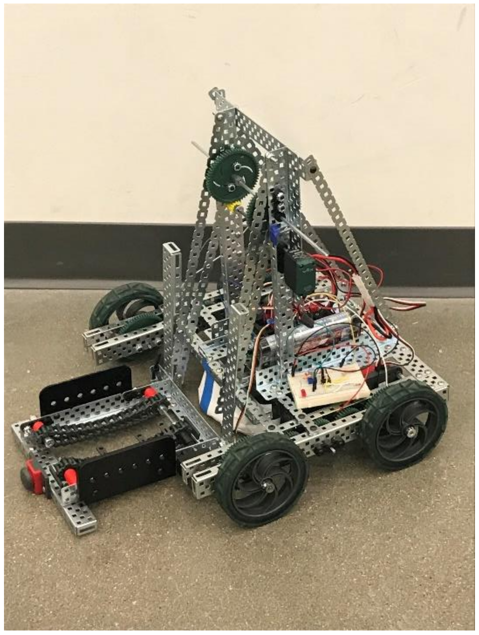
RBE 1001 C19, *Introduction to Robotics*
Final Project Report



Final Robot

"Jawbone"

Team 16: The Pikes

Member	Signature	Contribution (%)
Liam Costello	____Liam Costello______________	____30________
Conrad Tulig	____Conrad Tulig______________	____60________
Joey Wolfgang	____Joey Wolfgang___________ ___	____10________

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

Table of Contents

List of Figures

Introduction

Transporting pizzas will be a difficult challenge for a robot. It will have to overcome a dangerous speed bump, and be able to lift those pizzas several stories high. This will mean that the robot will need to be stable enough to go over this bump, as well as being able to transport pizzas and place them at various heights. The robot can also move Gompei; a heavy, green-colored wooden construction, to the friendly side of the field to score more points. The robot can also suspend itself from a 12'' pole at the end of the run in an Aerial Delivery.

The field is on an 8' x 12' field, with four college "dorms," two with 4 stories and two with 5 stories. The field is covered by a carpet with white tape lines on it: potential line-following guides. Additionally, there are two speed bumps, one on each side of the field. The speed bumps separate the field from the Construction Zone, which is where Gompei is located.

Overall, most strategies in this competition involve robots delivering pizzas, as it is the most consistent way to score points. Afterwards, the endgame strategy between Gompei and Aerial Delivery is largely open ended; one could attempt either objective or try to complete both. Programming the robot's autonomous function to deliver pizzas will be more difficult, as the height of each floor, the dorm, the path to the dorm, and the delivery system all have to be taken into account.

Preliminary Discussion

To start the brainstorming process for our competition strategy, we first began by analyzing the rules and learning about the variety of ways to score points. Based off our analysis, we concluded that the most optimal course of action would be to make Happy Dorms, allowing us to demonstrate understanding of the core mechanical, electrical, and programming fundamentals. We decided we would not pursue the points from Gompei, as we wanted to put more emphasis into developing a complex pizza delivery arm over focusing on pushing a heavy object.

We've devised two viable autonomous mode options. The first option is to carry one pizza into the construction zone and drop it off in the second floor of Faraday, at which point the robot can stay parked in the construction zone until the 20 seconds are up. Getting to Faraday will require following a white line over a speed bump and detecting the turn to Faraday to drop off the pizza. Option 2 involves delivering as many pizzas as possible to different dorm floors. In 20 seconds, this could potentially be 2 pizzas, one in the 2nd and 3rd floors of Faraday. By not going over the speed bump and instead directly back and forth between a pizzeria and Faraday on the outside, depending on how long the arm motor takes to move to each floor it can work. This would maximize the number of points our robot would receive, and get us closer to filling a dorm. However, we are still leaning towards the first option as the risk of only getting 1 pizza might not be worth a couple extra points.

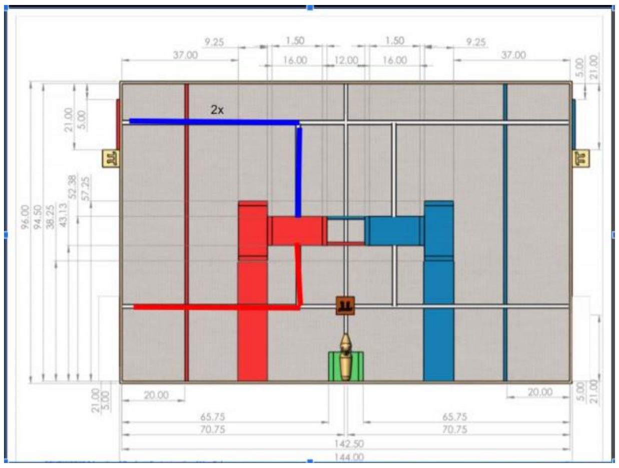


Figure 1.1 - Competition Board Strategies

After the autonomous mode, we would move on to complete the Happy Dorms of both Faraday and Messenger in RC mode on the outside of the buildings, not in the construction zone. After this is complete, and if time permits, we would move to hang on the bar. The time it would take to accomplish this would depend on what side of the board we are on.

Problem Statement

Design a robot to pick up and transport pizzas to dorm levels of various height, and be able to drive over a raised platform.

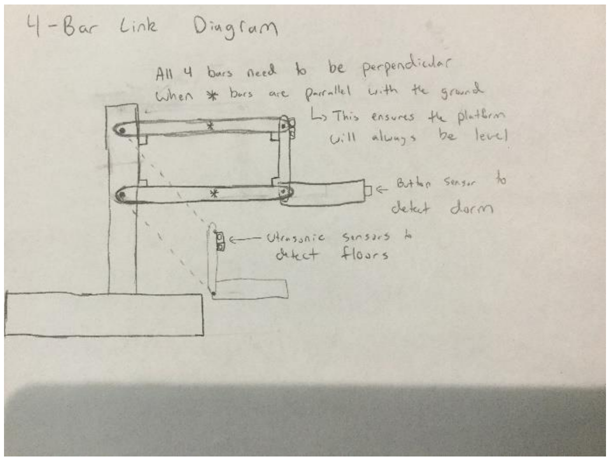
Design Specifications:

- Delivery Mechanism Priorities - These are design objectives our delivery mechanism for the pizza has to achieve.
    - Be able to transport a pizza from pizzeria to any of the dorms.
    - Identify levels of a dorm using ultrasonic sensors.
    - Deliver a pizza to each level of a dorm.
    - Utilize a push button to automatically deliver pizzas to dorm levels.
- Lift Priorities - These are design objectives for the four-bar linkage lift system for the delivery mechanism.
    - Lift should be able to raise both the arm and a pizza to appropriate heights, while keeping the delivery mechanism and pizza level.
    - Lift should utilize a switch sensor for maximum and minimum height detection.
    - Lift should raise slowly enough that the ultrasonic sensor can detect the different levels of the dorm, yet quickly enough to be time-efficient.
    - Lift will utilize a Potentiometer for proportional control.
    - Optional: the lift could be used to hang the robot from the hang bar.
- Chassis Priorities - These are design objectives our chassis has to achieve in order for the autonomous and RC modes of the competition to be successful.
    - Be able to drive via RC.
    - Be able to drive over a speed bump.

- Weight cannot exceed 10 lbs.
- Dimensions cannot exceed 15.25” x 15.25” x 18”
- Line follow during autonomous mode.
- Stable, straight driving during RC and autonomous modes.
- Chassis will utilize two encoders placed on the rear wheels to better control movement.

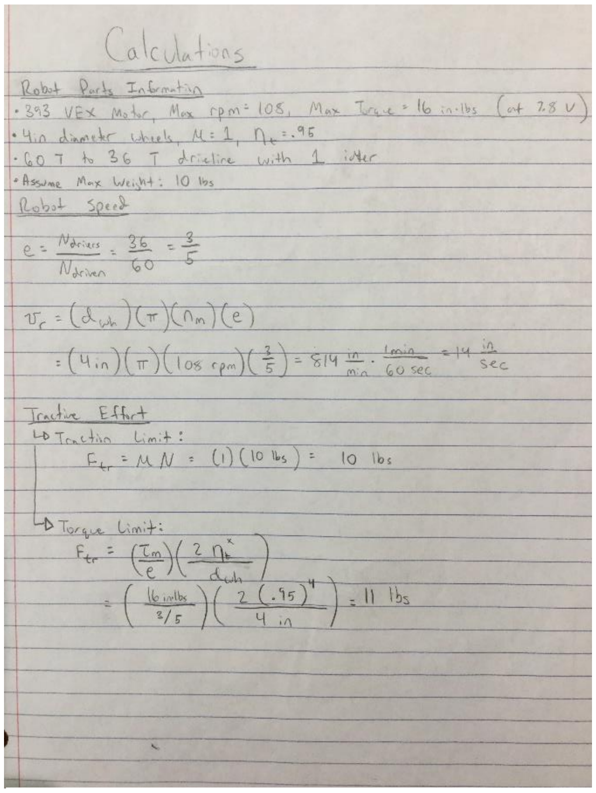
Preliminary Design

With these scoring options and challenges in mind, we decided on a design focused around making Happy Dorms. With the arm being the most important aspect of our robot, it needed to be accurate and consistent in its operation. In order to achieve a Happy Dorm, the robot must be able to precisely place a pizza in each floor. Starting with the designs for the delivery mechanisms, we quickly ruled out any methods of delivering the pizza through sliding. So, we ended up with two options involving delivering a pizza off a constantly level surface. The first option involved using two solenoids to launch the pizza forward off the platform and into the floor. While, they are great for their quick bursts of power to quickly push the pizza, but also seemed less controllable as the second option. With a lot more than 2 points of contact, we are leaning towards a conveyor belt delivery as it has a more consistent delivery system and still relatively fast speed depending on the sprocket size used, with the main disadvantage being heavier than the solenoid system with the entirety of the motor, sprockets, belt, and axles.


4-Bar Link Diagram
All 4 bars need to be perpendicular
when * bars are parallel with the ground
↳ This ensures the platform will always be level
*
*
Button Sensor to detect dorm
Ultrasonic Sensors to detect floors
Figure 2.1 - Four-Bar Linkage

For the lifting mechanism, our team unanimously decided on one 4 bar linkage design to raise the arm up and down whilst keeping the delivery platform level. The sensors to be used on adjusting the motion of this arm however were debated upon. While we initially planned to use our custom circuit on line following like in the labs, we had the idea to make a custom circuit to help count what floor the arm is on using ultrasonic sensors and op amps. The current floor number can then be used in conjunction with a potentiometer to employ proportional control on the arm motor. This would allow the arm speed to be set relative to how many floors away the arm is from the target floor. This custom circuit can also be used in conjunction with the RC portion to make it easier for the driver for the driver as by the driver could just press a button to raise the arm up a floor instead of precisely moving the arm up manually.

Finally, the decisions for the chassis mainly involved choices for wheel sizes, gear ratios, and how many motors to drive the robot. With our two possible strategies of either going for more pizzas at a faster speed or going for less pizzas but delivering the pizzas from inside the construction zone. The first strategy would involve a speed gear ratio to maximize the speed of the robot. The construction zone strategy would involve larger wheels to help climb the speed bump and torque gear ratios so the motors have the necessary torque to climb up the speed bump. We initially considered a 4-wheel drive for our robot using 4 motors as the double power output would have been fairly helpful, but due to limitations on the robot's battery this is probably not feasible. Our secondary design keeps the 4-wheel drive concept, but



Calculations

Robot Parts Information

- 393 VEX Motor, Max rpm = 108, Max Torque = 16 in·lbs (at 7.8 V)
- 4in diameter wheels, $\mu = 1$, $\eta_t = .95$
- 60 T to 36 T driveline with 1 idler
- Assume Max Weight: 10 lbs

Robot Speed

$$e = \frac{N_{drivers}}{N_{driven}} = \frac{36}{60} = \frac{3}{5}$$

$$v_r = (d_{wh})(\pi)(n_m)(e)$$

$$= (4\,in)(\pi)(108\,rpm)\left(\frac{3}{5}\right) = 814\,\frac{in}{min} \cdot \frac{1\,min}{60\,sec} = 14\,\frac{in}{sec}$$

Tractive Effort

↳ Traction Limit:

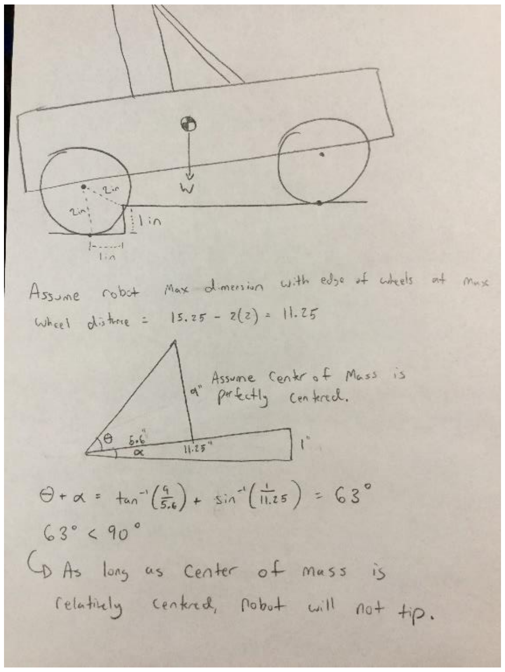
$$F_{tr} = \mu N = (1)(10\,lbs) = 10\,lbs$$

↳ Torque Limit:

$$F_{tr} = \left(\frac{\tau_m}{e}\right)\left(\frac{2\,\eta_t^x}{d_{wh}}\right)$$

$$= \left(\frac{16\,in\cdot lbs}{3/5}\right)\left(\frac{2\,(.95)^4}{4\,in}\right) = 11\,lbs$$

Figure 2.2 - Tractive Effort Calculations



Assume robot Max dimension with edge of wheels at Max wheel distance = 15.25 − 2(2) = 11.25

Assume Center of Mass is perfectly Centered.

$$\theta + \alpha = \tan^{-1}\left(\frac{9}{5.6}\right) + \sin^{-1}\left(\frac{1}{11.25}\right) = 63^\circ$$

$$63^\circ < 90^\circ$$

↳ As long as Center of mass is relatively centered, robot will not tip.

Figure 2.3 - Tipping Point Diagram

uses only 2 motors to achieve this through gearing both sides of the motor to the front and back wheels.

Selection of Final Design

Our final design for the robot cemented the different decisions we had to make outlined previously, taking into consideration mechanical analysis and energy efficiency as well as cost and time limitations. Towards this end, we shortened the height of the tower that the arm was connected too. This was due to the realization that the arm and load length combined would be too long as currently designed unless we designate length as our 18” dimension. The tower had to be lowered because at the time, height was our 18” dimension and the tower was around 17.25” in height. Additionally, we committed to the 2-motor, 4-wheel-drive design for the drivetrain. This ended up proving to be very costly in terms of materials; 20 motor bearings were used to ensure stable axle rotation, and 18 shaft collars were used to hold all the axles, wheels, and gears in place. However, this setup proved to be very efficient in energy consumption compared to a 4-motor setup. The setup also provides enough power to go over the speed bump, the one major obstacle on the board. 4 motors would provide far more power than what was required, would have costed more in Robo-bucks and dragged more on the battery.

The delivery mechanism for the arm has also gone through a few changes. After the initial decision to go forward with the conveyor belt delivery mechanism, the idea was put forth to try and 3d print the frame for the mechanism. With a 3d printed frame, the design could be as open ended as we wanted, it would save a lot on weight, and reduce the strain on the lift motor of the arm. However, we did not want to spend the time prototyping a mechanism frame; if something was wrong with it, it'd have to be scrapped and a new one would have to be printed. After the reduction of the tower height, we decided to build a mechanism and frame out of Vex parts. This proved to be faster and ultimately just as accurate as a 3d printed version.

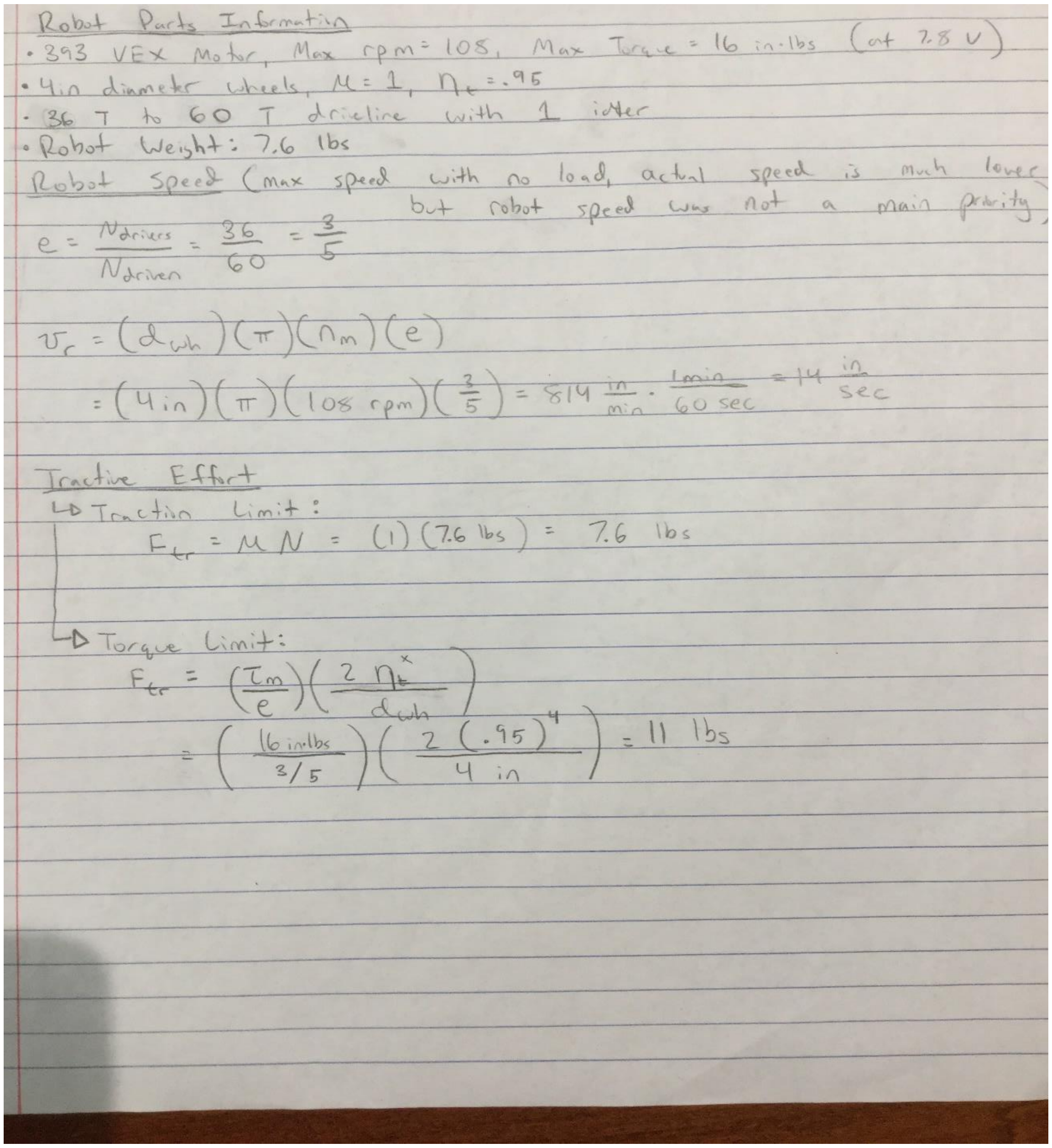
Another item on the robot that changed a few times was the light sensors on the bottom of the robot. They started out in a tight formation adjacent to one another. The programming-savvy members of our team felt that the closer spread would be more accurate and faster at catching subtle changes in the robots trajectory. This turned out to not be the case, and they were slowly spread apart to encompass a wider area. Along with the changes to width, the light sensors were also lowered to be closer to the ground, and a paper skirt was added to help them better differentiate the multiple line colors of the board.

The custom circuit hypothesized by the team was supposed to be 2 ultrasonic sensors that can differentiate between different ranges of distances, and, using 2 op amps, create a varying voltage that could allow a program to essentially "count the floors" of the dorms. Due to technical difficulties described in the circuit analysis, we were not able to utilize the sensor.

Final Design Analysis

Mechanical Analysis:

Beginning with the analysis of the chassis, we began by deciding upon the gear ratio to be used for the robot. Initially we did calculations assuming the robot would be at 10 lbs, and decided to keep the same 36 T to 60T gear train as this should work as long as the robot stays below this weight limit. While the efficiency of the drivetrain decreased due to using 4 wheel drive with only 2 motors (due to an idler gear used to drive the gears to the front and back wheels), the weight also decreased enough from 10 lbs to 7.6 lbs to compensate. Below are the new robot speed and tractive effort calculations:



Robot Parts Information

- 393 VEX Motor, Max rpm = 108, Max Torque = 16 in·lbs (at 7.8 V)
- 4in diameter wheels, $\mu = 1$, $\eta_t = .95$
- 36 T to 60 T driveline with 1 idler
- Robot Weight: 7.6 lbs

Robot Speed (max speed with no load, actual speed is much lower but robot speed was not a main priority)

$$e = \frac{N_{drivers}}{N_{driven}} = \frac{36}{60} = \frac{3}{5}$$

$$v_r = (d_{wh})(\pi)(n_m)(e)$$

$$= (4\,in)(\pi)(108\,rpm)\left(\frac{3}{5}\right) = 814\,\frac{in}{min} \cdot \frac{1\,min}{60\,sec} = 14\,\frac{in}{sec}$$

Tractive Effort

↳ Traction Limit:

$$F_{tr} = \mu N = (1)(7.6\,lbs) = 7.6\,lbs$$

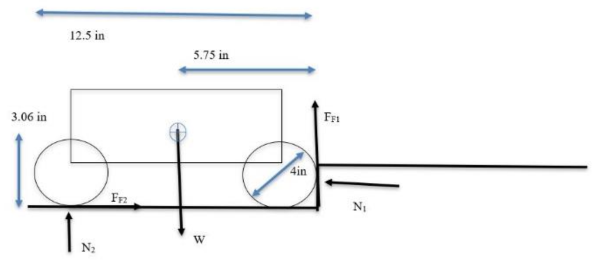
↳ Torque Limit:

$$F_{tr} = \left(\frac{\tau_m}{e}\right)\left(\frac{2\,\eta_t^x}{d_{wh}}\right)$$

$$= \left(\frac{16\,in{\cdot}lbs}{3/5}\right)\left(\frac{2\,(.95)^4}{4\,in}\right) = 11\,lbs$$

Figure 3.1 Tractive Effort Calculations

To calculate the Force of Friction required to overcome the speedbump, one must calculate the sum of x and y forces, as well as the sum of moments M_z, the robot experiences both before and after its first wheel crests the lip of the ridge.



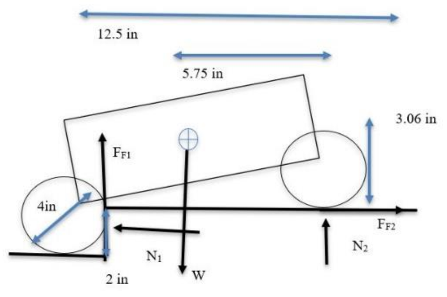
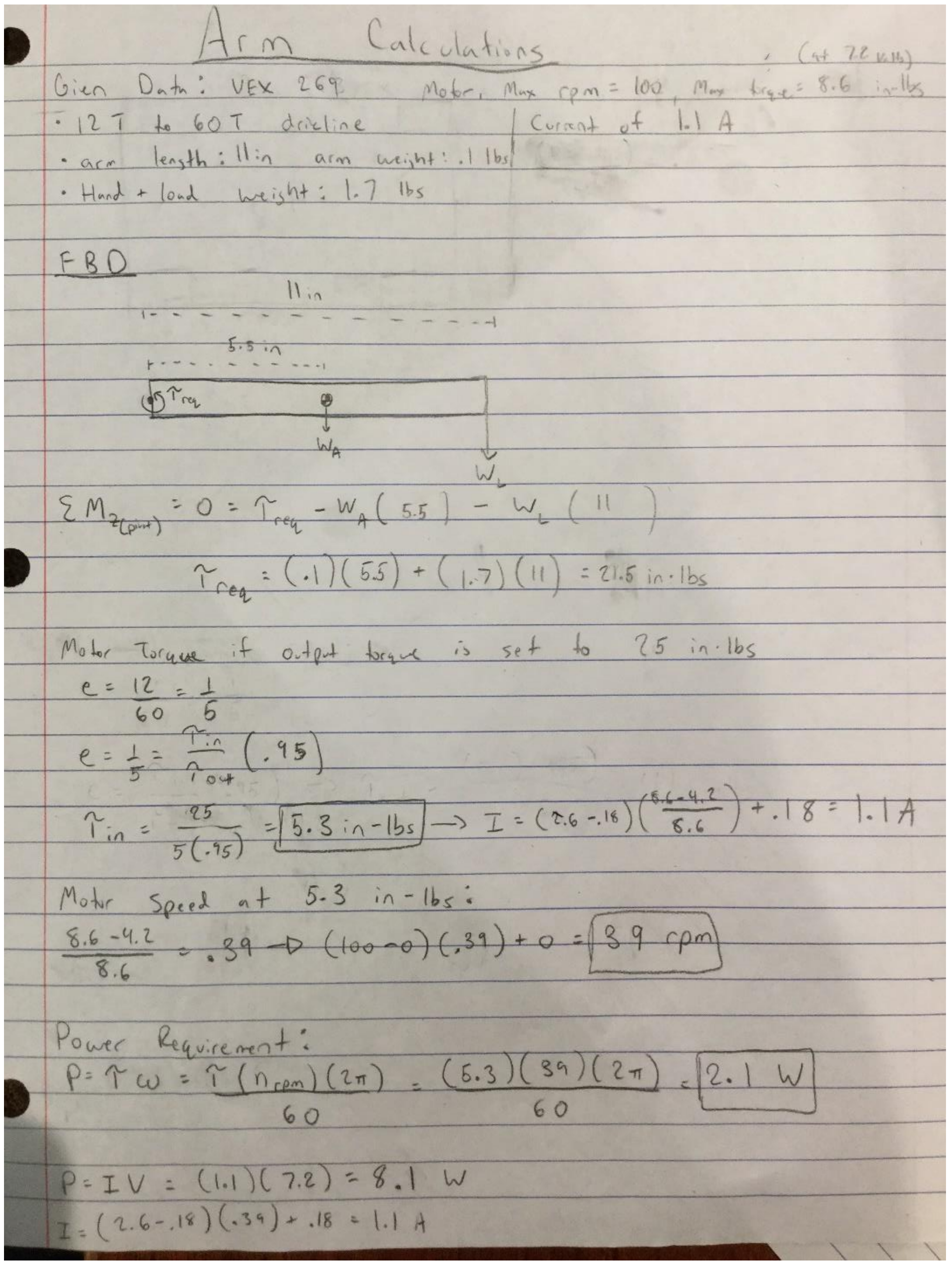


Figure 3.2 Speedbump Analysis 1

Figure 3.3 Speedbump Analysis 2

With a weight of 7.6 lbs, this number comes out to be 3.8 inch-pounds of force.

Moving on to the mechanical analysis of the arm, the final arm with the load came out to be relatively heavy at a total of 1.8 lbs, however at a current of 1.1 A, the 12 T to 60 T gear train provided enough torque along with the 269 VEX motor to lift the arm. In our calculations, we assumed the arm to have an evenly distributed weight with the delivery system and pizza at the end of the arm to be a point force load. As with our choice of being energy efficient with 2 motor drive over 4, we also assumed a low current of 1.1 A for the lift, which ended up to provide more than enough torque needed.



Arm Calculations

Gien Data: VEX 269 Motor, Max rpm = 100, Max torque = 8.6 in-lbs (at 7.2 Volts)

- 12 T to 60 T drieline | Current of 1.1 A
- arm length: 11 in arm weight: .1 lbs
- Hand + load weight: 1.7 lbs

FBD

11 in
5.5 in
τ_{req}
W_A
W_L

$$\Sigma M_{z(pivot)} = 0 = \tau_{req} - W_A(5.5) - W_L(11)$$

$$\tau_{req} = (.1)(5.5) + (1.7)(11) = 21.5 \text{ in}\cdot\text{lbs}$$

Motor Torque if output torque is set to 25 in·lbs

$$e = \frac{12}{60} = \frac{1}{5}$$

$$e = \frac{1}{5} = \frac{\tau_{in}}{\tau_{out}}(.95)$$

$$\tau_{in} = \frac{25}{5(.95)} = \boxed{5.3 \text{ in-lbs}} \longrightarrow I = (2.6 - .18)\left(\frac{8.6-4.2}{8.6}\right) + .18 = 1.1 A$$

Motor Speed at 5.3 in-lbs:

$$\frac{8.6-4.2}{8.6} = .39 \rightarrow (100-0)(.39) + 0 = \boxed{39 \text{ rpm}}$$

Power Requirement:

$$P = \tau\omega = \frac{\tau(n_{rpm})(2\pi)}{60} = \frac{(5.3)(39)(2\pi)}{60} = \boxed{2.1 \text{ W}}$$

$$P = IV = (1.1)(7.2) = 8.1 \text{ W}$$

$$I = (2.6 - .18)(.39) + .18 = 1.1 A$$

Figure 3.4 Arm Calculations

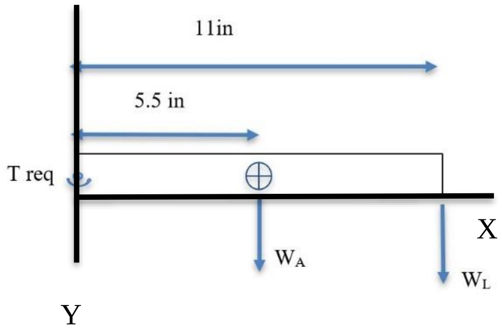
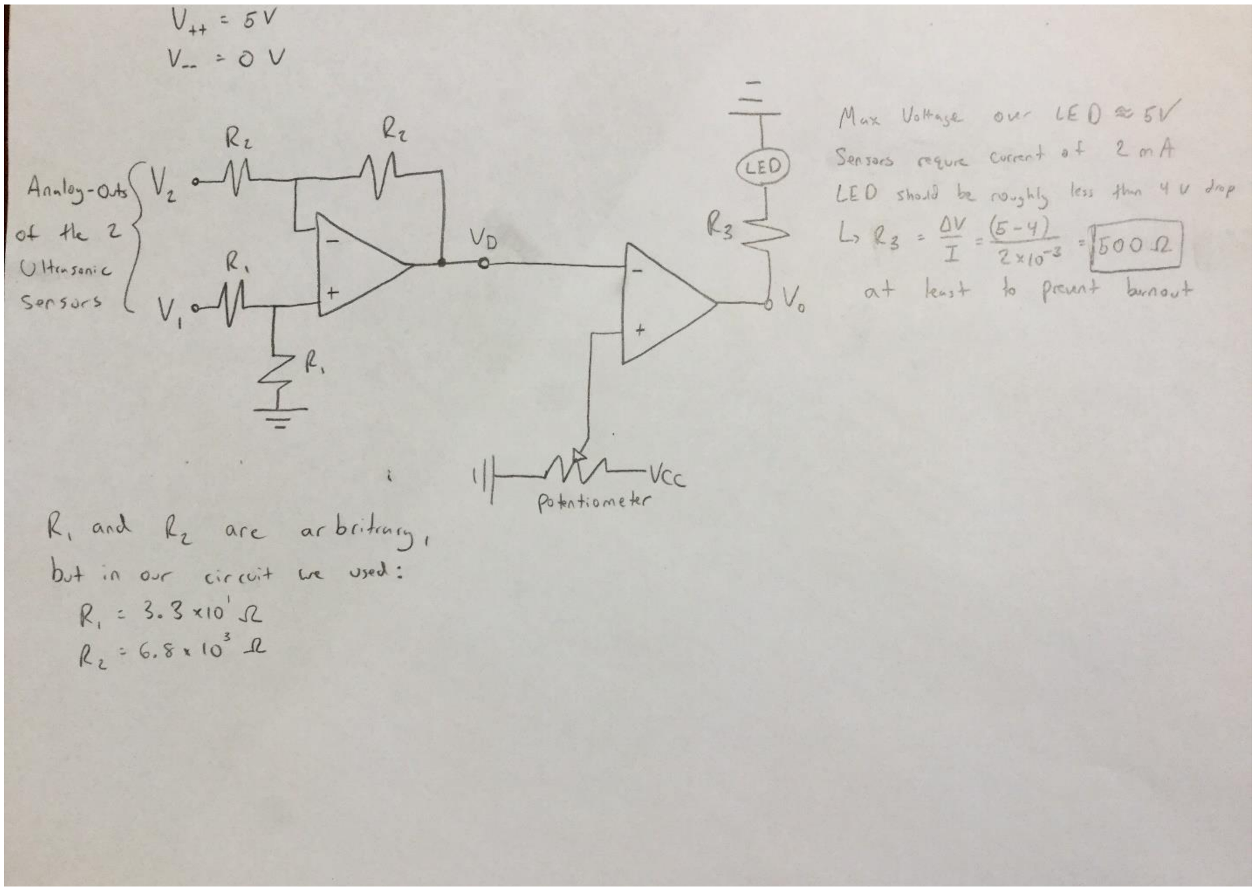


Figure 3.5 Arm FBD

Circuit and Sensors Analysis:

Our plan for the custom-built circuit involved using 2 ultrasonic sensors to detect the arm reaching a new floor of a building. To accomplish this, we used what we theorized would be a voltage output range from the analog out pin where if the sensor read a far distance it would output 5V down to 0V if the sensor read something very close. Using this assumption, the two analog out pins were inputted into the first op amp with specific resistors setup as a voltage difference computation by using a feedback loop. This voltage difference output was then inputted into a second op amp setup as a comparator where the voltage difference would be compared to a controlled voltage from a potentiometer. So if the voltage difference is higher than controlled voltage, the comparator would output a digital value of 1 for the Arduino to read. This would essentially occur when the top ultrasonic sensor read a very close object (the floor), versus the bottom sensor still reading a far distance (between floors). Unfortunately, after making the circuit we found that the analog output pin as output a constant voltage of about 3.2 V regardless of the distance in front of it. After more research we found that the assumption that the ultrasonic



$V_{++} = 5V$
$V_{--} = 0V$
Analog-Outs of the 2 Ultrasonic Sensors
V_2
V_1
R_2
R_2
R_1
R_1
V_D
LED
R_3
V_o
Vcc
Potentiometer
Max Voltage over LED ≈ 5V
Sensors require current of 2 mA
LED should be roughly less than 4 V drop
$\hookrightarrow R_3 = \frac{\Delta V}{I} = \frac{(5-4)}{2\times10^{-3}} = \boxed{500\,\Omega}$
at least to prevent burnout
R_1 and R_2 are arbitrary, but in our circuit we used:
$R_1 = 3.3\times10^1\,\Omega$
$R_2 = 6.8\times10^3\,\Omega$

Figure 3.6 Circuit Design

sensors' analog output pin varied the output voltage was incorrect, and we would have needed an analog to voltage converter to make the circuit functional.

Besides the two ultrasonic sensors, plenty of other sensors were used throughout the robot to help aid in the autonomous program. Starting with the chassis, the primary form of straight movement was aided by three VEX light sensors. These were placed between the 2 sets of wheels, but relatively in front the robot's virtual turning center to ensure the robot can make turning adjustments in the correct direction. The wheels themselves have encoders on their back wheels which aid in precise turning of the robot. The lifting mechanism uses a potentiometer to allow for proportional control of the arm. Finally, on the delivery mechanism itself there is a button sensor at the front of the conveyor belt to detect that the floor is close enough to deliver pizza. The implementation for most of these sensors are further described in the programming analysis.

Programming Analysis:

The overall organization of the final code was a multifile, class based program making use of the timed autonomous and remote control modes template. The RC portion used the premade class from the template with some added remote control functions. For the autonomous portion, an Auto class was used to make and run an instantiated Arm and Chassis class in a state machine based program. While there were only 1 arm and 1 chassis on this robot, this type of object based programming is good practice to allow for expandability of the code. The Auto class also consists of 2 main functions for the blue or red side of the field that are called depending on a jumper being plugged into a digital port or not. The Chassis class used two notable functions for the line following and turning of the robot. Since we decided on using the

VEX light sensors, we were able to zero the light sensors at the start of each program by having the middle sensor average the white line values and the side sensors average the board color values. Using these averaged thresholds, the robot could then navigate the lines by keeping the middle sensor on the white line. As for turning, encoders where used on the wheels to make sure each wheel turned a given amount before stopping. Finally, the Arm class notably contains 2 proportional control functions for the lifting mechanism. While we did have a proportional control function for lifting the arm to a target floor based off the current floor using the ultrasonic sensor circuit, that unfortunately was not able to be used. However, we still have a potentiometer based proportional control that raises the arm to a target arm position, slowing the arm down as it reaches the target angle.

Summary / Evaluation

While the robot was not the most outperforming statistically due to some setbacks, we more importantly learned allot on applying our the mechanical, electrical, and programming concepts that we learned and built up. For the mechanical aspect, our robot had a well-designed chassis with a well thought out drive train that in practice suited our needs. The 4-wheel drive was a great decision in allowing us to make accurate turns in place as well as a general increase in control over the maneuverability of the robot. In hindsight, omni wheels may have been a better choice for even easier stationary turning, but the normal wheels worked well enough for their role as long as the robot did not turn at top speed. The slightly more torque-based drivetrain both helped alleviate this issue as well as give us enough torque effort to prevent stalling. As for the mechanical design of the arm, we knew the stability of the arm was going to be lowered than desired due to using up most of our shafts and shaft collars on the wheel drivetrain. This did not prove to be an issue however for its intended functions and loads, and only being prone to breaking when falling over. The main takeaway from the arm mechanically however was the gear ratio. While we did calculate that our gear ratio was enough to move the arm up and down, and in practice it technically was enough, the arm moved too quickly for us to manually control accurately with the remote. Another set of 12 T to 60 T gears would have been very helpful in slowing down the arm to make it more controllable.

Moving on to the electrical and sensors aspect of the robot, while the custom circuit did not work in practice, the idea behind it was correctly thought out. We could have easily used our working line follow circuit from the lab, but we wanted to test our knowledge from what we learned in class and make an entirely new circuit of our own design. Despite it not working due to unforeseen functionality of the sensor's analog output, there was definitely a learning experience to be taken from the creation and debugging of the circuit.

The sensors and their implementation in the program were the highlight of robots design. Since we weren't using photoresistors for line following, we were able to use VEX line follower sensors which allowed us to make use of the ability to zero them at the start of every program. A common frustration we had with the photoresistors in the labs were that depending on the lighting in the room, they read a large range of varying values, and by being to zero them completely eliminated this issue. For the next well implemented sensor, a potentiometer used with the arm motor was extremely useful when paired with proportional control for the robot to autonomously bring the arm to the same height consistently despite us not being able to do it manually. Given more time, we would have liked to apply this useful proportional control concept to the wheels as well. While they did use the encoders for turning until a given encoder value was reached, applying proportional control to them would have greatly aided in making the motors also spin at the same speed in the process.

As for the program design as a whole, the multifile, class-based programming method proved to be a very useful way to make the code much more organized and readable. In addition, developing a state machine based autonomous program instead of a time and while loop locked program made it much easier to execute certain function while checking the sensors for changes in the environment.

Our robot had several positive and strong aspects to its design, but due to a few unfortunate oversights, it did not perform as well as we'd hoped in the CDR. Despite this, we believe that we have been able to use these mistakes as lessons to learn how to properly apply our mechanical, electrical, and programming knowledge to the project.

Appendix

Complete program documentation:

File: RBE_1101_Final_Auto__RC

```
/* This is the RBE 1001 Template as of
 *
 * 3/28/17
 *
 * This Template
 * is designed to run the autonomous and teleop sections of the final
 * competition. Write and test your autonomous and teleop code on your
 * own and place the code in auto.cpp or teleop.cpp respectively.
 * The functions will be called by the competition framework based on the
 * time and start button. DO NOT change this file, your code will be called
 * by the framework. The framework will pass your code a reference to the DFW
 * object as well as the amount of MS remaining.
 */
#include <DFW.h>
#include "MyRobot.h"



MyRobot myRobotInstance;
DFW dfw(&myRobotInstance); // 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"
		myRobotInstance.dfw=&dfw;// add a reference to the controller to your robot. this lets you
control your robot
  myRobotInstance.robotStartup();
}
void loop() {
		dfw.run();
}
```

File: MyRobt.h

```
#pragma once

#include "Servo.h"
#include "Auto.h"
#include <DFW.h>
#include <AbstractDFWRobot.h>


class MyRobot : public AbstractDFWRobot {
  public:
    Auto autoMode;
    DFW * dfw;
    int side;
    int sidePin;
    /**
       Called when the start button is pressed and the robot control begins
    */
    void robotStartup();
    /**
       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);
    /**
       Called at the end of control to reset the objects for the next start
    */
    void robotShutdown(void);
    /**
       Return the number of the LED used for controller signaling
    */
    int getDebugLEDPin(void) {
      return 13;
    };

    ~MyRobot() {};

};
```

File: MyRobot.cpp

```
#include "MyRobot.h"
#include "Arduino.h"


/**
   Called when the start button is pressed and the robot control begins
*/
void MyRobot::robotStartup() {
  Serial.println("Here is where I start up my robot code");

  sidePin = 28;
  pinMode(sidePin, INPUT_PULLUP);

  //Determines what side the robot is on based on a digital jumper wire
  if (digitalRead(sidePin)) {
    side = 0;
  }
  else {
    side = 1;
  }

  autoMode.initialize();
}
/**
   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 MyRobot::autonomous( long time) {
  Serial.print("\r\nAuto time remaining: ");
  Serial.print(time);
  if (side == 1) {
    autoMode.doAutoMode1();
  }
  else {
    autoMode.doAutoMode2();
  }
}
/**
   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 MyRobot::teleop( long time) {
  if (dfw->getCompetitionState() != powerup) {
    autoMode.chassis.rightMotor.write(180 - dfw->joystickrv());   //DFW.joystick will return 0-180 as an
int into rightmotor.write
    autoMode.chassis.leftMotor.write(dfw->joysticklv());      //DFW.joystick will return 0-180 as an int
into leftmotor.write
    if (dfw->r1()) {
      //Serial.println("r1");
      autoMode.arm.armMotor.write(180); //Moves arm up
    }
    else if (dfw->r2()) {
      //Serial.println("r2");
      autoMode.arm.armMotor.write(70);  //Moves arm down
    }
    else {
      autoMode.arm.armMotor.write(90);  //Holds arm
    }
  }
  if (autoMode.arm.buttonPressed()) {
    autoMode.arm.deliverPizza(140);     //Runs pizza dilvery conveyor belt (based off arm button, not
remote control)
  }
  else {
    autoMode.arm.deliverPizza(90);      //Stops converyor belt
  }
}
/**
   Called at the end of control to reset the objects for the next start
*/
void MyRobot::robotShutdown(void) {
  Serial.println("Here is where I shut down my robot code");

}
```

File: Auto.h

```
#pragma once

#include "Servo.h"
#include "Arm.h"
#include "Chassis.h"
```

```
//Arm class contains the state machines for the 2 different autonomoue programs and holds the
instantiated Arm and Chassis objects
class Auto {
  public:
    enum stateChoices { INITIAL_RAISE_ARM, LINE_FOLLOW_TO_FARADAY,
TURN_TO_FARADAY, RAISE_ARM, DRIVE_UP_TO_DORM, DELIVER_PIZZA, STOP_AUTO}
state;

    void initialize ();
    void doAutoMode1 ();
    void doAutoMode2 ();

    Chassis chassis;
    Arm arm;
};
```

File: Auto.cpp

```
#include "Auto.h"

//Initializes the chassis and arm objects as well as the autonomous state machine
void Auto::initialize () {

  chassis.initialize(10, 11, A9, A10, A11);
  arm.initialize(8, 7, A3, 27, 26);

  state = INITIAL_RAISE_ARM;
}

//Blue side autonomous state machine
void Auto::doAutoMode1 () {
  switch (state) {

    //Raises arm until it is at the level of the 3rd floor (45 deg)
    case INITIAL_RAISE_ARM:
      if (arm.raiseArmToDeg(45)) {
        state = LINE_FOLLOW_TO_FARADAY;
        Serial.println("LINE_FOLLOW_TO_FARADAY");
      }
      break;

    //Follows the white line until the right angle is detected at the left hand turn to Faraday
    case LINE_FOLLOW_TO_FARADAY:
      chassis.followLine();
      if (chassis.detectRightAngle(chassis.whiteLine, chassis.leftLine)) {
        state = TURN_TO_FARADAY;
        Serial.println("TURN_TO_FARADAY");
        chassis.motors(0, 0);
        chassis.resetEncoders();
      }
      break;

    //Turns left to Faraday until it has turned 90 deg (encoder value of 40)
    case TURN_TO_FARADAY:
      if (chassis.turn(1, 40)) {
        chassis.leftGoal = false;
        chassis.rightGoal = false;
        state = DRIVE_UP_TO_DORM;
        chassis.motors(0, 0);
      }
      break;

    /* If the custom circuit worked, arm would have been kept at ground level at the beginning and raised
now.
      case RAISE_ARM:
      if (arm.raiseArmToFloor(3)) {
      state = DRIVE_UP_TO_DORM;
      }
      break;
    */

    //Line follows to Faraday until the button sensor at the end of the arm is triggered
    case DRIVE_UP_TO_DORM:
      chassis.followLine();
      if (arm.buttonPressed()) {
        state = DELIVER_PIZZA;
        chassis.motors(0, 0);
      }
      break;

    //Activates conveyor belt to deliver pizza into Faraday until the Autonomous program is terminated
    case DELIVER_PIZZA:
      arm.deliverPizza(140);
      break;
  }
}

//Red side autonomous state machine
//Essentially the same as the blue side but:
//the line sensors are looking for a 90 degree right turn
//the robot then turns 90 degrees to the right
void Auto::doAutoMode2 () {
  switch (state) {
    case INITIAL_RAISE_ARM:
      if (arm.raiseArmToDeg(45)) {
        state = LINE_FOLLOW_TO_FARADAY;
        Serial.println("LINE_FOLLOW_TO_FARADAY");
      }
      break;
    case LINE_FOLLOW_TO_FARADAY:
      chassis.followLine();
      if (chassis.detectRightAngle(chassis.whiteLine, chassis.rightLine)) {
        state = TURN_TO_FARADAY;
        Serial.println("TURN_TO_FARADAY");
        chassis.resetEncoders();
      }
      break;
    case TURN_TO_FARADAY:
      if (chassis.turn(1, 40)) {
        chassis.leftGoal = false;
        chassis.rightGoal = false;
        state = DRIVE_UP_TO_DORM;
      }
      break;
    /*
      case RAISE_ARM:
      if (arm.raiseArmToFloor(3)) {
      state = DRIVE_UP_TO_DORM;
      }
      break;
    */
    case DRIVE_UP_TO_DORM:
      chassis.followLine();
      if (arm.buttonPressed()) {
        state = DELIVER_PIZZA;
        chassis.motors(0, 0);
      }
      break;
    case DELIVER_PIZZA:
      arm.deliverPizza(140);
      break;
  }
}
```

File: Chassis.h

```
#pragma once

#include "Servo.h"
#include "Encoder.h"

//Chassis class contains all functions, sensors, motors, and variables relating to the chassis
class Chassis {
  private:

    int colorThreshold;
    enum lineStates {ON_LINE, LEFT_OF_LINE, RIGHT_OF_LINE} state;
    int leftEncPos = 0;
    int rightEncPos = 0;
    int targetLeftEncPos;
    int targetRightEncPos;

  public:

    Servo leftMotor;
    Servo rightMotor;

    int leftLine;
    int middleLine;
    int rightLine;

    int whiteLine;
    int boardColor;

    int currentSpeed;

    bool leftGoal;
    bool rightGoal;

    void initialize (int leftPin, int rightPin, int leftLinePin, int middleLinePin, int rightLinePin);
    int zeroSensor(int sensorPin, int times);
    void followLine ();
    bool detectRightAngle(int color, int sideSensor);
    void motors(int leftSpeed, int rightSpeed);
    void drive(int driveSpeed, int dir);
    bool turn(int dir, int deg);
    void resetEncoders();
};
```

File: Chassis.cpp

```
#include "Chassis.h"
#include "Arduino.h"

Encoder leftEnc (22, 23);
Encoder rightEnc (24, 25);

//Initializes all the necessary motors, sensors, and variables for a chassis
void Chassis::initialize(int leftPin, int rightPin, int leftLinePin, int middleLinePin, int rightLinePin) {

  //Drive motor related info
  leftMotor.attach(leftPin, 1000, 2000);
  rightMotor.attach(rightPin, 1000, 2000);
  currentSpeed = 50;
  //Encoder related info
  leftEnc.write(0);
  rightEnc.write(0);
  leftEncPos = 0;
  rightEncPos = 0;
  targetLeftEncPos = 0;
  targetRightEncPos = 0;
  leftGoal = false;
  rightGoal = false;

  //Line follor related info
  leftLine = leftLinePin;
  middleLine = middleLinePin;
  rightLine = rightLinePin;

  pinMode(leftLine, INPUT);
  pinMode(middleLine, INPUT);
  pinMode(rightLine, INPUT);

  //Code assumes the middle line sensors starts on the white line and the left and right line sensors start on
the board color
  state = ON_LINE;
  boardColor = (zeroSensor (leftLine, 10) + zeroSensor (rightLine, 10)) / 2;  //board color averaged
between the left and right sensor averages
  whiteLine = zeroSensor (middleLine, 10);  //white line averaged
  colorThreshold = (boardColor - whiteLine) / 6;  //color threshold to be added onto the white line to catch
slightly higher values

  Serial.print("white line: ");
  Serial.print(whiteLine);
  Serial.print(", board color: ");
  Serial.println(boardColor);
}

//Zeros a sensor value by averaging it a given amount of times
int Chassis::zeroSensor (int sensorPin, int times) {
  int sum = 0;
  for (int i = 0; i < times; i++) {
    sum += analogRead(sensorPin);
  }
  return sum / times;
}

//Line follow function.
//If the middle line detects white, robot goes straight
//If the right line detects white, the robot turns right until the middle line detects white
//If the left line detects white, the robot turns left until the middle line detects white
void Chassis::followLine() {
  int leftColor = analogRead(leftLine);
  int rightColor = analogRead(rightLine);
  int middleColor = analogRead(middleLine);

  if (middleColor <= whiteLine + colorThreshold) {
    state = ON_LINE;
    drive(currentSpeed, 1);
  }
  else if ((leftColor <= whiteLine + colorThreshold) || (state == LEFT_OF_LINE)) {
    state = LEFT_OF_LINE;
    motors(currentSpeed / 2, currentSpeed);
  }
  else if ((rightColor <= whiteLine + colorThreshold) || (state == RIGHT_OF_LINE)) {
    state = RIGHT_OF_LINE;
    motors(currentSpeed, currentSpeed / 2);
  }
  else {
    drive(currentSpeed, 1);
  }
}

//Detects a right angle of a given color and on a given side
//Returns true if a right angle is detected consectively 3 times in a row to eliminate faulty sensor readings
bool Chassis::detectRightAngle(int color, int sideSensor) {
  int sum = 0;
  for (int i = 0; i < 3; i++) {
    int sideColor = analogRead(sideSensor);
    int middleColor = analogRead(middleLine);
    if ((middleColor <= color + colorThreshold) && (sideColor <= color + colorThreshold)) {
      sum++;
    }
  }
  return (sum == 3);
}

//Drives the left and right motors.
//Allows the value of 0 to actually stop the robot
void Chassis::motors(int leftSpeed, int rightSpeed) {
  leftMotor.write(90 + leftSpeed);
  rightMotor.write(90 - rightSpeed);
}

//Drives the robot forwards or backwards at a given
void Chassis::drive(int driveSpeed, int dir) {
  motors(driveSpeed + dir, driveSpeed - dir);
}

//Turns the robot a given direction until both encoders read the given deg value
bool Chassis::turn(int dir, int deg) {
  leftEncPos = leftEnc.read();
  rightEncPos = rightEnc.read();

  Serial.print(leftEncPos);
  Serial.print(", ");
  Serial.print(rightEncPos);


  int turnAmount = deg;
  if (abs(leftEncPos) < turnAmount) {
    leftMotor.write(120 * dir);
  }
  else if (abs(leftEncPos) >= turnAmount) {
    leftGoal = true;
    leftMotor.write(90);
  }

  if (abs(rightEncPos) < turnAmount) {
    rightMotor.write(40 * dir);
  }
  else if (abs(rightEncPos) >= turnAmount) {
    rightGoal = true;
    rightMotor.write(90);
  }

  return (rightGoal && leftGoal);
}

//Resests the encoder positions back to 0
void Chassis::resetEncoders() {
  leftEnc.write(0);
  rightEnc.write(0);
}
```

File: Arm.h

```
#pragma once

#include "Servo.h"

//Arm class contains all functions, sensors, motors, and variables relating to the arm
class Arm {
  private:

    int pot;
    int ultrasonic;
    int button;
    int Kp;
    int currentFloor;
    int lastPotPos;

  public:
    Servo armMotor;
    Servo beltMotor;
    void initialize (unsigned armMotorPin, unsigned beltMotorPin, unsigned potPin, unsigned
ultrasonicPin, unsigned buttonPin);
    bool raiseArmToDeg (int deg);
    bool raiseArmToFloor (int targetFloor);
    void countFloor ();
    bool checkForFloor();
    void deliverPizza (int motorSpeed);
    bool buttonPressed ();

};
```

File: Arm.cpp

```
#include "Arm.h"
#include "Arduino.h"

//Initializes all the necessary motors, sensors, and variables for an arm
void Arm::initialize(unsigned armMotorPin, unsigned beltMotorPin, unsigned potPin, unsigned
ultrasonicPin, unsigned buttonPin) {

  //Motor related info
  armMotor.attach(armMotorPin, 1000, 2000);
  beltMotor.attach(beltMotorPin, 1000, 2000);

  //Sensor related info
  pot = potPin;
  ultrasonic = ultrasonicPin;
  button = buttonPin;

  pinMode(pot, INPUT);
  pinMode(ultrasonic, INPUT);
  pinMode(buttonPin, INPUT_PULLUP);

  //Variables
  Kp = 14;  //proportional control value
  currentFloor = 0;
  lastPotPos = analogRead(pot) * 360 / 1023;

}

//Raises an arm to a specified "degree"
//Arm degrees are from ground position (0) to fully raised (100)
```

```
//Returns true when the arm has reached the target position
bool Arm::raiseArmToDeg(int deg){
  //translates arm positions to 0 - 100 scale
  int armPos = analogRead(pot) - 65;
  int desiredPos = deg * 120 / 100;

  //Proportional Control Error
  int error =  constrain((desiredPos - armPos) * Kp, -90, 90);

  armMotor.write(90 + error);

  return armPos > desiredPos;
}

// *** The next couple functions are using the custom circuit if had worked. Note that because of this the
code is untested. ***

//Raises arm to target floor
//Returns true if the arm has reach the target floor
bool Arm::raiseArmToFloor(int targetFloor) {

  //Porportional Control Error, based on floor difference
  int floorDifference = targetFloor - currentFloor;
  int error = constrain(30 * floorDifference, -90, 90);

  armMotor.write(90 + error);

  return currentFloor == targetFloor;
}

//Counts the floors as the arm passes them.
//If a floor is detected, another floor will not be looked for until the potentiometer has moved 5 deg
//Note: potentiometer was not adjusted to the 0 - 100 scale
void Arm::countFloor() {
  int currentPotPos = analogRead(pot) * 360 / 1023;
  int potDifference = abs(currentPotPos - lastPotPos);

  if(potDifference > 5) {
    if (checkForFloor()) {
      currentFloor ++;
      lastPotPos = analogRead(pot) * 360 / 1023;
    }
  }
}

//Reads the ultrasonic circuit to see if there is a floor
//Floor readings are averaged 5 at a time to eliminate minor misreadings in the sensor
//Returns true if the sensor read a floor at least 4 out of 5 times
bool Arm::checkForFloor() {
  int sum = 0;
  for(int i = 0; i < 5; i++) {
    if (digitalRead(ultrasonic) == 1) {
      sum ++;
    }
  }

  float avg = sum / 5;

  return avg >= 0.8f;
} // *** end of custom circuit functions ***

//Runs the converyor velt to deliver the pizza off the arm.
void Arm::deliverPizza(int motorSpeed) {
  beltMotor.write(motorSpeed);
}

//Checks if the button at the front of the arm is pressed
//Returns true if it has been
bool Arm::buttonPressed() {
  return (digitalRead(button) == 0);
}
```

BOM:

Part	Quantity	Price ($)
Bearing Flat	8	4.00
VEX Chain and Sprocket Kit	1	29.99
36 Tooth Gear	2	0.78
60 Tooth Gear	4	0.04
VEX Light Sensor	3	59.97
VEX 269 Motor	1	1.00
VEX Motor Controller	1	9.99
2” VEX Shaft	6	8.22
6” VEX Shaft	4	4.48
Shaft Collars	7	3.50
MaxBotix Sonar Sensor	2	51.90

VEX 4” Traction Wheel	4	20.00
Total: 192.87		