RBE 1001 C ’18, *Introduction to Robotics*

Final Project Report

“Cthulhu”

Team 13

Member:	Signature:	Contribution (%):
Samantha Grillo	Samantha Grillo	33.3
Teresa Saddler	Teresa Saddler	33.3
Tobias Schaeffer	Tobias Schaeffer	33.3

Grading:		
	Presentation	_____/20
	Design Analysis	_____/30
	Programming	_____/30
	Accomplishment	_____/20
	Total	_____/100

Table of Contents

Table of Figures

1 Introduction

Robots can be ideal tools for missions, as they do not face many of the limitations that humans do, but they often require systems that can navigate through complex environments and can interact with objects in these environments. The challenge requires a robot to be able to manipulate spheres of various sizes, called ORBs, ASTEROIDs, and STARs while maneuvering around a set field (Figure 1.1). These spheres must then be deposited into tubes with three levels of heights, called ORBITS, or placed into a zone at the edge of the field, called the BASE. An additional challenge is climbing and mounting a platform called the BLACK HOLE. The challenge includes a fully autonomous period, a teleoperational period, and a 30 second END GAME, in which the robot can score the large STAR or mount the BLACK HOLE.

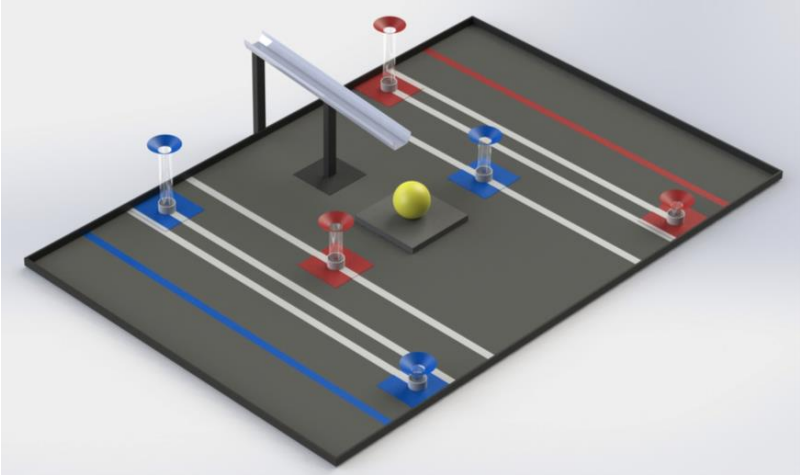

Figure 1.1: Field Diagram

The challenge can be broken down into three main parts:

1) The robot will need to move autonomously and through radio controls,
2) The robot will need a way of collecting objects, and
3) The robot will need to transport and deliver these objects.

In order to design a robot to fulfill these tasks, prioritization of the many subtasks is necessary, such that the optimal balance of capabilities can be found.

2 Preliminary Discussion

To decide what to focus on in the challenge, we looked at the point distribution and level of difficulty of each task through both the autonomous and teleoperational periods of the game and weighed our options accordingly. In our initial analysis, we ruled out a couple of scoring options; namely, scoring the STAR, and scoring in the 18" ORBITS. Our reasoning for ruling out the level three ORBITS was that in order for a mechanism to reach this high, the center of gravity of the robot would be raised, decreasing the stability of the robot, which in this case was a sacrifice we were not willing to make. As for the STAR, it was immediately clear to us that the dimensions of this object made scoring it highly difficult, especially considering the starting size constraints of the robot. With this consideration, we decided that it was not worth the increased complexity level and possible other functionality sacrifice it would take to enable the robot to score this.

With these scoring techniques eliminated, we decided our primary focus should be on scoring ORBs into the 6" and 12" ORBITS, as this would allow the robot to score points in both the autonomous and teleoperational periods. Additionally, we speculated that it would be easy to score the ORBs in the BASE using a mechanism primarily designed to score them into the ORBITS, so we decided to make this as a secondary goal. Similarly, we decided that adding the capability to collect and score ASTEROIDs to a robot that could already collect and score ORBs would be trivial, so this was added to our design goals. As we studied the END GAME, we reasoned that as long as our robot could maintain stability, it would not pose much of a challenge to climb the BLACK HOLE, so we decided to add this as another secondary goal.

We decided that it would be prudent to have multiple autonomous strategies, in order to increase the adaptability of the robot. Since we had determined the primary purpose of our robot

is to deposit ORBs into the 12" ORBITS, we decided that our primary autonomous mode will complete this task, but that we should have additional autonomous modes to score ORBs into the level one ORBITS and to score ORBs into the BASE.

In the teleoperational period, we decided that our focus will be scoring ORBs and ASTEROIDs into the level two ORBITS, but we will also consider scoring them into the level one ORBITS if the game state was such that this is more advantageous. In the END GAME period, we will cease scoring ORBs and ASTEROIDs, and attempt to climb the BLACK HOLE.

3 Problem Statement

Our robot needs to be able to efficiently move and collect and deposit objects in both autonomous and teleoperational periods.

High Priority—For the robot to be successful, it must:

- Maintain a stable 15.25" x 15.25"x 18" starting position
- Weigh less than 10.0-lbs
- Have a mechanism that can score ORBs into the level one and two ORBITS
- Use a non-trivial power transmission
- Have sensors that can facilitate accurate autonomous operation
- Use a custom-built electronic circuit

Medium Priority—While these goals may be dismissed in order to fulfill one of the tasks above, the robot should:

- Be able to climb the BLACK HOLE
- Score ORBS and ASTEROIDS in the BASE
- Maintain stability
- Use sensors to drive straight and turn accurately

Low Priority—If given additional time and resources, the robot would:

- Score the STAR
- Have a mechanism that could score ORBs into the level three ORBITS

4 Preliminary Designs

As we began considering the possible designs of our robot, we decided to begin by focusing on the drivetrain and whether our wheelbase or our wheel track should be wider. We considered how a proportionately wider wheelbase would increase the turning ability of the robot while decreasing its ability to drive straight, and how a proportionately wider wheel track would do the opposite. We decided that due to the demands of the challenge, neither should be favored, and that ideally our wheelbase and wheel track would be similar in dimension. To make driving as easy as possible, we decided to have two driven rear wheels, and two undriven front wheels. Given our desire to climb the BLACK HOLE, we determined that the robot required the 4" wheels at a minimum so the robot would be able to make it up the ledge; however, we decided that we should remove the tires from the front wheels in order to allow them to slide more to make driving easier. We chose to support the axles on either end by the chassis, in order to reduce the stress on the axles and increase the durability of the drivetrain.

For our lifting mechanism we discussed using a rack and pinion, a claw, and a four-bar-linkage with a scoop. Using a rack and pinion for the lifting mechanism would involve a pinion driving a rack with a scoop attached, such that the scoop could be positioned where the bottom was 0" from the floor, 6" from the floor, and 12" from the floor. As we thought through this, we realized that the nature of the rack was not ideal for this mechanism, unless we were to have the pinion near the top of the robot, which would decrease the stability of the robot. Due to this limitation, we decided to discard this design idea.

If the claw were to be used as a lifting mechanism, it would need to be able to open and close in order “pinch” ORBs and ASTEROIDs, and the arm would need to be able to move up and down, such that ORBs and ASTEROIDs could be picked up, then deposited in the level one and two ORBITS. This would require two different systems for the motion: one driving the arm, and another opening and closing the claw. Additionally, in order to grab both ORBs and ASTEROIDs, the claw would need to be able to hold spheres of different sizes, which poses a challenge. Due to the complexity of the claw design, we decided that this would not be ideal, and that a simpler way to do the same task would make more sense.

The most practical of the options was to use a four-bar-linkage with a scoop. Similar to the rack and pinion, this would require that the scoop be able to be positioned such that the bottom was 0" from the floor, 6" from the floor, and 12" from the floor; however, unlike the claw, this design involves a crank and a follower bar, attached in the lower-middle or back of the robot, being driven from the crank. This would also make the scoop being at different angles at different levels possible (Figure 4.1). As this lacked the drawbacks presented by the rack and pinion, while giving additional advantages, and was much simpler than the claw design as well as reliable and durable, we ultimately decided that this design was preferable.

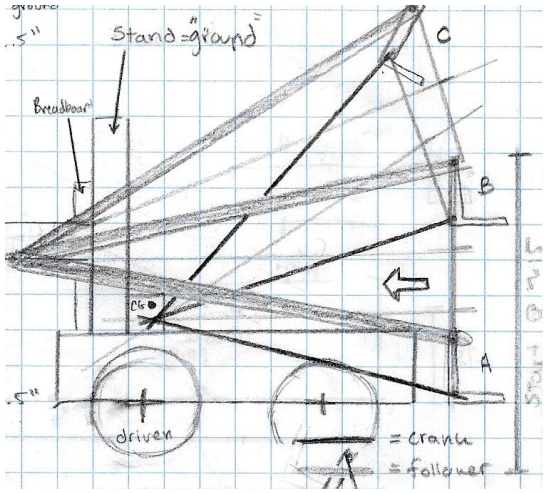

Figure 4.1: Preliminary Design Sketch

Next, we considered our custom electronic circuit. Initially, we planned to use a line tracker using photoresistors and an operational amplifier as our custom electronics circuit. This circuit could be used to track the lines on the field in order to align with the ORBITS. As we

thought about this further, however, we considered the issues we had been having with this circuit and the unreliability of it and decided to use VEX line tracking sensors for this purpose instead. As our robot still required a custom electronic circuit, we decided to use an LCD screen as a serial output for debugging, so that we could more easily understand what was happening as we tested our software. While we were originally going to place our line trackers above the virtual turning center, we were advised that they should be placed at least a little before this point in order to increase their accuracy, as turning compensations would not work properly otherwise.

5 Selection of Final Design

Our group went through several iterations of design throughout the process of building the robot. Some of the elements we tested and altered were the drivetrain, the construction of the lifter, the gearing of the four-bar linkage, and the mounting and use of various sensors.

While we were satisfied with the drivetrain initially, it presented a new challenge to our group once the four-bar linkage was added. The moment of the four-bar linkage inhibited our robot's rear-wheel drive to the point where it could no longer turn. Too much of the weight of the robot was shifted to the front wheels when the linkage was lifted off the ground, a position which the robot needed to assume for large portions of time during demonstration. Our group decided to turn the robot into a four-wheel-drive robot by altering the position of the front drive wheels slightly, adding tires to the previously tireless rims, and gearing the front wheels identically to the back wheels. The design was stable, it would still allow our encoders to function properly, and it still allowed for semi-smooth turning. While this increased the complexity of the design, ultimately it was beneficial in achieving our design goals.

Another problem our group faced was the construction of the four-bar linkage. The initial design that we created before the IDR was heavy, shaky, and could not pick up or hold

tennis balls. The robot was also starting to get heavy so we replaced the metal coupler with a reinforced cardboard coupler. By reinforcing the cardboard with metal, the coupler could still support weight while not weighing down the mechanism or the robot. An additional benefit of the cardboard coupler is it does not catch on the carpet as the metal did. While the cardboard coupler is certainly less sturdy than the metal version, the weight reduction was necessary to reduce the strain on the motor, and to stay within the weight limitations.

Despite the change of coupler, the four-bar needed something to keep the ping pong balls and tennis balls in the lifting mechanism during transport to ORBITS. Our group's solution was to create a domed flap from cardboard to attach to the front of the coupler, and power its rotation using a motor so that it can be opened and closed as a cover over the coupler (Figure 5.1). This cardboard piece is again lightweight, and without gearing the motor has more than enough power to lift and clamp down on tennis balls and ping pong balls. The only drawbacks to using this system are that the use of a motor adds more weight on the end of the four-bar, and that the robot sometimes has difficulty fully encapsulating the tennis ball in its coupler, which can be accounted for in operation. The savings of weight that we made using mostly cardboard instead of metal more than make up for the added weight of the motor, however, and the robot being able to pick up a ball is one of the most important aspects of the game. Despite the coupler now being able to acquire and carry balls, the four-bar mechanism was still flawed.

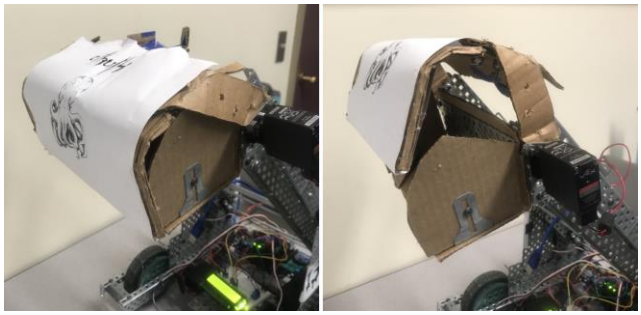

Figure 5.1: Coupler with Rotating Cover

The design of the four-bar mechanism was one of our main focuses after the IDR was complete. The mechanism was originally built in a rushed manner in order to assist in demonstrating the robot's ability to climb the BLACK HOLE. The first generation of the mechanism was thrown together with keps nuts, screws, and a 12:60 transmission on a 3-wire motor for rotating the crank. We essentially rebuilt this mechanism, adding a support midway down the follower for more stability, and plastic bearings and shaft collars on the axles (Figure 5.2). The most important change was a compound transmission (Figure 5.3), with two 12:60 reductions which improved the mechanisms operation significantly and resulted in a much lower gear ratio (*e*) of 0.04 on the arm. Before the change, the mechanism could be operated by remote, and was difficult to keep under control in a constant balance of avoiding motor stall and back drive. With the compound gearing, the motor now runs at a higher RPM which means there is less strain on the motor. The arm also raises at a controllable speed now and does not back-drive. We originally intended for a low *e* value because it would make the arm move in a smooth and predictable motion instead of the jagged motion we could otherwise expect with a

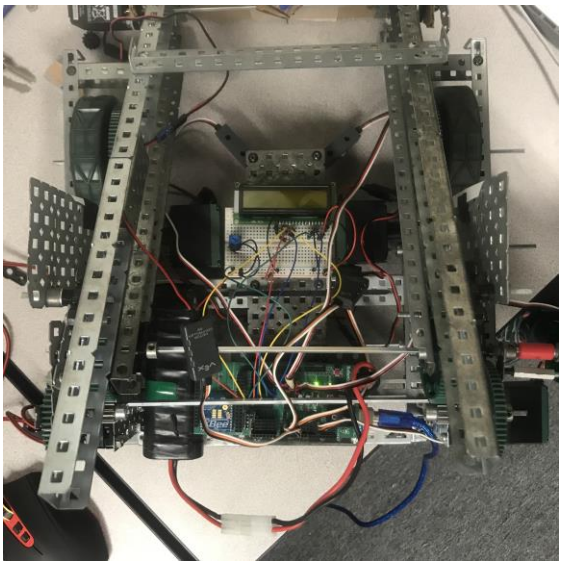

Figure 5.2: Four-Bar Mechanism

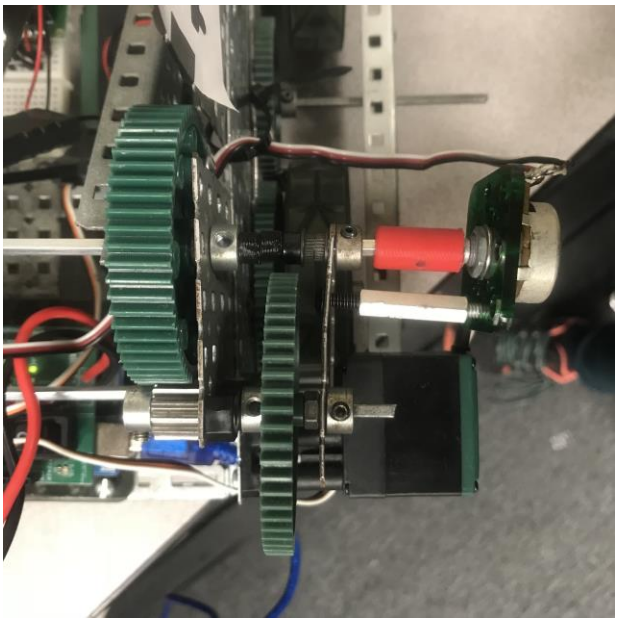

Figure 5.3: Four-Bar Compound Transmission

higher *e*. Therefore, the new compound gear reduction was the perfect gearing for our crank motor.

Our sensor systems also provided us with challenges. Our original plan for autonomous control of the robot was to have the robot drive straight using encoders to ensure that motor rpm was identical on both sides of the drivetrain, gyro to ensure precise turning angles, line trackers to determine position on the field and to approach ORBITS straight on, and a potentiometer to control the angle of the arm. We knew that our line trackers had to be located in front of the VTC, and with our initial design, this did not pose an issue, as with two-wheel drive the VTC was located near the rear of the robot and the sensors could be mounted on one of the cross-bars on the chassis; however, with the switch to four-wheel drive the VTC was moved forward, making the sensors difficult to mount without interfering with the four-bar mechanism. We solved this by adding a plate in the center of the chassis to mount the sensors on, so that they were still in front of the VTC. We also decided to lower the line trackers, making them more accurate, as they were initially about two inches off the ground and we were experiencing repeated inconsistencies with their readings (Figure 5.4). This entire configuration was tighter than we intended, but allowed us to still use the line trackers, an essential part in our autonomous strategy. The encoders did not pose a problem mechanically, but increased the complexity of our software significantly. Since we needed to track the rotations while also counting lines from the line trackers, this would require using interrupts, functionality which we were unable to work out with the object-oriented

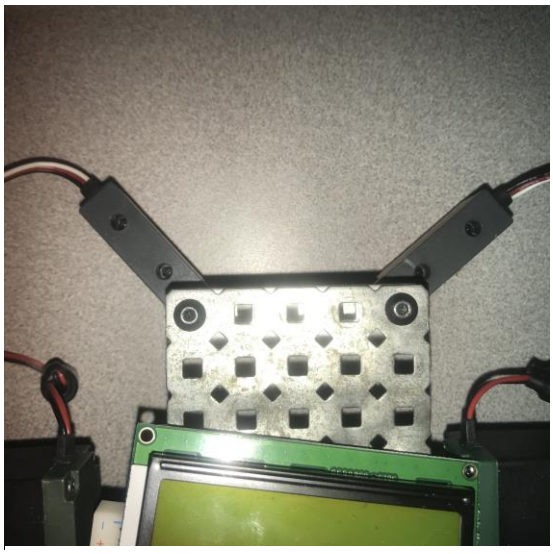

Figure 5.4: Line Tracking Sensor System

class system. After considering our options, we decided to forego the encoders, as their limited use did not justify the time required to make them work in code, and the robot was already able to drive sufficiently straight without a sensor system.

6 Final Design Analysis

6.1 Mechanical Analysis

$$Speed:\ v_{driveline} = d * \pi * n * e$$

$$v_{driveline} = (4\ in)\pi(80\ rpm)\left(\frac{60}{84}\right) = 713.76\frac{in}{min} \approx 12\frac{in}{sec}$$

$$Tractive\ Force: F_{tr} = \mu * N$$

$$F_{tr\ (driveline)} = 1 * (9\ lbs) = 9\ lbs$$

$$Torque\ Force: \tau_f = \left(\frac{\tau_{max}}{e}\right)\left(\frac{\eta}{d}\right)\left(\frac{1}{2}\right) = \left(\frac{14.76\ in \cdot lbs}{0.71}\right)\left(\frac{0.95}{4\ in}\right)\left(\frac{1}{2}\right) \approx 9.9\ in \cdot lbs$$

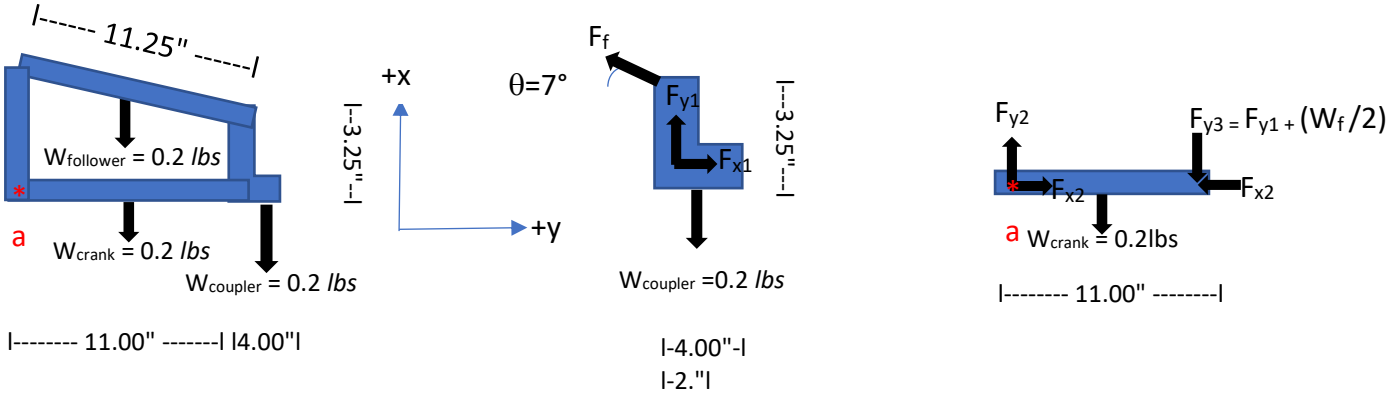

Figure 6.1: Four-Bar Free Body Diagrams

<u>Coupler Equations of Equilibrium:</u>

$$\sum F_{y\ (coupler)} = F_y + F_f * sin(7°) - 0.2\ lbs = 0$$

$$\sum F_{x(coupler)} = F_x - F_f * \cos(7°) = 0$$

$$\sum M_{za(coupler)} = F_f * cos(7°) * (3.25\ in)\ -\ (0.2\ lbs) * (2.0\ in) = 0$$

$$\sum M_{za} = F_f * (3.47) - 0.4\ in \cdot lbs = 0$$

$$\sum M_{za} = F_f = \frac{0.4}{3.47}$$

$$F_f\ = 0.12\ lbs$$

Output Torque of Four-Bar Motor:

$$\sum F_y = F_y + F_f * sin(7°) - 0.2\ lbs = 0$$

$$F_y = -(0.12)(0.12) + 0.2\ lbs$$

$$F_y = -0.0144 + 0.2\ lbs$$

$$F_{y1} = 0.186$$

$$F_{y3(crank)} = F_{y1} + \left(\frac{1}{2}\right)(W_{follower})$$

$$F_{y3(crank)} = 0.186\ lbs + (0.1\ lbs)$$

$$F_{y3} = 0.285\ lbs$$

$$\tau_{out} = F_{y3} * L_{crank} + (W_{crank})\left(\frac{1}{2}\right)(L_{crank})$$

$$\tau_{out}out = (0.285\ lbs)(11.00) + (0.2\text{lbs})(5.5) = 4.2in - lbs$$

$$e_{four-bar} = \frac{N_{drivers}}{N_{driven}} = \left(\frac{12 * 12}{60 * 60}\right) = 0.04$$

$$\left(\frac{\tau_{in}}{\tau_{out}}\right) * \eta = e$$

$$\tau_{in} = 0.19\ in \cdot lbs$$

$$Linear\ Interpolation \rightarrow 88.9\ rpm\ \rightarrow (88.9\ rpm)(0.04) \approx 3.6\ rpm\ of\ 4 - bar$$

As we assessed our design, we began with the typical driveline calculations, finding about 12-in/sec for the speed of the robot, a tractive force of 9-lbs, and a torque force of about 9.9-in·lbs. Since the tractive force is greater than the torque force, the robot is traction-limited. This limitation is more beneficial than having a torque-limited robot, as that would mean we chose a motor without enough power.

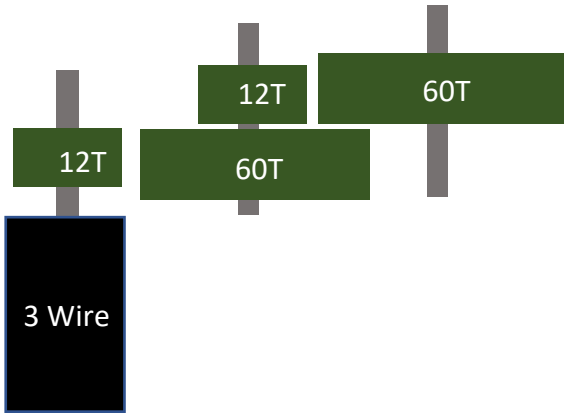

Figure 6.2: Four-Bar Gearing

Next, we assessed our four-bar linkage mathematically. We began with the FBD of the coupler, solved for the force the follower exerted on it at a 7° angle, then used that value, along with the weight of the rest of the four-bar linkage, to determine the torque output by the transmission of the four-bar, which came to 4.2-in·lbs. We then used this torque out and the equation relating torques to the speed ratio to solve for required motor torque, which came out to be 0.19-in·lbs. We then used linear interpolation to determine that the motor would rotate at 88.9-rpm, which would result in the four-bar rotating at 3.6-rpm.

6.2 Programming Methodology

Our robot operates differently in the autonomous and teleoperational periods, which required us to control the robot differently in our autonomous and teleoperational functions in the code. For our autonomous code makes use of proportional control using the gyro sensor for turning, proportional control using the potentiometer to raise and lower the four-bar, and line tracking and counting using the VEX line tracker sensors. The teleoperational code is modelled after the DFW Tank program, making use of the controller buttons to control the coupler cover and the four-bar linkage. The joysticks still operate the same as when running DFW Tank. We

also included messages to be printed on our LCD screen throughout the match, depending on the section of the demonstration the robot is operating in, in order to understand what fails if the robot doesn't perform as intended.

6.3 Sensor Integration

Our final robot made use of two VEX line trackers, one potentiometer, and one gyro. The line trackers are used to count and follow lines autonomously, and the potentiometer is used to track the position of the four-bar, and control angle when rotating the four-bar in both autonomous and teleoperational periods. The gyro is used to make precise 90° turns in the autonomous period.

6.4 Custom Electronics Circuit

Our custom electronic circuit is an LCD display is made from the RBE 1001 student kits, using the breadboard, the LCD display, a potentiometer, and several wires. The LCD display is used to debug the robot code, so that we can easily see what is going on as we test our code. We thought this use of the LCD display would be more effective than using the serial monitor in the Arduino IDE, as it can be used on-field without requiring a laptop with a wired connection to the robot. We are also able to use it during demonstration to ensure that the robot is ready for the autonomous function to be executed. We set up a startup display for the demonstration.

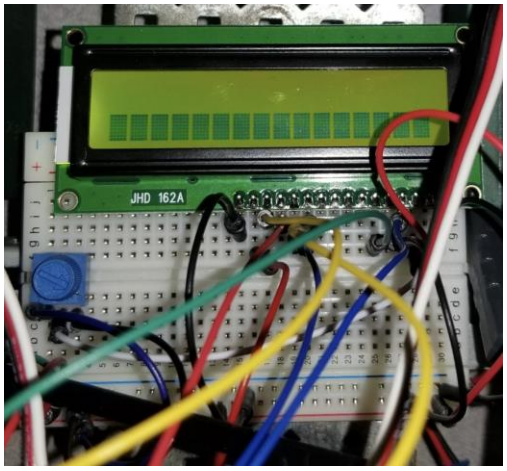

Figure 6.3: LCD Circuit

7 Summary and Evaluation

During the Critical Design Review (CDR) the robot performed almost perfectly during the autonomous period, but our potentiometer had become slightly uncalibrated so that the coupler was just a bit too low. This lower-than anticipated arm height caused the robot's coupler to push into the 12" ORBIT rather than hovering over it, and when the cover flap was raised, the balls missed the ORBIT completely. For the Optional Extended Demonstration (OED), we changed some of the four-bar code, adding a backup to the proportional control to make sure the four-bar is at its max height once it reaches the 12" ORBITS. During the teleoperational demonstration we scored nine points by scoring ORBs in the 12" ORBITS. We struggled picking up tennis balls with the cardboard edge of the coupler, so for the OED we added small spikes of zip ties to the top edge, which proved to be helpful in practice sessions before the OED. We are not able to climb the BLACK HOLE during the CDR like we hoped, and we decided we all needed to practice driving before the OED, also deciding made some of the controls more intuitive for the OED for ease of use.

When the OED came, our robot had some issues with the DFW code, as multiple times the DFW class would never get to calling the teleoperational code. While we remained uncertain why this happened, this bug disappeared after inhibiting us during the first two matches. The robot consistently mechanically, and we were able to demonstrate autonomously scoring four ORBS in a 12" ORBITS, and manually score one ASTEROID in the 12" ORBITS. We were not able to climb the BLACK HOLE as we hoped, though, as when the OED came it was not a high priority to do, as attempting to do so could have done more damage to our robot than the three points were worth. Overall, this project was successful, as we were able to achieve all of our high-priority design specifications, and one of our medium-priority design specifications.

8 Appendices

8.1 Appendix A: Program Documentation

```
/* final_project_two_auto.ino
 * Based off of RBE 1001 DFW Template
 */
#include <DFW.h>
#include "MyRobot.h"
MyRobot myRobot; // Instance of myRobot class for controlling
                 // entire robot
DFW dfw(&myRobot); // Instantiates the DFW object and setting the
                   // debug pin. The debug pin will be set high
                   // if no communication is seen after 2 seconds
void setup() {
  Serial.begin(9600); // Serial output begin. Only needed for
                      // debug
  dfw.begin(); // Serial1 output begin for DFW library. Buad and
               // port #."Serial1 only"
  myRobot.initialize();
  myRobot.dfw=&dfw;
}
void loop() {
  dfw.run();
}
/* MyRobot.cpp
 */
#include "MyRobot.h"
#include "Arduino.h"
/**
   These are the execution runtions
*/
void MyRobot::initialize(void) {
  leftLinePin = A2; // set pin for left line tracker to analog
                    // pin 2
  rightLinePin = A1; // set pin for right line tracker to analog
                     // pin 1
  potPin = A0; // set pin for potentiometer to analog pin 0
  leftmotor.attach(4, 1000, 2000); // left drive motor pin#,
                                   // pulse time for 0, pulse
                                   // time for 180
  rightmotor.attach(5, 1000, 2000); // right drive motor pin#,
                                    // pulse time for 0, pulse
                                    // time for 180
  armMotor.attach(11, 1000, 2000); // arm motor pin#, pulse time
                                   // for 0, pulse time for 180
  flapMotor.attach(7, 1000, 2000); // cover flap motor pin#,
```

```
                                    // pulse time for 0, pulse
                                    // time for 180
pinMode(rightLinePin, INPUT); // set pin for right line tracker
                              // to input
pinMode(leftLinePin, INPUT); // set pin for left line tracker
                             // to input
Ag = 10; // set the gain for proportional control of the arm
Tgain = 15; // set the gain for the proportional control for
            // turning
degreesTurned = 0; // set initial turn angle
armBot = 380; //potentiometer reading for bottom of robot arm
              // range; must be calibrated using calibration
              // code after pot readjustment
armTop = 745; //potentiometer reading for top of robot arm
              // range; must be calibrated using calibration
              // code after pot readjustment
lightThreshold = 900; // threshold for line tracker readings;
                      // must be calibrated using calibration
                      // code in each new lighting environment
rs = 22; // set register select pin
en = 24; // set enable pin
d4 = 25; // set d4 data line pin
d5 = 26; // set d5 data line pin
d6 = 27; // set d6 data line pin
d7 = 28; // set d7 data line pin
jumpPin = A3; // set pin for red/blue team jumper cable
autoPin = A4; // set pin for 6"/12" autonomous jumper cable
jumped = digitalRead(jumpPin); // true if on blue team
                               // false if on red team
shortAuto = digitalRead(autoPin); // true if running 6" ORBITS
                                  // auto
                                  // false if running 12"
                                  // ORBITS auto
/* Initialize LCD) */
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
lcd.begin(16, 2);
lcd.clear();
lcd.print("Lowering Arm");



bool isLowered = false; // True if arm is at bottom position
                        // (potentiometer reads armBot within
                        // margin of error)
                        // False if arm is not at bottom
                        // position
```

```
  /* Make sure cover flap is closed */
  flapMotor.write(60);
  delay(200);
  flapMotor.write(90);
  /* Lower arm */
  lcd.print("Lower the gates!");
  long time0 = millis(); // Initial time for lower arm attempt
  raiseTime = 5500; // Max amount of time to try to lower arm
                    // before giving up
  /* Lowers arm either until arm reaches bottom position
     (potentiometer reads armBot within margin of error) or until
     time allotted runs out */
  while (!isLowered && time0 + raiseTime > millis()) {
    isLowered = moveArm(armBot);
  }
  /* Begin and calibrate gyro */
  Wire.begin();
  /* Checks to make sure gyro is successful in initialization */
  if (!gyro.init()){ // Gyro unsuccessful in initialization
    lcd.clear();
    lcd.print("Gyro fail");
    while (1);
  }
  lcd.clear();
  lcd.print("Steady!");
  gyro.enableDefault();
  calx = calibrate();

  /* Signal that robot is done initializing and ready to begin
     autonomous */
  lcd.clear();
  lcd.print("Aim!");
}
```

```
/**
   Moves arm to given position
   Uses potentiometer to determine current position, and compares
   to given potentiometer value
   @param height Height to raise arm (potentiometer value that
                 should be read at goal height)
   @return True if successfully raised
           False if unsuccessfully raised
```

```
 */
bool MyRobot::moveArm(int height) {
  /* Initialize LCD */
  LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
  lcd.begin(16, 2);

  bool isRaised = false; // True if arm is within declared margin
                         // of height
                         // False if arm is not within declared
                         // margin of height
  float margin = 1.02; // Margin of acceptable error that is
                       // acceptable for arm to be in
  int current = analogRead(potPin); // Current height of arm
  /* Checks if height is lower or higher than goal height */
  if (margin * current < height) { // Lower than goal height
    delay(10);
    // Value to pass to arm motor
    long motorwrite = (90 - (Ag * (current - height)));
    /* Checks that motorwrite is between 0 and 180 */
    if (motorwrite > 180) { // motorwrite greater than highest
                            // possible value
      motorwrite = 180;
    } else if (motorwrite < 0) { // motorwrite less than lowest
                                 // possible value
      motorwrite = 0;
    }
    /* Display current arm position */
    lcd.clear();
    lcd.print(current);
    /* Move arm towards correct position */
    armMotor.write(motorwrite);
  } else if (current > height * margin) { // Higher than goal
                                          // height
    delay(10);
    // Value to pass to arm motor
    long motorwrite = (90 - (Ag * (current - height)));
    /* Checks that motorwrite is between 0 and 180 */
    if (motorwrite > 180) { // motorwrite greater than highest
                            // possible value
      motorwrite = 180;
    } else if (motorwrite < 0) { // motorwrite less than lowest
                                 // possible value
      motorwrite = 0;
    }
    /* Display current arm position */
    lcd.clear();
    lcd.print(motorwrite);
    /* Move arm towards correct position */
```

```
    armMotor.write(motorwrite);
  } else {  // Arm at correct position within margin of error
    isRaised = true;
    armMotor.write(90);
  }
  return isRaised;
}

/**
   Dumps balls from lift using cover flap
   @return void
*/
void MyRobot::dump() {
  /* Open cover flap */
  flapMotor.write(120);
  delay(300);
  flapMotor.write(90);
}

/**
   Detects whether line tracker senses a line
   @param pin Pin number for line tracker to check
   @return True if tracker senses line
           False if tracker does not sense line
*/
bool MyRobot::lineDetect(int pin) {
  return (analogRead(pin) < lightThreshold);
}




/**
   Checks line trackers and adjusts robot position accordingly to
   follow line until an ORBITS is reached
   @param p1 Pin for left line tracker
   @param p2 Pin for right line tracker
   @return void
*/
void MyRobot::lineFollow(int p1, int p2) {
  /* Initialize LCD */
  LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
  lcd.begin(16, 2);
  lcd.clear();
  lcd.print("line following");

  bool distReached = false; // True if ORBITS is reached
```

```
                                // False if ORBITS has not been
                                // reached
  /* Continuously checks line trackers and adjusts until ORBITS
     is reached */
  while (!distReached) {
    /* Check if line trackers detect lines */
    if (lineDetect(p1) && lineDetect(p2)) { // ORBITS reached
      lcd.print("Distance Reached!");
      distReached = true;
    } else if (lineDetect(p2)) { // Right side drifted onto line
      lcd.clear();
      lcd.print("Turn Right!");
      /* Turn robot towards the right */
      rightmotor.write(40);
      leftmotor.write(40);
    } else if (lineDetect(p1)) { // Left side drifted onto line
      lcd.clear();
      lcd.print("Turn Left!");
      /* Turn robot towards the left */
      rightmotor.write(150);
      leftmotor.write(150);
    } else { // Robot not on any line
      lcd.clear();
      lcd.print("Drive Straight!");
      /* Go straight at low speed */
      driveStraight(25);
    }
    delay(20);
  }
}

/**
   Turns robot given number of degrees
   Left turn is negative degrees
   Right turn is positive degrees
   Uses gyro to ensure accuracy
   @param degrees Number of degrees to turn robot
   @return void
*/
bool MyRobot::turn(int degrees) {
  int motorspeed = 0; // Speed to write to motors
  /* Check if robot has reached desired angle */
  if (abs(degreesTurned) < abs(degrees)) { // Robot hasn't
                                          // reached desired
                                          // angle
    /* Let gyro determine current angle */
    gyro.read();
```

```
    theta = ((theta - millis()) / 1000); // Set time since last
                                         // turn
    // Recalculate current angle
    degreesTurned += (15 * theta * readgyrox(calx) / 1000);
    // Calculate what speed to set motors to
    motorspeed = (90 - (Tgain * (degrees - degreesTurned)));
    /* Turn robot */
    leftmotor.write(motorspeed);
    rightmotor.write(motorspeed);
    theta = millis(); // Set to current time
    delay(1);
    return false;
  } else { // Robot has reached desired angle
    return true;
  }
}

/**
   Drives robot straight at given speed
   @param speed Speed to drive motors at
                Must be between -90 and 90
                Integers between -90 and -1 drive robot backwards
                0 stops robot
                Integers between 1 and 90 drive robot forward
   @return void
*/

void MyRobot::driveStraight(int speed) {
  /* Drive robot straight at given speed */
  leftmotor.write(90 + speed);
  rightmotor.write(90 - speed);
}

/**
 * Stop drivetrain motors
 * @return void
 */
void MyRobot::stopMotors() {
  /* Stop robot */
  leftmotor.write(90);
  rightmotor.write(90);
}

/**
   Backs robot up at speed of 60 for 250 milliseconds then stops
   @return void
*/
void MyRobot::backUp() {
```

```
  /* Drive robot backwards */
  driveStraight(-60);
  delay(250);
  /* Stop robot */
  leftmotor.write(90);
  rightmotor.write(90);
}

/**
   Called by the controller between communication with the
   wireless controller during autonomous mode
   @param time Amount of time remaining
   @return void
*/
void MyRobot::autonomous(long time) {
  int startTime = millis(); // Time that autonomous function is
                            // started
  /* Check if doing 6" or 12" ORBITS autonomous */
  if(shortAuto) { // Doing 6" ORBITS autonomous
    runShortAuto(time, startTime);
  } else { // Doing 12" ORBITS autonomous
    runLongAuto(time, startTime);
  }
  /* Stop motors in case robot didn't reach a stop before time
     ran out */
  stopMotors();
}

/**
 * Robot drives forward while raising arm until parallel with 12"
   ORBITS, then turns 90 degrees towards ORBITS, and dumps balls
   into ORBITS
 * @param time Amount of time given for auto
 * @param startTime Time that autonomous period began
 * @return void
 */
void MyRobot::runLongAuto(long time, int startTime) {
  /* Initialize LCD */
  LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
  lcd.begin(16, 2);
  lcd.clear();
  lcd.print("Fire!");
  delay(1000);

  int lineCount = 0; // Number of lines passed by robot
  bool turned = false; // True if robot has already turned
                       // False if robot has not already turned
  bool onLine = false; // True if robot is currently on a line
```

```
                        // False if robot is not currently on a
                        // line
  bool done = false; // True if robot has done everything it's
supposed to for autonomous
                     // False if robot has not done everything
                     // it's supposed to for autonomous
  /* Runs autonomous code while within time limit */
  while (millis() < startTime + time) {
    /* Checks if robot has done everything it's supposed to */
    if (!done) { // Robot has done everything it's supposed to
      /* Move arm up off ground */
      armMotor.write(180);
      delay(4000);
      /* Drive forward */
      driveStraight(50);
      delay(1500);
      /* Drives forward and raises arm while within autotime and
         robot hasn't reached fourth line */


      while (lineCount < 4 && millis() < startTime + time) {
        /* Raise arm to high position */
        moveArm(armTop);
        /* Display how many lines robot has passed */
        lcd.print(lineCount);
        /* Check if robot is on a line that it hasn't counted */
        if (lineDetect(leftLinePin) && lineDetect(rightLinePin)
            && !onLine) { // Robot is on a line it hasn't counted
          lineCount++;
          onLine = true;
        }
        /* Check if robot has moved off a line */
        if (!lineDetect(leftLinePin) && !lineDetect(rightLinePin)
            && onLine) { // Robot has moved off a line
          onLine = false;
        }
        /* Check if robot has reached fourth line */
        if (lineCount == 4) { // Robot has reached fourth line
          /* Stop robot and raise arm */
          stopMotors();
          armMotor.write(180);
          delay(500);
          armMotor.write(90);
          turned = false;
          degreesTurned = 0; // Set how much robot has turned to
                             // 0
          theta = millis(); // Set time since last turn to
                            // current time
```

```
          long time0 = millis(); // Starting time for turn
          /* Turns robot until robot reaches 90 degree turn or
             allotted time for turn runs out */
          while (!turned && ((millis() - time0) < 2000)) {
            /* Checks if robot is on red or blue team */
            if (jumped) { // Robot is on blue team
              turned = turn(-200);
            } else { // Robot is on red team
              turned = turn(80);
            }
            delay (10);
          }
          /* Follow 4th line until ORBITS*/
          lineFollow(leftLinePin, rightLinePin);

          /* Stop robot in front of ORBITS and deposit ORBS */
          stopMotors();
          dump();
          delay(1000);
          /* Backup from ORBITS */
          backUp();
          done = true;
        }
      }
    }
  }
}

/**
 * Robot raises arm, drives forward until 6" ORBITS is reached,
   then dumps ORBS into ORBITS
 * @param time Amount of time given for auto
 * @param startTime Time that autonomous period began
 * @return void
 */
void MyRobot::runShortAuto(long time, int startTime) {
  /* Initialize LCD */
  LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
  lcd.begin(16, 2);
  lcd.clear();
  lcd.print("Fire!");
  delay(1000);

  bool turned = false; // True if robot has already turned
                       // False if robot has not already turned
  bool onLine = false; // True if robot is currently on a line
                       // False if robot is not currently on a
                       // line
```

```
bool done = false; // True if robot has done everything it's
                   // supposed to for autonomous
                   // False if robot has not done everything
                   // it's supposed to for autonomous
bool armRaised = false; // True if arm has been raised to
                        // desired height
                        // False if arm has not been raised to
                        // desired height



/* Runs autonomous function until time runs out */
while (millis() < startTime + time) {
  /* Check if robot has done everything it's supposed to for
     autonomous */
  if (!done) { // Robot has not done everything it's suposed to
               // for autonomous
    long time0 = millis(); // Initial time for robot raising
                           // arm
    /* Raises arm until it reaches the desired height or until
       alloted time runs out or until autonomous period ends */
    while(!armRaised&& raiseTime + time0 > millis() && millis()
          < startTime + time) {
      armRaised = moveArm(armTop);
    }
    /* Drives over BASE line */
    driveStraight(50);
    delay(700);
    /* Runs until reaches 6" ORBITS, or autonomous period ends
     */
    while (!onLine && millis() < startTime + time) {
      /* Raise arm */
      moveArm(armTop);
      /* Check if robot reaches 6" ORBITS for first time */
      if (lineDetect(leftLinePin) && lineDetect(rightLinePin)
          && !onLine) { // Robot reached 6" ORBITS for first
                        // time
        onLine = true;
      }
      /* Check if robot at ORBITS */
      if (onLine) { // Robot at ORBITS
        /* Stop robot and raise arm */
        stopMotors();
        armMotor.write(180);
        delay(500);
        armMotor.write(90);
        /* Back robot up and dump balls into ORBITS */
        backUp();
```

```
          dump();
          delay(1000);
          /* Back robot up */
          backUp();
          done = true;
        }
      }
    }
  }
}

/**
   Called by the controller between communication with the
   wireless controller
   during teleop mode
   @param time the amount of time remaining
   @return void
*/
void MyRobot::teleop(long time) {
  /* Initialize LCD */
  LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
  lcd.begin(16, 2);
  lcd.print("Jesus take the ");
  lcd.setCursor(0, 1);
  lcd.print("wheel!");

  /* RC Controls */
  // DFW.joystick will return 0-180 as an int into
  // rightmotor.write
  rightmotor.write(180 - dfw->joysticklv());
  leftmotor.write(dfw->joystickrv());
  if (dfw->r1()) // Top right bumper raises arm
    armMotor.write(180);
  else if (dfw->r2()) // Bottom right bumper lowers arm
    armMotor.write(0);
  else // If neither bumper pressed, stop arm movement
    armMotor.write(90);
  if (dfw->l1()) // Top left bumper raises cover flap
    flapMotor.write(110);
  else if (dfw->l2()) // Bottom left bumper lowers cover flap
    flapMotor.write(70);
  else // If neither bumper pressed, stop cover flap movement
    flapMotor.write(90);
  /* 30 milliseconds between polling for teleop */
  delay(30);
}

/**
```

```
   Calibrates gyroscope
   @return Degrees needed to zero gyroscope
*/

int MyRobot::calibrate(void) {
  long calx = 0; // Degrees needed to calibrate gyroscope
  /* Finds calx value by taking large sample then averaging */
  for (int i = 0; i < 2000; i++) {
    gyro.read();
    calx = calx + gyro.g.x;
  }
  calx = calx / 2000;
  theta = millis(); // Sets theta to current time
  return calx;
}

/**
   Takes average of 5 gyroscope readings
   @param calx Degrees needed to zero gyroscope
   @return Normalized gyroscope readings
*/
int MyRobot::readgyrox(int calx) {
  double reading = 0; // Gyroscope angle reading
  /* Finds reading by taking large sample then averaging */
  for (int i = 0; i < 5; i++) {
    gyro.read();
    reading += (int)gyro.g.x;
    delay (1);
  }
  reading = reading / 5;
  return (reading - calx);
}

/**
 * Called when the start button is pressed and the robot control
begins
 */
void MyRobot::robotStartup(){

}

/**
 * Called at the end of control to reset the objects for the next
start
 */
void MyRobot::robotShutdown(void){

}
```

```
/* MyRobot.h */
#pragma once

#include "Servo.h"
#include <DFW.h>
#include <AbstractDFWRobot.h>
#include <L3G.h>
#include <Wire.h>
#include<LiquidCrystal.h>

class MyRobot : public AbstractDFWRobot {
  public:
    ~MyRobot() {}; // Destructor for object (default destructor)
    DFW * dfw; // Instance of DFW object
    /**
       Called when the start button is pressed and the robot
       control begins
    */
    void initialize(void);
    /**
       Called by the controller between communication with the
       wireless controller
       during autonomous mode
       @param time the amount of time remaining
       @param dfw instance of the DFW controller
    */
    void autonomous( long time);
    /**
       Called by the controller between communication with the
       wireless controller
       during teleop mode
       @param time the amount of time remaining
       @param dfw instance of the DFW controller
    */
    void teleop( long time);
    /**
       Return the number of the LED used for controller signaling
    */
    int getDebugLEDPin(void) {
      return 13;
    };
    /**
     * Dumps balls from coupler
     */
    void dump();
    /**
     * Follows line using line trackers
```

```
   */
  void lineFollow(int p1, int p2);
  /**
   * Determines whether line tracker senses line
   */
  bool lineDetect(int pin);
  /**
   * Turns robot given number of degrees
   */
  bool turn(int degrees);
  /**
   * Backs up robot then stops
   */
  void backUp();
  /**
   * Short version of autonomous that scores in the 6" ORBITS
   */
  void runShortAuto(long time, int startTime);
  /**
   * Long version of autonomous that scores in the 12" ORBITS
   */
  void runLongAuto(long time, int startTime);
  /**
   * Drives robot straight at given speed
   */
  void driveStraight(int speed);
  /**
   * Calibrates the gyroscope
   */
  int calibrate(void);
  /**
   * Finds normalized reading for gyroscope
   */
  int readgyrox(int calx);
  /**
   * Stops driveline motors
   */
  void stopMotors(void);
  void robotStartup(void);
  void robotShutdown(void);


private:
  unsigned potPin; // Pin for arm potentiometer
  Servo rightmotor; // Motor on right side of drivetrain
  Servo leftmotor; // Motor on left side of drivetrain
  Servo armMotor; // Motor controlling arm
  Servo flapMotor;
```

```
    /* moves arm towards given position */
    boolean moveArm(int height);
    int armTop; // Potentiometer value for the arm at high
                // position
    int armBot; // Potentiometer value for the arm at low
                // position
    int Ag; // Gain for proportional control of the arm
    int lightThreshold; // threshold for line tracker readings,
                        // below this threshold means line is
                        // sensed; must be calibrated using
                        // calibration code in each new lighting
                        // environment
    int Tgain; // Gain for proportional control for turning
    L3G gyro; // Instance of gyroscope
    int calx; // Degrees to zero gyroscope
    double theta; // Amount of time since last turn
    float degreesTurned; // Number of degrees robot has turned
                         // since it began turning
    int rs; // LCD register select pin
    int en; // LCD enable pin
    int d4; // LCD d4 data line pin
    int d5; // LCD d5 data line pin
    int d6; // LCD d6 data line pin
    int d7; // LCD d7 data line pin
    unsigned jumpPin; // Pin for jumper cable signifying whether
                      // robot is on blue or red team
    boolean jumped; // True if robot is on blue team
                    // False if robot is on red team
    unsigned autoPin; // Pin for jumper cable signifying whether
                      // robot is doing 6" or 12"ORBITS
                      // autonomous
    boolean shortAuto; // True if robot is doing 6" ORBITS
                       // autonomous
                       // False if robot is doing 12" ORBITS
                       // autonomous
    unsigned leftLinePin; // Pin for left line tracker
    unsigned rightLinePin; // Pin for right line tracker
    int raiseTime; // Max alloted time to let arm try to move to
                   // correct position
};

/* calibration_code.ino
 * This is used to calibrate the potentiometers and line trackers
 * on the robot
 * It is built off of the DFWTank program
 */
#include <DFW.h> // DFW include
#include <Servo.h> // servo library
```

```
#include <LiquidCrystal.h> // LCD library

// Pin for left line sensor
#define LeftPin A2
// Pin for right line sensor
#define RightPin A1
// Number of lines counted by sensors
int lineCount = 0;
// Current threshold value for line trackers
int lineCal = 930;
class DFWRobot: public AbstractDFWRobot {
    Servo rightmotor; // motor for right side of drivetrain
    Servo leftmotor; // motor for left side of drivetrain
    Servo armMotor; // motor for arm
    Servo flapMotor; // motor for cover flap
  public:
    DFW * dfwPointer; // DFW object
    /**
     * Attatches motors
     */
    void robotStartup() {
      leftmotor.attach(4, 1000, 2000); // left drive motor pin#,
                                       // pulse time for 0,pulse
                                       // time for 180
      rightmotor.attach(5, 1000, 2000); // right drive motor
                                        // pin#, pulse time for
                                        // 0,pulse time for 180
      armMotor.attach(11, 1000, 2000); // arm motor pin#, pulse
                                       // time for 0, pulse time
                                       // for 180
      flapMotor.attach(7, 1000, 2000); // cover flap motor pin#,
                                       // pulse time for 0, pulse
                                       // time for 180
    }
    /**
       Autonomous function for robot (does nothing)
     */
    void autonomous(long time) {};
    /**
     * Determines whether line sensor is over a line
     */
    bool lineDetect(int pin) {
      return (analogRead(pin) < lineCal);
    };
    /**
     * Teleoperational function for robot
     * Used for calibrating potentiometer and line trackers
     */
```

```
void teleop(long time) {
  /* Initialize LCD */
  int rs = 22; // LCD register select pin
  int en = 24; // LCD enable pin
  int d4 = 25; // LCD d4 data line pin
  int d5 = 26; // LCD d5 data line pin
  int d6 = 27; // LCD d6 data line pin
  int d7 = 28; // LCD d7 data line pin
  LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
  lcd.begin(16, 2);

  pinMode(LeftPin, INPUT); // Set pin with left line tracker
                           // to input
  pinMode(RightPin, INPUT); // Set pin with right line
                            // tracker to input

  /* Display line tracker data on LCD */
  lcd.clear();
  lcd.print("C ");
  lcd.print(lineCount);
  lcd.print(" L ");
  lcd.print(analogRead(LeftPin));
  lcd.setCursor(0, 1);
  lcd.print("R ");
  lcd.print(analogRead(RightPin));
  delay(10);

  /* Display potentiometer data on LCD */
  lcd.print("Pot:");
  lcd.print(analogRead(A0));
  delay(50);
  /* Checks if robot is over a line */
  if (lineDetect(LeftPin) && lineDetect(RightPin)) {
    // Robot over a line
    lineCount++;
  }
  /* RC Controls */
  if (dfwPointer->getCompetitionState() != powerup) {
    // DFW.joystick will return 0-180 as an int into
    // rightmotor.write
    rightmotor.write(180 - dfwPointer->joysticklv());
    // DFW.joystick will return 0-180 as an int into
    // leftmotor.write
    leftmotor.write(dfwPointer->joystickrv());
    if (dfwPointer->one()) // 1 moves arm down
      armMotor.write(0);
    else if (dfwPointer->two()) // 2 moves arm up
      armMotor.write(180);
```

```
        else armMotor.write(90); // if neither 1 or 2 pressed,
                                 // arm stationary
        if (dfwPointer->three()) // 3 moves cover flap up
          flapMotor.write(110);
        else if (dfwPointer->four()) // 4 moves cover flap down
          flapMotor.write(70);
        else // if neither 3 or 4 pressed, cover flap stationary
          flapMotor.write(90);
      }
    };
    void robotShutdown(void) {};
    int getDebugLEDPin(void) {
      return 13;
    };
};

DFWRobot robot;
DFW dfw(& robot );  // Instantiates the DFW object and setting
                    // the debug pin. The debug pin will be set
                    // high if no communication is seen after 2
                    // seconds

void setup() {
  Serial.begin(9600); // Serial output begin. Only needed for
                      // debug
  dfw.begin(); // Serial1 output begin for DFW library. Buad and
               // port #."Serial1 only"
  robot.robotStartup();// force a robot startup for testing
  robot.dfwPointer = &dfw; // pass a controller to the robot
}

void loop() {
  dfw.run();// Called to update the controllers output. Do not
            // call faster than every 15ms.
  robot.teleop(0);// run the teleop function manually
}
```

8.2 Appendix B: Bill of Materials

Item	Quantity	Unit Cost	Total Cost
High Strength Gear Kit	1	$20.00	$20.00
Advanced Gearing Kit	2	$20.00	$40.00
Line Trackers (3)	1	$30.00	$30.00
Gyro	1	$12.00	$12.00
		Total Cost	$102.00