# MAE 3780 Final Report

David van Wijk
Emily Youtt

December 2019

# 1 Introduction

The objective of the MAE 3780 Mechatronics Final Project is to design and build a robot to participate in the tournament style "Cube Craze" competition. Using identical design and cost constraints, groups of two and three students put their robots head to head in each match, ultimately attempting to bring back a larger number of cubes onto their respective starting zones after a period of one minute. The robots must be autonomous (ie. cannot receive control inputs from the participants during the matches) besides starting and stopping. To control the robot's actions, each group makes use of an Arduino Uno and a number of sensors. This report will outline our design ideation, manufacturing process and strategy development. We will analyze our design strengths and weaknesses and provide a detailed discussion of the competition performance.

# 2 Design Motivation

In initially developing ideas for robot designs, we thought about both constraints, and objectives that we wanted to achieve. To eliminated variables from our theoretical design equation, we created these constraints in addition to the preexisting competition constraints. From this list we conceptualized a design that both met the needs of our constraints and the demands of our objectives.

## 2.1 Design Constraints

To be comprehensive in our design motivation, we listed both the competition constraints and the ones we put on our design. Items 1-4 are the constraints on our design from the rules document, while items 5-6 are constraints we decided on.

1. The top-down profile of the robot will fit in an 8 inch x 8 inch square prior to the start of the round.
2. In its largest form, the robot must fit within an 18 inch diameter cylinder.
3. The robot's micro-controller and all accessories will be powered by one 9V battery and four AA batteries.
4. The wheels must be driven by Parallax Continuous Rotation Servos (#900-00008).
5. The robot will use the provided chassis and wheels.
6. The robot will be made from materials provided either in the lab or in the Emerson Machine Shop.

The constraint we placed on using the provided chassis and wheels was because we did not want to over-complicate the overall design and manufacturing aspects of the project. The chassis that was provided had ample mounting holes to accommodate add-ons such as the arms and sensors (See Figure 2 for full integration of arms onto robot). The wheels were large enough for our robot to drive at a reasonable speed. We briefly considered making our own chassis, however it seemed like lot of extra time and a large monetary cost for a minimal (if any) improvement to our design.

We decided to place a constraint on only using materials that were easily accessible because we deemed the resources that we had in the lab and the Emerson Machine Shop sufficient to create a fully functional robot. Table 1 shows all of the parts used on our robot, all from Cornell suppliers. Waiting for parts to arrive and risking that they fail with no chance of replacement were the main

motivations behind this constraint. In the end, this proved to be beneficial because we were able to replace a broken motor very close to the date of the competition.

## 2.2 Design Objectives

Our objectives for the functionality of the robot are as follows:

1. The robot should be able to grab blocks in addition to pushing blocks.
2. The robot's arms should be able to move after they are deployed (full positional control).
3. Sensor placement should allow for the robot to know if it is near a black edge at all times.
4. The robot's arms should be able to move independently of each other.

The objectives listed above are in order of most important to least important in our design priority. After watching the competition videos from the previous year, we saw that the teams who were most effective used a large arm and a pushing method to get the blocks on the other team's side. While our competition objective is the opposite (we need to "retrieve" blocks), the method of pushing would still be effective if we could first clear a path to get to their side and begin pushing towards ours. Thus, we decided that our arms needed to be able to grab blocks at the start and subsequently push blocks, which inspired our first design objective. Figure 1 shows the robot in "collection mode" with its arms past perpendicular to the robot. At the ends of the arms, we added small pieces of acrylic to be sure that the blocks would stay in the vicinity of the robot.

The second and fourth objectives are closely related; our goal was to be able to switch the arms between modes of collecting and pushing. Therefore, we could not have static arms that deployed at the start of the round and never moved afterwards. The second objective could be met using one motor attached to a gear train that moves both arms the same amount, at the same time. Although our overall strategy would work with both arms being rotated the same amount at all times (which would be the case with 1 motor), we also wanted to attempt to allow the arms to be at different positions relative to each other (i.e. one parallel to the chassis and one perpendicular to it), hence the fourth objective. We achieved this goal by using two motors, one for each arm after ensuring that we had enough money in our budget and ports on our Arduino to achieve this. Figure 3 clearly shows the connections between the motors and arms. Gears fixed to the motors mesh with gears built into the arm design, allowing them to rotate over 180 degrees. Figure 4 shows a more in-depth look at the intricacies of the arm design.

The third objective was important because when the arms were fully extended in front of the chassis and the QTI sensor was underneath the chassis, the robot would have deposited all of the collected blocks off of the arena before sensing the black border. There would also have been losses in algorithm efficiency if the robot was pushing blocks until a sensor underneath the robot recognized the black border. To achieve this objective, we attached one QTI sensor at the end of each arm. In Figure 1, the duct taped QTI sensors can be easily seen. A third QTI sensor was mounted at the back of the chassis to prevent the robot from driving off of the board backwards.

## 2.3 Overall Design Comments

Our final design met all of the constraints and objectives that we set at the beginning of the project. We were even able to implement the arms moving independent of each other. The arms were reliable and could collect blocks in addition to pushing them.

### 2.4 Cost Analysis

Table 1 in the Appendix has a comprehensive list of all parts included on our robot. The only parts we purchased outside of the provided equipment was a sheet of acrylic and one extra servo motor. The acrylic is the only component we used to construct the arms outside of the provided screws, nuts, and standoffs. Our total cost for the entire robot was $8 outside of the supplies that everyone was provided. If the costs of the Arduino, sensors, and other components were taken into account, the robot would likely cost around $35 without factoring cost of labor.

## 3 Strategy Flow Chart

For our final algorithm, we split up our code into three different phases. Figure 5 features the strategy flow chart, where the colored boxes indicate the start of a phase. The first phase of our algorithm was intended to move approximately a third of the blocks onto the starting side within the first 5-10 seconds. Based on if the robot started on the left, right or center, the robot moved forward and turned ninety degrees at the interface of the colors, and then traveled along the border collecting cubes for 2.5 seconds. Then, the robot turned and brought cubes back to the starting side. This initialized the second phase, in which the robot retracted the arms to the starting position and reversed. Retracting the arms before reversing onto the opposite color made sure no stray blocks were accidentally pushed onto the opposite side during this maneuver. After reaching the opposite side, the third phase began. In this infinite loop, we intended we to angle the robot to one side slightly, and then perform a series of angled forward and reverse motions, effectively having an infinite pushing sequence. Segmenting our code into different phases was very helpful as it allowed us to reuse different parts of our code even if the starting positions varied.

## 4 Robot's Strengths and Weaknesses

Section 2 describes the motivation behind our design. To summarize the mechanical design, we implemented two arms that fold out from the side of the vehicle and can be positioned at different angles relative to the vehicle and each other (see Figure 2 for retracted state and Figure 1 for partially extended). Although we met all of our constraints and objectives, there were still aspects of the robot that could have been improved.

### 4.1 Strengths

The main strength of our design is that the arms have such a wide reach and can be moved to different positions throughout the match. A lot of other robots were not able to move the arms after they were deployed, making it hard for them to both collect and push blocks. Instead, these robots were designed to collect the cubes in their chassis and hope that they would end up parking their robot back on their original side at the beginning of the match. With our design and algorithm, however, we are able to ensure that the cubes we collected or pushed would end up on our original side because we do not permanently store them. The pushing method was quite effective in our competition runs as we were able to push all cubes back onto our side in the matches that our QTI sensor worked (see Section 4.2 for more details on issues with the QTI sensors).

Another strength of our design was the fact that we designed all of our mechanical components in Autodesk Fusion. Firstly, this allowed us to work on the design at the same time, which increased overall efficiency. Secondly, since we used a number of small pieces, we were able to laser cut back

up pieces which was useful when we were testing the sturdiness of the arm actuation. We had to replace a few pieces more than once, and we were also able to easily implement design changes. Figure 4 features a render of our arm assembly.

### 4.2 Weaknesses

The main weakness of the robot was inaccurate sensor readings. All of our testing of the three QTI sensors on our robot indicated that they were all functional and configured correctly. We used different resistors to tune the sensors to be more reliable at differing distances from the ground. Despite all of this testing and preparation, on the day of the competition, the left QTI sensor did not work reliably which caused our robot to behave unpredictably for three out of the five matches. At the beginning of our code, we used the QTI sensors at the ends of the arms once deployed to determine the initial position of the robot (see beginning of Figure 5); faulty data from the left QTI made the robot execute code that was meant for a different starting position. We could have added little shields in front of the sensors or included a data averaging portion to our algorithm to ensure that the initial value that the QTI sensors reported was accurate.

Another weakness in our design was the lack of feedback on the positions of the arms. Since each arm was connected to a servo motor without feedback (refer to Figure 3 for a clear view of arm actuation), if another robot were to bump into the arms, it would alter the position of the arm and the robot would not be able to notice. We tried to mitigate the effects of this by including a "reset arms" action in our code at certain time intervals that would move them back to their fully retracted position and then move them forward to where we originally wanted them to be. This worked sufficiently, however if we wanted to be more precise, we would get a stepper motor with feedback so we could tell what angle the arms were at. If they were at the wrong angle, we could then adjust them back to the desired position.

Another weakness of the design is the differing motor speeds for the arms. One of the motors that controls the arms started to spin slower than the other, causing the arms to be lopsided. We did not have enough time to implement effective PWM control of the faster motor, so we spun the motor for the slower arm for a longer time to get to the same position of the faster one. If we had more time, we would be able to use PWM in order to eliminate the need to tune both arms individually. In addition, the differing motor speeds for both the arms and the wheels with fully charged batteries versus with used ones, changed the angle we turned or how far the arms moved with the same length of signal. This made some of our turns in the competition longer than we expected. We should have tested with new batteries before the competition to ensure we had the correct turn delays.

## 5 Competition Performance

### 5.1 User Positioning of Robot Led to Setback

On the morning of the competition, we did a number of test runs on the practice board that had been left outside in the hallway of the fourth floor of Upson Hall. We tested the robustness of the mechanical structure and arm deployment mechanisms, as well as the general flow of our algorithm. We tested a left, center and right side start three times for each, and were pleased with the results, as the robot behaved in accordance with our intended algorithm. However, during the first match, we experienced a mechanical problem that cost us the match. Due to the robot's initial placement relative to the left edge, when the left arm deployed, it just barely caught the edge of the board. This caused the contact points of the arm with the ground, clearly visible in Figure 4, to

jam into the side of the board, and after that point, the torque required for the left arm motor was not enough to free the arm and pull it onto the board. The robot remained stuck in this position for the entire match, unable to move forward. This first problem was not difficult to fix however. In the next match, the robot was placed slightly further from the edge, so that when the left arm deployed, the furthest point on the arm would never go over the edge of the border, and we would not run the risk of the arm getting stuck on the side of the board. This problem never occurred in any future matches, but there were a number of other problems that we soon discovered.

### 5.2 Success of Algorithm in Second Match

In the second match, our vision of the robot's performance was almost entirely mirrored on the board. The arms deployed perfectly, and the left QTI sensor successfully detected the black border, while the right QTI sensor detected not black. The robot proceeded into phase 1, and once it reached the yellow and blue border, turned ninety degrees clockwise as viewed from above. It collected blocks along the border for 2.5 seconds, and then turned ninety clockwise again, driving forward until the black border was reached. This successfully brought back a significant number of blocks to our starting side. The robot immediately began the phase two sequence, in which the arms swung backwards, reducing the robots overall areal profile to prevent pushing any blocks onto their side when the robot reversed. After reversing successfully to about two thirds of the way into the opposite side, phase three was executed without a hitch. Entering the infinite while loop, the robot began the push sequence, performing a series of short forward and backward movements as described in detail in the strategy flow chart. This match resulted in a six point victory, and was a successful demonstration of our algorithm, sensors and mechanical components working in harmony.

### 5.3 Sensor Failure Begins

In the third and fourth matches, we noticed a major problem that stemmed from the reliability of the left QTI sensor. After the successful deployment of the arms, the left QTI sensor read not black, even though it was directly on top of the black left border. This set our variable "side" to 2, which resulted in the execution of the algorithm we had written for a central starting position. The robot proceeded to turn right ninety degrees and move forward until the QTI sensors detected the rightmost black border. It then attempted to execute the rest of the central algorithm but was soon stopped short after clashing with the opposing robot. The two robots remained fixed in place for the rest of the competition, unable to move any additional blocks. However, due to the initial sensor error, the robot wasted precious time driving across the rear end of our side, not bringing any blocks onto the starting side. This resulted in a loss in the third round. In the fourth round, the left QTI sensor again did not detect the left black border, and it again executed the central algorithm, resulting in a loss.

### 5.4 Attempt to Correct for Sensor Failure with Robot Placement

After this sensor error in rounds three and four, we concluded that our left QTI sensor was broken or was unable to detect black. Without having enough time to replace the left QTI sensor between the fourth and fifth round, we improvised our strategy, assuming the robot would execute the central algorithm. Instead of positioning the robot forward, parallel to the border, we angled the robot toward the left side, in the hopes that the robot would turn ninety degrees and drive forward onto the opposite side, collected blocks until it hit the furthermost border. However, upon deploying the arms, the left QTI successfully detected black, which caused the robot to execute the left-side

algorithm. It drove forward and because it was angled towards the left border, by the time the right QTI detected black, triggering our external pin-change interrupt, the left wheel had already driven off of the border. This caused the robot to become stuck, and just like in the first round, it remained stuck for the entirety of the match, resulting in a loss.

### 5.5 Fixing Competition Performance Issues

This frustrating sequence of events made us reconsider our algorithm design choices, as well as some aspects of the mechanical design. To address the problem that arose with the QTI sensor, we could have done a number of things differently. Firstly, as can be seen in Figure 1, the QTI sensors were attached using duct tape, which could have caused the QTI sensors to shift slightly while the robot moved. As the QTI sensors are highly height sensitive, we could have improved this aspect of our design by incorporating QTI mounts the arms that we designed. This would also have made it easier to switch out any misbehaving QTI sensors during the competition. We also could have tried putting small shields around the sensors so no other light could interfere with them when they were collecting information.

Secondly, we could have used an average of QTI readings to make the decision on what side we were on, as we did with the color sensor to obtain the start color. We could have had a small for loop obtaining 20-50 QTI readings and averaged these values. If this average value was larger than 0.80 for example, we would be fairly confident that the QTI was reading black. This would account for any anomalies in the QTI readings and hopefully would have provided a more reliable start input.

Thirdly, we could have increased the rigidity of the mechanical arms to maintain a straighter position when extending. This would have prevented the arms from sagging slightly when they travelled over the edge of the board, which caused them to get stuck on the edge of the board in the first round. We could have had a more sturdy tightening mechanism, so simply added a short segment of string or wire towards the end of the arm to maintain tension when fully extended. All of these problems if implemented could hopefully have significantly improved the overall competition performance.

## 6 Conclusion

Over the course of the semester, we learned a great deal about analog circuits, active and passive components, wiring, and coding in C. This project tested our understanding of circuit theory and coding more than any exam or prelim. We learned how to integrate the mechanical design skills we practiced in MAE 2250, with the electrical design we studied in MAE 3780, to produce a functional, fully autonomous robot. It was both challenging and intriguing to solve problems that involved both mechanical and electrical knowledge for the first time. We would of course have liked to do better in the competition, but after a thorough analysis of our mechanical design and algorithm, we maintain that our electrical and mechanical design was sound. In the future, we should anticipate possible sensor failure and have a plan for replacing malfunctioning sensors, or have multiple checks in place so as not to rely heavily on initial sensor readings. Our arm design was challenging to implement but rewarding when it worked as it allowed our robot to be more modular than most other robots in the competition. Overall, this was an exciting and rewarding project and was the perfect intersection of electrical and mechanical engineering.

# 7 Appendix

## 7.1 Figures

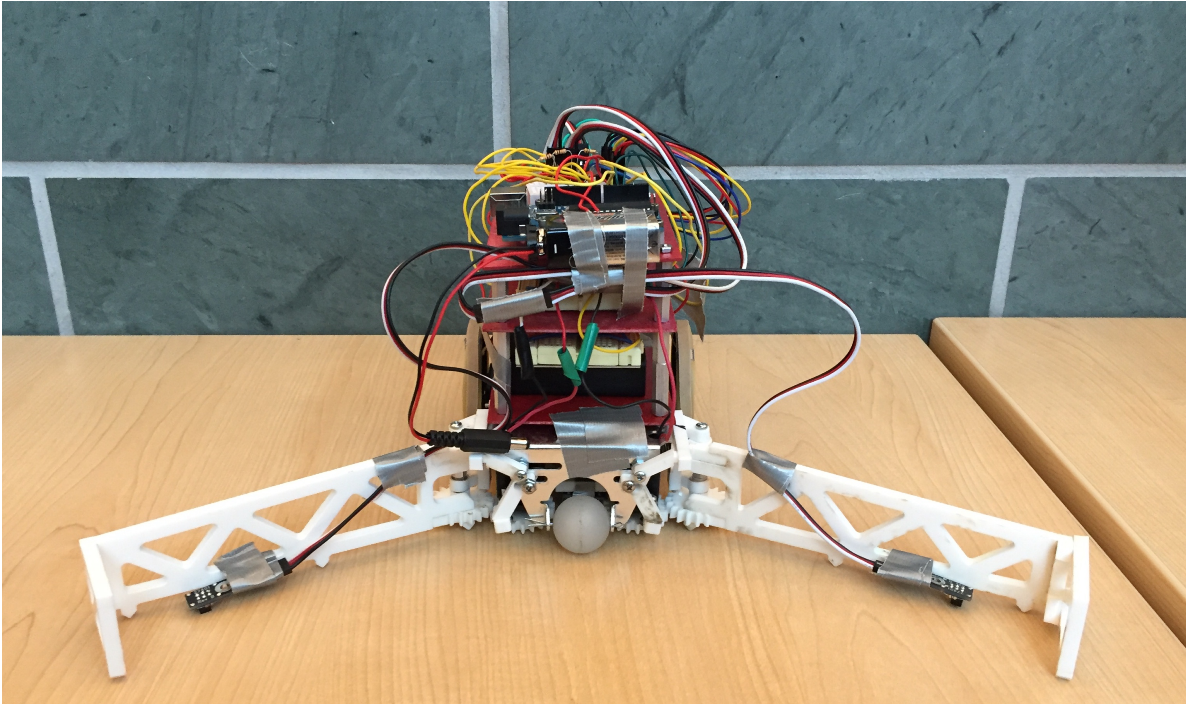

Figure 1: Final Robot, Collection Mode

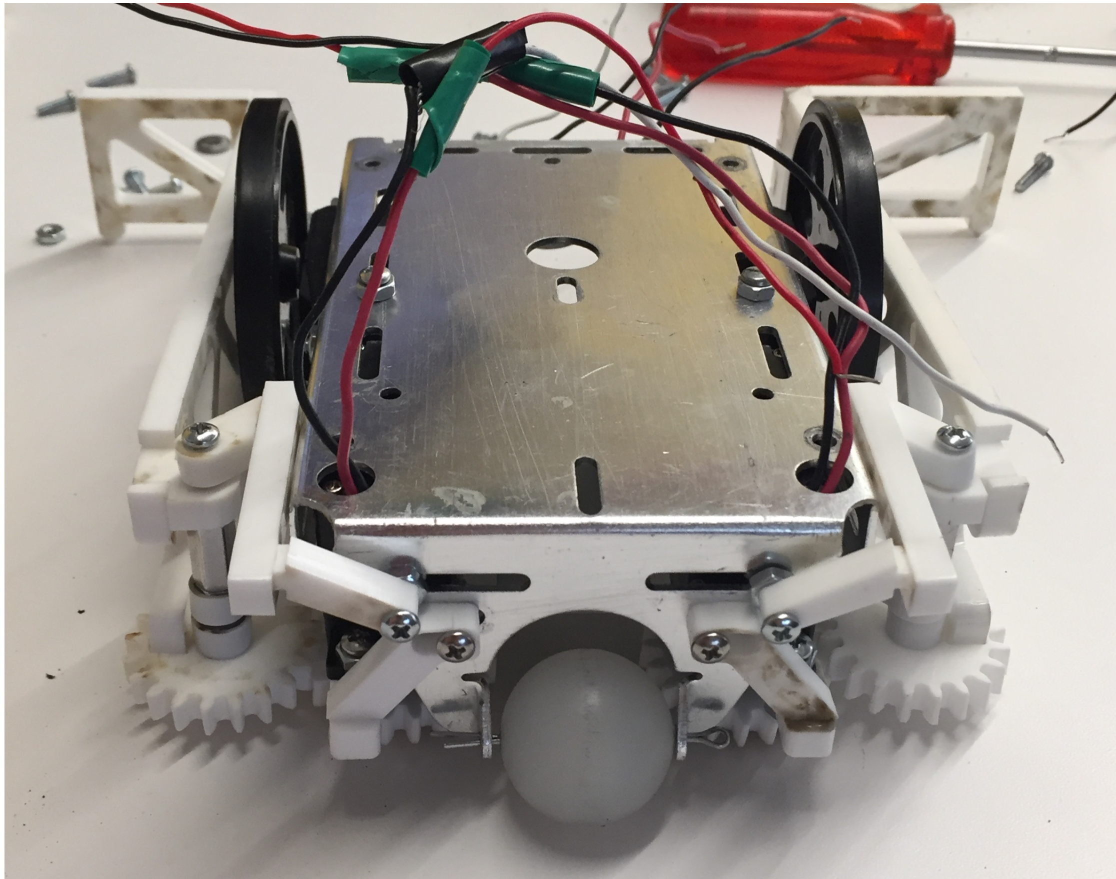

Figure 2: Complete Mechanical Design

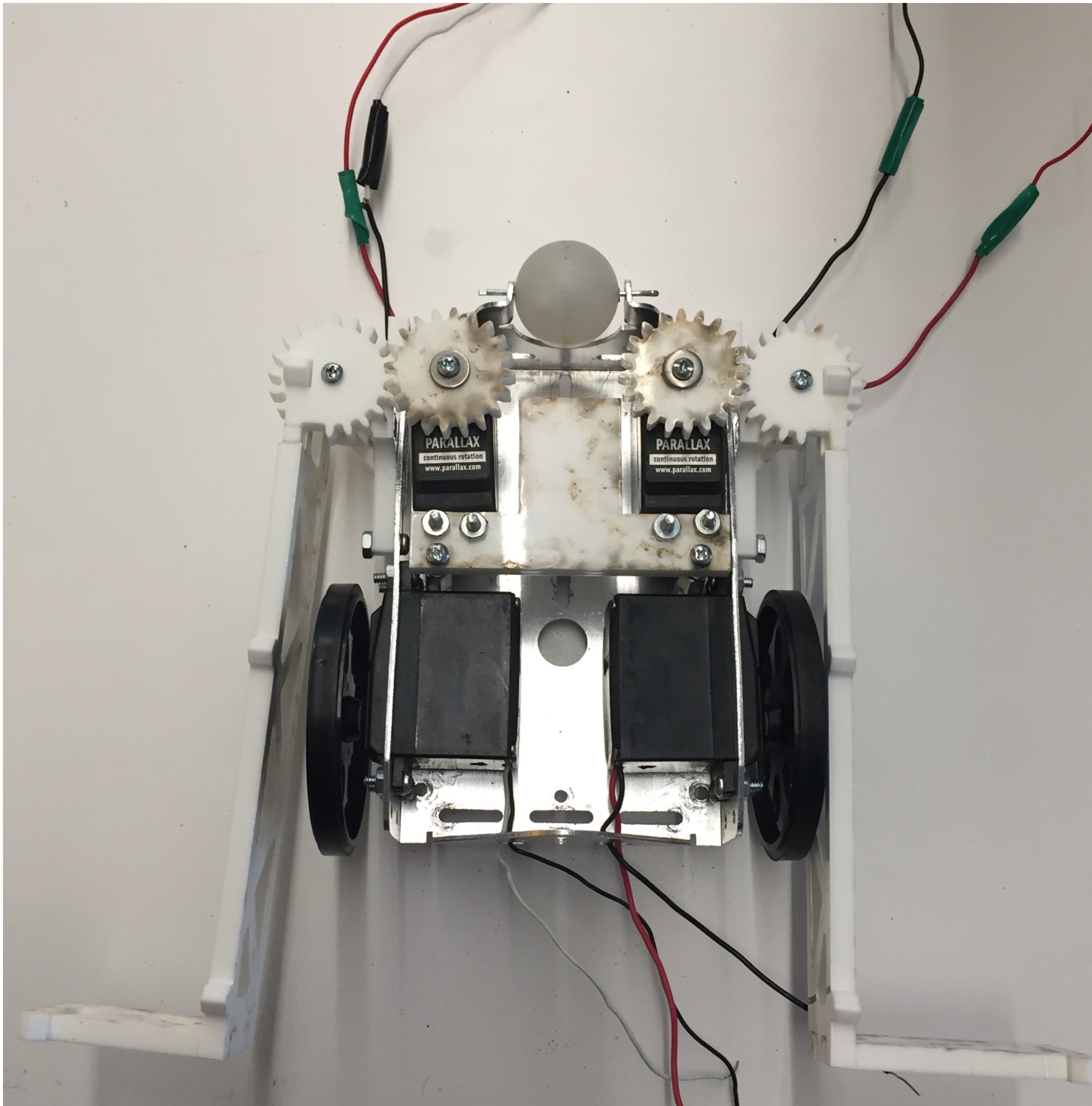


Figure 3: Arm Actuation Mechanism

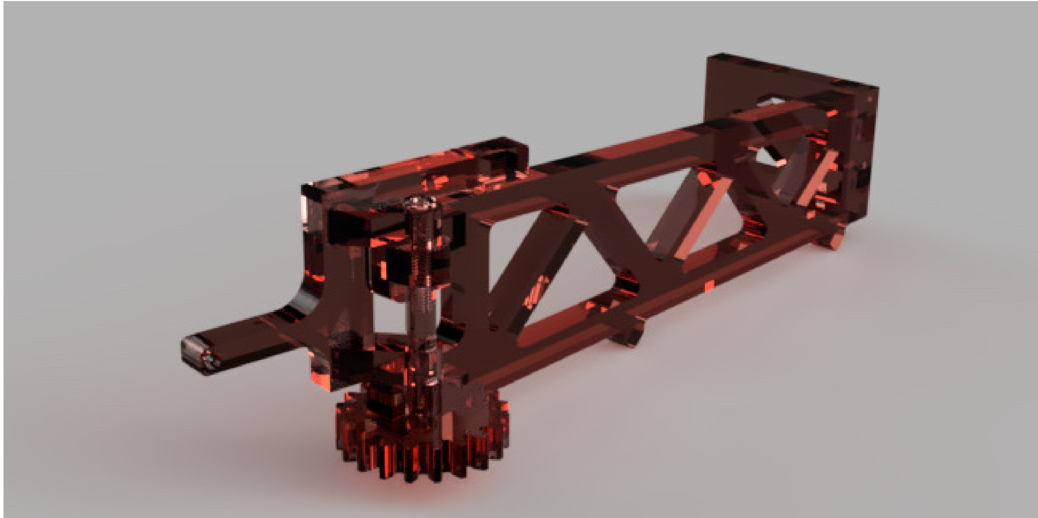

Figure 4: CAD Render, Arm Assembly

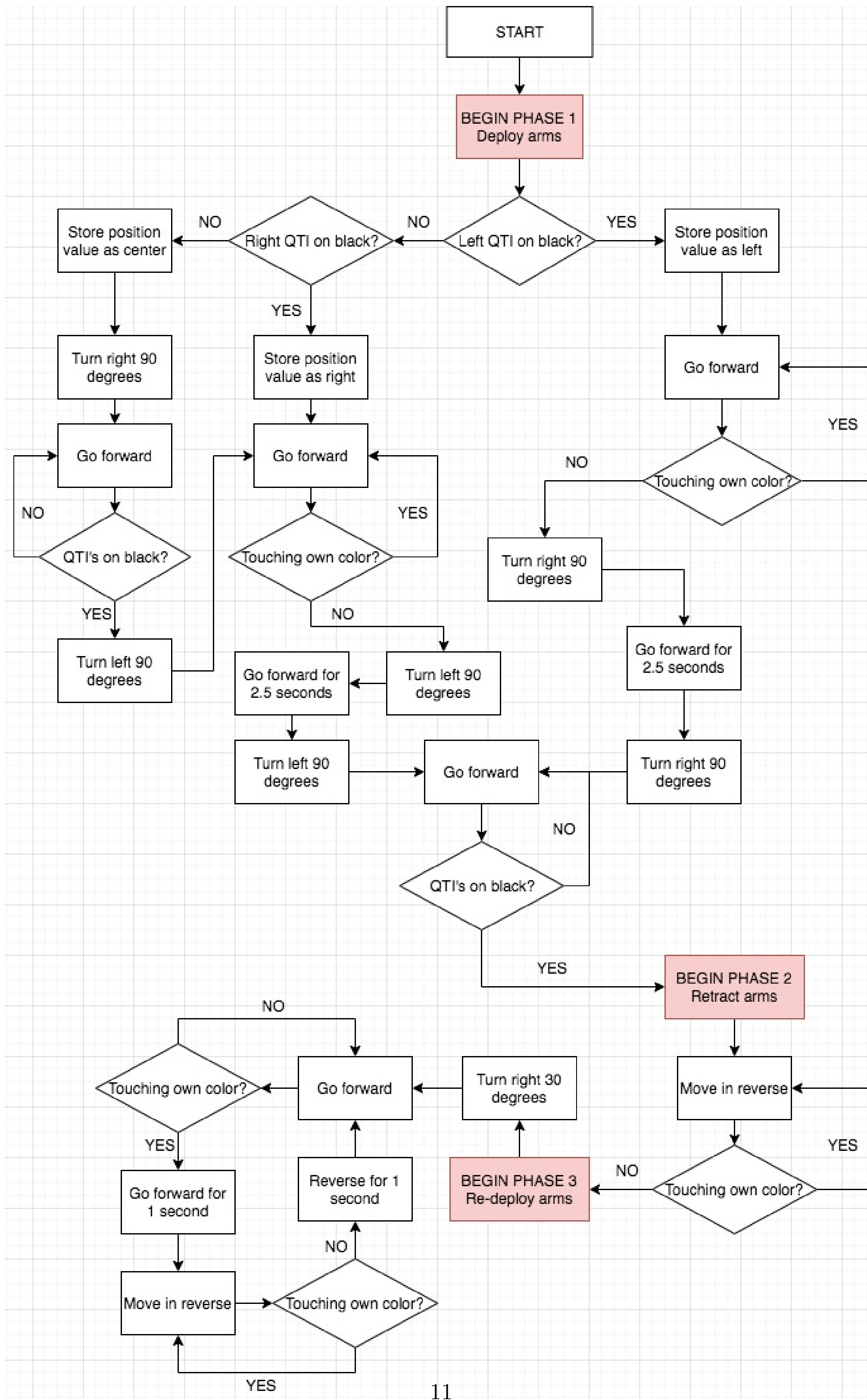




Figure 5: Strategy Flow Chart

### 7.2 Parts List

| Part | Supplier | Quantity | Cost per Item |
|---|---|---|---|
| Chassis | MAE Lab | 1 | $0 |
| Wheels | MAE Lab | 1 | $0 |
| Parallax Continuous Rotation Servos (provided) | MAE Lab | 3 | $0 |
| Parallax Continuous Rotation Servos (extra) | MAE Lab | 1 | $3 |
| 0.22” x 12” x 12” Acrylic Sheet | Emerson Machine Shop | 1 | $5 |
| #4-40 x 0.5” Phillips Head Screw | MAE Lab | 8 | $0 |
| #4-40 x 0.25” Phillips Head Screw | MAE Lab | 8 | $0 |
| #4-40 x 0.125” Phillips Head Screw | MAE Lab | 3 | $0 |
| #4-40 Nut | MAE Lab | 3 | $0 |
| #4-40 x 1” Male-Female Standoff | MAE Lab | 6 | $0 |
| #4-40 x 0.5” Male-Female Standoff | MAE Lab | 10 | $0 |
| #4-40 x 0.5” Female-Female Standoff | MAE Lab | 6 | $0 |
| Arduino Uno | MAE Lab | 1 | $0 |
| Breadboard | MAE Lab | 2 | $0 |
| Transistor | MAE Lab | 16 | $0 |
| 1000 Ω Resistor | MAE Lab | 16 | $0 |
| Diode | MAE Lab | 16 | $0 |
| Color Sensor | MAE Lab | 1 | $0 |
| QTI Sensor | MAE Lab | 3 | $0 |
| | | Total Cost | $8 |

Table 1: Comprehensive Parts List for Robot

## 7.3 Final Code

```c
/*
 * MAE3780_FinalCode.c
 *
 * Created: 11/19/2019 4:05:03 PM
 * Author : David van Wijk & Emily Youtt
 */


//Acquire the goods and get this bread
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include "serial.h"
void goForward()
{
   PORTD &= 0b10011111; // set PD5 and PD6 to low
   PORTD |= 0b01000000; // set PD6 to high (setting P0 to high)
   PORTD &= 0b01101111; // set PD4 to low (P0 = 0) so no short circuit
   PORTD |= 0b00010000; // set PD4 to high (setting P1 to high)
}
void goBack()
{
    PORTD &= 0b10011111; // set PD5 and PD6 to low
    PORTD |= 0b00100000; // set PD5 to high (setting P1 to high)
    PORTD &= 0b01101111; // set PD4 to low (P0 = 0) so no short circuit
    PORTD |= 0b10000000; // set PD7 to high (setting P1 to high)
}
void stop()
{
 PORTD &= 0b00001111; // set PD5 and PD6 to low (P0 = 0)

}
void turnLeft() // CCW from above CHANGE THIS SHIT!
{
      PORTD &= 0b10011111; // PD5 and PD6 low
      PORTD |= 0b01000000; // PD6 high
      PORTD &= 0b01101111; // set PD4 to low
      PORTD |= 0b10000000; // set PD7 to high
```

```c
}
void turnRight() //CW from above
{
    PORTD &= 0b10011111; // set PD5 and PD6 to low
    PORTD |= 0b00100000; // set PD5 to high
    PORTD &= 0b01101111; // set PD4 to low
    PORTD |= 0b00010000; // set PD4 to high
}


void armForward(int delayarm1, int diff)
{
    PCMSK0 &= 0b11001110;// disable interrupts when arms are moving
    PORTB &= 0b11110001; //Set PB1, PB2, and PB3 low (inputs to the H-bridge)
    PORTD &= 0b11110111; // Set PD3 to low (also input to H-bridge)
    PORTB |= 0b00001100; //Set PB2 (faster motor) and PB3 (slower motor) high
                         // to allow motors to spin the arms forward for different amounts of time
    for(int i = 0; i < delayarm1; i = i + 1) // delay until first faster
                                            // arm is in desired position
    {
        _delay_ms(1);
    }
    PORTB &= 0b11111011; // stop faster arm from moving
    for(int i = 0; i < diff; i = i + 1) // delay until second slower
                                        // arm is in desired position
    {
        _delay_ms(1);
    }
    PORTB &= 0b11110111;// stop slower arm from moving (should both now be in position)
    PCMSK0 |= 0b00110001;// re-enable interrupts from QTI
}
void armBack(int delayarm1, int diff)
{
    PCMSK0 &= 0b11001110;// disable interrupts when arms are moving
    PORTB &= 0b11110001; //Set PB1, PB2, and PB3 low (inputs to the H-bridge)
    PORTD &= 0b11110111; // Set PD3 to low (also input to H-bridge)
    PORTB |= 0b00000010; //Set PB1 (faster motor) high
    PORTD |= 0b00001000; // Set PD3 (slower motor) to high
```

```
    // to allow motors to spin the arms backwards for different amounts of time

    for(int i = 0; i < delayarm1; i = i + 1) // delay until first faster
    // arm is in desired position
    {
        _delay_ms(1);
    }
    PORTB &= 0b11111101; // stop faster arm from moving
    for(int i = 0; i < diff; i = i + 1) // delay until second slower
    // arm is in desired position
    {
        _delay_ms(1);
    }
    PORTD &= 0b11110111;// stop slower arm from moving (should both now be in position)
    PCMSK0 |= 0b00110001;// re-enable interrupts from QTI
}


int intTime1; // initialize time storing variable for gathering period for color sensor
int periodL=0; // variable for period of color sensor
int intTime2; // initialize time storing variable for gathering period for color sensor


ISR(PCINT2_vect) // interrupt that gets the period for the color sensor
{
    if((PIND&0b00000100)==0b00000100) // Check rising or falling edge
    {
        intTime2 = TCNT1; //Store current timer value at change
        } else {
        if(intTime2 > TCNT1){ // if the timer has wrapped around, find the difference between
                             // zero and the wrapped time and zero and the initial time and add them
            periodL = (((2^16)-1) - intTime2) + TCNT1;
            } else {
            periodL = TCNT1 - intTime2; // if the timer did not wrap around, subtract the initial
                                        // stored value from the current timer value
        }
    }
}
```

```
void initColor() // initializes all registers for color sensor as well as timer registers
{
    sei(); // Enable interrupts globally
    PCICR = 0b00000101; // Initialize Register B and D for PC interrupts
    DDRB &= 0b11111110; // Pin PB0 is input pin
    DDRD &= 0b11111011; // Pin PD2 is input pin
    TCCR1A = 0b00000000; // Normal mode
    TCCR1B |=0b00000001; // Set the timer prescalar to 1
    TCCR1B &=0b11111001; // Setting normal mode
}

int getColorL() // find the value of the color sensor and return it
{
    PCMSK2 |= 0b00000100; // Enable PD2 as a PC interrupt
    _delay_ms(10);          // Give the interrupt a sec
    PCMSK2 &= 0b11111011; // Prevent further interrupts
    return periodL*.0625*2; // Change to units of microseconds
}

void initQTI() // initialize registers for QTI inputs and QTI interrupt
{
    sei(); // enable global interrupts
    DDRB &= 0b11001110; // PB5 (right) and PB4 (left) and PB0 (rear) as inputs for QTI
    PCICR |=0b00000001;// enable pin change interrupts on PCMSK0 vector
    PCMSK0 |= 0b00110001; // enable pin change interrupts on PB0, PB4, and PB5
}
int QTI_L; // variable for storing value of left QTI in function
int getQTI_L() // gets the value of the left QTI sensor
{
    _delay_ms(100);
    if(PINB&0b00010000) // check if PB4 is high (input to the Arduino from left QTI)
    {
    QTI_L = 1; // return 1 if the left QTI sensor is high (on black)

    } else { // otherwise return low (not on black)
    QTI_L = 0;
    }
    return QTI_L; // return the value of the left QTI
```

```
}

int QTI_R; // variable for storing value of right QTI in function
int getQTI_R(){ // gets the value of the right QTI sensor
    _delay_ms(100);
        if(PINB&0b00100000) // check if PB5 is high (input to the Arduino from right QTI)
    {
        QTI_R = 1; // return 1 if the right QTI sensor is high (on black)

        } else {  // otherwise return low (not on black)
        QTI_R = 0;
    }
    return QTI_R; // return the value of the right QTI

}

int QTI_C; // variable for storing value of center QTI in function
int getQTI_C(){ // gets the value of the center QTI sensor
    _delay_ms(100);
    if(PINB&0b00000001) //check if PB0 is high (input to the Arduino from center QTI)
    {
        QTI_C = 1; // return 1 if the center QTI sensor is high (on black)

        } else { // otherwise return low (not on black)
        QTI_C = 0;
    }
    return QTI_C; // return the value of the center QTI

}

//side = 1 for RIGHT, side = 0 for LEFT
int translateColor(int value) // take input from getColor and assign it to a color
                              // based on experimental thresholds
{
 //Assign colors
 if (value < 100)
 {
  return 0; //yellow
 } else {
```

```
  return 1; //blue
     }
 }
int initialcolors[50]; // stores first 50 color samples when program starts to average them
int startPeriod; // stores period of color sensor initially
int startColor; // stores initial color
int currentColorL; // stores current color

int crossed;// changes to 1 when robot crosses to other side
int side; // to store where the robot starts (left, right, center)
int QTI_R; //PB5
int QTI_L; //PB4
int QTI_C; //PB0
int phase; // to tell interrupt what phase of the algorithm robot is in
int border; // to tell if the robot has reached the border in the pushing phase (phase 3)
int main(void)
{
    init_uart(); // Initialize serial
    initColor(); // Initialize the good stuff
    initQTI();


    DDRD |= 0b11111000; //set PD3 (right arm P0), PD4 (left wheel P1), PD5 (right wheel P1),
                        // PD6 (right wheel motor P0), and PD7 (left wheel motor P0)
                        // as outputs to H-bridges
    DDRB |= 0b00001110; //set PB1 (left arm P0), PB2 (left arm p1),
                        // and PB3 (right arm motor P1)as outputs for H-bridge


 phase = 1; // phase 1 of the algorithm: use arm QTIs to determine where starting, go to other color, turn,
            // return with blocks, deposit blocks
    for(int i = 0; i < 19; i = i + 1){ // average the first 20 values of the initial period to get accurate reading
        startPeriod = startPeriod + getColorL();
    }
    startPeriod = startPeriod/20;
    startColor = translateColor(startPeriod); // find the initial color from the averaged first 20 color readings

    // For checking accuracy of color sensor:
```

```
//printf("startPeriod is: %i", startPeriod);
//printf("\n");
//printf("startColor is: %i", startColor);
//printf("\n");

armForward(800, 550); // values for arm going into collection mode
_delay_ms(300);

QTI_L = getQTI_L(); // obtain QTI reading for left arm
QTI_R = getQTI_R(); // obtain QTI reading for right arm

if ((QTI_R == 1) & (QTI_L == 0)) // check if the right QTI is black and the left is not
{
    side = 0; // indicating on the right side
}
else if ((QTI_L == 1) & (QTI_R == 0)) // check if the left QTI is black and the right is not
{
        side = 1; // on left side
}
else // must be in center because both QTI sensors are not black
{
    side = 2; // in center
}


  // get initial value for current color for while loop

 periodL = getColorL();
 currentColorL = translateColor(periodL);

 //PHASE 1 COMMENCE

 if (side == 2) // if in the center, go to the right side first
 {
     turnRight();
     _delay_ms(1500); // TURN 90 DEGREES
     QTI_L = getQTI_L(); // initialize left QTI value (should be 0)
```

```
     while (QTI_L == 0){ // drive forward until the left QTI sees black (made it to the right side)
         goForward();
         QTI_L = getQTI_L();
         _delay_ms(10);
     }
     turnLeft();
     _delay_ms(1500); // TURN 90 DEGREES
 }

 // drive to other color side
 periodL = getColorL(); // initialize color
 currentColorL = translateColor(periodL);
 while(startColor == currentColorL) // drive until the current color is no longer the start color
 {
     phase = 1;
     periodL = getColorL();
     currentColorL = translateColor(periodL);
     goForward();
     _delay_ms(100);
 }


 if(side == 1) //left side
 {
     turnRight();
     _delay_ms(1500);//TURN 90 DEG to the right
     goForward();
     _delay_ms(2500);//move forward along divide between colors
     turnRight();
     _delay_ms(1500);//Turn 90 degrees to the right again
     stop();
 }
 else //right side or center
 {
     turnLeft();
     _delay_ms(1500);//TURN 90 DEG to the left
     goForward();
     _delay_ms(2500);//move forward along divide between colors
```

```
        turnLeft();
        _delay_ms(1500);//Turn 90 degrees to the left again
    }

    QTI_L = getQTI_L(); // get QTI values again (should both be 0)
    QTI_R = getQTI_R();

    while((QTI_L == 0) && (QTI_R == 0)) //Go until hits the black border
    {
        QTI_L = getQTI_L();
        QTI_R = getQTI_R();
        goForward();
        _delay_ms(10);
    }
    stop(); // stop at black border

   //PHASE 2: REVERSE AND RESET ARMS to deposit all blocks
       phase = 2;
       armBack(900, 300); // should be same as initial position (fully retracted)

periodL = getColorL(); // initialize current color
currentColorL = translateColor(periodL);

while(startColor == currentColorL) // drive backwards until see opponent's color
{
    goBack();
    _delay_ms(100);
    periodL = getColorL();
    currentColorL = translateColor(periodL);
}

goBack();
_delay_ms(3000); // drive back a little more
stop();// stop when get to other side


//PHASE 3: PUSHING SEQUENCE: drive forward until get to other side, drive a little further, reverse for
// a few seconds until back on opponent's side, turn right initially 30 degrees, repeat driving forward
// and turning 30 degrees until see right side, repeat and go to the other side
```

```
armForward(600, 300); // move arms to be perpendicular to the robot so can effectively push
periodL = getColorL(); // get the initial color at the beginning of the phase
currentColorL = translateColor(periodL);
border = 0; //border being 0 means that the robot should turn right first in the sweeping motion
    while (1) // infinitely loop over this code to do sweeping motion
    {
        periodL = getColorL(); // initialize current color
        currentColorL = translateColor(periodL);
        phase = 3; // change the phase to 3 so the interrupt will change border
        if (border == 0){ // check if the robot should turn right first or not
            turnRight();
            _delay_ms(250); // turn right 30 degrees
            while (currentColorL != startColor){ // drive to other side to push blocks onto original side
                periodL = getColorL();
                currentColorL = translateColor(periodL);
                goForward();
                _delay_ms(100);
            }
            goForward(); // once get to other side, drive drive forward a little more
            _delay_ms(1000);
            QTI_L = getQTI_L(); // check if on the edge
            if(QTI_L == 1) // if made it to the edge, set border to be 1 so next iteration will start turning the other way
            {
                border = 1;
                armBack(900, 300); // return arms to starting position
                armForward(600, 300); // put arms back to perpendicular to vehicle (cycle arms in case another robot bumped)

            }
            while (currentColorL == startColor){ // reverse back to opponent's side
                periodL = getColorL();
                currentColorL = translateColor(periodL);
                goBack();
                _delay_ms(10);
            }

            goBack();
            _delay_ms(3000);    // drive a little more onto opponent's side
```

```
        }
        else{ // if border is not 0, turn left initially
            turnLeft();
            _delay_ms(250); //turn left 30 degrees
            while (currentColorL != startColor){ // drive forward to other side
                periodL = getColorL();
                currentColorL = translateColor(periodL);
                goForward();
                _delay_ms(10);
            }
            goForward();
            _delay_ms(1000);

            QTI_R = getQTI_R(); // initialize right QTI sensor value
            if(QTI_R == 1) // if right QTI is on border, change border back to 0 so will start turning right again
            {
                border = 0;
                armBack(900, 300);
                armForward(600, 300);// cycle arms in case other robot bumped them
            }

            while (currentColorL == startColor){ // drive back until get to original side
                periodL = getColorL();
                currentColorL = translateColor(periodL);
                goBack();
                _delay_ms(100);
            }

            goBack();
            _delay_ms(3000); // drive back a little more
        }
        }
}

ISR(PCINT0_vect) // pin change interrupt for all 3 QTI sensors


{
    if (PINB & 0b00000001) // check if it is the rear (center) QTI
    {
        turnRight();
        _delay_ms(101); // turn right a little so doesn't get stuck
        goForward();
        _delay_ms(1000); // drive forward so doesn't reverse off of arena
    }
    if (PINB & 0b00100000) // check if it was the right QTI on black
    {
        if (phase == 3) // check if in the 3rd phase, if it is then change the border
                        // to be 1 so turns left in between pushing
        {
            border = 1;
        }
    }
    if (PINB & 0b00010000) // check if it was the left QTI on black
    {
        if (phase == 3) // if in the 3rd phase, change border so turns right in between pushing
        {
            border = 0;
        }
    }
}
```