The Soul Annoyed Robot: A Senior Project Report

Dayton Muxlow - Computer Science
Christian Johansen - Computer Engineering

June 18, 2019

CSC 491/492 & CPE 461/462

Faculty Advisor: Dr. John Seng
# Contents

1 Introduction 3

2 Problem Statement 3

3 Software 3

3.1 Architecture ................................................. 4
  3.1.1 Procedural Structure ................................. 5

3.2 Functionality ................................................ 6
  3.2.1 Limitations and Strategy ............................ 6
    3.2.1.1 Target Selection ................................ 6
    3.2.1.2 Stack Selection .................................. 6

  3.2.2 Difficulties and Solutions ......................... 6
    3.2.2.1 Chip Blockage .................................. 6
    3.2.2.2 Line Tracking ................................... 6

4 Hardware 7

4.1 Architecture ................................................. 7
  4.1.1 Chip Grabber ........................................... 7
  4.1.2 Conveyor Belt ......................................... 7
  4.1.3 Solenoid Launcher ..................................... 7

4.2 Interactions ................................................ 8
  4.2.1 Grabber to Conveyor Belt ............................ 9
  4.2.2 Conveyor Belt to Launcher ........................... 9

4.3 Circuitry .................................................. 9

5 Mechanical 10

6 Budget and Bill of Materials 13

7 Lessons Learned 14

  7.1 Conveyor Belt Problems ............................... 14
  7.2 Motor Disconnections .................................. 14
  7.3 Software Testing ......................................... 14

8 Conclusion 15

9 Appendix A: Code 16

  9.1 robotLoop.h .............................................. 16
  9.2 robotLoop.ino ............................................ 17
  9.3 motor.h .................................................. 23
  9.4 motor.ino ................................................ 25
1 Introduction

Our goal for this senior project was to create a competitive robot designed to compete in Roborodentia 2019. Our project started during the Winter 2019 quarter, and ended with the competition on May 18, 2019. During that time, we developed an accurate solenoid shooting mechanism, an elevated conveyor belt to carry poker chips, and a servo arm to scoop in stacks of poker chips. These hardware components were attached to a circular differential-drive wooden base designed to be easy to control. We also planned out our match strategy and implemented this strategy with software written in C/Wiring to program our Arduino Mega. The competition consisted of 4 other teams, each with very different approaches to the problem. We named our robot Soul Annoyed, due to our solenoid-based shooting mechanism. We competed in a double elimination match bracket, and won every round, claiming first place. However, two matches were very close, and we realized that being the best robot was not just about having a winning strategy, it was perhaps more important that the robot could execute any strategy reliably and efficiently.

2 Problem Statement

The competition consists of two opposing sides where robots from competing teams complete the same objective. For this competition, the basic objective was to collect stacks of poker chips from the edges of the course and shoot them into seven cups on the midpoint between the two sides. The match was scored by how many cups were "captured" by each robot, which meant one robot had more chips in any given cup than the other robot. The layout of the course is shown in Figure 1. The shooting mechanism required the robot to be able to aim accurately and shoot at least a distance of approximately fourteen inches. This was the most important problem to solve, because our robot started with five chips so if we could accurately shoot we could potentially capture some cups. Aiming was difficult because the only feedback the robot received from the course was a single strip of black tape pointing towards each cup.

The chip collection system was the other big problem to solve. The course had three stacks of five chips each, on the left, back, and right walls. The chips were placed on a platform about two inches behind the four inch wall, which meant the collection mechanism had to be very accurately placed at four inches in height in order to reach out several inches and grab the chips without missing the bottom chip.

Finally, the other problems we had to solve were similar to other competitions. These included accurate line following, wall detection, and accurate rotation and distance measurement.

3 Software

The problem proposed by Cal Poly’s Roborodentia 2019 is mainly a challenge of hardware, but there is certainly software involved, simple as it may be.
3.1 Architecture

As shown in Figure 2, the overall architecture of our software is fairly simple, with the robot loop performing any and all of the meaningful computations.

Other than using included packages, the only interactions that our robot loop makes with other entities are queries and updates to data sources, sinks, and stores.
3.1.1 Procedural Structure

Since the mind of our robot is an Arduino, our software is written in the programming language C, which, in combination with the task of the competition, lends to a procedural style. The only structures used are the "game_state" structure, which we use to store important data for the robot’s current game, such as:

- current iteration number,
- number of chips currently held,
- current robot position,
- current target cup,
- next target cup,
- currently disturbed stacks,
- and which direction the robot is facing,

and any structures defined in "Encoder.h", "DualVNH5019MotorShield.h", or "Servo.h". We wrote two classes, "robotLoop.c" and "motor.c", each of which has their own header. "motor.c" has any functions relating to directly operating the motor, while "robotLoop.c" contains all the strategy logic and controls for any other hardware. For function interactions and general program flow, see Figure 3.

---

Figure 3: Class Overview Diagram
3.2 Functionality

3.2.1 Limitations and Strategy

One of the most difficult parts of creating our software was creating a play strategy based on hardware limitations. Our robot used two wheels and two casters in order to move about the course, which allowed the bot to be accurate but sacrifices the speed that omni-directional wheels might have.

Due to this lack in speed we had to make every reload and every shot count, leading to a strategy that minimizes the distance the robot travels to reload while maximizing the number of unique cups visited between reloads. This resulted in a path where our robot grabs chips, shoots two chips each into its next two targets and one chip into its third target, and seeks the next reload.

3.2.1.1 Target Selection

Our robot selects targets in a linear, pong-like fashion, starting at the rightmost cup and targeting the next cup to the left, with each consecutive cup being our robot’s next target until it changes direction upon reaching the left wall.

3.2.1.2 Stack Selection

Our robot will always seek the closest undisturbed stack. This is done using the robot’s current position and the positions of all the stacks not currently in the list of disturbed stacks tracked by the game_state. Due to our aforementioned strategy and the three minute time limit on matches, our stack selection always results in our robot being placed near its next target.

3.2.2 Difficulties and Solutions

3.2.2.1 Chip Blockage

One of the most difficult parts of this project was getting our chip loading system to a functional state. One major problem was lodged chips causing the conveyor belt to stall. This was solved by reversing the belt intermittently within the shootChips function.

3.2.2.2 Line Tracking

We had a couple problems with implementing and fixing our line tracking. The first problem was detecting T intersections. Our code tracked which cup the robot was at by moving between T intersections, but the light levels were very similar both on T intersections and when the robot is slightly to the left or right, so two sensors see similar light levels. We fixed this problem by adjusting the thresholds in software and moving the 3 line sensors slightly farther apart so we had more of a difference between light levels. Another problem we faced with line following was that occasionally the robot would turn too sharply and drive off the line entirely. We fixed this by saving the last compensation movement direction in the code, and when the robot detected no line at all, it would turn sharply in the last direction it used (either left or right) until it found the line again. By the competition, we had
line following working pretty reliably, although with more work it could have been even better.

4 Hardware

4.1 Architecture

Hardware was easily the most difficult part of this project to design and implement. Our design was based around three main components: the chip grabber, the conveyor belt loader, and the solenoid launcher. The idea is that the chips move from one component to the next, allowing the grabber to scoop five chips at once while allowing the conveyor belt to feed the chips into the launcher one by one.

4.1.1 Chip Grabber

Our design for the grabber started as a hook design mounted onto a servo, enabling it to move out and to the side of the chip stack while the robot moves forwards and then scoop the chip stack into the robot in one circular servo motion. This design actually worked fairly well once we got the right shape of the hook. The first shape was a half circle with a handle attached to the servo, and this worked well for scooping but the bottom chip would easily fall out of the bottom. This was remedied by flattening out the center of the hook, creating more of a straight beam with two curved bits to keep the chips in a circular motion.

4.1.2 Conveyor Belt

Our initial design for feeding the stack of chips into the launcher was a tilted linear elevator, but we realized that was difficult to implement reliably with a single servo, and we wanted to minimize the servos used. After some brainstorming and discussion with our adviser, Dr. Seng, we came up with a conveyor belt concept. The stack of chips were loaded onto the conveyor belt, and small bits glued onto the belt would grab the bottom chip and move it up. The belt was driven by a continuous rotation servo. Due to the significant tilt of the belt shown in Figure 4, the chips on top of the bottom chip would slide down until they are picked up one by one. This functionality required significant tuning before the competition, and it was our least reliable component in the system, causing chips to double up in the launcher or get trapped. However, close to the competition we realized that the chips were magnetic, so we attached tiny magnets all around each belt which significantly improved the reliability of the belt. Some chips still got out of place during the competition, but without the magnets our robot would not have run very well.

4.1.3 Solenoid Launcher

The solenoid launcher was the first concept we decided to test, and ultimately defined our robot’s personality. We discussed a few ideas with Dr. Seng, such as a flywheel, a circular thrower, a catapult, or even a large rubber band. We decided on a solenoid because it was very mechanically simple, and we thought that this would give reproducible results that were easy to control. Our initial tests with a 12 volt solenoid went well. We built a launcher rail that was elevated so the chip would be
launched in the proper trajectory to land into the cups, and we were able to score about 80% of the time on average. Initially, we controlled and powered the solenoid with a relay and flyback diode connected to a 12.8 V battery pack, but later on we got a 4S high-current LiPo battery pack that gave our solenoid more power. Once we tested it, the solenoid shot the chips over two feet, which was enough distance to make it into the cups from the horizontal tape between the left and right chip stacks. This significantly decreased the amount of movement our robot needed to make, and that was a big factor in how well we did in the competition. There are a few downsides to the solenoid, however. We noticed that it was susceptible to changes in the battery voltage as it drained, but this was not a problem because the competition was short. The last downside was that the position of the chip when the solenoid hit is was very important to get right, so occasionally a chip would fall short because it wasn’t quite positioned correctly.

![Figure 4: Launcher and Conveyor Belt Architecture](image)

4.2 Interactions

The interactions between the three main hardware components were difficult to get right. There were two main transitions: the transition between the grabber scooping in the chip stack and aligning it on the belt so that they would smoothly move upwards in single file, and the transition between the conveyor belt and the diagonal sliding drop that placed the chip in the correct position above the solenoid arm. Each part required significant tuning to get working, and there were a few things that we could have included in our initial design to make this smoother.
4.2.1 Grabber to Conveyor Belt

The difficulty with this transition was that the chip stack had to go from being flat to being elevated at a steep angle on the belt. First, we had to modify the hook to hold in the chips while the belt grabbed them one by one, because at that steep angle the chips would easily fall out the front. There were also several other modifications we had to make to keep the chips from falling off the sides, getting stuck beneath the belts, or getting stuck between the bottom belt pulleys and the front wall of the robot. These fixes were implemented using a bunch of MDF wood and a combination of glue and screws. We could have definitely improved this behavior if we had designed the belt system more carefully and left some more room for either a gentler angle or a smoother elevation transition.

4.2.2 Conveyor Belt to Launcher

The transition from conveyor belt to launcher was very annoying and difficult to get right. The idea was the belt would pick up the bottom chip from the stack, while the rest of the chips would slide down, so only one chip would get pushed into the launcher section. This did not always work, and even at the competition we got two chips pushed into launcher section once or twice, causing both to be improperly launched. Our original design imagined the belt and diagonal chip loading slide at a much gentler angle than we actually used at build time. This was partially due to space constraints on the robot and the fact that we built and mounted the launcher first before prototyping the conveyor belt. However, we significantly improved this transition through the use of small magnets just strong enough to pick up one chip, and the use of a wooden "ceiling" above the top of the belt to guide any extra chips back down the belt.

4.3 Circuitry

The circuitry used in our design was simple, but had a few interesting components. The Arduino Mega was used to control everything on the robot, and we used a dual motor controller Arduino shield plugged into the top of it. The following is a list of the pins used on the Arduino to control the various components on the robot:

- Solenoid: Pin 3 DIGITAL OUT
- Grabber: Pin 5 PWM
- Belt: Pin 11 PWM
- VNH5019 Motor Controller: Pins 2, 4, 6, 7, 8, 9, 10, 12, A0, A1
- Left Encoder: Output A to pin 19, Output B to pin 18
- Right Encoder: Output A to pin 20, Output B to pin 21
- Left Line Sensor: Pin A8
- Center Line Sensor: Pin A9
- Right Line Sensor: Pin A10
• Chip Sensor: Pin A11

All of these components are shown in Figure 5.

The solenoid is controlled by the Arduino using a relay, which means the software required for shooting is a simple digitalOut, a delay of about 200 ms, and another digitalOut. The flyback diode is required across the solenoid terminals because during actuation the coils draw a large amount of current, and have a significant amount of inductance and voltage, so when the current is cut off the energy in the coil has to be safely dissipated. When we first designed the circuitry, we had all components running off of the LiPo battery, but during testing we discovered that the extremely fast, high current draw of the solenoid dropped the battery voltage enough for the Arduino to reset itself. To fix this problem, we used a space AA battery pack to drive the Arduino and the servos. The LiPo, battery pack, and relay can be seen in Figure 6.

The motor controller shield provided a easy interface with the Arduino. Although the module was a bit expensive, we were glad we got it because it worked easily and reliably (once we got the motor connections working, explained below in the Lessons Learned section).

Our sensor circuitry was simple. Three IR line sensors were read through analog pins on the Arduino, and since each required power and ground we installed a breadboard on the robot and used the power and ground rails. The encoders were simple enough, although it took some testing to get the orientation of the A and B inputs correct and the quadrature readings correct on the software side. The Pololu geared motors we used combined the motor outputs with the encoder outputs in one 6-pin header, so we had to use some extra cables to connect everything. The last sensor was the poker chip sensor. We used a simple analog light sensor from an old project that was embedded into the chip launcher rail, so detecting the presence of the chip was a simple value threshold check. The light sensor works by decreasing its resistance linearly with the increasing amount of light it sees. To hook up the sensor, we put a 10K ohm protection resister in series with the light sensor. One pin on the sensor was connected to 5V in, and the other was connected through the resistor to an analog input pin on the Arduino. Once this chip was on the launcher, the resistance increased, causing a significant voltage drop on the input pin. Setting up the servo circuitry was very simple because they have 3 pins, one to 5V, one to ground, and one to a PWM pin on the Arduino.

5 Mechanical

The main material we used to construct the robot was several sizes of MDF wood sheets. We chose this material because it was readily available and easy to shape and drill into. We designed the base to be round with cutouts on the sides for the wheels to keep rotations simple without having to worry about corners hitting the walls. The rest of the main structure was all cut out of MDF, such as the launcher rail, the diagonal slide, and the front panel that lines up with walls and mounts the grabber servo. These parts were modified as we went along and as we tested on the actual course. Figure 7 shows the wooden structure of the robot.
Figure 5: Circuitry Architecture Diagram

Figure 6: Hardware Parts
Figure 7: The structure of the robot.

Figure 8: Robot in Action
## 6 Budget and Bill of Materials

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Vendor</th>
<th>Total Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Drive Motors</td>
<td>Left and right wheels in a differential drive.</td>
<td>Cal Poly Robotics Club</td>
<td>$20</td>
</tr>
<tr>
<td>2 Casters</td>
<td>Attached to the front and the back of the robot base.</td>
<td>Pololu</td>
<td>$7.90</td>
</tr>
<tr>
<td>Dual Motor Controller</td>
<td>Allows the Arduino to regulate the drive motors.</td>
<td>Pololu</td>
<td>$50</td>
</tr>
<tr>
<td>Arduino Mega 2560</td>
<td>The main controller of the robot.</td>
<td>Already Own</td>
<td>$30</td>
</tr>
<tr>
<td>3 IR Line Sensors</td>
<td>For following the black tape lines.</td>
<td>Already Own</td>
<td>$8.85</td>
</tr>
<tr>
<td>1 Continuous Rotation Servo</td>
<td>For driving the conveyor belt.</td>
<td>Amazon</td>
<td>$20</td>
</tr>
<tr>
<td>1 Standard Servo</td>
<td>For the chip grabber.</td>
<td>Already Own</td>
<td>$15</td>
</tr>
<tr>
<td>Solenoid</td>
<td>Main chip launcher.</td>
<td>Dr. Seng</td>
<td>$20</td>
</tr>
<tr>
<td>LiPo Battery Pack</td>
<td>4S 2200 mAH battery pack for the solenoid and motors.</td>
<td>Amazon</td>
<td>$30</td>
</tr>
<tr>
<td>5V Relay</td>
<td>To control the solenoid actuation.</td>
<td>Amazon</td>
<td>$6.79</td>
</tr>
<tr>
<td>2 Wheels</td>
<td>Attached to left and right motors.</td>
<td>Pololu</td>
<td>$16.90</td>
</tr>
<tr>
<td>Timing Belt Kit</td>
<td>Used for the conveyor belt.</td>
<td>Amazon</td>
<td>$15.99</td>
</tr>
<tr>
<td>Misc Screws/Wood</td>
<td>Used to construct the frame.</td>
<td>Various</td>
<td>$44</td>
</tr>
</tbody>
</table>
7 Lessons Learned

The best projects are those that require plenty of improvisation, and this project had plenty. The most difficult part was designing and implementing the hardware to manage and launch the poker chips. We learned about the pros and cons of various launching mechanisms, and while we were happy with our results with the solenoid, it would have been interesting to try out a flywheel or catapult design. Picking the solenoid design was beneficial to us in the long run because it meant we had a reliable and extremely easy to control method of launching chips. We also initially thought we would need guides on our launcher to make sure the chip went the correct direction, but we realized later on that the solenoid was pretty consistent with the shooting angle, and the difficulty was getting the entire robot to correctly line up at the angle for shooting.

7.1 Conveyor Belt Problems

The area we struggled in the most was the conveyor belt. We initially wanted to use an elevator to load the chips into the launcher, but we realized a conveyor belt would be better a bit later than we should have. Also, with a bit of planning and experimentation with different materials, we could have made a better belt than the three timing belts we used. We learned that while glue was very useful for many things, it was difficult to use with hard rubber belts that needed to curve around pulleys. Additionally, if we had discovered that the chips could be used with magnets earlier, we would have taken advantage of that more and perhaps centered our belt design around magnetism.

7.2 Motor Disconnections

Another lesson was learned last-minute when our motors would occasionally disconnect due to bad connections between the 22 gauge wire and the terminal blocks on the motor controller shield. We ended up resoldering these connections several times, and it finally worked when we just gave up and soldered the wires directly to the motor controller circuit board rather than using the terminal blocks. We learned that often we forget to check the reliability of connections and that a bad connection can cause the robot to completely stop working in the middle of a match.

7.3 Software Testing

We also faced some challenges in implementing parts of the software. The higher-level strategy code was easy to understand and change, but getting the line following and encoder measurement to be reliable was difficult. However, the differential drive we used made this simpler, although perhaps a bit slower than other systems. If we repeated the project, we might decide to take the wall following approach instead of sticking to the lines. Several robots in the competition used simple contact switches and omni-directional wheels to navigate along walls. We learned that there are many approaches that all have different pros and cons, and the best approach depends more on the time and effort spent perfecting the implementation.
8 Conclusion

We set out on the project to compete in Roborodentia and hopefully win one of the prizes, and we definitely accomplished our goals. Although we ran into some problems and should have started testing all three hardware components earlier, we learned so much about robotics and ultimately solved or significantly improved all the problems we faced. We would like to thank our adviser Dr. John Seng for guiding us through our many design iterations and tests, and providing helpful advice when something wasn’t working correctly. We would also like to thank the volunteers in the Roborodentia club for designing and running the event. Finally, we would like to thank our fellow competitors for making the event fun and exciting.
9 Appendix A: Code

9.1 robotLoop.h

```c
#ifndef ROBOTLOOP_H
#define ROBOTLOOP_H

// Pins
#define SOLENOID 3
#define ARM 5
#define BELT 11
#define CHIPSSENSOR A11
#define BUTTON 50

// Directions
#define FORWARD 16
#define LEFT 0
#define BACK 3
#define RIGHT 6

// Short
#define CUP1 0
// Tall
#define CUP2 1
// Short
#define CUP3 2
// Tall
#define CUP4 3
// Short
#define CUP5 4
// Tall
#define CUP6 5
// Short
#define CUP7 6
// NONE
#define NONE 16

typedef struct game_state {
    uint8_t numLoops = 0;
    uint8_t chipsHeld = 5;
    uint8_t currentPosition = CUP6;
    uint8_t currentTarget = CUP6;
    uint8_t nextTarget = CUP7;
    uint8_t stacksDisturbed[3];
} game_state;
```
```c
uint8_t numDisturbedStacks = 0;

uint8_t facing = FORWARD;
}
```

```c
#include "robotLoop.h"
#include "motor.h"
#include <Servo.h>

// 2,4,6,7,8,9,10,12

game_state gameState;
Servo arm_servo;
Servo belt_servo;

void setup() {
    // put your setup code here, to run once:
    Serial.begin(9600);

    // Initialize solenoid
    pinMode(SOLENOID, OUTPUT);
digitalWrite(SOLENOID, HIGH);

    // Initialize arm servo
    arm_servo.attach(ARM);
closeArm();

    // Initialize belt servo
    belt_servo.attach(BELT);
belt_servo.write(90);

    // Initialize motors
    initMotors();

    // Initialize button
    pinMode(BUTTON, INPUT_PULLUP);

closeArm();
// Wait for button press
while (digitalRead(BUTTON) != LOW);
}```
void loop() {
  // put your main code here, to run repeatedly

  if (gameState.numLoops == 0){
    shootChips(5);
  } else {
    shootChips(2);
  }

  seekCurrentTarget();

  gameState.numLoops++;
}

//
// void goToLocation(uint8_t location) {
//  int8_t fork_distance = location - gameState.currentPosition;
//  uint8_t direction_to_move = fork_distance > 0 ? RIGHT : LEFT;
//  delay(350);
//  
//  // Rotate in the desired direction
//  if (gameState.facing == FORWARD) {
//    if (direction_to_move == RIGHT) {
//      rotateUntilLine(CW, 90);
//    } else if (direction_to_move == LEFT) {
//      rotateUntilLine(CW, 90);
//    }
//  } else if (gameState.facing == LEFT) {
//    if (direction_to_move == RIGHT) {
//      if (gameState.currentPosition == BACK) {
//        rotateUntilLine(CW, 90);
//        rotateUntilLine(CW, 90);
//      } else {
//        rotateUntilLine(CW, 180);
//      }
//    }
//  } else if (gameState.facing == RIGHT) {
//    if (direction_to_move == LEFT) {
//      if (gameState.currentPosition == BACK) {
//        rotateUntilLine(CW, 90);
//        rotateUntilLine(CW, 90);
//      }
//    }
//  }
//  else if (gameState.facing == BACK) {
//    if (direction_to_move == RIGHT) {
//      if (gameState.currentPosition == BACK) {
//        rotateUntilLine(CW, 90);
//        rotateUntilLine(CW, 90);
//      }
//    } else if (direction_to_move == LEFT) {
//      if (gameState.currentPosition == BACK) {
//        rotateUntilLine(CW, 90);
//        rotateUntilLine(CW, 90);
//      }
//    }
//  }
//  
//  return;
//}
else {
    rotateUntilLine(CW, 180);
}

}  
}

gameState.facing = direction_to_move;
delay(600);

for (uint8_t i = 0; i < abs(fork_distance); i++) {
    delay(50);
    moveToForkOrWall(direction_to_move);
}

gameState.currentPosition = location;
delay(300);
}

// Assumes starting position is farthest fork to the right
void seekCurrentTarget() {
    Serial.println("Seeking target...");
    goToLocation(gameState.currentTarget);
delay(500);
    aimAtTarget();
}

void openArm() {
    arm_servo.write(170);
}

void closeArm() {
    arm_servo.write(77);
delay(100);
    arm_servo.write(60);
}

void rotateBelt() {
    belt_servo.write(82);
}

void reverseBeltMS(int ms) {
    belt_servo.write(95);
delay(ms);
    stopBelt();
}

void stopBelt() {

belt_servo.write(90);
}

void loadChipToLaunch() {
  Serial.println("Loading chip...");
  rotateBelt();
}

void aimAtTarget() {
  Serial.println("Aiming...");

  // Move forward half the size of the bot.
  if (gameState.currentTarget != CUP1 && gameState.currentTarget != BACK &&
      gameState.currentTarget != CUP7 && gameState.facing != FORWARD) {
    driveDistance(FORWARDS, 110);
    delay(500);
  }

  if (gameState.facing == LEFT) {
    if (gameState.currentPosition == RIGHT) {
      rotateDistance(CW, 97);
    }
    else if (gameState.currentPosition == LEFT) {
      rotateDistance(CW, 86);
    }
    else if (gameState.currentPosition == BACK) {
      rotateDistance(CW, 79);
    }
    else {
      rotateDistance(CW, 90);
    }
  }

  else if (gameState.facing == RIGHT) {
    rotateDistance(CCW, 75);
  }
  delay(500);

  gameState.facing = FORWARD;

  if (gameState.currentPosition == CUP2 || gameState.currentPosition == CUP4 ||
      gameState.currentPosition == CUP6) {
    driveStraight(14);
  }
  else {
    driveDistance(BACKWARDS, 4);
  }
void punchChip() {
    Serial.println("Punching...");
    digitalWrite(SOLENOID, LOW);
    delay(200);
    digitalWrite(SOLENOID, HIGH);
    delay(300);
}

void shootChips(uint8_t numChips) {
    uint8_t chipsShot = 0;
    boolean timeExpired = false;
    long startTime = millis();
    
    rotateBelt();
    while (chipsShot < numChips && !timeExpired) {
        if (analogRead(CHIPSENSOR) < 500) {
            stopBelt();
            delay(350);
            punchChip();
            chipsShot++;
            gameState.chipsHeld--;
            
            rotateBelt();
        }
        else if (millis() - startTime > 20000) {
            timeExpired = true;
            gameState.chipsHeld = 0;
        }
        else if ((millis() - startTime) % 3000 == 0) {
            reverseBeltMS(500);
            rotateBelt();
        }
        if (gameState.chipsHeld == 0) {
            timeExpired = true;
            stopBelt();
            delay(100);
            
            seekClosestLoad();
            delay(250);
            performLoadingSequence();
        }
    }
}
stopBelt();

uint8_t nextTarget = abs((gameState.nextTarget - gameState.currentTarget) +
    gameState.nextTarget);
gameState.currentTarget = gameState.nextTarget;
gameState.nextTarget = nextTarget > CUP7 ? CUP6 : nextTarget;

driveDistance(BACKWARDS, 23);
}

void seekClosestLoad() {
    Serial.println("Seeking reload...");

    uint8_t closestLoad = NONE;
    uint8_t loadPositions[3] = {LEFT, BACK, RIGHT};

    for (uint8_t i = 0; i < 3; i++) {
        uint8_t currentStack = loadPositions[i];
        uint8_t isDisturbed = 0;

        for (uint8_t j = 0; j < gameState.numDisturbedStacks; j++) {
            uint8_t distStack = gameState.stacksDisturbed[j];
            if (distStack == currentStack) {
                isDisturbed = 1;
            }
        }

        if (isDisturbed == 0 && abs(gameState.currentPosition - currentStack) <
            abs(gameState.currentPosition - closestLoad)) {
            closestLoad = currentStack;
        }
    }

    openArm();
    delay(250);
    goToLocation(closestLoad);
    delay(250);
    // If we add delays to goToLocation, we may want to remove this delay

    if (closestLoad == BACK) {
        driveDistance(FORWARDS, 110);
        if (gameState.facing == LEFT) {
            rotateUntilLine(CCW, 90);
        }
        else if (gameState.facing == RIGHT) {
            rotateUntilLine(CW, 90);
        }
        delay(600);
gameState.facing = BACK;
moveToForkOrWall(BACK);
delay(500);
}
}

void refreshStacks() {
    gameState.numDisturbedStacks = 1;
    gameState.stacksDisturbed[0] = gameState.stacksDisturbed[2];
    gameState.stacksDisturbed[1] = gameState.stacksDisturbed[2];
}

void performLoadingSequence() {
    Serial.println("Reloading...");
    closeArm();
delay(700);

gameState.chipsHeld += 5;
    gameState.stacksDisturbed[gameState.numDisturbedStacks] = gameState.currentPosition;
    gameState.numDisturbedStacks++;
if (gameState.numDisturbedStacks == 3) {
    refreshStacks();
}

driveDistance(BACKWARDS, 10);
// Return to the line if in the back
if (gameState.facing == BACK) {
    driveDistance(BACKWARDS, 105);
delay(250);
rotateUntilLine(CCW, 90);
delay(500);
gameState.facing = RIGHT;
}
}

9.3 motor.h

#ifndef MOTOR_H
#define MOTOR_H

#define BASE_SPEED_LEFT 33
#define BASE_SPEED_RIGHT 32
#define MM_PER_TICK 0.0785398
#define TICKS_PER_MM 12.7323981
#define TICKS_PER_DEG (1.9118 * TICKS_PER_MM)

23
#define ENC_CHECK_INTERVAL 10
#define ENC_KP 4
#define STALL_CHECK_INTERVAL 100
#define STOPPING_COMP 50
#define LEFT_LINE A8
#define CENTER_LINE A9
#define RIGHT_LINE A10
#define THRESHOLD 700
#define THRESHOLD_MED 150
#define THRESHOLD_T 850
#define LINE_KP 0.2
#define STALL_THRESHOLD 5

enum Direction {
    FORWARDS = 1,
    BACKWARDS = -1
};

enum Rotation {
    CW = 1,
    CCW = -1
};

void initMotors();
void printLineSensors();
void readLineSensors();
int getLineError();
void moveToForkOrWall(uint8_t heading);
void rotateUntilLine(Rotation rot, int deg);
void driveDistance(Direction dir, int mm);
void rotateDistance(Rotation rot, int deg);
void driveStraight(int mm);
#endif
# 9.4 motor.ino

```cpp
#include "DualVNH5019MotorShield.h"
#include <Encoder.h>
#include "motor.h"

DualVNH5019MotorShield motors;
Encoder leftEnc(18, 19);
Encoder rightEnc(20, 21);

long leftEncGoal = 0;
long rightEncGoal = 0;
long leftEncPrev = 0;
long rightEncPrev = 0;

int leftLine = 0;
int centerLine = 0;
int rightLine = 0;

long startTimer = 0;
long endTimer = 0;

void initMotors() {
    motors.init();
    leftEnc.write(0);
    rightEnc.write(0);
}

void printLineSensors() {
    Serial.print("LINE SENSORS: ");
    Serial.print(leftLine);
    Serial.print(" ");
    Serial.print(centerLine);
    Serial.print(" ");
    Serial.print(rightLine);
    Serial.println();
}

void readLineSensors() {
    leftLine = analogRead(LEFT_LINE);
    centerLine = analogRead(CENTER_LINE);
    rightLine = analogRead(RIGHT_LINE);
}

int clamp(int value, int minimum, int maximum) {
    if (value < minimum) {
        return minimum;
    }
    else if (value > maximum) {
        return maximum;
    }
    return value;
}
```
return maximum;
}
return value;
}

/**
 * Returns the "error", which is how much off center the line is
 * as a number between -100 and 100. -100 means the line is to the left,
 * so the robot needs to move left, and 100 means the line is to the right,
 * so the robot needs to move right.
 */
int getLineError() {
  if (leftLine < THRESHOLD && centerLine >= THRESHOLD
      && rightLine < THRESHOLD) {
    return clamp(2 * (rightLine - leftLine), -50, 50);
  }
  else if (leftLine >= THRESHOLD_MED && centerLine >= THRESHOLD_MED
            && rightLine < THRESHOLD) {
    return -70;
  }
  else if (leftLine >= THRESHOLD && centerLine < THRESHOLD
            && rightLine < THRESHOLD) {
    return -100;
  }
  else if (leftLine < THRESHOLD && centerLine >= THRESHOLD_MED
            && rightLine >= THRESHOLD_MED) {
    return 70;
  }
  else if (leftLine < THRESHOLD && centerLine < THRESHOLD
            && rightLine >= THRESHOLD) {
    return 100;
  }
  else {
    return 0;
  }
}

void moveToForkOrWall(uint8_t heading) {
  driveDistance(FORWARDS, 15);
  motors.setSpeeds(BASE_SPEED_LEFT, BASE_SPEED_RIGHT);
  startTimer = millis();
  endTime = startTimer + STALL_CHECK_INTERVAL + 1000;
  leftEncPrev = leftEnc.read();
  rightEncPrev = rightEnc.read();
  delay(400);
  while (1) {
    startTimer = millis();
    if (leftLine < THRESHOLD && centerLine >= THRESHOLD
        && rightLine < THRESHOLD) {
      return clamp(2 * (rightLine - leftLine), -50, 50);
    }
    else if (leftLine >= THRESHOLD_MED && centerLine >= THRESHOLD_MED
             && rightLine < THRESHOLD) {
      return -70;
    }
    else if (leftLine >= THRESHOLD && centerLine < THRESHOLD
             && rightLine < THRESHOLD) {
      return -100;
    }
    else if (leftLine < THRESHOLD && centerLine >= THRESHOLD_MED
             && rightLine >= THRESHOLD_MED) {
      return 70;
    }
    else if (leftLine < THRESHOLD && centerLine < THRESHOLD
             && rightLine >= THRESHOLD) {
      return 100;
    }
    else {
      return 0;
    }
  }
}
if (startTimer >= endTimer) {
    if (abs(leftEnc.read() - leftEncPrev) < STALL_THRESHOLD
        || abs(rightEnc.read() - rightEncPrev) < STALL_THRESHOLD) {
        Serial.println("GOT STALL");
        break;
    } else {
        endTimer = startTimer + STALL_CHECK_INTERVAL;
        leftEncPrev = leftEnc.read();
        rightEncPrev = rightEnc.read();
    }
}

readLineSensors();
//printLineSensors();

if (heading == LEFT && centerLine >= THRESHOLD_T
    && rightLine >= THRESHOLD_T) {
    break;
} else if (heading == RIGHT && leftLine >= THRESHOLD_T
    && centerLine >= THRESHOLD_T) {
    break;
}

int line_error = getLineError();

motors.setSpeeds(BASE_SPEED_LEFT + (LINE_KP * line_error),
    BASE_SPEED_RIGHT - (LINE_KP * line_error));

motors.setBrakes(400, 400);
motors.setSpeeds(0, 0);
}

void rotateUntilLine(Rotation rot, int deg) {
    int seenLeft = 0;
    int seenCenter = 0;
    int seenRight = 0;
    int almostThere = 0;

    rotateDistance(rot, deg/2);

    motors.setSpeeds(rot * BASE_SPEED_LEFT, -1 * rot * BASE_SPEED_RIGHT);
    startTimer = millis();
    endTimer = startTimer + STALL_CHECK_INTERVAL + 1000;
    leftEncGoal = leftEnc.read() + (rot * (int)((float)deg * TICKS_PER_DEG));
    rightEncGoal = rightEnc.read() +
        (-1 * rot * (int)((float)deg * TICKS_PER_DEG));

leftEncPrev = leftEnc.read();
rightEncPrev = rightEnc.read();

while (1) {
    startTimer = millis();
    long leftEncCurr = leftEnc.read();
    long rightEncCurr = rightEnc.read();

    readLineSensors();
    //printLineSensors();

    if (rot == CW &&
        (centerLine >= THRESHOLD || rightLine >= THRESHOLD)) {
        break;
    }
    else if (rot == CCW &&
        (leftLine >= THRESHOLD || centerLine >= THRESHOLD)) {
        break;
    }

    if (startTimer >= endTimer) {
        long lDiff = abs(leftEncCurr - leftEncPrev);
        long rDiff = abs(rightEncCurr - rightEncPrev);
        if (lDiff > rDiff) {
            motors.setSpeeds(rot * (BASE_SPEED_LEFT -
                                (ENC_KP * lDiff / rDiff)), -1 * rot * BASE_SPEED_RIGHT);
        }
        else if (rDiff > lDiff) {
            motors.setSpeeds(rot * BASE_SPEED_LEFT, -1 * rot *
                                (BASE_SPEED_RIGHT - (ENC_KP * rDiff / lDiff)));
        }
        endTimer = startTimer + ENC_CHECK_INTERVAL;
        leftEncPrev = leftEncCurr;
        rightEncPrev = rightEncCurr;
    }
    Serial.println("Stopping rotation");
    motors.setSpeeds(0, 0);
}

void driveDistance(Direction dir, int mm) {
    motors.setSpeeds(dir * BASE_SPEED_LEFT, dir * BASE_SPEED_RIGHT);
    startTimer = millis();
    endTimer = startTimer + STALL_CHECK_INTERVAL + 1000;
    leftEncGoal = leftEnc.read() + (dir * (int)((float)mm * TICKS_PER_MM));
    rightEncGoal = rightEnc.read() + (dir * (int)((float)mm * TICKS_PER_MM));
    leftEncPrev = leftEnc.read();
    rightEncPrev = rightEnc.read();
}
while (1) {
    startTimer = millis();
    long leftEncCurr = leftEnc.read();
    long rightEncCurr = rightEnc.read();

    if (startTimer >= endTimer) {
        if (abs(leftEnc.read() - leftEncPrev) < STALL_THRESHOLD
            || abs(rightEnc.read() - rightEncPrev) < STALL_THRESHOLD) {
            break;
        } else {
            endTimer = startTimer + STALL_CHECK_INTERVAL;
            leftEncPrev = leftEnc.read();
            rightEncPrev = rightEnc.read();
        }
    }

    if (dir == FORWARDS && (leftEncCurr >= leftEncGoal
       || rightEncCurr >= rightEncGoal)) {
        break;
    }

    if (dir == BACKWARDS && (leftEncCurr <= leftEncGoal
       || rightEncCurr <= rightEncGoal)) {
        break;
    }

    readLineSensors();

    int line_error = getLineError();

    if (dir == FORWARDS) {
        motors.setSpeeds(BASE_SPEED_LEFT + (LINE_KP * line_error),
                         BASE_SPEED_RIGHT - (LINE_KP * line_error));
    }

    motors.setSpeeds(0, 0);
}

void rotateDistance(Rotation rot, int deg) {
    motors.setSpeeds(rot * BASE_SPEED_LEFT, -1 * rot * BASE_SPEED_RIGHT);
    startTimer = millis();
    endTimer = startTimer + ENC_CHECK_INTERVAL;
    leftEncGoal = leftEnc.read() + (rot *
       (int)((float)deg * TICKS_PER_DEG - STOPPING_COMP));
    rightEncGoal = rightEnc.read() + (-1 * rot *
       (int)((float)deg * TICKS_PER_DEG - STOPPING_COMP));
    leftEncPrev = leftEnc.read();
}
rightEncPrev = rightEnc.read();

while (1) {
    startTimer = millis();
    long leftEncCurr = leftEnc.read();
    long rightEncCurr = rightEnc.read();

    if (rot == CW && (leftEncCurr >= leftEncGoal
         || rightEncCurr < rightEncGoal)) {
        break;
    }
    else if (rot == CCW && (leftEncCurr < leftEncGoal
         || rightEncCurr >= rightEncGoal)) {
        break;
    }

    if (startTimer >= endTimer) {
        long lDiff = abs(leftEncCurr - leftEncPrev);
        long rDiff = abs(rightEncCurr - rightEncPrev);
        if (lDiff > rDiff) {
            motors.setSpeeds(rot * (BASE_SPEED_LEFT -
                                (ENC_KP * lDiff / rDiff)), -1 * rot * BASE_SPEED_RIGHT);
        }
        else if (rDiff > lDiff) {
            motors.setSpeeds(rot * BASE_SPEED_LEFT,
                             -1 * rot * (BASE_SPEED_RIGHT - (ENC_KP * rDiff / lDiff)));
        }
        endTimer = startTimer + ENC_CHECK_INTERVAL;
        leftEncPrev = leftEncCurr;
        rightEncPrev = rightEncCurr;
    }
    motors.setSpeeds(0, 0);
}

void driveStraight(int mm) {
    motors.setSpeeds(BASE_SPEED_LEFT, BASE_SPEED_RIGHT);
    startTimer = millis();
    endTimer = startTimer + STALL_CHECK_INTERVAL + 1000;
    leftEncGoal = leftEnc.read() + ((int)((float)mm * TICKS_PER_MM));
    rightEncGoal = rightEnc.read() + ((int)((float)mm * TICKS_PER_MM));
    leftEncPrev = leftEnc.read();
    rightEncPrev = rightEnc.read();

    while (1) {
        startTimer = millis();
        long leftEncCurr = leftEnc.read();
        long rightEncCurr = rightEnc.read();

        if (leftEncCurr >= leftEncGoal
             || rightEncCurr < rightEncGoal) {
            break;
        }
        else if (leftEncCurr < leftEncGoal
             || rightEncCurr >= rightEncGoal) {
            break;
        }

        if (startTimer >= endTimer) {
            long lDiff = abs(leftEncCurr - leftEncPrev);
            long rDiff = abs(rightEncCurr - rightEncPrev);
            if (lDiff > rDiff) {
                motors.setSpeeds(rot * (BASE_SPEED_LEFT -
                                    (ENC_KP * lDiff / rDiff)), -1 * rot * BASE_SPEED_RIGHT);
            }
            else if (rDiff > lDiff) {
                motors.setSpeeds(rot * BASE_SPEED_LEFT,
                                 -1 * rot * (BASE_SPEED_RIGHT - (ENC_KP * rDiff / lDiff)));
            }
            endTimer = startTimer + ENC_CHECK_INTERVAL;
            leftEncPrev = leftEncCurr;
            rightEncPrev = rightEncCurr;
        }
        motors.setSpeeds(0, 0);
    }
}
if (startTimer >= endTimer) {
    if (abs(leftEnc.read() - leftEncPrev) < STALL_THRESHOLD
        || abs(rightEnc.read() - rightEncPrev) < STALL_THRESHOLD) {
        break;
    } else {
        endTimer = startTimer + STALL_CHECK_INTERVAL;
        leftEncPrev = leftEnc.read();
        rightEncPrev = rightEnc.read();
    }
}

if (leftEncCurr >= leftEncGoal || rightEncCurr >= rightEncGoal) {
    break;
}

motors.setSpeeds(0, 0);