Abstract

The main objective of this project was to program the MechBot and use the Lego Mindstorms Lego kit to simulate a factory Automated Guided Vehicle (AGV). This was accomplished by making use of a variety of sensors on the MechBot as well as the EV shield which controlled the motorized arm that collected and delivered a payload to different locations. This obstacle course was constructed to mimic a typical industrial plant. The line and distance sensors were calibrated in order to allow the MechBot to sense obstacles such as the pop cans which had to be collected, as well as being able to sense electrical tape to stay on course throughout its operation. A camera was also hoisted within the Lego setup in order to detect color of specific pop cans that were located along the track. The MechBot had to complete a number of tasks all of which had to be accomplished autonomously which meant that the MechBot could not be touched during operation. Throughout the course of the project a number of technical issues were encountered and there were several phases where the code generated required extensive debugging as well as phases where the entire code had to be regenerated to account for a number of errors. Overall the programming phase was successful given that it was able to solve similar tasks that were expected to be encountered during the competition. The success of the project is still pending as the competition date has been scheduled for the same day as the report deadline. Enclosed in this report is a detailed analysis of the problem, possible solutions and discussion of technical issues and the code that was used to tackle the problem.

Table of Contents

1.0 Introduction

Automated Guide Vehicles are used consistently in industrial plants as they improve the efficiency and productivity of the plant itself. The project assigned to the design team was to program the MechBot to simulate a AGV that would pass through several obstacles and complete specific tasks. The MechBot was required to complete the following tasks:

1. Follow a line of electrical tape and stop at four T-junctions
2. Pick up cans at two of junctions
3. Drop off the cans at the other two junctions
4. Remove garbage of the track and continue its path

The first task was to follow the line of electrical tape until it met the first T-junction where the MechBot was required to stop and pick up the first pop can [1]. Upon completion of this obstacle, the MechBot was to continue following the line of electrical tape where it would encounter the second T-junction where it would have to drop the can off. The challenge with this these first two tasks was programming the AGV to recognize the color of the can and drop it off at the appropriate color coordinated drop off location. The first two tasks were repeated at the third and fourth T-junctions. However, in the middle of the obstacle course stood a piece of garbage that was placed on the track to test the AGVs safety system. The two bumper sensors would hit the obstruction and cause the MechBot to stop and remove the garbage out of its path before continuing its path. Each of the tasks came with its own set of challenges which if completed meant earning more extra points during the competition. Other challenges and how they were solved are discussed in this report.

2.0 Literature Review

Autonomous robots are a new emerging technology that have shaped our world for the better. In today's world, robots have the distinct ability of sensing their environment and adapting to various conditions that are thrown at them. The ability to think for themselves and account for their errors make them truly remarkable. Self-driving cars are now being developed and used on the road, no longer requiring the intervention of a human. This exciting concept is now possible since these machines are now able to sense the world around them and perform the task at hand, safely and efficiently.

The MechBot used in this project is also capable of being autonomous, however it must be programmed to do so by the user first. There is a total of four line sensors located at the bottom front, and three distance sensors located at the left, front and right of the robot. By initializing these sensors, the robot can perceive its surroundings better and correct its path along the way. Logical cases are often set up by the user prior to the test, so that the MechBot can perceive various roadblocks in its path and find a way to solve them efficiently. The four line sensors located at the bottom of the robot are responsible in sensing the black tape placed on the ground. If the sensors have been initialized properly by the user, the MechBot line sensors should aid the robot in following the tape and staying on the desired path. Problems arise however when the robot veers of the taped path and therefore is at a loss at what to do, since the line sensors are no longer playing a role in guiding the robot's motion.

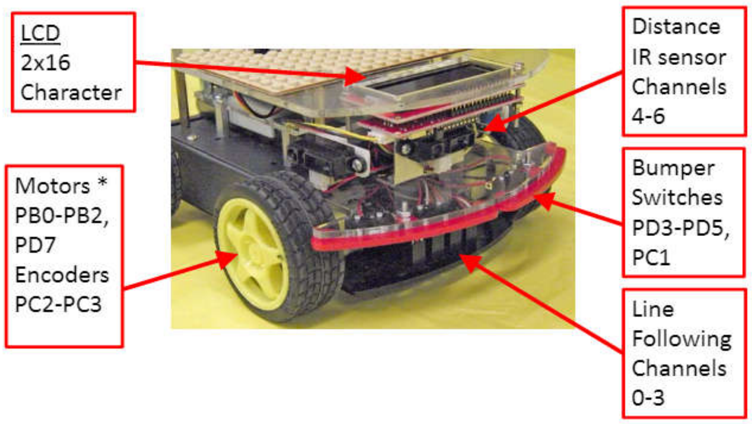


Figure 1: MechBot Components and sensors[1]

In this project, the array of sensors on the MechBot were utilized in conjunction with the EVShield. Below, each sensor and EVShield component is described independently, also the control system utilized for the system is also discussed.

2.1 Infrared Sensor

An Infrared Radiation sensor (IR) is a photoelectric sensor that works in the 750nm frequency range of the EM spectrum. It consists of a receiver and an emitter. Depending on the type of use (such as TV remote control to military operations) the range can span from mm and extend towards any distance if enough intensity is matched). This sensor functions by looking for discrepancies within the signals it sends out and the signals it receives. Any alteration signifies that something changed the conditions and then can be programmed to give a response. The receiver uses a light sensitive transistor so that it can be fine-tuned to the light produced by the emitter, hence reducing variations of different signals.

2.1.1 Distance Sensor

Distance sensor as the name implies is a sensor that measures how far an object is relative to its own position. As the object comes closer, the distance values decrease and as the object moves farther away, the distance values increase. It generates an analog signal between 0 and 3.2V depending on the distance of the object from the sensor. The distance is measured using the difference in time from the emission of the IR beam to the sensor detecting its reflection and then dividing that by the speed of light. Using this technique, it can calculate an arbitrary distance relative to its own position that an object is a certain amount of mm or m away from it if a straight line was drawn from the sensor to the object.

Figure 2: SHARP GP2D12 IR Distance Sensor [5]

2.1.2 Line Sensor

The Line following sensor makes use of the Infrared sensors on the MechBot in order to detect lines or objects underneath the sensor. These sensors will detect the black tape placed on the ground and give enough information to the MechBot to stay on the line. The sensor will first emit an LED light onto the surface. Since the electric tape is black, it will absorb most of this light and tell the sensor that it is on a dark line. This is due to the change in reflectivity of the light. The opposite phenomenon occurs on a more reflective surface. If the reflective light received is almost the same as what was emitted, it cannot be on a dark surface. Thus, this allows the MechBot to make a distinction between dark and light lines. The threshold values for light that the receiver gains are ultimately used to calibrate the line sensors on the MechBot. Obtaining the correct calibration is important, since it will determine how accurately the MechBot follows the black line of tape. These values will be programmed into the code to make the MechBot completely autonomous.

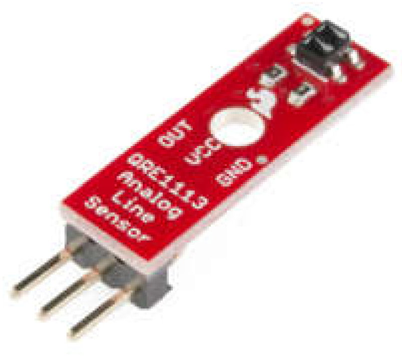


Figure 3: Line following Sensor for a MechBot [4]

2.2 EVShield & Peripherals

The EVShield is a third-party shield developed for the Arduino Uno and Duemilanove. It has pin headers that connect directly to the Arduino Uno's pins. It utilizes some of those pins for its own functions and lets the other pins bypass for further use. The shield provides, connection ports to Lego Mindstorms and NXT peripherals, thus making it very easy to develop bots without using custom components. It has 4 motor ports, 4 utility ports for the other sensors and built in LEDs. All the motors and sensors that connect to the shield must be EV3 or NXT compatible.

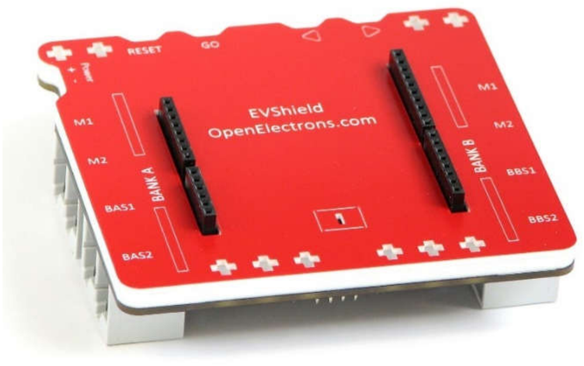


Figure 4: EVShield without the Arduino Uno connected [3]

The NXT motor is a multipurpose motor which can act like both servo and a normal DC motor. It contains an encoder which can be used to turn the motor precisely by degrees, but unlike normal servos, this motor can turn full 360 degrees. It can also run for an indefinite amount of time at a given speed.

The NXT cam is a low-resolution cam which can be used to detect colors of objects in front of it. The EVShield library for Arduino IDE contains functions that return the colors detected by the cam, the range of this camera also only extends to a few meters.

Figure 5: The NXT motors used for the arms and claws [7]

Figure 6: NXT Cam used to detect the color of the can [6]

2.3 PID Control

PID stands for Proportional, Integral, Derivative control. Its essence is to tune the incoming feedback error and realign it every time, so that it minimizes the error with every increment. It's used when there is a closed loop control. Closed loop control depends on a feedback to the system, and must control Response quality, Stability and Robustness. It will compare the feedback signal with the command signal and change the output if necessary. To use PID for MecBot it must need a feedback signal. The feedback signal it can acquire is the motor speed, which can be sensed by the encoder sensor (by measuring the pulses). This can be the feedback into the system.

Proportional constant accounts for present values of error; therefore, if values are really big, the control output will also be big, and same goes for if the values are small. Based on the current error between the set point and current output. This shows itself as a smooth control (instead of spiky control).

Integral constant accounts for past values of error. It constantly accumulates error, if the output is not meeting the requirements and the controller will apply a stronger action to it.

Derivative constant accounts for future values of error, and this is done by the speed of the error between output and set point.

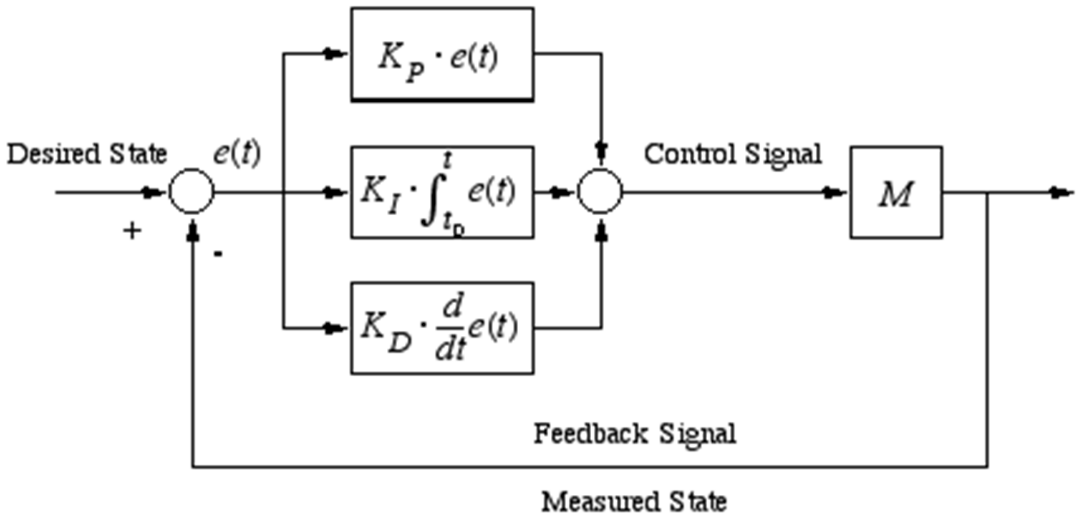


Figure 7: Visualization of a PID controller

2.4 Automated Guided Vehicles

Automated Guided Vehicles (AGV) are used widely in the industry because of their ability to transport loads and materials to places that would have otherwise needed a fork lift, conveyer or manual car transport^1. AGV's don't require the intervention of a human and can perform an assigned task with accurate precision without getting tired and weary. This repetition and performance has proven to reduce variances such as injury found commonly in human delivery operations. AGV's that work on wire and tape, like the MechBot, increase the productivity and functionality of the plant. They also increase the safety factor of the workplace by reducing the hazards of forklift trucks. Magnetic tape guided AGV's are guided by these magnets on the floor which are detected by the sensors underneath the vehicle. Wired AGV's are similar in nature, however they navigate along a continuous wire embedded into the floor of the facility [1]. These wires are detected by an antenna in the vehicle. AGV technology is now a wanted commodity in industries worldwide. Its ease of use, functionality and work efficiency make it the perfect tool to manage any physical barriers or obstacles encountered on the factory floor.

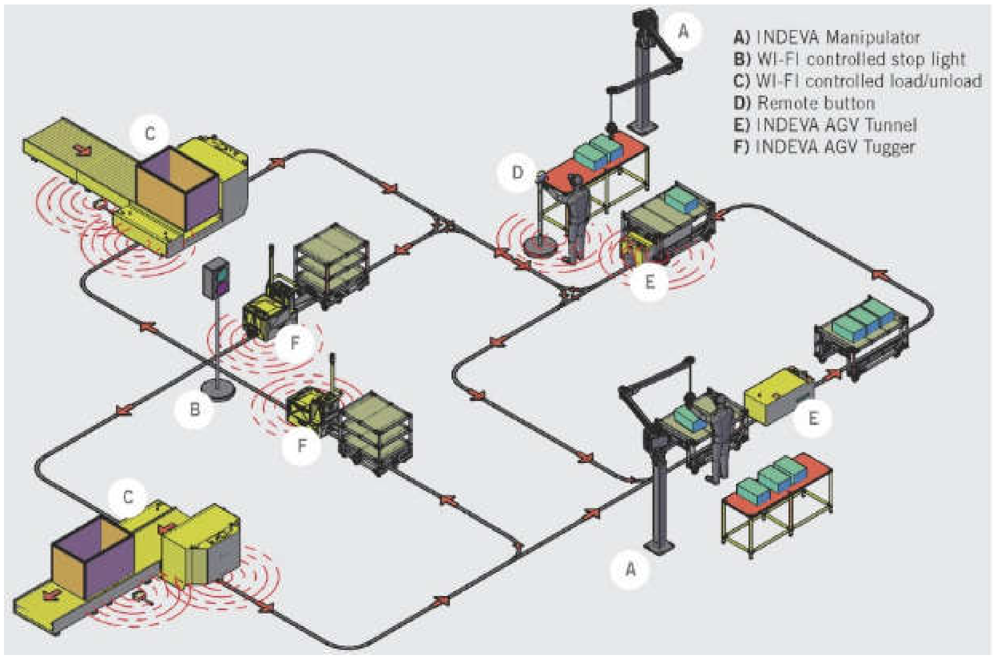


Figure 8: AGV applications visualized [2]

3.0 Problem Definition

The MechBot was required to simulate a factory AGV as it goes around an industrial plant. It will encounter various obstacles along its path and students must program the AGV to overcome these physical barriers and complete the course.

3.1 Characteristics

The AGV had to accomplish four main tasks in its program to complete the path through the industrial plant.

1. Line Sensing
2. Pick Up
3. Drop Off
4. Garbage/Safety

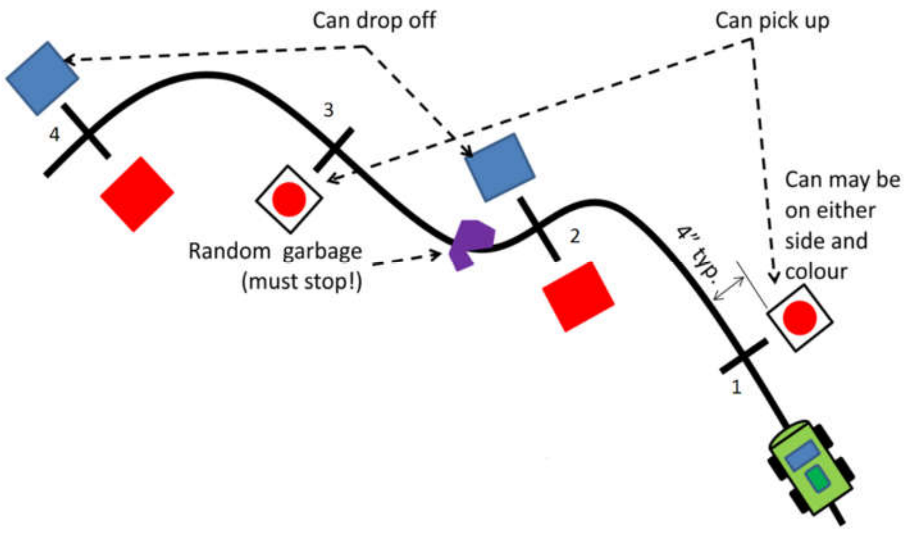

Can drop off
Can pick up
3
4
Can may be on either side and colour
2
4" typ.
Random garbage (must stop!)
1

Figure 9: Diagram of tasks that need to be accomplished by the MechBot

3.2 Requirements

- The MechBot will be completely autonomous.
- The MechBot will stop at each T junction.
- The MechBot will sense the can on either side and attempt to pick the can up.

- The MechBot will move to the next junction and drop off the can at the next T-junction.
- The MechBot will identify garbage on the track, stop to remove it and then continue its path
- Cans must remain on the paper after drop off.
- The can color must match paper color to receive full points.

3.3 Constraints

- The MechBot must not be touched.
- Cans may be either red or blue.
- Cans may be placed on either side of the black line.
- Red zone will be on the left, while the blue zone will be on the right.

3.4 Metrics

- Cans will be places on stands that are 4” × 4” × 4”
- Drop off points are 8.5” × 11” sheets of color paper.

Table 1: Problem Definition compiled into table form

Product Characteristics	Functional Requirements	Constraints	Performance Metrics
Line Sensing	Autonomous/ Must stop at T-junction	Must not be touched	$\%P_{error} = \frac{Experimental - Theoretical}{Theoretical} \times 100\%$
Camera Detection	Must sense the correct color	Red or Blue cans	
Arm Grip to pick and drop cans	Must pick can on either side/drop can off at next junction	Red zone on left/Blue zone on right	$A = L \times W$ $V = L \times W \times H$
Safe	Must identify garbage, stop and remove it	Buzz and Flash LED’s/ rated how safe out of 5	$P = \frac{5 - stars}{5}$

4.0 Possible Final Solutions

4.1 Line Following

The MechBot was placed on black tape, and it was supposed to follow the tape until the end. The bot was also supposed to stop at T junctions along the tape. As time was not a constraint in this project, more attention was given to the accuracy of the line following response of the MechBot. The can was also placed less than an inch away to the sides of the MechBot, if the MechBot were to follow the lines with a high overshoot, it would hit the can or be too far away for the arm to grab it.

A PID controller was integrated within the Arduino program to create a feedback system to smoothen the forward movement. The proportional integral derivative (PID) controller continuously calculated the error between the desired position and the current position, and applied a correction to create a more regular movement of the MechBot. PID values were adjusted so that the turns of the MecBot were smooth, when making the first initial turn to the T junction. Since accuracy was prime, and time constraints non-existent; the motor speed was adjusted to run a bit slower, so that the MecBot had more time in reading the line as it traversed it.

The line following function was calibrated by testing the line sensor values against black tape and the floor. The black tape was only wide enough to cover all four sensors, and only covered two sensors. The testing resulted in very contrasting values for the two possible scenarios, which also differed for each of the four line sensors. The values were recorded and integrated into a series of ‘if’ statements within the Arduino program that guided the MechBot to follow the path established by the black tape.

The electrical tapes sensors are wide enough only to cover the Left center sensor and Right center line reader sensor. To make the MechBot stop at T junction, all four sensors must first be confirmed to be on the black tape.

Thus, the solution to this, was to take preliminary readings and calibrate the sensors and find the minimum threshold that it must reach in order for the sensor to confirm that it has reached on the black tape. The values found were:

THRESHOLD LEFT: 770

THRESHOLD LEFT CENTER: 730

THRESHOLD RIGHT CENTER: 730

THRESHOLD RIGHT: 730

The solution for any turns within the track outlined by black tape was done by integrating PID control. An error value was assigned to every combination of sensors that read black tape. The table below shows the configuration of different cases

Table 2: Error values assigned for the PID controller for each sensor condition

Sensors greater than threshold	Error value
If only Left sensor is greater	-3
If only Left and Left Center is greater	-2
If only Left Center is greater	-1
If only Left Center and Right Center is greater	0
If only Right Center is greater	1
If only Right and Right Center is greater	2
If only Right is greater	3

4.2 Pick-up and drop-off of cans

The pickup and drop off of the cans was a system that integrated the MechBot and the EVShield. A two-way binary communication system was used to transfer information between the two programs. The Arduino program for the MechBot was designed such that when an object is detected at a 5 centimeters or greater distance, the automated guided vehicle would halt and send information to the EVShield regarding the pick-up of the can. Next, the camera on the automated guided vehicle proceeds to record the color of the can and the Lego assembly picks up the can. Once this is completed the automated guided vehicle returns to the line following function.

The cans had to picked up from the left or right side at the odd intersections, and had to be dropped off at the even intersections. The cans were placed 4 inches from the center of the black line, and their position was random. Thus, the sensor array available had to be used to determine if the cans were on the left or right side.

To determine the position of the cans, the left and right distance sensors on the MechBot were used. Same as the line sensors, these sensors also had to be calibrated. An analog value from the sensors had to be determined which determined the direction of the cans. These values were:

CAN_THRESHOLD_LEFT:140
CAN_THRESHOLD_RIGHT:140

When at the intersection, if one of the sideways distance sensors read a reading greater than or equal to the thresholds, the can was assumed to be in that direction and the specific subroutines for the arms were executed.

The grabbing subroutine consisted of controlling two motors. The first motor controlled the direction of the arm. This motor was only activated if the arm was currently facing the opposite direction. The direction was determined in software. After the arm was turned to the given direction, the second motor was activated in the forward direction to move the grip hand into the can. After the grip hand made contact with the can, the motors were not disabled, but were run at a lower speed, this provided some torque through the motor which kept the grip on the can firm. The speed was very low to prevent damage to the motors due to overheating.

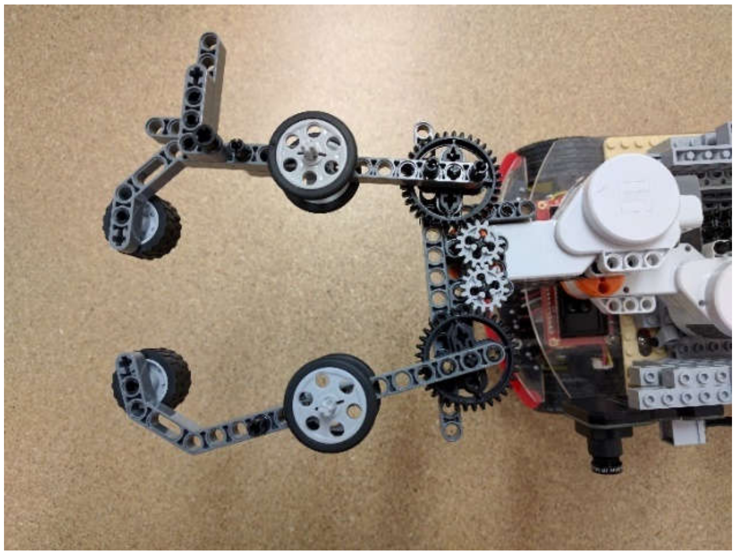


Figure 10: Grip hand on the MechBot

The release subroutine was very simple, when the EVShield received the command from the MechBot to drop the can, the grip hand stopped their motors to release the constant force on the can and then reversed to their initial positions, dropping the cans.

4.3 Garbage Safety

The front distance sensor was calibrated similarly to the side distance sensors. A sensor value of 450 was selected to stop the MechBot, this value was selected using trial and error, as the MechBot’s speed had to be taken into consideration. Therefore, the Arduino program for the MechBot included a function to halt if an obstacle was present at said arbitrary distance in front of the automated guided vehicle. The front bumpers could also have been used to achieve the same task, but as the MechBot already had functions to get data from the distance sensors, it was more time efficient to call those functions than to write new logic to handle the bumpers.

5.0 Black box model

The black box diagram for the automated guided vehicle is shown in figure 12 below. The black box diagram is a systems diagram for the automated guided vehicle and includes the functions of the PID controller, which is used for the movement of the MechBot, and the color recognition and arm motion functions, which are used for the pickup and drop-off of the cans.

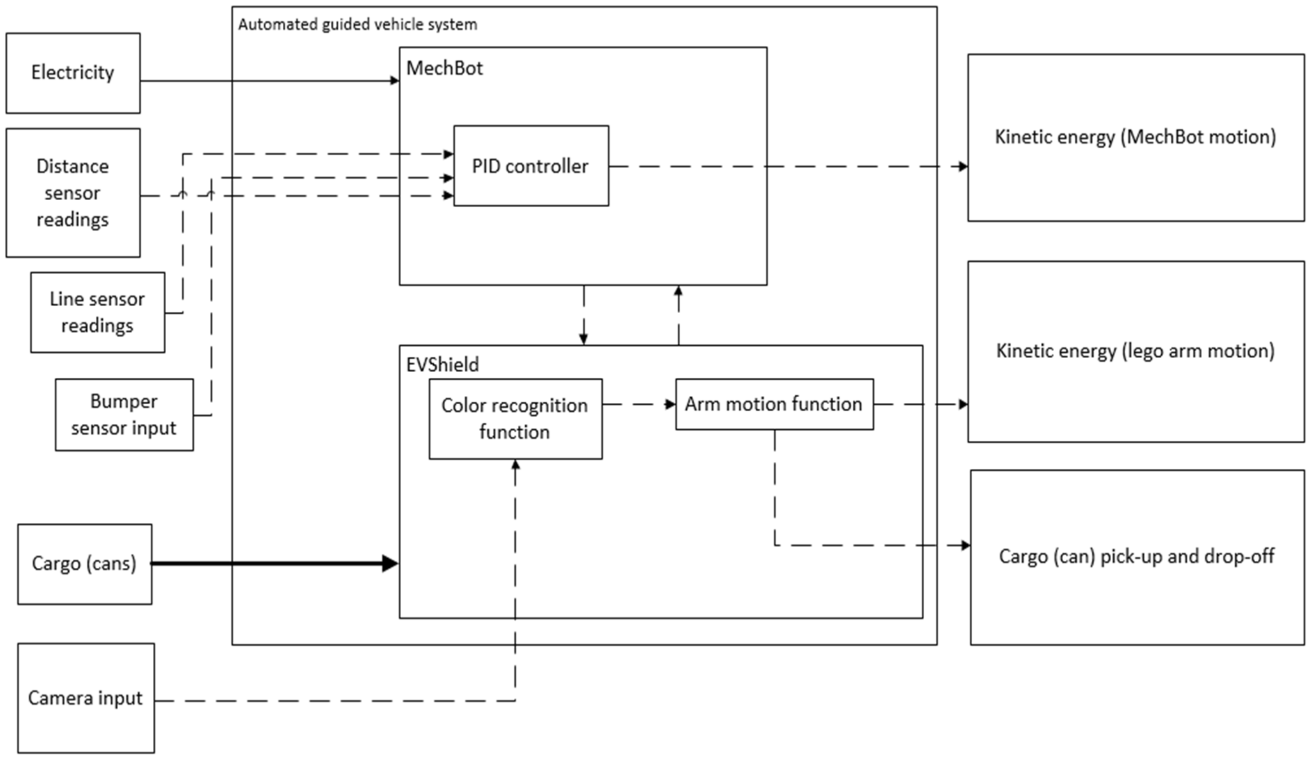

Automated guided vehicle system
Electricity
Distance sensor readings
Line sensor readings
Bumper sensor input
Cargo (cans)
Camera input
MechBot
PID controller
EVShield
Color recognition function
Arm motion function
Kinetic energy (MechBot motion)
Kinetic energy (lego arm motion)
Cargo (can) pick-up and drop-off

Figure 11: Backbox System of MechBot functions

6.0 Simulink & SimMechanics Model

A Simulink model was developed using the Simscape Multibody environment. The simulation was designed for the mechanism that was created using components from the Lego Mindstorms kit. The simulation of the mechanical system was based off the mechanical measurements of distance, therefore the lengths of the links and calculations of the centers of gravity were utilized to create a simulation that can accurately represent the design.

The simulation of the mechanism for pickup and drop-off begins with the machine environment block, which represents the mechanical system of the environment. Which allows for settings such as gravity to be modified. The machine environment block is connected to the ground, which represents the MechBot. The MechBot is fixed to the first set of Lego bricks and the first motor. This connection is represented by a weld because there is no relative movement. Body blocks are used to represent each set of Lego bricks and motors that had no motion relative to each other. Revolute joints are used to represent the output shafts of the motors. Joints are components that connect bodies together, and revolute joints are pin joints that allow one degree of freedom. For the automated guided vehicle, revolute joints provided single axis rotation around the y-axis.

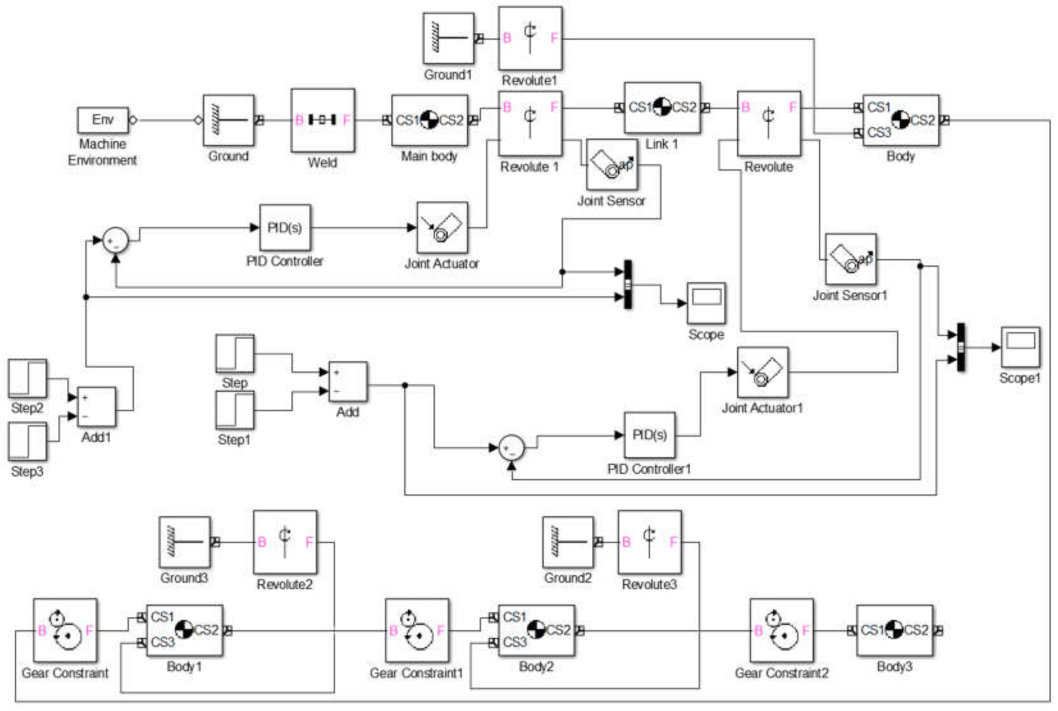


Figure 12: Block Diagram used to simulate mechanical system

Proportional integral derivative (PID) controllers are also included to power the actuators at the revolute joints. The block diagram contains actuator blocks that represent electrical motors that actuate the revolute joints. In order words, the actuator blocks cause the revolute joints to rotate about the y-axis. Physically, the actuators are the motors from the Lego Mindstorms kit and the power is internally generated. In figure 12 above, the actuators are powered by initial values of torque in newton meters, which are provided using a constant value block, and the PID controllers are connected to sum the difference and power the respective revolute joints over time. Sensor blocks are used to retrieve the previous values of measures regarding the rotation of the revolute joints. Sensors are devices that measure, or detect the change in, physical values. Scope blocks are used to output the values of each sensor block into charts. The control system for the actuator-sensor combination is a closed loop system that is initiated by constant torque values.

Figure 13 below is an image of the simulation window for the block diagram. The centers of gravity are represented by small black circles, whereas the bodies are represented by red links. The graphical representation is for the mechanism built using Lego Mindstorms and therefore the body at the top represents the link that is welded to the MechBot. The two longer bodies represent the two motors as well as the Lego bricks that connect them together. The U-shaped component represents the grabbing mechanism.

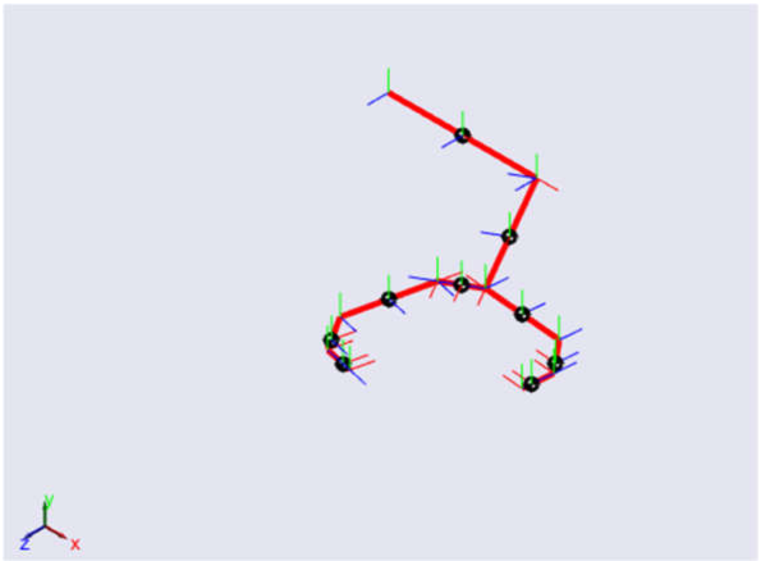


Figure 13: Graphical simulation in Simscape Multibody

The center of gravity was calculated for the first body, which was a combination of a rectangular block of Lego bricks with a motor protruding from the center. The body was modeled as a T-figure and the calculation is given below.

$$z_{CG} = \frac{(80.17)(7) + (5)(6.25)}{85.17} = 6.97\ cm$$

$$x_{CG} = \frac{(80.17)(-1.5) + (5)(1.55)}{85.17} = -1.32\ cm$$

$$y_{CG} = \frac{3.5}{2} = 1.75\ cm$$

The second body which represents the second motor with connections to Lego bricks was modeled as a rectangular prism, and the calculations are given below.

$$z_{CG} = \frac{8.25}{2} = 4.125\ cm$$

$$x_{CG} = \frac{-1.1}{2} = -0.55\ cm$$

$$y_{CG} = \frac{0.25}{2} = 0.125\ cm$$

Figure 14 below is an image of a chart representing the position of the first revolute joint. The first revolute joint is shown as a block labeled “Revolute 1” in Figure 12. Said revolute represents the first joint that causes movement of the grabbing mechanism relative to the MechBot.

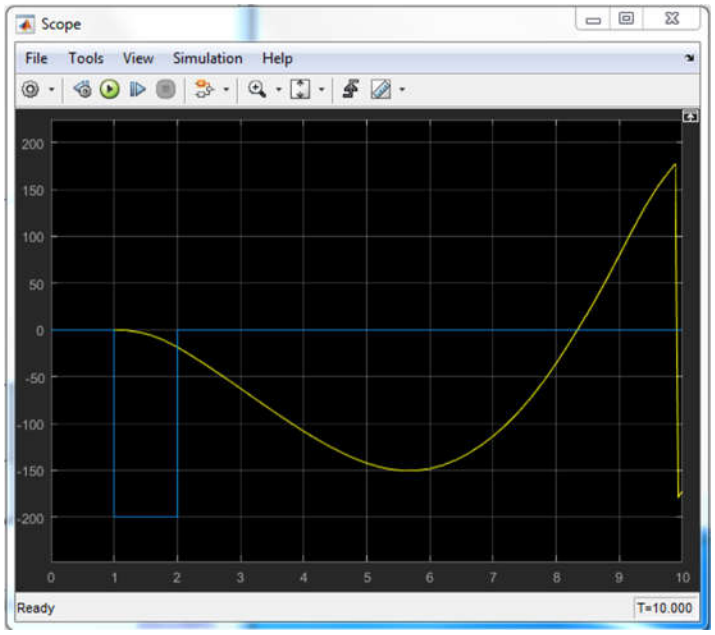


Figure 14: Actuation position for first revolute joint

Figure 15 below is an image of a chart representing the position of the second revolute joint. The second revolute joint is shown as a block labeled "Revolute" in Figure 12. Said revolute represents the second actuated joint that causes movement of the grabbing mechanism which, in turn, causes the gears to move two rigid bodies that surround the cans to complete the pickup and drop-off.

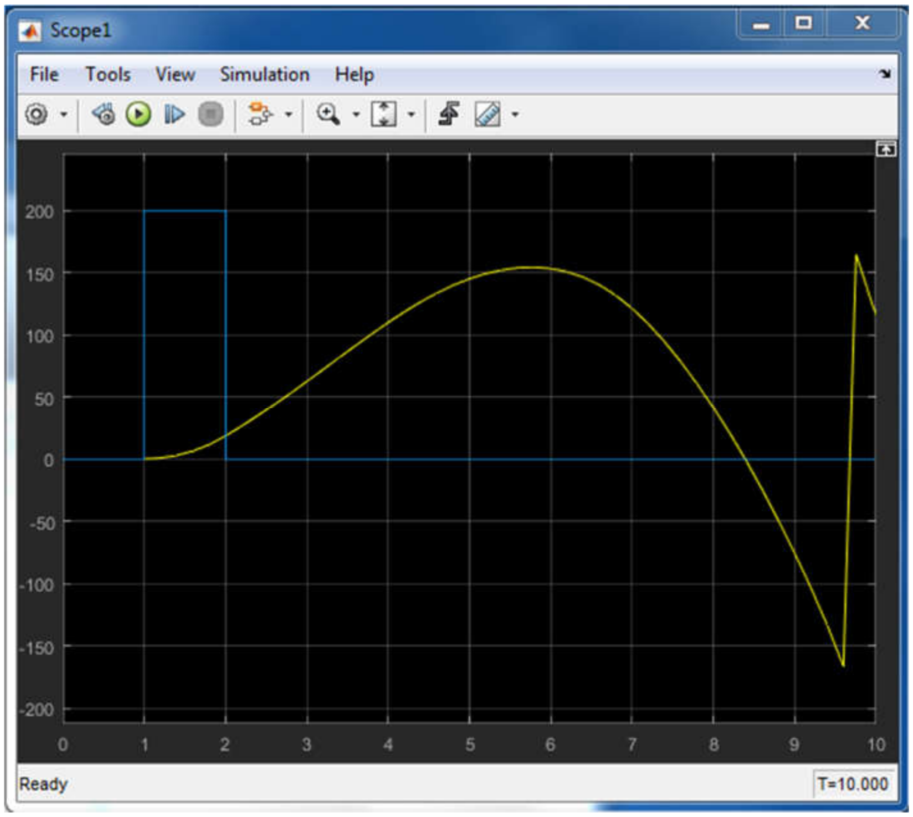


Figure 15: Actuation position for second revolute joint

A Simulink model was created for the motors, and given below in Figure 16. The model includes a step input which is applied to the electrical component of the motor, and a torque constant which is then summed and applied to the mechanical component of the motor. This simulates movement of the motor, and a feedback is used to close the loop.

A Ke value of 0.068 was used, which was applied to the 'Back EMF' block, and a Kt value of 0.275 was used, which was applied to the 'Kt' block. A constant value of 16.7 Newton-meters was used for the torque applied.

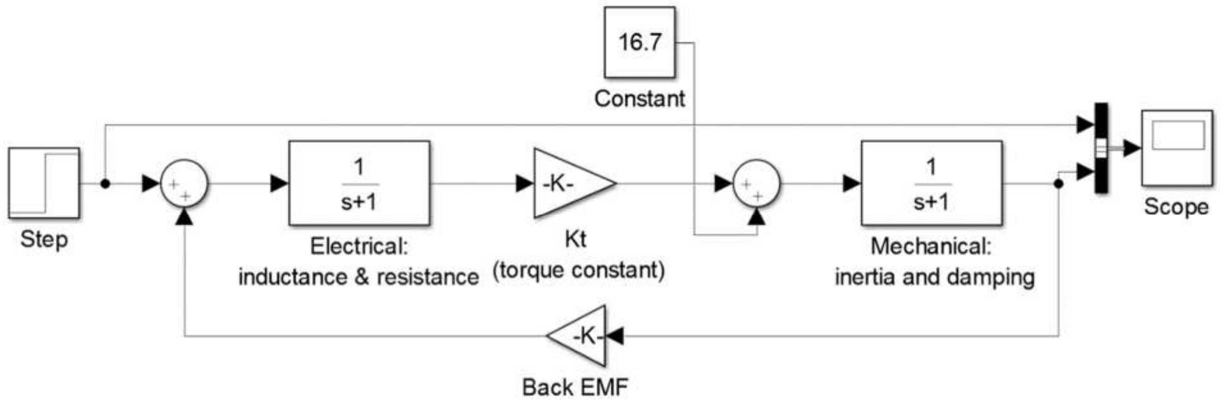


Figure 16: Simulink block diagram

Figure 17 below is an image of a chart that describes the change in torque applied, which includes both the step response and the output of the transfer function. The yellow line indicates the step input, and the blue line indicates the actual motor response with considerations of electrical and mechanical conditions. Specifically, inductance and resistance, as well as inertia and damping.

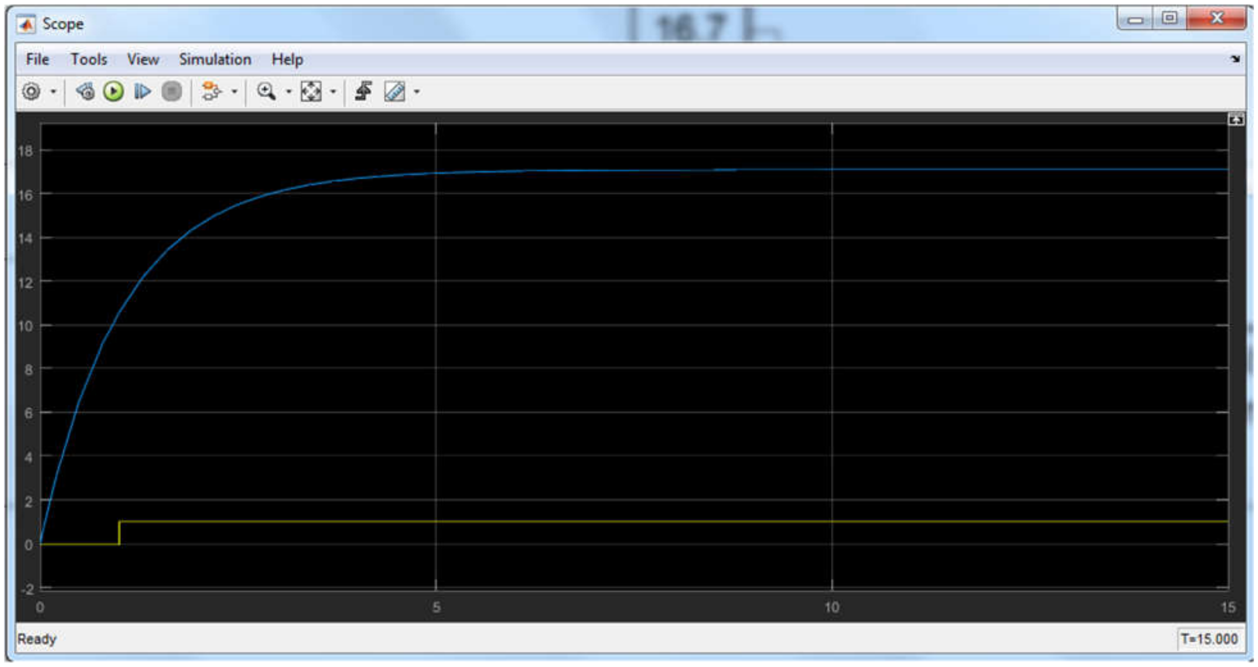


Figure 17: Simulink motor model chart

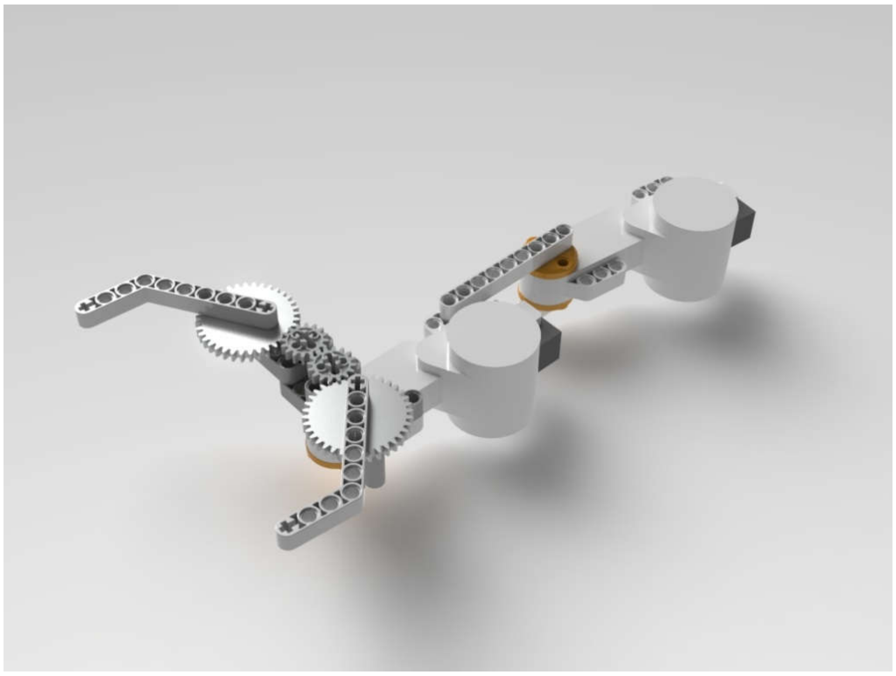


Figure 18: SolidWorks rendering of the Mechanical arm

The initial plan to start the SimMechanics phase of the project, was to construct a 3D model of the mechanical arm and merge this model into SimMechanics. By merging the CAD file into SimMechanics, a Simulink block diagram would be created illustrating the joint and components of the mechanical arm. This method was initially proposed in order to save time on the SimMechanics phase of the project. Since the SolidWorks file would generate the Simulink model, the group could then focus their attention to modeling the motion of the arm. However, problems were encountered when attempting to merge the two files together. Therefore the group decided to construct the Simulink block diagram from scratch by observing the actual model. Using this block diagram the motion of the arm was modeled and calculated.

7.0 Discussions

7.1 Mechanical

The mechanical component of the automated guided vehicle was the MechBot 2.5, which was coupled with the Lego Mindstorms EV3 kit in order to create a mechanical arm mechanism. The group made use of the Lego Mindstorms kit to first build a strong base, which essentially was the foundation for the mechanical design. This base was crucial in creating a platform for the camera, EV-Shield and mechanical arm.

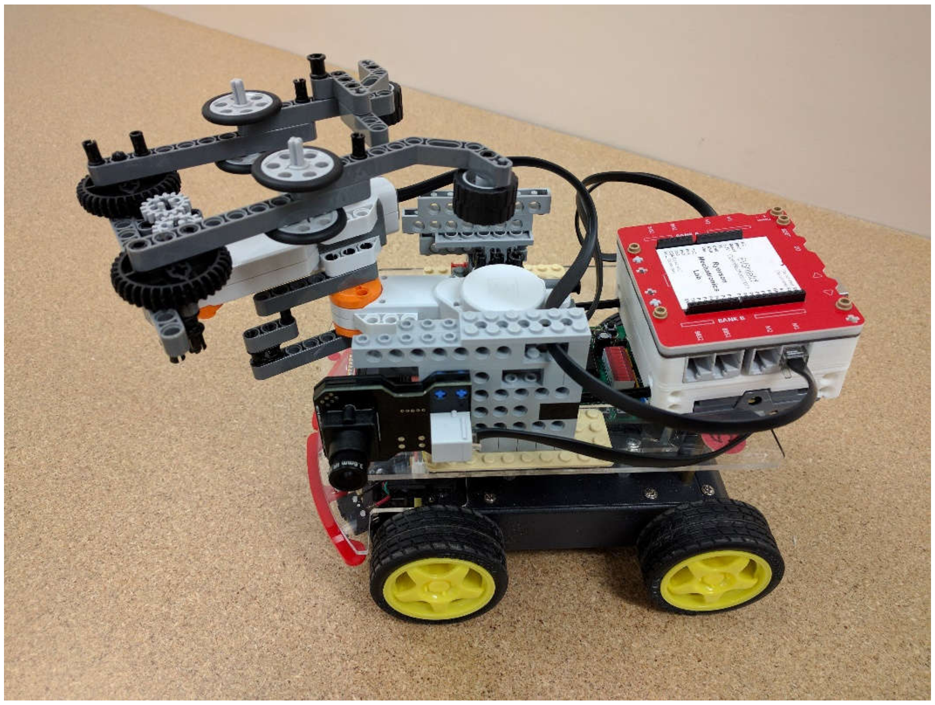


Figure 19: Assembled arm design with EVShield mounting on MechBot

The mechanical arm consisted of an interactive servo motor that was hoisted onto the Lego base. Two servo motors were connected together with Lego rods, in order to increase the length of the arm and ensure that the arm extended out far enough. At the motor hub, (which had a 360-degree rotation sensor) a system of Lego gears and Lego beams were used to construct the gripping hand of the arm. Rubber tires were used in this design to ensure that the arm had a tight grip, so that it would be able to efficiently pick up the pop cans. After testing different materials, the group decided that rubber would be the best material to provide the strongest grip.

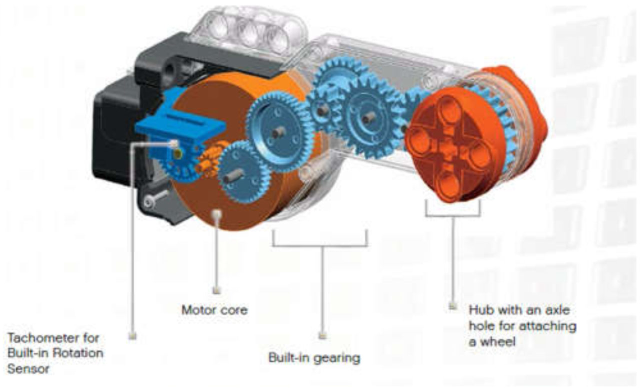

Motor core
Hub with an axle hole for attaching a wheel
Tachometer for Built-in Rotation Sensor
Built-in gearing

Figure 20: Internal schematic of the NXT Motor

The arm and the gripper also had end stops to their left/right and retracted positions respectively. This setup enabled the use of motor unlimited functions instead of degree functions, which significantly increased the repeatability of the arm and gripper movements.

The EV-Shield was hoisted at the back of the MechBot so that it would not obstruct the motion of the arm. It was also placed in the back so that the wires that connected the camera and motor would not get tangled up. The EV-Shield was connected to the base using two long Lego rods. As shown in the Figure # below, it was positioned appropriately so as to not cover the seven LED's on the MechBot. These LED's were initialized in the program and therefore had to be visually presented to the user during the testing phase.

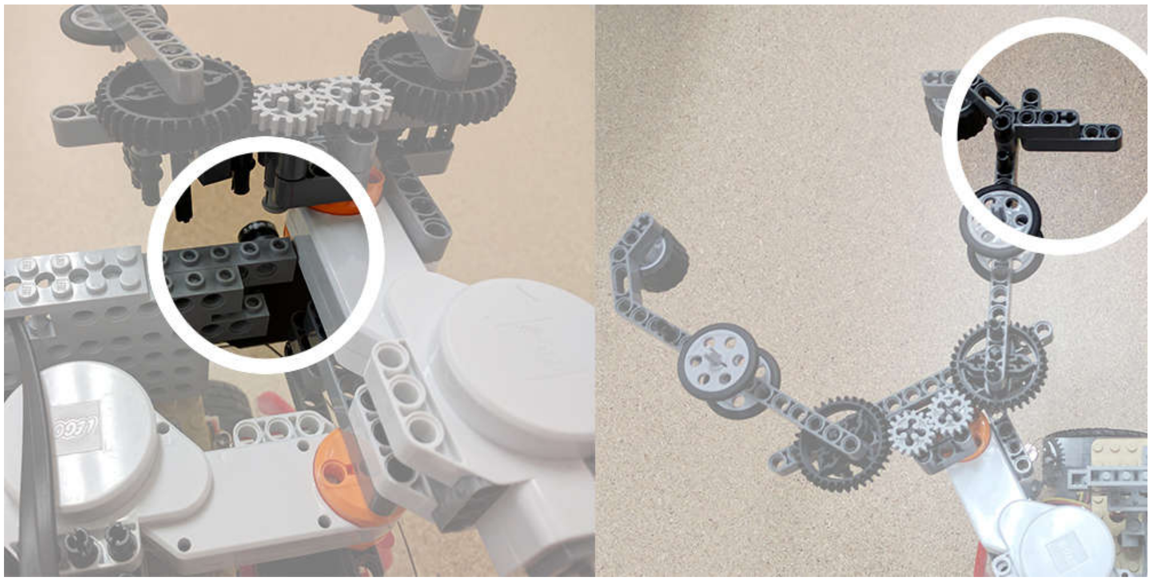


Figure 21: Physical end stops implemented in arm and grippers

7.2 Electrical

The automated guided vehicle system was composed of many electrical components, with the two major components being the MechBot and the EVShield. The MechBot contained a 9V lithium-ion rechargeable battery, which powered the Arduino microcontrollers as well as the peripheral circuit boards, sensors, and switches. The Arduino was rated at 5V; therefore, a voltage regulator was used to drop the 9 volts down to 5 volts. MOSFETs were integrated with the Arduino to switch power to the motor from the battery. Motor direction control was achieved by using an H bridge circuit and changing voltage polarity across the motor. Moreover, the motor speed was controlled using pulse width modulation, which regulated the average voltage over the duty cycle. The EVShield is contained the program to control its peripheral components, which were two NXT interactive servo motors. The EVShield was powered by a 9V battery, which was used to power the two 9V servo motors.

For the communication line between the MechBot and EVShield, the PC4 and PC5 pins on the MechBot were used to transmit the binary data and pins PB0 and PB1 were used to receive data on the EVShield. No special communications protocols like I2C or Serial were used.

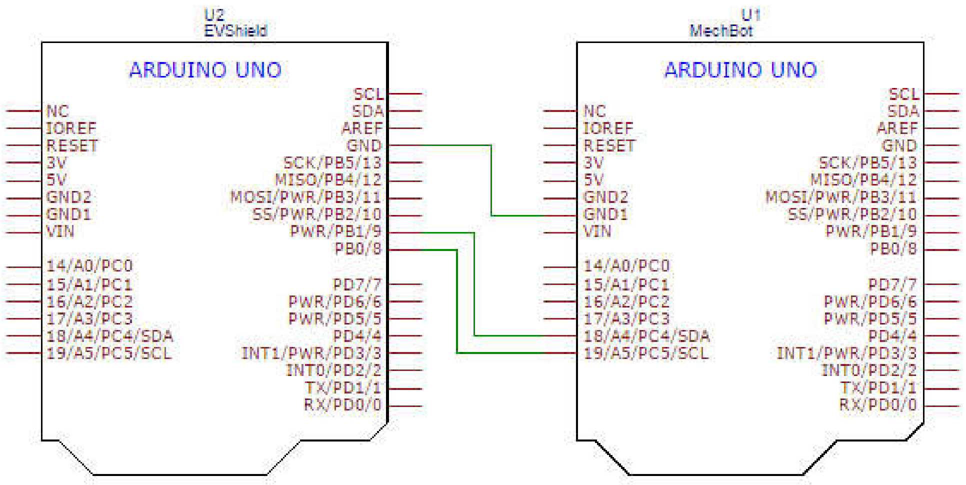
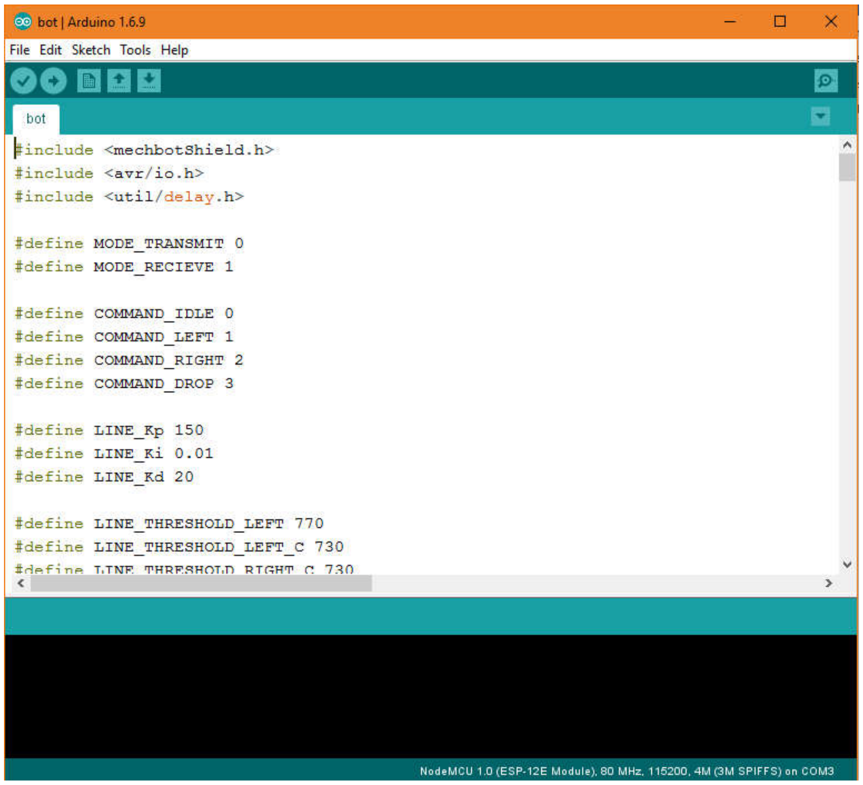


Figure 22: Communications line wiring diagram

7.3 Software

The automated guided vehicle consisted of the MechBot and the EVShield. Both contained separate programs that exchanged information using serial communication. The programs were both Arduino programs written in C code. The Arduino IDE was used to program both the Arduino Uno in the MechBot and EVShield.



```
#include <mechbotShield.h>
#include <avr/io.h>
#include <util/delay.h>

#define MODE_TRANSMIT 0
#define MODE_RECIEVE 1

#define COMMAND_IDLE 0
#define COMMAND_LEFT 1
#define COMMAND_RIGHT 2
#define COMMAND_DROP 3

#define LINE_Kp 150
#define LINE_Ki 0.01
#define LINE_Kd 20

#define LINE_THRESHOLD_LEFT 770
#define LINE_THRESHOLD_LEFT_C 730
#define LINE_THRESHOLD_RIGHT_C 730
```

NodeMCU 1.0 (ESP-12E Module), 80 MHz, 115200, 4M (3M SPIFFS) on COM3

Figure 23: Arduino IDE editor

All the actions performed by the MechBot were due to the harmony between the software and the hardware. The software required us to combine a lot of knowledge previously learnt. For instance, the Line Follow algorithm used a full PID control to determine its error in real time and make adjustments accordingly. All of this had to be done in real time and the MechBot wasn't supposed to stop and adjust for error. This was achieved by variating the motor speeds on each side of the bot to fix the errors. This required knowledge of closed loop control systems. The codes for both the MechBot and EVShield are visualized in the flowcharts below.

7.3.1 MechBot

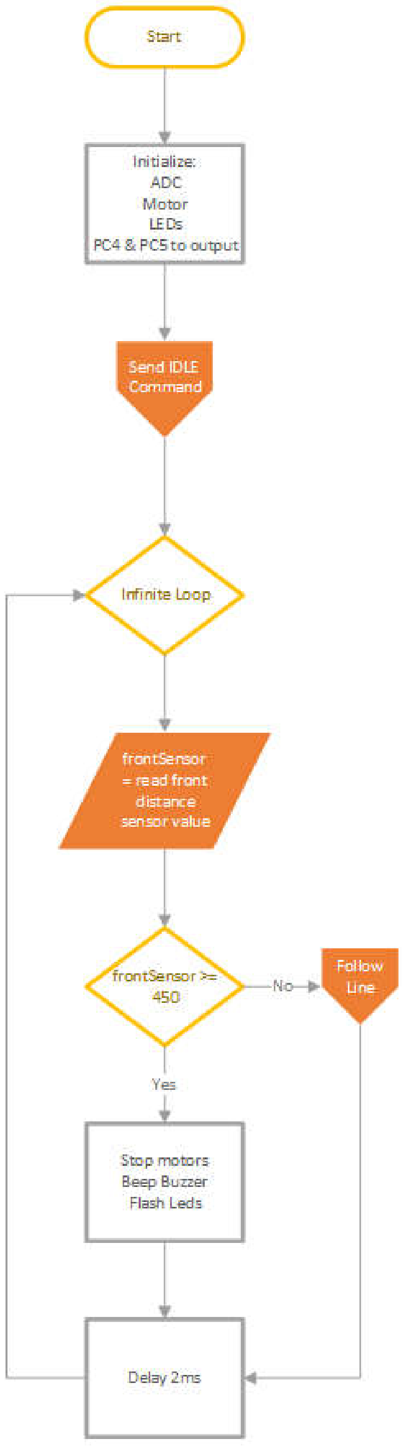


Figure 24: Main subroutine

The program that was on the MechBot was kept as simple as possible. All the complex tasks were split up into smaller functions which could be modular, this facilitates easy debugging of the code and also replacing logic is much easier with this implementation, as changing one of the functions doesn’t affect the entire system.

The main function consisted mostly of logic for initializing various inputs and outputs on the MechBot and initializing the infinite loop which checked for obstacles ahead constantly while also following the line.

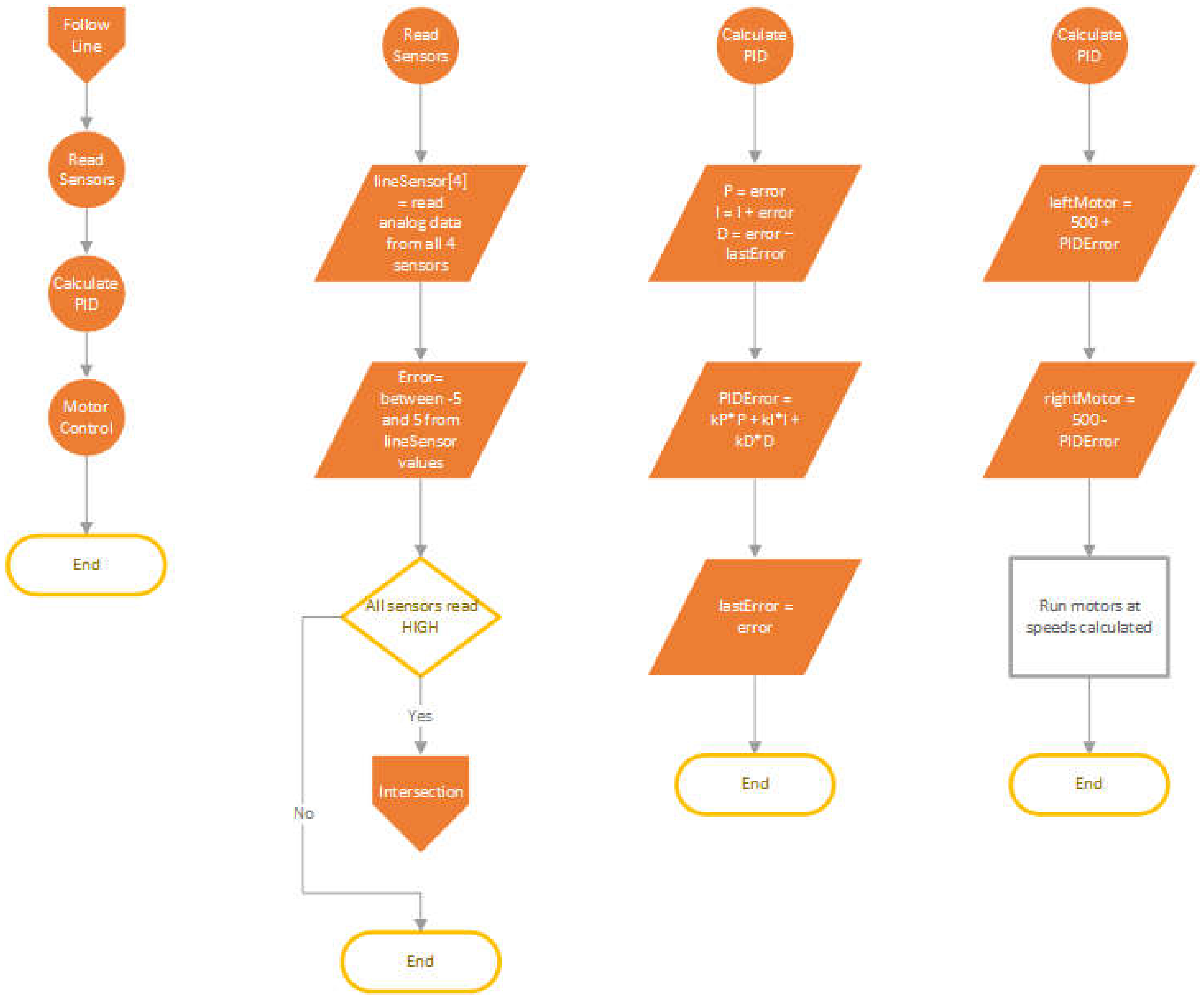

Follow Line
Read Sensors
Calculate PID
Motor Control
End
Read Sensors
lineSensor[4] = read analog data from all 4 sensors
Error= between -5 and 5 from lineSensor values
All sensors read HIGH
Yes
No
Intersection
End
Calculate PID
P = error
I = I + error
D = error – lastError
PIDError = kP*P + kI*I + kD*D
lastError = error
End
Calculate PID
leftMotor = 500 + PIDError
rightMotor = 500 - PIDError
Run motors at speeds calculated
End

Figure 25: Follow line subroutine

The line following subroutine was further divided into three functions, each doing separate things. The first function was supposed to read sensor values from the line follow sensors mounted at the bottom of the MechBot and generate an error value depending on the reading, the error values generated can be seen in Table 2 above. The error was then passed on to the PID controller where it was transformed into a PID error to control the motors. And the PID error was then added and subtracted to the motor speeds to control the speed at which each wheel should be rotating at.

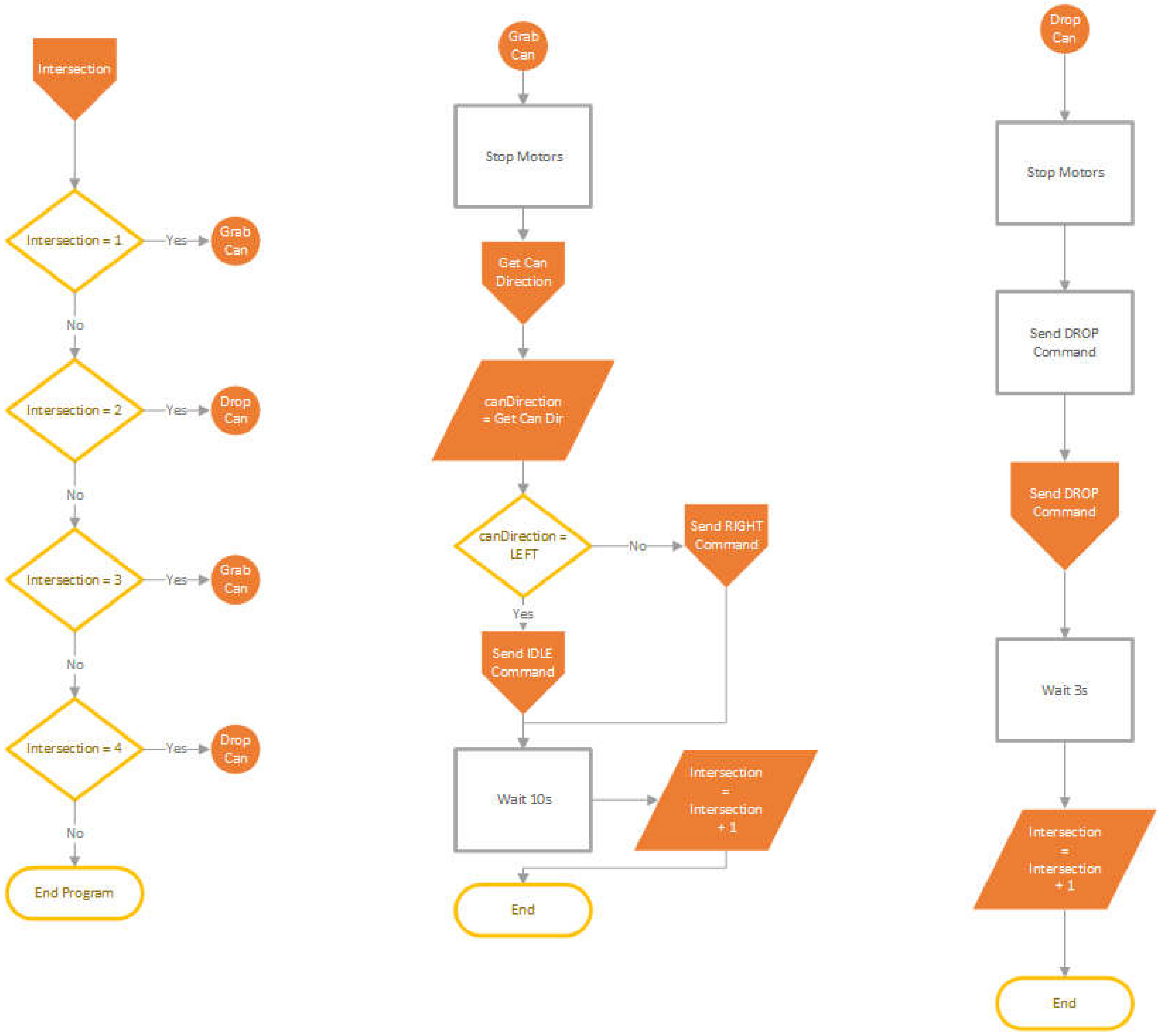


Figure 26: Intersection subroutine

When the line sensors detected a high value on all 4 sensors, the MechBot determined that it was at a T Junctions and it stopped its motors first. The MechBot kept a running count of the number of intersections that it encountered to determine the action that it was supposed to take. If the value for intersection count was odd, the MechBot was supposed to pick up a can, and for even it was supposed to drop it. It used the left and right distance sensors to get the direction of the can and sent that information to the EVShield. The communications used a binary protocol where the two pins acted like binary counters. The MechBot then waited for the EVShield to send a response back with a confirmation. The confirmation started the motors again and made the bot start to move to the next intersection. The conditions used in the communications protocol can be seen in the table below.

Table 3: Binary commands available for the MechBot to transmit

PC4	PC5	Command
0	0	IDLE
0	1	LEFT
1	0	RIGHT
1	1	DROP

The steps taken for sensing the direction of the can and sending the signal to the EVShield can be seen below in the flowcharts.

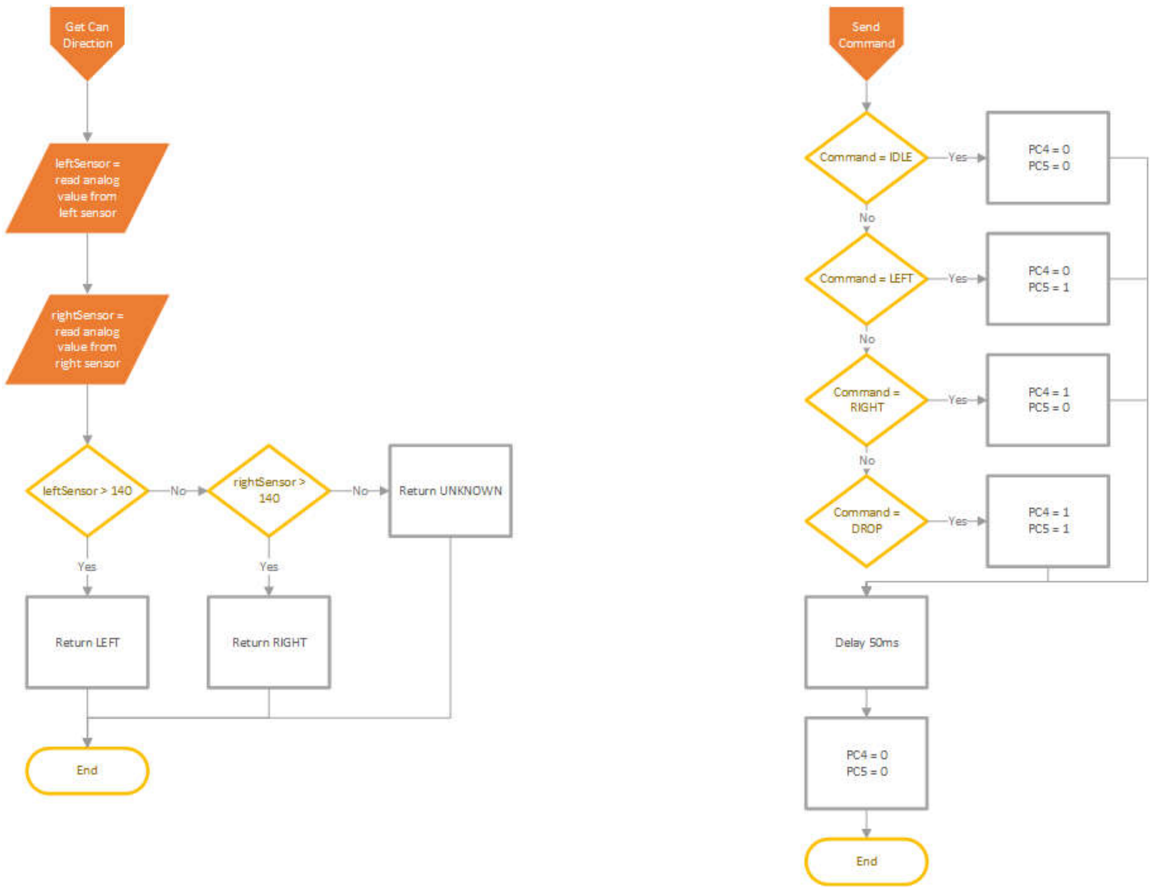


Figure 27: Can sensing and Signal Transmission logic

7.3.2 EVShield

The program on the EVShield was quite simple as it was only meant to react to commands given by the MechBot. After initializing the motors and camera, the EVShield kept polling pins PB0 and PB1 for changes. If a command was detected from the MechBot, it initialized the corresponding subroutine.

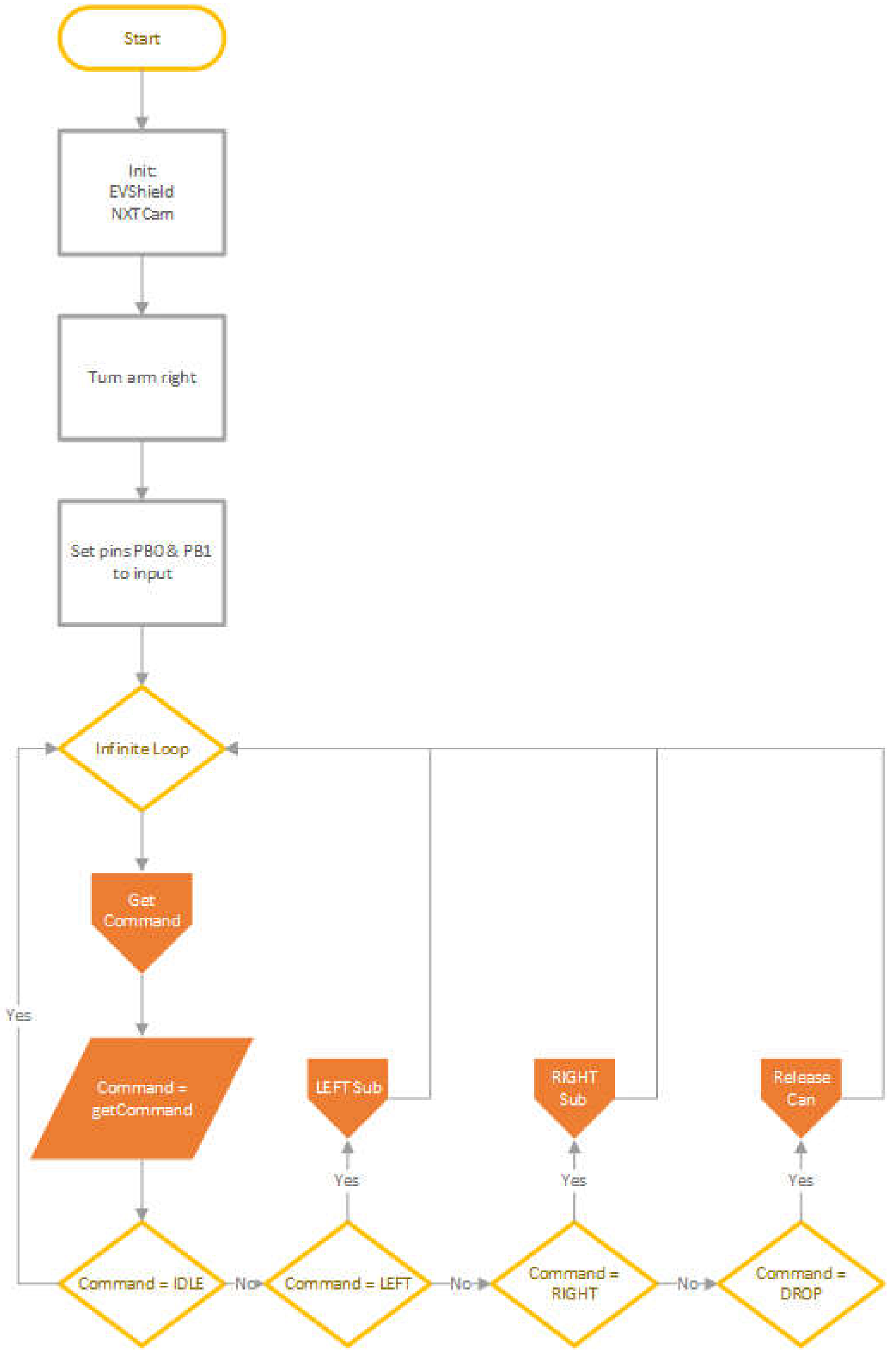


Figure 28: EVShield main subroutine

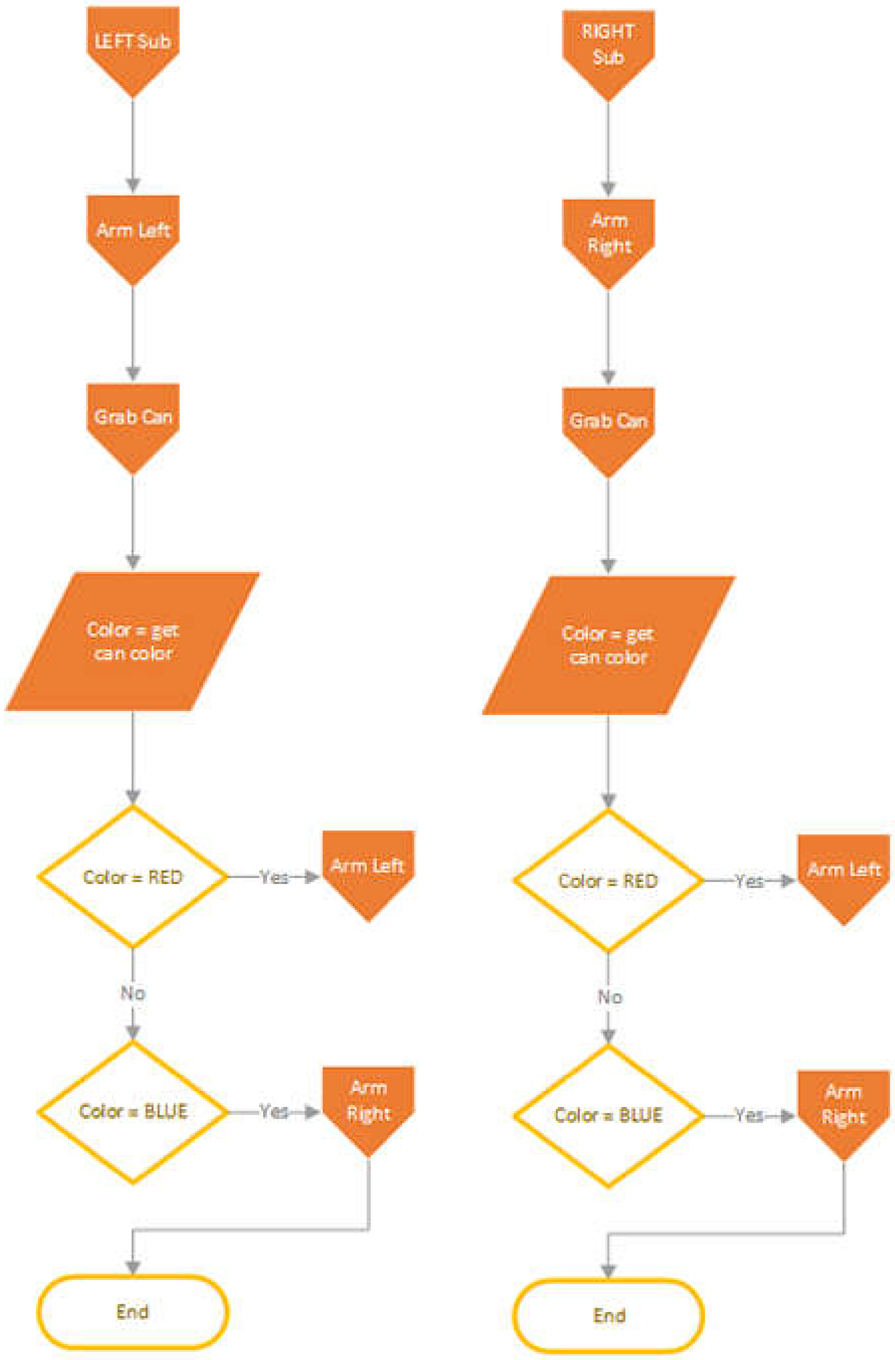


Figure 29: Subroutines for LEFT and RIGHT commands from the MechBot

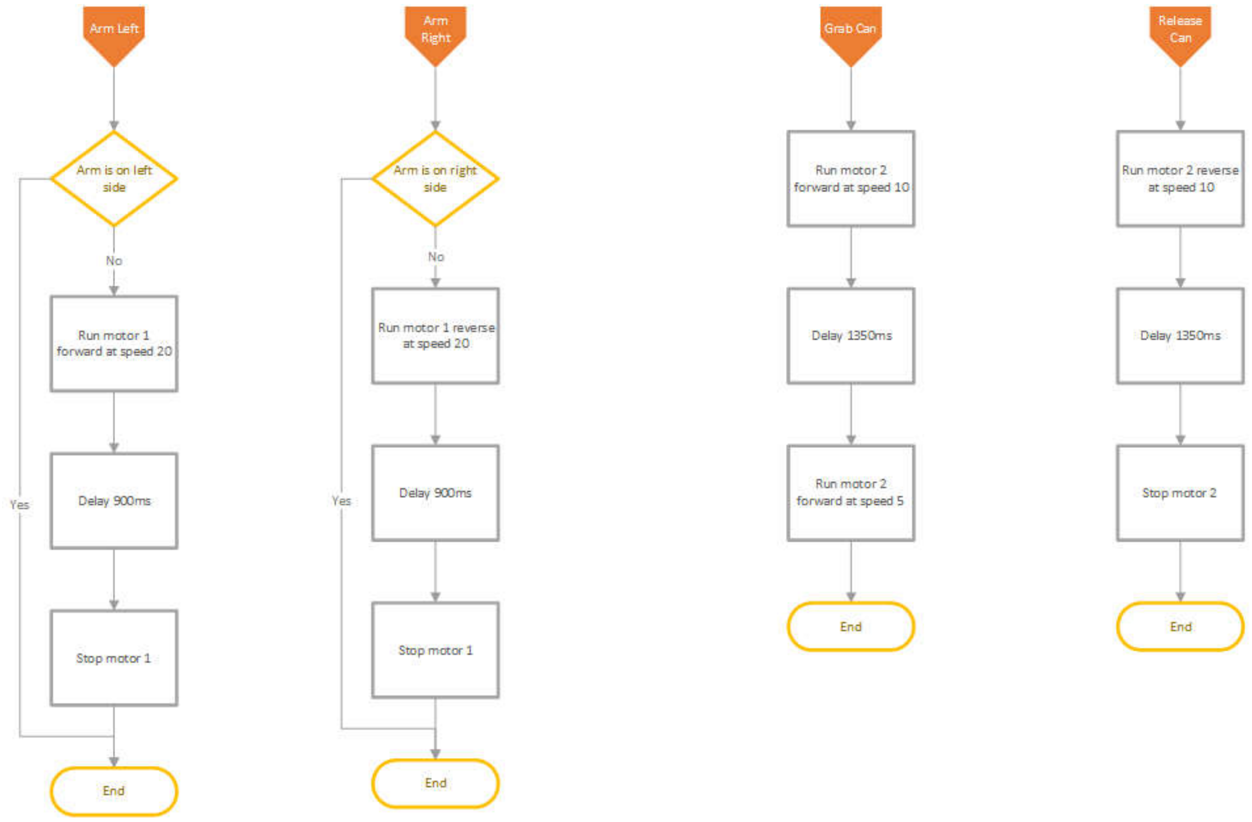


Figure 30: Subroutines for controlling motors to rotate arm and open and close grip hand

The motor control worked using delays instead of using the degree functions provided by the EVShield library as physical end stops were implemented in the design, and the degree function was quite unreliable in repeatability tests performed during the testing phase. This system achieved a high reliability thus was implemented into the system.

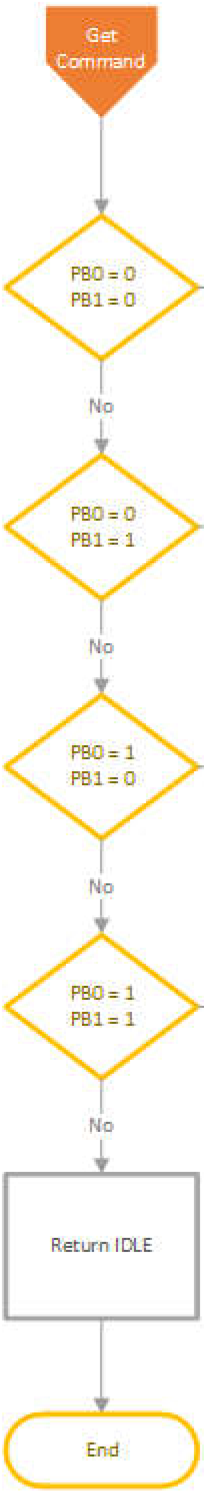


Figure 31: Subroutine for identifying the command from MechBot

8.0 Discussion of Team roles

Having defined the problem and discussed possible solutions that could be implemented for a successful program, the next step was to divide up the work and assign roles to each team member. Several resources were used by each of the team members to accomplish their respective tasks. The lecture slides were the most useful resource given as it contained vital information regarding programming the line and distance sensors which was the bulk of the code. As for mentorship, the team reached out to the Teaching Assistant (TA) as well as Dr. Farokh Sharifi for assistance during different stages of the project.

Table 4: Task Breakdown of Programming the MechBot

Task	**Members who assisted in completion of task**
Line Following	Tanmay Parikh Abhishek Patel
Picking up and detecting cans	Tanmay Parikh Abhishek Patel Gayan Ferdinands
Garbage Removal	Tanmay Parikh

Table 5: Task Breakdown of Writing the Report

Tasks	**Member assigned**
Abstract	Gayan Ferdinands
Introduction	Gayan Ferdinands
Literature review	Gayan Ferdinands
Problem definition	Gayan Ferdinands
Final solutions	Abhishek Patel
Black box model	Abhishek Patel
Sim Mechanics Model	Abhishek Patel
Discussion of mechanical, electrical and software implementation	Abhishek Patel Tanmay Parikh Gayan Ferdinands
Discussion of team roles	Gayan Ferdinands
Conclusion and Recommendations	Gayan Ferdinands
Appendix	Tanmay Parikh

9.0 Conclusions and Recommendations

The outcome of whether the MechBot could fulfill its objective of completing the obstacle course was still an unknown quantity, as the real test had not commenced yet. The MechBot had been put under many trials and had been performing well. After having calibrated the line sensors to the appropriate values, the robot could properly center itself on the black line. The line sensing algorithm was working smoothly and the MechBot would follow the path set by electrical tape and managed to stop at the T-junction. The EV-Shield was programmed to control the camera and the arm. Using the left and right buttons on the EV-Shield the motors on the arms were initialized and the arm was given and range of motion. Initially there was a problem with the motion of the arm. The arm would overshoot its angle of rotation when opening and closing. The group identified this issue as a calibration error. Through trial and error the correct value of rotation were obtained to correct the motion of the arm. The calibration of the motor rotation allowed the MechBot to grip the pop cans and release them when needed. The design of the arm was also altered to ensure that the arm had a smooth motion.

The camera failed to work properly since the appropriate calibration was not performed. The camera was hoisted onto the MechBot and incorporated into the overall design. However, its main function of detecting the color of the cans was not initialized as part of the control system. Another issue regarding the sensors was that initially, during the testing phase the MechBots were switched with different teams due to a shortage of equipment. Thus, the sensors had to be re-calibrated. It is recommended that teams conceptualize a solid mechanical design to sit atop the MechBot. This design would include the positioning of the EV-Shield, camera and the design of the arm. If this step is done well in advance, groups can move on to the second stage of the project, programming the tasks for the MechBot to complete.

Judging from all the trails that were performed and all the corrections made, the MechBot should complete the obstacle course with minimal errors. The project requirements were all met and completed in good time. Project conflict was not encountered and group members worked cohesively and together to accomplish and meet the project milestones.

References

[1] “Automatic Guided Vehicles,” *Indeva*. [Online]. Available: http://www.indevagroup.com/automatic-guided-vehicles/. [Accessed: 30-Nov-2016].

[2] “Automated Guided Vehicles,” *Bastian Solutions*. [Online]. Available: https://www.bastiansolutions.com/solutions/technology/automated-guided-vehicles. [Accessed: 30-Nov-2016].

[3] “EVShield for Arduino Duemilanove or Uno,” *Mindsensors*. [Online]. Available: http://www.mindsensors.com/arduino/16-evshield-for-arduino-duemilanove-or-uno. [Accessed: 30-Nov-2016].

[4] “SparkFun RedBot Sensor - Line Follower,” *Sparkfun*. [Online]. Available: https://www.sparkfun.com/products/11769. [Accessed: 30-Nov-2016].

[5] “Sharp GP2Y0A21YK0F IR Distance Sensor 10-80 cm,” *Parallax*. [Online]. Available: https://www.parallax.com/product/28995. [Accessed: 30-Nov-2016].

[6] “NXTCAM V4,” *Robot Advance*. [Online]. Available: http://www.robot-advance.com/art-nxtcam-v4-584.htm. [Accessed: 30-Nov-2016].

[7] “Interactive Servo Motor,” *Lego*. [Online]. Available: https://shop.lego.com/en-US/Interactive-Servo-Motor-9842. [Accessed: 30-Nov-2016].

Appendix

MechBot

```
#include <mechbotShield.h>

#include <avr/io.h>

#include <util/delay.h>


#define MODE_TRANSMIT 0

#define MODE_RECIEVE 1


#define COMMAND_IDLE 0

#define COMMAND_LEFT 1

#define COMMAND_RIGHT 2

#define COMMAND_DROP 3


#define LINE_Kp 150

#define LINE_Ki 0.01

#define LINE_Kd 20


#define LINE_THRESHOLD_LEFT 770

#define LINE_THRESHOLD_LEFT_C 730

#define LINE_THRESHOLD_RIGHT_C 730

#define LINE_THRESHOLD_RIGHT 770

#define LINE_MOTOR_SPEED 500


#define CAN_THRESHOLD_LEFT 140

#define CAN_THRESHOLD_RIGHT 140


#define GARBAGE_THRESHOLD 450


int lineSensors[4];

int distanceSensors[3];

int leftMotorSpeed = 0;

int rightMotorSpeed = 0;

int frontSensor = 0;
int intersections = 0;

int lineError = 0;
double lineP = 0;
double lineI = 0;
double lineD = 0;
double linePreviousError = 0;
double linePIDVal = 0;

boolean ledsOn = false;

int main(void) {

  initADC();
  initMotor();
  initLEDs();
  DDRD &= ~((1 << PD3) | (1 << PD4) | (1 << PD5));
  PORTD |= 0b00111000;
  PORTB |= 0b00000000;
  initSoftSerial();
  setLCDBackLight(255);
  clrLCD();
  idle();
  while (1) {
    frontSensor = analog(5);
    if (frontSensor >= GARBAGE_THRESHOLD) {
      clrLCD();
      motor(0, 0);
      lcdPrint("GARBAGE");
      beep(500, 1000000);
      flashLEDS();
      _delay_ms(500);
    } else {
      followLine();
    }
  }
```

```
}

/**
   OPERATIONS
*/
void intersection() {
  switch (intersections) {
    case 0:
      grabIntersection();
      break;
    case 1:
      dropIntersection();
      break;
    case 2:
      grabIntersection();
      break;
    case 3:
      dropIntersection();
      break;
  }
}

void grabIntersection() {
  motor(0, 0);
  _delay_ms(300);
  int canDirection = getCanDirection();
  clrLCD();
  if (canDirection == COMMAND_LEFT) {
    lcdPrint("LEFT");
    sendCommand(COMMAND_LEFT);
  } else if (canDirection == COMMAND_RIGHT) {
    lcdPrint("RIGHT");
    sendCommand(COMMAND_RIGHT);
  }
  _delay_ms(10000);
  motor(500, 500);
  _delay_ms(800);

  intersections++;

}


void dropIntersection() {

  motor(0, 0);

  _delay_ms(300);

  sendCommand(COMMAND_DROP);

  _delay_ms(3000);

  motor(500, 500);

  _delay_ms(800);

  intersections++;

}


void initLEDs() {

  DDRB |= (1 << PB0) | (1 << PB1) | (1 << PB2) | (1 << PB3) | (1 << PB4) | (1 << PB5)
;

  DDRD |= (1 << PD6) | (1 << PD7) ;

}


void flashLEDS() {

  if (ledsOn) {

    PORTB &= ~((1 << PB0) | (1 << PB1) | (1 << PB2) | (1 << PB3) | (1 << PB4) | (1 <<
PB5));

    PORTD &= ~((1 << PD6) | (1 << PD7));

    ledsOn = false;

  } else {

    ledsOn = true;

    PORTB |= (1 << PB0) | (1 << PB1) | (1 << PB2) | (1 << PB3) | (1 << PB4) | (1 <<
PB5);

    PORTD |= (1 << PD6) | (1 << PD7) ;

  }

  _delay_ms(500);

}


/**

   LINE FOLLOW
```

```
*/

void followLine() {

  readLineSensors();

  calculateLinePIDVal();

  PIDMotorControl(LINE_MOTOR_SPEED, linePIDVal);

}


void readLineSensors() {

  for (int i = 0; i < 4; i++) {

    lineSensors[i] = analog(i);

  }


  if ((lineSensors[0] >= LINE_THRESHOLD_LEFT) && (lineSensors[1] <
LINE_THRESHOLD_LEFT_C) && (lineSensors[2] < LINE_THRESHOLD_RIGHT_C) && (lineSensors[3]
< LINE_THRESHOLD_RIGHT))

    lineError = -3;

  else if ((lineSensors[0] >= LINE_THRESHOLD_LEFT) && (lineSensors[1] >=
LINE_THRESHOLD_LEFT_C) && (lineSensors[2] < LINE_THRESHOLD_RIGHT_C) && (lineSensors[3]
< LINE_THRESHOLD_RIGHT))

    lineError = -2;

  else if ((lineSensors[0] < LINE_THRESHOLD_LEFT) && (lineSensors[1] >=
LINE_THRESHOLD_LEFT_C) && (lineSensors[2] < LINE_THRESHOLD_RIGHT_C) && (lineSensors[3]
< LINE_THRESHOLD_RIGHT))

    lineError = -1;

  else if ((lineSensors[0] < LINE_THRESHOLD_LEFT) && (lineSensors[1] >=
LINE_THRESHOLD_LEFT_C) && (lineSensors[2] >= LINE_THRESHOLD_RIGHT_C) &&
(lineSensors[3] < LINE_THRESHOLD_RIGHT))

    lineError = 0;

  else if ((lineSensors[0] < LINE_THRESHOLD_LEFT) && (lineSensors[1] <
LINE_THRESHOLD_LEFT_C) && (lineSensors[2] >= LINE_THRESHOLD_RIGHT_C) &&
(lineSensors[3] < LINE_THRESHOLD_RIGHT))

    lineError = 1;

  else if ((lineSensors[0] < LINE_THRESHOLD_LEFT) && (lineSensors[1] <
LINE_THRESHOLD_LEFT_C) && (lineSensors[2] >= LINE_THRESHOLD_RIGHT_C) &&
(lineSensors[3] >= LINE_THRESHOLD_RIGHT))

    lineError = 2;

  else if ((lineSensors[0] < LINE_THRESHOLD_LEFT) && (lineSensors[1] <
LINE_THRESHOLD_LEFT_C) && (lineSensors[2] < LINE_THRESHOLD_RIGHT_C) && (lineSensors[3]
>= LINE_THRESHOLD_RIGHT))

    lineError = 3;

  else if ((lineSensors[0] < LINE_THRESHOLD_LEFT) && (lineSensors[1] <
LINE_THRESHOLD_LEFT_C) && (lineSensors[2] < LINE_THRESHOLD_RIGHT_C) && (lineSensors[3]
< LINE_THRESHOLD_RIGHT)) {
    if (lineError < 0) {
      lineError = -5;
    } else {
      lineError = 5;
    }
  } else if ((lineSensors[0] >= LINE_THRESHOLD_LEFT) && (lineSensors[1] >=
LINE_THRESHOLD_LEFT_C) && (lineSensors[2] >= LINE_THRESHOLD_RIGHT_C) &&
(lineSensors[3] >= LINE_THRESHOLD_RIGHT)) {
    intersection();
  }


}


void calculateLinePIDVal() {
  lineP = lineError;
  lineI = lineI + lineError;
  lineD = lineError - linePreviousError;

  linePIDVal = (LINE_Kp * lineP) + (LINE_Ki * lineI) + (LINE_Kd * lineD);

  linePreviousError = lineError;
}

void PIDMotorControl(int motorSpeed, double pidVal) {
  leftMotorSpeed = motorSpeed + pidVal;
  rightMotorSpeed = motorSpeed - pidVal;

  constrain(leftMotorSpeed, 0, 1000);
  constrain(rightMotorSpeed, 0 , 1000);

  motor(leftMotorSpeed, rightMotorSpeed);
}

/**
   CAN/GARBAGE SENSING
*/
```

```
void printDistanceVals() {
  distanceSensors[0] = analog(4);
  distanceSensors[1] = analog(5);
  distanceSensors[2] = analog(6);
  clrLCD();
  moveLCDCursor(0);
  lcdPrintDec(distanceSensors[0]);
  moveLCDCursor(8);
  lcdPrintDec(distanceSensors[1]);
  moveLCDCursor(16);
  lcdPrintDec(distanceSensors[2]);
}

int getCanDirection() {
  distanceSensors[0] = analog(4);
  distanceSensors[1] = analog(6);
  if (distanceSensors[0] > CAN_THRESHOLD_LEFT) {
    return COMMAND_LEFT;
  } else if (distanceSensors[1] > CAN_THRESHOLD_RIGHT) {
    return (COMMAND_RIGHT);
  } else {
    return COMMAND_IDLE;
  }
}

/**
   COMMS
*/
void setComMode(int mode) {
  switch (mode) {
    case MODE_TRANSMIT:
      DDRC |= (1 << PC4) | (1 << PC5);
      break;
    case MODE_RECIEVE:
      DDRC &= ~((1 << PC4) | (1 << PC5));
      break;
  }
}

void idle() {
  PORTC &= ~((1 << PC4) | (1 << PC5));
}

void sendCommand(int command) {
  setComMode(MODE_TRANSMIT);
  switch (command) {
    case COMMAND_IDLE:
      PORTC &= ~((1 << PC4) | (1 << PC5));
      break;
    case COMMAND_LEFT:
      PORTC &= ~(1 << PC4);
      PORTC |= (1 << PC5);
      break;
    case COMMAND_RIGHT:
      PORTC &= ~(1 << PC5);
      PORTC |= (1 << PC4);
      break;
    case COMMAND_DROP:
      PORTC |= ((1 << PC4) | (1 << PC5));
      break;
  }
  _delay_ms(50);
  idle();
}
```

EVSHield

```
#include <Wire.h>
#include <EVShield.h>
#include <EVs_NXTCam.h>

#define MODE_TRANSMIT 0
```

```
#define MODE_RECIEVE 1

#define COMMAND_IDLE 0
#define COMMAND_LEFT 1
#define COMMAND_RIGHT 2
#define COMMAND_DROP 3

#define ARM_POS_LEFT 2001
#define ARM_POS_RIGHT 2002

#define MOTOR_1_SPEED 20
#define MOTOR_1_TURN_DELAY 900
#define MOTOR_2_SPEED 10
#define MOTOR_2_GRAB_DELAY 1350

#define COLOR_RED 5
#define COLOR_BLUE 6

EVShield evShield(0x34, 0x36);
EVs_NXTCam nxtCam;

int currentCommand;
int currentColor;
int currentArmPos = ARM_POS_RIGHT;
int nblobs;
uint8_t color[8];
uint8_t left[8];
uint8_t top[8];
uint8_t bottom[8];
uint8_t right[8];

void setup() {
  Serial.begin(9600);
  evShield.init(SH_HardwareI2C);
  evShield.bank_a.motorReset();
  evShield.bank_b.motorReset();
  nxtCam.init(&evShield, SH_BAS1);
  nxtCam.disableTracking();
  nxtCam.selectObjectMode();
  nxtCam.sortSize();
  delay(1000);
  armRight();
  delay(500);
  setComMode(MODE_RECIEVE);
  //blinkLed(5, 0, 255, 0);
}

void loop() {
  //  if (evShield.getButtonState(BTN_LEFT) == true) {
  //    grabCan();
  //    delay(100);
  //  } else if (evShield.getButtonState(BTN_RIGHT) == true) {
  //    releaseCan();
  //    delay(100);
  //  }
  currentCommand = getCommand();
  if (currentCommand != COMMAND_IDLE) {
    switch (currentCommand) {
      case COMMAND_LEFT:
        leftSubroutine();
        break;
      case COMMAND_RIGHT:
        rightSubroutine();
        break;
      case COMMAND_DROP:
        releaseCan();
        delay(500);
        break;
    }
  }
}
```

```
void leftSubroutine() {
  evShield.bank_b.ledSetRGB(255, 255, 255);
  armLeft();
  delay(500);
  grabCan();
  delay(500);
  currentColor = getColor();
  evShield.bank_b.ledSetRGB(0, 0, 0);
  delay(50);
  if (currentColor == COLOR_RED) {
    evShield.bank_a.ledSetRGB(255, 0, 0);
    if (currentArmPos == ARM_POS_RIGHT) {
      armLeft();
    }
  } else {
    evShield.bank_a.ledSetRGB(0, 0, 255);
    if (currentArmPos == ARM_POS_LEFT) {
      armRight();
    }
  }
  delay(2000);
}

void rightSubroutine() {
  evShield.bank_a.ledSetRGB(255, 255, 255);
  armRight();
  delay(500);
  grabCan();
  delay(500);
  currentColor = getColor();
  evShield.bank_a.ledSetRGB(0, 0, 0);
  delay(50);
  if (currentColor == COLOR_RED) {
    evShield.bank_a.ledSetRGB(255, 0, 0);
    if (currentArmPos == ARM_POS_RIGHT) {
      armLeft();
    }
  } else {
    evShield.bank_a.ledSetRGB(0, 0, 255);
    if (currentArmPos == ARM_POS_LEFT) {
      armRight();
    }
  }
  delay(2000);
}

void blinkLed(int t, int R, int G, int B) {
  bool ledState = false;
  for (int i = 0; i < t * 2; i++) {
    if (ledState) {
      evShield.bank_a.ledSetRGB(0, 0, 0);
      evShield.bank_b.ledSetRGB(0, 0, 0);
      ledState = false;
    } else {
      evShield.bank_a.ledSetRGB(R, G, B);
      evShield.bank_b.ledSetRGB(R, G, B);
      ledState = true;
    }
    delay(500);
  }
  evShield.bank_a.ledSetRGB(0, 0, 0);
  evShield.bank_b.ledSetRGB(0, 0, 0);
}

/**
   COMMS
*/
void setComMode(int mode) {
  switch (mode) {
    case MODE_TRANSMIT:
      pinMode(8, OUTPUT);
      pinMode(9, OUTPUT);
      break;
    case MODE_RECIEVE:
      pinMode(8, INPUT);
      pinMode(9, INPUT);
      break;
  }
}

int getCommand() {
  if (digitalRead(8) == 0 && digitalRead(9) == 0) {
    return COMMAND_IDLE;
  } else if (digitalRead(8) == 0 && digitalRead(9) == 1) {
    return COMMAND_LEFT;
  } else if (digitalRead(8) == 1 && digitalRead(9) == 0) {
    return COMMAND_RIGHT;
  } else if (digitalRead(8) == 1 && digitalRead(9) == 1) {
    return COMMAND_DROP;
  }
  return COMMAND_IDLE;
}

void sendConfirmation() {
  setComMode(MODE_TRANSMIT);
  digitalWrite(8, HIGH);
  digitalWrite(9, HIGH);
  delay(10);
  digitalWrite(8, LOW);
  digitalWrite(9, LOW);
  setComMode(MODE_RECIEVE);
}

/**
   Camera
*/
int getColor() {
  if (currentArmPos != ARM_POS_LEFT) {
    armLeft();
  }
  delay(500);
  nxtCam.enableTracking();
  delay(1000);
  while (1) {
    nxtCam.issueCommand('J');
    delay(500);
    nxtCam.getBlobs(&nblobs, color, left, top, right, bottom);
    delay(500);
    nxtCam.issueCommand('K');

    uint8_t i;
    for (int i = 0; i < nblobs; i++) {
      if (color[i] == 2)  {
        nxtCam.disableTracking();
        return COLOR_RED;
      }
      if (color[i] == 3)  {
        nxtCam.enableTracking();
        return COLOR_BLUE;
      }
    }
  }
}

/**
   MOTOR 1 CONTROLS
*/
void armLeft() {
  evShield.bank_a.motorRunUnlimited(SH_Motor_1,
                                    SH_Direction_Forward,
                                    MOTOR_1_SPEED);
  delay(MOTOR_1_TURN_DELAY);
  evShield.bank_a.motorStop(SH_Motor_1, SH_Next_Action_BrakeHold);
  delay(500);
  currentArmPos = ARM_POS_LEFT;
}

void armRight() {
  evShield.bank_a.motorRunUnlimited(SH_Motor_1,
                                    SH_Direction_Reverse,
                                    MOTOR_1_SPEED);
  delay(MOTOR_1_TURN_DELAY);
  evShield.bank_a.motorStop(SH_Motor_1, SH_Next_Action_BrakeHold);
  delay(500);
  currentArmPos = ARM_POS_RIGHT;
}

/**
   MOTOR 2 Controls
*/
void grabCan() {
  evShield.bank_b.motorRunUnlimited(SH_Motor_1,
                                    SH_Direction_Forward,
                                    MOTOR_2_SPEED);
  delay(MOTOR_2_GRAB_DELAY);
  evShield.bank_b.motorRunUnlimited(SH_Motor_1,
                                    SH_Direction_Forward,
                                    5);
}

void releaseCan() {
  evShield.bank_b.motorRunUnlimited(SH_Motor_1,
                                    SH_Direction_Reverse,
                                    MOTOR_2_SPEED);
  delay(MOTOR_2_GRAB_DELAY - 100);
  evShield.bank_b.motorStop(SH_Motor_1, SH_Next_Action_BrakeHold);
}
```