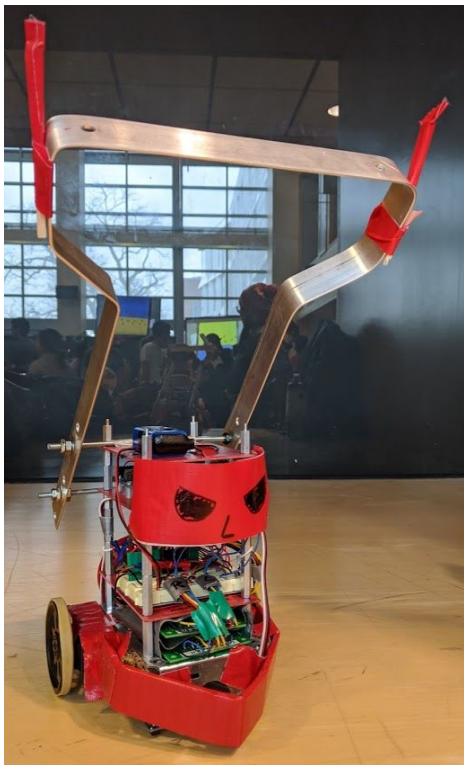
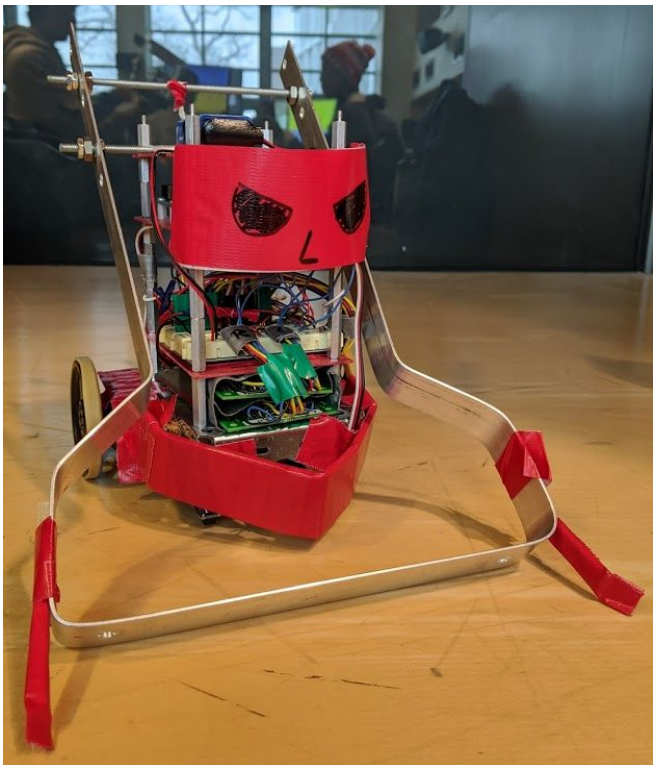
MAE 3780 Final Report

Team 11:
Sophia Kane, Daniel Morton, Ananth Palaniappan

“The Big Red Devil”

12/15/2019





Introduction:

The objective of this report is to explain the motivation behind the design of our robot for the 2019 Cube Craze challenge and analyze its performance in the challenge. For this year's competition, robots were scored based on how many cubes were on their starting side at the end of the 60 second competition. Our strategy was to win using a simple cube collection method -- a gate that collects cubes from the center of the board -- and a defense that eliminates the opposition's collected cubes. Our strategy is described in full in later sections. Through this report, we show that we supported our strategy by choosing a mechanical design that is stable, lightweight and durable, a computer program that ensures robust sensors and fast movement, and electronics that maximize speed. Our strategy was strong in theory but our implementation had problems due to limited testing of our code and a malfunctioning winch.

Motivation Behind Design:

Our design was primarily motivated by the strategy we believed to be most successful, requiring a large number of cubes collected in the first few seconds of the competition and a defensive strategy for later on. To accomplish this, we created a gate mechanism that would be lowered once our robot reached the center of the board, enclosing a large number of blocks, then backing up to deposit the collected cubes on the starting side. After repeating this a few times, the robot would then go into defensive mode during the second half of the round, referred to as "pushMode" in our code. The objective behind "pushMode" is to push any cubes on the opponent's side *off* of the playing board. This strategy, when implemented correctly, would likely secure a victory, as cubes off the the playing board generally do not count towards the final scoring (unless there is a tie).

For stability and to ensure a low center of gravity, we mounted the battery pack -- one of the heaviest components -- on the bottom of our robot. Along with this, we used standoffs to extend the servo motors from the chassis to make the base of the robot as wide as possible. Our robot was relatively tall, so the low center of gravity and wide base ensured that our robot was less likely to tip due to a collision with another robot. Along with this, to ensure the motor was able to support the gate in its raised position, we chose aluminium, which is very lightweight, to be the material of our gate, and a pivot position that would reduce the force on the motor and allow it to fit in the size limits.

Three BJT H-bridges (See Figure 1, right, for schematic) were constructed for this project, with each soldered to the proto-boards to ensure no components would come loose during testing and the competition, and that the robot was as compact as possible. Each input/output cable from the H-bridges was color-coded such that red = Vcc; black = ground; yellow = connections to motors; blue = P0 and P1 to the Arduino pins. This color-coding made it very simple to look for potential errors in the wiring, and each wire leading to/from the Arduino (including the P0/P1 wires for each H-bridge and the sensor outputs) was labeled with the pin number it connected to, in case anything came disconnected.

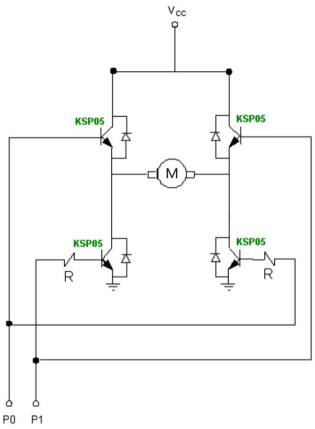


Figure 1: BJT H-Bridge

Description of tested alternate solutions:

An additional H-bridge was constructed using FQP30N06L N-channel MOSFETs, using a slight modification to the standard H-bridge schematic. The order of the pins (Gate, Drain, Source) were not in the same order as on the BJT (Collector, Base, Emitter), so the circuit had to be rewired such that the drain replaced the collector, the source replaced the emitter, and the gate replaced the base. An additional pull-up resistor was connected between the gate and source as per online robotics forum recommendations[1], and since the FQP30N06L included an internal flyback diode, an extra diode was not included. This H-bridge was built with the hopes that it would be more efficient in controlling the power to the motors - to be able to respond quicker to inputs and turn the motors faster when connected to the 9V as compared with the BJT H-bridge. However, upon testing it was found that the FQP30N06L H-bridge turned the wheels an average of 17% slower than the BJT H-bridge, and additional types of MOSFETs were tested but none led to a higher speed. Thus, this was not included in the final design.

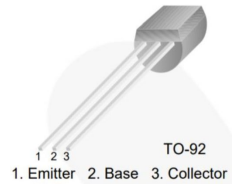


Figure 2: TO-92 BJT

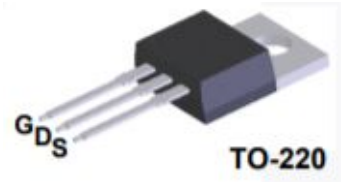


Figure 3: FQP30N06L MOSFET

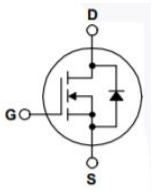
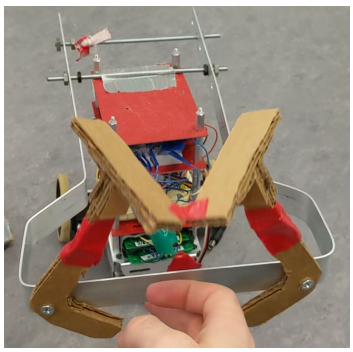
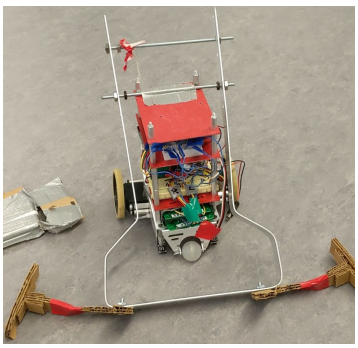


Figure 4: FQP30N06L Schematic





Figures 5 (left) and **6** (right): Gate with raised extensions (to fit within 8”) and lowered

The original design for the robot relied on extensions to the main gate to increase the block-grabbing capabilities by doubling the gate’s width from 8 to 16 inches, using 4-inch extensions on either side. These extensions were connected via a pinned hinge on the front of the gate, and were designed in an L-shape such that the force of the gate hitting the ground would flip them outwards. This mechanism worked perfectly during manual testing, but under the control of the rear winch, it was not able to hold the additional weight and would turn backwards due to the increased tension in the cable. The intention was to counterweight the rear end of the lever to offset the additional weight of these extensions, but with limited time before the competition, high-density weights that also maintained the size constraint on the robot could not be procured and thus these extensions were left off the final design.

---

1 https://electronics.stackexchange.com/questions/132458/mosfet-based-h-bridge-circuit?fbclid=IwAR3K2-u2UOS5_vaWVhMPbebnwV1rjA813k2ds5neg6ft61kW8wqZeQM_0Og

Flowchart of Strategy (and a description of the code behind the strategy):

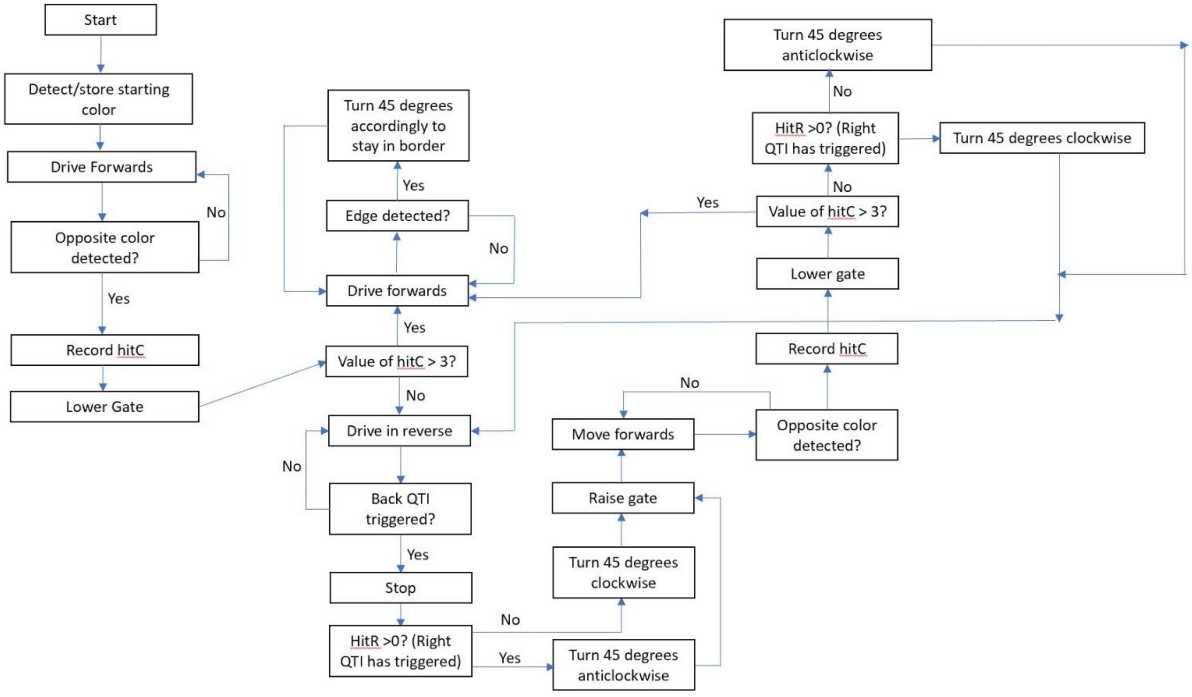


Figure 7: Strategy flowchart

For Milestone 4, the robot was programmed to move along the starting side of the board, grab blocks from the centerline, move them to the back, and go to other areas of the centerline to get the rest. While this worked well, it was highly dependent on the orientation of the robot relative to the rear black border, and did not consider interactions with the other robots. If the opponent got many blocks to their side quickly, our bot would not be effective for the remainder of the match. A collision with the other robot that reoriented our bot would completely mess up the strategy as well.

So while this remained the primary behavior of the robot, we also made multiple changes to account for these possible interactions with the other robot. These changes include:

- Depositing the blocks on our side slightly prior to hitting the back edge, so that the robot avoids pushing them back when turning and moving forwards again
- Correcting the turning behavior when moving backwards (see description of "fwd" variable)
- Implementing pushMode, a defensive state which triggers after collecting blocks 3 times, lowering the gate and pushing all blocks off the opponent's side
- Recording sensor trigger states (see description of "hit_" variables)

The following functions and variables were critical in executing our strategy throughout the competition:

my_delay_ms()

Allowed for variable delays in turnAngle() since _delay_ms() cannot accept variables as inputs.

turnAngle(int angle)

Allowed for simple control of the turning angle, using a predefined timeConstant which allows for a delay based on the desired angle. This allowed for inputs from -180 to 180, with orientations defined like a compass (forward = North, 0°). Note that ∓30, 45, and 90 degree turns were mainly used.

grabAndGo()

The basis for the first state of the competition, this function triggered whenever the color sensor triggered a change, then checked the state variables (see below section) to determine when to lower the gate, which direction to turn, and then move to the rear side of our board.

gateStatus

This variable was set to 0 or 1 based on the current position of the gate, so that a command to raise or lower the gate would only trigger if the current state was the opposite of the command. Without this, the rear motor could lower the gate while it was already low, introducing too much slack into the cable and rendering the gate unusable for the remainder of the match.

fwd

This variable was set to 1 if moving forward, and -1 if moving backward, which allowed us to easily switch the angle (using turnAngle(fwd*angle)) for turning when moving backwards, since a right QTI triggered when moving backward should cause a *right* turn rather than left.

hitC; hitR; hitL; hitB

These variables allowed us to record the number of times each sensor triggered, to change the behavior of the robot based on the relative position on the board and status within the match.

We also noticed that the sensors would occasionally record the wrong state, so the following operations were implemented which dramatically improved their reliability:

Averaged initial color sensor reading

At the beginning of the main() function, the robot takes 3 readings of the initial color period and averages them together. We occasionally noted that the sensor would register the wrong firstColorPeriod when taking just 1 measurement, which caused the robot to start spinning in place when it believed it was on the wrong side of the board. Implementing this average fixed this issue.

2 ms pause after PCINT0 (QTIs) triggers

To prevent QTI false positives (which were observed to occur roughly once every two minutes of testing), a 2 ms delay was inserted before any functions were executed to make sure the QTI was still triggered. Otherwise, the robot would continue as before.

Additionally, early in the process we implemented PWM to control our two main motors, with the intention that we could use this to easily adjust the speed and turning radius of the robot. Though we ended up using the fastest speed possible and the tightest turning radius, PWM was an effective way to control the amount of power sent to each motor, and we could use relatively simple functions to control the movement without needing to worry about shorting the H-bridges (a result of the deadTime between the output compares). For example, goForward(int speed) will drive the robot at the given input speed between -100 and 100, and translate that to a setPoint (between 0 and 255) for the OCR registers.

Analysis of Strengths and Weaknesses, and Competition Performance:

Strengths, what worked well:

The main strength of the Big Red Devil was the main gate - constructing it out of the 1/16" aluminum sheet allowed for a very flexible design which was very easy to bend into shape and drill through. Multiple holes were drilled so that we could narrow down which positions of the hinge and string connection maximized torque on the gate while minimizing the reverse torque on the winch and remaining in the 8" size constraint. Additional holes in the front provided flexibility to include attachments to the front of the gate (Figures 5 and 6). Additionally, as compared with other robots implementing a similar gate strategy, this selection of material was significantly more lightweight and durable, which was critically important as there was limited time to make fixes in the competition. Additionally, the weight and low CG of the design meant we could push most robots around the board when we ran into them.

The strategy, if implemented correctly, would also have been a strength of the design. Very few other teams in the competition had thought of pushing off the other team's blocks, and many left their blocks completely undefended. Additionally, implementing the defenses against interactions with other robots after the Milestone 4 code dramatically improved the odds of success in the tournament, as compared to without these.

Additionally, the three QTI sensors (left, right and back) prevented the robot from falling off the edges of the arena, and these worked excellently throughout the competition. The only time that our robot fell off the side was when another robot (which was even heavier than our bot) pushed us off.

Weaknesses, what didn't work well:

This design was certainly not without its flaws though, as indicated by the last-place finish within our round-robin group. We were expecting the robot to perform much better, and the primary issues can be chalked up to bugs in the most recently uploaded code, and non-ideal performance of the rear winch.

During the pre-competition testing, the robot was working very well, but this was using the old strategy which did not account for interactions with the other robot. On the night before the competition, a new version of the code was written and extensively tested/modified which implemented the final strategy, but there were still bugs that remained and were not fixed before check-in. There were two primary bugs which were identified after the competition: first, that the pushMode was enabled too early, and second, that the direction to turn at the border when starting on the left side of the board was not properly defined. For the pushMode error, the original while loop is supposed to break when (hitC <= 3) == FALSE, and other sections of the code defining the QTI and color sensor behavior had different sections of code to run when pushMode is enabled. However, the color sensor code ran the pushMode version when (hitC >= 3), when this should have been (hitC > 3), leading to the robot dropping the gate early and initiating pushMode on both sides of the board (instead of just the opponent's side). For the other bug in the code, starting on the left side of the board should have made the robot turn right at the border, but it instead turned left and sometimes forced itself into the edge of the board. This was due to a leftover (-) sign in the turnAngle() code from debugging the backwards QTI turning motion, which double-compensated and led to the opposite turn behavior.

The winch which raised and lowered the gate was flawed in that each subsequent raise was not as efficient as the previous - it would raise to a slightly lower position than it had before. Therefore, after a

few repetitions of the raiseGate function, the gate would not be high enough to move over the blocks and would just push them, even if pushMode was not active. This was primarily due to the motor slipping under the adverse torque - suggestions for improving this are included in the next section.

Interactions with other robots should also have been further considered, especially at the border. In one round-robin, the other robot pushed ours across the border before expected, adding a value to hitC and furthering our issues with the gate dropping into pushMode too early.

Suggestions of how to fix performance issues:

As mentioned in the previous section, the winch motor in the back was not able to generate enough torque to efficiently raise and lower the gate throughout the competition. This was due to the fact that the two-wire servo motor utilized had neither enough torque nor the ability to recall its position. Thus, this issue could have been avoided by utilizing a three-wire servo with position control, or a potentially better option here would be a stepper motor. Both of these options would be improved by purchasing a motor that can apply a greater torque.

Along with this, our robot had the winch oriented vertically, but it was pulling the string connected to the gate at a slight angle, which led to a risk of the string slipping out of place along the winch and losing its tension (this happened in one of the competition round-robins). This could be fixed by orienting the rear winch at a slight angle so it is better aligned with the string connected to the gate, but since this would be extremely difficult using the provided chassis, a custom chassis would need to be constructed to create an angled platform for this motor.

Additionally, the diameter of the winch could have been reduced - this would make the gate motion slower since more revolutions would be required to pull the gate up and down, but would decrease the adverse torque on the motor from holding the gate in the raised position. This would make it more consistent in holding its position and reduce the chances of slipping.

The bugs in the code as mentioned have already been fixed and the updated and commented code is included in the Appendix. Improving interactions with other bots would require a fundamental rethinking of the strategy, but one quick fix for the color sensor border issue mentioned earlier would be to disable the color sensor from initiating grabAndGo() temporarily, so that it doesn't trigger multiple times in a short period of time.

Conclusion:

While the performance of the robot in the competition was not optimal, this strategy and overall mechanical/electrical design would work well in future competitions, if the bugs were worked out in time and improvements were made as noted.

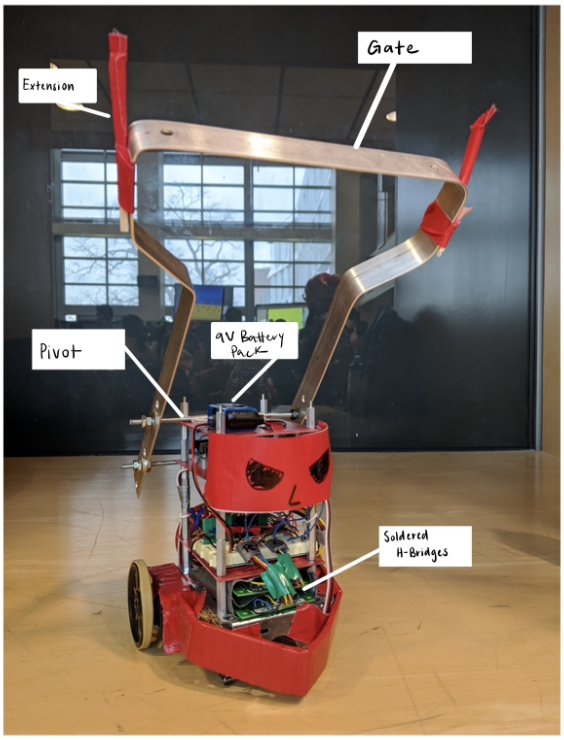
If there had been more time to build and test the robot, there are a few components we would have liked to add. A voltage amplifier would be able to increase the voltage powering the motors, meaning the robot will be able to react faster during the first part of the competition and reach the blocks earlier than the other robots. Additionally, connecting a compass to the analog ports of the Arduino would have been an excellent way to determine the current orientation on the board, especially if pushed off course by the other robot.

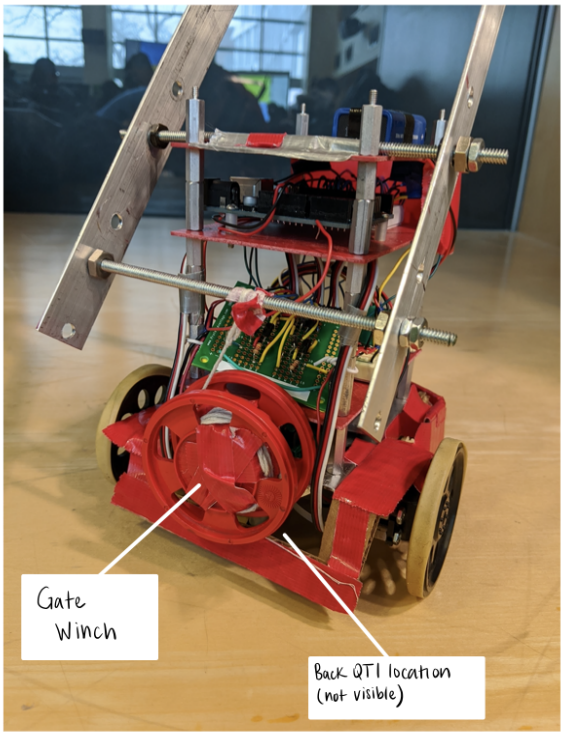
We can all certainly say that despite all the difficulties we encountered throughout the project, every single bug that we worked out was an incredible learning experience, and we are proud of what we produced even if it had its issues.

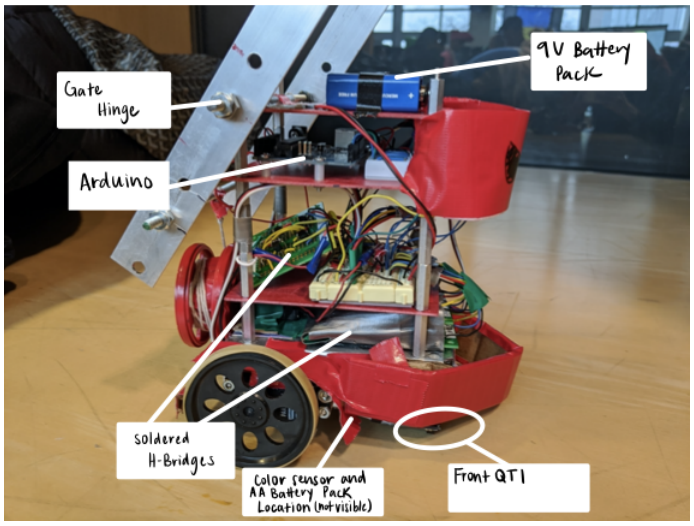
Appendix:

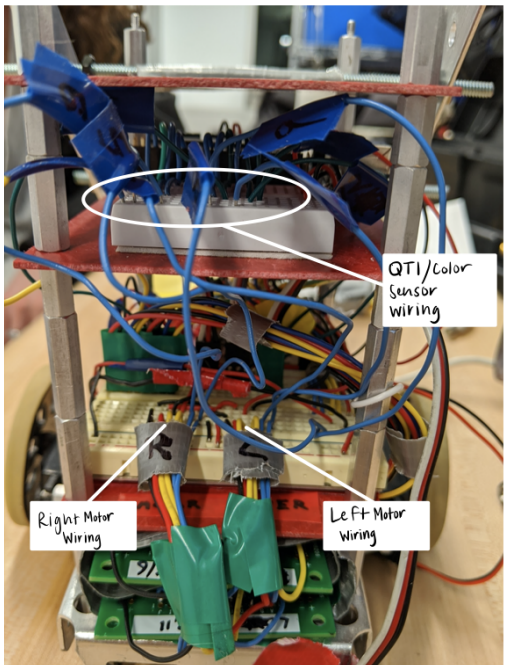
Table 1: Budget - Components not provided in lab			
Component	Unit Cost ($)	Quantity	Total Cost ($)
Nuts	0.05	8	0.40
Threaded rod	2.98	2	5.96
Aluminum Sheet	5.78	1	5.78
Extra Plastic Wheels	3.00	2	6.00
Velcro	1.5/yard	0.06 yards	0.09
Extra 9V battery for testing	3.12	1	3.12
Extra AA batteries for testing	0.8	4	3.20
Zip ties	0.03	2	0.09
MOSFETs	1.08	4	4.32
			28.96

Labeled Schematics:









Final code, modified to fix the bugs encountered in the competition:

```
//   ________              ____  _           ____           __   ____              _ __
//  /_  __/ /_  ___      / __ )(_)___ _    / __ \___  ____/ /  / __ \___ _    __(_) /
//   / / / __ \/ _ \    / __  / / __ `/   / /_/ / _ \/ __  /  / / / / _ \ | / / / /
//  / / / / / /  __/   / /_/ / / /_/ /   / _, _/  __/ /_/ /  / /_/ /  __/ |/ / / /
// /_/ /_/ /_/\___/   /_____/_/\__, /   /_/ |_|\___/\__,_/   /_____/\___/|___/_/_/
//                            /____/

// By MAE 3780 Bin 11 - Daniel Morton, Sophie Kane, Ananth Palaniappan

#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include "serial.h"
#include <stdlib.h>

/* ******* H-bridge wire color descriptions ***********************
Red: Power
Black: Ground
Yellow: Motor
Blue: Arduino pins
************************************************************** */

/* ******* Pin Descriptions  ***********************************
Right motor:          pin 6 - OC0A
                      pin 5 - OC0B

Left motor:           pin 11 - OC2A
                      pin 3 - OC2B

Rear motor:           pins 8 and 9

Front Left QTI:       pin 12

Front Right QTI:      pin 10

Rear QTI:             pin 13

Color Sensor:         pin 4

************************************************************** */

// Variables ////////////////////////////////////////////////////////////////////////

int deadTime = 10;    int setPoint0 = 0;    int setPoint2 = 0;                          // for
PWM and speed control
int speedR;           int speedL;           int timeConstant;     int turnTime;         // for
turn function
int period;           int threshold = 150;  int period_us;                              // for
color sensor
int firstColor;       int firstColorPeriod;   int blue = 1;       int yellow = 0;       // for
color sensor
int raiseTime = 700;  int lowerTime1 = 1000;  int lowerTime2 = 700;             // times in
milliseconds, for raising/lowering gate
                                                                                    //  ^
note lowerTime2 should equal raiseTime
int gatevar = 0;                                                                    //
define if in initial state or not
int hitR = 0;         int hitL = 0;         int hitB = 0;         // record number of times the
QTIs trigger NEW
int hitC = 0;  // record number of times the color sensor triggers??????????    NEW
int localTest = 0;
int fwd = 0;
```

```
int gateStatus = 1;
int hitBStore;  int store = 0;
int pushMode = 0;

// FUNCTIONS ////////////////////////////////////////////////////////////////////////

void setOCRRegisters(int setPoint0Temp, int setPoint2Temp, int deadTimeTemp)
{
	// this function sets the PWM compare values based on setpoint and deadtime
	OCR0A = setPoint0 - deadTime;
	OCR0B = setPoint0 + deadTime;
	OCR2A = setPoint2 - deadTime;
	OCR2B = setPoint2 + deadTime;
}

void my_delay_ms(int n) {  // the _delay_ms function does not allow for variable inputs, this
fixes that problem
	while(n--) {
		_delay_ms(1); // delay up to the input time value n
	}
}

void goForward(int speed) // speed is an integer between 0 and 100
{
	// define where the PWM interrupt locations happen through setpoint
	// 0 fwd motion at setpoint = middle = 127, max fwd movement at 255-deadtime

	setPoint0 = 127 + speed * (128-deadTime)/100;
	setPoint2 = 127 + speed * (128-deadTime)/100;
	setOCRRegisters(setPoint0, setPoint2, deadTime);
	fwd = 1; // record state of going forward
}

void goBackward(int speed)
{
	setPoint0 = 129 - speed * (128-deadTime)/100; // 129 here instead of 128 because this
goes down to 0+deadtime
	setPoint2 = 129 - speed * (128-deadTime)/100; //   with buffer room for integer
division in C rounding
	setOCRRegisters(setPoint0, setPoint2, deadTime);
	fwd = -1;  // record state of going backward
}

void stop(void)
{
	setPoint0 = 128; // middle of timing region so duty cycles are equal
	setPoint2 = 128;
	setOCRRegisters(setPoint0, setPoint2, deadTime);
}

void turnAngle(int angle)
{
	// angle defined like a compass, from the forward position a positive angle is a right
turn, negative for left
	// speed is defined as -100 to 100
	// angle is defined as -180 to 180

	if ( fwd*angle >0 ) // for a right turn
	{
		speedR = -99; speedL = 100;  // define motor speeds -  R (right) and L (left)
		setPoint0 = 127 + speedR * (128-deadTime)/100;  // Right motor: adjust PWM
setpoint based on input speed
		setPoint2 = 127 + speedL * (128-deadTime)/100;  // Left motor: adjust PWM
setpoint based on input speed
		setOCRRegisters(setPoint0, setPoint2, deadTime);  // send to function
	}
```

```
        if ( fwd*angle < 0 ) // for a left turn
        {
                speedR = 100 ; speedL = -99; // define motor speeds -  R (right) and L (left)
                setPoint0 = 127 + speedR * (128-deadTime)/100;  // Right motor: adjust PWM
setpoint based on input speed
                setPoint2 = 127 + speedL * (128-deadTime)/100;  // Left motor: adjust PWM
setpoint based on input speed
                setOCRRegisters(setPoint0, setPoint2, deadTime);  // send to function
        }

        // turn for a certain amount of time
        timeConstant = 10; // adjust based on experimentation
        turnTime = abs(angle)*timeConstant; // time is a function of angle - in ms
        my_delay_ms(turnTime);  // using this function so we can input a variable
}

void raiseGate(void)
{
        // pin 8 high will raise thee gate, pin 9 high will lower the gate
        if (gateStatus == 0){
                PORTB &= 0b11111110; // set pin 8 low before setting pin 9 high!
                PORTB |= 0b00000010; // set pin 9 high
                my_delay_ms(raiseTime); // allow for time for the spool to wind/unwind

                PORTB &= 0b11111101; // set pin 9 low to stop spinning the motor
                gateStatus = 1; // define current state of the gate
        }
}

void lowerGate1(void)
{
        PORTB &= 0b11111101; // set pin 9 low before setting pin 8 high!
        PORTB |= 0b00000001; // set pin 8 high
        my_delay_ms(lowerTime1); // allow for time for the spool to wind/unwind

        PORTB &= 0b11111110; // set pin 8 low to stop spinning the motor
        gateStatus = 0; // define current state of the gate
}

void lowerGate2(void)
{
        if (gateStatus == 1){
                PORTB &= 0b11111101; //set pin 9 low before setting pin 8 high
                PORTB |= 0b00000001; //set pin 8 high
                my_delay_ms(lowerTime2); //allow time for the spool to wind/unwind, less time
than lowerTime1

                PORTB &= 0b11111110; // set pin 8 low to stop spinning the motor

                gateStatus = 0; // define current state of the gate
        }
}

ISR(PCINT0_vect) // pin change interrupt: currently being used for QTI sensor
{
        cli(); // make sure this isn't interrupted

        if ( hitC <= 3)  // 0
        {
                _delay_ms(2); // check false positive

                // check conditions where multiple sensors trigger first, then check conditions
with only one triggered
```

```
		if ((PINB & 0b00000100) & (PINB & 0b00010000)){ // if both front sensors are
high
			lowerGate2();
			goBackward(100);
		}

		else if (PINB & 0b00100000)// if just the rear sensor triggers (PIN13)
		{
			// strategy: check which way to turn depending on what side has been hit

		// switched from hitR to hitL
			if ( hitR == 0 ){  // if the right side has NOT been hit already,
				turnAngle(45);  // keep going towards the right side
			}
			else {  // if the right side has already been hit
				turnAngle(-45); // keep going to the left side
			}

			goForward(100); // go forward if hits the back edge
			raiseGate();
			hitB = hitB+1; // record the hit
		}
		else if (PINB & 0b00000100) {  //if right QTI is triggered (PIN10)
			turnAngle(-90); // turn left
			goForward(100);
			hitR = hitR+1; // record the hit
		}
		else if (PINB & 0b00010000){  // if the left QTI is triggered (PIN12)
			turnAngle(90); // turn right
			goForward(100);
			hitL = hitL +1; // record the hit    NEW
		}

	} // END IF HITC

	else // hitC > 3 so pushmode is on
	{
		_delay_ms(10); // slightly larger delay to make sure if both trigger, this is
registered

		if ((PINB & 0b00000100) & (PINB & 0b00010000)){ // if both front sensors are
high
			//            printf("both");
			//lowerGate2();
			goBackward(100);
		}

		else if (PINB & 0b00100000)// if just the rear sensor triggers (PIN13)
		{

			turnAngle(20);  // move around board slightly
			goForward(100); // go forward if hits the back edge
			//raiseGate();
			hitB = hitB+1; // record the hit     NEW
		}
		else if (PINB & 0b00000100) {  //if right QTI is triggered (PIN10)
			turnAngle(-90); // turn left
			goForward(100);
			hitR = hitR+1; // record the hit     NEW
		}
		else if (PINB & 0b00010000){  // if the left QTI is triggered (PIN12)
			turnAngle(90); // turn right
			goForward(100);
			hitL = hitL +1; // record the hit    NEW
		}
```

```
        }

        sei();
}

ISR( PCINT2_vect )   // for color sensor - note pin change ISR triggered by any logical change
{
        if(PIND & 0b00010000) { // if pin 4 is high
                TCNT1 = 0b00000000; // reset timer
        }
        else{
                period = TCNT1; // NOT IN UNITS OF TIME YET!
        }
}

void initColor()   //setup code that will run once
{
        TCCR1A = 0b00000000; // normal mode -- see datasheet pg 170
        TCCR1B =  0b00000001; // prescaler of 1
}

int getColor()  // return period in units of time
{
        PCMSK2 = 0b00010000; // pin change interrupt on PCINT20 (pin4)
        _delay_ms(5); // 5 ms delay to allow for interrupt to trigger
        PCMSK2 &= 0b11101111;  // turn off interrupts on pin 4 for now

        period_us = 2*period*(.0625); // microseconds
        // 1/16 factor to account for system clock period
        // 2 factor to account for TCNT only measuring half a period

        return(period_us);
}

void PWMSetup(void)   // setup all registers to enable PWM and timer interrupts
{
                deadTime = 10; // small number of clock ticks for dead time
                setPoint0 = 245; // INITIALIZE, start off at max speed
                setPoint2 = 245; // 1 setpoint per timer

                // TIMER 0: 8-bit
                TCCR0A = 0b10110001; // see datasheet
                TCCR0B = 0b00000001; // prescaler of 1, may want to change to 8??
                OCR0A = setPoint0 - deadTime;
                OCR0B = setPoint0 + deadTime;

                //TIMER 2: 8-bit
                TCCR2A = 0b10110001; // same values as with timer0
                TCCR2B = 0b00000001; // prescaler of 1
                OCR2A = setPoint2 - deadTime;
                OCR2B = setPoint2 + deadTime;
}

void pinSetup(void)  // define functions of all pins - DDRD and DDRB
{
        DDRD = 0b01101000; //set pins 3 , 5 and 6 as outputs, 4 as input
        DDRB = 0b00001011; // set pin 8,9,11 output; 8, 12, 13 inputs
}

void pinChangeSetup(void)
{
        PCMSK0 = 0b00110100; // pins 10, 12, 13 for each qti sensor //
        PCICR = 0b00000101; // enable pcmsk0/2
}

void grabAndGo(void)
```

```
{
	cli();  // make sure not interrupted

	stop();

	if ( hitC == 0 )   {  // if hitting the other side for the first time,
		lowerGate1();  // fully lower the gate from the upright position
		_delay_ms(20);  // buffer time, seems to help the functions operate
		goBackward(100);  // go straight back if hitting the other color 1st time

	}
	else   {
		lowerGate2();  // lower the gate only from the slightly raised position
		_delay_ms(20);  // buffer time, seems to help the functions operate
		if (hitR == 0){   // if it has not hit the right side, it will approach the
other color at +45 degrees
			turnAngle(-45);  // turn to go straight back
			goBackward(100);
		}
		else{          // if it has hit the right side, it will approach the other color
at -45 degrees
			turnAngle(45);  // turn to go straight back
			goBackward(100);
		}
	}
	sei();
	hitC = hitC + 1; // record the hit so it doesn't lower the gate too far next time

	_delay_ms(2500);  // NEW:  raise gate after a set time to deposit on own side
	raiseGate();
}


// MAIN CODE
////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
int main(void)
{
	PWMSetup(); // setup PWM and corresponding registers
    pinSetup(); // setup pins I/O
	init_uart(); // turn on serial monitor
	pinChangeSetup();            // note: pcmsk2 is set up in getColor() function
	initColor();

	sei(); //enable interrupts globally

	// take the average over multiple values of the initial color sensor reading - to make
sure it's accurate
		int i = 0;    int colorVal = 0;   int colorSum = 0; int numVals = 3;
		while (i<numVals)  {
			colorVal = getColor();
			colorSum = colorSum+colorVal;
			i = i+1;
		}
		firstColorPeriod = colorSum/numVals; // figure out which color the robot starts
on
	// instead of using one measurement with firstColorPeriod = getColor();

	if (firstColorPeriod > threshold) // blue
	{
		firstColor = blue; // blue = 1
	}
	else
	{
		firstColor = yellow; // 0 = yellow
	}
```

```
    goForward(100); // start off forwards


while (pushMode == 0)
{
           if ( hitC <= 3) // 0, 1, or 2
           {
                  period_us = getColor(); // microseconds

                  if (period_us > threshold) // on blue
                  {
                         if(firstColor == yellow) // if started on yellow, (yellow = 0)
                         {
                         grabAndGo();
                         }

                  }
                  else // on yellow
                  {
                         if(firstColor == blue) // if started on blue (blue = 1)
                         {
                         grabAndGo();
                         }
                  }
           }
           else // there have been 3 or greater crosses over the color border
           {
                  lowerGate2();

                  if (period_us > threshold) // on blue
                  {
                         if(firstColor == yellow) // if started on yellow, (yellow = 0)
                         {
                         pushMode = 1; // activate pushMode when on opponent's side
                         }
                  }
                  else // on yellow
                  {
                         if(firstColor == blue) // if started on blue (blue = 1)
                         {
                         pushMode = 1; // activate pushMode when on opponent's side
                         }
                  }
           }

    } // end while pushmode == 0 loop

    while(pushMode == 1)
    {
           goForward(100); // push things forward off the board on the opponent's side
           lowerGate2();

           // code to make sure it stays on the opponent's side

           period_us = getColor(); // microseconds

           if (period_us > threshold) // on blue
           {
                  if(firstColor == blue) // if started on blue
                  {
                         raiseGate();
                         turnAngle(180); // raise gate and turn around
                  }
           }
           else // on yellow
```

```
                {
                        if(firstColor == yellow) // if started on yellow
                        {
                                raiseGate();
                                turnAngle(180); // raise gate and turn around
                        }
                }

        }
}
```