BobaBot - Boba Delivery Service

UNIVERSITY OF WATERLOO

Department of Mechanical and Mechatronics Engineering

A Report Prepared For:

The University of Waterloo

MTE 100 and MTE 121

Prepared By:

Group #3

Joey Maillette

Ethan Ahn

Karthigan Uthayan

Yuming He

Date: Tuesday, December 6, 2022

Summary

Students at the University of Waterloo are notorious for their studying habits. As a result, many become dehydrated, less motivated, or waste precious study time looking for something to quench their thirst. This is where BobaBot comes in, a Boba delivery robot. The BobaBot automates the delivery process typically done by a waiter or a deliveryman. The robot takes a drink from the user and delivers it to the selected table. Once the student or customer pays using a color-coded client card, the robot gives them the drink and returns to the starting position, completing its task.

BobaBot was constructed using LEGO Mindstorms EV3 Kit including motors, sensors, and the EV3 Brick which acted as the microcontroller, and Tetrix which was used to construct the chassis. The BobaBot software was coded using the text based RobotC programming language; its main program consisted of navigation, UI, motion, and payment processing code.

This report highlights the scope, criteria, mechanical design, and software implementation of the BobaBot project. BobaBot was able to meet all criteria and constraints set forth by the project team and successfully completed a demo of its main functionality.

Acknowledgements

The BobaBot team would like to acknowledge the MTE 100 and MTE 121 teaching staff for their support and guidance throughout this project. We thank Martin Soliman, for providing mechanical guidance and assistance, Ryan Consell, for providing software design considerations and guidance, and the WEEF TA office for consistent support throughout the project.

The team would also like extend our gratitude to Professor Sanjeev Bedi and Professor William Melek for organizing and designing the program and giving us this opportunity to apply what we have learned in the Mechatronics Engineering program so far.

Table of Contents

Table of Figures

Table of Tables

Introduction

Students at the University of Waterloo are notorious for their studying habits. As a result, many become dehydrated, less motivated, or waste precious study time looking for something to quench their thirst. This problem poses the following questions:

- How might one deliver beverages to thirsty and unmotivated engineering students?
- How might one save them time and energy?

This is where BobaBot comes in; a drink delivery robot that addresses these very questions. The BobaBot automates the delivery process typically done by a waiter or a deliveryman. The robot takes a drink from the user and delivers it to the selected table. Once the student or customer pays using a color-coded client card, the robot gives them the drink and returns to the starting position, completing its task.

Figure 1: BobaBot team

Scope

Functionality

Table and Drink Selection

The robot prompts for the table number on the EV3 display, which can be cycled through using the left and right EV3 buttons. There are 6 tables in the given room, so tables 1-6 can be chosen. After the desired table number is selected using the enter button, the user will be prompted for the size of the

drink (small, medium, and large) with corresponding prices. This can also be cycled through using left and right buttons and selected with the enter button.

Drink Insertion

The robot then opens the door and waits for a drink to be inserted by the user. Once the drink is inserted and activates the touch sensor, the door is closed, securing the drink in place.

Navigation

After the drink is secured, the robot navigates to the desired table using predefined drive commands. If the robot detects an obstacle ahead of it, the robot will stop and wait for the object to be cleared. In the case of an emergency, the robot has a kill switch: the up button, which will immediately end the program. While navigating, it is imperative the robot does not “run over” any tape. In a realistic environment this would be equivalent to crashing into a table which is an illegal operation for the robot.

Payment Processing

Once the robot reaches the designated table, it prompts the customer for a payment. After a colored client card is inserted into the card reader the robot charges the corresponding account and displays the new account balance. Once the payment is processed, the program waits until the card is removed.

Delivery

If the account associated with the client card has sufficient funds to pay for the drink, the door will open, revealing the drink. The robot then waits until the user removes the drink before closing the door. If they do not have sufficient funds, an error message will be displayed on the EV3 screen, and the door will remain shut.

Reset and Shutdown Procedure

After the drink is removed, or if the customer cannot pay for the drink, the robot will return to its starting position by retracing its steps, following the node specified list. Once it returns, the bot displays a message, telling the user if the drink has been delivered or not. If there is still a drink inside, the robot opens the door and waits until it is taken out, after which it closes. Finally, it displays the updated account balances. Once the enter button is pressed, the program ends, and the shutdown procedure is completed.

Interactions and Detection

Navigation (Gyro and Motor Encoders)

The robot navigates it environment by measuring how far it travels and how much it rotates on turns. To track distance the motor encoder on the left motor was used by converting encoder ticks to distance in centimeters. To measure rotation, the gyro was used in rate and angle mode, and reset before every turn.

Obstacle Detection (Ultrasonic Sensor)

The robot avoids colliding with moving obstacles by detecting them with the ultrasonic sensor. After the object passes through the minimum distance threshold the robot stops moving and waits until the obstacle exits the threshold.

Card Detection

Card color detection allows the robot to detect if a card is inputted and check the account it corresponds to. The card color is determined by the color sensor in the color mode using a nominal value of white.

Database (File IO)

To save the account balances across multiple deliveries the robot reads and writes to a text file saved in the “rc-data” directory at the start and end of the program respectively.

UI Interaction (EV3 Buttons)

The left, right, and enter buttons are used to allow the user to select the table and drink size.

Drink Insertion (Touch Sensor

To detect if a drink is inside the door, a touch sensor was used. The sensor was mounted towards the top of the door and will activate (value 1) if a drink is put all the way into the door. Once the drink is removed, the sensor value will go back to 0.

Emergency Shutdown Procedure

At any point during the main program, in the case of an emergency the up button can be pressed, terminating the program.

Scope Changes

Reset and Shutdown Procedure

Initially, the robot was designed to deliver drinks, one at a time, until the EV3 brick was low on battery. Once a low power level was detected, the robot would move to a charging station, which was dubbed to be the “reset” phase of the robot. However, the robot was then modified to deliver a single drink to fit within the constraints of the demonstration, and as a result, the reset phase became simply driving back to the original starting position and turning off.

Straw Delivery

Another minor task that was planned initially was to deliver straws alongside the drink. However, the mechanism required was trivial and did not require additional mechanical input. Furthermore, it posed a threat to the precision of our robot due to the added weight of the extra mechanical components. Because of this, the idea was abandoned during the prototyping phase.

Constraints and Criteria

Current Criteria

A key requirement for the robot was precision. For a drink delivery robot to function effectively, it is imperative the robot can deliver drinks as close as possible to the tables desired. While navigating, the robot should not deviate off the paths that it is programmed to follow, or run over the tape, as these would correspond to driving around in unwanted areas or driving straight into a table.

Another key requirement is the robot's ability to detect objects in front of it. Ideally, the robot should stop and wait for the object to clear. The robot running into objects corresponds to it hitting a person or an inanimate moving object, neither of which it should do.

Both these measurements are quantifiable; with precision being acceptable based on how far away the delivery is from its desired position, while object detection is quantified based on if an object is detected accurately within a specific range. The exact criteria required for the robot is found below.

Table 1: List of current criteria

Criteria	Description
Precision	Robot navigates to nodes with a margin of error of 30cm
Object Detection	If a moving object goes within 10cm - 45cm in front of the robot, the robot decelerates, stops, waits, and checks if the object is there before continuing.
Illegal Moves	Robot should not drive into tables indicated by tape lines

Previous Criteria

Initially, the robot was designed to deliver straws alongside the drink. This leads to a key piece of criteria - stability. As the robot was carrying multiple things, it was relatively easy to quantify this aspect of the previous design, allowing stability to be scaled by how many things are dropped. Moreover, the requirements in the previous criteria were further separated into 3 sections: good, satisfactory, and failed, depending on the robot's excellence in the defined categories. More details of the past requirements are in Table 2 below.

Table 2: List of previous criteria

Good	Satisfactory	Failed
Nothing is dropped	< 2 straws are dropped	Cup is dropped or spilled
No obstacles hit and < 10 cm error	No obstacles hit and < 20 cm error	Obstacle collision or > 20 cm error

However, as the robot was redesigned in later stages of prototyping to not incorporate straws due to the futility of the related mechanisms, this set of criteria was removed and replaced with the set mentioned in the previous section.

Constraints

Size

The first constraint considered was the size of the demo area. Due to slight imprecisions in the EV3 hardware specifically in sensors and motor encoder values, small errors compounded over large

distances. Thus, minimizing the delivery area would minimize the impact of these imprecisions, reducing the potential error from compounding over time.

The size of the delivery area was constrained to 3.75m. The furthest table was set 3 meters away from the starting point, and then 0.75m to the left/right of that 3m point. This can be seen more clearly in Figure 1 below.

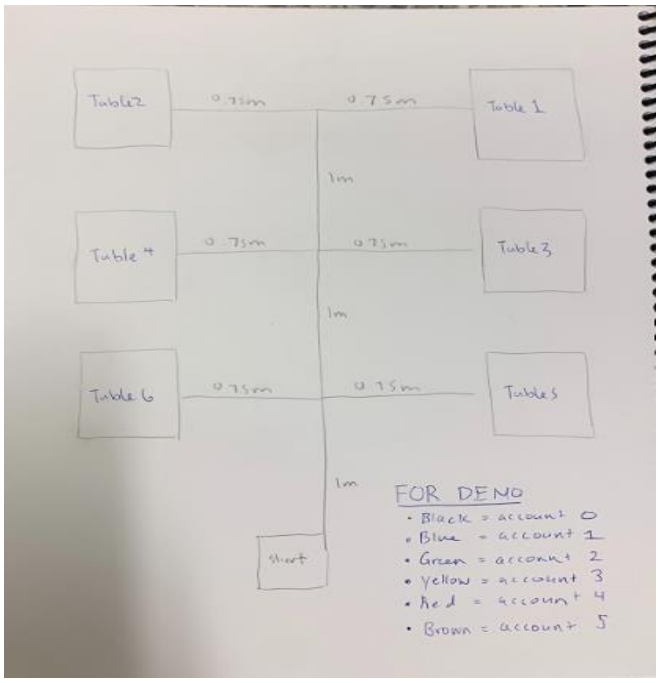


Figure 2: Delivery area layout

Payment System

The card detection system is based on the color sensor that connects the card color to an account. This means any individual with a card, or another object of similar color and size would be able to pay, even if they do not have our specific client card. This puts severe constraints on the security of our payment system, due to the limitations of the sensor used when detecting the card.

Additionally, since the color sensor can only detect 7 colors, the payment system was limited to having a maximum of 6 different accounts in its database (excluding a color used for a nominal value). They correspond to the colors black, blue, green, brown, yellow and red.

Mechanical Design and Implementation

The overall design of the robot was based on a cube shaped frame with the components attached inside of it with the purpose of the cosmetic design resembling the robot WALL-E from the Pixar movie [1]. The mechanical design elements consist of the chassis, drive system, drink storage, and card reader.

The drive system was mounted to the frame at the front and back of the robot using LEGO extensions and zip ties to ensure the frame was secured. The drink storage mechanism was mounted to the drive system facing the front to allow for easy access to the drink for the customer once it is dispensed. The card reader was mounted at the front of the frame near the top right for easier access for the card

payment, while the brick was placed on top of the robot facing upwards for the user to see the interface and for easier connection with the motors and sensors. The ultrasonic was mounted on top of the robot for aesthetic design and to make sure the readings don't interfere with the chassis of the robot.

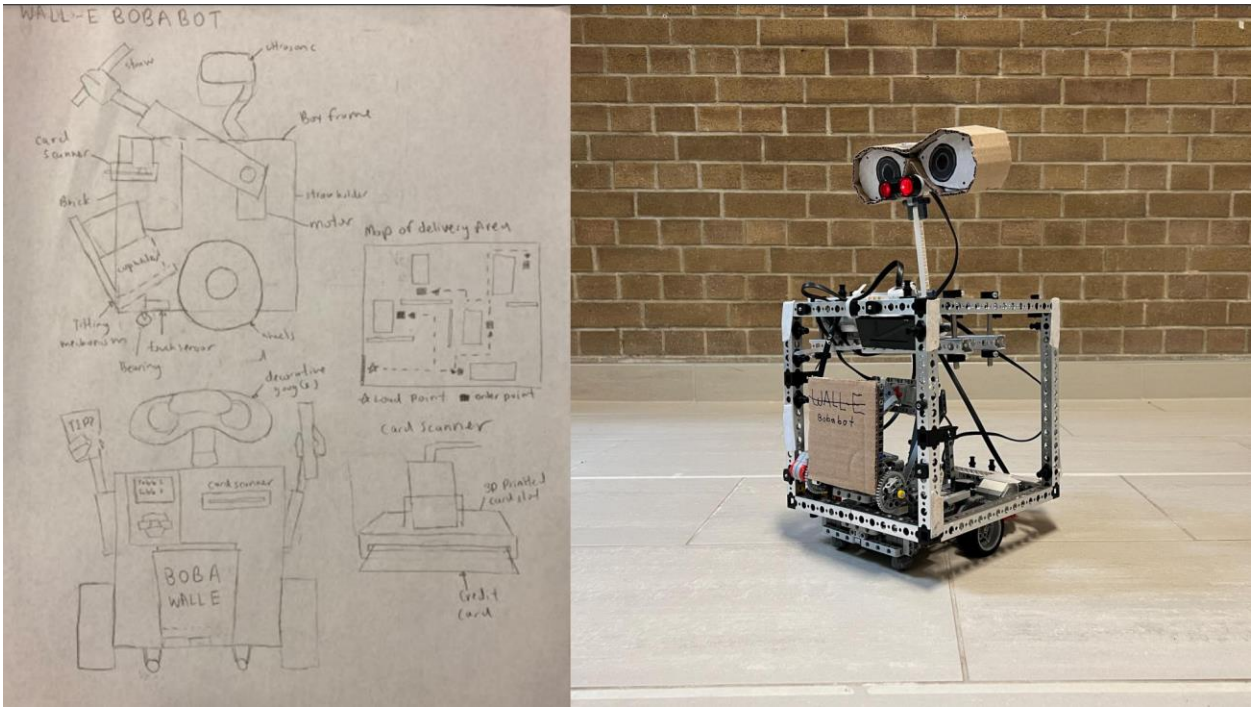


Figure 3: Side by side comparison of initial and final design

Chassis

The chassis of the robot was made from Tetrix parts in the shape of the cube with different bars mounted in order to attach the other components of the robot. The design was chosen based on inspiration of the cube shaped robot WALL-E, while giving flexibility to spacing when components had to be shifted around. The aluminum frame had the benefit of being light enough for the torque of the motors while having the structural integrity to support the other parts when the robot was moving. The frame was redesigned to make the sides smaller for balance issues for the drive system to support.

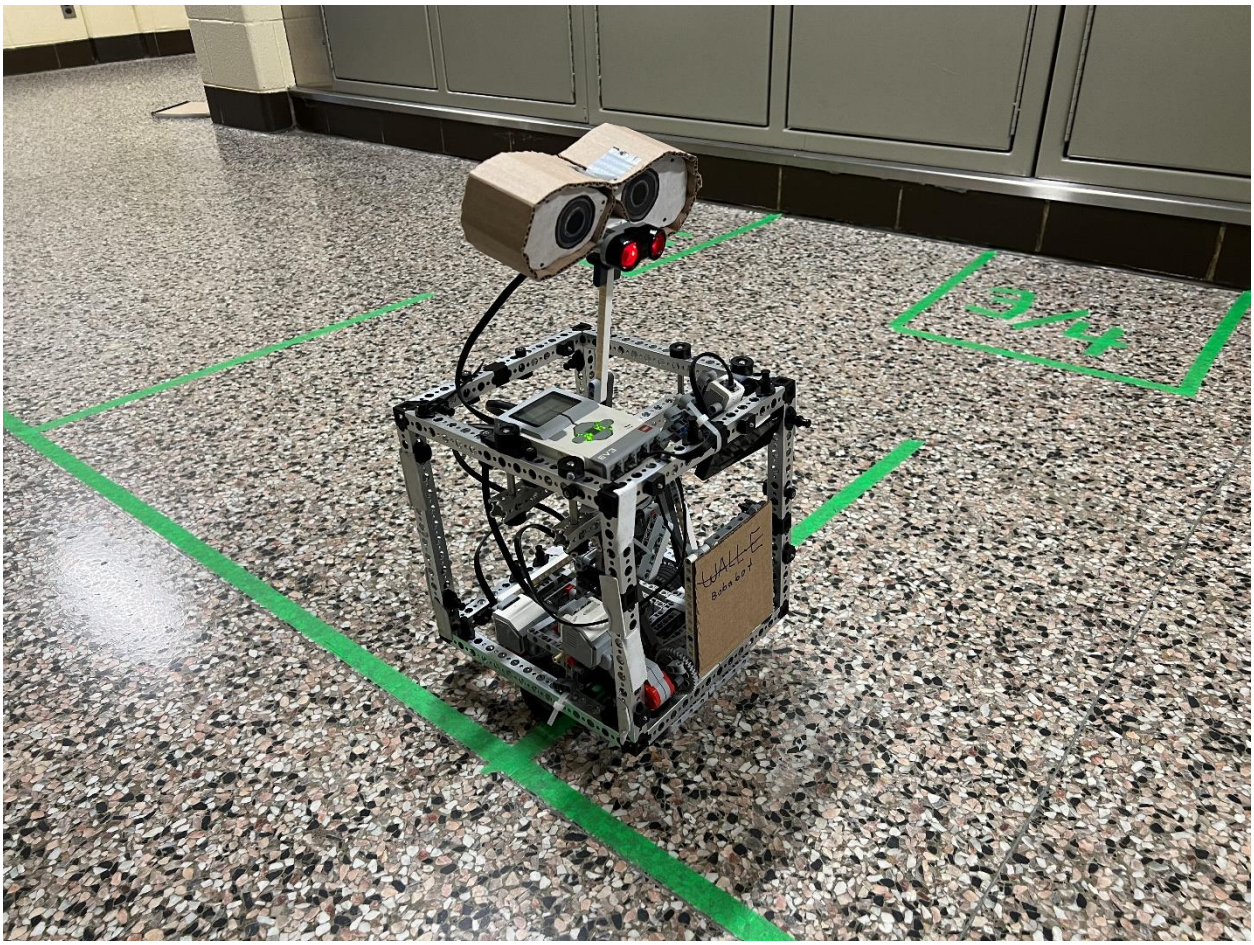


Figure 4: Chassis frame

Drivetrain

The drive system was rebuilt to have a larger base and stronger support. The motors were moved further apart from the initial drivetrain of the robot, while 3 bearings were added to the front and back of the robot in order to support the weight distribution of the mechanism. The two bearings at the front are used to support the drink storage and delivery mechanism. This was because most of the weight was concentrated at the front, especially with the weight shift when the drink is being dispensed. An extra bearing was added to the back for more stability as the weight of the drive motors would tip the robot backwards.

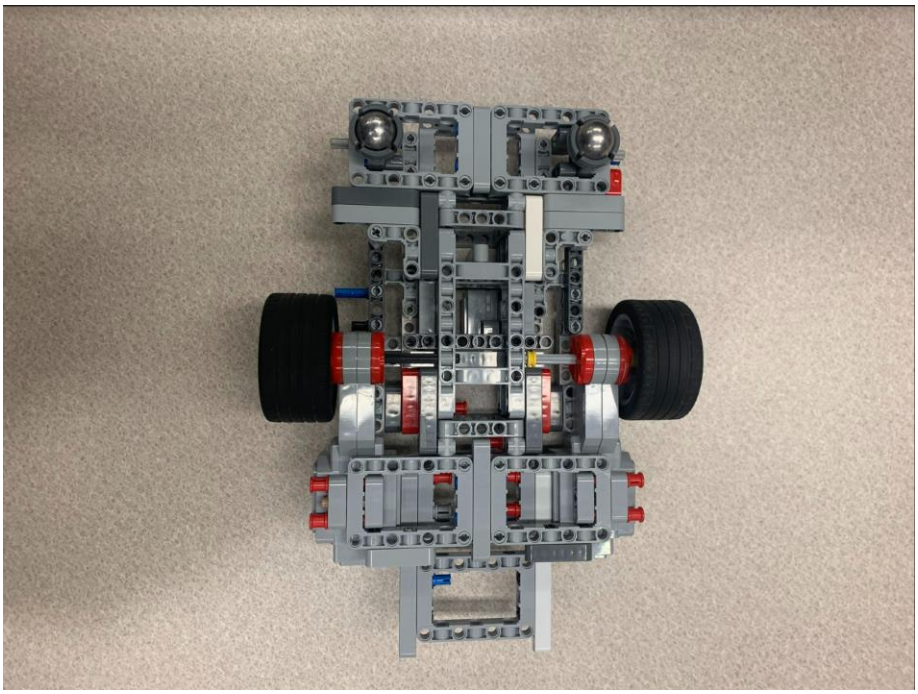


Figure 5: Bottom of drivetrain

While the gyro and ultrasonic are not exactly a part of the drivetrain, the sensor values of the two sensors are used in the drive system. Both sensors are mounted rigidly to the robot for readings to be accurate, with the gyro mounted to the base of the robot and the ultrasonic mounted on an extension above the robot facing forward to detect objects.

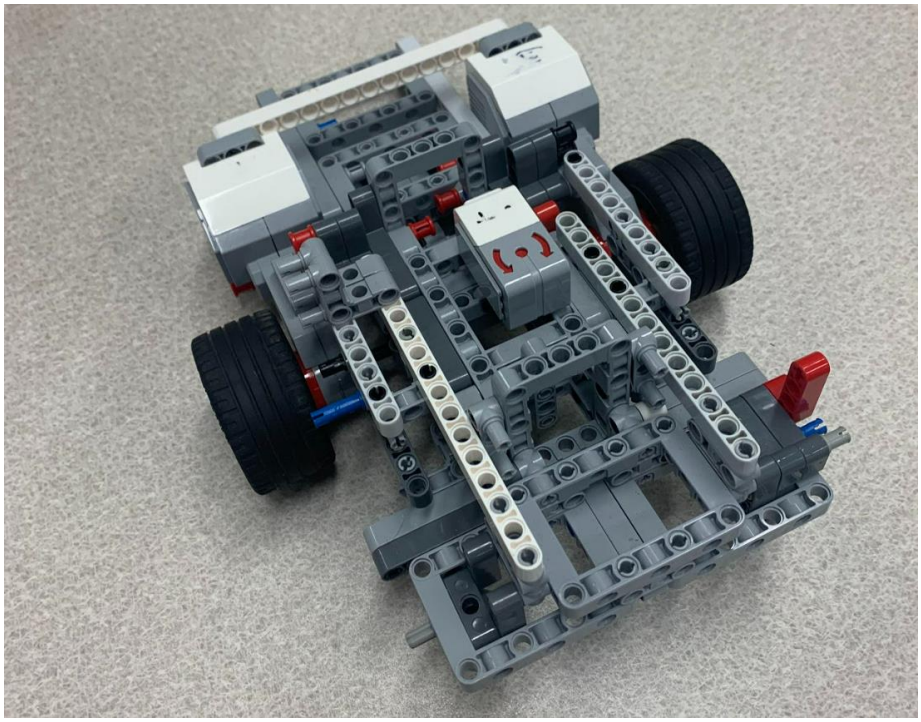


Figure 6: Gyro mounted to drivetrain

Drink Storage

The drink storage was made of LEGO. The component consists of the frame, touch sensor, and motor. The initial design involved 3D printing the frame, however it was instead constructed using LEGO to make integration with the rest of the frame easier.

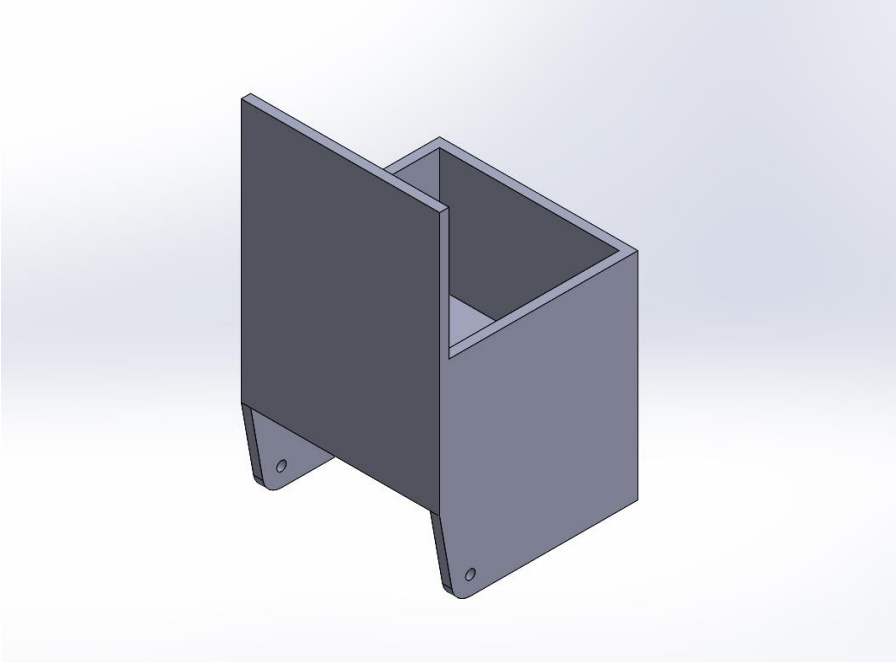


Figure 7: 3D model of drink storage

One design decision made was keeping the axis of rotation right below the frame. This was done to optimize torque from the motors to support the weight of the drink. It also makes it easier to mount to the outer frame and ensures the bottom of the frame won't rotate under the axis of rotation. A brake was also added to stop the mechanism from tipping over the angle it was supposed to. This decreased strain on the motor when the component was at its maximum extension and made it less likely to drop the drink or tip over the robot by accident.

Another design decision was to change where the touch sensor was mounted. In the original design the touch sensor was mounted at the bottom of the drink storage. This would put the axis of rotation way below the bottom of the drink, which would require more torque from the motors to support the weight. The touch sensor was then moved to the back of the drink storage to solve this problem, while giving the added benefit of the touch sensor not being released until the drink is almost completely taken out of the storage. This makes the wait between the touch sensor being released and the storage closing shorter and less ambiguous, since the customer would have already taken the drink mostly out of its storage.

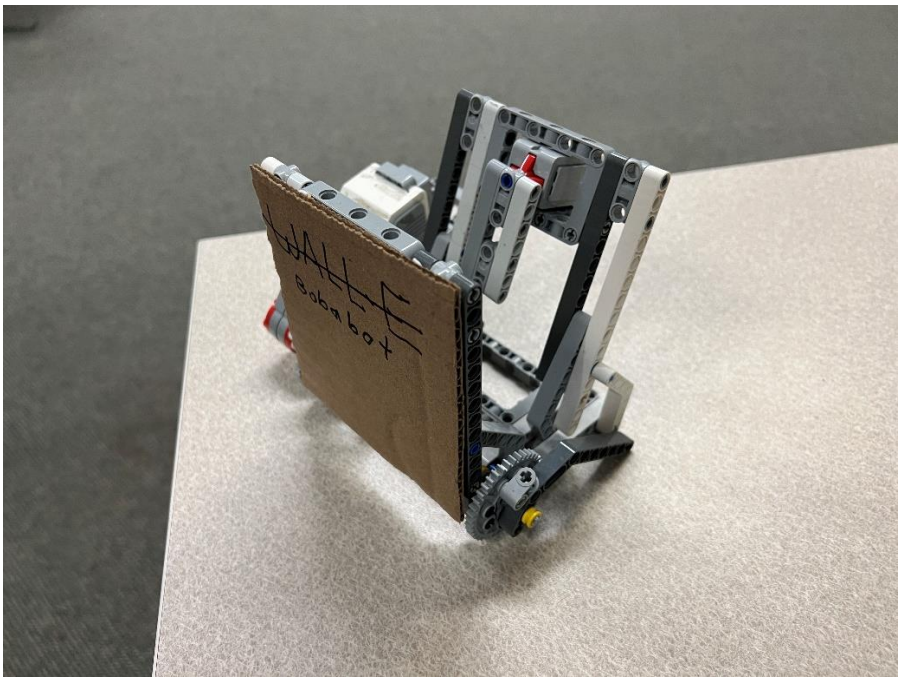


Figure 8: Drink holder with motor attached

Card Reader

The card reader uses a color sensor and a mount for cards to be inserted and processed. The card reader was designed with two main components: the first was the main frame made from LEGO with the color sensor mounted, and the second was a 3D printed interface attached to the outside of the frame in order for cards to be inserted into. The 3D printed part was made in order to make sure the card was always inserted at the right distance from the color sensor to have a viable reading. A white card was attached to the bottom of the card reader to act as a default reading with no account associated with the color, while the 6 other accounts are each associated with a distinct color: blue, red, brown, green, yellow, and black.

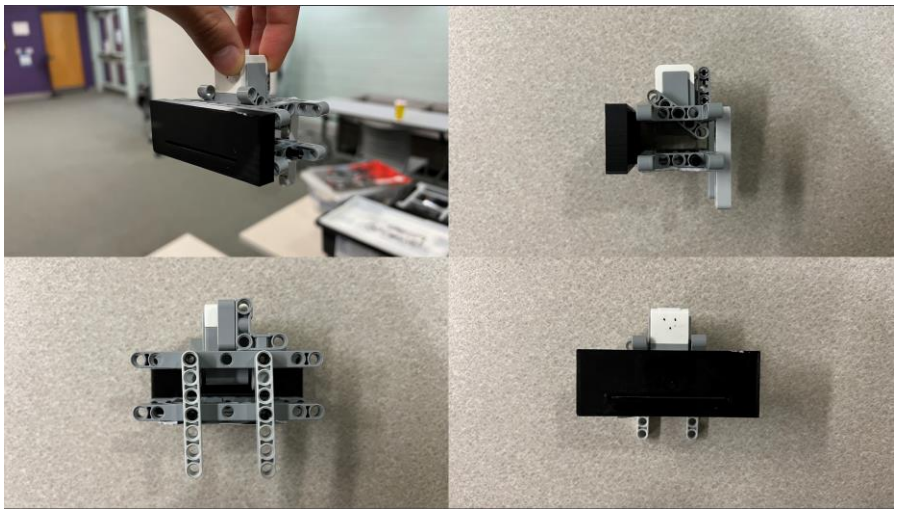


Figure 9: Card reader isometric (top-left), side (top-right), back (bottom-left), front (bottom-right)

Dimensions of the holes and spacing used to mount the part were taken using the LEGO frame. Tolerances for the diameter of the holes were approximately 0.1mm greater than the measurement to account for shrinkage when printing.

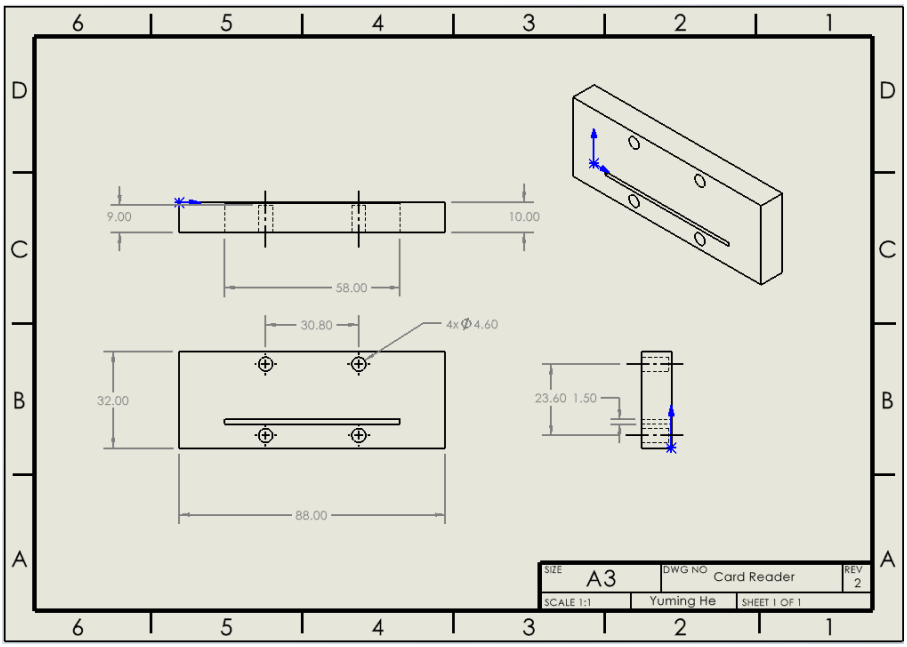


Figure 10: Drawing of 3D printed face

Outer Frame

The outer frame of the robot was made from cardboard, adding a cosmetic effect to the robot while serving the purpose of containing all of the internal frame. The frame consists of three main pieces, one piece that wraps around the robot, a square shaped panel that fits on top, and WALL-E goggles that are taped to the ultrasonic. The top panel also serves the purpose of mounting the ultrasonic. Since the ultrasonic was only mounted at the drivetrain and had to sit quite high up, the cardboard panel helped keep the ultrasonic still. The WALL-E goggles are for aesthetic design and serve no direct function.

For the demo, the outer frame was taken off the robot because it was difficult to access the charging and uploading ports of the brick.

Assembly

The overall assembly of the robot consists of different mechanisms used to mount the components to the drivetrain or chassis. It was crucial that the drivetrain and chassis were secure since all the other components are attached to them. One challenge was the integration of these two systems, as the drive system was built out of LEGO while the chassis was built out of Tetrix parts. The design decision was made to rebuild the drive system from scratch for it to match up with the chassis. Zip ties were used to secure the frame at the back while LEGO was used to attach the front.

The drink mechanism and ultrasonic were mounted directly to the drive system. It was especially important that the drink mechanism was secured to the LEGO since it was the only component other than the drivetrain powered by a motor and has a shifting center of mass. Using LEGO to integrate the system made it more secure and less likely to fall apart.

The card reader and brick were mounted to the top of the frame to make it more user accessible. Additionally, aluminum extrusions were added in order to connect the two. The LEGO and Tetrix could not be connected directly so they were secured with double sided tape and zip ties. Since these components do not need to be moved in any way, the tape and zip ties were enough to secure the parts.

The final component, the ultrasonic sensor, was secured to the drive system using the cardboard outer frame, which ensured that it remained stationary while the robot was moving.

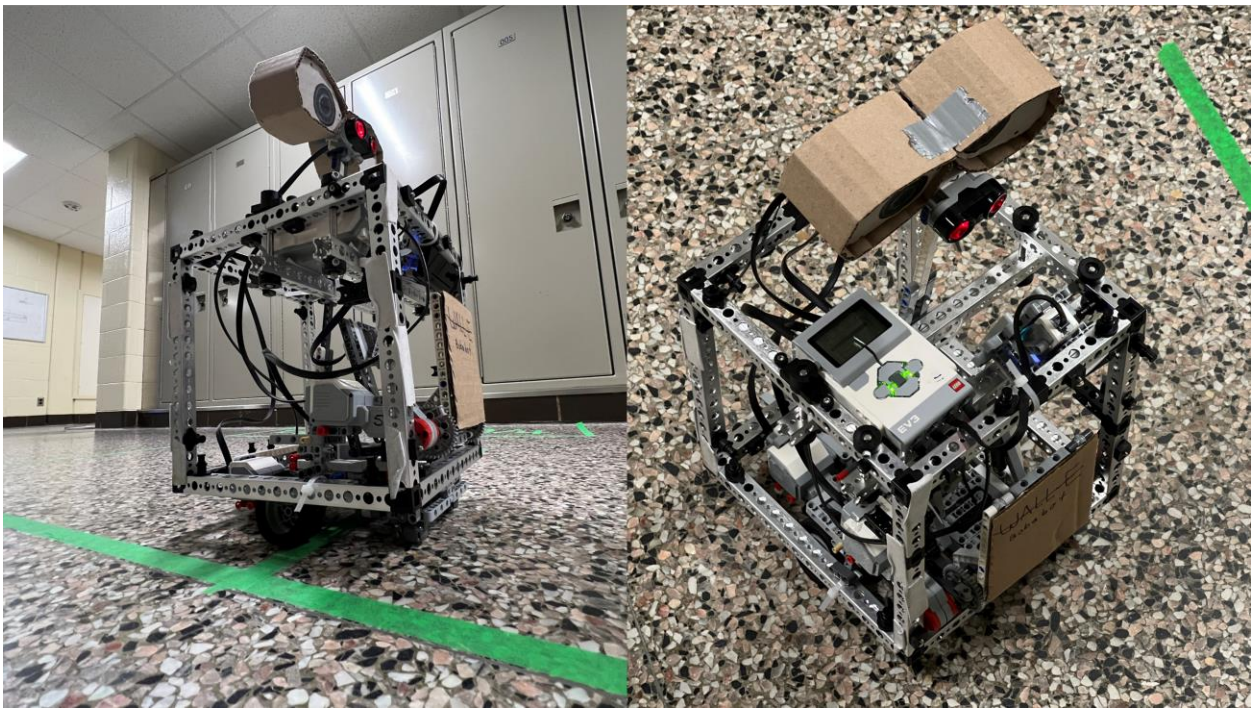


Figure 11: Full assembly of BobaBot (excluding outer frame)

Software Design and Implementation

The software for the robot is broken down into four distinct categories:

1. User Interface (UI) (written by Karthigan Uthayan)
2. Motion/Control (written by Joey Maillette)
3. Navigation (written by Ethan Ahn)
4. Payment Processing (written by Yuming He)

Alongside these major categories are helper functions such as configuration and wait functions that will be covered as a sub-category (“Control Functions”).

UI

The user interface code surrounded the interactions between the user and the robot. All of the interaction between the user and the robot was done through the buttons on the EV3 brick, where the user was able to select different inputs regarding the robot's decisions (table and drink selection).

selectTable()

The selectTable function prompts the user to select the desired table number of the screen of the EV3 and is called near the beginning of the program. The tables, starting at number 1 could be cycled through using the left button (decreases table number) and right button (increases table number), with the values updating on the EV3. The user is limited to only existing table numbers, with the program looping back to an existing table number if they went over the maximum or under the minimum. Once the user finds their desired table number, they can hit the enter button, which then erases the display and returns the integer corresponding to the table number, which is used later when the robot needs to travel to a table. (See Appendix A, line 669 for code and Appendix B.1.1 for flowchart)

selectDrink()

The selectDrink function works similarly to the selectTable function. It is called directly after the table is selected. The function prompts the user to select a drink size on the EV3 screen, from where they can cycle between small, medium and large using the left and right buttons, imitating the looping functionality described in the selectTable function. The size of drinks is stored within a local array of type string, alongside the associated price of the drinks of type float. Once the user selects the desired drink size and hits the enter button on the EV3, the display is cleared and a float with the associated price of the drink is returned. (See Appendix A, line 729 for code and Appendix B.1.2 for flowchart)

endMessage(bool paid)

The endMessage function is called at the end of the program, once the robot has returned to what it believes to be the original starting position. It displays a message depending on if the drink has been paid for or not, a value that is passed into the function. If the drink has been paid for, it will display that the drink has been delivered on the EV3 display. Similarly, if the drink is not paid for, there will be a message indicated such on the EV3 display. (See Appendix A, line 778 for code and Appendix B.1.3 for flowchart)

Motion/Control

The motion and control code surrounds all the physical movements that the robot undergoes when operating. Three of the main constraints and criteria for the problem (precision, illegal moves, and object detection) were directly linked with the precision and accuracy of the physical movements of the robot. Thus, the motion and control code were paramount in completing the main criteria.

configureAllSensors()

The configureAllSensors function was used just after waiting for the user to click the enter button upon the start of the program. Values that were configured include: the motor encoders for the drivetrain and the door and sensor modes/values for the gyro, ultrasonic, color, touch. (See Appendix A, line 169 for code and Appendix B.2.1 for flowchart)

waitButton(TEV3Buttons button)

The waitButton function is passed a specific button on the EV3 brick and waits for the button to be fully pressed and released. waitButton is used at the start of the program as well as during any UI phases of the program. (Note: waitButton is not used in the case of an emergency shutdown). (See Appendix A, line 221 for code and Appendix B.2.2 for flowchart)

conditions()

The conditions function is called after every while-loop in the program. Built into every while-loop is a condition that stops the while-loop if the kill switch has been pressed. The conditions function is then called and checks if the reason that while-loop had ended was because of the kill switch being pressed, and in that case, it calls stopTask(main) which stops the entire program at once. (See Appendix A, line 196 for code and Appendix B.2.3 for flowchart)

tooClose()

The tooClose function returns a Boolean and returns true if an object has been detected in front of the bot or false if an object isn't detected. The function is called inside of drive while-loop's and at the start of each motion function so that if an object is detected, the motion function knows to wait until the object moves. The function also plays a sound indicating an object has been detected. (See Appendix A, line 205 for code and Appendix B.2.4 for flowchart)

waitForCupIn()

The waitForCupIn function waits until a cup presses down on the touch sensor in the door then waits 2 seconds before ending the function. (See Appendix A, line 419 for code and Appendix B.2.5 for flowchart)

waitForCupOut()

The waitForCupOut function waits until the cup is removed from the door by waiting for the touch sensor to be released then waits 1 seconds before ending the function. (See Appendix A, line 432 for code and Appendix B.2.6 for flowchart)

openDoor(bool cupIn)

The openDoor function is passed a Boolean value of cupIn which is true if a cup is being put in or false if the cup is being removed. The function then opens the door to a specified angle (global constant DOOR_ANGLE) and calls either waitForCupOut() or waitForCupIn based on the value of cupIn. After one of those functions is called the door is then closed back to the closed position. (See Appendix A, line 445 for code and Appendix B.2.7 for flowchart)

motorsOn(int leftpower, int right power)

The motorsOn function is passed two integer values being desired left and right motor powers (for the drivetrain). The function then sets the respective motors to their desired power. (See Appendix A, line 234 for code and Appendix B.2.8 for flowchart)

motorsOff()

The motorsOff function sets both the left and right motor powers to 0, or off (for the drivetrain). (See Appendix A, line 241 for code and Appendix B.2.9 for flowchart)

driveDistCM(float dist, int angle)

The driveDistCM function is passed a float value for a distance (in cm) and an integer value for an angle (in degrees), with the main purpose of driving in a straight line over the desired distance.

The function first checks if the desired distance is 0 (in the case of the "0 0" node: outlined in the navigation design) which just stops the driveDistCM function completely. Then the function records the previous angle the robot was headed and resets the gyro to 0 to account for minor angular drift. The

function then calls the accelerate function (see below) and drives some distance until deceleration is necessary where it completes the desired distance. (Note: in the main while-loop gyroCorrection() is called repeatedly - see more in gyroCorrection() design)

If an object is detected during this sequence, driveDistCM will have called conditions(), waited for the object to move, then recursively calls itself passing the total desired distance subtracted by the distance already traveled, and the minor angular drift the bot has undergone. (See Appendix A, line 247 for code and Appendix B.2.10 for flowchart)

rotateRobot(int motorSpeed, int angle)

The rotateRobot function is passed an integer value for both a motor power and an angle. It then rotates the robot at that power to the desired angle in degrees. The convention for angles it uses is: 0° = forward, 90° = right, -90° = left and 180° = backwards, and it uses the same cosine ratios as accelerate() to accelerate over the turn rather than "jolt" to improve the precision and accuracy of the turn. (See Appendix A, line 383 for code and Appendix B.2.11 for flowchart)

accelerate(int motorSpeed, int sliceTime, bool acc)

The accelerate function of type integer is passed three values, an integer value for motor speed, an integer value for a slicing time, and a Boolean being true for acceleration and false for deceleration. The main purpose of this function is to smoothly accelerate or decelerate at the start and end of driveDistCM to allow for higher motion accuracy and precision.

The function is of type integer as the total distance traveled over the acceleration phase is returned into the main drive function so that it knows when to start decelerating at the exact same rate

Specifically, this function uses ratios from a transformed cosine function:

$$\%\ Motor\ Power = \frac{-cost(iteration\ count) + 1}{2}$$

to map the motor power of the drivetrain over a certain number of iterations (defined by a global constant ACCEL_FACTOR) to allow for the smoothest possible acceleration/deceleration phases. The function uses the slicing time to determine the time between iterations (the time at that motor power).

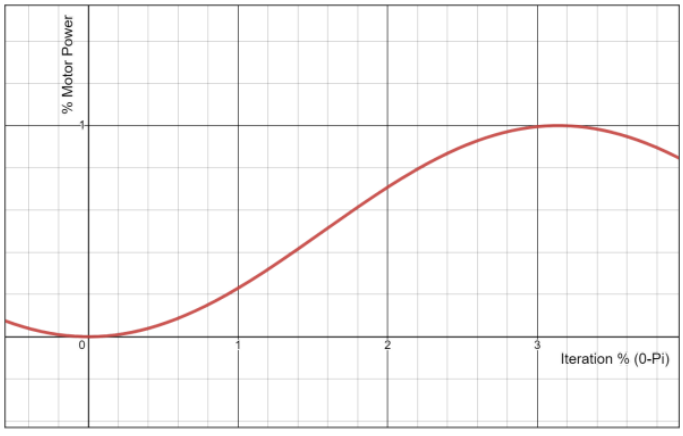


Figure 12: Graph of transformed cosine function used to map motor power [2]

(See Appendix A, line 310 for code and Appendix B.2.12 for flowchart)

gyroCorrection(int angle)

The gyroCorrection function is passed an integer value for the desired angle and is called during every iteration of drive while-loops. If the gyro has drifted off the desired value, it compensates by subtracting motor power of one side based on a constant (CORRECTION_FACTOR) and how far in degrees it has drifted. (See Appendix A, line 361 for code and Appendix B.2.13 for flowchart)

Navigation

The navigation algorithm was based on a node system, where a list of nodes produced a path for the robot to follow. Due to the limiting nature of RobotC (more on this in "Discussion") strings were used as nodes with (x,y)-pairs separated by a space. Each table was then represented by a list of strings that described the path to reach that table from a constant initial position.

driveToNode (string nextNode, string pos, int heading)

The driveToNode function was used to navigate to the next node in the table array given a current position and heading. Since the robot only traveled along paths perpendicular to each other the displacement in only either the x-direction or y-direction were considered. After calculating the change in direction and the distance the robot must travel, the function calls the rotateRobot and driveDistCM functions to move the robot to the next node. (See Appendix A, line 603 for code and Appendix B.3.1 for flowchart)

driveToTable(string *table, int nodes)

The driveToTable function navigates the robot from the initial position to a given table by iterating over each node in the table list and calling the driveToNode function. The function takes in a pointer that points to each node's coordinates and the number of nodes as an integer. (See Appendix A, line 630 for code and Appendix B.3.2 for flowchart)

returnToTable(string *table, int nodes, int last_heading)

The returnToTable function navigates the robot back from the given table to the initial position by iterating over the table list backwards. The last heading value is used to keep track of the direction the robot is facing between navigating to the table and back. After reaching the initial position the robot rotates 180 degrees. Similarly, to the previous function, returnToTable takes in the table pointer, and the number of nodes. (See Appendix A, line 649 for code and Appendix B.3.3 for flowchart)

getX(string node) and getY(string node)

These helper functions slice the node string into its x and y coordinates and convert them into float values. It does this by implementing the stringFind() and stringDelete() functions in RobotC to find the empty space and delete from or to that character in the getX() and getY() functions respectively. (See Appendix A, line 586 & 594 for code and Appendix B.3.5 and B.3.5 for flowcharts)

Payment Processing

The payment system consisted of color-coded cards that corresponded to a unique account number in a database. The accounts and balances were saved in a text document stored within the EV3's "rc-data" directory. RobotC's file input and output functions were used to read and write to the file in memory.

initialiseDB()

The initialiseDB function is a helper function used to set the initial account balances in the file. Due to issues with the EV3 file system, the file and initial values had to be written using a RobotC script rather

than manually (more on this in “Discussion”). The function opens a file stream in write mode and enters values from an array into successive lines in the file using a for loop. (See Appendix A, line 477 for code and Appendix B.4.1 for flowchart)

printDB()

The printDB function is a function to display the account balances from the database to the user on the LCD screen. The function opens a file stream in read mode and prints each line to the LCD screen using a for loop; the line number corresponds to the account number. (See Appendix A, line 491 for code and Appendix B.4.2 for flowchart)

createLocal(float *localDB)

The createLocal function fills a given array with the account balances from the database file. The function opens the file, referenced by the pointer passed into the function, in read mode and loops through each line, assigning the value to the corresponding index in the array. This produces a working copy of the database that the program can reference quickly without having to open and read a file. (See Appendix A, line 509 for code and Appendix B.4.3 for flowchart)

postLocal(float *localDB)

The postLocal function saves the current account balances stored in an array to the database file. The function opens the file, once again referenced by the pointer passed into the function, in write mode and loops through each element in the local array. It then writes over each corresponding account balance in the file with the value from the array. (See Appendix A, line 525 for code and Appendix B.4.4 for flowchart)

readCard(float cost, float *balances)

The readCard function reads the card inserted into the card reader and updates the account balance in the balances array by charging the given cost. The function waits for a card to be inserted using the color sensor and the color white as a null input. After a card is inserted the color sensor is used to read the color of the card and the corresponding account is matched. The function then checks whether the account has sufficient funds to pay for the drink. If the account has sufficient balance, then the account is charged the cost of the drink and the function returns true; otherwise, the account is not charged, and the function returns false signifying the user did not pay for the drink. Regardless of the outcome, after the payment is processed the function waits for the card to be removed before terminating. (See Appendix A, line 538 for code and Appendix B.4.5 for flowchart)

Data Storage

The two main data storage structures used within the program are the account balances stored in an array and the table paths stored in arrays.

Account Balances

The account balances are stored in two forms:

1. The local float array used in the program to update and read account balances
2. The persistent text file stored in the “rc-data” directory on the EV3

Table Paths

Table paths are stored as constant arrays of strings within the program. Strings were used to overcome the issue of 2-dimensional arrays in RobotC.

Testing

The testing of the robot occurred in five main phases which in chronological order included: motion and control testing, motion and navigation integration testing, UI testing, functionality testing, and precision testing. For the elementary testing phases, testing was done by either having a separate file with only the desired functions in it or isolating main() with only the functions being tested.

Phase 1: Motion and Control Testing

The motion and control testing consisted of testing all the functions highlighted in the motion/control software design. This testing included ensuring these functions completed their required tasks, troubleshooting any minor inconsistencies or bugs, and tweaking some global constants for higher precision before moving onto navigation testing.

Table 3: Layout of motion and control testing

Test #	Function(s)	Problem	Process/Resolution
1	waitButton	Ensuring they work with no bugs.	Make the program wait until the enter button is pressed at the start of the program.
2	conditions	How to emergency shut down main upon a button press?	Declare a global constant associated with a kill switch, check if it's been pressed in **every** while-loop, run conditions to stop main.
3	motorsOn, motorsOff, driveDistCM (Note: exclude acceleration and object detection)	Ensuring they work with no bugs.	Declare global constant converting motor encoders to distance in cm, make the program drive a certain distance.
4	rotateRobot (Note: exclude acceleration)	Ensuring it works with no bugs.	Make the program rotate the bot at various angles from (-180° to 180°)
5	waitForCupIn, waitForCupOut, openDoor	What angle should the door open at, and ensuring they work with no bugs.	Test different angles on the door and declare global constant to the optimal door angle, make the program wait for a cup to be put in, close the door, then open the door and wait for the cup to be removed.
6	acceleration, rotateRobot,	What is the optimal acceleration curve, how	Test different acceleration methods such as linear, parabolic, and trigonometric

Test #	Function(s)	Problem	Process/Resolution
	driveDistCM	to decelerate to the desired distance, and ensuring they work with no bugs.	acceleration, test different constants associated with each, make the program drive a certain distance with acceleration and deceleration phases, turn with acceleration and deceleration.
7	gyroCorrection, driveDistCM	How to correct the position of the robot in the case of angular drift, and ensuring they work with no bugs.	Test different methods of gyro correction such as decreasing one or both motor powers, or increasing one or both motor powers, make the program drive a certain distance and measure angular drift.

Phase 2: Motion and Navigation Integration Testing

The motion and navigation integration testing consisted of integrating the motion code with the navigation code and troubleshooting any minor inconsistencies or bugs. It was also ensured that the navigation code would work with any "room layout" (i.e. x and y coordinates).

Table 4: Layout of motion and navigation testing

Test #	Function(s)	Problem	Process/Resolution
1	driveToNode, driveToTable, returnToStart	Node structure was not recognized properly causing improper execution of paths.	Change code to represent points as strings and paths as an array of strings.
2	getX, getY	Functions were the returning wrong value. (getX retuned Y and getY returned X)	Switch the names of the functions.
3	driveToNode	Robot turned the wrong direction when facing in the negative x direction and navigating to a negative y value. (This was because atan2 never returns -180)	Add check if the heading is negative and change the calculation for the angle change.
4	driveToNode	Robot would turn a slight amount when delta_angle was 0.	Check if delta_angle is 0 and don't call rotateRobot.

Phase 3: UI/Payment Processing Testing

The UI and payment processing testing consisted of isolating the UI and payment processing code in its own file. Strings were displayed regarding payment data, table selection, drink selection and account information to help debug the code.

Table 5: Layout of UI and payment processing testing

Test #	Function(s)	Problem	Process/Resolution
1	selectTable	User was able to select a negative table number, or a table that was not in the scope of possibility. (Not tables 1-6)	Created a loop, where incrementing past the maximum value (6) would reset the table number to the lowest value (1). Similar thing was done with minimum, so no tables out of scope would appear.
2	selectDrink	Did not return or display the correct amount based on the user selected drink cost.	Set the function to return a float instead of an integer, and making it display to 2 decimal places.
3	endMessage	Function did not display the correct message.	Passed in Boolean for payment, changing message based on if the drink was paid for or not

Phase 4: Functionality Testing

The functionality testing consisted of integrating each of the motion/control, navigation, and UI/payment code and testing the full program from start to finish. Testing and debugging continued until the bot consistently made deliveries with all of the criteria checked.

Test #	Function(s)	Problem	Process/Resolution
1	driveDistCM, driveToNode, UI functions	Choosing which node based off the user table selection (i.e. How does one store the table data)	Ordinarily a function to choose what table would have been added, however, since RobotC doesn't allow passing 2D arrays (table nodes were stored in 2D arrays) into functions, a series of 12 if-statements were added into main to choose which table the bot would navigate to.
2	main	The bot would end each run by facing the opposite direction, requiring between runs for the bot to be	At the end of main rotateRobot was called so that it would end in the same position at the start allowing for less user interaction needed for the bot's operation.

Test #	Function(s)	Problem	Process/Resolution
		physically picked up and rotated.	
3	printDB	Database values were never displayed so the user was unable to verify if the payment processing worked.	The printDB function was added at the start and end of main so changes in account balance could be seen in real time.
4	main	Ensuring it worked with no bugs.	Made the program test every single table with every possible drink selection.

Phase 5: Precision Testing

The precision testing consisted of changing global constants relating to the motion/control and navigation code. This testing was paramount in the bot's success as it brought the bot from being within a ~20cm range on average to within a ~10cm range on average.

Table 6: Layout of precision testing

Test #	Constant(s)	Problem	Process/Resolution
1	CM_TO_DEG (cm to encoder conversion)	The robot was traveling slightly short of the desired distance.	The reason for this was because the tires were under a lot of compression from the weight and thus the bot thought it was traveling further than it actually was. Radius of the tire was decreased slightly to fix.
2	ACCEL_FACTOR (How many iterations the acceleration undergoes)	The robot was accelerating too fast and had some “jerk” motion still associated with it.	The number of iterations was increased so that there was a smoother acceleration and declaration phase.
3	CORRECTION_FACTOR (How much the motor power is decreased based on how far off the angle it has drifted)	The robot was moving in a zig-zag pattern while driving in a straight line, or in other words overcompensating for the angular drift.	The correction factor was decreased so that the bot would correct less drastically and increase the accuracy on driving straight.

Test #	Constant(s)	Problem	Process/Resolution
4	Battery Life of robot	Different values of the battery life would yield different results, such as one motor being stronger than the other and one motor starting before the other.	Adjusted all constants to be as optimal as possible on 100% battery life.

Discussion

The major issues encountered while developing BobaBot's software was storing table paths and increasing the robot's precision.

Storing Table Paths

To store table paths the program was originally going to use a struct called Node which stored a float x and y coordinate. However, issues arose when trying to pass this data structure into functions and create arrays of Nodes. The compiler would not recognize the correct data type when passing nodes into functions or reading arrays, and RobotC did not have support for dynamic initialization of struct objects.

To overcome this obstacle a 2D array of float values was used to represent the path to a table instead of a 1D array of Node objects. However, RobotC requires arrays to be passed into functions as pointers which further complicates the passing of 2D arrays. Theoretically, to pass a 2D array to a function in RobotC, one must pass a pointer to a pointer denoted as "**". When trying to access values from these pointers, normal array notation fails and even pointer notation becomes more complicated. This posed a problem when trying to use 2D arrays to store x,y coordinates for nodes in table paths.

This led to the final iteration of the navigation code which uses 1D arrays of strings to store paths to tables. Each string has two values (x and y coordinates) that are separated by a space. Helper functions were then used to split these strings and convert them into float values for the program to use.

Precision

Most of the troubleshooting and testing revolved around one main issue which was inconsistencies in the robot's physical accuracy and precision. Some test runs the robot would act near perfect, but then two runs later it would fail our criteria. Many questions arose when trying to fix these inconsistencies including:

- The root cause of the issues
- Whether it can be fixed with the given hardware
- Whether it is due to a factor of randomness within the code

To isolate the sole cause of the issue, it was realized that adjusting constants in the code to fix precision was most successful when keeping the battery life of the robot constant.

This led to the decision to run all tests at (near) 100% so that any precision inconsistencies stemming from the battery life changing would be removed.

Verification

The constraints of the design of the robot were kept in careful consideration throughout the construction and programming of the device. Because of this, the constraints (delivery size and security system) were met quite easily. The delivery size was constrained in the software since the longest table paths were controlled by the node lists. The node with the largest coordinate (which dictated the distance the robot would travel) was (75, 300). This meant that, at its furthest point, the robot would only have to travel 375 cm to and from the desired table, fitting comfortably in the constraint provided.

The constraints on the payment system were met quite readily, as the color sensor is already limited to 7 different colors to begin with. The limits with the color sensor were considered for in the software section, as there were only 6 accounts in the database, 1 tied to each color and 1 used as a nominal value. The card reader was further designed to limit the possible objects one can use to imitate the colored cards, reinforcing the security of the constrained system.

Project Plan

Project Plan and Contributors

The project was split into many different components, where various people worked on different things based on their expertise, availability, and task at hand.

Table 7: Initial project plan and contributors

Deliverables	Due Date	Contributors	Notes
Drive System	Wednesday, Nov 2	Yuming, Ethan, Karthigan	Motor powered with 2 wheels and bearings.
Cup Holder	Friday, Nov 4	Ethan	Tilting mechanism with motors, cup holder, and touch sensor.
Card Reader	Saturday, Nov 5	Joey, Ethan	3D printed with card insert and color sensor mounts.
Basic Functions	Sunday Nov 6	Joey	Creating various functions related to driving, including rotate, drive distance, wait button
Inner Frame	Sunday, Nov 6	Karthigan, Yuming	Ultrasonic sensor, support, wiring, and balance for other parts.

Deliverables	Due Date	Contributors	Notes
Interfacing Sensors/Motors	Tuesday, Nov 8	Joey	Creating functions to power each mechanical component of the bot.
Path Finding	Friday, Nov 11	Ethan, Karthigan	Coding specific paths to different tables, ultrasonic object detection.
Card Payment Database	Saturday, Nov 12	Yuming	Link EV3 with computer and web database to update different user balances.
Outer Frame	Sunday, Nov 20	Yuming	Friendly looking WALL E design with brick interface and straw holders.
Main() Code	Tuesday, Nov 22	Everyone	Integrating all the functions into one main “functional” code.
Functionality Testing	Wednesday, Nov 23	Karthigan, Ethan	Testing the functions of the code, in other words testing main().
Precision Testing	Thursday, Nov 24	Yuming, Joey	Changing constants that affect motion and control to yield highest precision and accuracy.

Revisions & Deviations from the Plan

The overall chronology of the project plan was not altered too drastically. However, some timelines were shifted, and revisions were made based on availability. Furthermore, some components of the proposed plan were finished far quicker than anticipated (cup holder, card reader, frames), while other components (drivetrain, path finding) had to be revisited due to the vast number of unforeseen errors that came with these deliverables.

The mechanical portion of the designed robot was overall finished much faster than it was thought to take. However, large components of it, specifically the drivetrain, had to be remodeled due to stability issues, which lead to this component being the last mechanical component completed, despite the proposed plan dictating that it should be completed first.

The software side of the robot had a similar experience, with most of it being finished alongside the construction of the physical robot itself. However, more functions and complexity with RobotC lead to some functions of the program to lag behind the proposed schedule; most apparently demonstrated with the path finding algorithm that had many bugs. Additionally, due to time constraints the cloud database and connection to an external device was abandoned.

The robot was completed roughly on schedule, with the testing phase adequately allowing for the robot to be polished and refined before the project. The plan was generally stuck to with few major revisions in terms of its chronological order.

Conclusions

In conclusion, the robot was able to complete the problem of delivering drinks to students at different locations and processing payments. The constraints and criteria of the project were met; the robot was able to stop after detecting moving objects, did not make any illegal moves over tables, and was precise within the margin of error. The drink used to test the robot was not spilled/punctured/dropped at any point and only delivered when the correct card payment was processed.

The mechanical components used in the design include: the drive which allowed the robot to navigate the room; the card reader which allowed the robot to recognize different accounts; and the drink holder which received and dispensed the drink.

The major software components used in the design include: the navigation system which calculated rotations and distances needed to travel to each table; the payment system which recognized and persisted payments from accounts, the UI which allowed the user to select size and table, and the motion code.

Overall, the team is satisfied with BobaBot's performance despite design constraints and technical difficulties.

Recommendations

Mechanical Recommendations

Different mechanical system: Based on testing results, the LEGO EV3 system has shown itself to have unreliable hardware. The main issue comes down to consistency with the changes in performance due to battery level, the motors not starting on time together, and compounding error when the robot has to drive further distances. One recommendation is to use a different operating system with more consistent and power motors and reliable sensors to improve performance over a larger area.

Another recommendation for a future design of this project would be a redesign of the drivetrain. Specifically, the wheels should be switched out for drive belts. The wheels that were used were unable to hold the relatively large weight of the robot above it, causing severe compression in the tires that led to instability and the rubber of the tires slipping off the rim of the wheel. Although this stability was fixed with the addition of ball bearings, they lead to a whole new set of problems that are required for constant cleaning of the cylindrical balls. Furthermore, the tires started slipping off the rims of the wheels used during the testing phase, which led to huge margins of error in precision and caused severe drifting of the robot. All these conflicts would have been avoided with the use of drive belts; thus, it would be beneficial if these were used instead.

Software Recommendations

In order to improve the codebase a library or different language would be used to add support for passing 2D arrays into functions, dynamic definition of structures, and connection to the internet or other devices. Structures would be used to represent nodes instead of space separated strings, reducing the need for the getX and getY helper functions. Additionally, a cloud database would be used to increase accessibility and security instead of a locally saved, unencrypted text file. With internet connection capabilities or the ability to connect to other devices an application could be developed to order the drink online eliminating the need for any human interaction between the client and server.

This would also eliminate failed deliveries due to insufficient funds on an account since the user could pay ahead of time on the application before the drink is delivered. This not only reduces waste but increases the security of the payment system in place of color-coded cards.

References

[1] A. Stanton, Director, *WALL-E.* [Film]. United States of America: PIXAR, 2008.

[2] "desmos," desmos, 1 12 2022. [Online]. Available: https://www.desmos.com/. [Accessed 21 11 2022].

Appendix A. Source Code

```
/*
RobotC EV3 Culminating Project
      "The Boba Bot"
       Version 1.0

Joey Maillette, Ethan Ahn
Yuming He, Karthigan Uthayan

Description
--------------------------------

Assumptions
--------------------------------
*/

// UI
int selectTable();
float selectDrink();
void EndMessage(bool paid);

// motion/control
void conditions();
bool tooClose();
void configureAllSensors();
void waitButton(TEV3Buttons button);
void motorsOn(int leftpower, int rightpower);
void motorsOff();
void driveDistCM(float dist, int angle);
int accelerate(int motorSpeed, int sliceTime, bool acc);
void gyroCorrection(int angle);
void rotateRobot(int motorSpeed, int angle);
void waitForCupIn();
void waitForCupOut();
void openDoor(bool cupIn);

// navigation
int driveToNode(string nextNode, string pos, int heading);
int driveToTable(string *table, int nodes);
void returnToStart(string *table, int nodes, int last_heading);
float getX(string node);
float getY(string node);

//tables
const string TABLE_1[3] = {"0 0", "0 300", "75 300"};
const string TABLE_2[3] = {"0 0", "0 300", "-75 300"};
const string TABLE_3[3] = {"0 0", "0 200", "75 200"};
const string TABLE_4[3] = {"0 0", "0 200", "-75 200"};
const string TABLE_5[3] = {"0 0", "0 100", "75 100"};
const string TABLE_6[3] = {"0 0", "0 100", "-75 100"};

// database
void initialiseDB();
void printDB();
void createLocal(float *localDB);
void postLocal(float *localDB);
bool readCard(float cost, float *balances);

// global constants
const int MOTOR_SPEED = 30;
```

```
60. const int ROTATE_SPEED = 15;
61. const int DOOR_SPEED = 20;
62. const int DOOR_ANGLE = 60;
63. const float CM_TO_DEG = 180 / (2.8 * PI);
64. const int TOO_CLOSE = 60;
65. const int ACCEL_FACTOR = 100;
66. const int MOTOR_LIMIT = 5;
67. const int CORRECTION_FACTOR = 1;
68. const int SLICE_TIME = 5;
69. const int NUM_ACCOUNTS = 6;
70. const int NUM_NODES = 3;
71. const int NUM_TABLES = 6;
72.
73. // global constants for readability
74. const int LEFT_MOTOR = motorA;
75. const int RIGHT_MOTOR = motorD;
76. const int DOOR_MOTOR = motorB;
77. const int GYRO = S4;
78. const int ULTRASONIC = S2;
79. const int COLOR = S3;
80. const int CUP_TOUCH = S1;
81. const TEV3Buttons KILL_SWITCH = buttonUp;
82.
83. // Main
84. task main()
85. {
86.    waitButton(buttonEnter);
87.    configureAllSensors();
88.
89.    printDB();
90.
91.    float balances[NUM_ACCOUNTS];
92.    createLocal(balances);
93.
94.    int table = selectTable();
95.    float cost = selectDrink();
96.    string currentTable[NUM_NODES];
97.    int last_heading;
98.
99.    openDoor(true);
100.
101.   if (table==1)
102.   {
103.     last_heading = driveToTable(TABLE_1, NUM_NODES);
104.   }
105.   else if (table==2)
106.   {
107.     last_heading = driveToTable(TABLE_2, NUM_NODES);
108.   }
109.   else if (table==3)
110.   {
111.     last_heading = driveToTable(TABLE_3, NUM_NODES);
112.   }
113.   else if (table==4)
114.   {
115.     last_heading = driveToTable(TABLE_4, NUM_NODES);
116.   }
117.   else if (table==5)
118.   {
119.     last_heading = driveToTable(TABLE_5, NUM_NODES);
120.   }
```

```
121.   else
122.   {
123.     last_heading = driveToTable(TABLE_6, NUM_NODES);
124.   }
125.
126.   bool paid = readCard(cost, balances);
127.
128.   if(paid)
129.   {
130.     openDoor(false);
131.   }
132.
133.   if (table==1)
134.   {
135.     returnToStart(TABLE_1, NUM_NODES, last_heading);
136.   }
137.   else if (table==2)
138.   {
139.     returnToStart(TABLE_2, NUM_NODES, last_heading);
140.   }
141.   else if (table==3)
142.   {
143.     returnToStart(TABLE_3, NUM_NODES, last_heading);
144.   }
145.   else if (table==4)
146.   {
147.     returnToStart(TABLE_4, NUM_NODES, last_heading);
148.   }
149.   else if (table==5)
150.   {
151.     returnToStart(TABLE_5, NUM_NODES, last_heading);
152.   }
153.   else
154.   {
155.     returnToStart(TABLE_6, NUM_NODES, last_heading);
156.   }
157.
158.   EndMessage(paid);
159.   if (!paid)
160.   {
161.     openDoor(false);
162.   }
163.
164.   postLocal(balances);
165.   printDB();
166. }
167.
168. // function configures all the sensors
169. void configureAllSensors()
170. {
171.   // reset all motor encoders
172.   nMotorEncoder[LEFT_MOTOR] = nMotorEncoder[RIGHT_MOTOR] =
173.   nMotorEncoder[DOOR_MOTOR] = 0;
174.
175.   // sensor ports
176.   SensorType[GYRO] = sensorEV3_Gyro;
177.   SensorType[ULTRASONIC] = sensorEV3_Ultrasonic;
178.   SensorType[COLOR] = sensorEV3_Color;
179.   SensorType[CUP_TOUCH] = sensorEV3_Touch;
180.   wait1Msec(50);
181.
```

```
182.   // color sensor initialization
183.   SensorMode[COLOR] = modeEV3Color_Color;
184.   wait1Msec(50);
185.
186.   // gyro initialization
187.   SensorMode[GYRO] = modeEV3Gyro_Calibration;
188.   wait1Msec(100);
189.   SensorMode[GYRO] = modeEV3Gyro_RateAndAngle;
190.   wait1Msec(50);
191.   resetGyro(GYRO);
192. }
193.
194. // function checks while-loop stopping conditions
195. // ie. kill switch or object detection
196. void conditions()
197. {
198.   // checks if the kill switch was pressed, end the program
199.   if (getButtonPress(KILL_SWITCH))
200.   {
201.     stopTask(main);
202.   }
203. }
204.
205. bool tooClose()
206. {
207.   // if an object stopped the while-loop
208.   if (SensorValue(ULTRASONIC) <= TOO_CLOSE)
209.   {
210.     playSound(soundException);
211.     return true;
212.   }
213.   // if the while-loop condition stopped the while-loop
214.   else
215.   {
216.     return false;
217.   }
218. }
219.
220. // function to wait for a button to be pressed and released
221. void waitButton(TEV3Buttons button)
222. {
223.   // wait for it to be pressed
224.   while (!getButtonPress(button) && !getButtonPress(KILL_SWITCH))
225.   {}
226.     conditions();
227.   // wait for it to be released
228.   while (getButtonPress(button) && !getButtonPress(KILL_SWITCH))
229.   {}
230.     conditions();
231. }
232.
233. // function that turns motors on at specific left and right speed
234. void motorsOn(int leftpower, int rightpower)
235. {
236.   motor[LEFT_MOTOR] = leftpower;
237.   motor[RIGHT_MOTOR] = rightpower;
238. }
239.
240. // function that powers both motors off
241. void motorsOff()
242. {
```

```
    motor[LEFT_MOTOR] = motor[RIGHT_MOTOR] = 0;
}

// function that drives to a specific distance in cm
void driveDistCM(float dist, int angle)
{
    // for (0,0) nodes
    if (dist == 0)
    {
        return;
    }

  // reset the gyro before the distance is travelled to account for minor
  // angular drift
  nMotorEncoder[LEFT_MOTOR] = 0;
  resetGyro(GYRO);
    int start_angle = getGyroDegrees(GYRO)+angle;

  // stopping condition set to false
  bool recall_fnc = false;
  int decelerateDist = 0;

  // accelerate() accelerates the robot and returns the distance in motor
  // encoder ticks that it took to accelerate so we know when to start
  // decelerating at the same rate
  decelerateDist = accelerate(MOTOR_SPEED, SLICE_TIME, true);

  // while conditions:
  // distance travelled so far < desired distance travelled there is
  // no object
  // kill switch isn't pressed
  time1[T1] = 0;
  while (nMotorEncoder[LEFT_MOTOR] < ((dist)*CM_TO_DEG - decelerateDist) &&
         SensorValue(ULTRASONIC) > TOO_CLOSE && !getButtonPress(KILL_SWITCH))
  {
              gyroCorrection(-start_angle);
  }

  // stopping condition is set based on conditions()
  recall_fnc = tooClose();
  conditions();

  // to make sure motor values are same before deceleration phase
    motorsOn(MOTOR_SPEED, MOTOR_SPEED);

  // if the reason it stopped was because there was an object detected
  if (recall_fnc)
  {
    // wait for the object to move
    while (SensorValue(ULTRASONIC) <= TOO_CLOSE && !getButtonPress(KILL_SWITCH))
    {}
    conditions();

    // recursively calls driveDistCM by the amount of distance the bot still
    // needs to travel
    driveDistCM(dist - nMotorEncoder[LEFT_MOTOR] / CM_TO_DEG,
                getGyroDegrees(GYRO));
  }
}

/*
```

```
304. function that controls the speed of the bot while in acceleration
305. or deceleration phases. (uses trigonometric acceleration)
306.
307. sliceTime is the time between speed updates
308. acc is true for accelerating and false for decelerating
309. */
310. int accelerate(int motorSpeed, int sliceTime, bool acc)
311. {
312.   bool recall_fnc = false;
313.   float current_power = 0;
314.   float fraction = 0;
315.
316.   recall_fnc = tooClose();
317.   if (recall_fnc)
318.   {
319.     while (SensorValue(ULTRASONIC) <= TOO_CLOSE && !getButtonPress(KILL_SWITCH))
320.     {}
321.     conditions();
322.   }
323.   // loops over a constant number of steps (set to 100 right now)
324.   for (int i = 0; i < ACCEL_FACTOR; i++)
325.   {
326.     // calculates the fraction of the motorSpeed the robot needs to be on during
327.     // this "step"
328.     fraction = (-1 * (cos(i * PI / ACCEL_FACTOR) + 1) / 2) + 1;
329.
330.     // if it is accelerating
331.     if (acc)
332.     {
333.       current_power = motorSpeed * fraction;
334.       motorsOn(current_power,
335.               current_power+fraction); // +1 to compensate for motor imbalance
336.     }
337.
338.     // if its decelerating
339.     else
340.     {
341.       current_power = motorSpeed - (motorSpeed * fraction);
342.       motorsOn(current_power,
343.               current_power+(1-fraction));
344.     }
345.
346.
347.     time1[T2] = 0;
348.
349.     // times each for-loop and checks if the kill switch is pressed
350.     while (time1[T2] < sliceTime && !getButtonPress(KILL_SWITCH))
351.     {}
352.     conditions();
353.     displayString(5, "%d", getGyroDegrees(GYRO));
354.   }
355.
356.   // returns the distance travelled over acceleration phase in motor encoder
357.   // ticks
358.   return (nMotorEncoder[LEFT_MOTOR]);
359. }
360.
361. void gyroCorrection(int angle)
362. {
363.     if (getGyroDegrees(GYRO) == angle)
364.     {
```

```
365.         motorsOn(MOTOR_SPEED, MOTOR_SPEED);
366.     }
367.
368.     else if (getGyroDegrees(GYRO) < angle)
369.     {
370.         motorsOn(MOTOR_SPEED, MOTOR_SPEED-CORRECTION_FACTOR*abs(angle-
371.                 getGyroDegrees(GYRO)));
372.     }
373.
374.     else
375.     {
376.         motorsOn(MOTOR_SPEED-CORRECTION_FACTOR*abs(angle-getGyroDegrees(GYRO)),
377.                 MOTOR_SPEED);
378.     }
379. }
380.
381. // function smoothly rotates the robot over an angle from -180 to 180 (90 being
382. // right)
383. void rotateRobot(int motorSpeed, int angle)
384. {
385.     float current_power = 0;
386.   float fraction = 0;
387.   // loops over a constant number of steps (set to 100 right now)
388.   for (int i = 0; i < ACCEL_FACTOR; i++)
389.   {
390.     // calculates the fraction of the motorSpeed the robot needs to be on during
391.     // this "step"
392.     fraction = (-1 * (cos(i * PI / ACCEL_FACTOR) + 1) / 2) + 1;
393.     current_power = motorSpeed * fraction;
394.
395.     // if turning left, left motor value negative, right motor value positive
396.     if (angle < 0)
397.     {
398.       // note: motor limit is used to ensure no speed below 1 is read as 0
399.       // (integer)
400.       motorsOn(current_power - motorSpeed - MOTOR_LIMIT,
401.                motorSpeed - current_power + MOTOR_LIMIT);
402.     }
403.     // if turning right, right motor value negative, left motor value positive
404.     else
405.     {
406.       motorsOn(motorSpeed - current_power + MOTOR_LIMIT,
407.                current_power - motorSpeed - MOTOR_LIMIT);
408.     }
409.
410.     // each loop it turns the desired angle/the number of loops
411.     while (abs(getGyroDegrees(GYRO)) < (abs((float)angle / ACCEL_FACTOR) * (i + 1)) &&
412.            !getButtonPress(KILL_SWITCH))
413.     {}
414.     conditions();
415.   }
416.   motorsOff();
417. }
418.
419. // function that waits for the cup to be put in
420. void waitForCupIn()
421. {
422.   while (!SensorValue[CUP_TOUCH] && !getButtonPress(KILL_SWITCH))
423.   {}
424.   conditions();
425.
```

```
426.   time1[T1] = 0;
427.
428.   while (time1[T1] < 2000 && !getButtonPress(KILL_SWITCH))
429.   {}
430.   conditions();
431. }
432. // function that waits for the cup to be taken out
433. void waitForCupOut() {
434.   while (SensorValue[CUP_TOUCH] && !getButtonPress(KILL_SWITCH))
435.   {}
436.   conditions();
437.
438.   time1[T1] = 0;
439.   while (time1[T1] < 1000 && !getButtonPress(KILL_SWITCH))
440.   {}
441.   conditions();
442. }
443.
444. // function that controls the door movement based on if the cup is being
445. // inserted or removed
446. void openDoor(bool cupIn)
447. {
448.   nMotorEncoder[DOOR_MOTOR] = 0;
449.   motor[DOOR_MOTOR] = DOOR_SPEED;
450.
451.   // opens the door
452.   while (nMotorEncoder[DOOR_MOTOR] < DOOR_ANGLE && !getButtonPress(KILL_SWITCH))
453.   {}
454.   conditions();
455.   motor[DOOR_MOTOR] = 0;
456.
457.   // if a cup is being inserted, wait for it to be inserted
458.   if (cupIn) {
459.     waitForCupIn();
460.   }
461.   // if a cup is being removed, wait for it to be removed
462.   else
463.   {
464.     waitForCupOut();
465.   }
466.
467.   motor[DOOR_MOTOR] = -DOOR_SPEED;
468.
469.   // close the door
470.   while (nMotorEncoder[DOOR_MOTOR] > 0 && !getButtonPress(KILL_SWITCH))
471.   {}
472.   conditions();
473.   motor[DOOR_MOTOR] = 0;
474. }
475.
476. // helper function to initialise the database in the EV3 brick's memory
477. void initialiseDB()
478. {
479.   const int MAX_SIZE = 7;
480.   float iBalances[MAX_SIZE] = {100.0, 97.5, 24.5, 37.6, 56.9, 45.6, 85.3};
481.   long fout = fileOpenWrite("db.txt");
482.   for (int i = 0; i < MAX_SIZE; i++)
483.   {
484.     fileWriteFloat(fout, iBalances[i]);
485.   }
486.
```

```
487.   fileClose(fout);
488. }
489.
490. // helper function to see what values are in the txt file
491. void printDB()
492. {
493.   const int MAX_SIZE = NUM_ACCOUNTS;
494.   long fin = fileOpenRead("db.txt");
495.   float balance = 0;
496.   for (int i = 0; i < MAX_SIZE; i++)
497.   {
498.     fileReadFloat(fin, &balance);
499.     displayString(i, "Account#: %d, Balance: %.2f", i, balance);
500.   }
501.
502.   waitButton(buttonEnter);
503.   fileClose(fin);
504.   eraseDisplay();
505. }
506.
507. // create local version of database by reading from file and writing to the
508. // array
509. void createLocal(float *localDB)
510. {
511.   const int SIZE = NUM_ACCOUNTS;
512.   long fin = fileOpenRead("db.txt");
513.   float balance = 0;
514.   for (int i = 0; i < SIZE; i++)
515.   {
516.     fileReadFloat(fin, &balance);
517.     localDB[i] = balance;
518.   }
519.
520.   fileClose(fin);
521. }
522.
523. // writes to the file from the local array
524. void postLocal(float *localDB)
525. {
526.   const int SIZE = NUM_ACCOUNTS;
527.   long fin = fileOpenWrite("db.txt");
528.   for (int i = 0; i < SIZE; i++)
529.   {
530.     fileWriteFloat(fin, localDB[i]);
531.   }
532.
533.   fileClose(fin);
534. }
535.
536. // reads card and processes payment
537. bool readCard(float cost, float *balances)
538. {
539.   displayString(4, "Price: %.2f", cost);
540.   displayBigTextLine(7, "INSERT CARD");
541.   while (SensorValue[COLOR] == (int)colorWhite && !getButtonPress(KILL_SWITCH))
542.   {}
543.   conditions();
544.
545.   eraseDisplay();
546.   time1[T1] = 0;
547.   displayBigTextLine(5, "PROCESSING...");
```

```
548.   while (time1[T1] < 1000 && !getButtonPress(KILL_SWITCH))
549.   {}
550.     conditions();
551.
552.   // read color value and check if account has sufficient funds to pay for drink
553.   int accountNum = SensorValue[COLOR] - 1;
554.   if (SensorValue[COLOR] == 7)
555.   {
556.     accountNum = 5;
557.   }
558.   bool payed = true;
559.
560.   eraseDisplay();
561.   if (balances[accountNum] - cost >= 0)
562.   {
563.     // charge account based on cost
564.     balances[accountNum] -= cost;
565.
566.     displayString(2, "Account #%d:", accountNum);
567.     displayString(4, "New Balance: %.2f", balances[accountNum]);
568.     playSound(soundUpwardTones);
569.   }
570.   else
571.   {
572.     payed = false;
573.     displayBigTextLine(2, "No Funds");
574.     playSound(soundDownwardTones);
575.   }
576.
577.   displayBigTextLine(6, "REMOVE CARD");
578.   while (SensorValue[COLOR] != (int)colorWhite && !getButtonPress(KILL_SWITCH))
579.   {}
580.   conditions();
581.   eraseDisplay();
582.
583.   return payed;
584. }
585.
586. int getX(string node)
587. {
588.     string new_node = node;
589.     int split1 = stringFind(new_node, " ");
590.     stringDelete(new_node, split1, strlen(new_node) - split1);
591.     return atoi(new_node);
592. }
593.
594. int getY(string node)
595. {
596.     string new_node = node;
597.     int split1 = stringFind(new_node, " ");
598.     stringDelete(new_node, 0, split1);
599.     return atoi(new_node);
600. }
601.
602. // drives to the a given node from a position and current heading
603. int driveToNode(string nextNode, string pos, int heading)
604. {
605.     // calculate drive distances and angle change
606.     int delta_x = getX(nextNode) - getX(pos);
607.     int delta_y = getY(nextNode) - getY(pos);
608.     int delta_angle = 0;
```

```
609.     int next_heading = 0;
610.     if (heading >= 0)
611.     {
612.         delta_angle = (180/PI)*(atan2(delta_x, delta_y)) - heading;
613.         next_heading = heading + delta_angle;
614.     } else {
615.         delta_angle = (180/PI)*(atan2(delta_x, abs(delta_y))) - heading;
616.         delta_angle *= -1;
617.         next_heading = heading + (-delta_angle);
618.     }
619.
620.     if (delta_angle != 0)
621.     {
622.         rotateRobot(ROTATE_SPEED, delta_angle);
623.     }
624.     driveDistCM(abs(delta_x + delta_y), 0);
625.
626.     return next_heading;
627. }
628.
629. // drive to a given table
630. int driveToTable(string *table, int nodes)
631. {
632.     // initialise position and heading
633.     string pos = "0 0";
634.     int heading = 0;
635.     string nextNode;
636.
637.     // navigate to each node in the path
638.     for (int i = 1; i < nodes; i++)
639.     {
640.         nextNode = *(table + i);
641.         heading = driveToNode(nextNode, pos, heading);
642.         pos = *(table + i);
643.     }
644.
645.     return heading;
646. }
647.
648. // drives to home from a given table
649. void returnToStart(string *table, int nodes, int last_heading)
650. {
651.     // initialises the start position as the last node in the table array
652.     int last_element = nodes - 1;
653.     string pos = *(table + last_element);
654.
655.     int heading = last_heading;
656.     string nextNode;
657.
658.     // navigates through the paths list backwards to return to home
659.     for (int i = last_element - 1; i >= 0; i--)
660.     {
661.         nextNode = *(table + i);
662.         heading = driveToNode(nextNode, pos, heading);
663.         pos = *(table + i);
664.     }
665.
666.     rotateRobot(ROTATE_SPEED, 180);
667. }
668.
669. int selectTable()
```

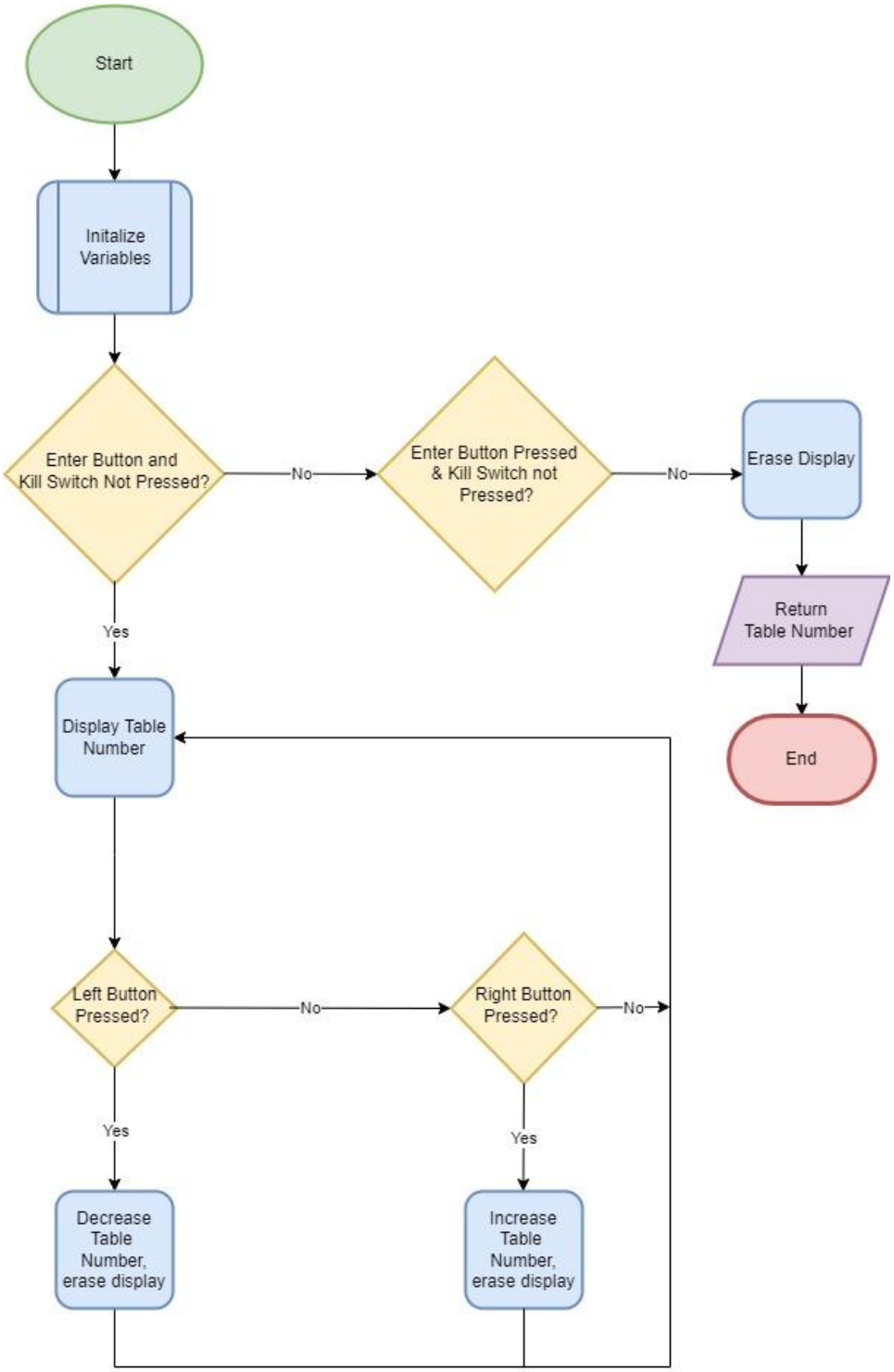
```
{
  // counter for table number (starts at 1)
  int table_num = 1;
  int MAX_TABLE = NUM_TABLES; // max amount of tables

  // displays string while the enter button is not pressed
  while (!getButtonPress(buttonEnter) && !getButtonPress(KILL_SWITCH))
  {
    // decreases table number
    if (getButtonPress(buttonLeft)) {
      // waits for user to let go of button and lowers table num
      while (getButtonPress(buttonLeft) && !getButtonPress(KILL_SWITCH))
      {}
      table_num--;
      eraseDisplay(); // erases previous num on display

      // goes to highest if they go less than 1
      // so table numbers are in range 1-6
      if (table_num < 1)
      {
        table_num = MAX_TABLE;
      }
    }

    // increases table number
    if (getButtonPress(buttonRight))
    {
      // waits for user to let go of button and increases num
      while (getButtonPress(buttonRight) &&  !getButtonPress(KILL_SWITCH))
      {}
      table_num++;
      eraseDisplay();

      // loops back to 1 if they go over the max amount of tables
      if (table_num > MAX_TABLE)
      {
        table_num = 1;
      }
    }

    // displays the the desired table number on Ev3 screen
    displayBigTextLine(3, "Table number %d", table_num);
  }

  // breaks loop if enter button is pressed
  // does nothing until released
  while (getButtonPress(buttonEnter) && !getButtonPress(KILL_SWITCH))
  {}

  eraseDisplay();

  // returns table num for payment
  return table_num;
}

/*allows user to select drink size
 returns float based on associated
  price of drink size */

float selectDrink()
{
```

```
731.   // assumes these are the only items in the menu
732.   string drinkMenu[] = {"small", "medium", "large"};
733.   float prices[] = {5.00, 5.50, 6.50}; // corresponding prices
734.   int menu_num = 0;                    // lowest index of array
735.   const int MAX_ITEM = 2;              // amount of menu items - 1
736.
737.   while (!getButtonPress(buttonEnter) && !getButtonPress(KILL_SWITCH))
738.   {
739.     // decrements menu_num when right button pressed
740.     if (getButtonPress(buttonLeft))
741.     {
742.       while (getButtonPress(buttonLeft) && !getButtonPress(KILL_SWITCH))
743.       {}
744.       menu_num--;
745.       eraseDisplay();
746.       // if you go over the menu options
747.       if (menu_num < 0) {
748.         menu_num = MAX_ITEM; // goes to the last option
749.       }
750.     }
751.
752.     // increments menu_num when right button pressed
753.     if (getButtonPress(buttonRight))
754.     {
755.       while (getButtonPress(buttonRight) && !getButtonPress(KILL_SWITCH))
756.       {}
757.       menu_num++;
758.       if (menu_num > MAX_ITEM) // if you go over the menu options
759.       {
760.         menu_num = 0; // goes back to the first option
761.       }
762.       eraseDisplay();
763.     }
764.
765.     // displays the menu item on the screen
766.     displayBigTextLine(7, "Size: %s", drinkMenu[menu_num]);
767.     displayBigTextLine(10, "$%.2f", prices[menu_num]);
768.   }
769.
770.   while (getButtonPress(buttonEnter) &&!getButtonPress(KILL_SWITCH))
771.   {}
772.   eraseDisplay();
773.
774.   // returns the price of menu item
775.   return prices[menu_num];
776. }
777.
778. void EndMessage(bool paid) //saved when paid or not
779. {
780.     if(paid == 0) //if not paid
781.     {
782.         displayString(5, "Not enouugh money");
783.     }
784.     else //paid
785.     {
786.         displayString(5, "Drink has been delivered");
787.     }
788.   waitButton(buttonEnter);
789.   eraseDisplay();
790. }
```

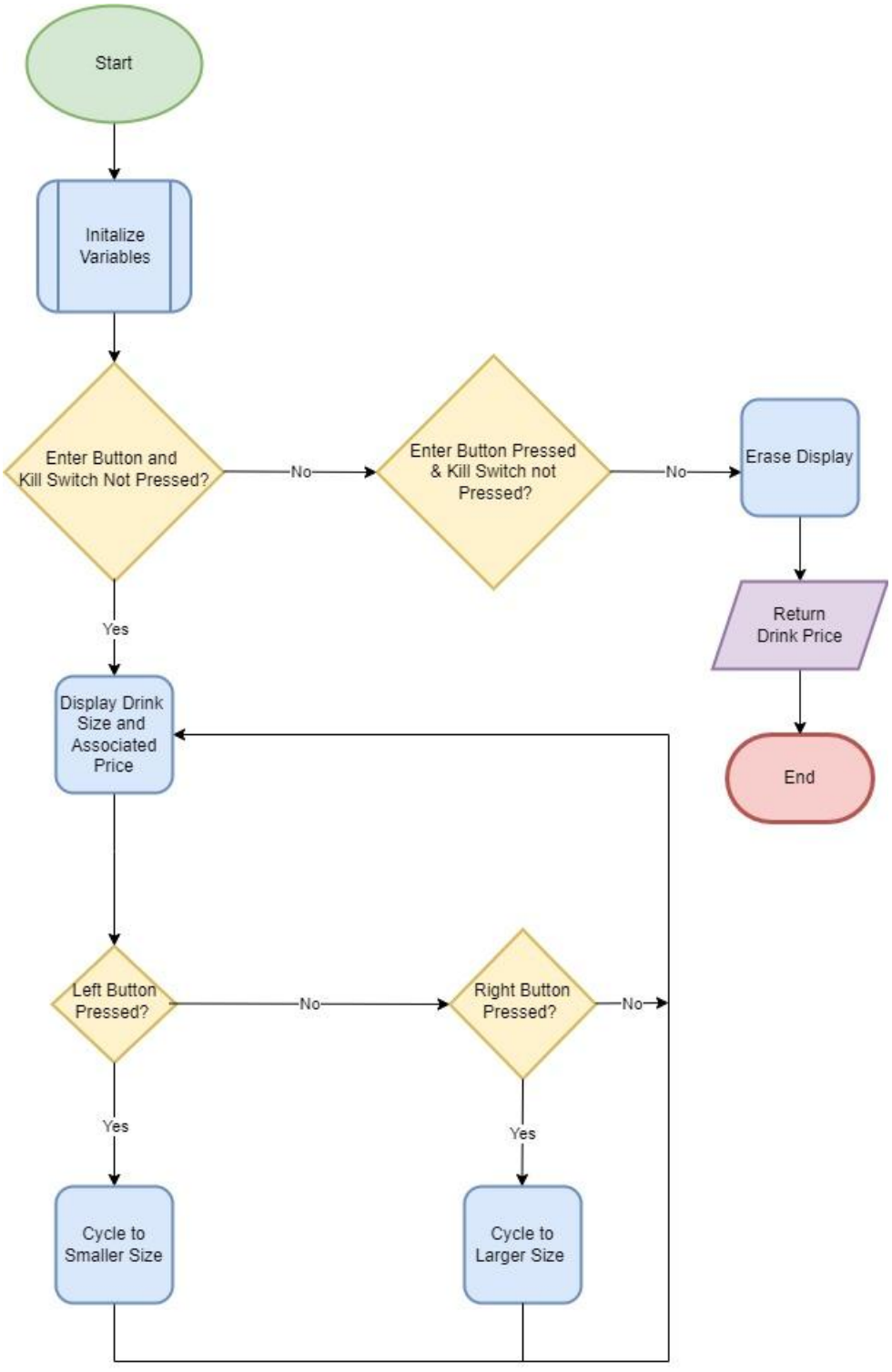
Appendix B. Flowcharts

B.1. UI Flowcharts

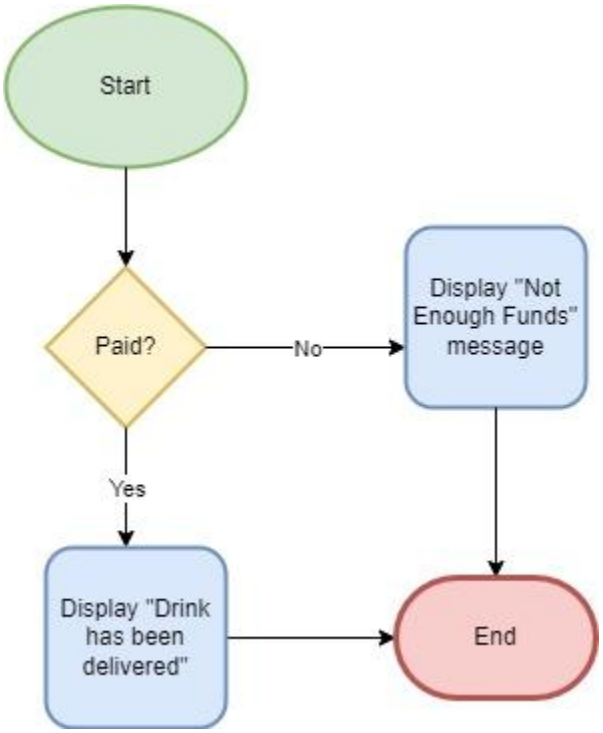
B.1.1 selectTable()



B1.2 selectDrink()

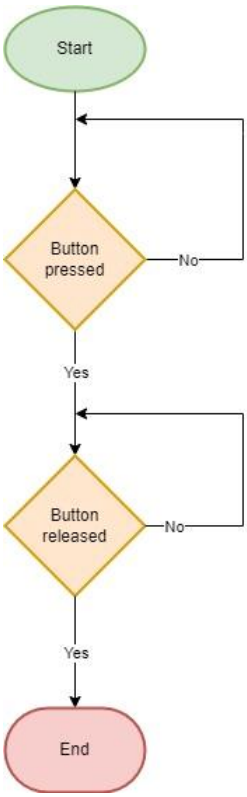


B.1.3 endMessage(bool paid)

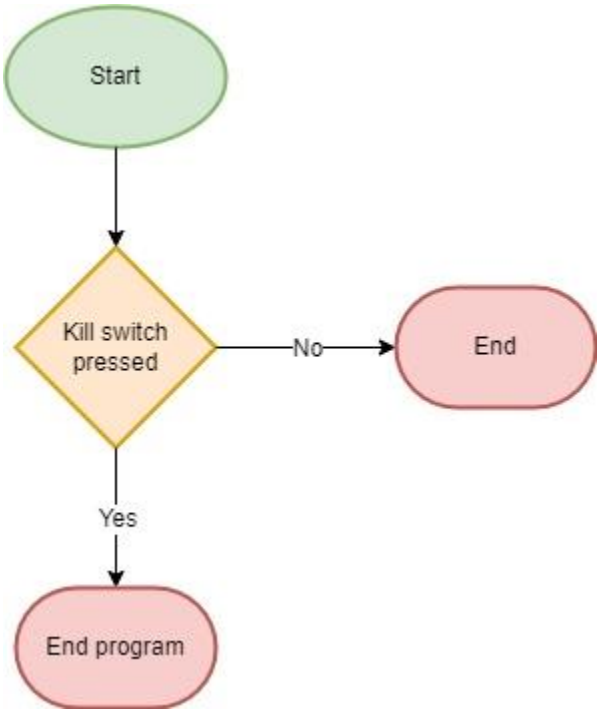


B.2 Motion/Control Flowcharts

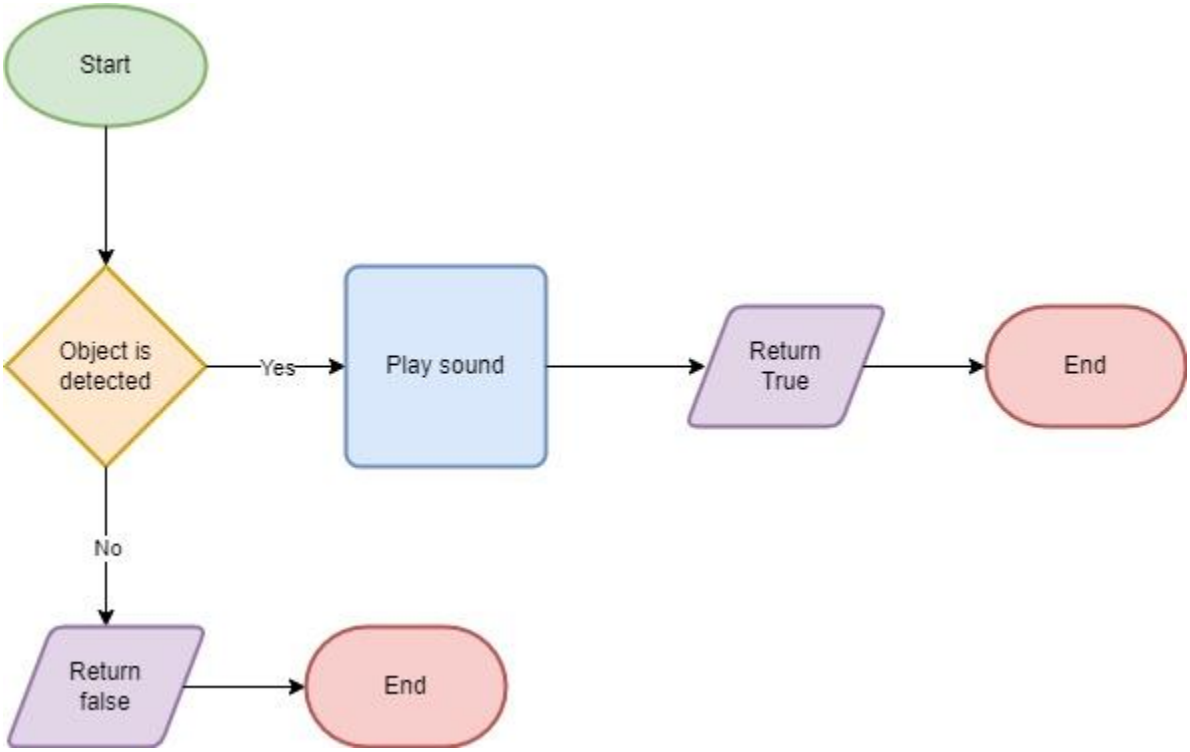
B.2.1 waitButton(TEV3Buttons button)



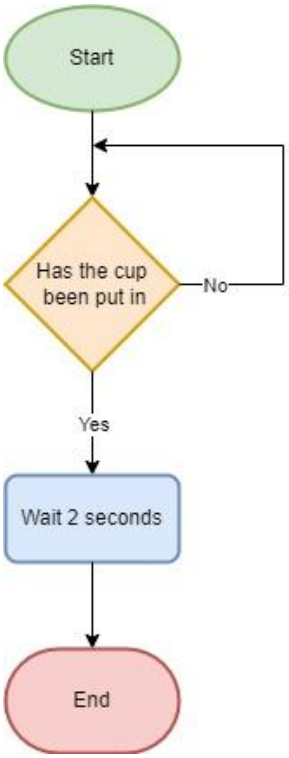
B.2.2 conditions()



B.2.3 tooClose()

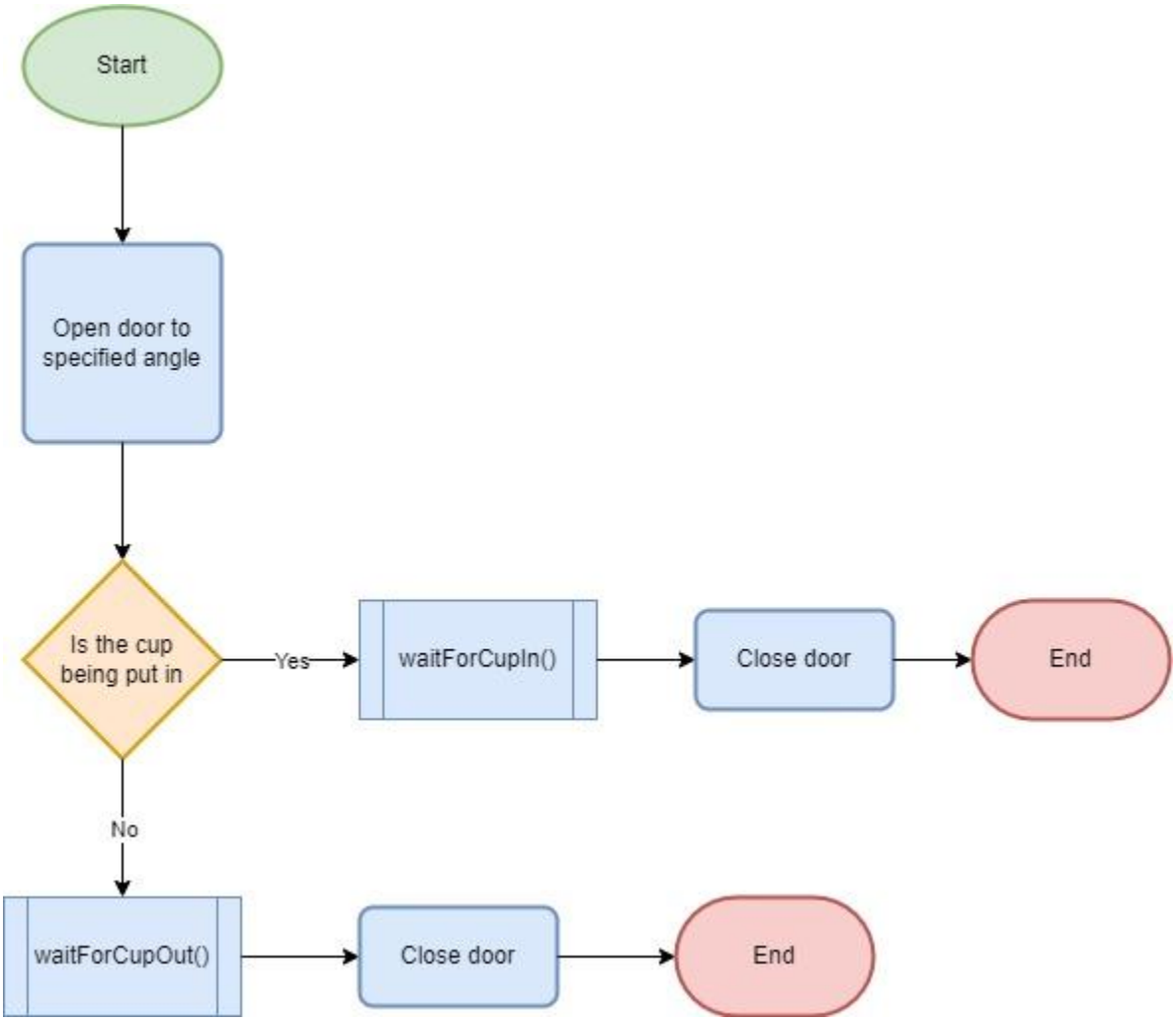


B.2.4 waitForCupIn()

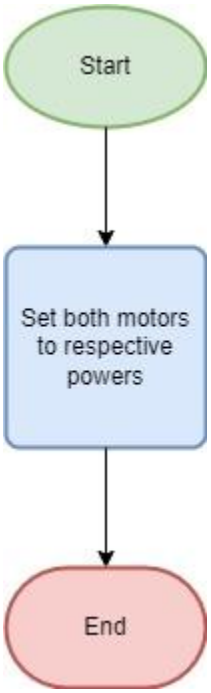


B.2.5 waitForCupOut()

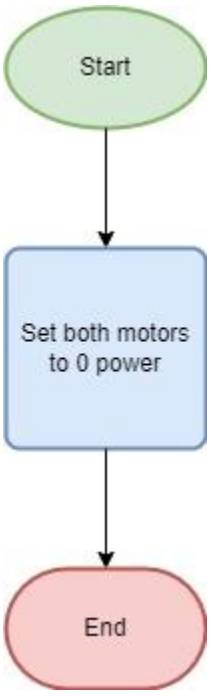
B.2.6 openDoor(bool cupIn)



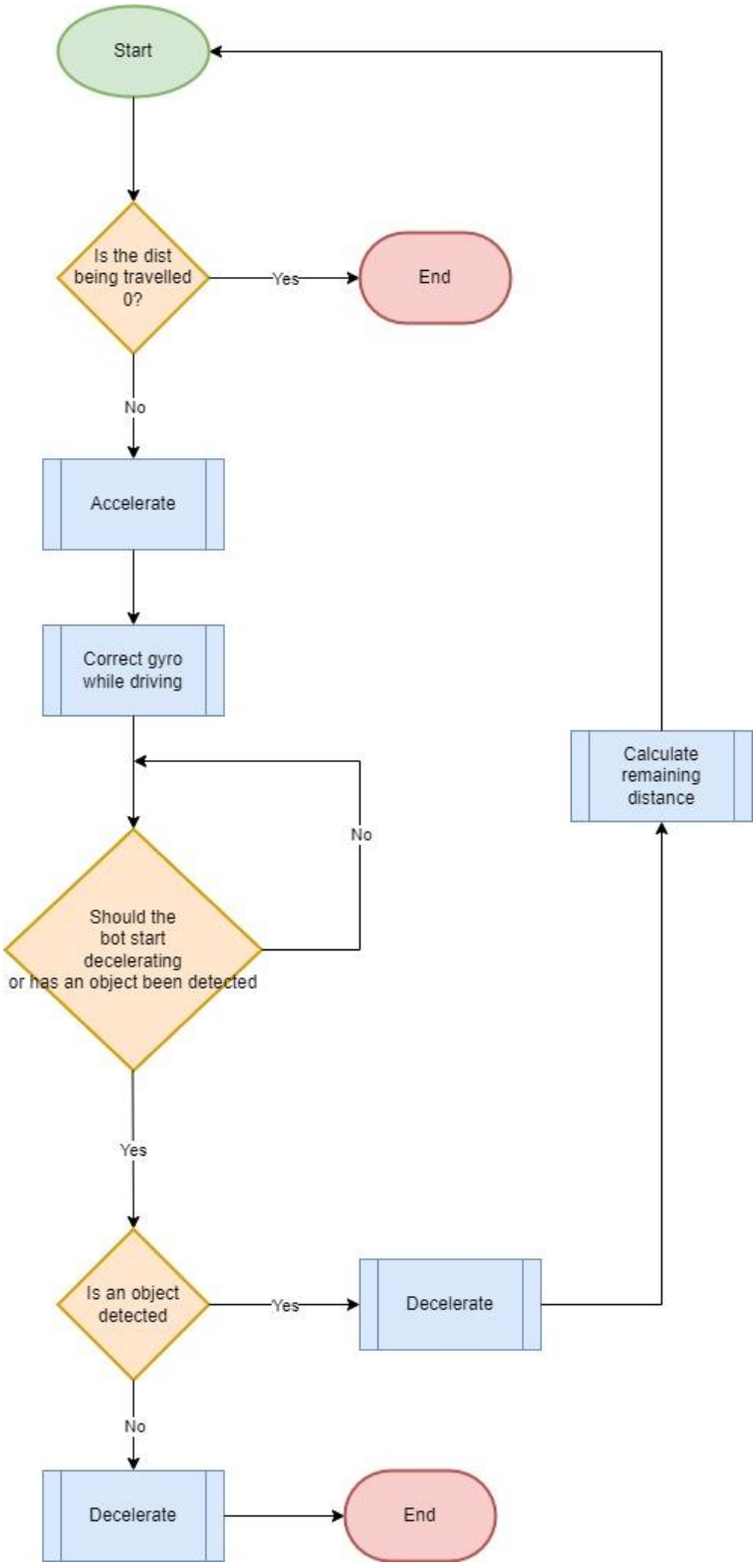
B.2.7 motorsOn(int leftpower, int rightpower)



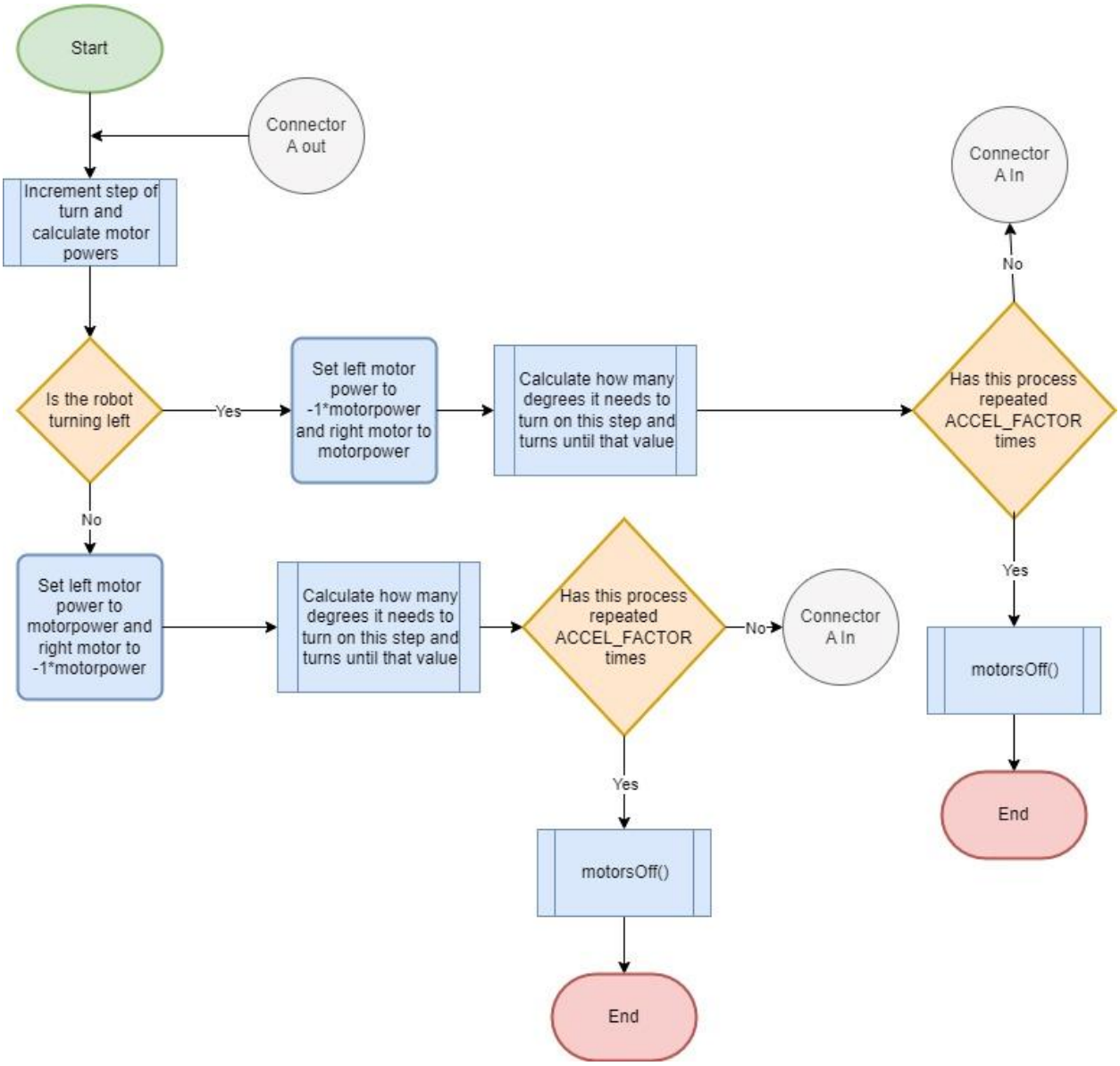
B.2.8 motorsOff()



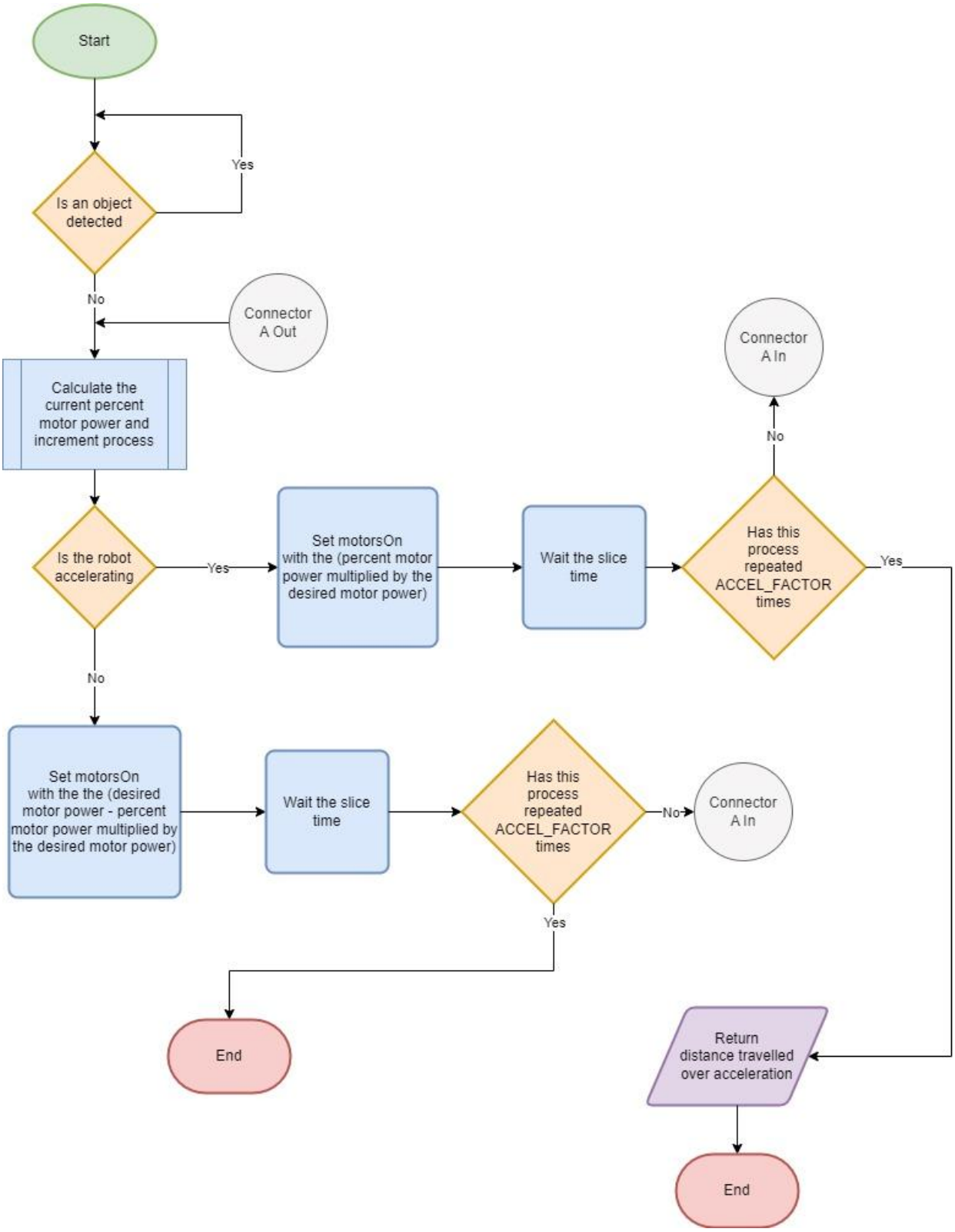
B.2.9 driveDistCM(float dist, int angle)



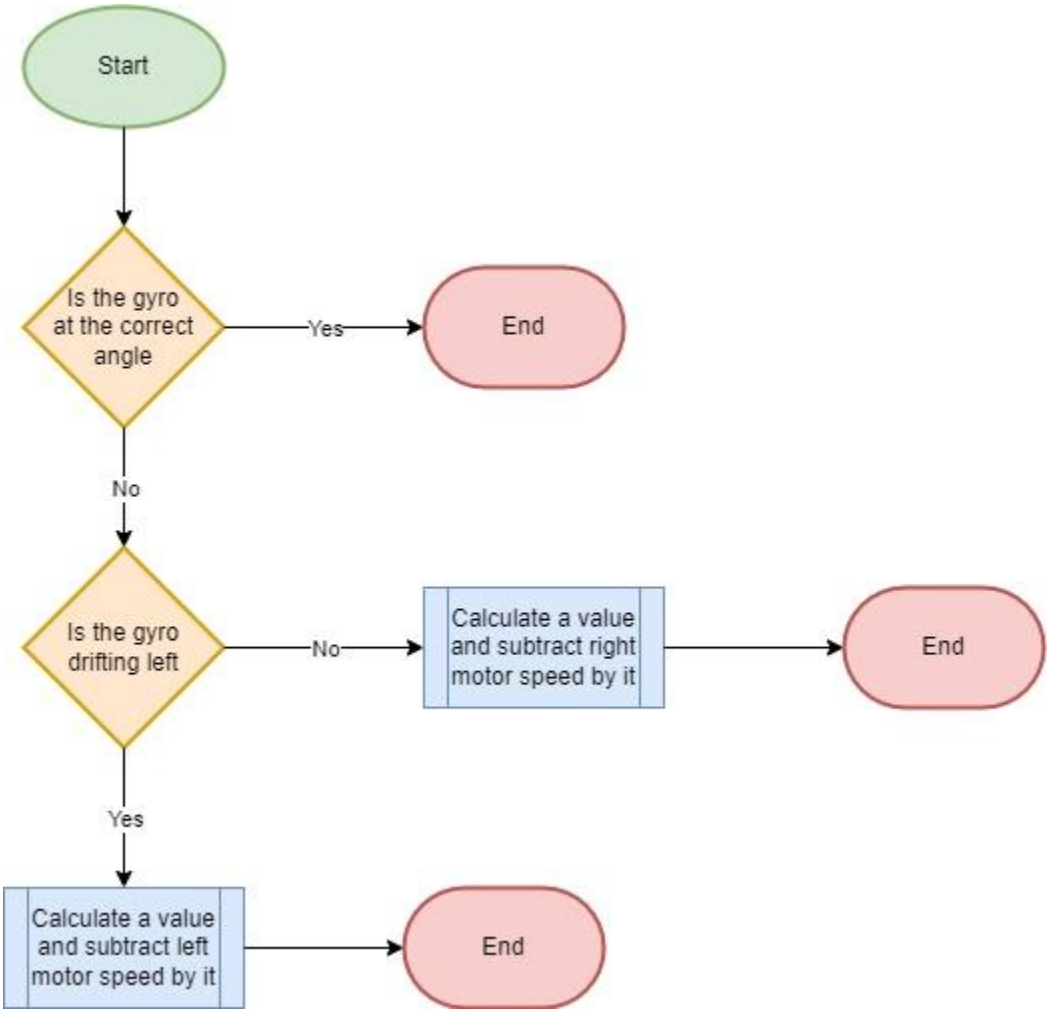
B.2.10 rotateRobot(int motorSpeed, int angle)



B.2.11 accelerate(int motorSpeed, int sliceTime, bool acc)

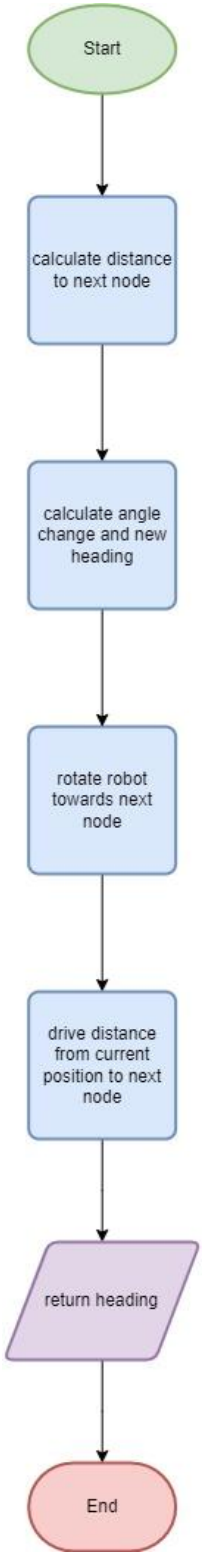


B.2.12 gyroCorrection(int angle)

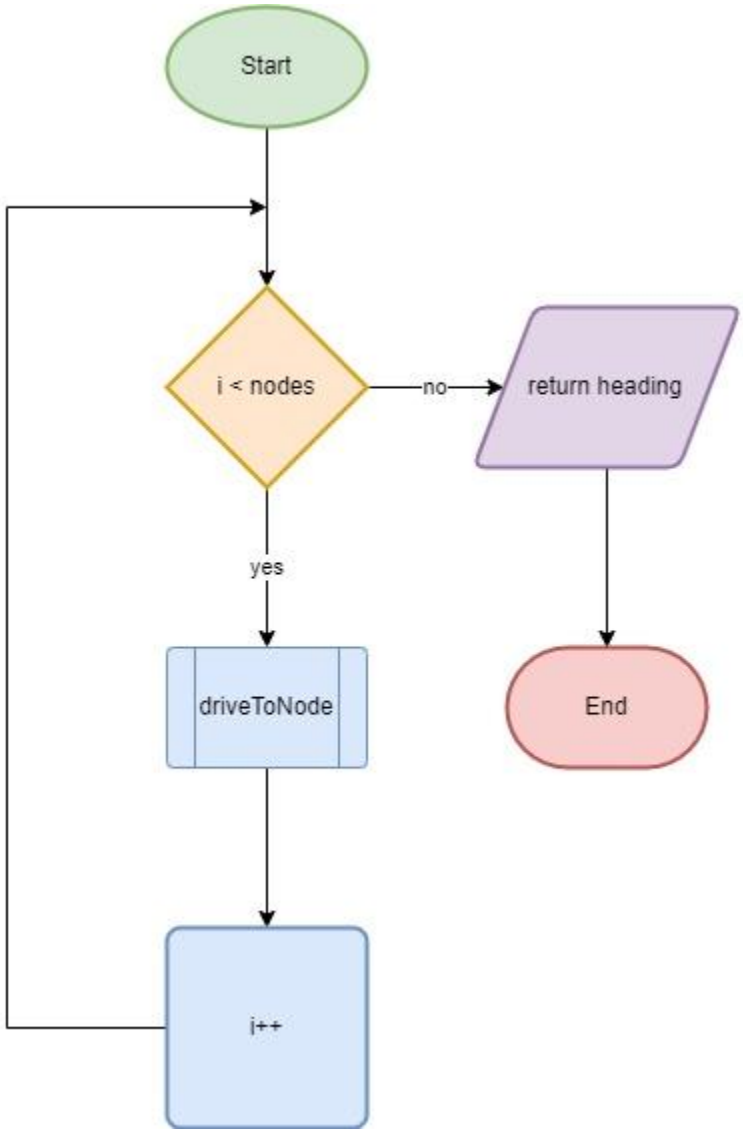


B.3 Navigation Flowcharts

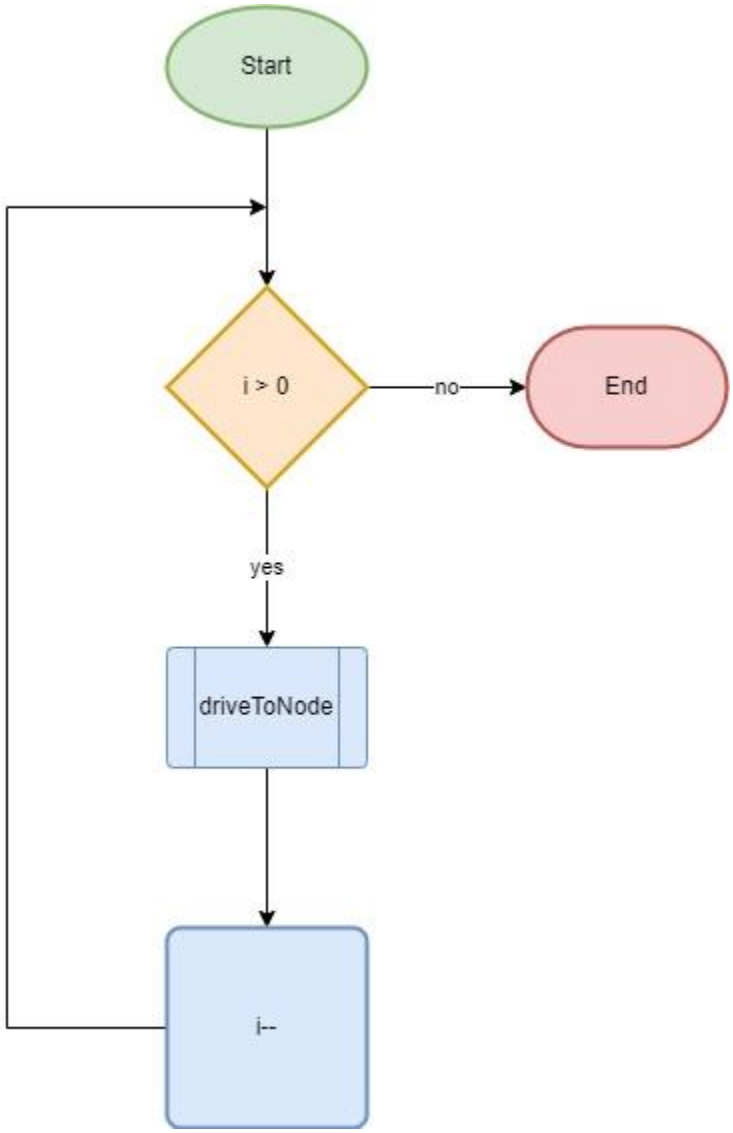
B.3.1 driveToNode(string nextNode, string pos, int heading)



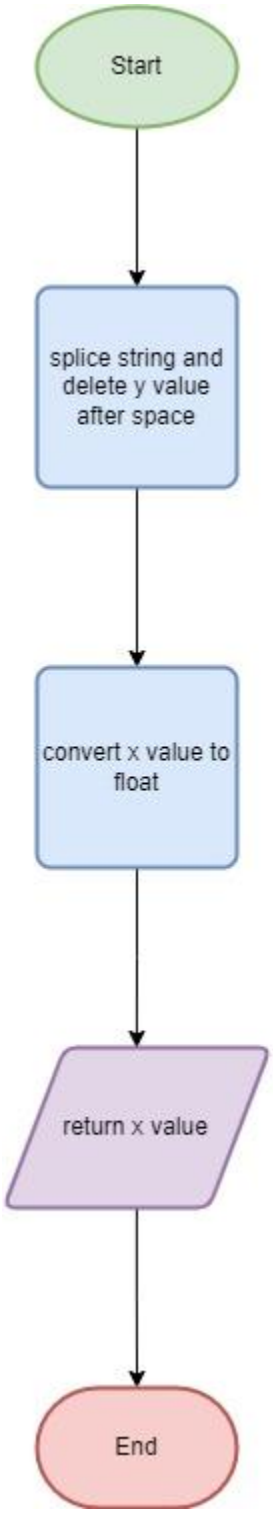
B.3.2 driveToTable(string *table, int nodes)



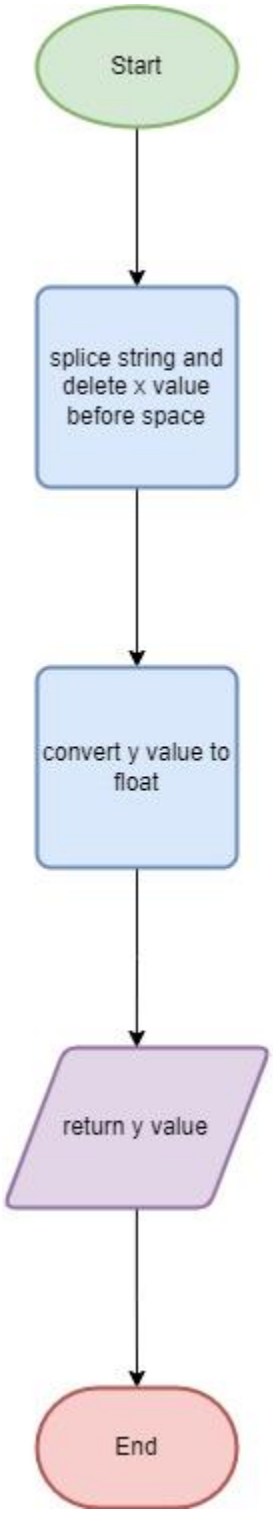
B.3.3 returnToStart(string *table, int nodes, int last_heading)



B.3.4 getX(string node)

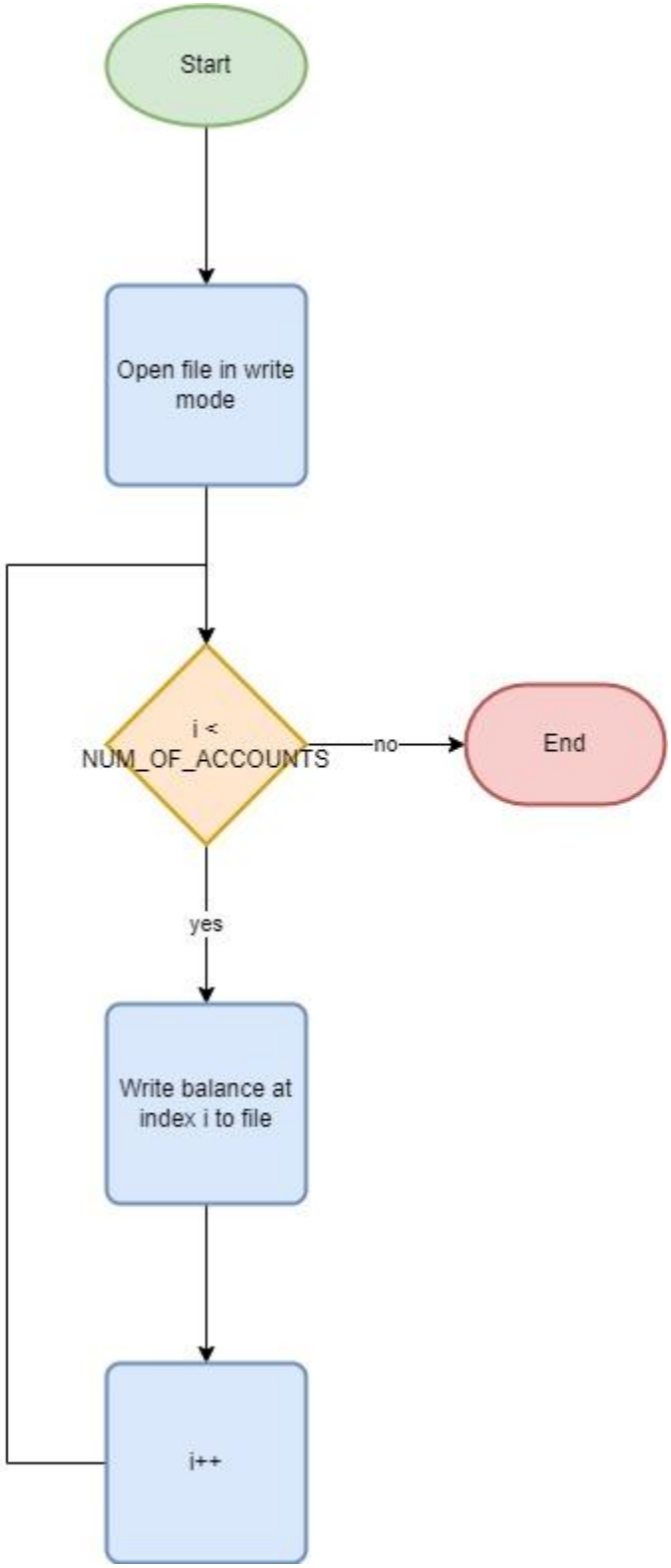


B.3.5 getY(string node)

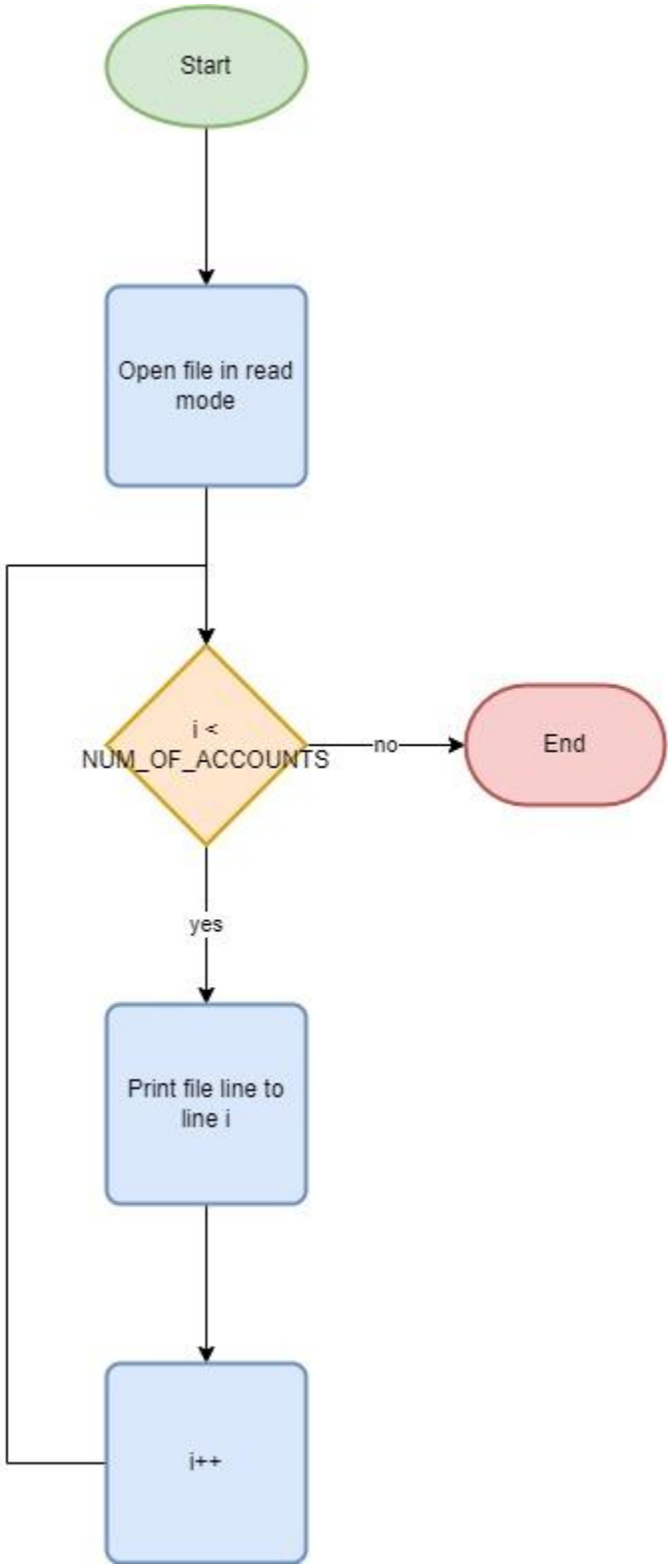


B.4 Payment Processing Flowcharts

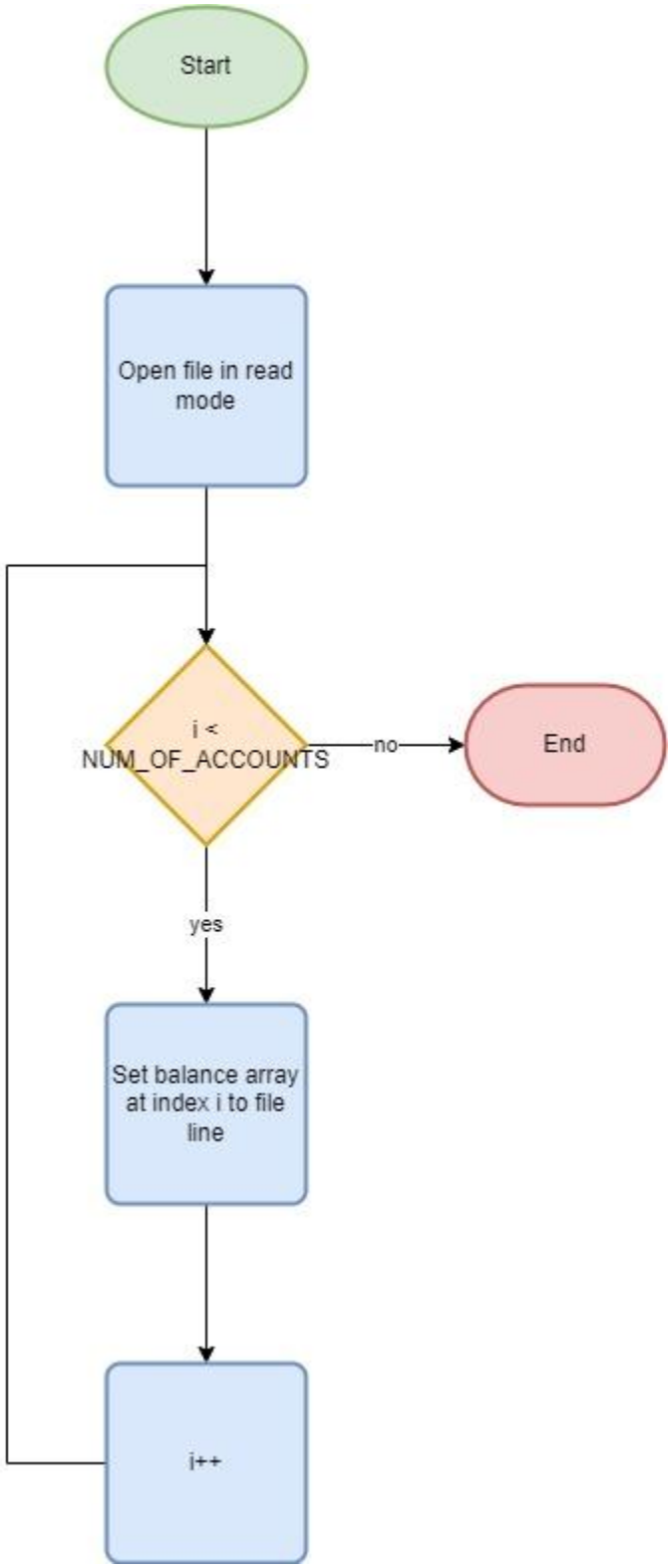
B.4.1 initialiseDB()



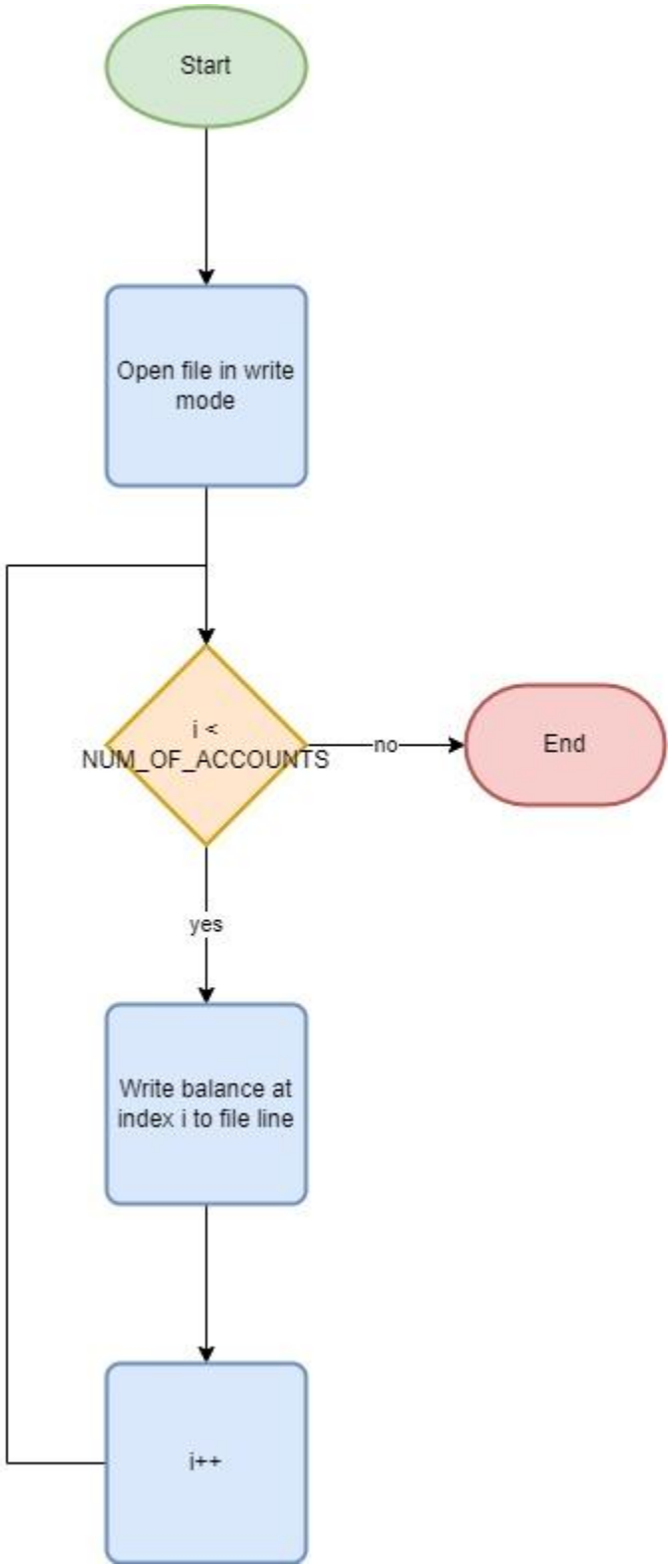
B.4.2 printDB()


Start
Open file in read mode
i < NUM_OF_ACCOUNTS
no
End
yes
Print file line to line i
i++

B.4.3 createLocal(float *localDB)



B.4.4 postLocal(float *localDB)



B.4.5 readCard(float cost, float *balances)