BarBot – Final Design Report

UNIVERSITY OF WATERLOO

Department of Mechanical and Mechatronics Engineering

A Report Prepared For:

MTE100/MTE121 courses at the University of Waterloo

Prepared By:

Warren Cao – 20993651

Dylan Finlay – 20998986

Marlon Poddalgoda – 20994145

Camron Sabahi-Pourkashani - 21009799

Date: Tuesday, December 6, 2022

Table of Contents

Table of Figures iv

List of Tables v

Summary 1

Acknowledgements 3

Introduction 4

Scope 4

- Measurements and Inputs 5
- Interactions with the environment 6
- Motor usage 6
- Changes in Scope 6

Criteria and Constraints 7

- Criteria 7
- Constraints 8

Mechanical Design and Implementation 9

- Housing of EV3 Brick 9
- Free-Rotating Ultrasonic Sensor 10
- Simulated Payment System 11
- Drink-Lifting Mechanism 11
- Drink Tray 13
- Drivetrain and Wheel System 14

Software Design and Implementation 19

- Data Storage 19
- Functions 19
  - Setup 19
  - User Interface 19
  - Obstacle Avoidance 20
  - Driving 21
- Decisions and Trade-Offs 23
- Testing 23

Problems and Resolutions 25
Verification 25
Project Plan 29
Original Plan 29
Splitting Up the Tasks 30
Execution of the plan 30
Conclusions 31
Recommendations 32
References 33
Appendix A: Flowcharts and Integration Testing A-1
Appendix B: Source Code A-11

Table of Figures

Figure 1: Task List 5
Figure 2: Initial Design Sketches for BarBot 9
Figure 3: Configuration of EV3 Brick 10
Figure 4: Front View of the BarBot with ultrasonic assembly visible 11
Figure 5: Close-up view of worm gear rotating spur gear for tray-lift mechanism 12
Figure 6: SolidWorks model of 3D-printed drink tray 13
Figure 7: Pictorial view of tread assembly 15
Figure 8: Bowing of the front of the robot due to excessive weight 15
Figure 9: Placement of ball bearings underneath robot 17
Figure 10: Robot backing into turns, altering the driving route 18
Figure 11: Criteria 26
Figure 12: Project Checklist 29
Figure 13: Task Main Flowchart A-1
Figure 14: setUpSequence Function Flowchart A-2
Figure 15: killSwitchCheck Function Flowchart A-2
Figure 16: changePlatformH Function Flowchart A-3
Figure 17: checkObstacle Function Flowchart A-3
Figure 18: isObjectDetect Function Flowchart A-4
Figure 19: Accelerate Function Flowchart A-5
Figure 20: rotateRobot Function Flowchart A-5
Figure 21: getDistances Function Flowchart A-6
Figure 22: adjustDrift Function Flowchart A-6
Figure 23: adjustBeforeTurn Function Flowchart A-7
Figure 24: driveDist Function Flowchart A-8
Figure 25: overshoot Function Flowchart A-9
Figure 26: Object Detection Integration Test Function A-10
Figure 27: Object Detection Integration Test Main A-10

List of Tables

Table 1: Integration Testing Procedures 24

Summary

This report details the engineering design process and construction of the BarBot, an autonomous drink-delivery robot. The objective of this report is to outline the development behind the numerous hardware and software systems used in the BarBot's design, as well as the various critical decisions made throughout the project.

The BarBot was constructed to increase efficiency within the service industry, and to operate in a fast-paced and disruptive environment. Importantly, the BarBot was designed to meet several key criteria and constraints to complete its objective. These specifications included smooth acceleration when driving and stopping, consistent object detection and avoidance, straight driving, accurate rotation and lifting of the tray to deliver drinks to table height upon reaching the customer's table.

There were numerous constraints that impacted the development of the BarBot, with the most problematic constraint being the inaccuracies related to the gyroscopic sensor and the imbalance between the large servo motors used when driving. These two issues culminated in motor drift, causing the robot to skew towards the edges of the pathway, and introducing potential hazards to both patrons, drinks, and the robot itself. Several potential solutions were created to work around these constraints including various hardware redesigns, dedicated software solutions and changes to the layout of the environment.

With respect to the hardware, the BarBot featured a stable base which integrated the Lego EV3 Intelligent Brick and was powered by two large servo motors for optimal driving and rotation. A key component of the BarBot was the scissor-lift tray mechanism which was built to safely lift beverages to table height to assisted customers in receiving their orders. The rotating ultrasonic assembly was, with the ultrasonic sensor mounted on a medium motor to check all directions. The ultrasonic sensor was the primary detection mechanism for objects within the robot's path and was responsible for detecting objects on both sides to ensure the safety of the BarBot and customers.

Development of the software was done throughout the production of the BarBot and included various trivial and non-trivial functions used in the BarBot's operations. Core functions of the BarBot include the "DriveDist" function which drove the robot along a precise path directly to the customer's table and back to the bar area, "ObjectDetect" which was called once an object was detected and would drive around the obstacle, "ChangePlatformH" which controlled the tray height, as well as other supplementary functions.

The final production release of the BarBot performed all the necessary tasks as designed and demonstrated the advantages of automation within the service industry. Although the BarBot met all the criteria for the project, several recommendations have been proposed to improve the design and ensure the BarBot is meeting and/or exceeding all requirements. This includes a redesign of the base to alleviate weight off the large servo motors, improving the turning radius and drift by using tighter tracks or tensioners, and by using more accurate parts such as better gyroscopes and ultrasonic sensors.

Acknowledgements

The design group for the BarBot, composed up of Warren Cao, Camron Sabahi-Pourkashani, Dylan Finlay and Marlon Poddalgoda, would like to express their gratitude towards the University of Waterloo, and the many teaching assistants and professors that offered their help throughout the project.

This project could not have been completed without the guidance and feedback offered by the teaching staff of MTE 100 and MTE 121, as well as the instructional aid offered by both Professor Mohammad Nassar, PhD, P.L. Eng., and Professor Ryan Consell, P.L. Eng.

The design group would also like to thank the University of Waterloo Department of Mechatronics and Mechanical Engineering for supplying the project with the necessary parts and aid to complete the design, assembly, and testing.

Introduction

There are over one million servers in the US alone [1], yet human servers are not as efficient, aware, or predictable as their automated counterparts. The largest being the human factor which causes mistakes and errors such as spilt drinks, forgotten tables and but not limited to mistaking a customer's order.

The BarBot was a specialised Lego EV3-powered robot that safely and efficiently delivered drinks, which in turn eliminated human error. By replacing human waiters with a robot, the bot was capable of increasing efficiency in the workplace, reducing customer wait times by traveling along the most efficient route, and prevent spillage or tampering of the drinks. Using six inputs, it could compute the location of tables and drove to the calculated locations and rose the drink tray to the table upon its arrival. Additionally, the project was capable of simulating payments with the use of a touch sensor, as well as detecting and safely avoiding moving and stationary obstacles within the restaurant while performing its primary role. With human factors removed, the service industry can increase productivity and efficiency resulting in reduced customer wait times and better customer service.

Scope

The main functionality of the BarBot has been to deliver drinks to customers within a restaurant setting. While designing the BarBot it was important to consider what tasks are performed by waiters in the service industry (seen in Figure 1 below), and how an automated solution would be able to complete the same tasks without compromising on the quality of service.

Upon start-up, the BarBot would ensure the functionality of the drink tray by lifting and lowering the tray mechanism, and then would proceed to read the delivery orders off a file. The robot then drove towards the bar, where it would lift its tray once more to accept drinks from the bartender. Once the bartender pressed the touch sensor down, the bot would drive a precalculated distance in the horizontal direction, turn 90 degrees, then drive again in the vertical direction until it reached the customer's table. The robot announced the delivery of the drinks using the integrated speaker on the EV3, and then raised the drink platform for the customers. The BarBot would then ask for payment, where the customer could simulate payment using the touch sensor. The bot would then lower the drink platform and drive back to the bar area. This process will repeat until all orders specified by the text file are delivered.

Once the final order was completed, the BarBot initiated shutdown procedures, where it would return to the housing area and position itself towards the bar area for the next time it started.

Once all the motors were off and the robot was in place, the program ended. If the BarBot was shutdown by the kill switch, the bot would decelerate to a stop and ensure all the motors are off, where it then would play an audio queue warning the user of the shut down.

Start up Tasks

1. Raise and Lower tray
2. Drive to Bar
3. Read number of orders to complete

Operation

1. Raise tray to get drinks from bartender
2. Drive to table in horizontal direction
3. Rotate 90 degrees
4. Drive to table in vertical direction
5. Announce to customers drinks have arrived
6. Raise platform for customers
7. Ask for payment
8. Drive back to bar once the users have paid

Environment Reaction

1. For expected environment
    a. Robot drives around tables and drives in a path where it knows there should not be any obstacles
2. For unexpected environment
    a. If a moving object appears, the robot quickly slows down to a stop and waits 5 seconds
    b. If the object is still there after 5 seconds, meaning it was not an moving object or person then the robot drives around the object
    c. If the object is not there after 5 seconds, that means it was a moving object such as a person walking through the path of the robot
    d. Once confirmed the moving object is not there anymore, the robot accelerates to its driving speed and continues on its path

Shut down Procedure

1. If the robot has finished all its deliveries:
    a. The robot returns to the housing area
    b. Rotates 180 degrees to be facing the bar the next time it starts
    c. Ensures all the motors are off
    d. End
2. If the robot is stopping due to the kill switch:
    a. The robot accelerates to zero
    b. Ensures all the motors are off
    c. Plays the corresponding sound files
    d. End

Figure 1: Task List

Measurements and Inputs

The BarBot takes in various measurements and inputs from a range of sources. At the start of the program, the robot first reads from an embedded file containing the various drink orders, where it will then deliver drinks and return to the bar each time. While driving, the BarBot uses motor encoders to accurately measure the distance driven and compares the encoder values with calculated limits to stop at the correct table. This completed using the getDistances function, which calculates the distance to each table using a grid system. The ultrasonic sensor mounted at the front of the robot continuously detects the presence of an object within 40cm, and if detected will call the ObjectDetect function to stop the robot. A touch sensor mounted

on the top of the robot simulates a card tap payment system and detects user input when at the customer's table and at the bar. One important measurement is the readings from the gyroscopic sensor, used not only to make various rotations but also to correct any drifting of the motors. Inaccuracies present in the motors resulted in the BarBot drifting to the sides. So, to combat the issue a function was created that would continuously read the gyro and adjust the motor power to straighten the robot.

Interactions with the environment

Interactions with the environment are done through user input, sensor input and visual/audio information. Since the robot was designed with restaurant conditions in mind, it was important to ensure the safety of customers and the robot itself during normal operations. It uses user input to ensure that customers receive their drinks, and that the robot does not drive away too soon. If the ultrasonic sensor detected an object, the BarBot was programmed to wait five seconds to determine if the object was moving first, and then drive around the object in a rectangular movement. Throughout the BarBot's operations, visual information would be displayed onto the EV3 screen and auditory recordings would play to ensure information was relayed to the customer/user and would include messages such as "Drink this", "Give me money", and "Move out of the way".

Motor usage

The BarBot uses several EV3 compatible motors to perform its various tasks. Two large servo motors attached at the front to the base provide power to the treads and drive the robot, and are responsible for rotation, acceleration, and deceleration. A medium motor was mounted to the ultrasonic head and was used to rotate the ultrasonic sensor, providing a 180-degree field of view for the robot. This enabled the BarBot to have strong obstacle detection, and the ability to check in multiple directions for objects to prevent accidents. The final medium motor was used in the scissor-lift tray mechanism, where the motor was connected to a worm gear that would spin a spur gear, lifting the tray up above 25cm.

Changes in Scope

Initial ideas for the BarBot included various features that could not be incorporated into the final product. The very first idea created was of a juicing robot that could produce drinks from lemon halves, but this idea proved too difficult and was turned into drink delivery. Originally, the concept included the use of a drink tank, where the robot would pour drinks into cups using specialised valves. Since parts were limited, the use of valves and liquid storage was instead

replaced by the delivery of a cup on the back of the robot, where the robot would be able to lift the cup to table height and deliver drinks directly to patrons.

Concepts for the BarBot used a winch system to pull the tray up, where a large servo motor would pull a string that was attached to the tray on guided rails. The winch-pulley system would use a high number of parts and was also determined to be too heavy and large for the BarBot and was replaced with a scissor-lift mechanism instead.

Criteria and Constraints

The robot is required to meet several criteria and constraints to operate within a workplace. Many of these criteria are related to the problem that the project was designed to solve and have been created to ensure that the robot is capable of safely delivering drinks to patrons. The constraints of the project have been placed by the operating environment of the robot, the abilities of the parts provided for the prototype, and issues culminating from the lack of accuracy within the EV3 parts.

Criteria

With the BarBot designed to replace human waiters within the service industry, it was important that the robot met all the necessary criteria to correctly deliver drinks to the customer's tables.

The BarBot needed to safely deliver drinks to the customers, loading up drinks at the bar area and efficiently traveling to the proper table location. While driving, the BarBot should smoothly accelerate and decelerate when making changes in speed, to prevent drink spillage.

Objects would have to be detected in advance and consistently, and if detected the robot would come to a smooth stop and wait. If the object were still being detected, the BarBot would have to move around the object in a safe manner, checking both sides to ensure accident prevention. It was important that the robot did not hit any tables along the way and kept straight driving and correct rotations. This meant that the robot would need to adjust for any drift and ensure that all turns were correctly performed throughout the entire program using the gyroscopic sensor.

The BarBot also needed to calculate the distance from the bar to each table, and always deliver to the correct table as listed within the file. If the BarBot overshot its destination, then it would remember the original table distance and drive back to the customer's table to complete the

order. Upon delivery, the BarBot needed to lift its tray and present the drink to the customer, accepting payment afterwards and safely lowering the tray back down. Auditory recordings should be played at the correct situations and be clear and concise, as well as audible to the customers.

Constraints

Various constraints were clear early in the project and were either a result of the parts that were used, the function of the BarBot, or requirements for the project completion.

One crucial constraint was that the BarBot should not spill any amount of the drinks throughout the program, which meant restricting the speed during driving and ensuring that the robot smoothly accelerated and decelerated to prevent sudden jerks in motion. The BarBot needed to be built purely with the Lego pieces available to the design group and with a limited number of motors and sensors, as parts for the project became scarce almost at once. This resulted in the design group contacting other groups to trade parts, as well as part requests from the WEEF TA office.

The BarBot had to drive in straight lines and stay within the middle of the path. This constraint proved to be difficult, as imbalances within the motors would often cause drifting and lead the robot to drive towards the sides of the pathway, which would compound overtime and result in the robot being severely off course. To mitigate this, the design group performed various hardware redesigns and created the “adjustDrift” function, a function that would continuously check the gyro values and adjust the motor powers accordingly.

The robot also needed to rotate at the correct angle, making exact 90 degree turns multiple times throughout the program. This constraint was important to ensure that the robot did not turn and drive at an angle, which could cause the robot to not complete deliveries and potentially collide with tables or obstacles. The included LEGO EV3 gyroscopic sensor did not have the accuracy needed for the project and resulted in the BarBot occasionally undershooting or overshooting turns, even when the rotation angle was set to 90 degrees within the software. Adding gyro checks within the rotation function ensured that the robot completed every turn correctly and did not drive away until 90 degrees of turning was reached.

The BarBot also had to have a kill switch within every function, which could be activated at any time during operation. The kill switch function in the BarBot was designed to smoothly decelerate the robot and initiate shut down procedures, in case of an emergency. This meant that the kill switch function would call the acceleration function, meaning that during acceleration the kill switch could not be pressed.

Mechanical Design and Implementation

The mechanical design of the BarBot was formed of six major hardware assemblies: the housing for the EV3 Lego brick, free-rotating ultrasonic sensor, simulated payment system, drink-lifting mechanism, drink-tray, and drivetrain and wheel system.

The first meeting was held on October 22nd, where the design team devised multiple initial designs. The first concept sketches of the hardware implementation, in Figure 2, comprised of the EV3-Brick situated in the middle, with the ultrasonic sensor at the front and a winch-lifting tray at the rear. The first sketch contained six wheels for stability and 3 large motors, two for driving and one for the winch system.

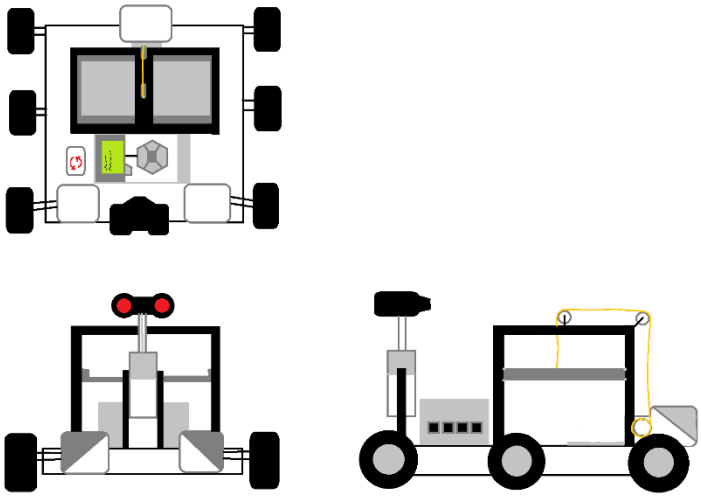


Figure 2: First BarBot Design Sketches

Housing of EV3 Brick

Throughout the timeline of the project, the position and housing of the EV3 brick remained relatively constant. To provide accessibility to the sensor and motor ports, the EV3 Lego brick was configured as demonstrated by Figure 3. Relative to the ground, the position of the brick was low to provide a stable center of gravity, which was vital to providing a balanced drive, and prevent spillage of the drinks, a main objective of the project.

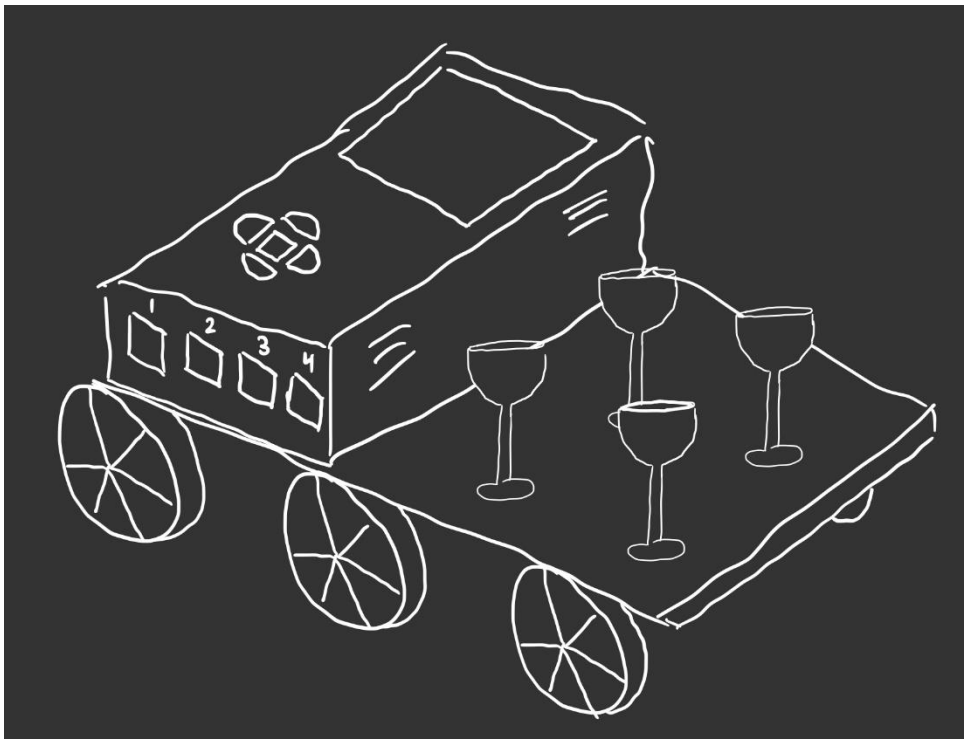


Figure 3: Configuration of EV3 Brick

In the first design sketches, the Lego brick was placed centrally in the design, however the placement of it was quickly moved to the front of the robot to offset the weight of the scissor lift and drinks and avoid tipping of the drinks.

Free-Rotating Ultrasonic Sensor

In a realistic restaurant scenario, the BarBot will meet obstacles in its way to the table, whether it be a person in front of it, a dropped dish, or. To supply safety and satisfaction to both the patrons, drinks, and robot, the project group devised that an ultrasonic sensor be placed on the robot to detect obstacles in the robot's path. It was decided that obstructions within a restaurant would be placed in two categories: stationary and moving objects.

For moving objects, the BarBot would deaccelerate, and wait for the object to continue out of its path. However, if the object had not moved after a specified period, then the bot implements a dynamic object avoidance system. In this system, the importance of a free-rotating ultrasonic sensor became clear. The rotation allows for the BarBot to safely check for obstacles to its left and right and continue around the object on an unobstructed route.

In the initial design, the ultrasonic sensor was placed stationary upon the front of the BarBot. However, it would have been impractical to effectuate a fixed ultrasonic sensor as obstructions

may appear on any side of the robot. Therefore, the ultrasonic sensor was placed upon a medium servo motor, which was mounted upright as seen in Figure 4.

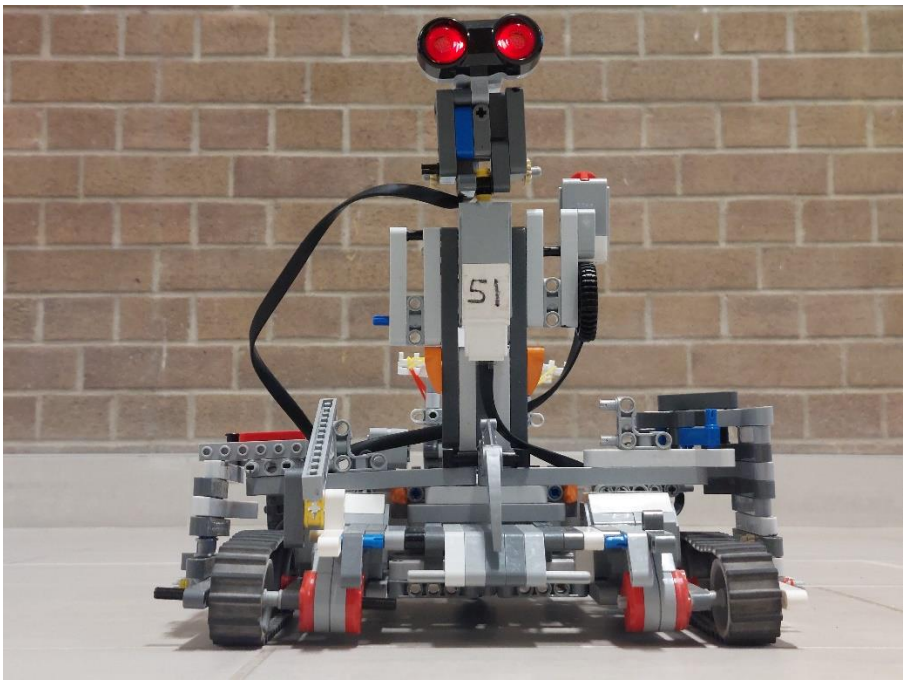


Figure 4: Front View of the BarBot with ultrasonic assembly visible

Simulated Payment System

To supply demonstration of portability to realistic restaurant scenarios, a simulated credit/debit card contactless payment system was attached to the robot beside the ultrasonic head.

The EV3 Lego touch sensor was used in place of a debit terminal and in order to be easy to access, placed at table height, level to the final height of the scissor-lift when raised. Before the BarBot would release the drinks to the customers, a simulated payment of a card tap on the touch sensor was needed.

Drink-Lifting Mechanism

As a result of the low placement of the EV3 brick, the hardware was built low to the ground to keep a lower centre of gravity and therefore increased stability. As a result, the drinks would also be housed low to the ground. Therefore, to provide accessibility, satisfaction, and ease of use to restaurant patrons as well as bartenders, it was devised that the BarBot must be able to lift drinks to a specified height.

The final selected system implemented the use of two types of gears to transfer rotational force from the motor to lift the tray upwards. Upon a medium servo motor, an axle with an

attached worm gear would rotate along its centre line as shown in Figure 5. The rotational force from the worm gear would be transferred to a spur gear upon which the bottom leg of a scissor-lift mechanism was attached.

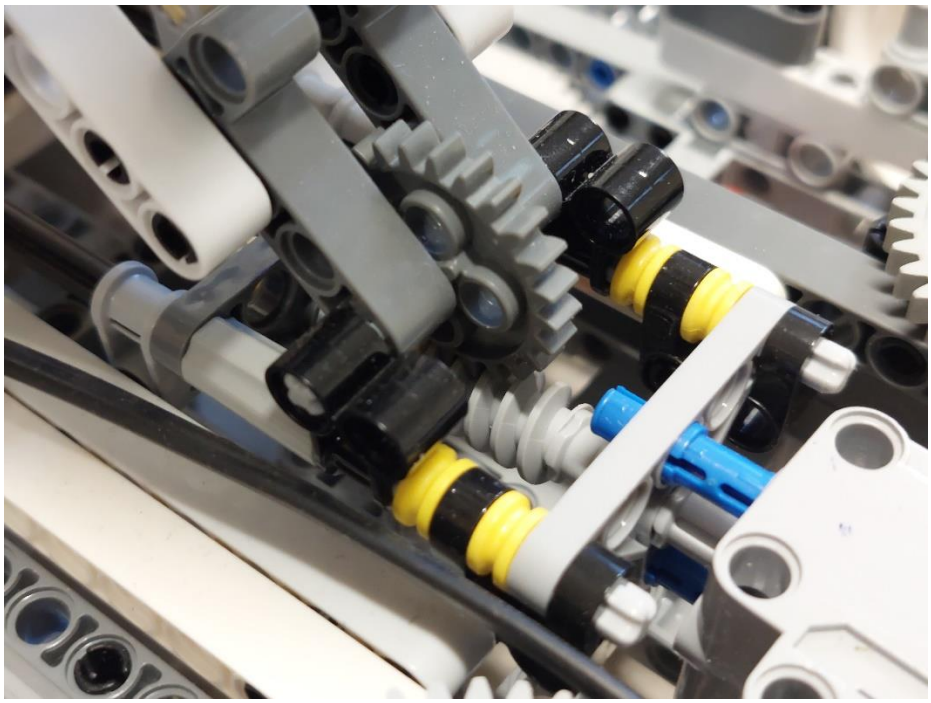


Figure 5: Close-up view of worm gear rotating spur gear for tray-lift mechanism

As the worm gear is rotated by the medium motor, the spur gear would rotate as well, and raise the leg of the scissor-lift upwards. As a result, the scissor-lift would raise the drink tray to the specified table height.

To keep the drink platform horizontal, the scissor-lift mechanism was attached by a free-sliding piece onto a rail on both the platform and in the housing by the motor in order to provide a free-range of motion while preventing spillage of drinks.

Throughout the design process, numerous alternatives of drink-lifting mechanisms were discussed before the final scissor-lift design was selected. In the initial design sketches, it was proposed that a cable-winch system be used. This design used two large motors and the use of gears to spool a string, which was attached on both sides to the platform around a cylinder, to slowly draw the drink platform upwards.

However, this design was not chosen because of the need for extra motors, and it was theorized that due to the imperfection and inconsistency of the EV3 Lego motors, the synchronization of the dual motor system would be difficult. As well, cable systems require

towers to the side of the platform to provide height to draw the platform towards. This would be unstable as the centre of gravity would be much higher. Additionally, this system would be hard to store as the robot would require a large housing space.

As a result, the second idea, the scissor-lift, was chosen and implemented. This design was much more stable, as the support needed to lift the tray to the height is much less. As well, the design was much easier to store as the scissor lift compressed when not in use.

Drink Tray

The tray for carrying drinks to customers was needed and was to be placed upon the scissor lift. A tray must be able to securely provide support from all sides to prevent drinks from spilling.

The final design of the tray was 3D-printed to provide a snug, custom fit to the cups. Printed by classmate Donna Kim on her personal Monoprice Mini 3D-Printer, the ABS plastic design involved a 4mm platform upon which two hollow cylindrical extrusions are attached. This could hold two of the Tetrix red solo cups. The radius of the extrusions matched the radius of the cups just underneath the lip. The design of the drink tray is shown below in Figure 6. This design ensured that the largest cup would be housed within the cylinder. This would have meant that nothing short of an inversion of the robot would cause the cups to fall out if there is sufficient weight within the cups.

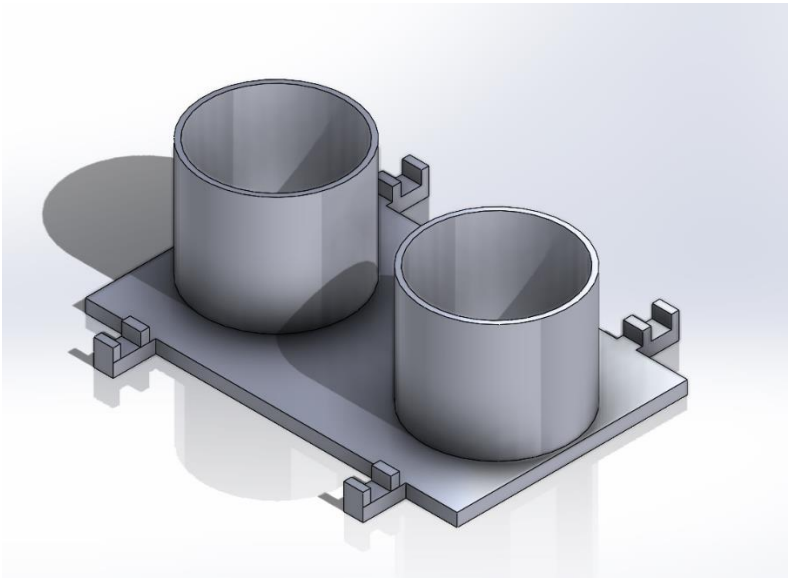


Figure 6: SolidWorks model of 3D-printed drink tray

On the initial concept sketches, a tray was not included. The need for a tray was found at the second meeting of the design group. Two other iterations of a tray were considered, a solid

rectangular-prism-shaped tray, with the radii of each tier of the solo cup within providing a one-to-one fit, and a hollow version of the same shape.

The initial devised design was the solid rectangular-prism tray. This design would hold four cups arranged in a square formation. This was meant to be attached to the EV3 robot via mini 3D-printed locking pins, which would slide through the holes of the Lego technic arm pieces and into slots on the side of the tray, preventing movement. Shortly thereafter, the design was hollowed out to save plastic and reduce the weight of the tray, which when heavy, could tip over.

However, when consulting with the technicians at WATiMake, they advised that due to the tight timeline, it would be exceedingly difficult and stressful to print the rectangular-prism tray design, whether hollow or solid, as it would take almost a week to print. The result of the feedback was the simplified two-cup design modeled on SolidWorks and printed instead.

Drivetrain and Wheel System

A consistent driving system in a busy restaurant would be imperative as it would prevent accidents and ensure the safety of both the robot as well as the customers. Consequently, the drivetrain of the BarBot received the most revisions and redesigns in order to provide the most consistent routes to deliver drinks.

The drivetrain's final design in the BarBot features a tank-like dual tread system as well as three ball-bearings to provide three extra points of contact to ensure stability. Two Lego EV3 large motors powered the drivetrain. These delivered power through an axle to a front wheel hub contained within the tread. As the motor powered the front wheel hub to turn, then the tread would rotate as well, with a second wheel hub at the rear of the tread to keep tension, as seen in Figure 7. The tread used on the robot comprised of rubber, which provided good traction on carpet, but struggled on tile as a result of the smaller surface area due to the smaller points of contact.

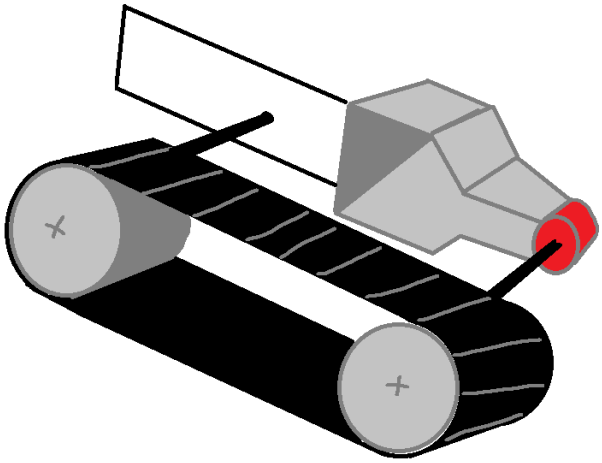


Figure 7: Pictorial view of tread assembly

The robot's centre of gravity lied between the tracks, and the track system was wide to further stabilize the robot. Consequently, the weight of the robot would cause the drivetrain to bend inwards, proved on Figure 8. This issue compounded on top of hardware motor imbalances to cause large inaccuracies on routes. The deviation was almost 30 degrees off path, increasing with a farther distance.

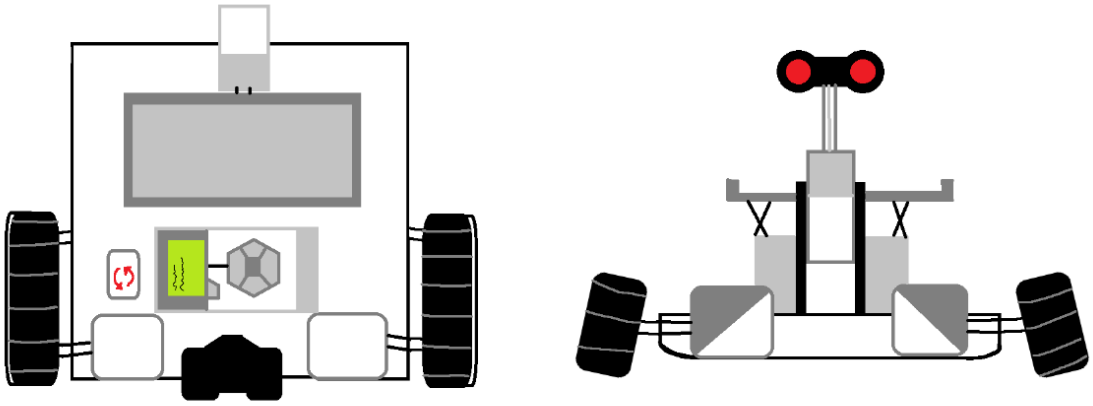


Figure 8: Bowing of the front of the robot due to excessive weight

In the final design, various Lego pieces connected the outer edge of the tread to the main frame of the robot to decrease the bow of the tracks. However, due to the pliability of the Lego pieces

and the gaps between pieces when connected with the provided connecting pins, this only limited the bow and did not remove it altogether.

In order to further combat this issue, a further elastic system was implemented on the undercarriage of the robot. Protruding double connecting pins were attached to the outside of the tread as well as to the main undercarriage frame and elastics were wound the pins to forcefully draw the tracks back to an acceptable position. All these measures in combination lessened the bow as much as possible and aided the robot to drive straight.

The initial design sketches of the robot featured six wheels as seen in Figure 2. The six wheels would have provided a stable base for the robot as it featured many points of contact, and a wide base. This system included a front wheel drive of the two front wheels, which pulled the robot forwards.

However, due to a lack of parts, the first revision to the driving system was that the robot should only have four wheels. While this reduced the points of contact, it was decided that the robot be kept low to the ground to prevent tipping to reduce the size of the issue.

Later during a hardware meeting, the design group realized that this design would be implausible because as the robot turned, the wheels along the side or back of the robot remained in line with the robot and could only turn along one axis. As a result, the non-powered back wheels would drag along the ground, and the robot would have major difficulty turning.

Therefore, the second revision to the driving system was to swap the two stationary rear wheels with two caster wheels designed from the Lego EV3 pieces. Caster wheels are wheels that can pivot 360 degrees and provide motion in any direction. Therefore, when the robot turned, the caster wheels would turn as well and prevent drag along the back of the robot.

After designing and attaching the caster wheels to the robot, the housing for the caster wheels proved to be much more elevated than the front wheels themselves. This offset the horizontality of the drink platform and spilled the drinks, thus failing one of the main goals of the robot.

Consequently, a third revision of the rear wheel system was needed. In place of the two caster wheels was a singular ball bearing. This wheel system was similar to the default configuration of the Lego EV3 robot, with a dual wheel drivetrain in the front of the robot, and a singular ball bearing in the rear. The ball bearing acted as a miniature caster wheel and could rotate freely within its housing in any direction.

Due to the weight of the robot, a singular rear ball bearing disrupted the stability of the robot, and it could easily tip side to side. As well, the uneven weight distribution also caused difficulty in turning as the ball bearing had so much weight that it had difficulty rotating, and so the robot would lose traction and slowly rotate around, with the ball bearing as its pivot point.

The fourth revision of the wheel system added two more ball bearings. The configuration of the ball bearings is shown in Figure 9. These spread the weight of the robot more evenly, and as a result, robot was much more stable.

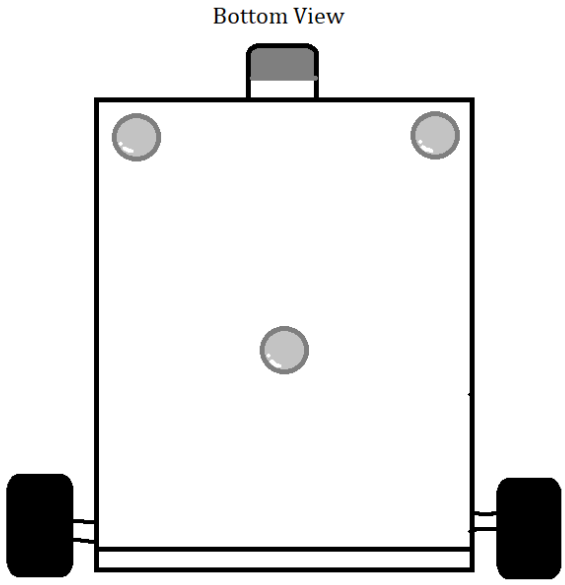


Figure 9: Placement of ball bearings underneath robot

When running tests on the turns of the robot, the robot turned slowly, with the wheels still not able to fully gain traction on the carpet. Additionally, the robot would not turn consistently, with the robot making a j-shaped motion like Figure 10. The robot's front turning wheel would not gain traction, and the back wheel would push the robot backwards and away, producing this turn. These introduced extra variables to account for into the driving that would drastically alter the route of the robot and run the risk of the robot backing up into an object. Without an ultrasonic sensor on the back of the robot, this would pose a safety hazard in a realistic scenario, and therefore needed to be revised.

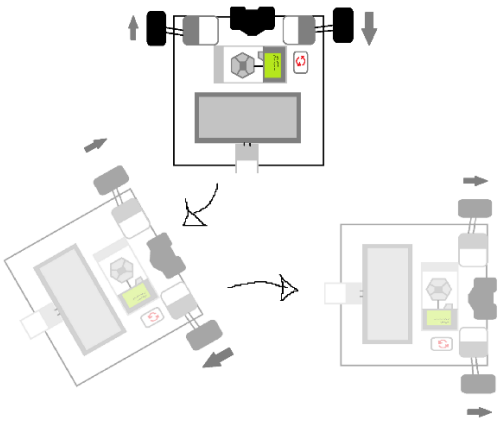


Figure 10: Robot backing into turns, altering the driving route

The fifth revision of the wheels targeted the two wheels at the front that were being powered by the motor. Switching from the 2.75cm radius Lego EV3 wheels to the one large Lego EV3 wagon wheel on each side, the design group hoped that the traction pattern on the tire of the wagon wheels would help the wheel grip the carpet better and swing the back end of the robot around to eliminate the previous j-shaped turn.

The wheels successfully fixed the turning issue, but due to the decreased with of the tires as well as the increased radius, the tires were bowing inwards under the weight of the robot and having difficulty gaining traction on the carpet.

The sixth revision to the wheels were the addition of another wagon wheel on each side of the front to a total of two wheels per side. This provided an increased area of contact for the wheels and more points of contact to help with the bowing of the wheels.

As a result of this revision, the turns were fully as intended, however the bowing of the wheels were still major, and the robot was extremely inconsistent in driving a straight line. Even after reinforcement of the sides of the wheel to the upper frame of the robot, the wheels would still bow.

The final revision of the robot was to implement tracks. A track system provided the largest area of contact and was the easiest to reinforce to combat the bowing issue. This required a complete rebuild of the front end of the robot as well as the sides. The track system incorporated one track on each side of the robot, and along with the aforementioned reinforcements, was able to drive the robot straight consistently.

Even with all the reinforcements in place to aid the robot to drive straight, in the end the fusion of software and hardware solutions were needed to drive directly to a target without fail.

Software Design and Implementation

The general layout of the software design was broken up into four distinct groups which successfully demonstrate the main objectives of the robot and its tasks: setup, user interface, obstacle avoidance, and driving. These specific categories were used to group the software since all four were crucial to a successful operation. The stated categories are justified with the task list in Figure 1. For a general flow of task main refer to Figure 13.

Data Storage

One of the inputs used was file input, where the robot read the number of orders to complete from the file, followed by the table numbers in order of priority for the robot to deliver drinks to. This data was read into a variable so the robot could pass it to a function to calculate the optimal route to the designated table in its current travels.

Functions

Setup

The first function used for setup was given by the MTE121 Teaching Staff. SensorConfguration configures all the sensors throughout the robot and sets certain sensors to the correct mode, for example the gyro sensor is set to the rateAndAngle mode.

The second function used for setup was the setupSequence function. Written by Camron Sabahi-Pourkashani, the void function had a single parameter which was the distance the robot had to drive from its housing space to the bar. The function operated the scissor lift, moving it up and down to ensure the mechanical aspects were operating correctly. It then drove to the bar, rotated to be facing the correct orientation for when it began its delivery course, and then lifted the scissor lift for the bartender to place the drinks required for delivery. For a flowchart refer to Figure 14.

User Interface

Since the BarBot was situated in a high human populated environment, it is paramount the user interface be intricate enough for the intentions of the robot to be clear. Specifically, there were several ways the robot can effectively demonstrate its objectives. For example, displaying the current table number which the robot is enroute to would allow other tables to recognize

whether the drinks are for them or not. This communication is executed with the usage of RobotC's built in function displayString.

Although, displaying strings to the screen of the EV3 brick would count towards communication with the environment and clients, the next step higher would be for the robot to vocally communicate. With the usage of audio files recorded by Professor Mohammad Nassar, PhD, P.L. Eng, the Lego EV3 BarBot utilized different audio based on different triggers and timings.

Aside from the use of built-in functions, such as playSoundFile, two functions were dedicated to some sort of user interface.

The kill switch check function is a void function which takes a boolean that holds whether the up button on the EV3 was pressed. The function, written by Dylan Finlay, is called after every change in scope throughout the source code. Meaning, after every while loop is broken and after the compiler leaves a function the killSwitchCheck is called to check if the change in scope was due to the kill switch button being pressed. Refer to Figure 15 for flowchart.

The changePlatformH is also a void function with a boolean switch, holding the value for whom the platform is being raised. If raised for the table, the boolean is true, and the robot will use voice files to greet the clients and announce the arrival of their drinks. If being raised for the bar to load drinks or to check its operation during the setup sequence, the switch is false which does not allow the brick to play sound file. The function raised the platform by using motor B and driving the motor in the positive direction until its motor encoder value reaches the constant variable which holds the max height before damaging the scissor lift. For flowchart, see Figure 16.

Obstacle Avoidance

The second category of software operations was dedicated to the ability of detecting and avoiding both moving and stationary objects that could be presented as interferences in the robot's regular procedure. The following are the functions used to check for objects.

First was the function which checks for objects on the left and right sides of the robot. Written by Camron Sabahi-Pourkashani, checkObstacle takes in two boolean variables, one which holds if an object has been detected, and the second being a boolean switch that holds which way the motor should rotate, true being clockwise and false being contraclockwise. The function returns a boolean, true if an object has been detected in the direction it checked. False, if no objects detected on the side of the robot which was checked. See Figure 17.

The second function used for object detection and avoidance was isObjectDetect - written by Marlon Poddalgoda. The function returned an integer value and had no parameters. This function was only called when the robot detected an obstacle in the specified range while driving. If the function was called, a timer was used to check if the object was still there after five seconds. If it was, the robot registered it as a stationary object. If the object is no longer there, then it was recognized as a moving object and the delivery system continues with its order. If the object was stationary, checkObstacle is called. The robot then traveled three sides of a rectangle around the obstacle on the clear path determined by the ultrasonic sensor. The rectangle the robot drives has a one to two ratio, width to length. When the function completed all its tasks, it returned the centimeter length of the long side of the rectangle which the robot drove to account for the distance in its driving procedure. Because no matter the orientation the longer side of the rectangular is always in the same direction as the table which the robot was travelling to. Note that if the robot determined the object to be moving and in turn did not drive the rectangle around the object, the function returns zero. For an in-depth flowchart refer to Figure 18.

Driving

The remaining functions were chosen to optimize the driving of the robot, which included minimizing the constraints from hardware such as drift and compounding turning angles.

The only trivial function throughout the program was the function cmToEncoder, written by Camron Sabahi-Pourkashani which takes in a distance in centimetres and return said value in encoder counts based on the radius of the robot's wheels.

The first non-trivial function, was the void acceleration function, written by Warren Cao, which took in the desired final motor speed as a parameter then used for loops and if statements to smoothly increment or decrement the speed of motors A and D, which used the front left and right wheels, respectively. See Figure 19 for thorough flowchart.

Next, is the rotateRobot which was taken from the MTE 121 course and modified by Camron Sabahi-Pourkashani. The function remained type void but took two parameters; the absolute value of the turn angle and a boolean switch which holds the direction the robot should rotate. The robot then rotated according to the parameters using a while loop. For an in-depth flowchart refer to Figure 20.

To decide how the robot should travel to the table, getDistances – written by Camron Sabahi-Pourkashani was passed the integer value of the table number by value and passed the variables which hold the x and y distances the robot should travel by reference. The function

used integer division and modulo operators to split the table number in the row and column number where the table is situated. It then used an equation to calculate the exact distance to travel in the x and y directions and stored it in the two variables which were passed by reference, xDist and yDist. For a visual flowchart see Figure 21.

The next two functions were written to compensate for mechanical drawbacks and reduce their impacts on the operation.

First is adjustDrift, a void function without parameters written by Camron Sabahi-Pourkashani which used the gyro sensor to detect if the robot deviated from its path by more than 2 degrees in either direction. If it had, motor D was adjusted according to quickly push the robot back on course. The reason for only using motor D was because the function which drove the robot to the table used the motor encoder value from motor A, hence motor D's speed is adjusted to avoid over or undershooting the final destination. For a visual flowchart refer to Figure 22.

The next function written adjusted the angle of the robot before a turn. This is because adjustDrift only adjusted drift greater than ±2 to avoid falling in an infinite loop inside the function. Hence when the robot reaches the end of a leg and wanted to turn, it could be stopped on an angle varying from negative two to positive two degrees. The function checks the gyro to decide if the robot is on an angle or not. If so, the robot was rotated to be situated at zero degrees to end angle compounding through its turns. Figure 23 presents a high-level flowchart of the function's process

The main driver function of the program is the void function driveDist, written by Camron Sabahi-Pourkashani which took an integer value for the distance the robot must travel as well as the motor power to drive at. The function uses a while loop to find when to stop the motors. The only way the while loop exits is if the robot reached its target distance or if the kill switch button was pressed. In the while loop the robot checked if there was an object within its detection range. If there was, then the robot slowed to a stop and called isObjectDetect to determine how to interact with the object. The robot also used variables to temporarily hold the encoder count of motor A in case the robot must drive around an object and reset the encoder count along the way. A second variable was used to store the returned value from isObjectDetect. After the object detection routine was finished the compiler returned to the scope of driveDist, it used the encoder count and temporary variables to calculate the total distance driven towards the end goal. Next, the driveDist function calls the overshoot function to check for and deal with the scenario that the robot passed its destination while it was driving around an obstacle. Refer to Figure 24 for a clear flowchart of the function.

Overshoot was a void function written by Warren Cao which took in the encoder count of the distance driven by the robot, and the encoder limit for the robot to reach its distance. If the robot has driven past its desired end point, it uses rotateRobot to turn the robot 180 degrees, drive back the difference of the two values passed to the function, then turned another 180 degrees to be orientated in the correct direction. See Figure 25 for more detail.

Decisions and Trade-Offs

Throughout the software development of the robot, different problems and circumstances called for evaluation of current progress versus tasks left until completion to determine and implement solution plans. For example, instead of adding more features such as a full payment system, said time was better utilized in solving the drift and increasing turning accuracy. This decision prioritized a smooth and efficient drive over extra features. Because, if the robot drove with drift and inaccurate turns, it would have hit tables and would have not executed an effective delivery even if it could have been able to have a more complex payment system.

Testing

Similar to any project containing a large software portion, there were two phases of testing to properly assemble all the functions. The first being unit test where each function was tested on its own. The second kind being integration testing where the functions were added together one by one and ensuring they work together and integrate properly.

For the unit testing each function was downloaded to the robot with a short task main to only test said function and its edge cases.

The integration testing of the program proved to have been much harder. In the specific case of the BarBot, with each new function created it used and called nearly all other functions created before it. That was because the robot had three working tasks, driving, obstacle avoidance, delivering. However, there were 10 + functions that went into those tasks. Starting from the most basic function, one after the other more functions were integrated until all tests came out successful. See Figure 26 and Figure 27 for a more in-depth analysis of the integration testing code. Table 1 below details the various integration test which includes the functions integration, test cases and the expected behaviour of the test cases.

Functions Integrated	Test Cases	Expected Behaviour
isObjectDetect, checkObstacle, rotate Robot, accelerate	Cases: Moving object in front; object in front and on right; object in front and on left	1. For the moving object the robot should move straight. 2. For case two the robot should drive a rectangle on the left. And for case 3 a rectangle on the right. 3. In all three cases the robot should accelerate to a stop
driveDist, isObjectDetect, accelerate, killSwitchCheck,	Cases: drive proper distance; drive distance with object avoidance in the middle	1. The robot should first be able to drive the correct distance without the case of object detection. 2. In the second case the robot should be able to drive around an object and still stop at the correct position 3. The robot should shut down whenever the kill switch is pressed
driveDist, isObjectDetect, overshoot, killSwitchCheck	The robot should return to the correct endpoint when it passes it due to the object detection routine	1. When the robot finished its object detection routine, it should recognize that it has passed the marker, turn around, drive back to the correct point, and turn back around. 2. Robot should shut down if the kill switch is pressed while correcting overshoot
getDistances, rotateRobot, driveDist, isObjectDetect, overshoot, killSwitchCheck	Drive to table without obstacles Drive to tables with obstacles and no overshoot Drive to table with obstacle and overshoot	1. The robot should drive the legs needed to get to the table 2. Should properly deal with object detection and possible overshoot
driveDist, rotateRobot, adjustBeforeTurn adjustDrift	Drive to table	1. The robot should adjust motor powers while driving to fix its drift. 2. Should have a slight rotation before any turns to return the angle to zero.

Table 1: Integration Testing Procedures

Problems and Resolutions

The most significant problem faced was developing software to solve the mechanical drawbacks of the Lego EV3 system. The drift created by imbalanced motors caused compounding drift in the drive of the robot added to the inaccurate turning due to poor precision and accuracy of the gyro sensor.

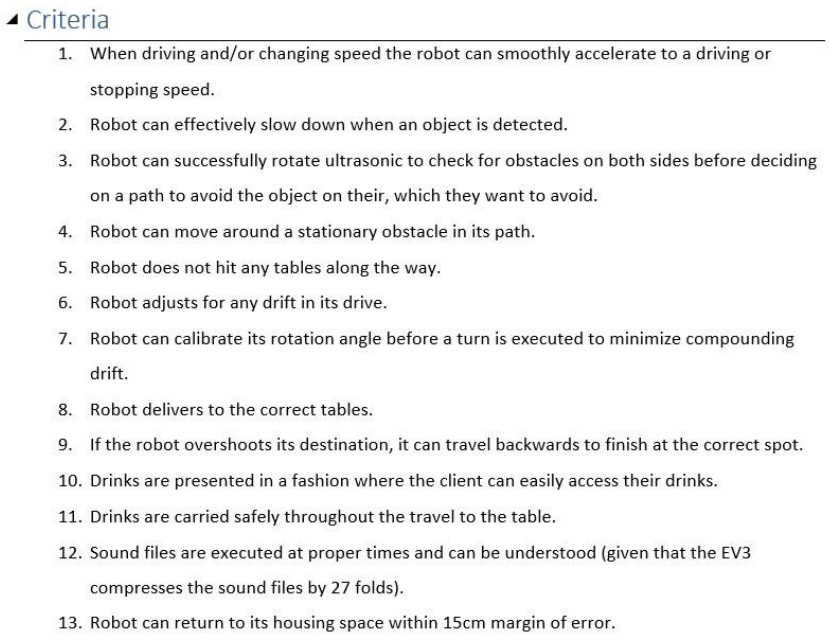
To solve the drift an extra function was created where numerous tests were run to find the best range of drift detection to perfect the drive. That was because if the robot corrected drift as soon as it deviated from zero degrees it would never exit the function because the gyro does not have an exact reading so it would always be off by a slight amount. For this reason, numerous trials were run to find the lowest range which did not cause any problems.

For the turning, the adjustBeforeTurn function was created and implemented to reduce angle inaccuracies which greatly increased turning accuracy resolved angle compounding in turning. The function is explained in depth both above in the driving functions section as well as in Figure 23.

Verification

Although there were many adjustments within the mechanical and software design of the project, its overall purpose remained unchanged. All changes made were done so that the robot could reach the criteria that it was designed to achieve. The robot was made to safely deliver drinks within a restaurant setting, and the criteria was built to ensure that the robot did not hit any obstacles or tables and did not spill any of the drinks. For an updated list of the criteria used during the demo, refer to Figure 27 below.

Criteria



Criteria

1. When driving and/or changing speed the robot can smoothly accelerate to a driving or stopping speed.
2. Robot can effectively slow down when an object is detected.
3. Robot can successfully rotate ultrasonic to check for obstacles on both sides before deciding on a path to avoid the object on their, which they want to avoid.
4. Robot can move around a stationary obstacle in its path.
5. Robot does not hit any tables along the way.
6. Robot adjusts for any drift in its drive.
7. Robot can calibrate its rotation angle before a turn is executed to minimize compounding drift.
8. Robot delivers to the correct tables.
9. If the robot overshoots its destination, it can travel backwards to finish at the correct spot.
10. Drinks are presented in a fashion where the client can easily access their drinks.
11. Drinks are carried safely throughout the travel to the table.
12. Sound files are executed at proper times and can be understood (given that the EV3 compresses the sound files by 27 folds).
13. Robot can return to its housing space within 15cm margin of error.

Figure 11: Criteria

To meet the first criteria of smooth acceleration when there was a change in speed or a stop, the robot proved to be concise with its movements and show slow accelerations every time it began driving from a stopped position. The motor speed would increase at a slow pace, rather than jumping to a high speed. Additionally, it could be seen in the demo that the robot would never have any sudden stops. Every time it approached a stop, the BarBot gradually decreased in motor speed, until a full stop. This was all done through the acceleration function, written by Warren Cao, which was incorporated throughout the code to make sure that no drinks would be spilt due to sudden stops or starts while driving.

The second constraint was met through the object detection function which used the ultrasonic sensor to detect any obstacles in the robot's path. When delivering drinks, the ultrasonic sensor was constantly checking for possible obstacles in its path. In the demo, a human walked in front of the ultrasonic sensor while it was in the middle of its journey to one of the tables, and the BarBot successfully slowed to a stop without hitting the person. The robot also came to a full stop when a water bottle was detected in its path. In the first case, the human was a moving obstacle so after the ultrasonic sensor realized that its path was cleared up again, it continued the original path. However, the water bottle was stationary. In this instance, the robot rotated

its ultrasonic sensor to the left and right sides to check for a safe way to get around the water bottle, which was exactly as it was supposed to do according to the third criteria item.

To ensure that the robot would not hit any tables along the way, a restaurant environment was simulated. This was done by using chairs as place holders for tables. The replacement was only due to size restriction in the demonstration. In the demonstration the BarBot successfully delivered drinks without hitting any chairs/tables nor any obstacles along its delivery to two different tables.

The most difficult criteria to achieve was solving the drift in the robot's drive. After designing a function to solve the issue, adjustDrift. It was noted in the demonstration that whenever the robot began to veer off its path, the gyro sensor was able to detect it, correct its angle and ensure that it travelled a straight line.

The seventh criteria was met in a similar way. Every time that the robot came to a stop where it needed to make a turn, the adjustBeforeRotate function was called. The function would check the gyro and return the robot to a zero-degree heading if it were not already. Hence, when it made the turn, there was not compounding from previous turns or from the drift in the robot. During the demonstration it was clear to the viewers that the robot made a small turn in one direction, adjusting for the drift, before it made any turn required for delivery.

Since the BarBot was built for a restaurant or bar setting, it used file input, so that it could take in any table number in any sized grid and deliver to that table. The program was design to suit any grid and deliver to any table. In the demonstration, a two-by-two layout was used, and the robot was set to deliver drinks to two different tables. It successfully made it to both tables and lifted the tray so customers could take their drinks. This demonstrated the success of the eighth criteria; delivery to correct tables.

The ninth criteria discussed the possibility of overshooting the destination and correcting the position. This was properly verified using the object detection of the water bottle. After the rectangular path to avoid the object, the robot had then gone past the table where it was supposed to deliver to. The overshoot function detected the incorrect position and corrected itself by driving back to the table before delivery the drinks.

The next criteria - presentation of the drinks, meant ensuring that access to removing the drinks was easy. Demonstrated through the scissor lift mechanism along with the design of the tray. The BarBot successfully went to the correct tables in its demonstration and lifted the drinks up to an accessible height for the customers so that the drinks could be taken with ease. This was

also completed through the design of the tray, which ensured that the cups were held in a locational clearance fit to avoid spillage, with extra room added near the top of the cup and between the different cups, so that customers could easily grab their drinks. In demonstration, the drinks were taken out of the tray by the customer with no difficulty.

The demonstration showed the drinks were safely delivered to each table without falling or spillage along the way. The safe accelerations, along with the tray design, the smooth lift of the tray, and many other factors ensured the eleventh criteria was met in the demonstration, and all drinks were seen to be carried safely without any spillage.

Along with its many features for safe driving and accessibility, the next criteria item describes how sound files were used to hep with regular tasks and help users and the environment understand the intentions. In the demonstration, there were multiple scenarios where the sound files were used. The robot successfully announced to detected objects to get out of its way. In the case of the human, the request was heard, and the human moved. In the case of the water bottle, the BarBot announced that it would be going around the object, before moving. Many more user-friendly audio files were shown in the demonstration, including a notice when drinks had arrived at the table, as well as a message to pay for the drinks. Audio could also be heard when the kill switch was activated, to let the user know that the kill switch operation was taking place.

The last criteria asked the robot to return to its housing space within a fifteen-centimetre margin for error. In the demonstration, a piece of tape was used to indicate the starting position of the robot in its housing unit before it moved up to the bar. After completing its last order, it returned to the bar. Since there were no more orders left it turned around and drove back to its housing unit. Once it arrived, it turned around so that it could be in the correct orientation for the next use. In the demonstration it was shown that the robot made it back to the original location and was within fifteen centimetres of the tape.

Evidently, all the criteria for the final demonstration were met, but there was one constraint that was made in the preliminary report that was taken out of criteria for the final project demonstration. The original plan for the BarBot stated that there would be no human interaction needed for it to properly execute all its functions and perform all its tasks. Many adjustments were made so that this could stay true, but there were certain uncontrollable factors that had to be taken care of. Although the drift was fixed in the driving and in the turns, the robot still had some inaccuracy while turning. This was due to lack of precision in the gyro sensor. Despite the countermeasures and software functions created to solve the issue, the

robot had to be slightly adjusted when returning to the bar to avoid a large angle compound for delivery to the second table. This way, when the robot began its path to deliver to a new table, it was not starting off on an angle. Aside from the one adjustment, there is no human interaction throughout its operation.

Project Plan

Original Plan

Due to the nature of this project, there was a large amount of work needed to be completed in a relatively short timeframe. Because of this the most favoured plan of attack was to divide responsibilities to work on various parts of the project at once. In-person meetings were held twice per week to assess progress, name upcoming deadlines, help group mates with any possible roadblocks in their work as well as collaboratively work on the project together for a few hours. The first meeting was used to decide a project idea as well as a tasks checklist as shown below in Figure 12.

Item	Due Date	Completed
Project Proposal	October 26	Yes
Finalize Design	November 3	Yes
Informal Presentation	Novermber 4	Yes
Finalize Hardware	November 9	Yes
Formal Presentation	November 11	Yes
Software Presentation	November 16	Yes
Finish Individual Function	November 17	Yes
Unit Testing	November 18	Yes
Integration Testing	November 22	Yes
Finalize Software	November 22	Yes
Last Demo Test	November 24	Yes
Demo	November 25	Yes
Final Report	December 6	Yes

Figure 12: Project Checklist

The above checklist provides a clear and simple description of goals and their respective deadlines. By accomplishing each task by the deadline there was enough breathing room to solve errors that arose. When preparing for presentations, in person meeting proved to be more effective than those that were online. In person meetings also showed where confusion would arise and made it easier to ensure all members understood the project goals.

The plan prioritized hardware and presentations in the earlier stages of the project. The goal was to complete the physical part of the robot quickly to allow for breathing room when the time came for software testing. Small components such as the tray were not urgently needed

since they played no role in the functions of the robot needed for software testing. Hence the rush to complete majority of the mechanical design quickly so that software development and testing was able to being while the final tweaks were made to the mechanical aspect.

The plan was created so that all functions would be written individually and passed through unit testing before starting integration. Excess time was left in the original plan to account for debugging, testing, and solving unpredicted obstacles. The project plan was made to be simple and minimize confusion to ensure all due dates were met.

Splitting Up the Tasks

Dividing the project and assigning tasks was one of the key decisions, because if done correctly the development would be exponentially easier. To increase efficiency, majority of the project was completed in group setting rather than individual. Although working as a group, each part of the project had a lead which was the individual was the most experiences. For example, Warren Cao took lead on the hardware development and the remaining members helped where needed.

The reason for in person work was to have group help readily accessible. If no help was needed, individuals worked on separate tasks that were handed out. Camron Sabahi-Pourkashani took lead for software and the robot's early functions were written to start testing. Marlon Poddalgoda worked heavily on the formal presentation for the group, writing out all the necessary information and organizing it into a PowerPoint. Dylan Finlay began taking measurements of the cups, and modeling the original 3D design for the tray, fitting the slots precisely to each cup.

Once the focus was onto the development software, a list of the functions needed was written, and the functions were assigned out separately to each member. Work was done in person to simultaneously code and test the different functions of the robot, as well some help with debugging. Working together, distinct functions were tested, and they would be put together as fit. A shared one-drive folder made accessibility easy for the team. Any member could open the main file and work on fixing the problems that others struggled with. This was necessary since one problem continued to arise after another was resolved. This is also the reasoning for why working together in-person was vital. Ideas could be shared and joined to solve each issue that came about.

Execution of the plan

Due to the short timeline given for the project, efficiency was necessary in completing every task. Since the extra parts submission was done quickly and because the mechanical build was a

faster process than expected, the development was ahead of schedule within the first week. However, like any other project, errors and constraints began to arise which delayed development back to the original predicted timeline.

In terms of software development, the first functions were completed with ease however as the functions became more complex, more time was needed for debugging, unit testing and ultimately integration testing with other functions. Considering the extra time needed, all functions were still complete by their respective deadlines. The problem responsible for falling slightly behind schedule was the development and testing of the functions to adjust and solve for drift and imprecise turning. Said functions required more time as they needed more testing to optimize the application and reduction of mechanical drawbacks such as drift in the drive. Although the more difficult functions cause slight delays in delivery the project was still delivered on time without compromising any functionality.

In the end, there were unexpected obstacles along the path of the project's development, but all tasks were completed on time with the desired outcome.

Conclusions

The BarBot was built as an autonomous drink server, designed to safely deliver drinks from the bar to its customers, with efficiency and precision. It would reduce the tasks/responsibilities for any waiter or server safely and reduced human error to zero. Too often, drinks get spilled, orders messed up, drinks forgotten, and customers become unhappy. By removing human error, customer satisfaction is guaranteed.

Once the bot turned on, it used file input to read from a list of tables that it had to deliver to. It drove itself from its home location up to the bar where the bartender loaded the drinks and sent the robot to be on its way. The BarBot drove itself to the desired table and delivered the drinks to its customers, using voice commands to interact with them. It waited for the drinks to be taken and once the customers indicated that they had paid, the bot made its way back to the bar where it then waited for the bartender to load the next round of drinks. The robot would read off the file to find out which table it was delivering to next, the full process would continue until there were no tables left to deliver to. Once that would occur, the BarBot drove itself back home and powered itself off.

The sleek and efficient design was created using a dual-tread system and a low center of gravity, to ensure movement with precise measurements and sharp turns. The robot used a scissor-lift mechanism to raise its 3D printed tray, providing better access to the drinks. The

rotating ultrasonic sensor was included in the design to detect any obstacles that entered the delivery path and be able to look around and decide on the best path to take, in avoiding that obstacle. These are just a few of the many mechanical design features that made the project stand out.

The intricate software design successfully accounted for any challenges that the robot faced. With multiple functions to correct the driving of the robot and ensure straightness in its path, the delivery was more accurate and precise. It could detect any obstacles in its path and determine on its own whether that object was stationary or moving. If the object happened to be stationary, then the robot could safely drive around it and return to its original path, with no errors. If it were detected that the tables were passed, it would return to the correct table and fix the gap that was made in the distance. All tasks and operations were autonomous, and if there were ever to be extenuating circumstances where a problem would come about, a kill switch function was added. If this button was pressed, then the robot would slow its speed down to zero and stops all tasks.

This autonomous drink server successfully completed all these functions on its own and proved that it could meet all those criteria. The BarBot was a one-of-a-kind design that could successfully take away the need for any server.

Recommendations

Although the design of the BarBot drafted effectively and efficiently, there is always room for improvement. There were no major problems when the final product was delivered however, a few minor fixes could have potentially improved the robot's capabilities.

A physical and software component that could have been added in is a touch sensor that could have been used as another prototyped payment system. The colour sensor could have been programmed to detect different bills and approve the payments before lifting the drinks to the customers. Another improvement that could have been implemented is with the delivery capabilities. It could have been designed to deliver to numerous tables before returning to the bar, using multiple arrays to store the different travel distances. This would have also required an upgrade to the size of the tray.

These recommendations, along with other ideas, would have been possible if the mechanical drawbacks, drift, and inaccurate turns, did not require as much time to solve.

Due to the circumstances given, the project was made to be a scaled down version of what it potentially could be. If the BarBot were to be used in industry, then all the components would require a rescale to be larger. This would be needed for many reasons, including the tray being able to reach table height. If this change were made, then the tray could also be made larger, to fit more drinks and fit regular sized drinks – instead of only shot glasses. Said changes would make it more realistic to want to add in software as mentioned, that allows the multiple table delivery in one trip. For industry use, it would also be recommended to use reliable hardware, especially in terms of motors and the gyro sensor. This would ensure that there is no drift and compounding angle inaccuracies

References

[1] United States Government, "SEX BY OCCUPATION FOR THE CIVILIAN EMPLOYED POPULATION 16 YEARS AND OVER," United States Census Bureau, March 2021. [Online]. Available: https://data.census.gov/table?q=B24010:+SEX+BY+OCCUPATION+FOR+THE+CIVILIAN+EMPLOYED+POPULATION+16+YEARS+AND+OVER&g=0100000US,$04000$001&tid=ACSDT1Y2021.B24010. [Accessed 25 November 2022].

Appendix A: Flowcharts and Integration Testing

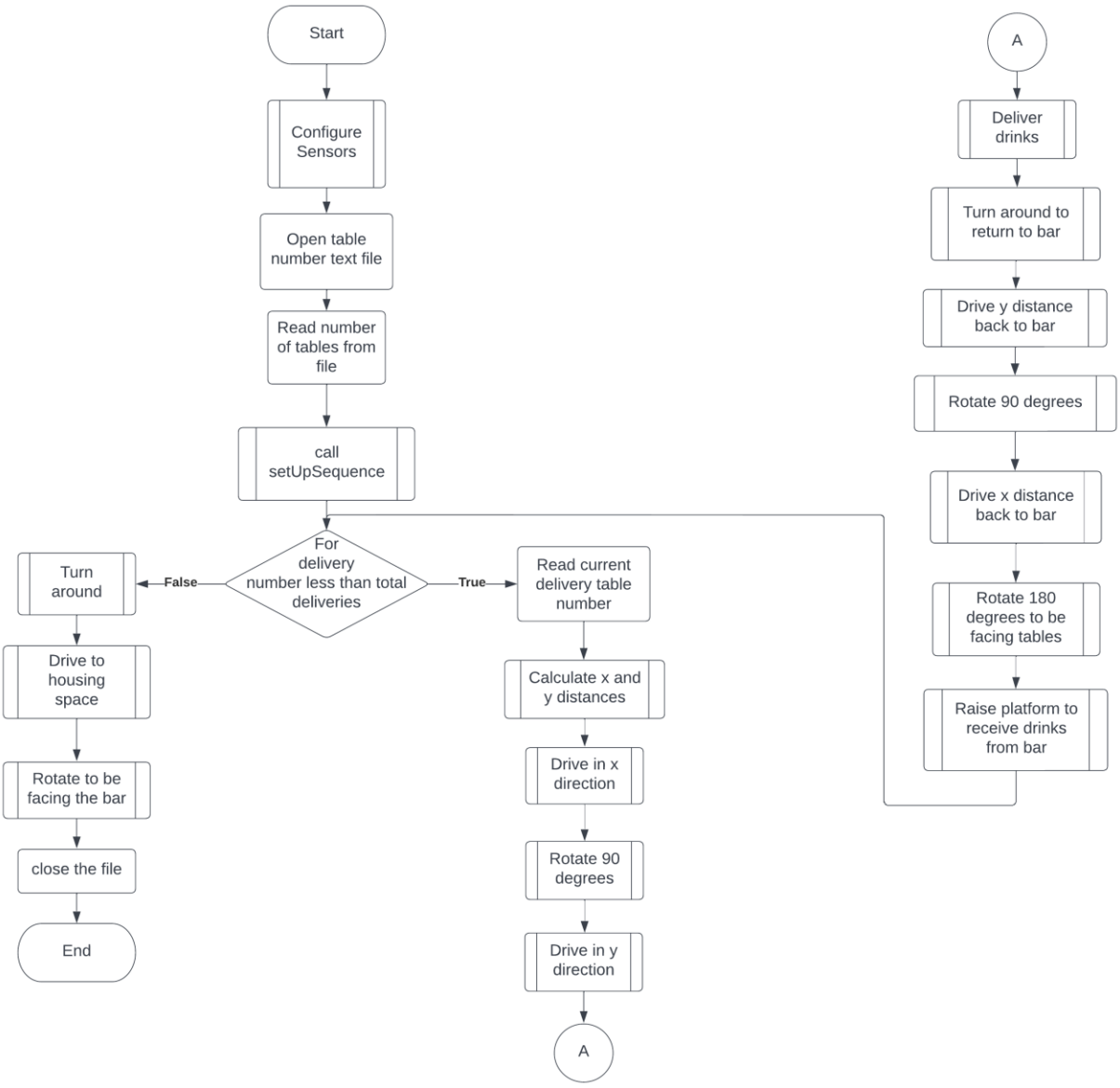


Figure 13: Task Main Flowchart

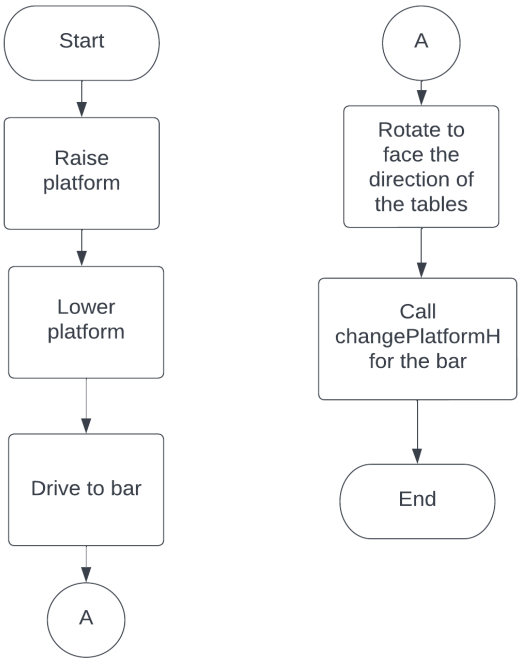


Figure 14: setUpSequence Function Flowchart

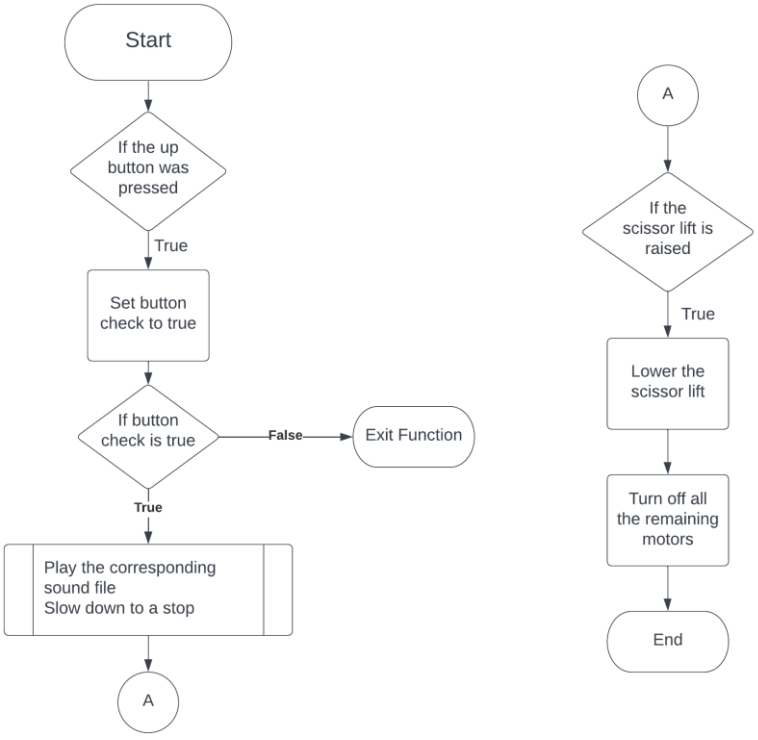


Figure 15: killSwitchCheck Function Flowchart

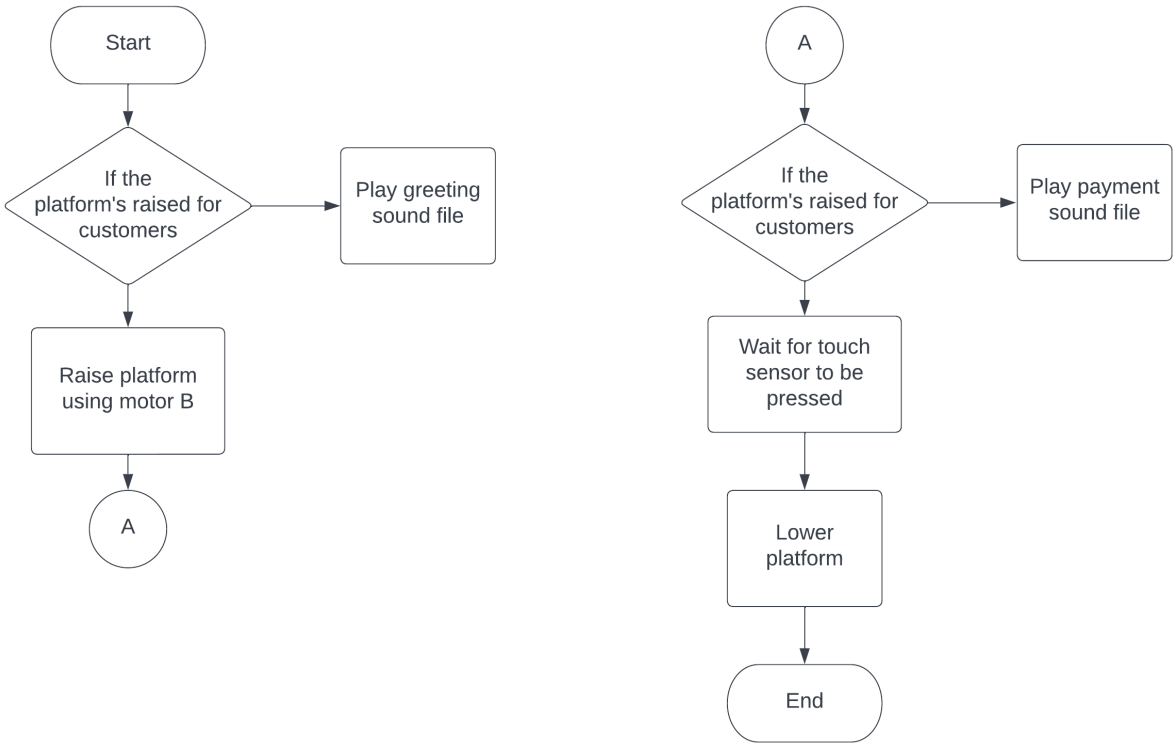


Figure 16: changePlatformH Function Flowchart

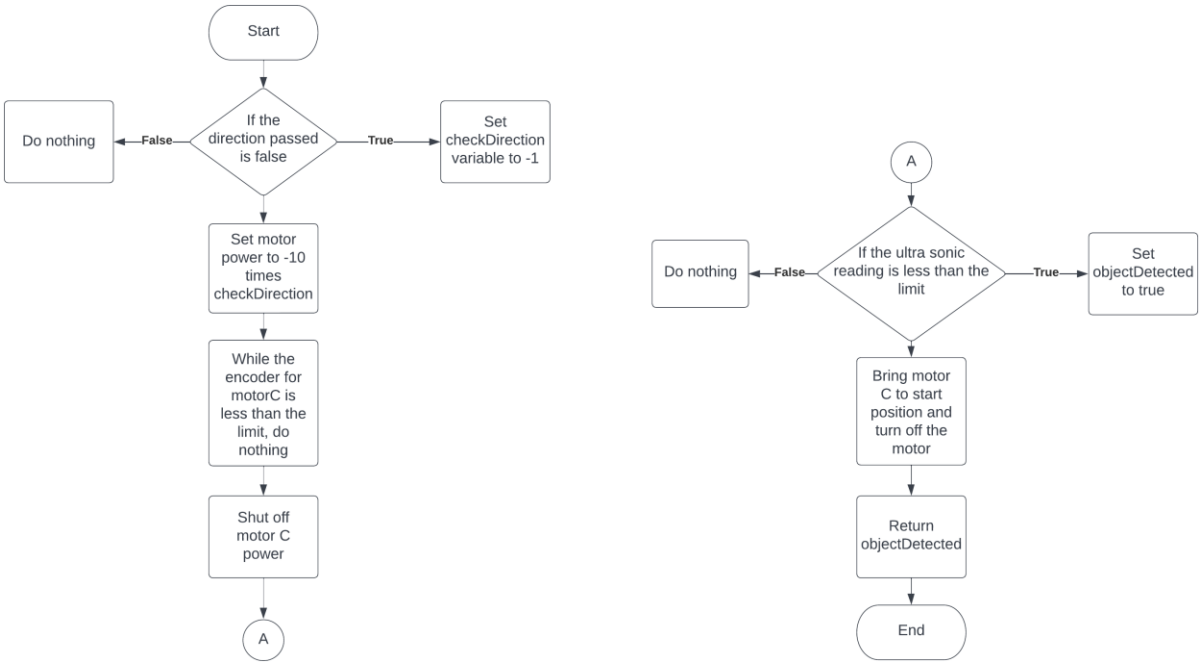


Figure 17: checkObstacle Function Flowchart

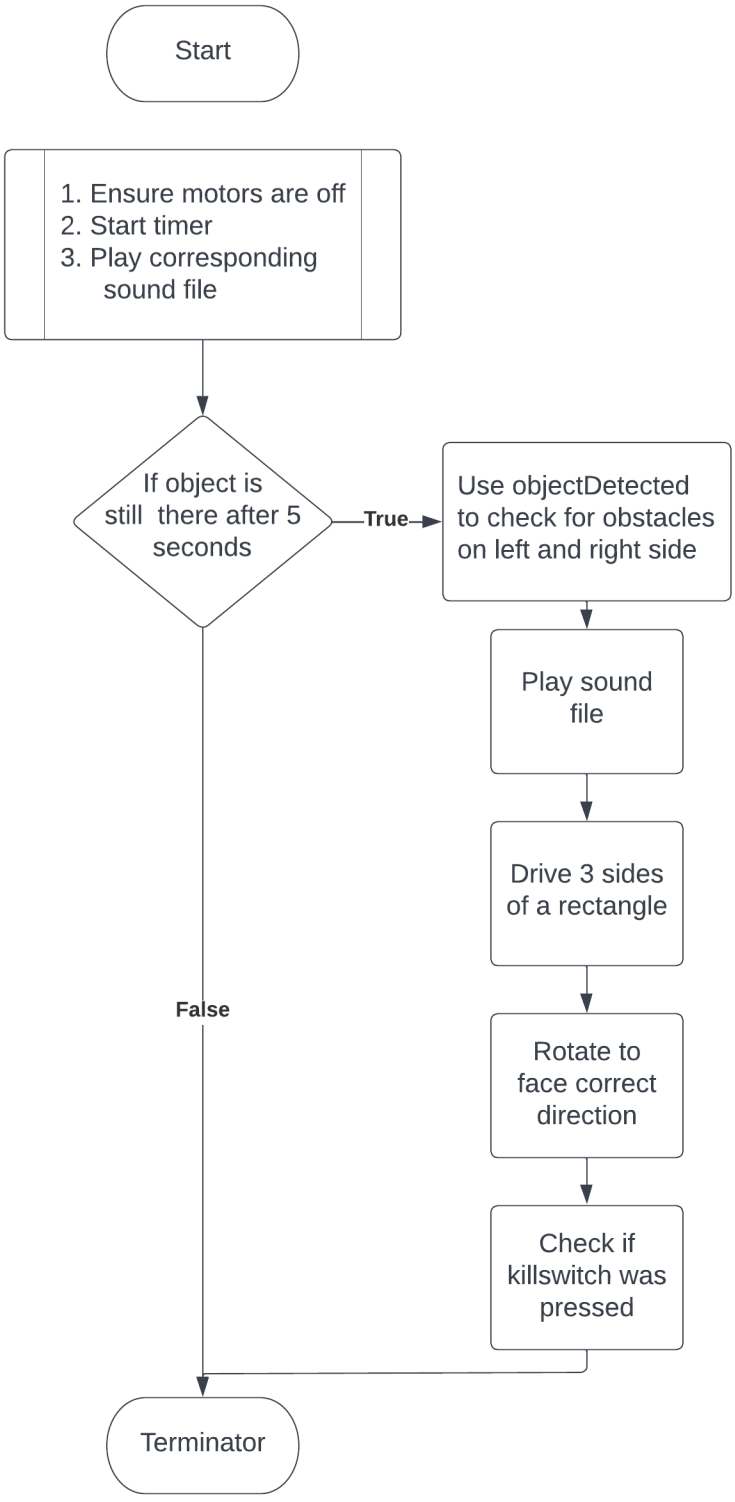


Figure 18: isObjectDetect Function Flowchart

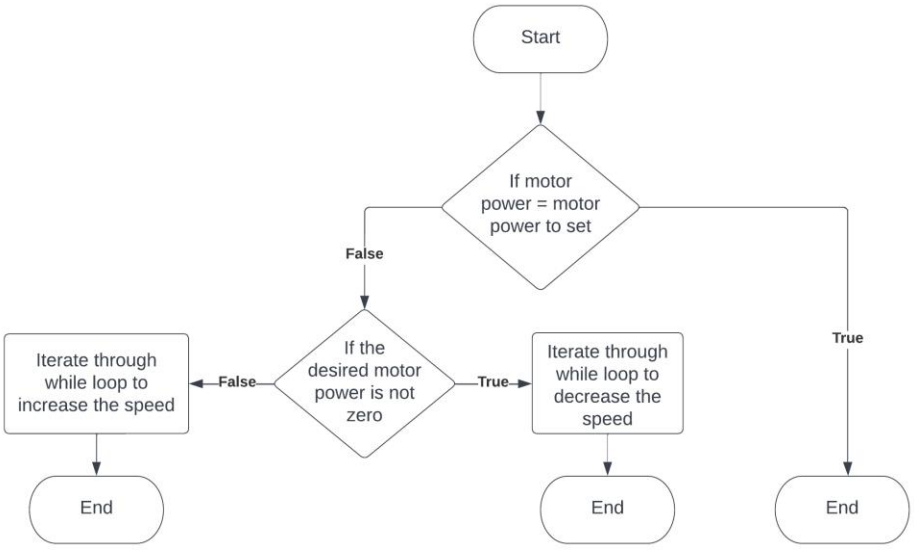


Figure 19: Accelerate Function Flowchart

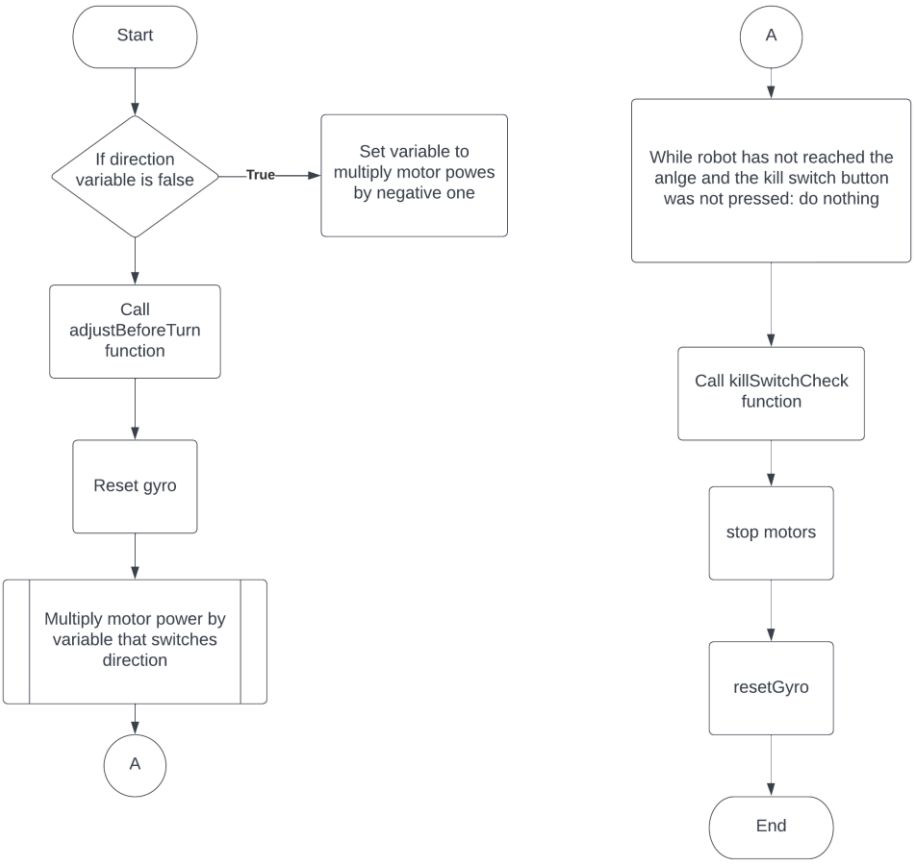


Figure 20: rotateRobot Function Flowchart

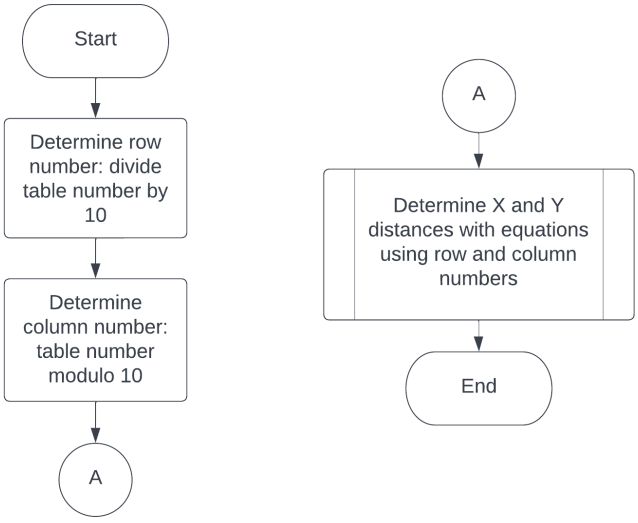


Figure 21: getDistances Function Flowchart

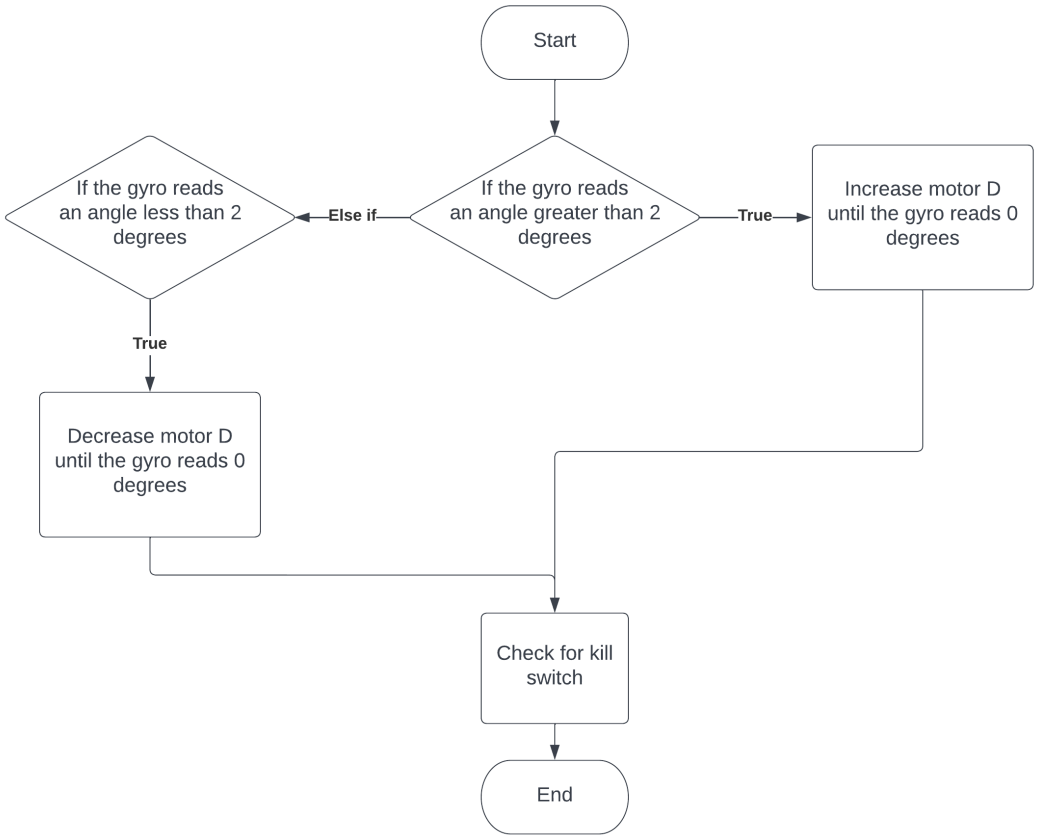


Figure 22: adjustDrift Function Flowchart

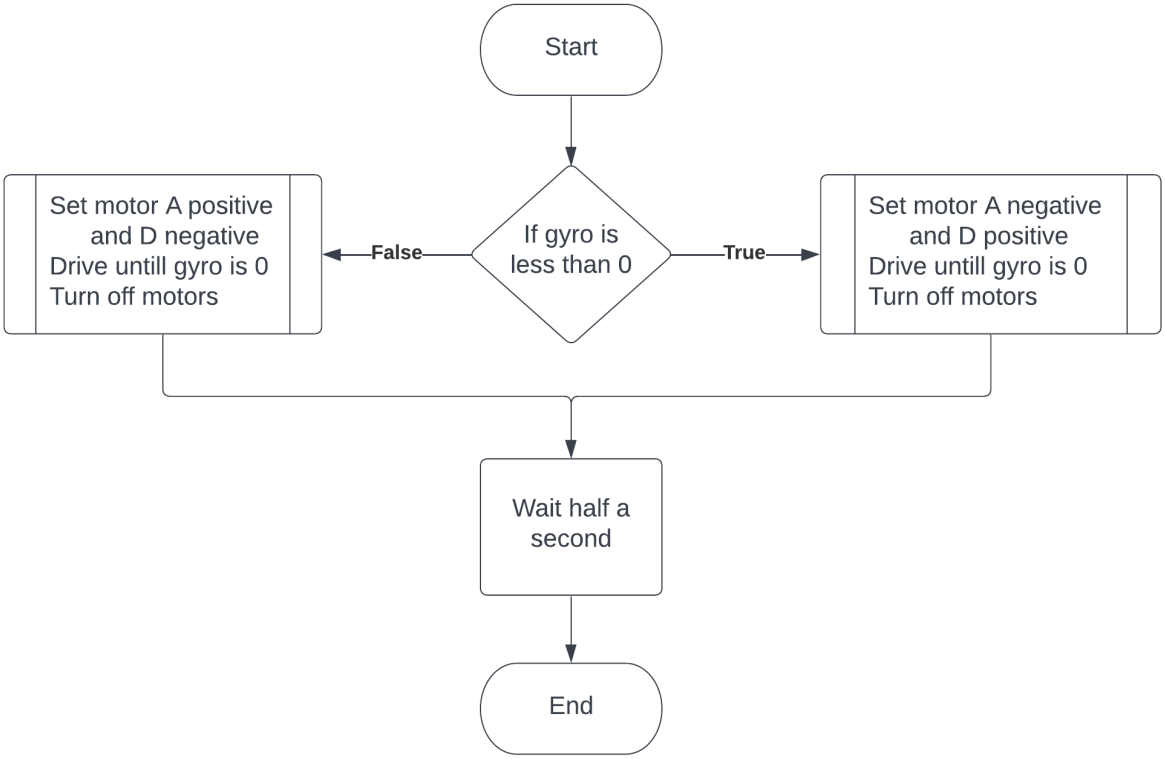


Figure 23: adjustBeforeTurn Function Flowchart

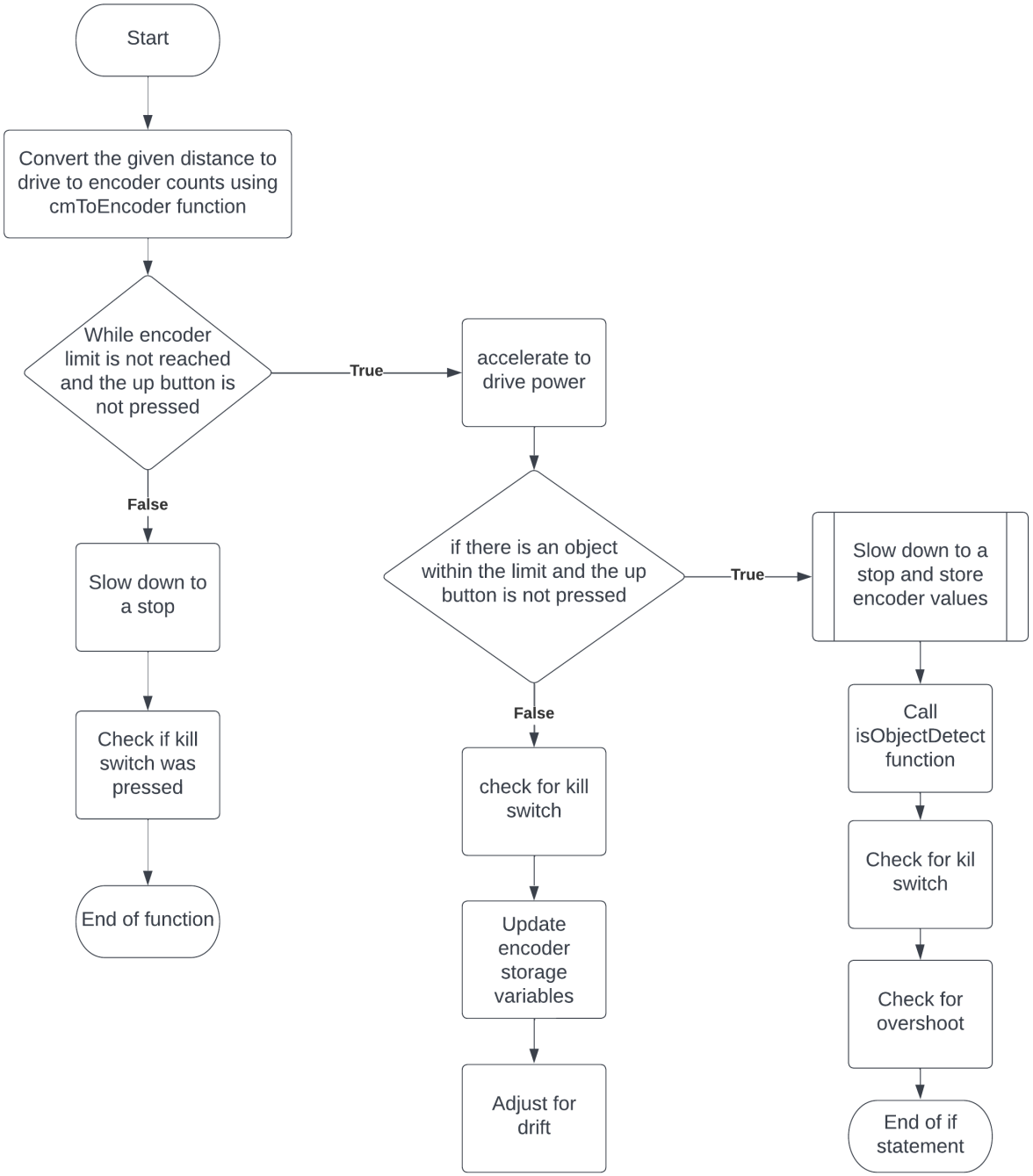


Figure 24: driveDist Function Flowchart

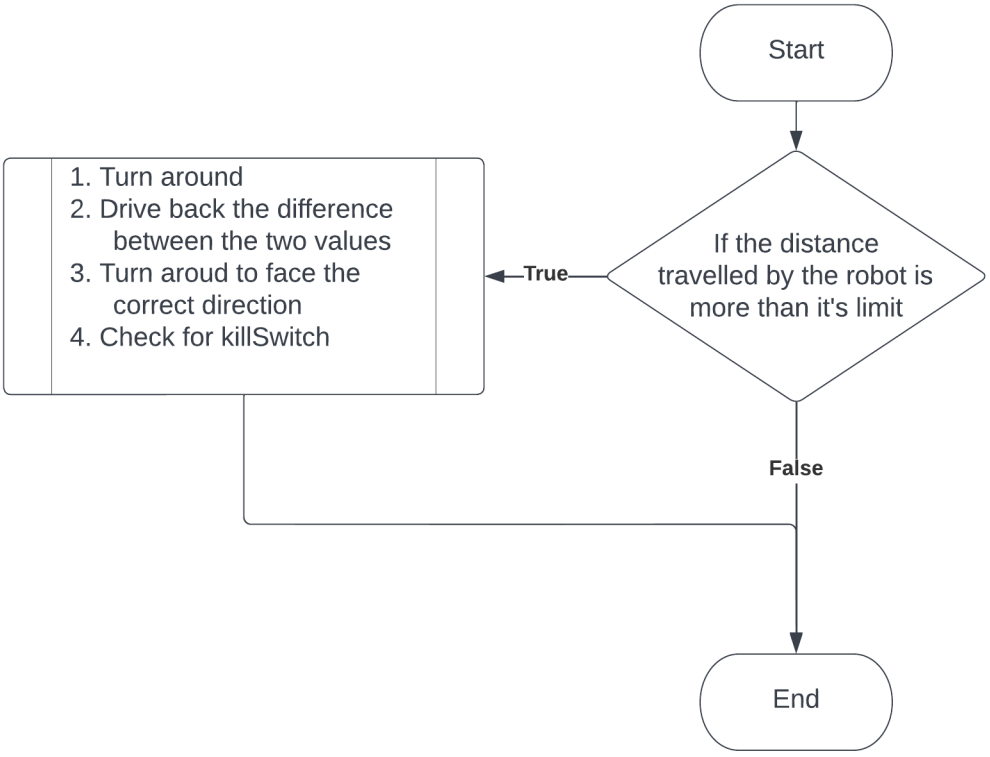


Figure 25: overshoot Function Flowchart

```
void isObjectDetect( )
{
  const int ENCODERLIMIT = 90;
  const int TURNPOWER = 25;
  const int MOTOR_POWER = 50;
  const int MIN_DISTANCE = 40;
  const int DRIVE_DIST = 60;
  // variable initialization
  bool turnCW = true;
  bool objectDetected = false;
  // reset motor encoder and stop robot
  nMotorEncoder[motorC] = 0;
  accelerate(0);
  wait1Msec(5000);
  if (SensorValue[S4]<MIN_DISTANCE)
  {
    // check left side
    objectDetected = checkObstacle( objectDetected, true);
    if (objectDetected)
    {turnCW = true;}

    // check right side
    objectDetected = checkObstacle(objectDetected, false);
    if (objectDetected)
    {turnCW = false;}

    while( abs(nMotorEncoder[motorC]) < ENCODERLIMIT)
    {}
    motor[motorC] = 0;
    // check if object is still there after 5 seconds
    if (SensorValue[S4]<MIN_DISTANCE)
    {
      // rotate robot opposite to obstacle detect
      rotateRobot( 90, TURNPOWER, turnCW);
      // drive robot in rectangle around object
      for (int count=0;count < 4; count+= 2)
      {
        nMotorEncoder[motorA] = 0;
        // drive in rectangle
        accelerate(MOTOR_POWER);
        motor[motorA] = motor[motorD] = MOTOR_POWER;
        while( nMotorEncoder[motorA] < (cmToEncoder(DRIVE_DIST)*(count+1) ) )
        {}
        motor[motorA] = motor[motorD] = 0;
        // rotate robot
        rotateRobot( 90, TURNPOWER, !turnCW);
      }
      nMotorEncoder[motorA] = 0;
      // drive
      accelerate(MOTOR_POWER);
      motor[motorA] = motor[motorD] = MOTOR_POWER;
      while( nMotorEncoder[motorA] < cmToEncoder(DRIVE_DIST) )
      {}
      motor[motorA] = motor[motorD] = 0;
      // turn robot
      rotateRobot( 90, TURNPOWER, turnCW);
    }
  }
}
```

Figure 26: Object Detection Integration Test Function

```
task main()
{
  configureAllSensors();

  // test code
  motor[motorA] = motor[motorD]=50;
  while (SensorValue[S4] > 40)
  {}
  isObjectDetect();

  // test code
  motor[motorA]= 52;
  motor[motorD]=50;
  wait1Msec(1000);
  motor[motorA]=motor[motorD]=0;
}
```

Figure 27: Object Detection Integration Test Main

Appendix B: Source Code

```
#include "PC_FileIO.c"

//Global variable initialization

const int DRIVEPOWER = 60;
const int TURNPOWER = 25;
const int RIGHT_ANGLE = 90;
const int TURNAROUND  = 178;
const bool FORCLIENT = true;
const bool FORBAR = false;
const int OBJDETECTLIMIT = 35;

/*unction given by MTE 121 course
that configures the sensors*/
void sensorConfiguration()
{
      SensorType[S1] = sensorEV3_Touch;
      SensorType[S4] = sensorEV3_Ultrasonic;
      SensorType[S2] = sensorEV3_Gyro;
      wait1Msec(50);
      SensorMode[S2] = modeEV3Gyro_Calibration;
      wait1Msec(100);
      SensorMode[S2] = modeEV3Gyro_RateAndAngle;
      wait1Msec(50);
}

/*Trivial function that takes in a distance
in centimeteres and
returns that value in encoder counts*/
float cmToEncoder( float distance )
{
      const float conversion = 180/(PI*1.75);
      return distance * conversion;
}


/*This function accelerates and
decelerates the robot
by passing the motor power,
the robot should accelerate to*/
void accelerate (int motor_power)
{
      if ( motor[motorA] == motor_power )
      {     return; }

      else if(motor_power != 0)
      {
```

```
            float i = 8.0;
            while (i >= 1.0)
            {
                  motor[motorA] = motor[motorD]
                  = (motor_power/i);
                  wait1Msec(60);

                  if (i>3.0)
                  { i--;}

                  else
                  {     i -= 0.25;}
            }
      }
      else
      {
            float i = 8.0;
            int current_power = 0;

            current_power = motor[motorA];

            while (i >= 1.0)
            {
                  motor[motorA] = motor[motorD]
                  = current_power - (current_power/i);
                  wait1Msec(80);

                  if (i>3.5)
                  { i--; }
                  else
                  { i -= 0.25; }
            }
      }
}


/*This function checks for obstacles on
either the left or the right side
depending on the state of the direction boolean*/
bool checkObstacle( bool objectDetected, bool direction)
{
      // true == CCW rotation i.e looks to the left of robot
      // false == CW rotation i.e looks to the right of robot

      int checkDirection = 1;

      const int encoderRotationLimit = 90;

      if (! direction)
      {checkDirection = -1;}
```

```
	nMotorEncoder[ motorC ] = 0;
	motor[motorC] = -10 * checkDirection;
	while( abs(nMotorEncoder[motorC]) < encoderRotationLimit)
	{}
	motor[motorC] = 0;

	if (SensorValue[S4] < 30)
	{
		objectDetected = true;
		displayString(5, "OBJECT DETECTED");
	}

	wait1Msec(500);
	motor[motorC] = 10 * checkDirection;
	while ( abs(nMotorEncoder[motorC]) > 0)
	{}

	motor[motorC] = 0;
	return objectDetected;
}

/*This functions runs the routine when the killswitch is pressed
Uses the acceleration function*/
void killSwitchCheck (bool buttonCheck)
{
	// Checks to see if killswitch button has been pressed
	if (getButtonPress(buttonUp))
	{	buttonCheck = true;}

	if (buttonCheck == true)
	{
		/*slows down the robot to a stop and stops the
		rotating ultrasonic and platform motors*/
		setSoundVolume(100);
		playSoundFile("KillSwitchActivated.rsf");
		accelerate(0);

		/* If the scissor lift is raised lower it
		to the original position*/
		if (nMotorEncoder[motorC] != 0)
		{
			motor[motorC] = -10;
			while(nMotorEncoder[motorC] > 0)
			{}
			motor[motorC] = 0;
		}
		motor[motorB]=motor[motorC]=0;

		/*displays to the user that the kill switch is
		pressed then stops the program*/
```

```
            displayString(5, "Kill switch pressed");
            displayString(9, "Barbot shutting down");
            setSoundVolume(100);
            playSoundFile("BarbotShuttingDown.rsf");
            wait1Msec(5000);
            stopAllTasks();
      }
}


void adjustBeforeTurn()
{
      if (getGyroDegrees(S2) < 0)
      {
            motor[motorD] = 10;
            motor[motorA] = -10;
            while( getGyroDegrees(S2) < 0)
            {}
            motor[motorA] = motor[motorD] = 0;
      }

      else if (getGyroDegrees(S2) > 0)
      {
            motor[motorA] = 10;
            motor[motorD] = -10;
            while(getGyroDegrees(S2) > 0)
            {}
            motor[motorA] = motor[motorD] = 0;
      }

      wait1Msec(300);

}

/*This function gets the rotation angle, motor power
to turn at and a boolean to determine which direction
the robot should rotate. it also has a killswitch
incorporated in all the different scopes of while
loops and for loops likes in every other function
throughout the project*/
void rotateRobot( int rotationAngle, int turnPower, bool turnCW )
{
      // Passing true for direction will rotate the robot clockwise
      // Passing false for direction will rotate the robot counter clockwise
      int directionFactor = 0;
      bool buttonCheck = false;

      directionFactor = 1;

      if( !turnCW )
```

```
	{	directionFactor = -1; }

	adjustBeforeTurn();

	resetGyro(S2);

	motor[ motorA ] = directionFactor * turnPower;
	motor[ motorD ] = directionFactor * turnPower * -1;
	killSwitchCheck(buttonCheck);

	while( (abs( getGyroDegrees( S2 ) ) < rotationAngle)
		&& (!getButtonPress(buttonUp)) )
	{}
	killSwitchCheck(buttonCheck);

	motor[ motorA ] = motor[ motorD ] = 0;

	resetGyro(S2);
}

/*This function checks if the robot overshot its
	marker when it
	drops out of the object avoidance scope/routine*/
void overshoot(int encoder_limit, int current_encoder)
{
	bool turnDirection = true;
	bool buttonCheck = false;
	if (current_encoder > encoder_limit)
	{
		rotateRobot (TURNAROUND, TURNPOWER, turnDirection);

		float dist_drive = current_encoder - encoder_limit;

		nMotorEncoder[motorD] = 0;

		motor[motorA] = motor[motorD] = 50;

		while((nMotorEncoder[motorD] < dist_drive )
			&& (!getButtonPress(buttonUp)))
		{}

		killSwitchCheck(buttonCheck);
		motor[motorA] = motor[motorD] = 0;
		rotateRobot (TURNAROUND, TURNPOWER, turnDirection);
	}
}


/*
This function runs the routine if an object is detected
```

```
Uses checkObstacle and rotateRobot function. The function
also return the distance it travelled on the long side of
the recgtangle it drives to adjust the encoderCount
when it breaks out of the function because this function
is used in the driveDist function*/
int isObjectDetect()
{
      const int TURNPOWER = 20;
      const int OBJAVOIDDIST = 45;
      int avoidedObject = 0;
      bool turnCW = true;
      bool objectDetected = false;
      bool buttonCheck = false;

      // reset motor encoder and ensure robot stopped
      nMotorEncoder[motorC] = 0;
      motor[motorA]=motor[motorD] = 0;
      clearTimer(T1);

      playSoundFile("MoveOutOfTheWay.rsf");
      while(time1[T1] < 5000)
      {}

      if (SensorValue[S4] < OBJDETECTLIMIT)
      {
            avoidedObject = 2;
            // check left side
            objectDetected = checkObstacle( objectDetected, true);
            if (objectDetected)
            {turnCW = true;}

            // check right side
            objectDetected = checkObstacle(objectDetected, false);
            if (objectDetected)
            {turnCW = false;}
            setSoundVolume(100);
            playSoundFile("IGoAround.rsf");
            // rotate robot opposite to obstacle detect
            rotateRobot( RIGHT_ANGLE , TURNPOWER, turnCW);

            // drive robot in rectangle around object
            for (int count=0; (count < 2) && (!getButtonPress(buttonUp));
count++)
            {
                  nMotorEncoder[motorA] = 0;
                  // drive in rectangle
                  accelerate(DRIVEPOWER);
                  motor[motorA] = motor[motorD] = DRIVEPOWER;
                  killSwitchCheck(buttonCheck);
```

```
                  while( ( nMotorEncoder[motorA] < (
                  cmToEncoder(OBJAVOIDDIST)*(count+1) ) )
                        && (!getButtonPress(buttonUp)) )
                  {}
                  killSwitchCheck(buttonCheck);

                  motor[motorA] = motor[motorD] = 0;

                  // rotate robot
                  rotateRobot( RIGHT_ANGLE , TURNPOWER, !turnCW);

                  killSwitchCheck(buttonCheck);
            }

            nMotorEncoder[motorA] = 0;
            // drive
            accelerate(DRIVEPOWER);
            motor[motorA] = motor[motorD] = DRIVEPOWER;
            while( (nMotorEncoder[motorA] < cmToEncoder(OBJAVOIDDIST))
                  && (!getButtonPress(buttonUp)) )
            {}
            killSwitchCheck(buttonCheck);
            motor[motorA] = motor[motorD] = 0;

            // turn robot
            rotateRobot( RIGHT_ANGLE , TURNPOWER, turnCW);

            killSwitchCheck(buttonCheck);
      }
      killSwitchCheck(buttonCheck);

      return OBJAVOIDDIST * avoidedObject;
}

void changePlatformH( bool forWho)
{
      const int RAISEPOWER = 10;
      const int ENCODERLIMIT = 894;
      bool buttonCheck = false;

      wait1Msec(1000);
      if (forWho)
      {
            setSoundVolume(100);
            playSoundFile("DrinkThis.rsf");
      }

      nMotorEncoder[ motorB ] = 0;
      motor[motorB] = RAISEPOWER;
```

```
	while ((abs(nMotorEncoder[motorB]) < ENCODERLIMIT)
		&& (!getButtonPress(buttonUp)))
	{}
	killSwitchCheck(buttonCheck);
	motor[motorB] = 0;

	// Waits for a tap, this confirms that all the drinks have been removed
	if (forWho)
	{
		setSoundVolume(100);
		playSoundFile("GiveMeMoneyLoud.rsf");
	}

	while (SensorValue[S1] == 0)
	{}
	while(SensorValue[S1] == 1)
	{}
	resetGyro(S2);
	//return motor back to low position
	motor[motorB] = -10;
	while ((abs(nMotorEncoder[motorB]) > 0)
		&& (!getButtonPress(buttonUp)))
	{}
	killSwitchCheck(buttonCheck);
	motor[motorB] = 0;
}


void adjustDrift()
{
	bool buttonCheck = false;

	if (getGyroDegrees(S2) < -2)
	{
		motor[motorD] = DRIVEPOWER + 20;
		while( (getGyroDegrees(S2) < 0)
			&& ( !getButtonPress(buttonUp)))
		{}
		killSwitchCheck(buttonCheck);
		motor[motorD] = DRIVEPOWER;
	}

	if ( getGyroDegrees(S2) > 2 )
	{
		motor[motorD] = DRIVEPOWER - 20;
		while( (getGyroDegrees(S2) > 0)
			&& ( !getButtonPress(buttonUp)))
		{}
		killSwitchCheck(buttonCheck);
		motor[motorD] = DRIVEPOWER;
```

```
	}
}


/*This function drives the robot a certain distance
	it uses isOBjectDetect and other functions like acceleration to
	smoothly drive the robot while avoiding objects*/
void driveDist( int dist2Drive, int drivePower )
{
	const int OBJTOL = 50;
	int encoderLimit = 0;
	int encoderCount = 0;
	int prevEncCount = 0;
	bool buttonCheck = false;

	encoderLimit = cmToEncoder(dist2Drive);

	nMotorEncoder[motorA] = 0;

	while( ( encoderCount < encoderLimit )
		&& ( !getButtonPress(buttonUp) ) )
	{
		int avoidedDist = 0;
		accelerate( drivePower );
		motor[motorA] = motor[motorD] = drivePower;

		if ( (SensorValue[S4] < OBJDETECTLIMIT)
			&& ( !getButtonPress(buttonUp) )
			&& (encoderCount < (encoderLimit - cmToEncoder(OBJTOL) ) )
)
		{
			accelerate(0); // slow down to speed 0 i.e stop
			avoidedDist = isObjectDetect();
			encoderCount += cmToEncoder(avoidedDist);
			prevEncCount = 0;
			nMotorEncoder[motorA] = 0;
			killSwitchCheck(buttonCheck);
			overshoot(encoderLimit, encoderCount);
			nMotorEncoder[motorA] = 0;
		}

		killSwitchCheck(buttonCheck);
		encoderCount += nMotorEncoder[ motorA ] - prevEncCount;
		prevEncCount = nMotorEncoder[ motorA ];
		adjustDrift();
	}

	accelerate(0);
	motor[motorA] = motor[motorD] = 0;
	killSwitchCheck(buttonCheck);
```

```
}

/*All the following constants are in centimeters
      This function assumes the distance between tables
      are the same as the table widths and lengths */
void getDistances(int tableNumber, float &xDist, float &yDist)
{
      const int D_BW_TBL = 100;
      const int TABLE_W = 75;
      const int TABLE_L = 75;
      const float HALF = 0.50;
      int row = 0;
      int col = 0;

      row = tableNumber / 10;
      col = tableNumber % 10;
      xDist = row * D_BW_TBL + ( row - 1 ) * TABLE_W - (HALF * D_BW_TBL);
      yDist = col * D_BW_TBL + ( col - 1 ) * TABLE_L + (HALF * D_BW_TBL);
}


void setUpSequence( int housingY )
{
      const int RAISEPOWER = 10;
      const int ENCODERLIMIT = 915;

      wait1Msec(1000);
      nMotorEncoder[ motorB ] = 0;
      motor[motorB] = RAISEPOWER;

      while (abs(nMotorEncoder[motorB]) < ENCODERLIMIT)
      {}
      motor[motorB] = 0;

      wait1Msec(1000);

      motor[motorB] = -10;
      while(abs(nMotorEncoder[motorB]) > 0)
      {}
      motor[motorB] = 0;

      driveDist( housingY, DRIVEPOWER);
      rotateRobot( RIGHT_ANGLE, TURNPOWER, false);
      changePlatformH(FORBAR);
}


task main()
{
      sensorConfiguration();
```

```
resetGyro(S2);

/*switched all turnCW to the opposite sign ie
true changed to false and false changed to true
Constant and variable initialization*/
const int lengthOfRobot = 40;
const int HOUSINGY = 20 + (lengthOfRobot/2);
float xDist = 0;
float yDist = 0;
bool turnCW = true;

//File initialization
TFileHandle FileIn;
bool fileOpened = openReadPC(FileIn, "tables_txt.txt");
if (!fileOpened)
{
      displayString(5, "File Opening Error");
      wait1Msec(5000);
      stopAllTasks();
}

int numOrders = 0;
readIntPC(FileIn, numOrders);
displayString(2, "%d tables to deliver to", numOrders);

setUpSequence( HOUSINGY );

for( int orderNum = 0; orderNum < numOrders; orderNum++)
{
      int tableNumber = 0;
      readIntPC( FileIn, tableNumber);
      displayString(7, "Table number %d", tableNumber);
      wait1Msec(500);

      getDistances(tableNumber, xDist, yDist);

      displayString(1, " xDist = %d", xDist);
      displayString(3, "yDist = %d", yDist);

      //These lines drive to the table
      driveDist( xDist , DRIVEPOWER);
      turnCW = true;
      rotateRobot( RIGHT_ANGLE, TURNPOWER, turnCW);
      driveDist( yDist, DRIVEPOWER);

      //Customer Delivery
      changePlatformH(FORCLIENT);

      //Turn around and to return to the bar
      rotateRobot( TURNAROUND, TURNPOWER, turnCW);
```

```
            driveDist( yDist, DRIVEPOWER );
            turnCW = false;
            rotateRobot( RIGHT_ANGLE, TURNPOWER, turnCW );
            driveDist( xDist, DRIVEPOWER );

            //Rotate 180 to be facing the tables again
            rotateRobot( TURNAROUND, TURNPOWER, turnCW );
            changePlatformH(FORBAR);
      }

      //Shutdown sequence

      //rotateRobot( TURNAROUND, TURNPOWER, turnCW );

      turnCW = false;

      rotateRobot( RIGHT_ANGLE, TURNPOWER, turnCW);

      driveDist( HOUSINGY, DRIVEPOWER);


      //Turn to be facing forward for next start

      rotateRobot( TURNAROUND, TURNPOWER, turnCW);


      closeFilePC( FileIn );

}
```