanism Level 2 Block Diagram 26
Figure 14: Main Processor ARM V7 on Raspberry Pi 28
Figure 15: Main Processing Unit Level 2 Block Diagram 29
Figure 16: Main Processor Software Overview Level 2 Diagram 30
Figure 17: Main Processor to Motor Controller Software Level 2 Diagram 30
Figure 18: Software X-Coordinate Sections Level 2 Diagram 31
Figure 19: Software Image Processing Algorithm Level 2 Diagram 31
Figure 20: Software Image Processing Overview Level 2 Diagram 32
Figure 21: Software-Assisted Motor Control Level 2 Diagram 32
Figure 22: Motor Controller Level 2 Block Diagram 35
Figure 23: Force Diagram for Wheels 35
Figure 24: Sabertooth 2x05 Motor Controller 38
Figure 25: Power Distribution Level 2 Block Diagram 39
Figure 26: Power Distribution Diagram 40
Figure 27: Power Board Schematic 41
Figure 28: Power Board 2D Layout 41
Figure 29: Power Board 3D Layout 42
Figure 30: Power Board Rev. 1 43
Figure 31: 5V DC-DC Converter Rev.1 Bench Testing 43
Figure 32: 24V DC-DC Converter Rev.1 Bench Testing 44
Figure 33: Power Board Rev. 2 44
Figure 34: 5V DC-DC Converter Rev. 2 Bench Testing 45
Figure 35: 24V DC-DC Converter Rev 2 Bench Testing 45
Figure 36: Robot Mechanical CAD Model 46
Figure 37: Finished Robot Mechanical Design 47
Figure 38: Beacon Hardware Schematic 48
Figure 39: Beacons 49

Figure 40: Beacon Circuitry 49
Figure 41: Beacon Operation and Tracking Using Color and Shape Recognition 50
Figure 42: Software Parallel Hierarchy of Parallel Threads.....51

Design Responsibilities

Deboshri Sadhukhan, Electrical Engineering, is the Project Manager overseeing all team members and design progress as well as in charge of the design of power distribution design and battery implementation as well as wire management and integration.

John Supel, Electrical Engineering, is the Hardware Manager overseeing major aspects of electrical and mechanical hardware design as well as the shooting mechanism and motor control system implementation.

Keith Martin, Electrical Engineering, is the Software Manager in charge of image processing and motor control software design and implementation as well as support of electrical hardware specification.

Troy Bowers, Electrical Engineering, is the Archivist in charge of documentation, radio controller implementation and system interfacing, as well as support of electrical hardware specification.

Abstract

The goal of the ROObockey project is to design and construct a floor hockey robot that can competitively shoot a puck. The robot design quickly locates a specific beacon through the use of image processing and uses a pneumatic shooting mechanism to send a puck to a specified target. The beacons act as possible player or goal positions in a hockey game. The robot also utilizes a wireless controller device to allow a user to maneuver the robot across a hockey field.

Key Features:

- Accurate Image processing to precisely determine position relative to other beacons
- Ability to pass and shoot a hockey puck based on determined location
- Radio control for robot maneuverability
- Robust mechanical design to withstand standard game play

1. Introduction

As technology advances, image processing is being used in numerous applications especially within robotics. Examples of image processing can be found in autonomous vehicles and commercial robots. The Robogames Competition held in San Mateo, California challenges students to use new technologies to design a robot hockey player capable of passing and shooting a puck as well as maneuvering across a hockey arena. ROObockey is inspired by the current competitions but the design includes a more accurate and intelligent shooting mechanism.

2. Problem Statement

2.1. Need

A game of hockey consists of multiple challenges, including the ability to accurately pass a puck to another player or shoot a puck into the goal. Hockey robots currently rely on human control to determine the target and pass the puck accordingly. A sensing system used to precisely locate the position of another player would eliminate human error in passing the puck. This sensing system could be expanded in the future to multiple robots to create a robot hockey team with intelligent passing and shooting capabilities.

2.2. Objective

The objective of ROObockey is to design and build a hockey robot capable of accurately shooting a puck at a goal target. Stationary objects will be used as target beacons for shooting the puck. The robot will be wirelessly controlled for manual operation. It will also receive a shooting command from the user.

2.3. Background (Research Survey)

2.3.1. Overview

The main idea of the project is to design a robot that can implement software-assisted shooting of a hockey puck. The approach to this task is to use image processing of various shapes and colors to distinguish between the different targets. The camera used for supplying the input to the image processing system could use a lens that can provide a wider field of view. [3] An electromechanical shooting mechanism will be used to pass and launch the puck to the user-defined targets.

2.3.2. Relevant Technologies

At the present time, hockey robot patents pertain to humanoid robots that simulate a full scale, typical ice hockey game. An idea of the patent holder was to suspend full-scale controllable robot hockey players from the ceiling and to control them from a remote location. Another patent idea consists of using a rotary motor hockey puck launcher. There are no patents held for hockey robots used with wireless controlled applications. [1][2]

2.4. Objective Tree

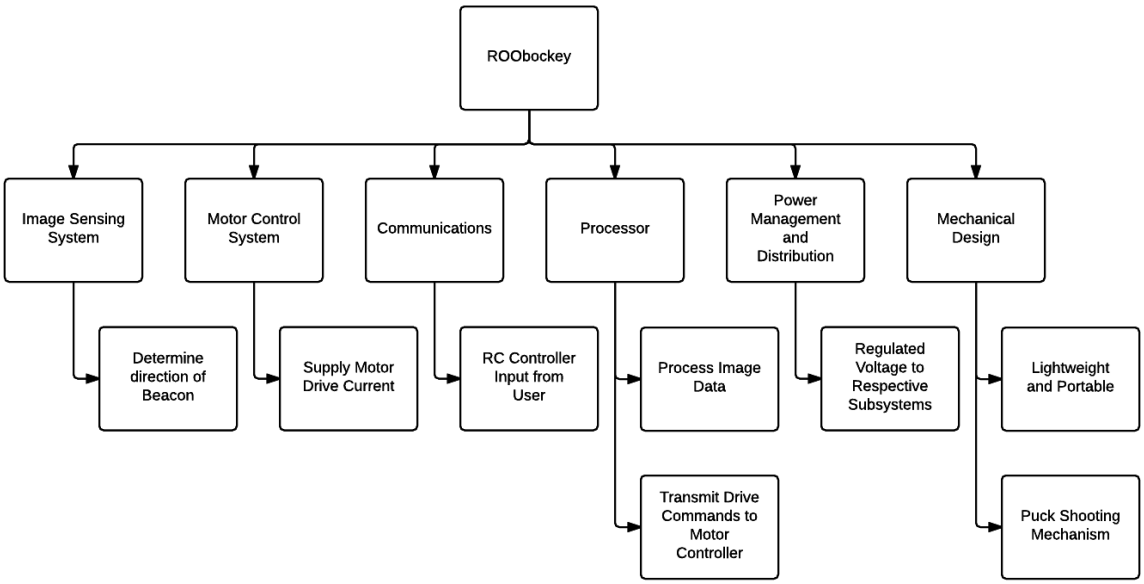


Figure 1: Objective Tree

3. Design Requirements Specification

The ROObockey robot consists of multiple sensor inputs. The system will require battery power to operate. The system will receive wireless control input, interpret the control and image data, and then operate the motors.

3.1. Engineering and Marketing Requirements

Table 1: Engineering and Marketing Requirements

Marketing Requirements	Engineering Requirements	Justification
5	Use high resolution webcam to identify which object the robot should send the puck towards	Must have high enough precision with digital hardware to narrow in on which object to send the puck up to 25'
3	Battery Voltage: 12V Battery Capacity: 10Ah	Must have enough operating voltage and enough capacity to operate for an extended period of time
6,2	Robot can be lifted by one person and weighs approximately under 15 lbs.	Robot must be easily maneuverable and be moved by a single person
6	Robot must reside within 18" x 18" x 18" dimensions	Robot must be within reasonable dimensions
1	Must be able to operate under manual control from a user with a 2.4GHz radio controller	Option to aim manually and autonomously
4	Use reliable wireless controller to maneuver robot and shoot hockey puck	Must have a reliable range of up to 25’ and several buttons

Marketing Requirements

1. Ability to send the hockey puck manually where the user commands
2. Ability to maneuver and spin 360 degrees
3. Long lasting, rechargeable battery
4. Uses wireless control to operate the robot
5. Ability to autonomously pass and shoot the puck with user input
6. Lightweight, portable and compact design

4. Technical Design

4.1. Level 0

The ROObockey robot is designed to simulate the feel of a game of hockey using a wirelessly controlled robot. The robot will implement a small, lightweight design to allow the user to smoothly maneuver the robot around. A shooting mechanism will be implemented to propel the puck toward a goal beacon. The robot will include a sensor system to recognize and distinguish between targets. The aiming and shooting feature will be software aided to simplify the controls and interactions needed from the user, allowing for quick and simple shots.

Receiving quality sensor data is a prominent concern since the automated shooting and passing completely relies on this information. The robot will use this sensor data to determine the direction of a goal and set itself to that orientation. Additionally, the time it takes the robot to calculate that direction is crucial. In order to maintain smooth gameplay, the robot will need to pass shortly after the pass command is received.

The following is a simple Level 0 block diagram of the entire system.

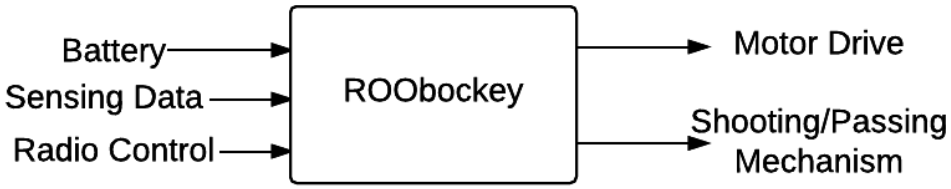


Figure 2: Level 0 Block Diagram

Table 2: Level 0 Input-Outputs

Module	Level 0 Description
Inputs	Rechargeable Battery Sensor Data Wireless Control Inputs
Outputs	Motor Drive Shooting/Passing Mechanism
Functionality	Small, lightweight maneuverable design Reliably detect goal beacon Ability to shoot the puck Efficient battery usage Reliable connection to wireless controller

4.2. Level 1

4.2.1. Hardware

The hockey robot will be battery powered allowing for untethered operation. The battery will be at a higher voltage than what the sensors, wireless controller, and processor can require. A voltage regulator will be used to safely power the devices.

The power management and distribution system will provide the step-up and step-down voltages required for the system to operate.

Wireless controller input will allow the user to control the robot. The communication device will control the maneuvering of the robot and shooting command.

The sensor network will consist of a camera to locate a beacon and a sensor to determine whether the robot currently has the puck. The sensor data is all sent to the processor where all the needed calculations can be processed.

The main processor unit will handle signal and data processing coming from the sensor network and wireless controller input. It will also be in charge of controlling the motor control system and shooting mechanism.

The motor control system will drive the DC motors. This is a switch between the battery and the motors, controlling the speed and direction of rotation for the motors.

The shooting mechanism will consist of a device capable of projecting the hockey puck forward. It receives the release command from the processor.

The Hardware Level 1 block diagram shown in Figure 3 below consists of each of these systems along with the corresponding inputs and outputs throughout the system.

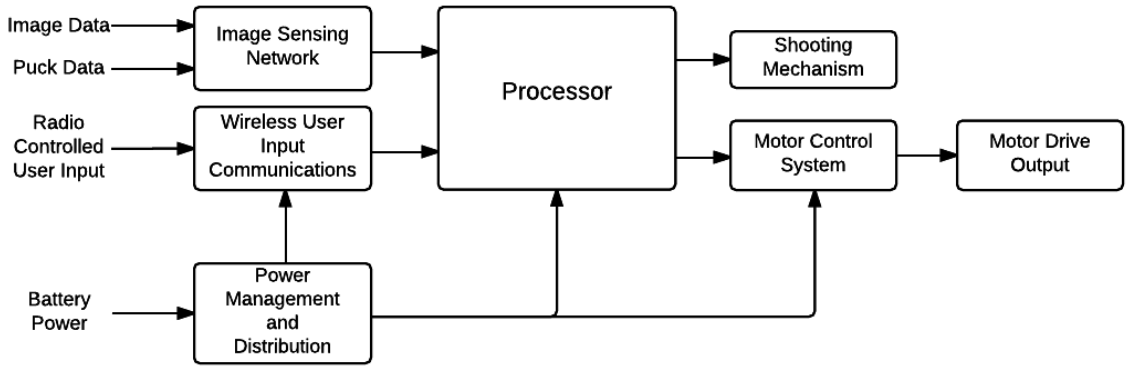


Figure 3: Hardware Level 1 Block Diagram

Table 3: Sensor Network Level 1 Input-Outputs

Module	Sensor Network
Inputs	Recognize direction of other beacons
Outputs	Direction of beacons
Functionality	Obtain location of beacons and position robot accordingly

Table 4: Wireless Controlled Communications Level 1 Input-Outputs

Module	Wireless Controlled Communications
Inputs	Radio control from user
Output	Radio control data to processor
Functionality	Communication from user to robot

Table 5: Main Processing Unit Level 1 Input-Outputs

Module	Main Processing Unit
Inputs	Data from Image Sensor Radio Communications Power
Output	Data to motor control system Communications System state
Functionality	Controls all operations for the robot

Table 6: Shooting Mechanism Level 1 Input-Outputs

Module	Shooting Mechanism
Inputs	Battery power User control input
Outputs	Output force
Functionality	Shoots the puck

Table 7: Motor Controller Level 1 Input-Outputs

Module	Motor Control System
Inputs	Processor control for motor
Outputs	Motor drive current
Functionality	Drives motors to maneuver robot

Table 8: Power Distribution Level 1 Input-Outputs

Module	Power Management and Distribution
Inputs	Battery charging current
Outputs	Power for modules to operate
Functionality	Maintain supply voltage to all modules

4.2.2. Software

The design incorporates color and shape recognition image processing to locate the beacon the user wishes to pass and shoot the puck towards. The software receives the camera input, processes the images while searching for the beacon, and then makes a decision on how to direct the motors. The PWM leaving the processor will tell the motors how fast to spin the motors by varying the duty cycle. An overview of the software process flow is shown below in Figure 4, the Software Level 1 block diagram. It incorporates the camera input, shape and color detection, and algorithm to move the motors.

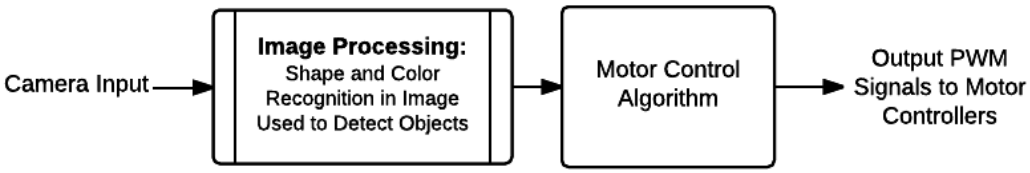


Figure 4: Software Level 1 Block Diagram

Table 9: Camera Input Level 2 Input-Outputs

Module	Camera Input
Inputs	Camera power and camera lens
Outputs	Camera feed
Functionality	Retrieve video in front of robot

Table 10: Shape Recognition Level 2 Input-Outputs

Module	Shape and Color Recognition
Inputs	Camera feed
Outputs	Angle between robot and desired object
Functionality	Search for the beacon the user wants to shoot the puck towards and calculate how much the robot needs to rotate in order to line up with the beacon

Table 11: Robot Motor Position Algorithm Level 2 Input-Outputs

Module	Robot Motor Position Algorithm
Inputs	Angle between robot and desired object
Outputs	PWM to motor controller to drive motors
Functionality	Tell the motor controller how much to spin the motors to line up with beacon

4.3. Level 2

4.3.1. System Overview

The system across the entire robot consists of the main subsystems: processor, power management, motor controller, shooting mechanism, sensor network, and wireless controller. The processor interacts with every subsystem and receives inputs from all sensors. The puck shooting mechanism and motor controller will not be powered by the processor but will receive logic-level inputs from the processor. The System Overview Level 2 diagram is shown below in Figure 5.

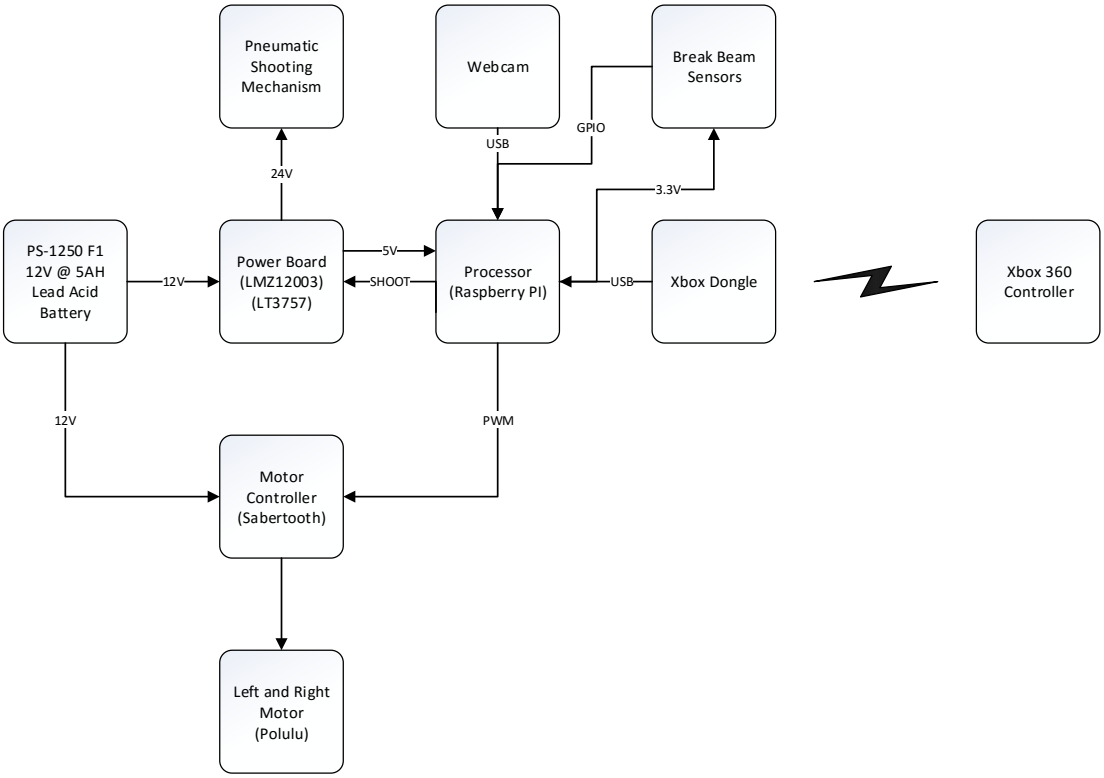


Figure 5: System Overview Level 2 Block Diagram

4.3.2. Sensor Network [KM]

Candidates for possible sensors included infrared light sensing switches, pulsed infrared light sensors, and utilizing Wii game system nunchuck infrared cameras. The first two solutions allowed for cheap and simple ways to track an object under certain conditions. The pulsed infrared light sensor ideas were not chosen as viable sensors for this project due to their susceptibility to electrical noise, imprecise manner to distinguish among different beacons, and very sensitive hardware resulting in false readings. The Wii nunchuck camera idea was not chosen due to its range limitation. It would require an extremely bright LED beacon in order to be tracked and would constantly dissipate large amounts of power on the order of 2 Watts per LED module.

After considering the previous choices as the primary sensor for the robot, the use of a camera with accompanying image processing will be a more dependable and effective senor. The camera connected to the processor will utilize a lens with a short focal length to yield a wider field of view very similar to a backup camera on many new motor vehicles. Digital camera filters will most likely be required to decrease distortion of the chosen wide camera lens and decrease motion blur while the robot is in motion. Through the use of image processing, it is possible to convert the original wide-view camera input to a standard image. [4] The lens choice will be a full-frame fisheye lens, rather than the heavily distorted circular fisheye lens.

OpenCV image processing software is a possible software candidate to utilize with a camera on the robot. OpenCV is open source software that runs in C++ that has all of its libraries in the public domain. MATLAB software, after including plug-ins, is also capable of image processing for this application, but its programs can only be run in MATLAB which requires a software license. For these reasons, the robot will be developed using OpenCV, particularly due to its versatility. The implementation of a wide-view lens coupled with OpenCV software greatly increases the camera's field of view and will allow the robot to more easily detect objects in front of it.

The camera will be placed on top of the robot facing the front. It will receive power and transmit video from the Raspberry Pi over USB. The camera will constantly be looking for the beacons by searching for predefined shapes and colors. The software will record where the beacons are located within each image. After finding the x-coordinate for each beacon, the degrees the robot must rotate until the beacon is directly in front of the robot is determined. When the user presses a button on the wireless controller, the software will rotate by the predetermined degree amount and face the desired beacon. For example, one of the beacons will be circular in shape while also being distinctly colored green.

Below in Figure 6 is the Level 2 image processing hardware block diagram, showing the main construction and implementation of an image processing sensor network on the robot.

Figure 6: Image Processing Level 2 Hardware Block Diagram

Table 12: Image Processing Level 2 Input-Outputs

Module	Image Processing
Inputs	5V Regulated Power
Outputs	Image Feed to Processor
Functionality	Sends Video Feed to Processor to Locate Potential Object to Send the Puck Towards. The Lens is Equipped with a Fish Eye Lens to Increase its Field of View

Table 13: Motor Control Algorithm Using PWM Module

Module	Motor Control Algorithm Using PWM Module
Inputs	Manual User Input and Desired Angle to Rotate Motors
Outputs	Processor Sends PWM to the Motor Controller to Move and Rotate the Robot
Functionality	Interpret the User's Commands and Rotate to Find the Object to Send the Puck Towards Using Image Processing

Table 14: Software Image Shape Search Algorithm

Module	Software Image Shape Search Algorithm
Inputs	Live Camera Feed
Outputs	Desired Angle to Rotate Motors
Functionality	Instructs the Motor Controller How Much the Robot Needs to Rotate in Order to Align Itself With the Puck Recipient

The sensor network itself is responsible for providing stable, software-assisted control of the robot. The user is able to use standard manual control of the robot, but the user has the added functionality to use the image sensing system to locate the object automatically. The image processing algorithm will constantly be looking for the two beacons. When the shoot button is pressed, the robot will find the last known location of the beacon and send the puck in that direction. The software will check that the robot has possession of the puck using an infrared break-beam sensor prior to the robot searching for the puck. If the robot does not have the puck, it will continue to operate manually as if the user did not press a button to launch the puck. The intended camera for use in the project is a high resolution USB camera with a wide 120 degree fisheye lens shown in Figure 7. The fisheye lens will require the image processing algorithm to convert the fisheye image to a standard image format. [3]

Figure 7: 5MP USB Camera with 120 Degree Lens (PN: Genius Widecam F100)

4.3.3. Wireless Controlled Communications [TB]

The wireless controller will implement a 2.4GHz transmitter to receiver for controlling the movement of the robot and launching the puck. The 2.4GHz band is open for unlicensed use and also allows for an expanded amount of channels for control. A standard 2.4GHz, 6-channel wireless transmitter can be used as a controller input, and this allows for flexibility in control options as any other 2.4GHz transmitter can be interfaced with the controller using a proper receiver and software integration. The controller will take inputs for turning left and right, moving forwards and backwards, and shooting the puck.

Many 2.4GHz options exist for wireless control and communication including Wi-Fi, Bluetooth, and the aforementioned wireless transmitter. Bluetooth and Wi-Fi communication both use a standard 802.11 data link but Bluetooth transfer rates are significantly lower than Wi-Fi, 24Mbps compared to 250Mbps [6] [7]. In actuality, the data transfer rates of Wi-Fi and Bluetooth with minimal interference are closer to 11Mbps for Wi-Fi and 723Kbps for Bluetooth [3]. Using Wi-Fi communication would require setting up a wireless access point on the robot and much more power for a system that should need little power to operate, and it would be much more suitable for a data heavy wireless interface rather than wireless controls. Bluetooth on the other hand, with its lower data transfer rate uses less power and has a shorter range of operation. The use of an analog Radio Controlled wireless transmitter would be the simplest control solution, allowing a receiver to be connected directly to the processor and deliver analog voltage control for smooth operation of the robot. Off-the-shelf wireless transmitters use tactile switches instead of pushbuttons for additional inputs which would not be conducive to passing and shooting control. This results in the option of using a 2.4GHz wireless Xbox 360 controller, which has joysticks as well as trigger and button controls. This would allow for intuitive control of the robot's movement as well as passing and shooting.

The means by which the 2.4GHz wireless controller operates with little interference is called frequency hopping spread spectrum (FHSS). The 2.4GHz band is divided into channels between 2.4 and 2.48GHz and this allows the transmitter and receiver to change frequency channels when experiencing interference [8]. As

compared to lower frequency wireless controllers, a 2.4GHz communications system will allow very little interference when operating the robot [9].

In order to set up a communications interface with an Xbox 360 controller wirelessly, a USB dongle will be connected to the processor board, as shown in Figure 8 below. A Raspberry Pi processor requires the proper drivers to recognize the controller wirelessly and compute the digital inputs from the controller [10]. This can be done by repurposing a Linux joystick/pad driver for Xbox controllers [11] [12]. The processor will also be used for motor control so that the direction input from the user will be processed directly with the motor controllers. Using a board other than a Raspberry Pi, such as a Pic or Arduino, may not provide enough processing power to handle both the controller and motor in one interface. Using an Arduino in particular would work for the receiver interface alone but processing the controls would still require a second processor on the robot, so unless the processors are computed in parallel this would not be a fully wireless solution [13].

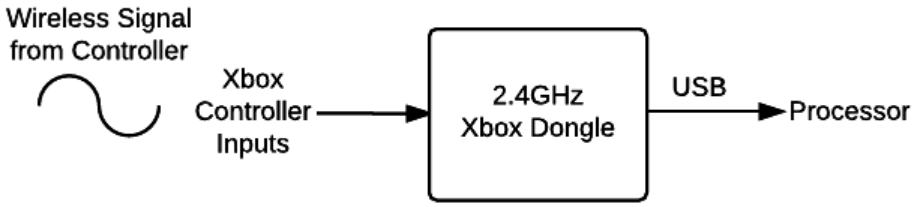


Figure 8: Wireless Controlled Communication Level 2 Diagram

Table 15: 2.4GHz Xbox Controller / Dongle Level 2 Input-Outputs

Module	2.4GHz Xbox Controller/Dongle
Inputs	User Input Control Left/Right Joystick Movement Button/Trigger Pass and Shoot
Outputs	Digital Control Signal
Functionality	Controller for driving robot and passing/shooting puck

4.3.3.1. Wireless Controller Preliminary Testing

The controller will drive the robot using tank-controls. This means that the left and right joysticks on the controller will drive the left and right motors respectively. The joysticks use potentiometers to output a voltage based on the direction and force they are pressed. The controller uses an ADC to convert the voltage output to a positive or negative digital integer which is then transmitted to the processor. Preliminary testing of the controller and Xbox controller driver with a Raspberry Pi board demonstrated the inputs could be transmitted and received wirelessly and

showed the sensitivity and range of the controller. Running the Xbox controller driver and viewing the inputs in the Raspbian terminal showed pushing the sticks forward and backward vary the digital input from -32768 to +32767 approximately. The left joystick is read as X1, Y1, and the right is X2, Y2. These digital inputs can then be interpreted by the processor and mapped to the motor controller for driving. The motor drive algorithm will take the joystick inputs and output them as PWM signals for the left and right motors as shown in Figure 14. The Raspberry Pi has hardware GPIO on pins or a UART TX pin which can be used for PWM control of the motors. Future testing of the motors and controller hardware will allow a proper duty cycle to be selected for PWM.

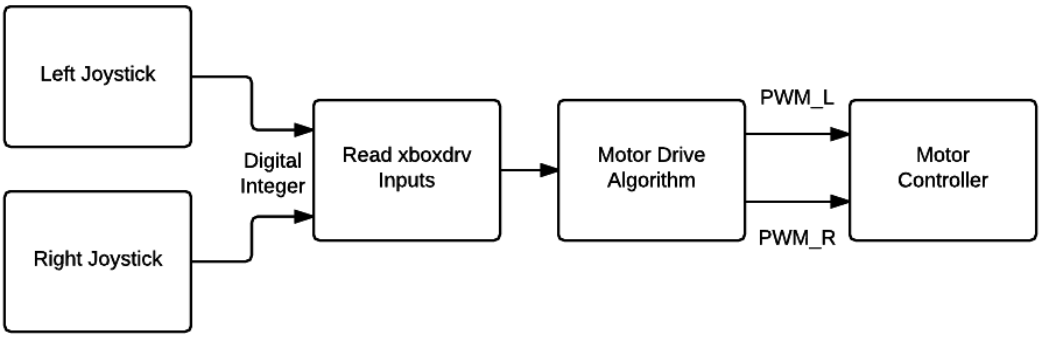


Figure 9: Wireless Controlled Communication Level 3 Diagram

Table 16: Xbox Controller Driver

Module	Xbox Controller Driver
Inputs	Digital Control Signal
Outputs	Digital Control Signal
Functionality	Interfaces Xbox controller hardware with processor software

Table 17: Motor Drive Algorithm Input-Outputs

Module	Motor Drive Algorithm
Inputs	Digital joystick mappings
Outputs	Angle/Distance Calculations for Movement Current to Motor
Functionality	Drive motors for robot navigation

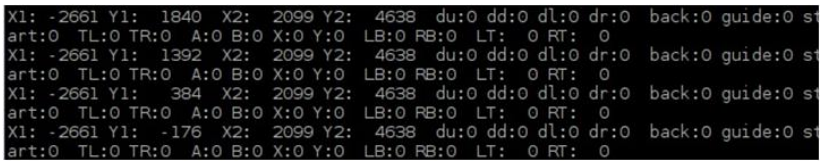


Figure 10: Controller Inputs in Raspbian Terminal

Other controller inputs include the directional pad buttons, left and right triggers and bumper buttons, and the letter-face buttons A, B, X, and Y. The processor will read the input values from the controller driver and the targeting and shooting algorithm will receive button inputs for their respective actions. The joystick values will be assigned to the motor PWM signals, and the software will implement an "if" statement loop to drive the motors. The processor will only look at the y-values of the joystick inputs and will look for positive values to drive the motors forwards and negative values for backwards. Combining these parameters with an "if" loop will allow the robot to be tank-controlled. The pseudo code for the controller driving implementation is shown below.

4.3.3.1.1. Pseudo Code

```
main(void){
 read_xboxdrv();
 drive_motors();
}

drivemotors(int motor_select, int y){
if(leftmotor && y>0)
move_leftmotor_forward;
if(leftmotor && y<0)
move_leftmotor_backward;
if(rightmotor && y>0)
move_rightmotor_forward;
if(rightmotor && y<0)
move_rightmotor_backward;
}
```

4.3.3.2. Wireless Controller Implementation

Repurposing the built-in Linux joystick drivers allowed the use of the Xbox controller joysticks as inputs. The joystick/gamepad mapping code can be viewed in Part D.iii. of the appendix. Gamepad buttons were then added to match the face and top "bumper" buttons on the controller. This allowed for a variety of customized button inputs to be used for operations, such as manual shooting control by holding the right bumper button and then tapping a face button to

activate the shooting mechanism as well as shutting down the Pi board by holding all of the face buttons down for five seconds.

The digital integer joystick values had to be scaled to match the PWM values for motor direction and speed. Calculations were programmed to scale the values in real-time while the driver received input from the controller and then transmitted to the motor controller via UART communication, which can be viewed in Part D.iv. of the Appendix. The button inputs were mapped to specific GPIO pins using the WiringPi C libraries and testing showed that a pin could be set high when a button was pressed, thus actuating the solenoid for the shooting mechanism with a 3.3V input.

4.3.4. Shooting Mechanism [JS]

Friction will be the only force affecting the hockey puck when it is projected forward. Friction can be calculated using the following equation, which include the coefficient of friction and mass of the puck.

$$F_{friction} = \mu_{puck} * m_{puck} * g$$

$$F_{friction} = 0.5(0.071\ kg)\left(9.8\frac{m}{s^2}\right) \cong 0.35\ N$$

The standard floor hockey puck is made of polyethylene which has a coefficient of friction of 0.25 and a mass of 2.5 oz. The coefficient of friction is doubled to compensate for different surface material that the puck will be used on. Additionally, since the friction force is the only force acting on the hockey puck after leaving the shooting mechanism, the negative friction force can be set equal to the mass times acceleration. Solving for the acceleration shows that the hockey puck deaccelerates at a rate of $-5\ {}^{m}/_{s^2}$ as soon as the hockey puck leaves the shooting mechanism.

Due to the design requirements the hockey puck is required to be fired up to 25 feet. The following equation can be used to calculate the velocity that the hockey puck needs to achieve before leaving the shooting mechanism.

$$V_f^2 = V_i^2 + 2\alpha\Delta S$$

$$V_i \cong 9\ {}^{m}/_{S}\ (30\ {}^{ft}/_{s})$$

To meet the requirement of 25 feet the puck will need to leave the shooting mechanism with a velocity of at least $30\ ft/s$.

A pneumatic cylinder system will easily obtain the required speeds to project the hockey puck forward. A small pneumatic system is sized to meet the weight and size specifications while having the capabilities of shooting the hockey puck forward. The pneumatic system consists of a storage or reservoir of high pressure gas that is regulated down to a usable level for the cylinder. To control the actuation of the pneumatic cylinder a solenoid valve is used to control the intake and exhaust of the gas into the cylinder. The following in Figure 11 is a 3D model of the aforementioned pneumatic system.

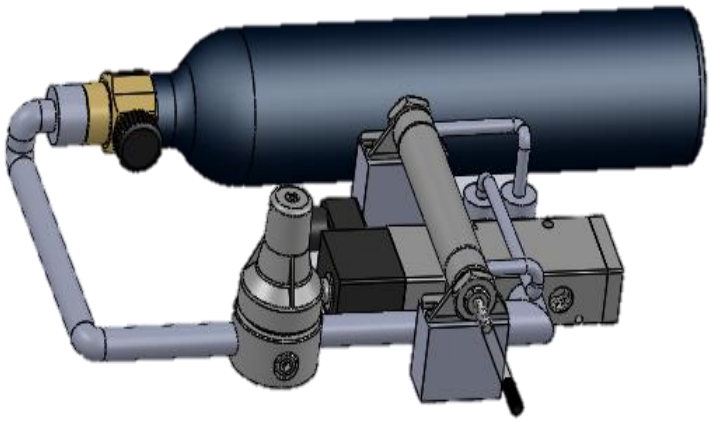


Figure 11: Shooting Mechanism CAD Model

A pneumatic cylinder's velocity is dependent on the volume flow rate forced into the cylinder. The following equation relates the volume flow rate to cylinder velocity.

$$C_v = \frac{Area * Stroke * A * C_f}{Time * 29}$$

Where $Area = \pi\ x\ Radius^2$, $Stroke = Cylinder\ Travel\ (in.)$, $A = Pressure\ Drop\ Constant$, $C_f = Compression\ Factor$, and $Time = In\ Seconds$. From previous calculations the required cylinder speed is 30 ft/s, a standard piston area of 0.44 square inches, and a stroke length of two inches. Additionally, the pressure drop constant and compression factor are pulled from the following chart with an inlet pressure of 60 psi. [28]

Inlet Pressure (psi)	C_f Compression Factor	"A" Constants for Various Pressure Drops		
		2 psi ΔP	5 psi ΔP	10 psi ΔP
10	1.6		0.102	
20	2.3	0.129	0.083	0.066
30	3.0	0.113	0.072	0.055
40	3.7	0.097	0.064	0.048
50	4.4	0.091	0.059	0.043
60	5.1	0.084	0.054	0.040
70	5.7	0.079	0.050	0.037
80	6.4	0.075	0.048	0.035
90	7.1	0.071	0.045	0.033
100	7.8	0.068	0.043	0.031
110	8.5	0.065	0.041	0.030
120	9.2	0.062	0.039	0.029

Figure 12: Pneumatic constants [28]

Therefore, the following calculation can be concluded:
$C_v \approx 80\ \ cubic\ feet/min$

The required volume flow rate to move the cylinder at 30 ft/s is $6\ cubic\ feet/min$.

The following in Figure 13 is a hardware input-output block diagram to better depict the shooting mechanism subsystem.

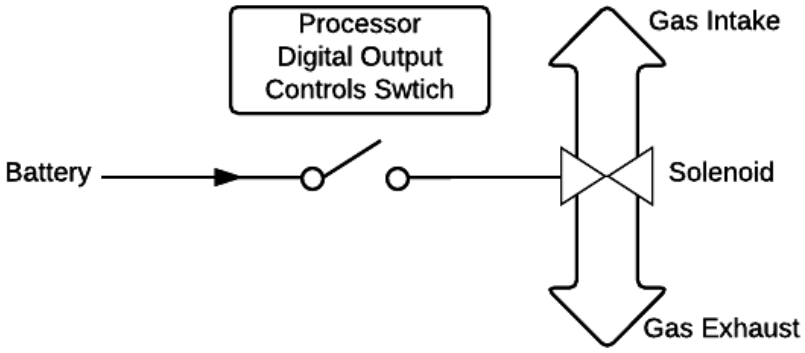


Figure 13: Shooting Mechanism Level 2 Block Diagram

Table 18: Shooting Mechanism Level 2 Input-Outputs

Module	Shooting Mechanism
Designer	John Supel
Inputs	Battery power Fire command from Processor
Outputs	Solenoid Valve Position
Functionality	The shooting mechanism will take power from the battery to change the position of the solenoid valve. The position of the solenoid valve controls the direction and volume of airflow to the pneumatic cylinder.

The pneumatic system will consist of the following components that will be connected together to work as a single system. [24]

Table 19: 9oz CO2 Tank Specifications

Component	5 oz. CO2 Tank
Inputs	Pressurized CO2 gas
Outputs	High pressure CO2 gas
Functionality	The CO2 tank is a high pressure reservoir that can be released on demand.

Table 20: Paintball Tank to Pneumatic Prop Regulator Specifications

Component	Paintball Tank to Pneumatic Prop Regulator
Inputs	High pressure CO2 gas from tank
Outputs	Adjustable output from 0 to 60 psi
Functionality	Brings the high pressure gas to a steady and safe pressure of 60 psi.

Table 21: 4-way 5-port valve with 1/4 inch ports Specifications

Component	4-way 5-port valve with 1/4 inch ports
Inputs	Low pressure CO2 from regulator
Outputs	Intake and exhaust hoses to cylinder
Functionality	The dual acting cylinder needs gas pressure to extend the shaft and pressure into the opposite direction to return the shaft. In the off position the cylinder will be pressurized to compress the cylinder. In the on position the cylinder will extend outward.

Table 22: ¾ inch bore double-acting universal mount cylinder Specifications

Component	¾ inch bore double-acting universal mount cylinder
Inputs	Air port to extend the cylinder
Outputs	Air port to retract the cylinder
Functionality	The cylinder has two ports that will be connected to the solenoid valve. The solenoid valve will control the extraction and retraction of the cylinder.

4.3.4.1. Shooting Mechanism Testing

The designed pneumatic system is assembled and tested. The assembly of the pneumatic system went just as planned with only slight leaking at some threaded connections. Leaking is easily fixed by applying thread sealant at each connection. Through testing many flaws in the design are underestimated and neglected. The neglected factor into the pneumatic design is the maximum air flow through orifices or ports. Using a cylinder with a 1/8 inch NPT port, the maximum airflow capable is 16.9 CFM [29]. It is clearly visible that this restriction is well below the required air flow of 80 CFM. To compensate for this restriction a much smaller bore

cylinder is used. By using an 8mm bore cylinder the required airflow is decreased to 15 CFM. Unfortunately, an 8mm diameter bore cylinder has even smaller intake ports, thus decreasing the allowed airflow into the cylinder. This issue persists due to the fact that our application is unique, which would require a custom cylinder to be fabricated to meet our specifications. Speed testing for the 8mm bore cylinder proved this issue as a max speed of 5 ft/s is achieved, which is much lower than 30 ft/s.

4.3.5. Main Processing Unit [KM]

The entire robot system is dependent on using the sensor for the robot. Since image processing is the chosen sensor for this project, the only way to keep up with its computations is to use a full-scale computer processor rather than a microprocessor. After considering incorporating an image processing system on the robot, the Raspberry Pi ARM processor is a logical choice to work with image processing and motor drive control. The Raspberry Pi 2 pictured in Figure 14 will be the development processor for the robot.

Figure 14: Main Processor ARM V7 on Raspberry Pi

The entire concept of using a processor brings up the questions of whether it is possible to make the printed circuit board or buy an off-the-shelf development board. Given the time constraints of the project and flexibility with funds, the option to purchase a development board will be taken. It is more beneficial to purchase something that works than to construct and debug the hardware of a printed circuit board that is fabricated.

The processor will be required to receive manual control commands wirelessly and provide inputs to the motor controller. It will also need to be conducting image processing on the camera video feed while receiving user input commands from the wireless controller. The Raspberry Pi code will need to contain an interrupt service routine to control the motors when the processor receives commands from the wireless controller; the image processing algorithm will be constantly running while the robot is in use [5]. The Pi should be able to stop image processing, send commands to the motor controller over PWM after receiving commands from the wireless controller. The camera will be connected to the processor over USB. A FIFO

buffer will be implemented in order to keep a stable image stream ready for importing images using the image processing algorithm. A buffered image stream will provide a simple way to mitigate data loss and keep data ready for processing. The user will be able to press a button on the controller to let the robot know which beacon it should launch towards. The robot will then automatically rotate to locate the beacon and send the puck in that direction. If the robot does not locate the beacon after the robot rotates for a while, then the robot will return to manual control. If the user does not press a button, then the robot motor control system will only accept commands from the controller. The hardware overview diagram in Figure 15 shows the system overview in more detail.

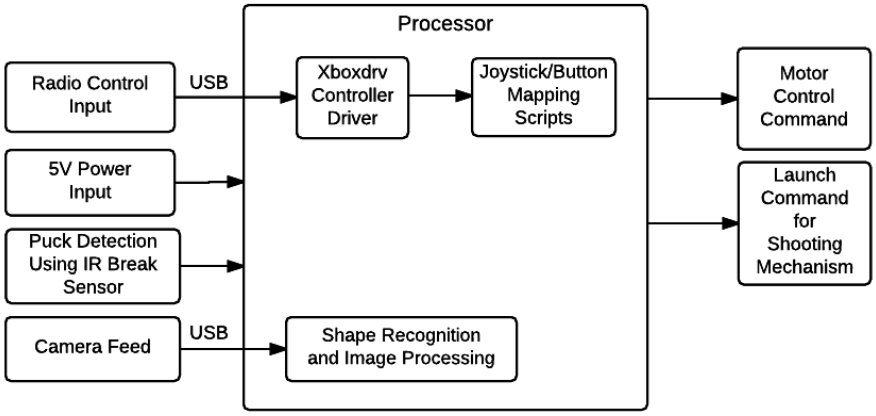


Figure 15: Main Processing Unit Level 2 Block Diagram

The main processor receives inputs from the camera, infrared puck detector and wireless controller. The processor will also command the motor controller to control the motors after interpreting the input from the wireless controller. When the user commands the robot to launch the puck, the robot will verify that it has the puck prior to launching the puck to help preserve pressure in the CO_2 tank. If the robot has possession of the puck and the user gave the command to launch the puck, the processor will command the shooting mechanism to launch the puck.

The software for the robot is broken up into multiple sections. The system can be described in a top level software diagram as the one shown in Figure 16.

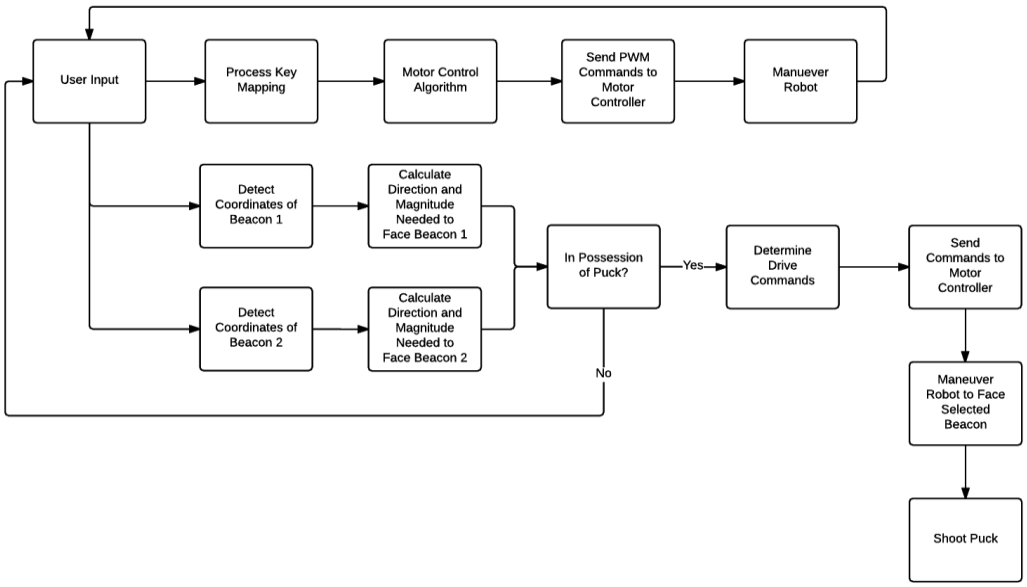


Figure 16: Main Processor Software Overview Level 2 Diagram

Figure 17 shows the algorithm run each time the user provides input to the wireless controller for the user input stage of the software.

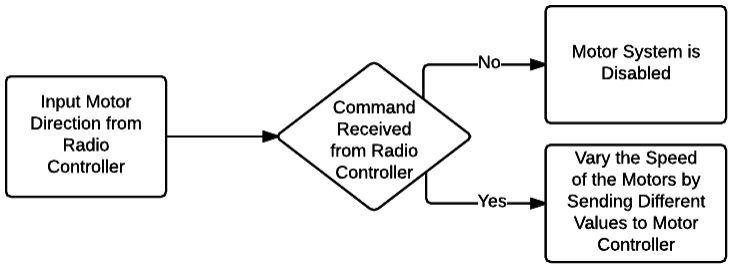


Figure 17: Main Processor to Motor Controller Software Level 2 Diagram

In the OpenCV V3.0.0 image processing section of the software, the fisheye camera video input will have to be converted from a distorted, fisheye image to a standard linear image. [3] The code will use OpenCV libraries to track the beacons using their known colors and shapes. The beacon detection algorithm will locate the x-coordinate of each beacon for each image and approximate their angle to a section of x-coordinates, such as the depiction in Figure 18.

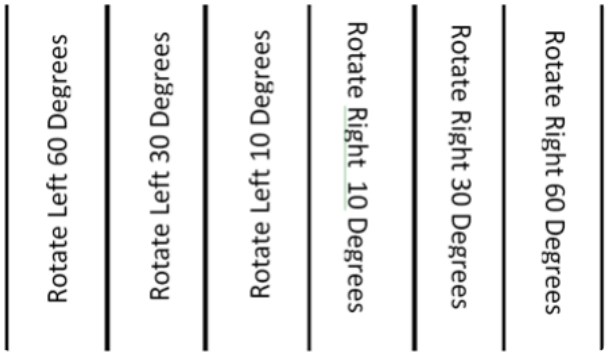


Figure 18: Software X-Coordinate Sections Level 2 Diagram

Once the robot has verification that it has possession of the puck from the IR break-beam sensor, the robot will begin tracking the x-coordinates of all the beacons. Each beacon will be within one of the sections determined above and the robot will rotate. The software iteration can be seen in Figure 19.

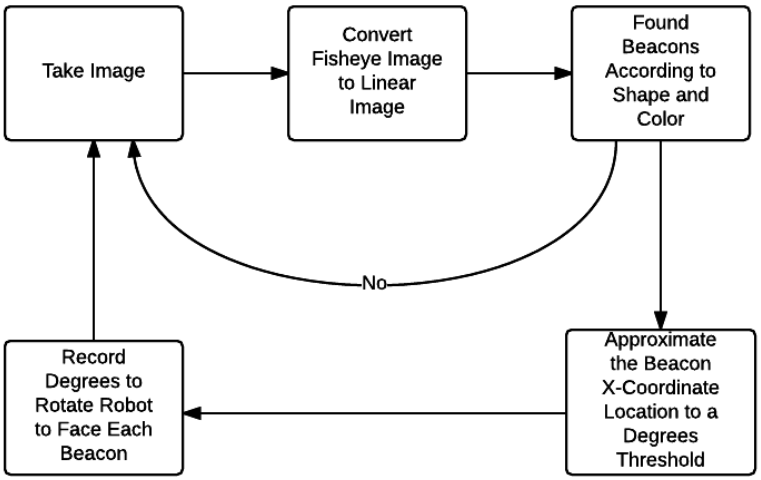


Figure 19: Software Image Processing Algorithm Level 2 Diagram

Table 21: Image Processing Algorithm Level 2 Inputs-Outputs

Module	Image Processing Algorithm
Inputs	Captured Image
Outputs	Degrees for Robot to Rotate
Functionality	Sends Degrees to Rotate to the Motor Control Algorithm

Figure 20 below shows the software overview of the image processing system. Since the camera input utilizes a fisheye lens, the image borders will need to be corrected to look like a standard image. Next, the image will be stored into a First-Input-First-Output buffer; this will allow for the image processing algorithm to retrieve images from the buffer without losing image frames. The image processing algorithm will constantly search the images for the beacons and their respective shapes and colors. When the beacons appear in the image, their x-coordinates are recorded. At the same time, their approximate degrees from the center of the image are calculated using the image section that they appear in as shown in Figure 18 above. When the user presses a button that represents one of the beacons, the robot will rotate by the approximate degree value and immediately send a logic-level launch command to the shooting mechanism to shoot the puck.

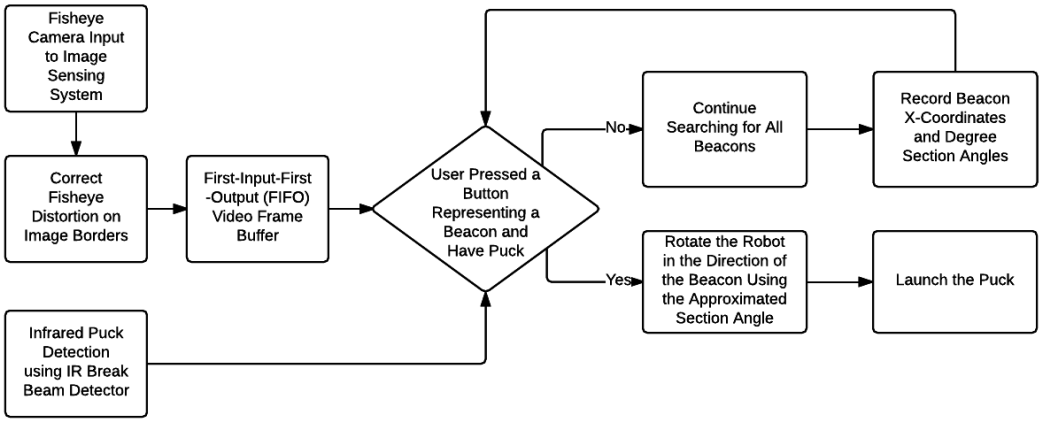


Figure 20: Software Image Processing Overview Level 2 Diagram

After the user presses a button to shoot the puck to a beacon, the processor will send a PWM signal to the motor controller. The PWM duty cycle will control the speed the robot as it rotates in tank drive; the rotational acceleration will be optimized through testing. The robot will send a tunable PWM signal control to align with the intended degree value from the image processing algorithm. The motor control algorithm is shown in Figure 21.

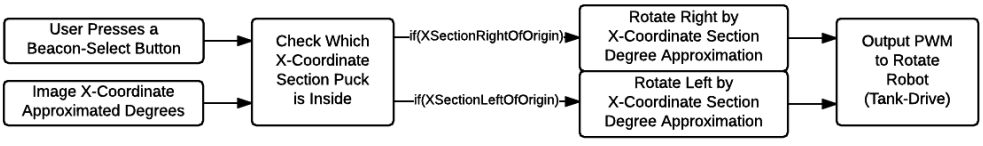


Figure 21: Software-Assisted Motor Control Level 2 Diagram

Table 23: Software-Assisted Motor Control Level 2 Input-Outputs

Module	Software-Assisted Motor Control Algorithm

Inputs	Wireless Controller Beacon-Select Button Image X-Coordinate Approximated Degrees
Outputs	PWM Motor Drive Input Causes Robot to Rotate
Functionality	Motor Control Algorithm Causes Robot to Rotate if the User Utilizes Software-Assisted Mode.

The processor is responsible for carrying out all operations on the robot. The controller gives the user the opportunity to have the image processing algorithm to find the last known position of the targeted beacon and launch the puck in that direction. Using this mode, the user will be able to accurately send the puck towards the desired target. This will also provide a faster reaction time for the robot to aim the puck after the user specifies which beacon to launch it towards. This method is also used because the camera will have "blurry" images while the robot is in motion and will not be able to have dependable feedback for the robot to know when to stop rotating. Having the degrees to rotate to face the beacon predetermined will be a more effective way to align the robot with the beacon.

4.3.5.1. Pseudo Code

The following algorithm illustrates how the system software will be constructed. The processor will be constantly receiving camera images while also being able to receive control input from the user. When the wireless controller sends commands to the processor, the code will run an interrupt service routine where the controller data is recorded and commands are directed to the motor controller. After the ISR is completed, the code will resume at its last location.

```
InitializeCamera_Wireless Controller();
While(1){
        ReceiveCameraFeed();
        LocateBeaconUsingImageProcessing_ShapeColorSizeMatching();
        Switch (StateVariable) {        //FiniteStateMachineEquivalent
        Case UserCommandFromWirelessController: //This will be an interrupt
        service routine
                CommandMotorsToMove();
                Break;
        Case BeaconFound:
                ContinueOperatingRobotButRememberBeaconPosition();
                Break;
```

```
Case UserPushesButtonToLaunchPuck &&
PuckDetectedWithIRBreakSensor:
        CommandMotorControllerToRotateToBeaconLocationAndLaunch()
;
        Break;
}
        }
```

Table 24: Input Motor Direction from Wireless Controller

Module	Input Motor Direction from Wireless Controller
Inputs	Digital Wireless Controller Joystick Values
Outputs	Command Motor Controller How Much to Move Motors
Functionality	Allows the User to Operate the Robot Using Manual Control

Table 25: Input USB Camera Feed

Module	Input USB Camera Feed
Inputs	Captured Ambient Light
Outputs	Camera Frames going to Processor via USB
Functionality	Provides the Input Camera Feed for the Image Processing

4.3.6. Motor Control System [JS]

The motor control system consists mainly of the motor driver and the DC motor. The motor driver is a device that allows a low power digital signal control a high power motor. This useful device easily allows a processor to control the speed and direction of the DC motor.

The following is a hardware input-output block diagram to better depict the subsystem.

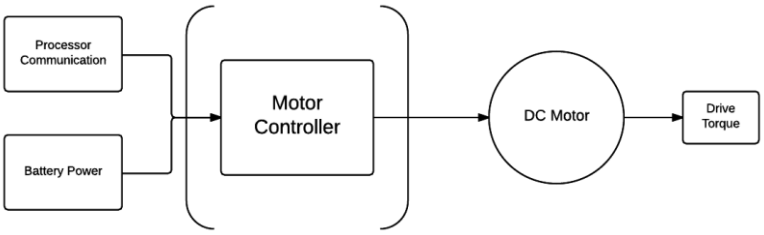


Figure 22: Motor Controller Level 2 Block Diagram

Table 26: DC Motor Level 2 Input-Outputs

Module	DC Motor
Inputs	Voltage and Drive Current
Outputs	Motor Drive Torque
Functionality	Receives the appropriate voltage from the motor controller. Voltage polarity determines the direction of which the motor will spin. Additionally, the motor controller supplies the drive current which directly correlates with the torque the motor exerts.

It is very important to calculate all required values for the motor control system as this is a high power subsystem and can cause serious damage to the rest of the project. The power requirements are determined from the specifications for the DC motors. The following diagram shows the forces acting on the wheel.

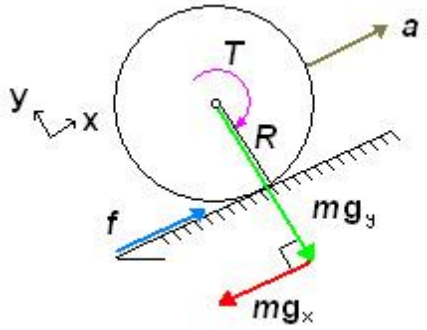


Figure 23: Force Diagram for Wheels

In this situation the robot will travel completely in the horizontal direction, allowing the summation of all forces (F) to equal the friction force.

$$\Sigma F_x = M * a = f$$

The inertia of the robot is ignored as these equations are used to determine the maximum parameters or “worst case values”, which is found when the robot is

initially not moving. The friction force produces the torque (T) on the motor and the following equation can be substituted in.

$$M * a = \frac{T}{R} \implies T = M * a * R$$

The above torque equation represents the total torque required to accelerate the robot. The total torque value must be divided by the total number of drive wheels (N) to obtain the torque needed for each drive wheel.

$$T = \frac{M * a * R}{N}$$

Finally, the efficiency I of the motor, gearing, and slip must be taken into account.

$$T = \left(\frac{100}{e}\right) * \frac{M * a * R}{N}$$

This is the torque required and inefficiencies for each motor. A safe and reasonable value for acceleration is chosen and implemented into the equation.

From the design specifications, the required torque can be calculated. For the robot mass (M), the maximum weight of 15 pounds (6.8 kg) is used. A reasonable acceleration of $4 ft/s^2$ ($1.2192 m/s^2$) is chosen. A standard 4 inch tire is used, so the radius (R) is 2 inches (0.0508 m). Lastly, the design will include two wheels and a rough estimation of 65% efficiency for the mechanical design.

$$T = \left(\frac{100}{e}\right) * \frac{M * a * R}{N} = \left(\frac{100}{65}\right) * \frac{(6.8kg)\left(1.2192 \frac{m}{s^2}\right)(0.0508m)}{2}$$

$$= 0.324 \text{ N} \cdot \text{m}$$

A robust and low-cost gearmotor is chosen as it satisfies the torque and mechanical requirements. This gearmotor consists of a powerful Pololu motor and a 20.4:1 metal gearbox. The following table shows the specifications for the gearmotor [25].

Table 27: 3202 Pololu Gear Motor Specifications

	3203 Pololu Metal Gearmotor
Motor Operating Voltage	12V
Gearbox Output Speed	$480\ rpm\ free\ run$
Gearbox Torque	$T = 0.6\ Nm$
Stall Current	5500 mA

Lastly, the angular velocity for a nominal 12V battery is 480 rpm, which is a reasonable and safe operating speed for the robot.

Table 28: Motor Controller Level 2 Input-Outputs

Module	Motor Controller
Inputs	Battery Power Digital Communication from Processor
Outputs	Voltage and Driver Current
Functionality	Acts as a switch between the battery and DC motor. Capable of setting voltage polarity and pumping motor drive current. Outputs are digitally controlled by the processor.

The motor controller specifications are based on the DC motor, as the controller needs to be capable of supplying what the motor demands. The first consideration is the motor's nominal voltage. The motor controller must be capable of operating within the motor voltage range. The next consideration is the continuous current the controller will need to supply the motor. A "rule of thumb" used in industry is making sure that the maximum current rating is at least double the continuous current of the motor. A nice feature that some controllers have is over current and thermal protection, which would be very helpful in this application. Another consideration is the control method; the motor controller is a tool used to control the power supplied to the motor. A compatible communication protocol must be used between the motor controller and processor. The final consideration is whether to use a single or double motor controller [17].

The motor controller used for the design will be the Sabertooth 2x5 module. The controller has the following specifications and does meet the demand of the motors. [26]

Table 29: Motor Controller Specifications

	Sabertooth 2x5
Operating Voltage	6V – 18V
Continuous Current	5A
Peak Current	10A
Output Channel	2
Protection	Current limit and thermal protection

Figure 24: Sabertooth 2x05 Motor Controller

Due to the lack of test equipment, the motors were not tested. Inserted into the robot the motors and motor controller ran flawlessly even though the final weight of the robot was more than desired.

4.3.7. Power Distribution [DS]

The power distribution network consists of a rechargeable battery delivering the required power to each subsystem. Each subsystem requires a specific voltage and it is imperative to provide each component with a steady supply. A voltage spike could cause serious problems to voltage sensitive integrated components, so it is important to maintain clean and safe voltage.

A linear regulator uses an active (BJT or MOSFET) device controlled by a high gain differential amplifier. It compares the output voltage with a reference voltage and adjusts the active device to maintain a constant voltage. The linear regulator's power dissipation is directly proportional to its output current. Typical efficiency can be 50% or lower, but the noise generated from the linear regulator is much lower than a switching regulator [20].

A switching regulator converts the input voltage by switching the voltage with a power MOSFET or BJT switch. The filtered output is compared to a reference voltage and adjusts the circuit that controls the on and off times of the switch. A typical switching regulator can achieve efficiencies in the 90% range but requires much more filtering on the output. Switching regulators are usually used for high current applications [20].

The following is a block diagram of the power distribution to each subsystem.

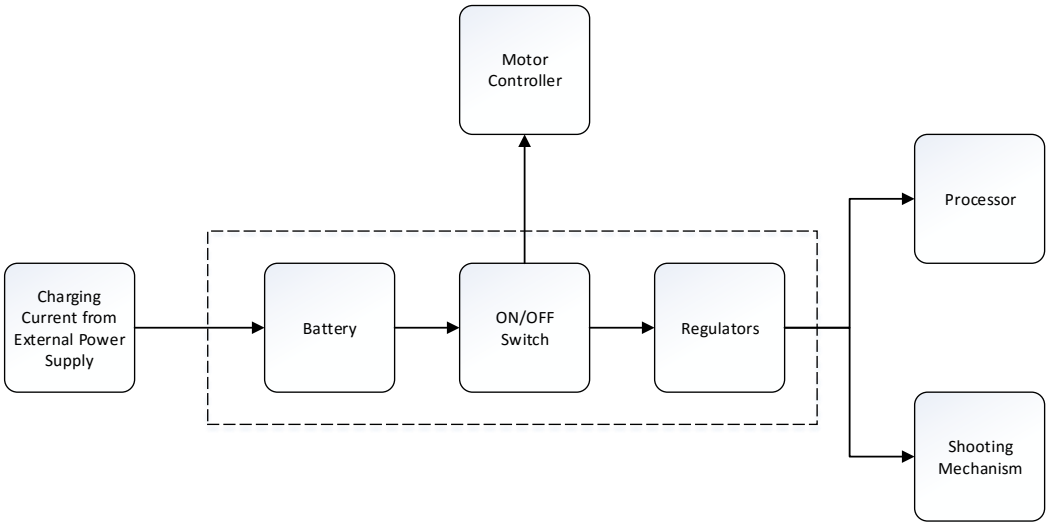


Figure 25: Power Distribution Level 2 Block Diagram

Table 30: Power Distribution Level 2 Input-Outputs

Module	Power Distribution
Inputs	Battery Charging Current
Outputs	Motor Controller Power Shooting Mechanism Power Regulated 5V Regulated 24V
Functionality	The rechargeable battery will require the ability to be recharged after significant use. To protect the system and eliminate any parasitic losses, a mechanical ON/OFF switch and fuse will be implemented. The battery will supply the needed power used for the motor controller and shooting mechanism. Additionally, the battery voltage will be regulated to a lower safe level for the system electronics. Lastly, the battery status will be readout for user's convenience.

4.3.7.1. Power Distribution Implementation

The voltage regulators chosen for this application are switching regulators. By using a switching regulator, the design can be as efficient as possible. This is important since the robot is run off battery power only and operating time is maximized as much as possible. Figure 26 shows the power distribution among the various subsystems.

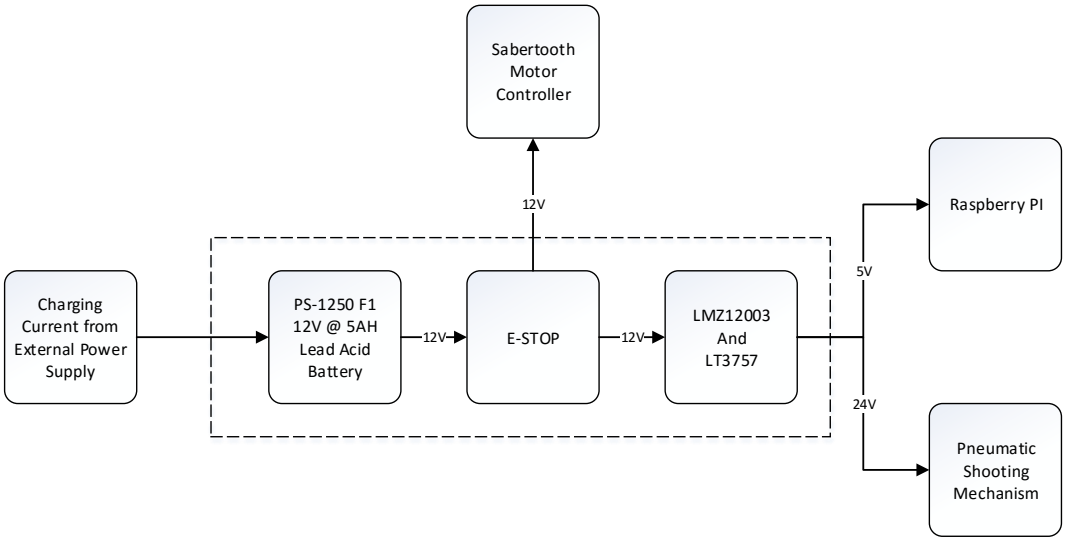


Figure 26: Power Distribution Diagram

The system will consist of one 12V lead acid battery rated at 5Ah. The batteries will be directly connected to a manual E-Stop distributing to the other subsystems. The motor controller requires 12V and solenoid requires 24V for the shooting mechanism to operate. [23]

A Texas Instrument LMX12003TZ switching regulator IC is used in this design for its simplicity and features. The LMX12003TZ is used as the DC-DC converter to provide regulated 5V at a maximum 2A load to the Raspberry Pi. The IC allows variable input voltage up to 20V, constant output current up to 3Aand includes protection against in-rush currents and faults. The LMZ12003TZ requires three external resistors and 4 external capacitors to complete the power solution. A single resistor adjusts the switching frequency up to 1MHz. Figure 27 shows the schematic for the DC-DC switching regulator circuit. The schematic is designed to handle 10-14V input with a constant 5V output for up to 3A at a switching frequency of approximately 440kHz. Calculation for the schematic can be found in Appendix A: 5V DC-DC Convertor Calculations. [21] [22]

A Linear Technologies LT3757 Boost, Flyback, SEPIC and Inverting Controller is used in this design for its features. The LT3757 is used as the DC-DC converter to provide a regulated 24V at a maximum 2A load to the pneumatic shooting mechanism. The IC allows variable input voltage up to 40V and has a programmable soft-start. A single resistor adjusts the switching frequency up to 1MHz. The design is set for a variable input voltage of 8-16V with a switching frequency of 300kHz. Calculation for the schematic can be found in Appendix A: 5V DC-DC Convertor Calculations.

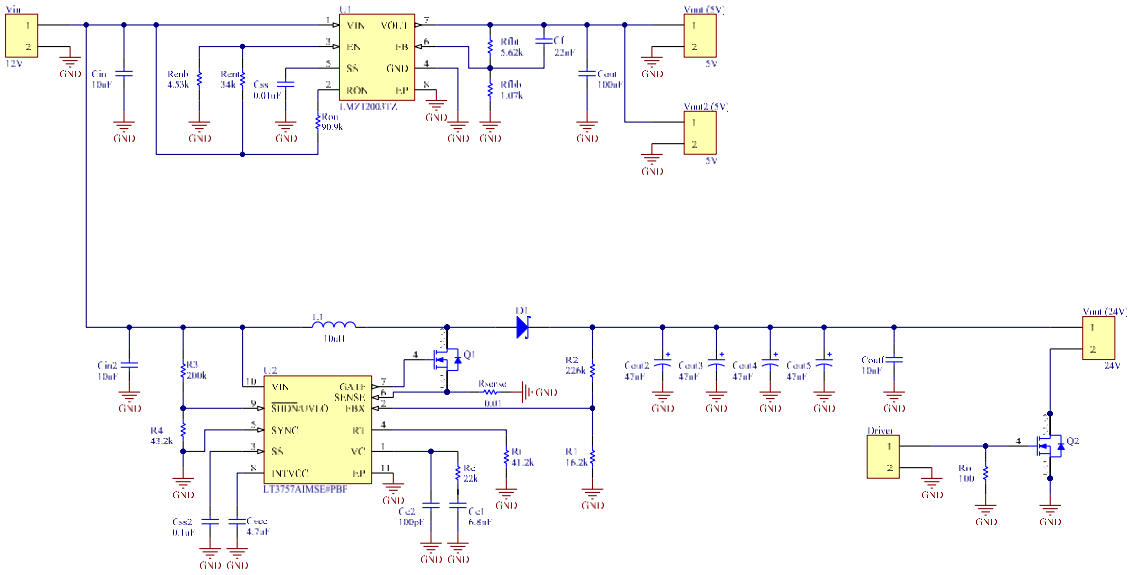


Figure 27: Power Board Schematic

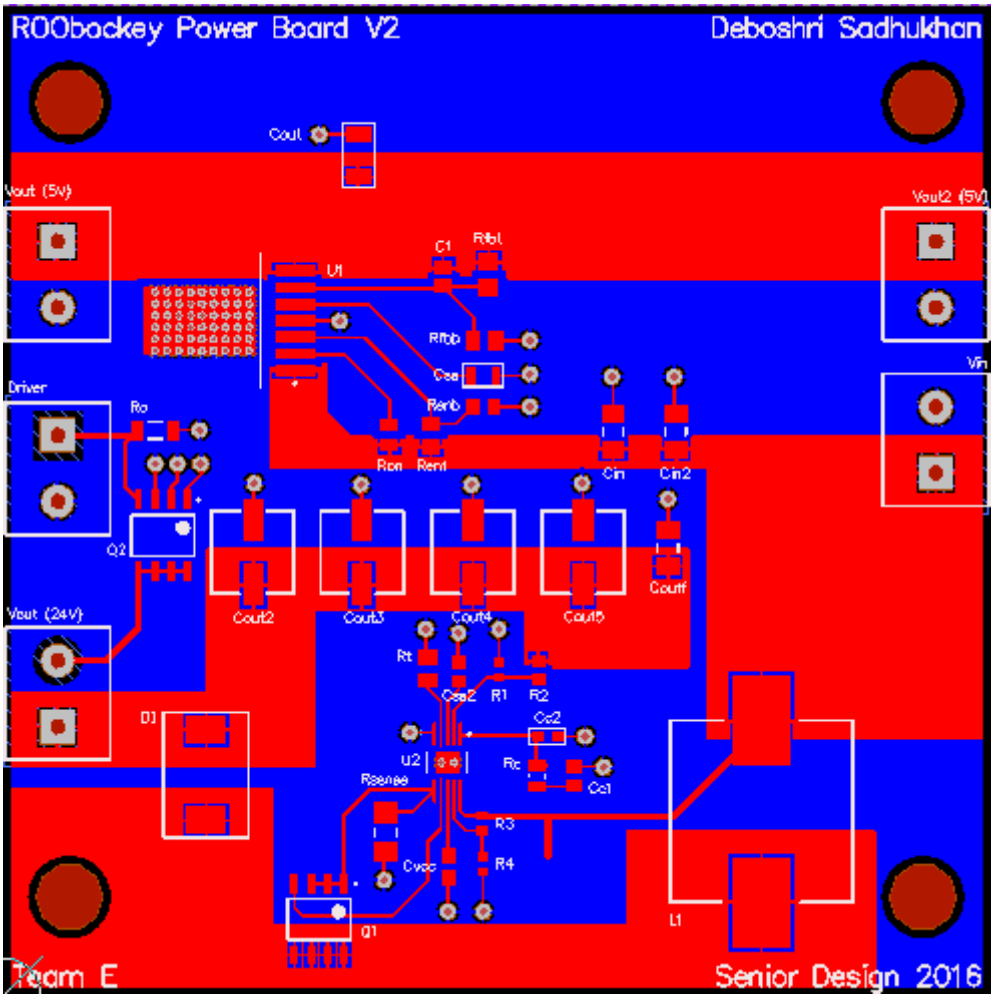


Figure 28: Power Board 2D Layout

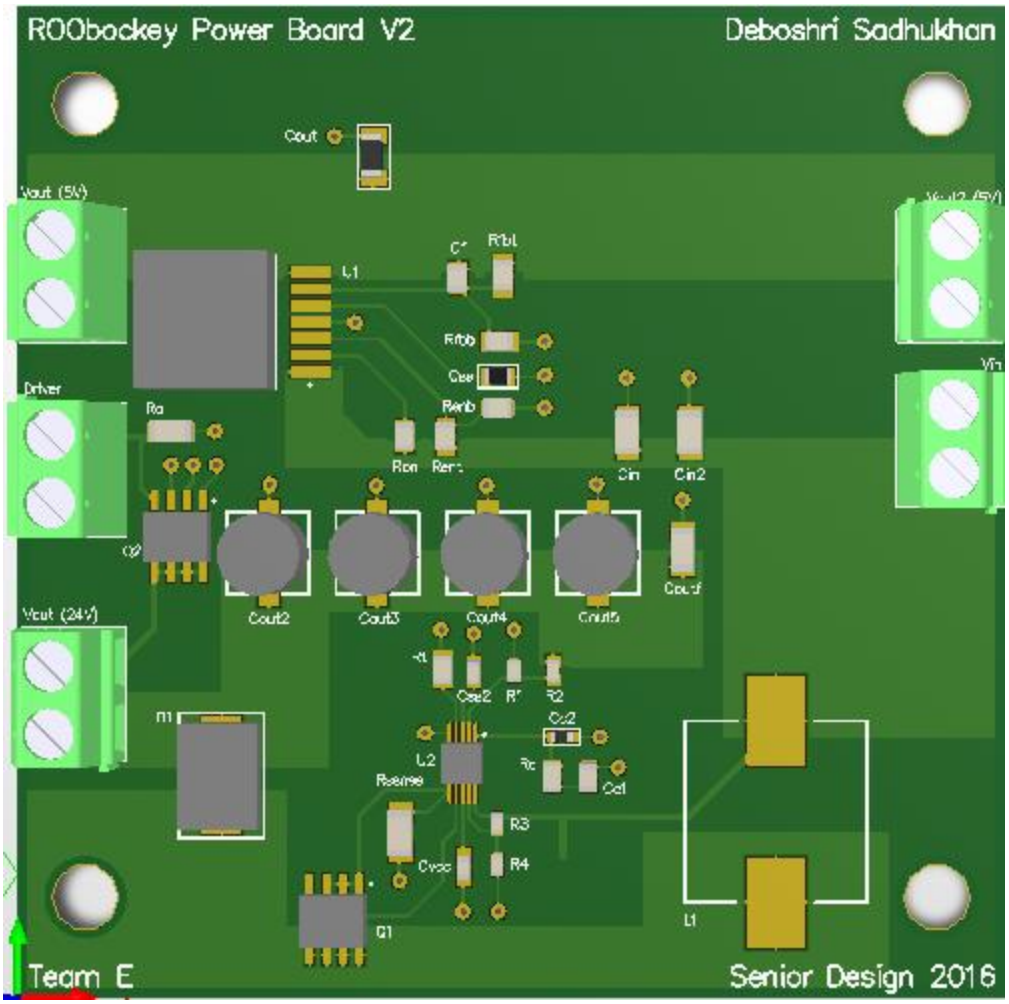


Figure 29: Power Board 3D Layout

4.3.7.2 Power Distribution Testing

The power distribution system went through two revisions of PCB design and testing. The first board purchased showed numerous shorts between power and ground. Upon investigation, the problem was caused by the plating on the connector VIAs being shorted to the ground plane on the bottom side of the board due to manufacturing. While the manufacturing tolerance is 6mils minimum and the boards had 10mils spacing, it wasn't enough. After further modification to the board, the circuits were functional. Below are figures of both board and testing of both revisions of the board.

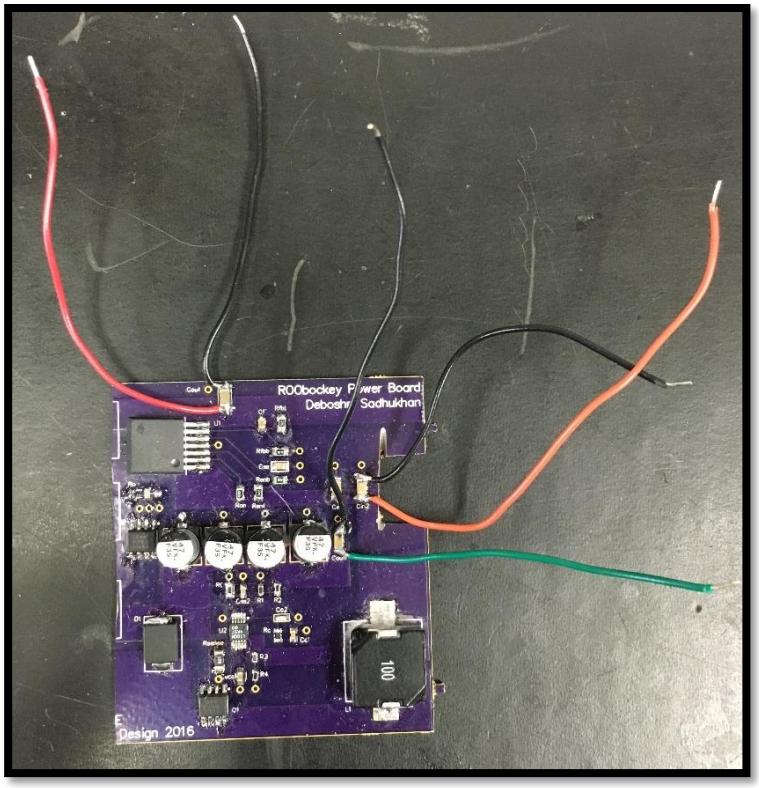


Figure 30: Power Board Rev. 1

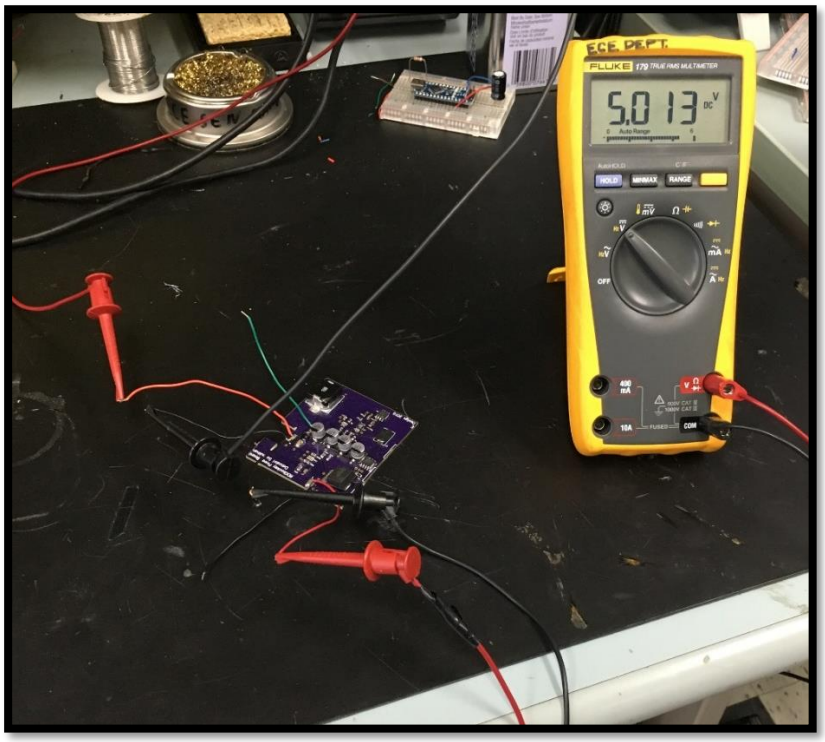


Figure 31: 5V DC-DC Converter Rev.1 Bench Testing

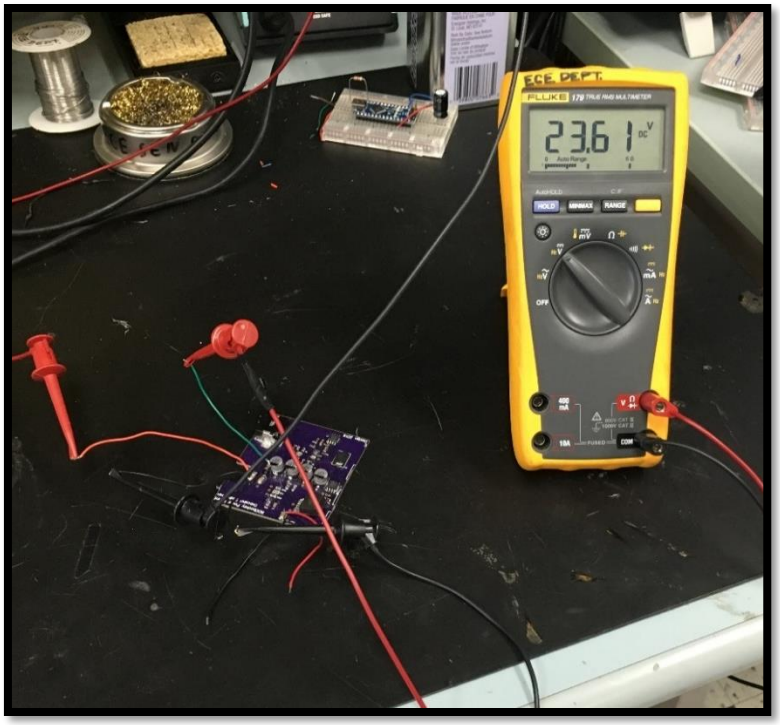


Figure 32: 24V DC-DC Converter Rev.1 Bench Testing

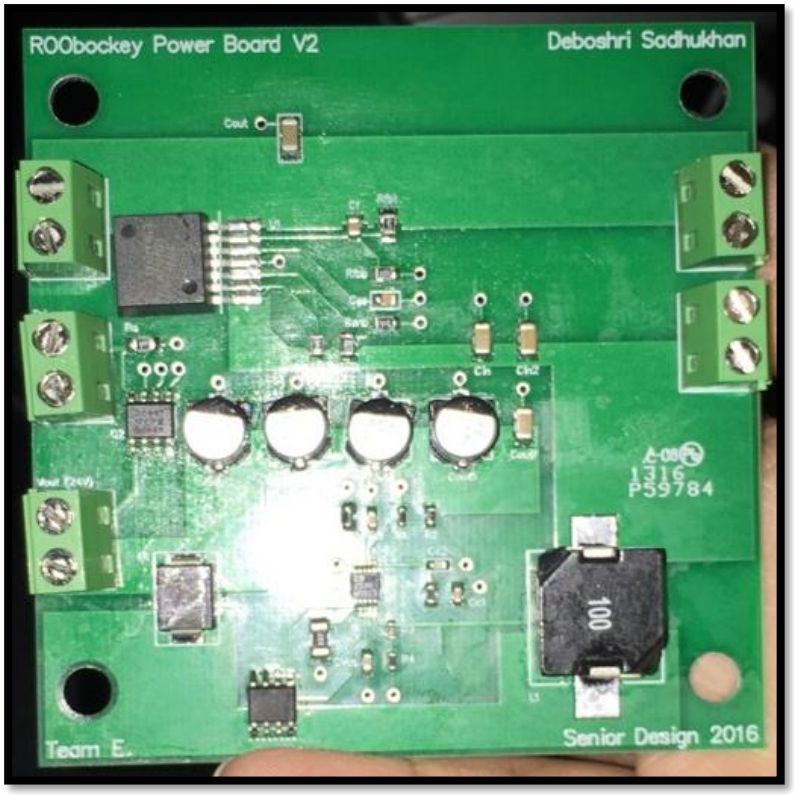


Figure 33: Power Board Rev. 2

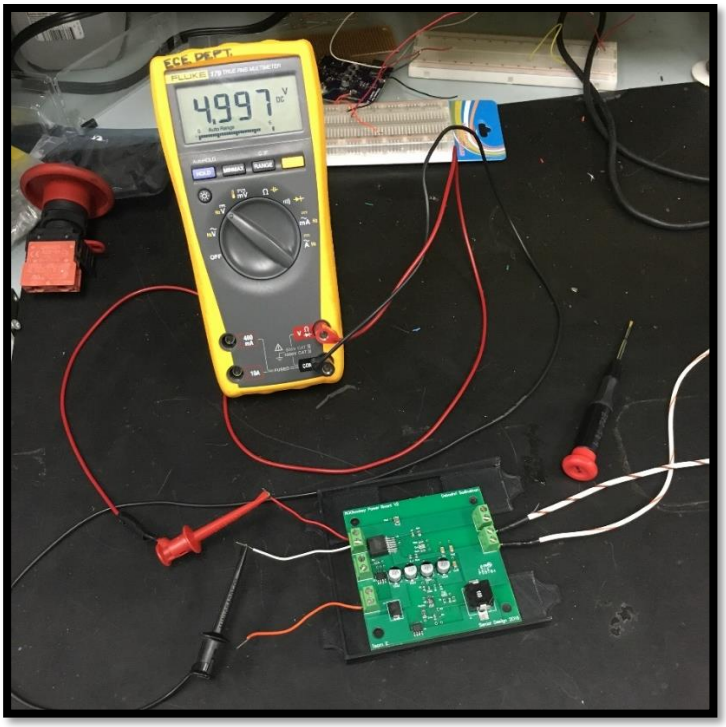


Figure 34: 5V DC-DC Converter Rev. 2 Bench Testing

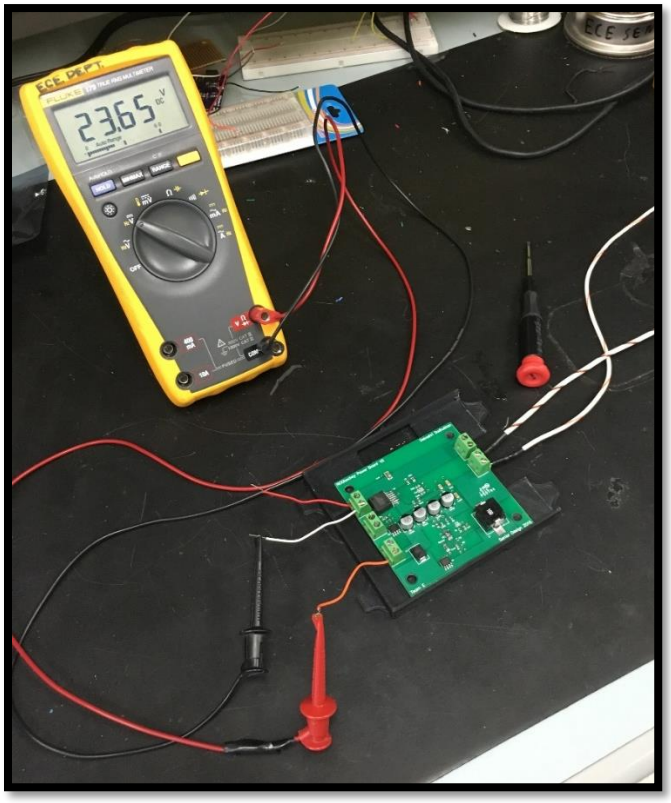


Figure 35: 24V DC-DC Converter Rev 2 Bench Testing

4.3.8. Mechanical Design [JS]

The hockey robot will follow the standard form used in national robotic hockey competitions [15]. The following specifications are used for the hockey competitions and are incorporated into this design.

Table 31: Mechanical Specifications

Mechanical Specifications	
Robot Weight	15 lbs.
Robot Dimensions	18” x 18” x 18“
Shooting Mechanism	Horizontal Projection Only

A 15 lb. limitation is essential, as it allows the design to be lightweight and easily transported by the user. Additionally, the lightweight design lowers the need for powerful motors to maneuver the robot.

Lexan polycarbonate is a lightweight high-impact plastic that is virtually unbreakable [19]. This allows the mechanical structure to be built from this material and keep the weight as light as possible. Additionally, the high-impact plastic allows for robust construction so the user doesn’t have to worry about breaking the outer structure. Lexan polycarbonate has a weight per volume of 0.042 lbs./in^3.

The current design is 18”x14”x3.5”. The following 3D model is a design that shows a robot that efficiently meets the engineering requirements.

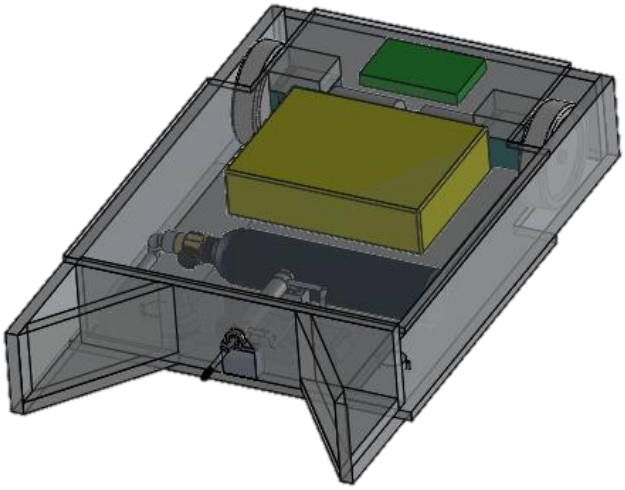


Figure 36: Robot Mechanical CAD Model

The current design incorporates two rear drive 4 inch wheels and two ball transfers in the front to support the weight of the robot. Ball transfers are used in the front of the robot to open up space for the shooting mechanism. Additionally, the body

of the robot is cut out of Lexan polycarbonate to provide a light weight strong structure. The front of the robot involves a wedge design to allow the robot to drive into the hockey puck and align the puck in front of the shooting mechanism.

The finished mechanical design is shown in Figure 37 below. The pneumatic cylinder was moved on top of the lid to free up space in the body for the shooting mechanism, power board, and motor controller. The guides on the front were left open because closing them was deemed extraneous and it made space for wiring the break beam and the shooting mechanism plate. The final build ended up being over the weight spec primarily because of the size of the pneumatic cylinder and battery.

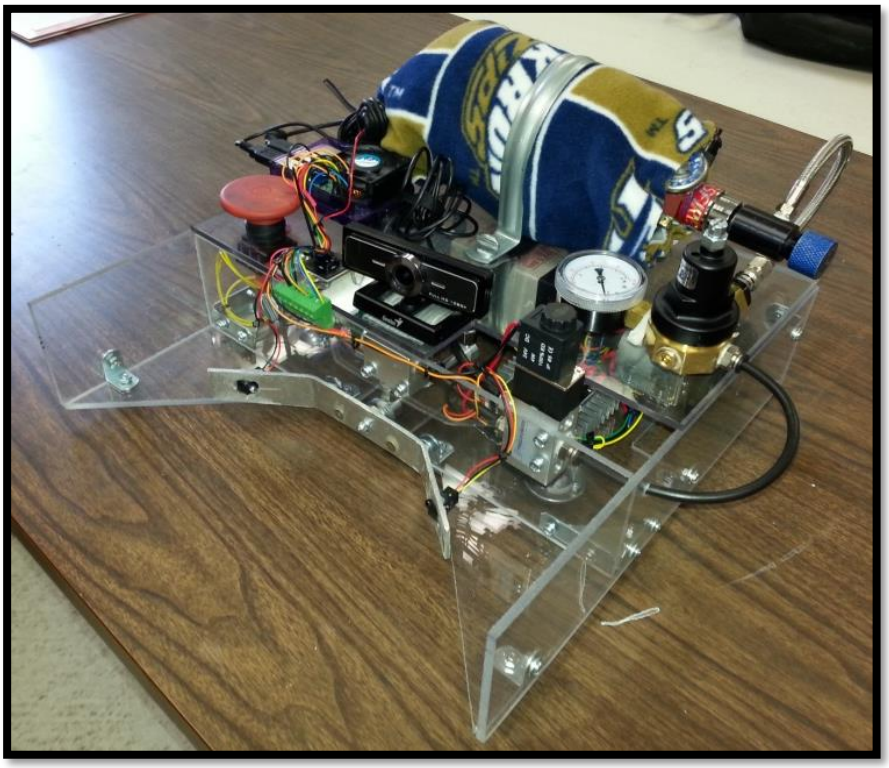


Figure 37: Finished Robot Mechanical Design

4.3.9. Sensor Beacons [DS & KM]

The use of standard objects for beacons results in significant noise in the color recognition portion of the software. Also, shadows were preventing a given beacon from being tracked when light shines on the beacons at various intensities, resulting in the beacon having different HSV color filter values that the intended color filter values. After running into these issues, light emitting beacons are chosen to allow the camera to locate a beacon with relatively constant HSV filter values in several lighting conditions and beacon angles with the camera.

The image processing system requires shape and color recognition of three beacons. These beacons were made out of wood and lined with NEOPixel LEDs. Each beacon runs off an Energizer 6V Lantern Battery connected to an Arduino Nano to program the color. The hardware circuit diagram is shown in Figure 38 as well as the Beacon Assembly in Figures 39 and 40.

The beacons used as the detected objects for the image processing is consist of a single Atmel 328p microcontroller on an Arduino Nano board. The Arduino communicates with Neopixel LEDs on the beacons using a 1-wire protocol. The Neopixel library from Adafruit allows the Arduino software to communicate with the Arduino. The Software is attached in "section C.i." in the Appendix. Figure 35 shows the beacon in operation.

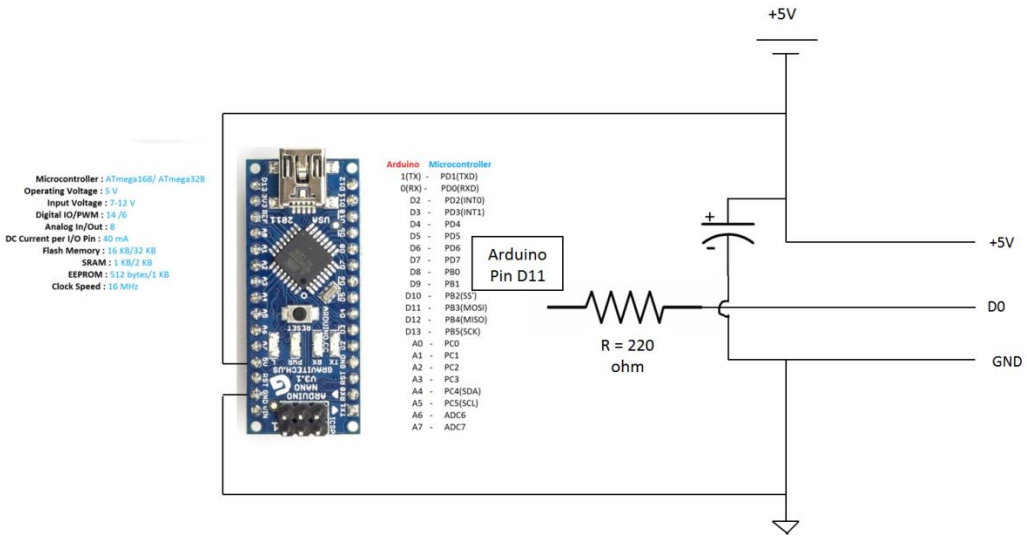


Figure 38: Beacon Hardware Schematic

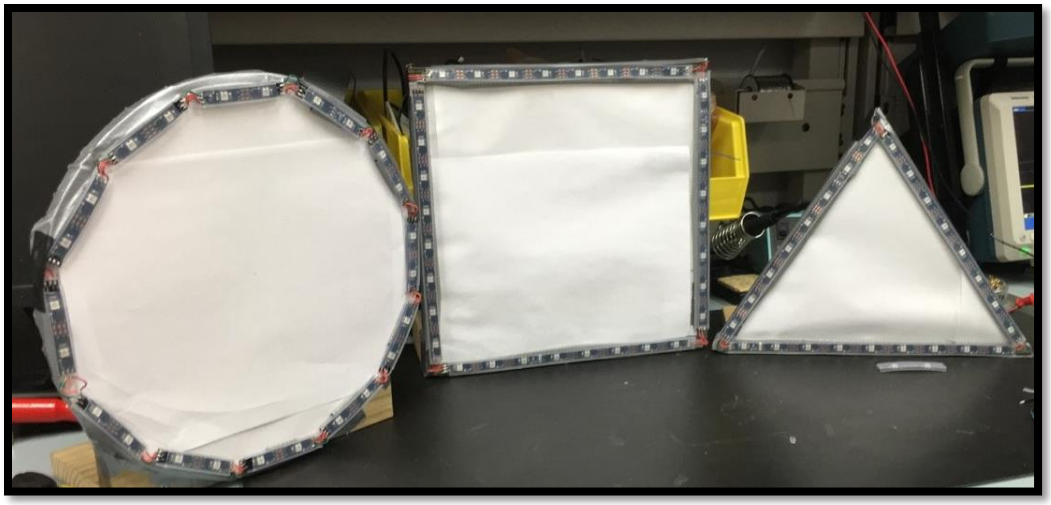


Figure 39: Beacons

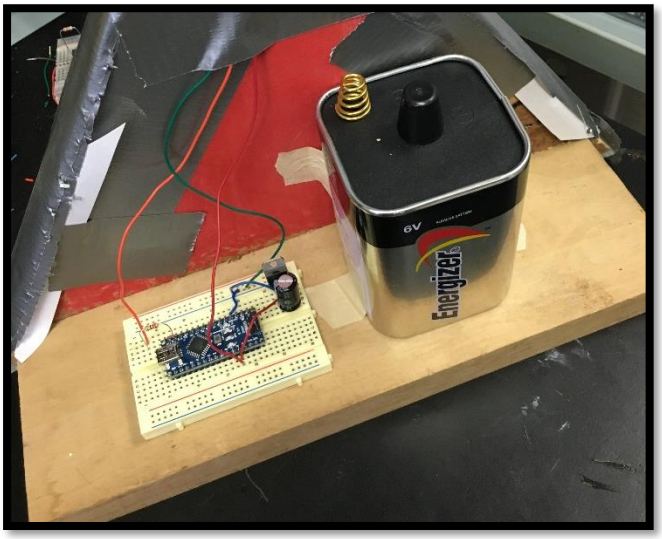


Figure 40: Beacon Circuitry

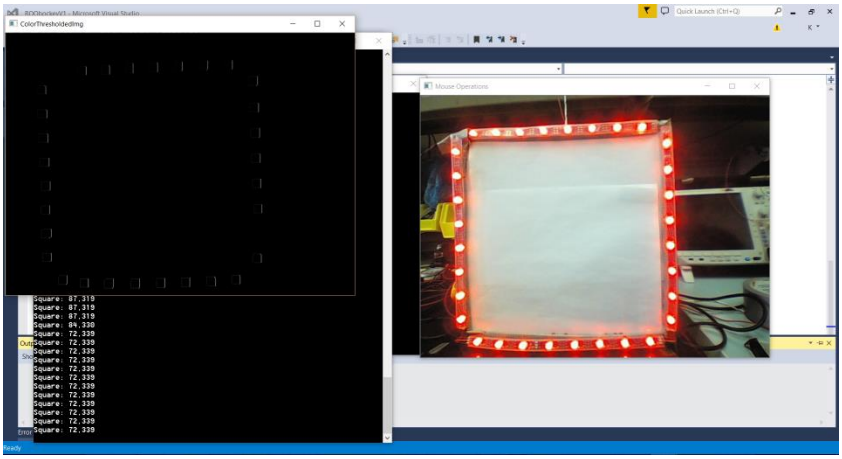


Figure 41: Beacon Operation and Tracking Using Color and Shape Recognition

4.3.10. Software Implementation [KM]

The sensor network was designed to control the motors from the camera system as an input. When OpenCV was used in parallel with the control of the motors, the robot movements are unpredictable due to a significant delay with commanding the motors. The delay was necessary in order for OpenCV to filter the input images to our needs with color and shape. The issue of having limited processor resources was definitely a large problem in getting the system to work together. Image processing is a very CPU intensive task and while the Raspberry Pi is able to operate with image processing, the control of the robot is not able to happen in the same processor thread. Due to this overcoming issue, the robot control and image processing were separated into separate processing threads. The control of the motor controller, general-purpose-input-output (GPIO) and Xbox 360 Controller were separated and in parallel with the image processing software running. After adding the two threads to the code, it is found that since the threads shared variables in the code, the code had to wait for the threads to line up with each other (also called joining threads) in order to use the current global variable values in both the two separate threads.

Figure 42 shows the software implementation of the entire system with the parallel threads operating the C++ software branches.

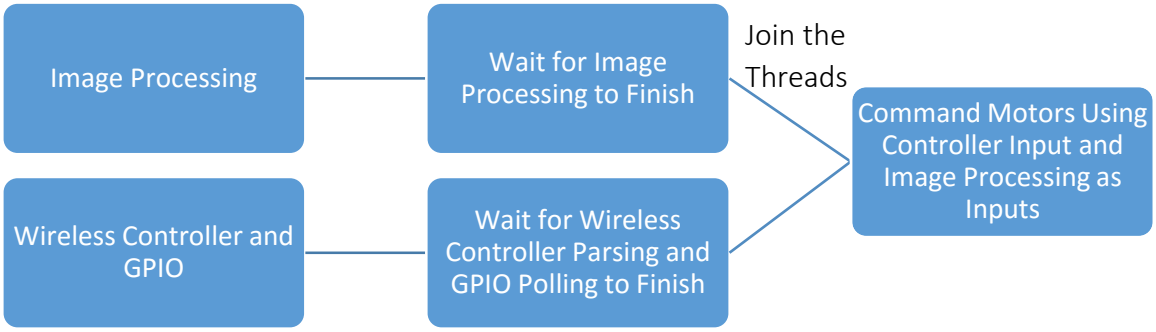


Figure 42: Software Parallel Hierarchy of Parallel Threads

The GPIO library used is WiringPi and the motors are commanded by the Sabertooth 2x25 motor controller using UART commands from the Rapsberry Pi 2 and the launching mechanism are controlled using the same library by setting the GPIO pin 18 high. The pin assignments are shown below in Table 32. The image processing library consists of using OpenCV3.0.0. The code is included in Part D of the Appendix.

The ROOBockey project has the following I/O to and from the Raspberry Pi 2.

Table 32: GPIO Pin Assignments

Inputs	
Break Beam Sensor	GPIO pin 27
Outputs	
Shoot Pin Output	GPIO pin 18
UART for Communicating with 2x25 Motor Controller	GPIO pin 14 (TXD0)

The software is implemented as described in the above level 1 and level 2 sections and is attached in section B of the Appendix. The software is able to detect colors and shapes while also being able to operate the robot in manual control. The robot is not able to align itself with the targets while operating the image processing code within the time requirement set in the project requirements. The robot operation is significantly slower that what is determined in the project requirements.

The code detects known beacons by waiting for the user to select an area of a known beacon (Green Triangle, Blue Rectangle or Red Octagon) color within the input image frames that the user wishes to track the color. The code will then automatically set the software HSV color filter for the input video and begin searching for the associated shape of the beacon of that color. The code then passes the value of the x-coordinates to the motor controller operating functions and will cause the robot to orient itself with the desired beacon.

During the project implementation, the webcam choice from the previous semester actually hindered the color recognition portion of the image processing. The camera chosen for the project (Genius F100) has a hardware exposure filter within the camera. When the image processing would see the beacon at different angles, the light from the beacon would cause the camera to saturate or “white-out” where the camera would then adjust the exposure filter to prevent that from happening. When the camera changes the hardware filter in the camera, the software filter is no longer valid and needs to be re-calibrated in order to see the chosen beacon. By eliminating the hardware filter from the chosen webcam or finding a webcam that gives the option to disable the exposure of the camera, the project would be able to function as described in the project plan.

Because of the load on the processor, the Raspberry Pi 2 is not able to successfully control the robot at the same time as running the image processing software. If given the ability to pursue this project in the future, a laptop would have been chosen as the development platform rather than the Raspberry Pi. Also the code would be written to split the load on the processor and utilizing all four processor cores. One core would run the color filter code, one core would run the shape filter and the wireless controller and GPIO would be run on the remaining processor cores.

5. Operation Instructions

5.1. Robot Startup

First, the E-Stop on the robot lid must be connected and switched off to properly ensure safety during the startup operation. The lid can then be lifted to connect the battery terminals. After this, the lid is secured with the thumb screws and the pneumatic cylinder is connected to the solenoid hookups.

Next, the Raspberry Pi must be connected to a display monitor via HDMI to access and run the driving program. The E-Stop can then be pushed to power up the robot. Once the E-Stop is pushed down, the entire robot is energized since the battery terminals are connected and all hot terminals are live. The driving and targeting program can be executed from the Raspbian terminal to start driving operation.

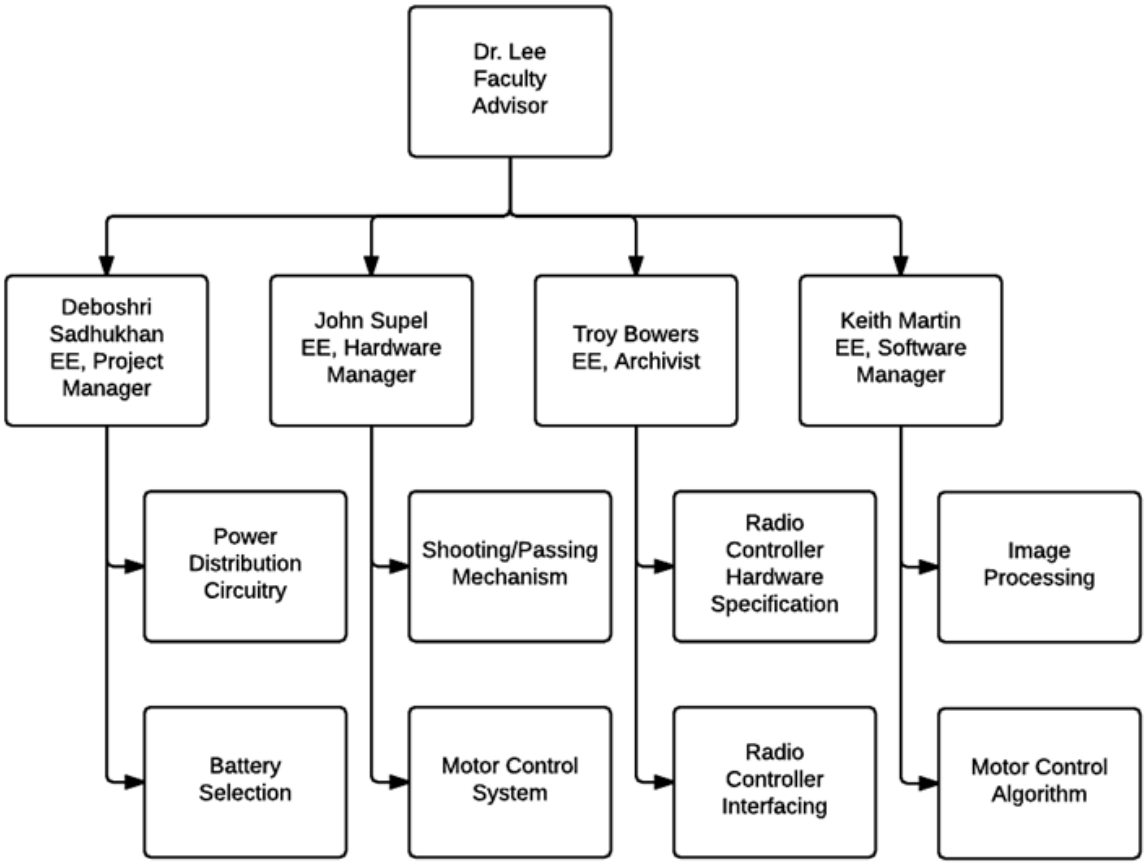
5.2. Driving

The robot is driven using the two joysticks on the Xbox controller in a tank control manner. The left joystick drives the left motor and the right joystick drives the right motor. Upward pitch on the joystick drives the motor forward and downward pitch drives it backwards. Speed can be controlled by increasing or decreasing the pitch of either joystick. The joysticks must be pushed in either forward or backwards directions, diagonal and left or right directions do not control the motors. In order to turn the robot, one joystick must be pushed further than the other, thus increasing the speed of that motor and creating a turning radius. To drive the robot straight, both joysticks must be pushed in the same direction with the same amount of pitch, thus engaging both motors at the same speed.

5.3. Shooting and Targeting

Manual shooting of the puck can be done by holding the RB button on the top right of the Xbox controller and pressing the A button once. The RB button engages manual shooting while the A button discharges the solenoid. To target a beacon, the corresponding colored button is pressed (A for green, B for red, X for blue) and then the robot will line up and shoot the puck accordingly.

6. Design Team Information



7. Conclusions and Recommendations

The ROObockey system will give the user a more enjoyable experience for playing against other hockey robots. Through the use of image processing, the robot is able to locate known beacons. The user operates the robot manually and can drive smoothly and launch the puck towards targets using simple wireless control. Allowing the user to launch the puck using the software assisted image processing algorithm resulted in a slow driving response from the robot which could be fixed using a better processor than the one on the Pi. If the Pi had a stronger processor, it would handle the image processing program through the use of parallel threading and would also be able to drive the robot. This would lighten the load on the image processing thread and allow for smoother turning and faster driving control input when implementing image processing for finding the targets. Budgetary constraints prevented the use of a more powerful main processor board that could have handled all software control without latency in robot operation. Lower cost ICs as well as standardized resistors and capacitors could be used on the power board to reduce cost. A camera without an internal hardware filter would allow the robot to find the beacons using just the software HSV filter without the hardware filter interfering.

References

[1] Google Patents, 'Mechanized robots for use in instruction, training, and practice in the sport of ice and roller hockey', 5,647,747, 1997. URL: https://www.google.com/patents/US5647747

[2] Google Patents, 'Hockey puck shooting machine', 3,822,688, 1974. URL: http://www.google.com/patents/US3822688

[3] Brauer-Burchardt, C.; Voss, K., "A new algorithm to correct fish-eye- and strong wide-angle-lens-distortion from single images," in Image Processing, 2001. Proceedings. 2001 International Conference on , vol.1, no., pp.225-228 vol.1, 2001 doi: 10.1109/ICIP.2001.958994. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=958994&isnumber=20726

[4] Docs.opencv.org, 'OpenCV: Fisheye camera model', 2015. [Online]. Available: http://docs.opencv.org/master/db/d58/group__calib3d__fisheye.html#gsc.tab=0. [Accessed: 27- Oct- 2015].

[5] H. Al-Hertani, 'Accessing The Hardware PWM Peripheral on the Raspberry Pi in C++', Hertaville, 2014. [Online]. Available: http://www.hertaville.com/rpipwm.html. [Accessed: 27- Oct- 2015].

[6] Learn.sparkfun.com, 'Bluetooth Basics - learn.sparkfun.com', 2015. [Online]. Available: https://learn.sparkfun.com/tutorials/bluetooth-basics/common-versions. [Accessed: 28- Oct- 2015].

[7] Paul, 'Wi-Fi Direct vs. Bluetooth 4.0: A Battle for Supremacy', *PCWorld*, 2015. [Online]. Available: http://www.pcworld.com/article/208778/Wi_Fi_Direct_vs_Bluetooth_4_0_A_Battle_for_Supremacy.html. [Accessed: 28- Oct- 2015].

[8] EETimes, 'Avoiding Interference in the 2.4-GHz ISM Band | EE Times', 2015. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1273359. [Accessed: 28- Oct- 2015].

[9] EETimes, 'A short history of spread spectrum | EE Times', 2015. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1279374. [Accessed: 28- Oct- 2015].

[10] J. Bunker, 'Learn the Basics of Spread Spectrum R/C | Make: DIY Projects, How-Tos, Electronics, Crafts and Ideas for Makers', *Make: DIY Projects, How-Tos, Electronics, Crafts and Ideas for Makers*, 2015. [Online]. Available: http://makezine.com/2015/06/24/skill-builder-intro-spread-spectrum-rc/. [Accessed: 28- Oct- 2015].

[11] M. Dyson, 'Using an Xbox 360 Wireless Controller with Raspberry Pi | Matt Dyson', *Mattdyson.org*, 2013. [Online]. Available: http://mattdyson.org/blog/2013/01/using-an-xbox-360-wireless-controller-with-raspberry-pi/. [Accessed: 28- Oct- 2015].

[12] Pingus.seul.org, 'Userspace Xbox/Xbox360 USB Gamepad Driver for Linux', 2015. [Online]. Available: http://pingus.seul.org/~grumbel/xboxdrv/. [Accessed: 28- Oct- 2015].

[13] T. Slominski, 'How to Set-Up & Use Your Xbox 360 Controller on Linux', *OMG! Ubuntu!*, 2013. [Online]. Available: http://www.omgubuntu.co.uk/2013/07/dealing-with-xbox-controllers-in-ubuntu. [Accessed: 28- Oct- 2015].

[14] Hackaday.io, '360 controller for RC', 2015. [Online]. Available: https://hackaday.io/project/2862-360-controller-for-rc. [Accessed: 28- Oct- 2015].

[15] Robogames.net, 'Robot Hockey Rules', 2015. [Online]. Available: http://robogames.net/rules/hockey.php. [Accessed: 28- Oct- 2015].

[16] RobotShop Blog, 'Drive Motor Sizing Tutorial - RobotShop Blog', 2012. [Online]. Available: http://www.robotshop.com/blog/en/drive-motor-sizing-tutorial-3661. [Accessed: 28- Oct- 2015].

[17] RobotShop Blog, 'How to Make a Robot - Lesson 5: Choosing a Motor Controller - RobotShop Blog', 2010. [Online]. Available: http://www.robotshop.com/blog/en/how-to-make-a-robot-lesson-5-motor-controller-3695. [Accessed: 28- Oct- 2015].

[18] Robotshop.com, '14.4V, 720RPM 300oz-in Planetary Gearmotor', 2015. [Online]. Available: http://www.robotshop.com/en/planetary-gearmotor-720rpm.html#Useful Links. [Accessed: 28- Oct- 2015].

[19] Usplastic.com, '3/8" x 12" x 12" Lexan™ Polycarbonate Sheet | U.S. Plastic Corp.', 2015. [Online]. Available: http://www.usplastic.com/catalog/item.aspx?itemid=28079&catid=704. [Accessed: 28- Oct- 2015].

[20] Analog.com, 'What Is A Voltage Regulator And How Does It Work? | Analog Devices', 2015. [Online]. Available: http://www.analog.com/en/products/landing-pages/001/fundamentals-of-voltage-regulators.html. [Accessed: 28- Oct- 2015]

[21] Analog, Embedded Processing, Semiconductor Company, Texas Instruments, 'LMZ12003 3-A Simple Switcher® Power Module with 20-V Maximum Input Voltage', 2015. [Online]. Available: http://www.ti.com/lit/ds/symlink/lmz12003.pdf. [Accessed: 07- Dec- 2015].

[22] Webench.ti.com, 'WEBENCH® Designer (Release Date: Thu Nov 19 11:37:40 2015, 1725379 bytes)', 2015. [Online]. Available:http://webench.ti.com/webench5/power/webench5.cgi?origin=ti_panel&lang_chosen=en_US&VinMin=10&VinMax=14&O1V=5&O1I=3&op_TA=30. [Accessed: 07- Dec- 2015].

[23] BatterySpace.com, 'LiFePO4 38120S (M size) Cell: 3.2V 10Ah, 100A Surge, 32Wh - UN38.3 Passed', 2015. [Online]. Available:http://www.batteryspace.com/lifepo438120pmsizecell32v10ah100asurgerate32whwith6mscrewterminal-unapproved.aspx. [Accessed: 07- Dec- 2015].

[24] FrightProps, 'Pneumatics', 2015. [Online]. Available: http://www.frightprops.com/pneumatics.html. [Accessed: 07- Dec- 2015].

[25] Pololu.com, 'Pololu - 20.4:1 Metal Gearmotor 25Dx50L mm HP 12V', 2015. [Online]. Available: https://www.pololu.com/product/3203/specs. [Accessed: 07- Dec- 2015].

[26] Dimensionengineering.com, 'Sabertooth 2X5 regenerative dual motor driver - analog, R/C, and serial motor control', 2015. [Online]. Available: https://www.dimensionengineering.com/products/sabertooth2x5. [Accessed: 07- Dec- 2015].

[27] Learn.adafruit.com, 'Overview | IR Breakbeam Sensors | Adafruit Learning System', 2015. [Online]. Available: https://learn.adafruit.com/ir-breakbeam-sensors. [Accessed: 07- Dec- 2015]

[28] B. Manufacturing Company, *Pneumatic Application & Reference Handb^{oo}k*, 1st ed. Monee: Bimba Manufacturing Company, 2011, ". 8.

[29] "Air Flow Through Orifices", *Aircompressorworks.com*, 2016. [Online]. Available: http://www.aircompressorworks.com/airflowthroughorifices.html. [Accessed: 21- Apr- 2016].

Appendix

A. 5V DC-DC Convertor Calculations [DS]

Enable Divider / R_{ENT} / R_{ENB}:

$$\frac{R_{ENT}}{R_{ENB}} = \frac{V_{IN\_UVLO}}{1.18} - 1$$

Desired V_{IN_UVLO} = 10V

$$\frac{R_{ENT}}{R_{ENB}} = \frac{10}{1.18} - 1 = 7.4746$$

R_{ENT} = 34k and R_{ENB} = 4.53k

$$V_{IN\_UVLO} = 1.18\left(\frac{34000}{4530} + 1\right) = 10.0365\ V$$

Output Voltage:

$$V_O = 0.8\left(1 + \frac{R_{FBT}}{R_{FBB}}\right)$$

Desired V_O = 5V

$$\frac{R_{FBT}}{R_{FBB}} = \frac{5}{0.8} - 1 = 5.25$$

R_{FBT} = 5.62k and R_{FBB} = 1.07k

$$V_O = 0.8\left(1 + \frac{5620}{1070}\right) = 5.00187\ V$$

Soft-Start Capacitor:

$$C_{ss} = 8 * 10^{-6}\left(\frac{t_{SS}}{0.8}\right)$$

Desired t_{ss} = 1ms

$$C_{ss} = 8 * 10^{-6}\left(\frac{10^{-3}}{0.8}\right) = 0.01\mu F$$

Output Capacitor (C_O) Selection:

$$C_O \geq I_{STEP} * V_{FB} * L * \frac{V_{IN}}{4 * V_O * (V_{IN} - V_O) * V_{OUT-TRAN}}$$

$$C_O \geq 3 * 0.8 * (6.8 * 10^{-6}) * \frac{12}{4 * 5 * (12 - 5) * (33 * 10^{-3})}$$

$$C_O \geq 42.39\mu F \ \rightarrow 100\mu F$$

On Resistor (R_{ON}) Selection:

$$R_{ON} \geq \frac{150 * 10^{-9}}{1.3 * 10^{-10}} * V_{IN\_MAX}$$

$$R_{ON} \geq \frac{150 * 10^{-9}}{1.3 * 10^{-10}} * 14$$

$$R_{ON} \geq 16.1538\ k\Omega$$

Switching Frequency:

$$f_{SW} = \frac{V_O}{(1.3 * 10^{-10}) * R_{ON}}$$

Desired f_{SW} = 420kHz

$$R_{ON} = \frac{5}{(1.3 * 10^{-10}) * (420 * 10^3)} = 91.575\ k\Omega$$

R_{ON} = 90.9 kΩ

$$f_{SW} = \frac{5}{(1.3 * 10^{-10}) * (90.9 * 10^3)} = 423.119\ kHz$$

B. 24V DC-DC Convertor Calculations [DS]

Input Voltage:

$$V_{VIN,FALLING} = 1.22\left(\frac{R_3 + R_4}{R_4}\right)$$

$$V_{VIN,FALLING} = 1.22\left(\frac{200 * 10^3 + 43.2 * 10^3}{43.2 * 10^3}\right)$$

$$V_{VIN,FALLING} = 6.868V$$

$$V_{VIN,RISING} = 2\ x\ 10^{-6} R_3 + V_{IN,FALLING}$$

$$V_{VIN,RISING} = 2\ x\ 10^{-6}(200 * 10^3) + 6.868$$

$$V_{VIN,RISING} = 7.268$$

Output Voltage:

$$V_{OUT} = 1.6\left(1 + \frac{R_2}{R_1}\right)$$

$$V_{OUT} = 1.6\left(1 + \frac{226 * 10^3}{16.2 * 10^3}\right)$$

$$V_{OUT} = 23.92V$$

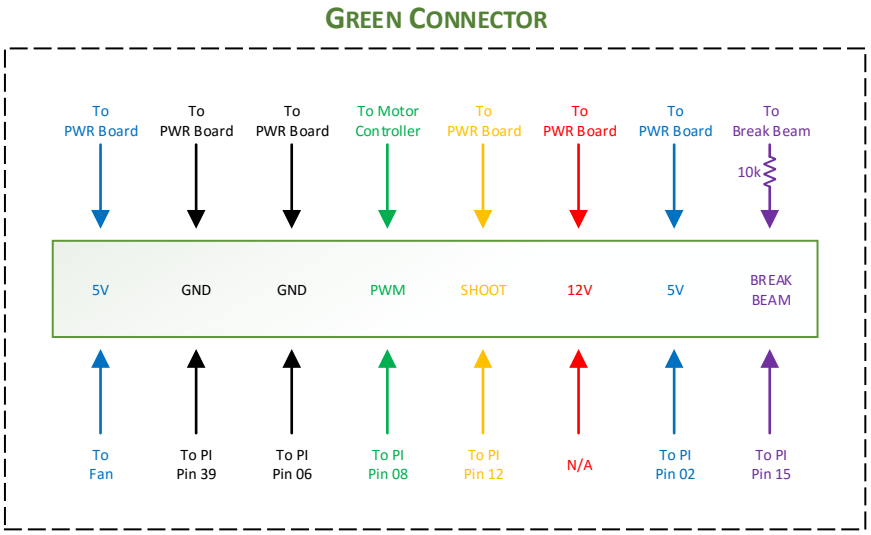
Duty Cycle Max:

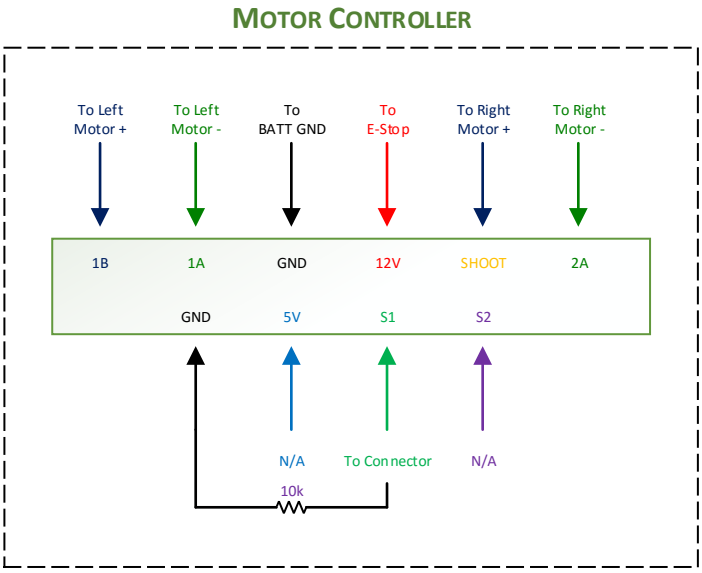
$$D_{MAX} = \frac{V_{OUT} - V_{IN(MIN)}}{V_{OUT}}$$

$$D_{MAX} = \frac{23.92 - 8}{23.92}$$

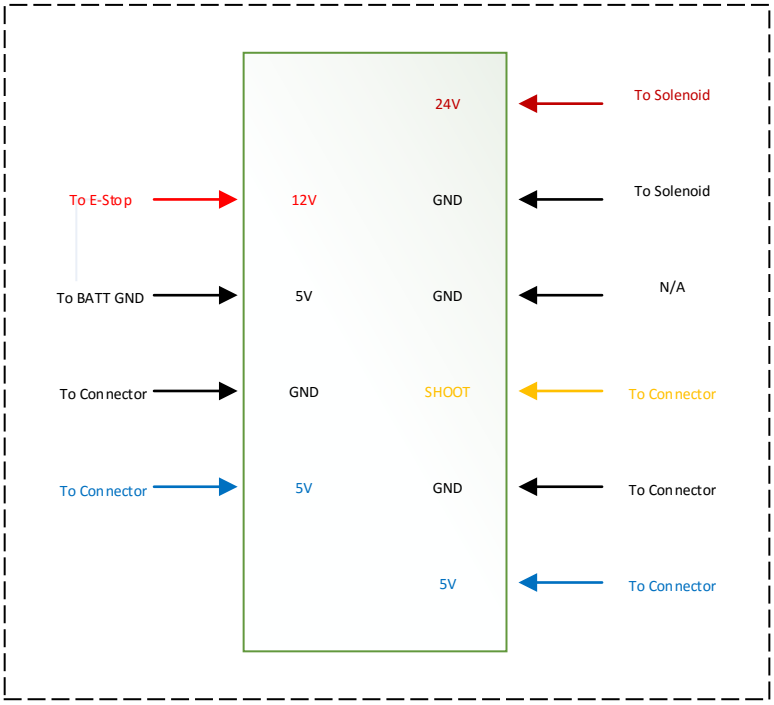
$$D_{MAX} = 0.667$$

C. Wiring Diagrams [DS]

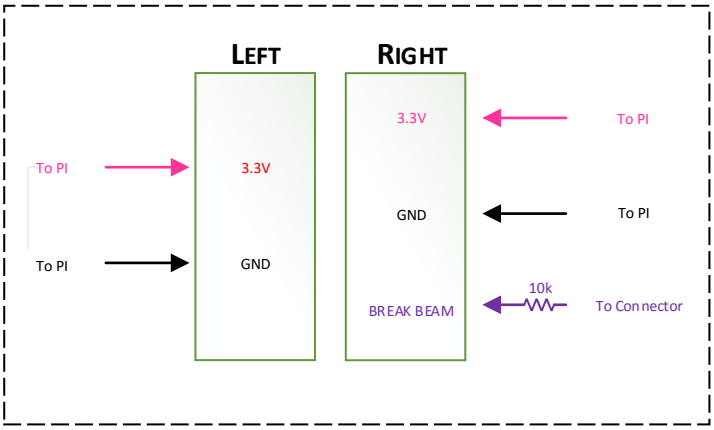




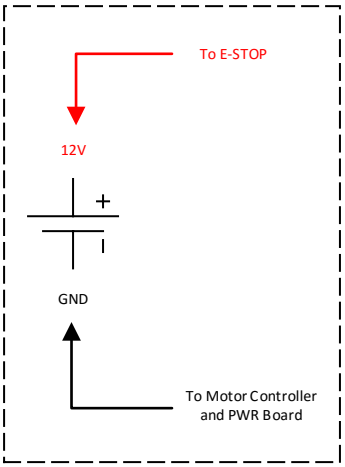
POWER BOARD



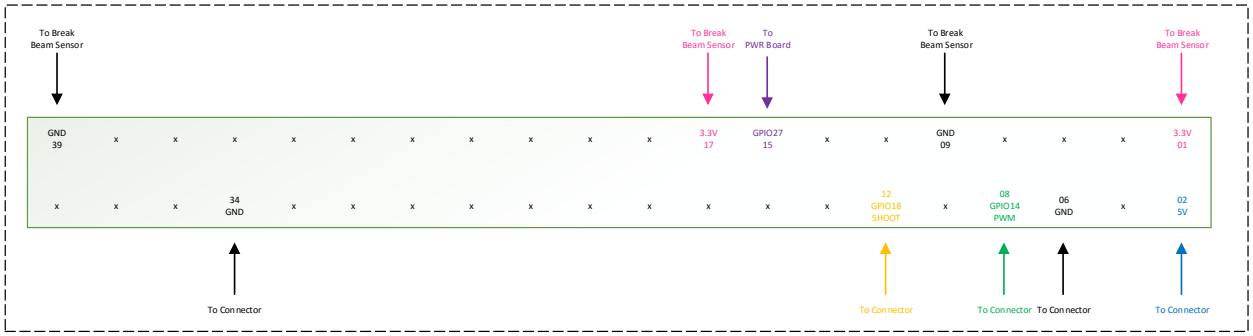
BREAK BEAM SENSORS



BATTERY



RASPBERRY PI



D. Software Code included in the project for the Raspberry Pi 2:

i. Main

<main.cpp>

```
////////////Keith Martin 2015-2016 - ROObockey Senior Design Team E - University of Akron : Design of a floor hockey puck shooting robot
////////////main.cpp - Project Used to Track Various Target Beacons of Different Shapes and Colors
//////////*NOTE THAT I AM USING OPENCV-Version3.0.0 WITH MICROSOFT VISUAL STUDIO 2013*/
////////////Installation guide - https://www.youtube.com/watch?v=et7tLwpsADw
////////////OpenCV3.0.0 Install setup is included in "OpenCV_3_Windows_10_Installation_Tutorial-master" folder within this Github post
//////////* Object detector program (uses known shapes and colors to track beacons)
/////////* It loads an image and tries to find simple shapes (rectangle, triangle, circle, etc) in it.
/////////* This program is a modified version of `squares.cpp` found in the OpenCV sample dir*/

#define USE_EXTERNS
#define MAIN_CPP
#include "defs.hpp"

#include "GPIO_UART.hpp"
#include "Xbox360Controller.hpp"
#include "main.hpp"
#include "ObjectTracking.hpp"

//Multi-Core Operation Headers
#include <thread>
#include <chrono>
#include <mutex>

std::mutex inputLock;

int main(void) {

//Initialize the Xbox360 Wireless Controller and UART Module on the Raspberry Pi 2
	initController();
	initGPIO_Uart();


//Launch the thread for the Image Processing on the Raspberry Pi 2
	std::thread imageProcessingThread([]() -> void {
		while(1) {
```

```
                auto start = std::chrono::high_resolution_clock::now();

                {
                        imageProcessingRoutine();
                }

                auto end = std::chrono::high_resolution_clock::now();
                std::chrono::duration<double, std::milli> elapsed = end - start;
                std::chrono::duration<double> second(0.3);

                if(elapsed.count() >= 0.3) {
                        std::this_thread::sleep_for(second - elapsed);
                }
        }
    });

//Launch the thread for the Xbox 360 Wireless Controller and GPIO polling
    std::thread inputOutputThread([]() -> void {
        while(1) {
                auto start = std::chrono::high_resolution_clock::now();

                {
                        std::lock_guard<std::mutex> lock(inputLock);

                        parseXbox360Controller();
                        gpioPinOperations();
                }

                auto end = std::chrono::high_resolution_clock::now();
                std::chrono::duration<double, std::milli> elapsed = end - start;
                std::chrono::duration<double> second(0.1);

                if(elapsed.count() >= 0.1) {
                        std::this_thread::sleep_for(second - elapsed);
                }
        }
    });
//the code should never get to this point because it is stuck in the above while() loops
    imageProcessingThread.join();
    inputOutputThread.join();
    return 1;
}
```

<main.hpp>

```
#ifndef MAIN_HPP
#define MAIN_HPP

extern int initController(void);
extern int parseXbox360Controller(void);
extern void imageProcessingRoutine(void);
extern int gpioPinOperations(void);
extern int initGPIO_Uart(void);

#endif /* MAIN_HPP */
```

ii. Object Tracking

<ObjectTracking.cpp>

```
#include "defs.hpp"
#include "ObjectTracking.hpp"
#include "Xbox360Controller.hpp"
#include <thread>
#include <wiringPi.h>
#include <string>

#define OBJECTTRACKING_CPP

////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////
/*Color Detection Stuff Here*/
/*Shape Detection Stuff At the Bottom*/
////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////

const string trackbarWindowName = "Trackbars";

MouseCalibrateFilter MouseInfo; //create class declaration (create the object that stores the
mouse information)
MouseCalibrateFilter *MouseHSVCalibrationPtr = &MouseInfo; //use pointer to modify the
values within the functions "clickAndDragRectangle() and mouseRecordHSV_Values()"

//initial min and max HSV filter values.
//these will be changed using trackbars
//NOTE: THESE ARE GLOBAL VARIABLES, NOT THE VARIABLES USED TO TUNE EACH INDIVIDUAL
CLASS BEACON FILTER
int H_MIN = 0;
int H_MAX = 256;
int S_MIN = 0;
int S_MAX = 256;
int V_MIN = 0;
int V_MAX = 256;


const string mouseWindowName = "Mouse Operations";

/*Define Shapes and Colors for Known Target Beacon Colors and Shapes:
* create some Beacon objects so that we can use their member functions/information
* the text "Color_shape" tells the class definition (In "Beacons.c") what shape and color category
the beacon falls in*/
```

```
Beacon RedOctagon("RedOctagon");
Beacon BlueRectangle("BlueRectangle");
Beacon GreenTriangle("GreenTriangle");

//Now define the vectors in case multiple beacons need to be tracked
vector<Beacon> RedOctagonVector;
vector<Beacon> BlueRectangleVector;
vector<Beacon> GreenTriangleVector;
int averageBeaconPosition = 0;


void imageProcessingRoutine(void){
	Mat src0;
	Mat ColorThresholded_Img0, ColorThresholded_Img, outputImg0, outputImg, src,
HSV_Input;
	vector<vector<Point> > contours;
	vector<Vec4i> hierarchy;

#ifdef USING_WEBCAM
	VideoCapture cap(CAMERA_NUMBER); //Open the Default Camera
	if (!cap.isOpened()) exit(EXIT_FAILURE); //Check if we succeeded in receiving images from
camera. If not, quit program.
	cap.set(CV_CAP_PROP_FRAME_WIDTH, FRAME_WIDTH); //Set height and width of
capture frame
	cap.set(CV_CAP_PROP_FRAME_HEIGHT, FRAME_HEIGHT);
#else
	//Testing the program using sample images copied into working project directory
	src0 = imread("images.png").clone(); //clone used to pass the Mat around in functions as
a "deep copy"
	//src0 = imread("basic-shapes-2.png").clone(); //clone used to pass the Mat around in
functions as a "deep copy"
	//src0 = imread("circlesOnWall.png").clone(); //clone used to pass the Mat around in
functions as a "deep copy"
	//src0 = imread("pic3.png").clone(); //clone used to pass the Mat around in functions as a
"deep copy"
	//src0 = imread("pic5.png").clone(); //clone used to pass the Mat around in functions as a
"deep copy"
#endif


#ifdef CALIBRATION_MODE
	//Create trackbars that you can manually change in order to alter the HSV filter minimum
& maximum values
	//I commented this out beacuse it will not run in Linux (Raspberry Pi2)
```

```
	//The HSV filter is calibrated using the mouse in CALIBRATION_MODE or hardcoded to the class HSV values in Beacons.cpp when each beacon object is created
	//createObjectTrackingParameterTrackbars();

	//create a window before setting mouse callback
	namedWindow(mouseWindowName);
	//set mouse callback function to be active on "Webcam Feed" window
	//we pass the handle to our "frame" matrix so that we can draw a rectangle to it as the user clicks and drags the mouse
	//NOTE: THE "OnMouse" function parameter for "setMouseCallback()" for setMouseCallback is a function with parameters: (int, int, int, void*);
	setMouseCallback(mouseWindowName, clickAndDragRectangle, &src);
#endif


#ifdef USING_WEBCAM
	cap >> src0; //get a new frame from camera
#endif

	src = src0.clone(); //get a "deep copy" (physical, not pointer) copy the input video frame
	cvtColor(src, HSV_Input, COLOR_BGR2HSV); //convert the input BGR color image to a HSV image


#ifdef CALIBRATION_MODE
	//set HSV values from user selected region
	mouseRecordHSV_Values(src, HSV_Input);

	if ((calibratingTrackColorFilteredObjects(src, HSV_Input, contours, hierarchy, ColorThresholded_Img0)) > 0) { //number of objects detected > 0 and < "MAX_NUM_OBJECTS"
		ColorThresholded_Img = ColorThresholded_Img0.clone(); //had to clone the image to pass a "deep copy" to the shape detection function
		shapeDetection(ColorThresholded_Img, contours, hierarchy, outputImg0); //search for shapes in the color filtered thresholded image
		outputImg = outputImg0.clone(); //had to clone the image to pass a "deep copy" to the output "imshow"
	}
#else
	if (Ba == 1) { //check if button on controller for beacon color was pressed
		if ( (trackColorFilteredObjects(src, HSV_Input, GreenTriangleVector, contours, hierarchy, ColorThresholded_Img0) ) > 0) { //number of objects detected > 0 and < MAX_NUM_OBJECTS
			ColorThresholded_Img = ColorThresholded_Img0.clone(); //had to clone the image to pass a "deep copy" to the shape detection function
```

```
                        shapeDetection(ColorThresholded_Img, contours, hierarchy, outputImg0);
//search for shapes in the color filtered thresholded image
                        outputImg = outputImg0.clone(); //had to clone the image to pass a "deep
copy" to the output "imshow"
                        averageBeaconPosition = chooseBeaconToShootAt();
                        alignWithBeacon(averageBeaconPosition);
                }
        }
        if (Bx == 1) { //check if button on controller for beacon color was pressed
                if ((trackColorFilteredObjects(src, HSV_Input, BlueRectangleVector, contours,
hierarchy, ColorThresholded_Img0)) > 0) { //number of objects detected > 0 and <
MAX_NUM_OBJECTS
                        ColorThresholded_Img = ColorThresholded_Img0.clone(); //had to clone
the image to pass a "deep copy" to the shape detection function
                        shapeDetection(ColorThresholded_Img, contours, hierarchy, outputImg0);
//search for shapes in the color filtered thresholded image
                        outputImg = outputImg0.clone(); //had to clone the image to pass a "deep
copy" to the output "imshow"
                        averageBeaconPosition = chooseBeaconToShootAt();
                        alignWithBeacon(averageBeaconPosition);
                }
        }
        if (Bb == 1) { //check if button on controller for beacon color was pressed
                if ((trackColorFilteredObjects(src, HSV_Input, RedOctagonVector, contours,
hierarchy, ColorThresholded_Img0)) > 0) { //number of objects detected > 0 and <
MAX_NUM_OBJECTS
                        ColorThresholded_Img = ColorThresholded_Img0.clone(); //had to clone
the image to pass a "deep copy" to the shape detection function
                        shapeDetection(ColorThresholded_Img, contours, hierarchy, outputImg0);
//search for shapes in the color filtered thresholded image
                        outputImg = outputImg0.clone(); //had to clone the image to pass a "deep
copy" to the output "imshow"
                        averageBeaconPosition = chooseBeaconToShootAt();
                        alignWithBeacon(averageBeaconPosition);
                }
        }


#endif //CALIBRATION_MODE



#ifdef SHOW_OPENCV_IMAGES
```

```
	if(src.data)
	{
		imshow(mouseWindowName, src); //show Input BGR Mat video frame in new
window
	}

	if(ColorThresholded_Img0.data)
	{
		imshow("ColorThresholdedImg", ColorThresholded_Img0);
	}

	if(ColorThresholded_Img.data)
	{
		imshow("OutputColor&ShapeDetectedImg", ColorThresholded_Img);
	}

	if(outputImg.data)
	{
		imshow("OutputImg", outputImg);
	}

	waitKey(5); //delay in milliseconds so OpenCV does not consume all processor time.
"imshow" will not appear without this waitKey() command

#endif //SHOW_OPENCV_IMAGES

}



void on_trackbar(int, void*) {
	// This function gets called whenever a trackbar position is changed
}



//create trackbars and insert them into their own window as sliders to control tracking
parameters
//these sliders allow the user to tune the HSV to show the intended object color
void createObjectTrackingParameterTrackbars(void) {
	namedWindow("Trackbars", 0); // create window for trackbars
	char TrackbarName[50]; // create memory to store trackbar name on window
	sprintf(TrackbarName, "H_MIN", H_MIN);
```

```
	sprintf(TrackbarName, "H_MAX", H_MAX);
	sprintf(TrackbarName, "S_MIN", S_MIN);
	sprintf(TrackbarName, "S_MAX", S_MAX);
	sprintf(TrackbarName, "V_MIN", V_MIN);
	sprintf(TrackbarName, "V_MAX", V_MAX);

	//3 parameters are: the address of the variable that is changing when the trackbar is moved(eg.H_LOW),
	//the max value the trackbar can move (eg. H_HIGH),
	//and the function that is called whenever the trackbar is moved(eg. on_trackbar)
	createTrackbar("H_MIN", trackbarWindowName, &H_MIN, H_MAX, on_trackbar);
	createTrackbar("H_MAX", trackbarWindowName, &H_MAX, H_MAX, on_trackbar);
	createTrackbar("S_MIN", trackbarWindowName, &S_MIN, S_MAX, on_trackbar);
	createTrackbar("S_MAX", trackbarWindowName, &S_MAX, S_MAX, on_trackbar);
	createTrackbar("V_MIN", trackbarWindowName, &V_MIN, V_MAX, on_trackbar);
	createTrackbar("V_MAX", trackbarWindowName, &V_MAX, V_MAX, on_trackbar);
}


//filter noise in the HSV image by eroding and dilating the image. this will prevent false color detection in the image
void morphOps(Mat &thresh) {
	//create structuring element that will be used to filter image using "dilate" and "erode" on the image.
	//the element chosen here is a 3px by 3px rectangle
	Mat erodeElement = getStructuringElement(MORPH_RECT, Size(3, 3));
	//dilate with larger element so make sure object is nicely visible
	Mat dilateElement = getStructuringElement(MORPH_RECT, Size(8, 8));
	erode(thresh, thresh, erodeElement); //eliminate noise
	//erode(thresh, thresh, erodeElement);
	dilate(thresh, thresh, dilateElement); //enhance groups of pixels in thresholded image
	//dilate(thresh, thresh, dilateElement);
}


//CALIBRATION TEST FUNCTION is used to calibrate HSV filter after input BGR image and output contours detected if there are not too many
size_t calibratingTrackColorFilteredObjects(Mat &InputMat, Mat &HSV, vector<vector<Point> > &contours, vector<Vec4i> &hierarchy, Mat &threshold) {

	//Generate a binary image from the HSV input image
	inRange(HSV, Scalar(H_MIN, S_MIN, V_MIN), Scalar(H_MAX, S_MAX, V_MAX), threshold);
```

```
	//Dilate and Erode the image frame in order to filter out noise and enhance the desired color
	morphOps(threshold);

	//find contours in filtered image
	findContours(threshold, contours, hierarchy, CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE);
	//use moments method to find our filtered object
	double refArea = 0;
	//////////////size_t numObjects = hierarchy.size(); //counts the objects seen after applied threshold
	size_t numObjects = contours.size(); //counts the objects seen after applied threshold

	if ((numObjects > 0) && (numObjects<MAX_NUM_OBJECTS)) { //if number of objects > MAX_NUM_OBJECTS we have a noisy filter
		return numObjects; //function passes if objects are detected, but there are not too many detected objects (from bad filter)
	}else {
		putText(threshold, "TOO MUCH NOISE! ADJUST FILTER", Point(0, 50), 1, 2, Scalar(0, 0, 255), 2); //too many objects after filter
		//putText(InputMat, "TOO MUCH NOISE! ADJUST FILTER", Point(0, 50), 1, 2, Scalar(0, 0, 255), 2); //too many objects after filter
	}
	return 0;
}


//Function is used for each Beacon to filter input BGR image and output contours detected if there are not too many
size_t trackColorFilteredObjects(Mat &InputMat, Mat &HSV, vector<Beacon> &theBeaconsVector, vector<vector<Point> > &contours, vector<Vec4i> hierarchy, Mat &threshold) {
	inRange(HSV, theBeaconsVector[0].getHSVmin(), theBeaconsVector[0].getHSVmax(), threshold); //HSV input image and output a thresholded binary (black and white) image
	morphOps(threshold); //filter the thresholded binary image

	//find contours in filtered image
	findContours(threshold, contours, hierarchy, CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE);

	//use moments method to find our filtered object
	double refArea = 0;
	//////////////size_t numObjects = hierarchy.size(); //counts the objects seen after applied threshold
```

```
	size_t numObjects = contours.size(); //counts the objects seen after applied threshold

	if ((numObjects > 0) && (numObjects<MAX_NUM_OBJECTS)) { //if number of objects > MAX_NUM_OBJECTS we have a noisy filter
		return numObjects; //function passes if objects are detected, but there are not too many detected objects (from bad filter)
	}

	else putText(threshold, "TOO MUCH NOISE! ADJUST FILTER", Point(0, 50), 1, 2, Scalar(0, 0, 255), 2); //too many objects after filter
	return 0;
}




//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Mouse Calibration of HSV color filter
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//This function is used to calibrate the HSV values for the color filter
//The function works when the user clicks the left mouse button and highlights the input image color area to get the minimum and maximum HSV values
void clickAndDragRectangle(int event, int x, int y, int flags, void* param) {
		//only if calibration mode is on will we use the mouse to change HSV values
#ifdef CALIBRATION_MODE

		//get handle to video feed passed in as "param" and cast as Mat pointer
		Mat* videoFeed = (Mat*)param;

		if (event == CV_EVENT_LBUTTONDOWN && MouseHSVCalibrationPtr->mouseIsDragging == false) {
			MouseHSVCalibrationPtr->initialClickPoint = Point(x, y); //keep track of initial point clicked
			MouseHSVCalibrationPtr->mouseIsDragging = true; //user has begun dragging the mouse
		}
		//user is dragging the mouse
		if (event == CV_EVENT_MOUSEMOVE && MouseHSVCalibrationPtr->mouseIsDragging == true) {
			MouseHSVCalibrationPtr->currentMousePoint = Point(x, y); //keep track of current mouse point
```

```
                        MouseHSVCalibrationPtr->mouseMove = true; //user has moved the
mouse while clicking and dragging
                }
                //user has released left button
                if (event == CV_EVENT_LBUTTONUP && MouseHSVCalibrationPtr-
>mouseIsDragging == true) {
                        MouseHSVCalibrationPtr->rectangleROI = Rect(MouseHSVCalibrationPtr-
>initialClickPoint, MouseHSVCalibrationPtr->currentMousePoint); //set rectangle ROI to the
rectangle that the user has selected
                        //reset boolean variables
                        MouseHSVCalibrationPtr->mouseIsDragging = false;
                        MouseHSVCalibrationPtr->mouseMove = false;
                        MouseHSVCalibrationPtr->rectangleSelected = true;
                }
                if (event == CV_EVENT_RBUTTONDOWN) {
                        //user has clicked right mouse button, so Reset HSV Values
                        H_MIN = 0;
                        S_MIN = 0;
                        V_MIN = 0;
                        H_MAX = 255;
                        S_MAX = 255;
                        V_MAX = 255;
                }
                if (event == CV_EVENT_MBUTTONDOWN) {
                        //user has clicked middle mouse button
                        //enter code here if needed.
                }
#endif
}



//This function is used to record the HSV values in the main while loop to constantly check if the
mouse was clicked and the callback
//altered the HSV values beacuse a region was selected in the input image
void mouseRecordHSV_Values(Mat frame, Mat hsv_frame) {
        //save HSV values for RegionOfInterest (R.O.I.) that user selected to a vector
        if ((MouseHSVCalibrationPtr->mouseMove == false) && (MouseHSVCalibrationPtr-
>rectangleSelected == true)) {

                //clear previous vector values
                if (MouseHSVCalibrationPtr->H_ROI.size()>0) MouseHSVCalibrationPtr-
>H_ROI.clear();
```

```
		if (MouseHSVCalibrationPtr->S_ROI.size()>0) MouseHSVCalibrationPtr->S_ROI.clear();
		if (MouseHSVCalibrationPtr->V_ROI.size()>0) MouseHSVCalibrationPtr->V_ROI.clear();

		//if the rectangle has no width or height (user has only dragged a line) then we don't try to iterate over the width or height
		if (MouseHSVCalibrationPtr->rectangleROI.width < 1 || MouseHSVCalibrationPtr->rectangleROI.height < 1) {
			cout << "Please drag a rectangle, not a line" << endl;
		}
		else {
			for (int i = MouseHSVCalibrationPtr->rectangleROI.x; i<MouseHSVCalibrationPtr->rectangleROI.x + MouseHSVCalibrationPtr->rectangleROI.width; i++) {
				//iterate through both x and y direction and save HSV values at each and every point
				for (int j = MouseHSVCalibrationPtr->rectangleROI.y; j<MouseHSVCalibrationPtr->rectangleROI.y + MouseHSVCalibrationPtr->rectangleROI.height; j++) {
					//save HSV value at this point
					MouseHSVCalibrationPtr->H_ROI.push_back((int)hsv_frame.at<cv::Vec3b>(j, i)[0]);
					MouseHSVCalibrationPtr->S_ROI.push_back((int)hsv_frame.at<cv::Vec3b>(j, i)[1]);
					MouseHSVCalibrationPtr->V_ROI.push_back((int)hsv_frame.at<cv::Vec3b>(j, i)[2]);
				}
			}
		}

		MouseHSVCalibrationPtr->rectangleSelected = false; //reset rectangleSelected so user can select another region if necessary

		//set min and max HSV values from min and max elements of each array
		if (MouseHSVCalibrationPtr->H_ROI.size()>0) {
			//NOTE: min_element and max_element return iterators so we must dereference them with "*"
			H_MIN = *std::min_element(MouseHSVCalibrationPtr->H_ROI.begin(), MouseHSVCalibrationPtr->H_ROI.end());
			H_MAX = *std::max_element(MouseHSVCalibrationPtr->H_ROI.begin(), MouseHSVCalibrationPtr->H_ROI.end());
			cout << "MIN 'H' VALUE: " << H_MIN << endl;
			cout << "MAX 'H' VALUE: " << H_MAX << endl;
```

```
			}
			if (MouseHSVCalibrationPtr->S_ROI.size()>0) {
				S_MIN = *std::min_element(MouseHSVCalibrationPtr->S_ROI.begin(),
MouseHSVCalibrationPtr->S_ROI.end());
				S_MAX = *std::max_element(MouseHSVCalibrationPtr->S_ROI.begin(),
MouseHSVCalibrationPtr->S_ROI.end());
				cout << "MIN 'S' VALUE: " << S_MIN << endl;
				cout << "MAX 'S' VALUE: " << S_MAX << endl;
			}
			if (MouseHSVCalibrationPtr->V_ROI.size()>0) {
				V_MIN = *std::min_element(MouseHSVCalibrationPtr->V_ROI.begin(),
MouseHSVCalibrationPtr->V_ROI.end());
				V_MAX = *std::max_element(MouseHSVCalibrationPtr->V_ROI.begin(),
MouseHSVCalibrationPtr->V_ROI.end());
				cout << "MIN 'V' VALUE: " << V_MIN << endl;
				cout << "MAX 'V' VALUE: " << V_MAX << endl;
			}
		}
		if (MouseHSVCalibrationPtr->mouseMove == true) {
			//if the mouse is held down, we will draw the click and dragged rectangle to the
screen
			rectangle(frame, MouseHSVCalibrationPtr->initialClickPoint,
Point(MouseHSVCalibrationPtr->currentMousePoint.x, MouseHSVCalibrationPtr-
>currentMousePoint.y), GREEN, 1, 8, 0);
		}
}


////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////
/*Shape Detection Below*/
////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////


//Helper function to display text in the center of a contour
void setLabel(Mat& im, const string label, vector<Point> &contour) {
		int fontface = cv::FONT_HERSHEY_SIMPLEX;
		double scale = 0.4;
		int thickness = 1;
		int baseline = 0;
		Size text = cv::getTextSize(label, fontface, scale, thickness, &baseline);
		Rect r = cv::boundingRect(contour);
		Point pt(r.x + ((r.width - text.width) / 2), r.y + ((r.height + text.height) / 2));
```

```
	rectangle(im, pt + cv::Point(0, baseline), pt + cv::Point(text.width, -text.height), WHITE,
CV_FILLED);
	putText(im, label, pt, fontface, scale, BLACK, thickness, 8);
}


// the function draws all the contours in the image in a new window (colors in BGR, not RGB)
void drawContours(Mat& image, const vector<vector<Point> > &contours, string title) {
	size_t i = 0;
	string shape;
	for (i = 0; i < contours.size(); i++) {
		const Point* p = &contours[i][0];
		int n = (int)contours[i].size();
		//if (title == "DetectingGreenTriangles") {
		if (contours[i].size() == 3) { //check if each contour is a triangle
			polylines(image, &p, &n, 1, true, GREEN, 3, LINE_AA); //yellow for
triangles
		}
		//if (title == "DetectingRedRectangles") {
		if (contours[i].size() == 4) { //check if each contour is a square
			polylines(image, &p, &n, 1, true, BLUE, 3, LINE_8); //blue for rectangles
			//drawContours(image, contours, i, BLUE, FILLED, 8, hierarchy);
			//drawContours(test1, contours, i, BLUE, LINE_8, 8, hierarchy);
			//drawContours(test2, contours, i, BLUE, LINE_4, 8, hierarchy);
			//drawContours(test3, contours, i, BLUE, LINE_AA, 8, hierarchy);
		}
		//if (title == "DetectingPurplePentagons") {
			if (contours[i].size() == 5) { //check if each contour is a pentagon
				polylines(image, &p, &n, 1, true, PURPLE, 3, LINE_AA); //purple for
pentagons
			}
		//if (title == "DetectingRedOctagons") {
			if (contours[i].size() == 6) { //check if each contour is a hexagon
				polylines(image, &p, &n, 1, true, RED, 3, LINE_AA); //red for
octagons
			}
		//if (title == "DetectingYellowCircles") {
			if (contours[i].size() > 6) { //check if each contour is a circle
				polylines(image, &p, &n, 1, true, YELLOW, 3, LINE_AA); //yellow for
circles
			}
	}
}
```

```
// Records the (X,Y) position of each Beacon
void RecordBeaconPosition(Beacon &theBeacon, vector<vector<Point> > &contours,
vector<Beacon> &theBeaconsVector) {
	//for (int index = 0; index >= 0; index = hierarchy[index][0]) {
	size_t numObjects = contours.size(); //counts the number of shapes detected
	for (int index = 0; index < numObjects; index++) {
		Moments moment = moments((Mat)contours[index]);
		double area = moment.m00;
		Point center((int)(moment.m10 / area), (int)(moment.m01 / area)); //determine
the center of the detectd object using moments

		//if the area is less than 20 px by 20px then it is probably just noise
		//if the area is the same as the 3/2 of the image size, probably just a bad filter
		//we only want the object with the largest area so we save a reference area each
		//iteration and compare it to the area in the next iteration.
		if (area > MIN_OBJECT_AREA) {
			theBeacon.setXPos(center.x);
			theBeacon.setYPos(center.y);
			theBeaconsVector.push_back(theBeacon); //add additional element to the
end of the BeaconsVector Vector

#ifdef ShowDetectedObjects
			cout<<theBeacon.getShape()<<": "<<
theBeacon.getXPos()<<","<<theBeacon.getYPos()<<endl;
#endif //ShowDetectedObjects

		}
	}
}


//Helper function to find a cosine of angle between vectors from pt0->pt1 and pt0->pt2
static double angle(Point pt1, Point pt2, Point pt0) {
	double dx1 = pt1.x - pt0.x;
	double dy1 = pt1.y - pt0.y;
	double dx2 = pt2.x - pt0.x;
	double dy2 = pt2.y - pt0.y;
	return (dx1*dx2 + dy1*dy2) / sqrt((dx1*dx1 + dy1*dy1)*(dx2*dx2 + dy2*dy2) + 1e-10);
}


//Function to locate all contours (shapes) within a given image
```

```
void shapeDetection(Mat& inputImage, vector<vector<Point> > contours, vector<Vec4i>
hierarchy, Mat& outputImage) {

	Beacon theBeacon("unknown_shape");
	vector<Point> approx; //each discovered contour (shape) found from approxPolyDP()
function
	outputImage = inputImage.clone(); //copy the input Mat image to get the size of the
image
	outputImage = Scalar(0, 0, 0); //black background for output image (overwrites the input
image cloned values)

	for (int i = 0; i < contours.size(); i++) {
		// Approximate contour with accuracy proportional to the contour perimeter
		approxPolyDP(cv::Mat(contours[i]), approx, cv::arcLength(cv::Mat(contours[i]),
true)*0.02, true);
		size_t vertices = approx.size(); // Number of vertices of polygonal curve
		// Skip small or non-convex objects
		if ((fabs(contourArea(contours[i])) < 100) || (!isContourConvex(approx)))
			continue;

		// Detect and label triangles (vertices == 3)
		if (vertices == 3) {
			setLabel(outputImage, "TRI", contours[i]); //label triangles on output
image
			drawContours(outputImage, contours, "DetectingTriangles");
			RecordBeaconPosition(GreenTriangle, contours, GreenTriangleVector);
//show YellowTriangle x,y pixel coordinates to terminal window
		}

		//if ((vertices >= 4) && (vertices <= 6)) {
		if ((vertices > 3) && (vertices < 7)) {
			// Get the cosines of all corners
			vector<double> cos;
			for (int j = 2; j < vertices + 1; j++)
				cos.push_back(angle(approx[j%vertices], approx[j - 2], approx[j -
1]));
			// Sort ascending the cosine values
			sort(cos.begin(), cos.end());
			// Get the lowest and the highest cosine
			double mincos = cos.front();
			double maxcos = cos.back();

			// Detect and label squares (vertices == 4)
```

```
                        // Use the degrees obtained above and the number of vertices to
determine the shape of the contour
                        //if (vtc == 4 && mincos >= -0.1 && maxcos <= 0.3) {
                        if ((vertices == 4) && (mincos >= -0.25) && (maxcos <= 0.3125)) { //angles
between 72 and 105 are acceptable (90 is ideal)
                                setLabel(outputImage, "RECT", contours[i]); //label rectangles on
output image
                                drawContours(outputImage, contours, "DetectingRectangles");
                                RecordBeaconPosition(RedOctagon, contours,
BlueRectangleVector); //show RedRectangle x,y pixel coordinates to terminal window
                        }

                        // Detect and label octagons (vertices == 8)
                        else if ((vertices == 8) && (mincos >= -0.875) && (maxcos <= -0.625)) {
//angle b/t 128.68 and 151.045 degrees (ideally 135 degrees for octagon)
                                setLabel(outputImage, "OCT", contours[i]); //label hexagons on
output image
                                drawContours(outputImage, contours, "DetectingOctagons");
                                RecordBeaconPosition(RedOctagon, contours, RedOctagonVector);
//show BlueHexagon x,y pixel coordinates to terminal window
                        }

                        //Now detect circles using better method that "Houghcircles"
/*              }else {
                        // Detect and label circles (vertices > 6)
                        double area = contourArea(contours[i]);
                        Rect r = boundingRect(contours[i]);
                        int radius = r.width / 2;

                        if ((abs(1 - ((double)r.width / r.height)) <= 0.2) &&
                                (abs(1 - (area / (CV_PI * pow(radius, 2)))) <= 0.2)) {
                                setLabel(outputImage, "CIR", contours[i]);
                                drawContours(outputImage, contours, "DetectingCircles");
                                RecordBeaconPosition(GreenCircle, contours, GreenCircleVector);
                        }
*/
                }
        }
}


string intToString(int number) {
        stringstream ss;
        ss << number;
```

```
	return ss.str();
}


void DrawTarget(int x, int y, Mat &frame) {
	//use some of the openCV drawing functions to draw crosshairs on your tracked image!

	//'if' and 'else' statements to prevent memory errors from writing off the screen (ie. (-25,-25) is not within the window)
	circle(frame, Point(x, y), 20, GREEN, 2);
	if (y - 25>0)
		line(frame, Point(x, y), Point(x, y - 25), GREEN, 2);
	else line(frame, Point(x, y), Point(x, 0), GREEN, 2);
	if (y + 25<FRAME_HEIGHT)
		line(frame, Point(x, y), Point(x, y + 25), GREEN, 2);
	else line(frame, Point(x, y), Point(x, FRAME_HEIGHT), GREEN, 2);
	if (x - 25>0)
		line(frame, Point(x, y), Point(x - 25, y), GREEN, 2);
	else line(frame, Point(x, y), Point(0, y), GREEN, 2);
	if (x + 25<FRAME_WIDTH)
		line(frame, Point(x, y), Point(x + 25, y), GREEN, 2);
	else line(frame, Point(x, y), Point(FRAME_WIDTH, y), GREEN, 2);

	putText(frame, intToString(x) + "," + intToString(y), Point(x, y + 30), 1, 1, GREEN, 2);

}


int chooseBeaconToShootAt(void) {
	int i = 0;
	int avgXValue = 0;
	int MillisWaitTime = 100; //milliseconds until the code is run again
	int alignment = 0;

	if ( (Ba && Bx) || (Bb && Bx) || (Ba && Bb) ) { //check if multiple buttons wer pushed. If so, leave function
		return 0;
	}

	if (Ba == 1) { //green beacon was chosen by the user pressing the green button on the wireless controller
		int numGreenTriangleBeacons = GreenTriangleVector.size();
		if (numGreenTriangleBeacons == 0) { //see if beacon is on image frame (if vector is empty, rotate the robot clockwise)
```

```
                        /*if (millis() > MillisWaitTime) {
                                sendMotorControllerSpeedBytes(UART_ID, 80, 176); //rotate the
robot clockwise in order to find the beacon
                                MillisWaitTime += MillisWaitTime;
                                return 0;
                        }*/
                }else { //some beacons were found
                        for (i = 0; i < numGreenTriangleBeacons; i++) {
                                avgXValue += GreenTriangleVector[i].getXPos(); //average the x
coordinates of the detected Green Triangles
                        }
                        avgXValue = avgXValue / numGreenTriangleBeacons; //average the x
coordinates of the detected Green Triangles
                        alignment = (FRAME_WIDTH / 2) - avgXValue;
                        return alignment;
                }
        }

        if (Bx == 1) { //blue beacon was chosen
                int numBlueRectangleBeacons = BlueRectangleVector.size();
                if (numBlueRectangleBeacons == 0) { //see if beacon is on image frame (if vector
is empty, rotate the robot clockwise)
                        /*if (millis() > MillisWaitTime) {
                                sendMotorControllerSpeedBytes(UART_ID, 80, 176); //rotate the
robot clockwise in order to find the beacon
                                MillisWaitTime += MillisWaitTime;
                                return 0;
                        }*/
                }else { //some beacons were found
                        for (i = 0; i < numBlueRectangleBeacons; i++) {
                                avgXValue += BlueRectangleVector[i].getXPos(); //average the x
coordinates of the detected Green Triangles
                        }
                        avgXValue = avgXValue / numBlueRectangleBeacons; //average the x
coordinates of the detected Green Triangles
                        alignment = (FRAME_WIDTH / 2) - avgXValue;
                        return alignment;
                }
        }


        if (Bb == 1) { //red beacon was chosen
                int numRedOctagonBeacons = RedOctagonVector.size();
```

```
		if (numRedOctagonBeacons == 0) { //see if beacon is on image frame (if vector is
empty, rotate the robot clockwise)
			/*if (millis() >= MillisWaitTime) {
				sendMotorControllerSpeedBytes(UART_ID, 80, 176); //rotate the
robot clockwise in order to find the beacon
				MillisWaitTime += MillisWaitTime;
				return 0;
			}*/
		}else { //some beacons were found
			for (i = 0; i < numRedOctagonBeacons; i++) {
				avgXValue += RedOctagonVector[i].getXPos(); //average the x
coordinates of the detected Green Triangles
			}
			avgXValue = avgXValue / numRedOctagonBeacons; //average the x
coordinates of the detected Green Triangles
			alignment = (FRAME_WIDTH / 2) - avgXValue;
	return alignment;
		}
	 }

	 return 0;
}

void sendMotorCommand(int right, int left) {
   sendMotorControllerSpeedBytes(UART_ID, right, left + 128);
}

//Resulution for output images are:
//const int FRAME_WIDTH x FRAME_HEIGHT == (640x480 window)
int alignWithBeacon(int pixelsFromCenter) {
	std::lock_guard lock(inputLock);

	if (pixelsFromCenter > 320) { //check to see if the center of the frame is lined up with the
beacon
		//move the robot right to center the beacon with the center of the camera frame
		sendMotorCommand(96, 32);
		return 1;
	}

	else if (pixelsFromCenter > 240) { //check to see if the center of the frame is lined up with
the beacon
		//move the robot right to center the beacon with the center of the camera frame
		sendMotorCommand(96, 32);
		return 1;
```

```
        }

        else if (pixelsFromCenter > 160) { //check to see if the center of the frame is lined up with
the beacon
                //move the robot right to center the beacon with the center of the camera frame
                sendMotorCommand(80, 48);
                return 1;
        }

        else if (pixelsFromCenter > 50) { //check to see if the center of the frame is lined up with
the beacon
                //move the robot right to center the beacon with the center of the camera frame
                sendMotorCommand(72, 56);
                return 1;
        }

        else if (pixelsFromCenter > -50) { //check to see if the center of the frame is lined up with
the beacon
                //move the robot left to center the beacon with the center of the camera frame
                //void sendMotorControllerSpeedBytes(int UART_PORT_ID, int
LeftYvalueControllerInput, int RightYvalueControllerInput)
                sendMotorCommand(56, 72);
                        return 1;
        }

        else if (pixelsFromCenter > -160) { //check to see if the center of the frame is lined up
with the beacon
                //move the robot left to center the beacon with the center of the camera frame
                sendMotorCommand(48, 80);
                return 1;
        }

        else if (pixelsFromCenter > -240) { //check to see if the center of the frame is lined up
with the beacon
                //move the robot left to center the beacon with the center of the camera frame
                sendMotorCommand(32, 96);
                return 1;
        }

        else if (pixelsFromCenter > -320) { //check to see if the center of the frame is lined up
with the beacon
                //move the robot left to center the beacon with the center of the camera frame
                sendMotorCommand(32, 96);
                return 1;
```

```
	}


	return 0;
}
```

<ObjectTracking.hpp>

```
#ifndef OBJECTTRACKING_HPP
#define OBJECTTRACKING_HPP

#include "Beacons.hpp"

//This allows the user to calibrate the HSV threshold color filter to detect an object within the camera image
class MouseCalibrateFilter {
public:
	//declare variables to use for using mouse rectangle area to get minimum and maximum HSV values automatically
	//NOTE: THIS FEATURE WILL ONLY WORK WHEN IN CALIBRATION_MODE (set in defs.hpp)
	bool mouseIsDragging;//used for showing a rectangle on screen as user clicks and drags mouse
	bool mouseMove;
	bool rectangleSelected;
	//keep track of initial point clicked and current position of mouse
	Point currentMousePoint;
	Point initialClickPoint;
	Rect rectangleROI; //this is the ROI that the user has selected
	vector<int> H_ROI, S_ROI, V_ROI;// HSV values from the click/drag ROI region stored in separate vectors so that we can sort them easily
};

//initial min and max HSV filter values.
//these will be changed using trackbars
extern int H_MIN;
extern int H_MAX;
extern int S_MIN;
extern int S_MAX;
extern int V_MIN;
extern int V_MAX;

//functions used for calibrating the HSV values used for filtering the color detections
```

```
void clickAndDragRectangle(int event, int x, int y, int flags, void* param);
void mouseRecordHSV_Values(Mat frame, Mat hsv_frame);

int alignWithBeacon(int);

//Tracking Library Function Declarations
extern void imageProcessingRoutine(void);
void on_trackbar(int, void*);
void createObjectTrackingParameterTrackbars(void);
void morphOps(Mat &thresh);
static double angle(Point pt1, Point pt2, Point pt0);
void setLabel(Mat& im, const string label, vector<Point> &contour);
void drawContours(Mat& image, const vector<vector<Point> > &contours, string title);
void shapeDetection(Mat& inputImage, vector<vector<Point> > contours, vector<Vec4i>
hierarchy, Mat& outputImage);
size_t calibratingTrackColorFilteredObjects(Mat &InputMat, Mat &HSV, vector<vector<Point> >
&contours, vector<Vec4i> &hierarchy, Mat &threshold);
size_t trackColorFilteredObjects(Mat &InputMat, Mat &HSV, vector<Beacon> &theBeacon,
vector<vector<Point> > &contours, vector<Vec4i> hierarchy, Mat &threshold);
void RecordBeaconPosition(Beacon &theBeacon, vector<vector<Point> > &contours,
vector<Beacon> &theBeaconsVector);
void DrawTarget(int x, int y, Mat &frame);
string intToString(int number);
int chooseBeaconToShootAt(void);
extern void sendMotorControllerSpeedBytes(int UART_PORT_ID, int LeftYvalueControllerInput,
int RightYvalueControllerInput);


#endif /* OBJECTTRACKING_HPP */
```

iii. Xbox 360 Controller

<Xbox360Controller.cpp>

```
/* this is the linux kernel 2.2.x way of handling joysticks using the xpad driver. It allows an arbitrary
* number of axis and buttons. It's event driven, and has full signed int
* ranges of the axis (-32768 to 32767). It also lets you pull the joysticks
* name. The only place this works of that I know of is in the linux 1.x
* joystick driver, which is included in the linux 2.2.x kernels
*/
/* Be sure to install the Wiring Pi library on the Raspberry pi
This code works on Raspberry Pi 2, but I am not sure about RPi1 or RPi3
Guide: http://wiringpi.com/download-and-install/
*/

#include "defs.hpp"
#include <wiringPi.h> //Utilize the "WiringPi GPIO library"
#include "Xbox360Controller.hpp"
#include "GPIO_UART.hpp"

//Joystick Interfacing with Linux Event File js0
int joy_fd, num_of_axis = 0, num_of_buttons = 0, x;
char name_of_joystick[80];
struct js_event js; //Raw input from controller event

//These are the raw input values from the controller (copied from "struct js_event js")
int axis[6];
bool button[11];

#define JOY_DEV "/dev/input/js0" //Define the device that the controller data is pulled from

int initController(void) {

	if ((joy_fd = open(JOY_DEV, O_RDONLY)) == -1) {
		printf("Couldn't open joystick\n");
		return (-1);
	}

	ioctl(joy_fd, JSIOCGAXES, &num_of_axis);
	ioctl(joy_fd, JSIOCGBUTTONS, &num_of_buttons);
	ioctl(joy_fd, JSIOCGNAME(80), &name_of_joystick);

	printf("Joystick detected: %s\n\t%d axis\n\t%d buttons\n\n"
		, name_of_joystick
```

```
		, num_of_axis
		, num_of_buttons);

	fcntl(joy_fd, F_SETFL, O_NONBLOCK);   /* use non-blocking mode */

	return 1;
}


int parseXbox360Controller(void) {

	//index used for parsing the input wireless Xbox360 controller data from js0 event
	int i = 0;
	//used to run non-blocking delay for the GPIO pins
	static int nextMilliSecondCountGPIO = 0;
	static short launchedPuck = 0;
	//used to control the shutdown of the RPi2 using either GPIO or the Xbox360 controller
	static short shutdownCount = 0;

	/* read the joystick state */
	read(joy_fd, &js, sizeof(struct js_event));

	/* see what to do with the event */
	switch (js.type & ~JS_EVENT_INIT) {

		case JS_EVENT_AXIS:
			axis[js.number] = js.value;
			break;
		case JS_EVENT_BUTTON:
			button[js.number] = js.value;
			break;
	}


	//Check to see if the controller is spitting back useful data (if not, reset/ignore the
incoming data)
	if (goodData == 0) {
		for (i = 0; i < sizeof(axis); i++) {
			axis[i] = 0;
		}
		for (i = 0; i < sizeof(button); i++) {
			button[i] = 0;
		}
		js.value = 0;
```

```
	}


	//Assign Variables
	Lx = axis[0];
	Ly = -axis[1];
	if (num_of_axis > 2) Lt = axis[2];
	if (num_of_axis > 3) {
		Rx = axis[3];
		Ry = -axis[4];
	}
	if (num_of_axis > 4) Rt = axis[5];

	Ba = button[0];
	Bb = button[1];
	Bx = button[2];
	By = button[3];
	BlBump = button[4];
	BrBump = button[5];
	Bsel = button[6];
	Bstart = button[7];
	BlStick = button[8];
	BxboxCenterIcon = button[9];
	BrStick = button[10];

	//Check to see if the buttons are in their neutral state
	if (!BxboxCenterIcon) { //if center button is pressed, dont do anything (center button is
software E-Stop for robot)
		if (!Bstart && !Bsel && !BlStick && !BrStick) {
			if (!BlBump) {
				goodData = 1;
			}
		}
	}else {
		goodData = 0;
		return 0;
	}


	//check to see if the wireless Xbox360 controller is giving valid data and if so, start using
the GPIO pins in the RPi2
	if (goodData == 1) {

#ifdef SOFTWARE_EMERGENCY_STOP
```

```
		while (BxboxCenterIcon == 1) { //if center button is pressed, dont do anything
			sendMotorControllerSpeedBytes(UART_ID, 64, 192); //halt motors
			digitalWrite(shootPinOutput, LOW);
			int breakBeamLEDOutput = 0;					//GPIO pin 17
output a test output for the Break Beam
			int shootPinOutput = 1;						//GPIO
pin 18 output controls the solenoid discrete output
			int controllerConnectedLEDOutput = 3;				//GPIO
pin 22 output controls the solenoid discrete output
			int enableAndGateOutput = 4;					//GPIO
pin 23 output controls the solenoid discrete output
		}
#endif

		if (millis() > nextMilliSecondCountGPIO) {

			digitalWrite(controllerConnectedLEDOutput, HIGH);

			//BrBump is override for shooting permissive
			if ((BrBump) || (shootPermissive)) {
				if (Ba == 1) {
					digitalWrite(shootPinOutput, HIGH);
					launchedPuck++;
					printf("launchedPuck %d times\r\n", launchedPuck);
					shootPermissive = 0;
				}
			}

			nextMilliSecondCountGPIO += 300;
		}

		if (Ba == 0) {
			digitalWrite(shootPinOutput, LOW);
		}

		//Read active high input for breakBeam sensor (garage door obstruction sensor)
		if (digitalRead(breakBeamInput) == 1) {
			digitalWrite(breakBeamLEDOutput, HIGH);
		}
		else {
			digitalWrite(breakBeamLEDOutput, LOW);
		}

		//shutdown the RPi2 safely using either the controller or the controller buttons
```

```
		if ((digitalRead(shutdownPiSwitchInput) == 1) || (Ba && Bb && Bx && By)) {
			if (shutdownCount > 5) {
				system("sudo shutdown -P now");
			}
			shutdownCount++;
		}else {
			shutdownCount = 0; //reset shutdown counter
		}


		sendMotorControllerSpeedBytes(UART_ID, Ly, Ry); //send left and right joystick
scaled values to Sabertooth 2x25 motor controller using UART


#ifdef PRINT_CONTROLLER_DEBUG_DATA
		printf("\r\n%d,%d,%d,%d, %d,%d,%d,%d, %d,%d,%d: ", Ba, Bb, Bx, By, BlBump,
BrBump, Bsel, Bstart, BlStick, BxboxCenterIcon, BrStick);
		printf("\r\n%d, %d, %d, %d, %d, %d", Lx, Ly, Lt, Rx, Ry, Rt);
		printf("\r\n");
		fflush(stdout);
#endif //PRINT_CONTROLLER_DEBUG_DATA


		printf("  \r\n");
		fflush(stdout);

		return 1;
	}
}
```

<Xbox360Controller.hpp>

```
#ifndef XBOX360CONTROLLER_HPP
#define XBOX360CONTROLLER_HPP


#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/joystick.h>


//Function Declarations
int initController(void);
```

```
int parseXbox360Controller(void);
extern void sendMotorControllerSpeedBytes(int UART_PORT_ID, int LeftYvalueControllerInput,
int RightYvalueControllerInput);


//THESE ARE THE PROTECTED CONTROLLER VALUES
//THESE VARIABLES ARE THE FINAL OUTPUT VALUES THAT CAN BE USED FOR THE CONTROLLER
//THIS WAS IMPLEMENTED BECAUSE THE ***CONTROLLER INPUTS ALL 1's WHEN IT
CONNECTS***
//TO GET PAST THIS ISSUE, THE CODE WAITS UNTIL THE BUTTONS RETURN TO A ZERO STATE
BEFORE IT CONTINUES
//Declare all buttons (including select,start along with leftstick & rightstick presses
extern bool Ba, Bb, Bx, By, BlBump, BrBump, Bsel, Bstart, BlStick, BrStick, BxboxCenterIcon;

//Declare all joysticks (16 bit signed integers)
extern int Lx, Ly, Rx, Ry, Lt, Rt;


#endif //XBOX360CONTROLLER_HPP
```

iv. GPIO and UART Communication

<GPIO_UART.cpp>

```
//File utilizes some of the I/O on the pin headers for the RaspberryPi2
//The pins used are discrete I/O and UART (TX only, not RX)


#include "defs.hpp"
#include "GPIO_UART.hpp"
#include <unistd.h>                    //Used for UART
#include <fcntl.h>                     //Used for UART
#include <termios.h>                   //Used for UART
#include <wiringPi.h>
#include <wiringSerial.h>
#include <errno.h>


//#define bool _Bool //I had to use booleans ("bool"), but Linux uses "_Bool" for boolean
variables
#define JOY_DEV "/dev/input/js0" //Define the device that the controller data is pulled from
```

```
#define UART_TXD0 "/dev/ttyAMA0"

/*
//PIN ASSIGNMENTS -- Discrete Inputs/Outputs:
//NOTE: THESE USE BROADCOM NUMBERS SINCE WiringPi DID NOT MAP THEM CORRECTLY
//NOTE: Output at end of variable means "real-world output"
//PLEASE LOOK HERE FOR CORRECT PINOUT DIAGRAM:
http://wiringpi.com/pins/

int breakBeamInput = 2;                                    //GPIO pin 27 input from
break beam (garage-door-like sensor)
int shutdownPiSwitchInput = 5;                             //GPIO pin 24 input to run script to
nicely power off RPi2 PowerLED

//Input at end of variable means "real-world input"
int breakBeamLEDOutput = 0;                                //GPIO pin 17 output a test LED
output for the Break Beam
int shootPinOutput = 1;                                    //GPIO pin 18 output controls
the discrete output solenoid valve for the launching mechanism
int controllerConnectedLEDOutput = 3;                      //GPIO pin 22 output controls the solenoid
discrete output
int enableAndGateOutput = 4;                               //GPIO pin 23 output permits the
motor controller and shootpin from turning on during the Pi's boot-up (the Pi turns all I/O on
during bootup)
*/

//initialize the GPIO and UART pins for the Raspberry Pi 2
int initGPIO_Uart(void) {

        //manually configure the GPIO pins for inputs or outputs using terminal commands
        //terminal commands: https://projects.drogon.net/raspberry-pi/wiringpi/the-gpio-utility/
        system("gpio mode 0 out"); //set GPIO pin 1 to output (breakBeamLEDOutput pin 17)
indicates puck is held by robot
        system("gpio mode 1 out"); //set GPIO pin 1 to output (shootpin pin 18)
        system("gpio mode 3 out"); //set GPIO pin 3 to output (controllerConnectedLEDOutput
pin 22)
        system("gpio mode 4 out"); //set GPIO pin 4 to output //GPIO pin 23 output controls the
solenoid discrete output

        system("gpio mode 2 in"); //set GPIO pin 2 to input //GPIO pin 27 input from break beam
(garage-door-like sensor)
        system("gpio mode 5 in"); //set GPIO pin 5 to input //GPIO pin 24 input from break beam
(garage-door-like sensor)
```

```
	//Initialize the Wiring Pi Libary
	pinMode(breakBeamInput, INPUT);
	pullUpDnControl(breakBeamInput, PUD_UP); // Enable pull-down resistor on button
	pinMode(shutdownPiSwitchInput, INPUT);
	pullUpDnControl(shutdownPiSwitchInput, PUD_DOWN); // Enable pull-down resistor on
button

	pinMode(breakBeamLEDOutput, OUTPUT);
	pinMode(shootPinOutput, OUTPUT);
	pinMode(controllerConnectedLEDOutput, OUTPUT);
	pinMode(enableAndGateOutput, OUTPUT);


	//initialize the UART @ 19200 BAUD
	if ((UART_ID = serialOpen(UART_TXD0, 19200)) < 0) {
		fprintf(stderr, "Unable to open serial device: %s\n", strerror(errno));
		return 0;
	}

	//Initialize WiringPi -- using Broadcom processor pin numbers
	wiringPiSetupGpio();

	digitalWrite(enableAndGateOutput, HIGH); //enable the AND GATE and allow the UART
and breakBeam outputs to turn on

	usleep(2000000); //wait 2 seconds (in microseconds) to act as a power up delay for the
Sabertooth Motor Controller
	serialPutchar(UART_ID, 0xAA); //Send the autobauding character to Sabertooth first to
stop motors from twitching!
	usleep(100000); //wait 100ms (in microseconds) before commanding motors


	if (wiringPiSetup() == -1) {
		fprintf(stdout, "Unable to start wiringPi: %s\n", strerror(errno));
		return 0;
	}

	printf("Initialized GPIO and UART!\r\n");
	//if return(0), something did not get initialized correctly
}
```

```
//Feed Xbox controller Joystick (16-bit integer using built-in xpad driver in Linux) as an input
//and send the scaled output to the "Sabertooth 2x25 Motor Controller V2.00" as a Simplified Serial Input
void sendMotorControllerSpeedBytes(int UART_PORT_ID, int LeftYvalueControllerInput, int RightYvalueControllerInput) {
	//Left Motor:		0: Full reverse		64: Neutral		127: Full Forward
	//Right Motor:		128: Full reverse		192: Neutral		255: Full Forward

	static int maxControllerJoystickInput = 32767;
	static int DEADZONE = 16384; //uses only half of the joystick range as usable values (32768/2 == 2^14 == 16384)
	unsigned char RightMotorSerialOutput = 0, LeftMotorSerialOutput = 0;
	static int Ldelta = 127 - 63; //will have actual slope of ((127-64)/(2^14)) or ((127-64)>>14), but that will be later using fixed point and bit shifting
	static int Rdelta = 255 - 191; //will have actual slope of ((255-192)/(2^14)) or ((255-192)>>14), but that will be later using fixed point and bit shifting
	//millis() values for running parts of the code routinely, but allow the code to be "unblocked" without using delays
	int nextMilliSecondCountLeftUART = 0, nextMilliSecondCountRightUART = 0;

	//check if the controller is outside the y-axis dead zone for each joystick
	if (abs(LeftYvalueControllerInput) > DEADZONE) {

		bool negLeftInput = 0;

		if (LeftYvalueControllerInput < 0) { //set negative input flag and convert to positive value
			LeftYvalueControllerInput = (unsigned int)LeftYvalueControllerInput; //make joystick positive
			negLeftInput = 1;
		}
		//input is positive
		if (negLeftInput == 0) { //input is positive
			LeftMotorSerialOutput = (unsigned char)(64 + (((LeftYvalueControllerInput - DEADZONE)*Ldelta) >> 14)); //64 is motr controller offset for Left Motor Neutral
		}
		//input is negative
		if (negLeftInput == 1) { //input is negative
			LeftMotorSerialOutput = (unsigned char)(-(64 - (((LeftYvalueControllerInput - DEADZONE)*Ldelta) >> 14))); //64 is motr controller offset for Left Motor Neutral
		}
```

```
		//clip the maximum allowed magnitude for y-axis joystick at their expected maximum values
		if (LeftMotorSerialOutput > 126) LeftMotorSerialOutput = 127;

		//clip the minimum allowed magnitude for y-axis joystick at their expected minimum values
		if (LeftMotorSerialOutput < 2) LeftMotorSerialOutput = 1;
	}

	//if controller y-values are within DEADZONE, command the left motor to neutral (off)
	else {
		LeftMotorSerialOutput = 64;
	}

	//check if the controller is outside the y-axis dead zone for each joystick
	if (abs(RightYvalueControllerInput) > DEADZONE) {

		bool negRightInput = 0;

		if (RightYvalueControllerInput < 0) { //set negative input flag and convert to positive value
			RightYvalueControllerInput = (unsigned int)RightYvalueControllerInput; //make joystick positive
			negRightInput = 1;
		}

		if (negRightInput == 0) { //input is positive
			RightMotorSerialOutput = (unsigned char)(192 + (((RightYvalueControllerInput - DEADZONE)*Rdelta) >> 14)); //192 is motr controller offset for Right Motor Neutral
		}

		if (negRightInput == 1) { //input is negative
			RightMotorSerialOutput = (unsigned char)(-(192 - (((RightYvalueControllerInput - DEADZONE)*Rdelta) >> 14))); //192 is motr controller offset for Right Motor Neutral
		}

		//clip the maximum allowed magnitude for y-axis joystick at their expected maximum values
		if (RightMotorSerialOutput > 254) RightMotorSerialOutput = 255;
```

```
                //clip the minimum allowed magnitude for y-axis joystick at their expected
minimum values
                if (RightMotorSerialOutput < 129) RightMotorSerialOutput = 128;
        }

        //if controller y-values are within DEADZONE, command the right motor to neutral (off)
        else {
                RightMotorSerialOutput = 192;
        }


#ifdef PRINT_SERIAL_DATA
        printf("Serial_L:%d\r\n", LeftMotorSerialOutput);
        printf("Serial_R:%d\r\n", RightMotorSerialOutput);
#endif //PRINT_SERIAL_DATA


        if (millis() > nextMilliSecondCountLeftUART) {
                serialPutchar(UART_PORT_ID, LeftMotorSerialOutput);
                nextMilliSecondCountLeftUART += 120; //run this "if() statement in 120
milliseconds
        }

        if (millis() > nextMilliSecondCountRightUART) {
                serialPutchar(UART_PORT_ID, RightMotorSerialOutput);
                nextMilliSecondCountRightUART += 80; //run this "if() statement in 80
milliseconds
        }

        printf("afterMotorDataSend\r\n");
}


int gpioPinOperations(void) {

        int nextMilliSecondCountLEDS = 0;
        int nextMilliSecondCountGPIO = 0;
        int nextMilliSecondCountBreakBeam = 0;
        static short shutdownCount = 0;

        /*Now read the input GPIO pins for control necessary pins that need a higher update
rate*/
        if (millis() > nextMilliSecondCountLEDS) {
```

```
		//goodData not set, so controller not connected
		if (goodData == 0) {
			printf("BadData\r\n");
			digitalWrite(controllerConnectedLEDOutput, LOW);
		}

		//detect whether a puck is in the launching cavity. If so, permit the user to use the
shooting mechanism
		if ( digitalRead(breakBeamInput) == 1) {
			printf("BreakBeam ON\r\n");
			digitalWrite(breakBeamLEDOutput, HIGH);
			shootPermissive = 1;
		}
		else {
			digitalWrite(breakBeamLEDOutput, LOW);
			printf("BreakBeam OFF\r\n");
		}
		nextMilliSecondCountLEDS += 150;
	}


#ifdef PERMIT_SHUTDOWN_PI_USING_GPIO_OR_CONTROLLER
	/*Now read the input GPIO pins for things that dont need a high update rate*/
	if (millis() > nextMilliSecondCountGPIO) {
		if ((digitalRead(shutdownPiSwitchInput) == 1) || (BxboxCenterIcon == 1)) {
			//if shutdown counter reaches 10 seconds, shutdown the Pi
			if (shutdownCount > 10) {
				//system("sudo shutdown -k now"); //send test shutdown message
				system("sudo shutdown -P now"); //shutdown Pi and power off
immediately
			}
			shutdownCount++;
		}
		else {
			shutdownCount = 0; //reset shutdown counter
		}

		nextMilliSecondCountGPIO += 1000; //check every 1000ms (1 time every second)
	}
#endif //PERMIT_SHUTDOWN_PI_USING_GPIO_OR_CONTROLLER


	return 1;
}
```

<GPIO_UART.hpp>

```
#ifndef UART_HPP
#define UART_HPP


int gpioPinOperations(void);
int initGPIO_Uart(void);
void sendMotorControllerSpeedBytes(int UART_PORT_ID, int LeftYvalueControllerInput, int
RightYvalueControllerInput);

#endif //UART_HPP
```

v. Beacon Detection

<Beacons.cpp>

```
#define USE_EXTERNS
#include "Beacons.hpp"


/*Define Shapes and Colors for Known Target Beacons:
Green Triangle
Blue Rectangle
Red Octagon
*/


Beacon::~Beacon() { // I never have to free-up the class, so this is not run...
	// nothing here in the deconstructor
}

Beacon::Beacon(string name) { // this is the class declaration that I am actually using...
	if (name == "GreenTriangle") {

		//TODO: use "calibration mode" to find HSV min
		//and HSV max values

		setHSVmin(Scalar(0, 0, 0));
		setHSVmax(Scalar(0, 255, 0));

		setColor(GREEN);
		setShape("Triangle");
	}

	if (name == "BlueRectangle") {

		//TODO: use "calibration mode" to find HSV min
		//and HSV max values

		setHSVmin(Scalar(0, 0, 0));
		setHSVmax(Scalar(255, 0, 0));

		setColor(BLUE);
		setShape("Rectangle");
	}
```

```
	if (name == "RedOctagon") {

		//TODO: use "calibration mode" to find HSV min
		//and HSV max values

		setHSVmin(Scalar(0, 0, 0));
		setHSVmax(Scalar(0, 0, 255));

		setColor(RED);
		setShape("Octagon");
	}


	if (name == "TestingObjectDetecting_NOT_Recording_Locations") { //do not record any
locations of the beacons (this beacon is used for testing)
		setShape("TestingObjectDetecting_NOT_Recording_Locations");
	}
}


int Beacon::getXPos() {
	return Beacon::xPos;
}
int Beacon::getYPos() {
	return Beacon::yPos;
}
void Beacon::setXPos(int x) {
	Beacon::xPos = x;
}
void Beacon::setYPos(int y) {
	Beacon::yPos = y;
}


Scalar Beacon::getHSVmin() {
	return Beacon::HSVmin;
}
Scalar Beacon::getHSVmax() {
	return Beacon::HSVmax;
}
void Beacon::setHSVmin(Scalar min) {
	Beacon::HSVmin = min;
}
void Beacon::setHSVmax(Scalar max) {
```

```
	Beacon::HSVmax = max;
}


Scalar Beacon::getColor() {
	return Color;
}
void Beacon::setColor(Scalar s) {
	Color = s;
}
String Beacon::getShape() {
	return Shape;
}
void Beacon::setShape(string s) {
	Shape = s;
}
```

<Beacons.hpp>

```
#ifndef BEACONS_HPP
#define BEACONS_HPP

#include "defs.hpp"
#include <string>

class Beacon {

public:
	Beacon(std::string name);
	Beacon() = delete;
	~Beacon();

	int getXPos();
	void setXPos(int x);
	int getYPos();
	void setYPos(int y);

	Scalar getHSVmin();
	Scalar getHSVmax();
	void setHSVmin(Scalar min);
	void setHSVmax(Scalar max);

	Scalar getColor();
	void setColor(Scalar c);
	String getShape();
```

```
        void setShape(string s);


private:
        int xPos, yPos;
        String Shape;
        Scalar Color;
        Scalar HSVmin, HSVmax;
};

#endif /* BEACONS_HPP */
```

vi. Variable Definitions

<defs.cpp>

```
#include "defs.hpp"

/* shared/global variables
 this_is_global;*/


////////////////////////////////////////////////////////////////////////////////////////////////////

/*GLOBAL VARIABLES TO SHARE WITH GPIO_UART.CPP*/
//flag used to eliminate random noise from when wireless Xbox360 controller connects with all values @ 1
int goodData = 0;
//user needs the breakbeam sensor to operate the puck launcher
int shootPermissive = 0;
//UART port ID for Tx to motor controller
int UART_ID = 0;

std::mutex inputLock;

////////////////////////////////////////////////////////////////////////////////////////////////////

//THESE ARE THE PROTECTED WIRELESS CONTROLLER VALUES FOR THE XBOX360 WIRELESS CONTROLLER
//THESE VARIABLES ARE THE FINAL OUTPUT VALUES THAT CAN BE USED FOR THE CONTROLLER
//THIS WAS IMPLEMENTED BECAUSE THE ***CONTROLLER INPUTS ALL 1's WHEN IT CONNECTS***
//TO GET PAST THIS ISSUE, THE CODE WAITS UNTIL THE BUTTONS RETURN TO A ZERO STATE BEFORE IT CONTINUES
//Declare all buttons (including select,start along with leftstick & rightstick presses
bool Ba = 0;
bool Bb = 0;
bool Bx = 0;
bool By = 0;
bool BlBump = 0;
bool BrBump = 0;
bool Bsel = 0;
bool Bstart = 0;
bool BlStick = 0;
bool BrStick = 0;
bool BxboxCenterIcon = 0;
```

```
//Declare all joysticks (16 bit signed integers)
int Lx = 0;
int Ly = 0;
int Rx = 0;
int Ry = 0;
int Lt = 0;
int Rt = 0;

////////////////////////////////////////////////////////////////////////////////////////////////////

//PIN ASSIGNMENTS -- Discrete Inputs/Outputs:
//NOTE: THESE USE BROADCOM NUMBERS SINCE WiringPi DID NOT MAP THEM CORRECTLY
//NOTE: Output at end of variable means "real-world output"
//PLEASE LOOK HERE FOR CORRECT PINOUT DIAGRAM:
//http://wiringpi.com/pins/

int breakBeamInput = 2;                                    //GPIO pin 27 input
from break beam (garage-door-like sensor)
int shutdownPiSwitchInput = 5;                             //GPIO pin 24 input to run
script to nicely power off RPi2 PowerLED

//Input at end of variable means "real-world input"
int breakBeamLEDOutput = 0;                                //GPIO pin 17 output a test
output for the Break Beam
int shootPinOutput = 1;                                    //GPIO pin 18 output
controls the solenoid discrete output
int controllerConnectedLEDOutput = 3;                      //GPIO pin 22 output controls the
solenoid discrete output
int enableAndGateOutput = 23;                              //GPIO pin 16 output controls
the output enable discrete output

////////////////////////////////////////////////////////////////////////////////////////////////////


//default display capture window frame width and height (640x480 window)
const int FRAME_WIDTH = 640;
const int FRAME_HEIGHT = 480;
```

<defs.hpp>

```
#include "opencv2/core/core.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/imgcodecs.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <iostream>
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <mutex>
#include <thread>

using namespace cv;
using namespace std;


#ifndef DEFS_HPP
#define DEFS_HPP

extern std::mutex inputLock;

////////////////////////////////////////////////////////////////////////////////////////////////////

/*GLOBAL VARIABLES TO SHARE WITH GPIO_UART.CPP*/
//flag used to eliminate random noise from when wireless Xbox360 controller connects with all values @ 1
extern int goodData;
//user needs the breakbeam sensor to operate the puck launcher
extern int shootPermissive;
//UART port ID for Tx to motor controller
extern int UART_ID;

////////////////////////////////////////////////////////////////////////////////////////////////////

//PIN ASSIGNMENTS -- Discrete Inputs/Outputs:
//NOTE: THESE USE BROADCOM NUMBERS SINCE WiringPi DID NOT MAP THEM CORRECTLY
//NOTE: Output at end of variable means "real-world output"
//PLEASE LOOK HERE FOR CORRECT PINOUT DIAGRAM:
//http://wiringpi.com/pins/

extern int breakBeamInput;                                          //GPIO pin 27 input from break beam (garage-door-like sensor)
```

```
extern int shutdownPiSwitchInput;                    //GPIO pin 24 input to run script to
nicely power off RPi2 PowerLED

//Input at end of variable means "real-world input"
extern int breakBeamLEDOutput;                       //GPIO pin 17 output a test
output for the Break Beam
extern int shootPinOutput;                           //GPIO pin 18 output controls
the solenoid discrete output
extern int controllerConnectedLEDOutput;             //GPIO pin 22 output controls the solenoid
discrete output
extern int enableAndGateOutput;                      //GPIO pin 23 output controls
the solenoid discrete output

////////////////////////////////////////////////////////////////////////////////////////////////////

//default display capture window frame width and height (640x480 window)
extern const int FRAME_WIDTH;
extern const int FRAME_HEIGHT;

////////////////////////////////////////////////////////////////////////////////////////////////////

//THESE ARE THE PROTECTED WIRELESS CONTROLLER VALUES FOR THE XBOX360 WIRELESS
CONTROLLER
//THESE VARIABLES ARE THE FINAL OUTPUT VALUES THAT CAN BE USED FOR THE CONTROLLER
//THIS WAS IMPLEMENTED BECAUSE THE ***CONTROLLER INPUTS ALL 1's WHEN IT
CONNECTS***
//TO GET PAST THIS ISSUE, THE CODE WAITS UNTIL THE BUTTONS RETURN TO A ZERO STATE
BEFORE IT CONTINUES
//Declare all buttons (including select,start along with leftstick & rightstick presses
extern bool Ba;
extern bool Bb;
extern bool Bx;
extern bool By;
extern bool BlBump;
extern bool BrBump;
extern bool Bsel;
extern bool Bstart;
extern bool BlStick;
extern bool BrStick;
extern bool BxboxCenterIcon;

//Declare all joysticks (16 bit signed integers)
extern int Lx;
extern int Ly;
```

```
extern int Rx;
extern int Ry;
extern int Lt;
extern int Rt;

////////////////////////////////////////////////////////////////////////////////////////////////////

#define PRINT_CONTROLLER_DATA 1
#define PRINT_SERIAL_DATA 1
//#define SOFTWARE_EMERGENCY_STOP 1
#define PERMIT_SHUTDOWN_PI_USING_GPIO_OR_CONTROLLER 1

//#define ShowDetectedObjects 1
#define SHOW_OPENCV_IMAGES 1
#define USING_WEBCAM 1 //flag that is set to control whether the user uses the camera for input or a still picture as an input
#define CALIBRATION_MODE 1 //calibrate the HSV filter for a specific color
#define CAMERA_NUMBER 0 //flag to set source of video: "camera 0" is the builtin laptop webcam, "camera 1" is usb webcam
#define MAX_NUM_OBJECTS 15 // Program will only track 30 objects at a time (this is just in case noise becomes a problem)
//#define MIN_OBJECT_AREA 400 //Only allow larger objects //200*200
//#define MIN_OBJECT_AREA 1000 //Only allow larger objects
#define MIN_OBJECT_AREA 20000 //Only allow larger objects
#define JOYSTICK_DEADZONE 10000 //only care about joystick values outside of the deadzone + & - around 0


#define BLUE (Scalar(255, 0, 0)) //BGR, not RGB (I do not know why colors are flipped for OPENCV
#define GREEN (Scalar(0, 255, 0))
#define RED (Scalar(0, 0, 255))
#define YELLOW (Scalar(0, 255, 255))
#define PURPLE (Scalar(255, 0, 255))
#define WHITE (Scalar(255, 255, 255))
#define BLACK (Scalar(0, 0, 0))

#endif /* DEFS_HPP */
```

E. Beacon Design

i. Beacon Software

```
// NeoPixel Ring simple sketch (c) 2013 Shae Erisson
// released under the GPLv3 license to match the rest of the AdaFruit NeoPixel library
//Modified by Keith Martin

#include <Adafruit_NeoPixel.h>
#ifdef __AVR__
  #include <avr/power.h>
#endif


//Choose which color the beacon should be for the ROOBockey LED beacon:
//Comment out the colors that you do not want the beacon to be
#define GREEN_BEACON 1
//#define RED_BEACON 1
//#define BLUE_BEACON 1
//#define YELLOW_BEACON 1


// Which pin on the Arduino is connected to the NeoPixels?
// On a Trinket or Gemma we suggest changing this to 1
#define PIN          11 //I chose to hook Data line for neopixels to pin "D11" with a 330 or 470
ohm resistor

// How many NeoPixels are attached to the Arduino?
//#define NUMPIXELS      10
#define NUMPIXELS      32

// When we setup the NeoPixel library, we tell it how many pixels, and which pin to use to send
signals.
// Note that for older NeoPixel strips you might need to change the third parameter--see the
strandtest
// example for more information on possible values.
Adafruit_NeoPixel pixels = Adafruit_NeoPixel(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);

int delayval = 100; // delay for half a second

void setup() {
  pixels.begin(); // This initializes the NeoPixel library.
}
```

```
void loop() {

    // For a set of NeoPixels the first NeoPixel is 0, second is 1, all the way up to the count of pixels minus one.

    for(int i=0;i<NUMPIXELS;i++) {

      // pixels.Color takes RGB values, from 0,0,0 up to 255,255,255
      #ifdef GREEN_BEACON
        pixels.setPixelColor(i, pixels.Color(0,255,0)); // green color
      #endif

      #ifdef RED_BEACON
        pixels.setPixelColor(i, pixels.Color(255,0,0)); // red color
      #endif

      #ifdef BLUE_BEACON
        pixels.setPixelColor(i, pixels.Color(0,0,255)); // blue color
      #endif

      #ifdef YELLOW_BEACON
        pixels.setPixelColor(i, pixels.Color(255,255,0)); // yellow color
      #endif

      pixels.show(); // This sends the updated pixel color to the hardware.

      delay(delayval); // Delay for a period of time (in milliseconds).

    }
}
```

F. Parts Request

Qty.	Part Num.	Description	Suggested Vendor	Unit Cost	Total Cost
1	DEV-13724	Raspberry Pi 2 - Model B (8GB Bundle)	Sparkfun	$39.99	$ 39.99
1	Genius WideCam F100	Wide Angle Camera Module	Newegg	$33.50	33.50
1	5mm LED Version	IR Beam Break Sensor	Adafruit	$6.50	6.50
1		Electrical Components	DigiKey	$61.23	61.23
	282836-2	TERM BLOCK 2POS SIDE ENTRY 5MM	DigiKey		-
	GRM319R61E106KA12D	CAP CER 10UF 25V X5R 1206	DigiKey		-
	ERJ-6ENF4531V	RES SMD 4.53K OHM 1% 1/8W 0805	DigiKey		
	ERJ-6ENF3402V	RES SMD 34K OHM 1% 1/8W 0805	DigiKey		
	GRM216R71H103KA01D	CAP CER 10000PF 50V X7R 0805	DigiKey		
	ERJ-6ENF9092V	RES SMD 90.9K OHM 1% 1/8W 0805	DigiKey		
	LMZ12003TZX-ADJ/NOPB	IC BUCK SYNC ADJ 3A TO-PMOD-7	DigiKey		
	ERJ-6ENF1071V	RES SMD 1.07K OHM 1% 1/8W 0805	DigiKey		
	ERJ-6ENF5621V	RES SMD 5.62K OHM 1% 1/8W 0805	DigiKey		
	C0805C223K5RACTU	CAP CER 0.022UF 50V X7R 0805	DigiKey		
	GRM31CR60J107ME39L	CAP CER 100UF 6.3V X5R 1206	DigiKey		
				Total:	$141.22

Qty.	Part Num.	Description	Suggested Vendor	Unit Cost	Total Cost
2	1102	19:1 Metal Gearmotor 37Dx52L mm	Pololu	$24.95	$49.90
2	1084	Stamped Aluminum L-Bracket Pair for 37D mm Metal Gearmotors	Pololu	$7.95	$15.90
2	3275	Scooter/Skate Wheel 84×24mm - Black	Pololu	$2.95	$5.90
2	2674	Aluminum Scooter Wheel Adapter for 6mm Shaft	Pololu	$4.95	$9.90
1	0715-CO2-PBG	CO2 Regulator (No Tank)	FrightProps	$79.99	$79.99
1	0923-0001	4-Way 5-Port Valve with 1/4 inch ports + 3 x 1/4 Threads - 1/4 Tubing at 24V	FrightProps	$27.81	$27.81
2	0742-0349	1/8 Threads - 1/4 Tubing	FrightProps	$0.78	$1.56
1	0738-0336	Quick Exhaust Valve 1/8 IN/OUT, 1/4 Exhaust + 1/8 Threads	FrightProps	$12.20	$12.20
50	0714-0009	1/4 inch Polyethylene Airline tubing (Price per Foot)	FrightProps	$0.10	$5.00
1	5TKX2	Air Cylinder 8mm Bore Diameter, 50 mm stroke Length	Grainger	$36.45	$36.45
1	5TLH9	Cylinder Foot Bracket	Grainger	$10.28	$10.28
				Total:	$254.89

Qty.	Part Num.	Description	Suggested Vendor	Unit Cost	Total Cost
2	120NP-AM5	Brass Pipe Adaptor 1/8" NPT Female - M5 Male Nickel Plated, REDUCER	Mettle Air	$2.54	$5.08
1	8574K22	Impact-Resistant Polycarbonate Sheet, 3/16" Thick, 24" x 48", Clear	McMaster Carr	$63.65	$63.65
2	1088A31	Inside Corner-Reinforcing Bracket - 2" Length of Sides	McMaster Carr	$1.93	$3.86
12	1556A24	Bracket - Zinc-Plated Steel, 7/8" Length of Sides	McMaster Carr	$0.43	$5.16
				Total:	$77.75

Qty.	Description	Suggested Vendor	Vendor Part Num.	Unit Cost	Total Cost
1	Raspberry Pi Model B+ / Pi 2 Case Lid - Purple	Adafruit		$5.00	$5.00
1	Pi Model B+ / Pi 2 Case Base - Purple	Adafruit		$3.00	$3.00
1	Microsoft JR9-00011 Xbox 360 Wireless Controller for Windows	Newegg	N82E16823109243	$39.95	$39.95
1	PCB for Power Board (Includes 3 copies)	OSH Park		$12.50	$12.50
1	DigiKey Board Components	Digikey		$55.35	$55.35
3	IC REG CTRL BST FLYBK INV 10MSOP	Digikey	LT3757AIMSE#PBF-ND		
50	RES SMD 43.2K OHM 1% 1/10W 0603	Digikey	P43.2KHCT-ND		
50	RES SMD 200K OHM 1% 1/10W 0603	Digikey	P200KHCT-ND		
5	FIXED IND 10UH 10A 16.8 MOHM SMD	Digikey	SRP1270-100MCT-ND		
10	MOSFET N-CH 40V 19A 8SOIC	Digikey	SI4840BDY-T1-E3CT-ND		
10	CAP CER 0.1UF 16V X7R 0603	Digikey	490-1532-1-ND		
10	CAP CER 4.7UF 16V X5R 0603	Digikey	490-10481-1-ND		
10	CAP CER 100PF 50V NP0 0603	Digikey	490-1427-1-ND		
10	CAP CER 6800PF 25V X7R 0603	Digikey	490-11530-1-ND		
10	RES SMD 22K OHM 1% 1/4W 0603	Digikey	P22KBYCT-ND		
50	RES SMD 41.2K OHM 1% 1/10W 0603	Digikey	P41.2KHCT-ND		
5	RES SMD 0.01 OHM 1% 1/4W 1206	Digikey	WSLC-.01CT-ND		
50	RES SMD 226K OHM 1% 1/10W 0603	Digikey	P226KHCT-ND		
50	RES SMD 16.2K OHM 1% 1/10W 0603	Digikey	P16.2KHCT-ND		
10	RES SMD 100 OHM 1% 1/4W 0603	Digikey	P100BYCT-ND		
15	CAP ALUM 47UF 20% 35V SMD	Digikey	PCE3842CT-ND		
				Total:	$115.80