Your Personal Tic-Tac-Toe Robot

Department of Mechanical and Mechatronics Engineering

A Report Prepared For:

MTE 100 & MTE 121

Prepared By: Group 1 Stream 8

Aidan Hynes (21015093), William McIntyre (21009235), Winters Xia (20995653), Jeffrey Luo (21004466)

Date: Monday, December 5, 2022

Summary

This report looks at the design process, implementation, creation and testing of a Tic-Tac-Toe Robot that allows quick and easy fun when one does not have a partner. With this goal in mind, the group built a robot that can successfully play Tic-Tac-Toe in a constrained environment. The report will go over nine primary sections:

- Introduction
- Scope
- Constraints and Criteria
- Mechanical
- Software Design and Implementation
- Verification
- Project Plan
- Conclusions
- Recommendations

Each section will be split into subsections outlining how, and why the robot does what it does, while also explaining the choices that went into it and some reflections after completion. Overall, the project was a success that ended with a working robot that could successfully play Tic-Tac-Toe, and four group members who expanded their mechanical and programming knowledge.

Acknowledgments

We would like to acknowledge Aidan Hynes for his past minimax code made in Python. We would also like to acknowledge Alex Liu for helping figure out the File I/O code. Additionally, we acknowledge M. Koulouras for his YouTube video [1] serving as a inspiration for our penholder and functionality of the robot.

Table of Contents

List of Figures

List of Tables

1.0 Introduction

1.1 Problem

At the University of Waterloo, all the engineering students are academically motivated, and thus do not have the time to find a partner to play games such as tic-tac-toe. We want more engineers to be able to play tic-tac-toe, so we decided to make a robot that made playing easy, efficient, and fun. This robot will not only play the game, but can draw the board, play on different difficulty levels, and keep track of your past scores. This robot will keep all engineers company and provides a fun game to play during breaks.

2.0 Scope

2.1 Main Function

The robot's function is to play Tic-Tac-Toe with the user. There are many tasks the robot must be able to complete in order to achieve this goal, but they all stem from three primary goals. To begin, the robot must initially correctly draw the Tic-Tac-Toe board. This ensures that the user knows where each valid square is, allowing for easy scanning and input. The second significant task is to make move decisions based on the difficulty the user selects in the menu. There is an easy, medium, and hard mode and in each mode the robot makes different kinds of decisions (discussed further in Section 5.0). The last main task the robot must complete in order to be successful is recording high scores in a file. This file contains the high scores for each difficulty and after each game the robot will update the file if a high score is surpassed. The main tasks play a significant role in evaluating the success of the robot, but there are others that must be completed as well. A task list outlining the general flow of use and what must be successful is detailed below:

- The robot starts when it is turned on, and immediately turns on the GUI and asks for input on difficulty from the user.
- When the difficulty is selected, the robot draws the tic-tac-toe board.
- The robot operates by resetting to the default position (sensor/pen in bottom left closest to robot), and awaits a move, signaled by the user touching the touch sensor.
- The robot scans the board using the colour sensor, and outputs to the user if they made an invalid play.
- The robot plays a spot based on difficulty level (easy will be random valid spot, medium will either pick the best move or random move, and hard mode will always make the best move).
- Continue looping through playing the game, user vs robot, until a terminal board condition is met (the user or robot wins, or there is a tie).
- End game, and display to the user the result, as well as the cumulative score (1 for winning on easy, 2 for winning on medium, 3 for winning on hard) and the option to play again. These scores will also be scored on a file that can be downloaded.
- If the user selects to end, exit the code, if the user selects to play again, another board is created.

2.2 Inputs and Interaction with Environment

To find success, the robot must be able to take in inputs from the environment and the user. The main inputs will be from the EV3 brick buttons and motor encoders, as well as a light sensor, a touch sensor, and a file.

Once the program starts up the user is taken to a menu screen. The four main options on the menu are to choose easy, medium, or hard mode and to view the high scores. The left button (as seen in Figure 1.) on the brick is for easy mode, the middle button is for medium mode, the right button is for hard mode, and the down button is used to view the high scores. The up button is used to exit the game. The user is taken back to this menu after each game is completed.

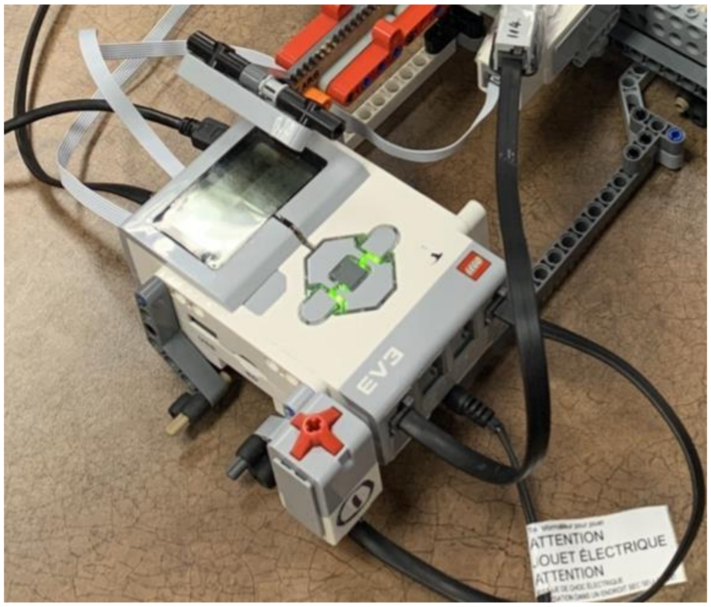


Figure 1: The Brick Setup with Touch Sensor.

Throughout the robot's operation, the arms must move accurately in both the x and y direction using a rack and pinion system and motors, visible in Figure 2. All the functions that involve moving the robot arms require the use of motors and motor encoder. One medium motor is used to translate the arm in the x direction and a large motor is used to translate another arm in the y direction. These motors in use with the motor encoder allows the robot to move to specific positions on the board accurately. Additionally, when it is used in tandem with a timer it allows the robot to reset to an origin position (see Section 5.2 for code, and Figure 2. for a visual representation). The final motor is a small motor that is used for moving the pen up and down, and has a motor encoder to calculate its distance travelled.

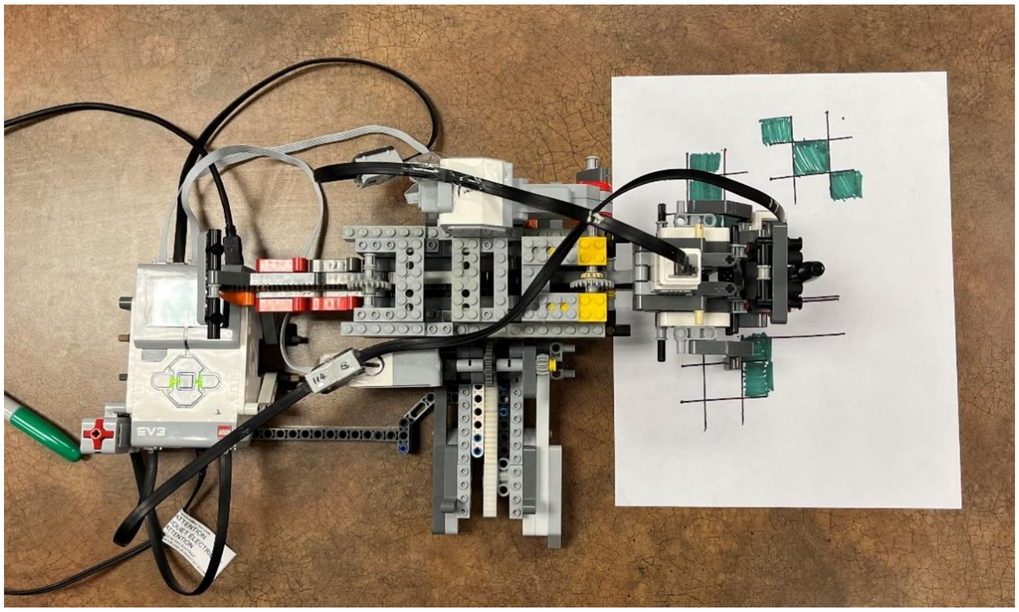


Figure 2: The Robot at the Origin Position.

During a game, the robot must be able to scan and evaluate the board. It can do this with the help of a light sensor that is calibrated to check for light intensity. If the sensor reads that more light is reflected than absorbed, then it knows that the spot it scanned is empty. If less light is reflected and more light is absorbed the robot knows that the player made a move in that spot. The light intensity is measured between zero to one hundred. If the light intensity measured at a position is less than twenty-five, the robot recognizes this spot as the user's move. Figure 3 below shows the light sensor scanning the board.

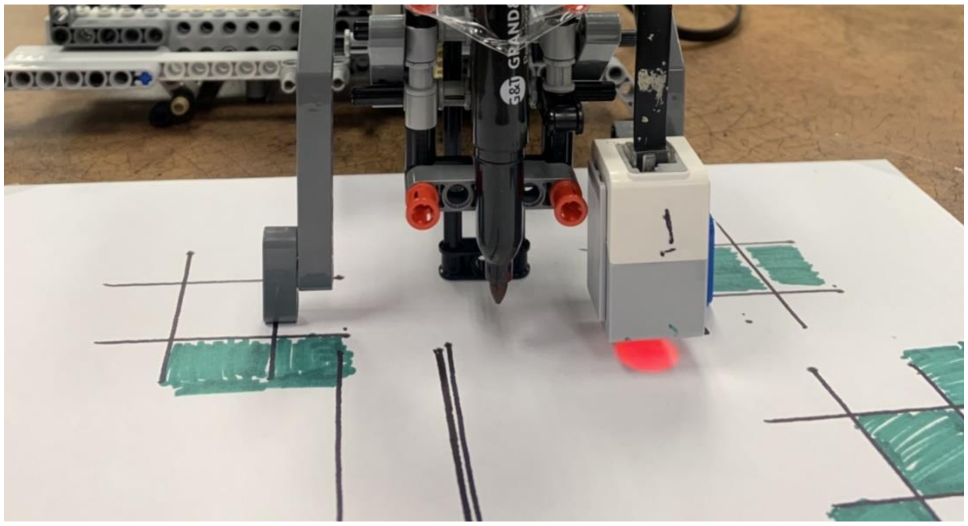


Figure 3: The Light Sensor Scanning the Board.

Finally, during the game the robot will know when to begin its move by indication from the user pushing the touch sensor. Once the user presses this sensor the robot will begin scanning the board and taking its turn. The robot is also able to record and read high scores off an input file stored in the brick, containing three rows, one for each difficulty. The robot will be able to read and display high scores from this file. It will also input new high scores if the past scores have been beaten.

All these inputs and interactions with the environment are vital to the successful implementation of the mechanical side of the project, and to get a working robot when combined with software. The buttons, light sensor, motor encoders, timers, touch sensor and files all play a significant role in the project.

2.3 Task Completion and Shutdown

Throughout the program there are many tasks. Some tasks are straightforward and have concrete beginnings and endings, whereas others loop for an unknown number of times dependant on the situation.

An example of a straightforward function that has a concrete beginning and end is the drawBoard function. The robot will draw four lines that form the board. Once the four lines have been drawn and the robot returns to the origin position the robot knows it has completed the draw board task. For tasks that loop, the robot will complete a few calculations that determine the number of times the function needs to be looped. One example of this would be the graphical user interface, or GUI. The GUI will loop on the condition that the left, middle, right, and up buttons are not pressed. When one of these buttons is pressed the program will exit the loop and the robot will know that the task is completed. Whenever there is a loop in a task there is always a condition that determines when the task ends.

Due to fact that most functions contain loops, the program will not be able to exit at any time. The user exits the program in the menu screen by pressing the up button.

2.4 Changes to Scope

Throughout the project many changes were made to the scope and design. The minimax function that makes the robot's decisions went through a considerable redesign.

After creating the initial implementation of minimax, the group realized that it was more difficult and less accurate to create a subjective version of minimax. Subjective minimax gives each spot a value based on the connections it creates for the player minus the connections it creates for the opponent.

Instead, the group redesigned the minimax algorithm to be objective, as there are a maximum of eight moves to bring the board to a terminal state which can be evaluated reasonably. Objective minimax recursively evaluates the board to a terminal state, taking the board state value at that point and factoring in how long it took to get there (the depth).

Originally, a timer was intended to be used for a shutdown procedure. The program was supposed to shutdown after 1 minute of inactivity. This would have involved adding loops and conditions to an already growing list of conditions, so the inactivity timer function was cancelled.

A few planned functions were also cancelled because they would not have helped the program but only add more lines of code. Also, the robot was originally supposed to draw an X where its move would be. This became optional because making sure the program worked was a higher priority.

There were also many changes made to the mechanical design of the robot. Originally the robot was designed to have a jointed arm that would be able to extend across the entire board. The rack and pinion design was chosen over the jointed arm because it was simpler and less part intensive. Additionally, it would make the movement code much simpler than the jointed arm. There were also 2 major iterations of the rack and pinion design too.

3.0 Constraints and Criteria

3.1 Constraints

The constraints remained relatively unchanged throughout the course of the project. The sole meaningful change in constraints was that the user's colour originally could not be the same as the robot's colour. But since light reflection was used instead of colour detection, and the robot records where its pieces are, this constraint became obsolete.

A major constraint was the limitations on the size of the board. Due to the shortage of gear racks, the robot's arms are only able to move a certain distance in the x and y directions (seen in Figure 4.). Thus, the size of the board needed to be fit within that range. This also creates the constraint that the robot must draw the board. This makes the code simpler since the robot knows exactly where each position on the board is and these positions will not vary. To prevent the robot from catching on the paper and messing up the game, the paper must also be taped down. The final constraint is what allows the robot to know where the player's piece is. Due to the limitations of the colour sensor, the user must shade in the entire box in which they want to make a move. This ensures that the sensor will be able to consistently pick up the user's move.

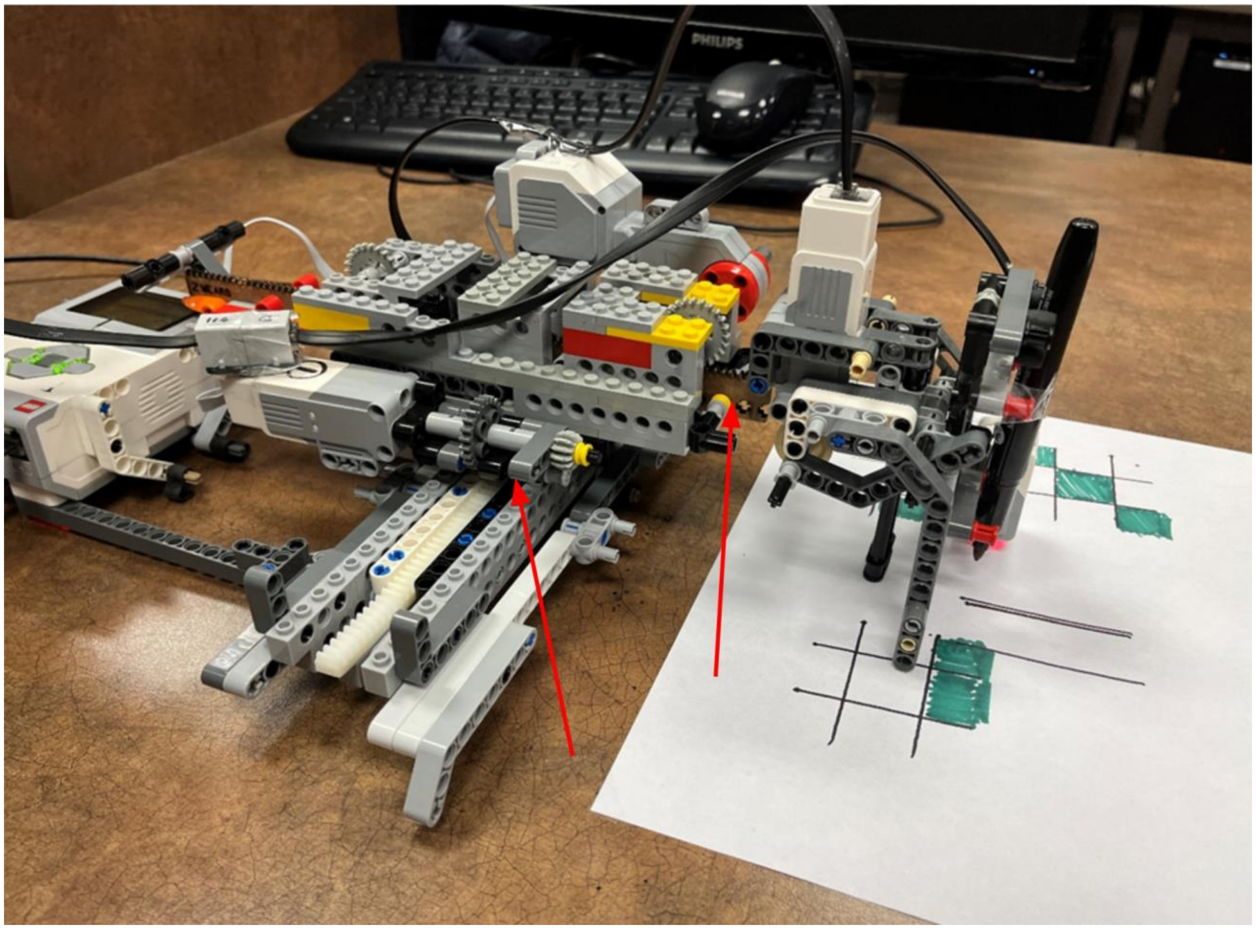


Figure 4: The Rack and Pinion System, Showing Limited Movement Space.

The two most valuable constraints that drove many of the design choices of the robot and code are the limits on x and y movement and the robot must draw the board. The shortage of gear racks resulted in a reduction in size of our robot. This also resulted in the reduction in size of the board. Having the robot draw the board meant another function needed to be created.

The requirements that the paper is taped down and the user must fill in the entire box are both optional but makes the game run more smoothly. Even if the robot is not able to sense the user’s move it will display a message requesting the user to remake their move or fill in their square a bit more so that the sensor can find where the user’s move is.

A list of the constraints and changes that occurred are described in more detail in Section 6.0.

3.2 Criteria

The criteria for the robot have not changed since the beginning. The program must begin with a menu allowing the user to select a difficulty or view high scores. There must be an option to exit the code at some point. The robot must successfully draw the board. The robot must be able to scan the board and determine which spots on the board are open using the EV3 colour sensor. It should react to the user’s move with a decision that corresponds to the appropriate difficulty. It should be able to determine the best move as a reaction to the player’s move and it should correctly execute its move. The program should also be able to record and keep high scores if the file is not reset with another program. All these criteria guided the design of the mechanical and software portions of the robot.

4.0 Mechanical Design

4.1 Overall Design

The Tic-Tac-Toe robot is a stationary robot that uses rack and pinions to move a sharpie around a piece of paper. Originally the two contending designs were a jointed arm like those found in an automobile factory (See Figure 5.), or the rack and pinion gantry system (See Figure 6.). Each design had its pros and cons.

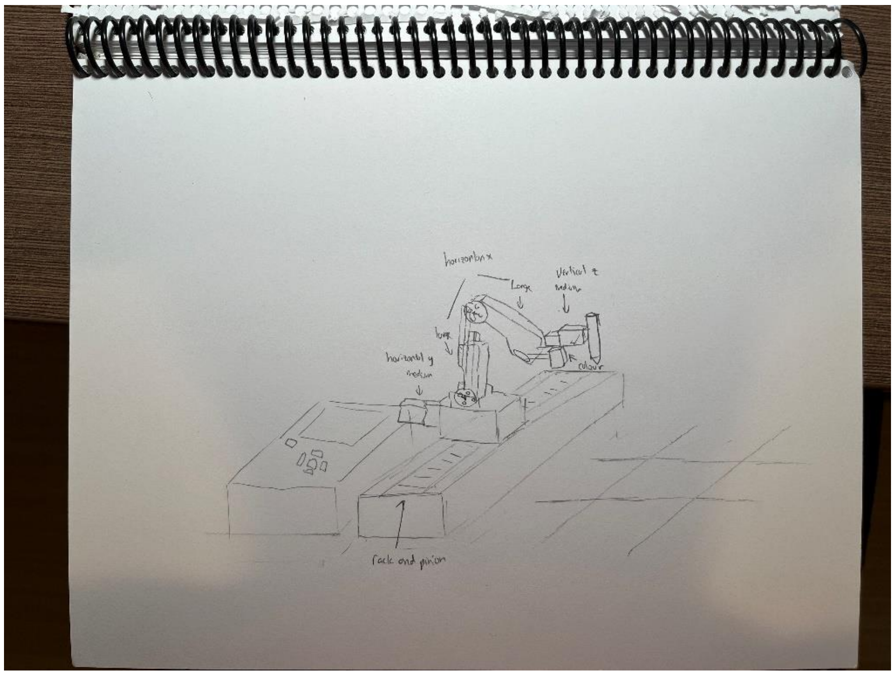


Figure 5. The jointed arm design sketch.

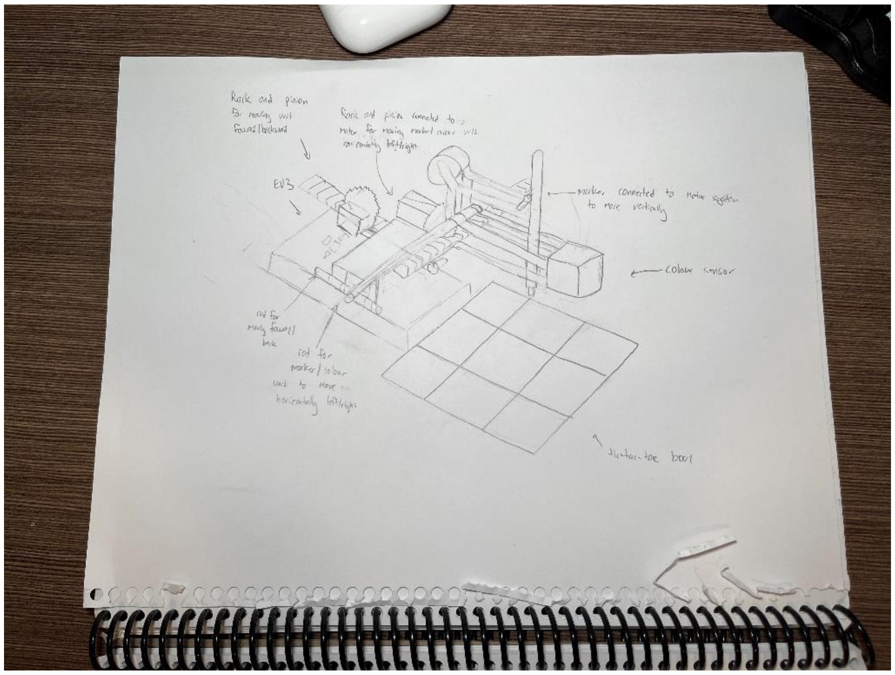

Rack and pinion for moving unit forward/backward
Rack and pinion connected to motor for moving marker/colour unit horizontally left/right
EV3
marker connected to motor system to move vertically
colour sensor
rod for moving forward/back
rod for marker/colour unit to move horizontally left/right
tic-tac-toe board

Figure 6. The rack and pinion design sketch.

The jointed arm would have more range and thus was more flexible when it came to board size. However, because it would have rotational and translational movement, programming any move function would become a massive headache.

The rack and pinion system would be easier to construct and would be less part intensive compared with the jointed arm design. It would also be simpler to program because of only translational movement. This design was chosen because it would be easier to design and construct in the limited time given (seen in Figure 7.).

4.2 Main Base

The main base of the robot consists of the EV3 brick and the rack base. Due to the low coefficient of static friction between plastic and basically any other material, the base needed to have some sort of other material to keep it stable. This resulted in the usage of 1x2 rubber axle connectors to prevent the base from slipping. However, the 1x2 rubber axle connectors are slightly thinner than the average Lego technic lift arm so bent lift arms were used to displace the 1x2 rubber axle connectors so that only the rubber parts would be in contact with the table. This increased the stability of the robot through the increase in the coefficient of static friction between the robot and the table. The EV3 brick was brought further away from the rack base because the user needed to be able to view the display in order to use the menu.

Since the rack base is the primary part keeping the robot stable, the 1x4 gear racks were used instead of the laser cut gear rack. The brick gear racks provided a wider base on which the driving gears could rest on. In addition, by combining technic lift arms and technic bricks, a very stable base was constructed. However, this was still not enough stability to support the main mechanical section. Two guiding slots were built into both sides of the rack base as well to add to the stability. This in turn prevented the central housing from tipping as a result of the constant change in centre of gravity as the extension arm moves back and forth in the y direction. In the end, the choice further reduced the range of motion of the robot but was important for stability.

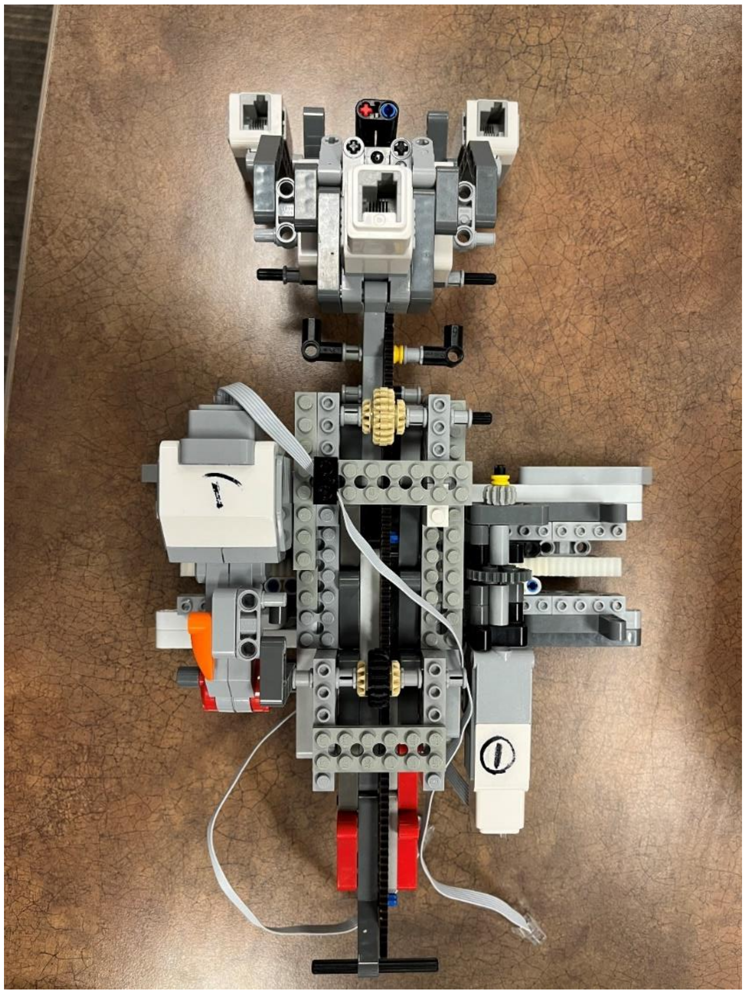


Figure 7. Top View of the Robot, with the Base Shown (Horizontal Portion).

4.3 Central Housing and Extension Arm

The main mechanical section has two parts, the central housing and the extension arm. The central housing holds both movement motors and the guide rail for the extension arm. The original central housing was much smaller than the final version. However, with testing, the smaller housing was not able to support the extension arm. The central housing was completely reworked to incorporate a larger guiding rail for the gear rack. The medium motor that controls motion in the x direction is attached on the right side of the central housing. The medium motor controls the driving gear. On the left side there is another gear that is free floating which provides more stability.

On top of the housing there are two gears which control the extension arm. One of these gears is driven by a large motor which is attached on the left side of the housing. The extension arm required improvisation with parts usage as the laser cut gear rack that was provided was roughly half of the width of a technic lift arm. This caused an issue because it could not be centred along the arm. As a result, it was left offset, but the driving gears above would also be offset. Many technic lift arms were placed on the sides of the gear rack to provide more stability to the extension arm and to provide a means to move the arm along the guiding rails. The central housing and extension arm can be seen below in **Error! Reference source not found.**8.

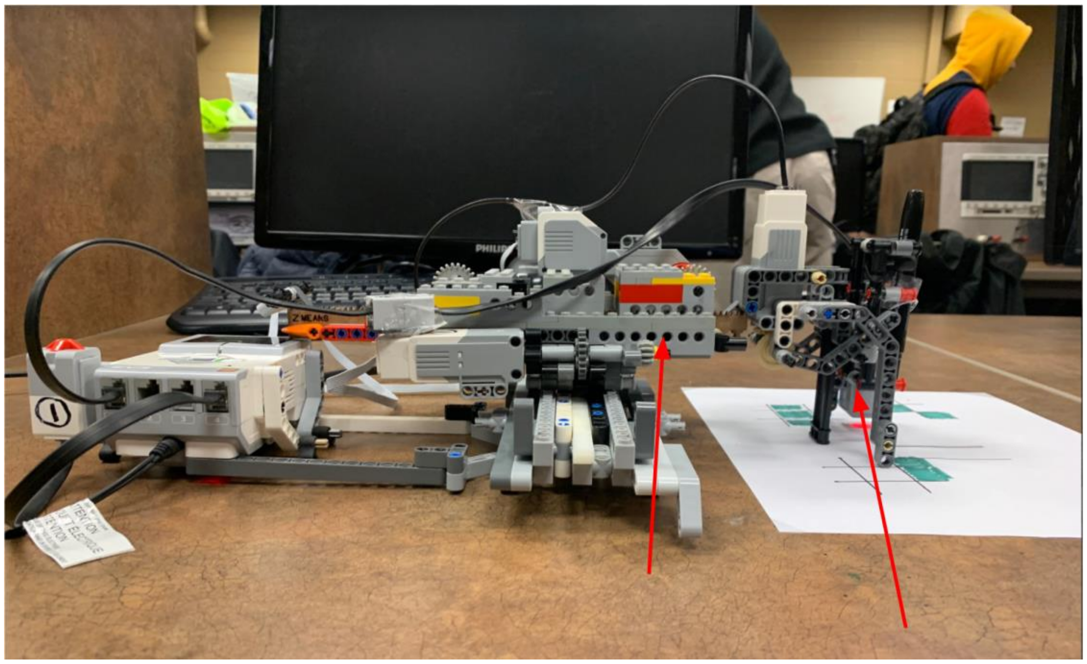


Figure 8. Central Housing (Left) and Extension Arm (Right).

The sharpie lift system is attached to the end of the extension arm. This system uses a medium motor facing downward. This required the use of two bevel gears to raise an arm connected to the sharpie lift. The sharpie lift is free floating on a vertical axle, and it is resting on the arm that is connected to the medium motor. When the arm rises so does the sharpie lift. To keep the extension arm from wobbling up and down two guiding arms are placed on either side to act as supports, and tape to keep it in place. The setup is below in Figure 9.

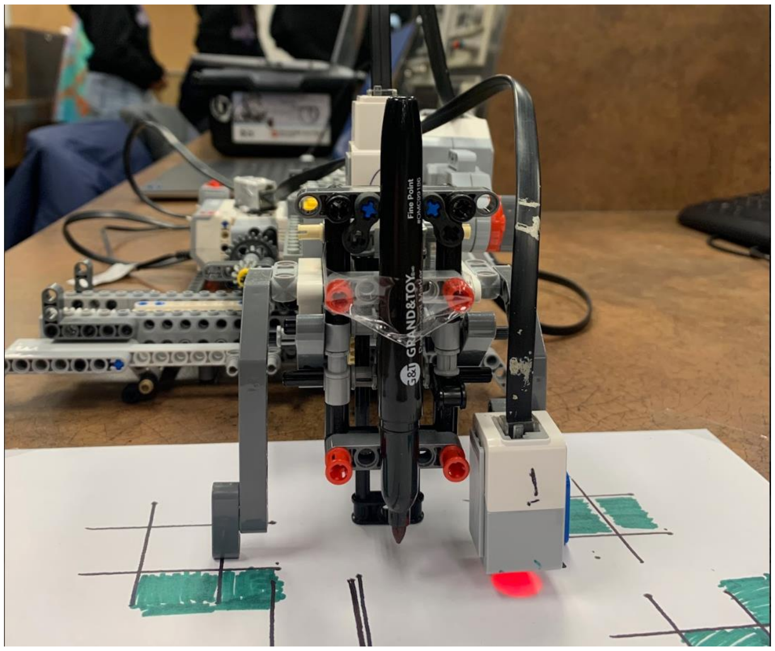


Figure 9. View of Robot from Front with Sharpie Lift System.

4.4 Sensors

The touch sensor is attached to the EV3 Brick allowing for the use of the shorter cables. The light sensor is attached one of the supports on the extension arm to the left of the sharpie, as seen in Figure 10. This displacement needs to be considered when programming the robot to scan the board. On the other hand, this decreases the size of the board even more. Also, as a result of the large distance between the EV3 Brick and the light sensor an extra long cable was needed. This extra long cable however, kept getting in the way of the gears so it needed to be taped to the robot.

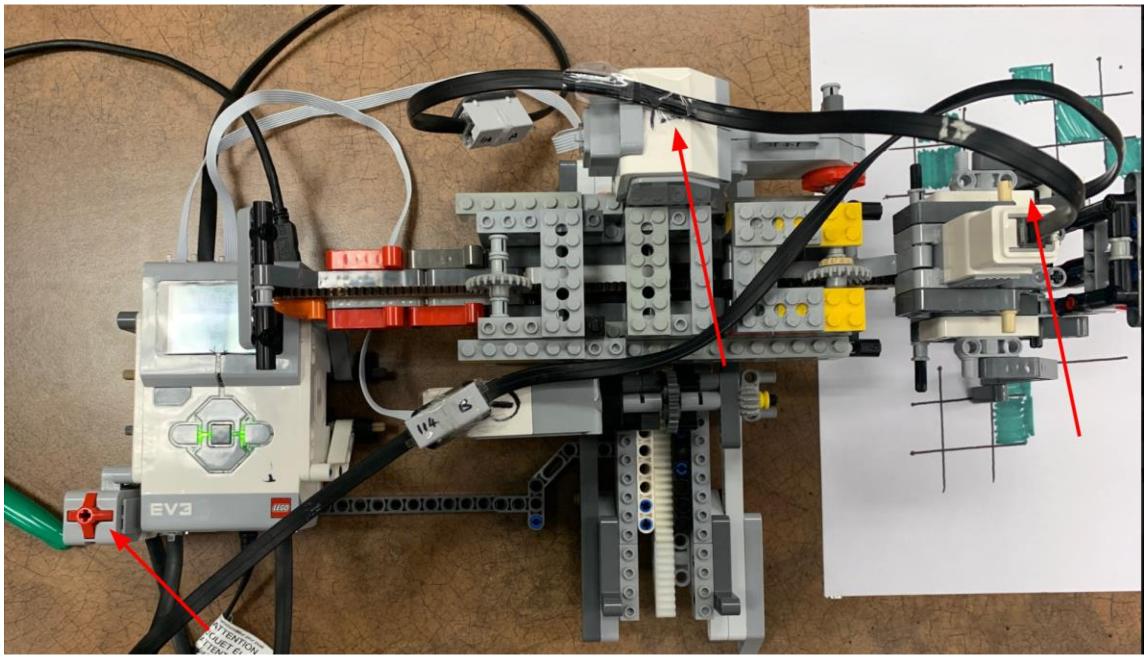


Figure 10. (From Left to Right) Touch Sensor, Extension Cables and Colour Sensor.

4.5 Assembly

The entire robot can be separated into three main sections for easy transport. These three sections are the EV3 Brick, the rack base, and the main mechanical section. The EV3 Brick and rack base form the actual base of the robot. The purpose of this structure is to provide a secure platform on which the main mechanical portion can travel on without tipping. The main mechanical portion can be further separated into two sections. The first section is the central housing. This central structure has both directional motors attached to it. This housing is also the section that travels in the x direction on the main base of the robot. The second section is the extension arm. This arm slides into the central housing along its gear rack. This provides the y directional movement. At the end of the extension arm is a system that allows the sharpie to be raised and lowered using a rotating lift arm and two guiding axles. Next to this system is the light sensor.

5.0 Software Design and Implementation

5.1 Tasks

The main program for the robot is broken into three sections. The menu, the decision-making algorithm, and the game playing portion. These three sections each have separate functions within them and are split into different sections based on the criteria list. Each criteria set has one to four functions corresponding to it. Any section of code that executes more than once became a function. For instance, the criteria stating the program should start with a menu consists of two functions. One function is the display and menu that takes in user input. The other function is used to display the high scores. A list of all tasks required for the robot to be successful on demo day is below, and the functions correlating to these large tasks are described in Section 5.2:

- Robot starts up successfully, taking in difficulty from the user, and builds the board. This sets up the game for the rest of the round and begins our program.
- Robot correctly scans all board spots and successfully stores the updated board condition, and outputs to the user if an invalid move occurs.
- Robot makes a move based off selected difficulty, and correctly plays in that spot.
- Robot continues to play the game until a terminal condition is met, and then correctly shuts down or plays again depending on the user choice displaying the cumulative score of the user's past games on an output file.

There is also another program called createEmptyFileTTT that creates and/or resets the file that stores the high scores. This part of the code needed to be run separately because if it was not, the code would reset the file each time it opened. Because of the way RobotC file I/O works the robot will not be able to read a text file that is edited by the user. This meant that the robot had to create the file itself including the 0s. Therefore, if the robot reset the file each time the program started, the high scores would be deleted. Thus, a separate program had to be created for the file creation/reset.

5.2 Functions

There are a wide range of functions within the main code for the robot, each with a unique task and use. These functions are split between all group members and have differing parameters as well as return types. Table 1 shows a list of these functions, and provides clarity of the parameters, return type and author of each. Each function is explained in further detail in the following sections, and has a correlated flowchart found in Appendix B in order.

Table 1: Functions in Main Program Including Parameters, Return Type & Author.

Function Name	Parameters	Returns	Author
displayScores()	None	Void	Winters
writeOutput()	None	Void	Winters
writeToHighscore()	4 long integers	Boolean	Winters
GUI()	None	Integer	Winters
checkForWin()	1 integer array, 1 integer	Boolean	Aidan
isTerminal()	1 integer array	Boolean	Aidan
minimax()	1 integer array, 1 integer, 2 long integers, 1 Boolean	Integer	Aidan

bestRobotMove()	1 integer array	Integer	Aidan
robotMove()	1 integer array, 1 integer	Integer	Will
trackReset()	None	Void	Will
moveMotor()	2 integers, 1 Motor	Void	Will
MovePenSens()	1 integer, 1 Boolean	Void	Will
penReset()	None	Void	Jeffrey
penDown()	None	Void	Jeffrey
drawboard()	None	Void	Jeffrey

5.2.0 displayScores()

The displayScores function is a void type that does not take in any input. This function opens the TTT_High_Scores.txt file that is on the Brick. This file contains the previous high scores in each difficulty and the current score. The displayScores function reads these values and displays them on the brick.

5.2.1 writeOutput()

The writeOutput function is a bool type that takes in four long integerss in the form of the easy high score, medium high score, hard high score and current score. This function opens TTT_High_Scores.txt and replaces the values in the file with the new values if they have changed.

5.2.2 writeToHighscore()

The writeToHighscore function as a void type that takes in a bool for gameResult, to state whether the user has won, and int for difficulty, one for easy, two for medium, and three for hard. The difficulty variable is also the number of points received for winning a game in the corresponding difficulty. The program then compares the current user score with past high scores in which ever difficulty the user is playing in. It then uses the writeOutput function to update the file if there was any change.

5.2.3 GUI()

The GUI function is an int function that does not have any inputs but returns a one, two, or three depending on the difficulty the user chooses. This function loops while the left, middle, right, and up button are not pressed. The program will display the menu screen giving the user options. If the left button is pressed easy mode is selected, if the middle button is pressed medium mode is selected, if the right button is pressed hard mode is selected, if the up button is pressed the program will exit, and if the down button is pressed the robot displays the scores for six seconds before retuning back to the menu.

5.2.4 checkForWin()

CheckForWin is used to find if a win condition is reached for a given player. The checkForWin function takes in a pointer to the board at its current state and what player value to use to search for a win. From the given player value, the function checks the board using the relationships between the possible row and columns, as shown below. Next, the function checks the diagonals separately, as there are just two diagonal conditions. The function returns true if a win condition for the given player value is met (any three in a row on the board) and false otherwise.

5.2.5 isTerminal ()

IsTerminal is used to check if the board is in an end (or terminal) state, which occurs when a player has won, or there are no more open spots on the board. The isTerminal function takes in a pointer to the board at its current state. The function counts the number of open spots and then checks if any player has won or if the number of open spots is zero. The function returns true if there is a terminal state and false otherwise.

5.2.6 minimax()

Minimax is an algorithm the robot uses to find the "best" possible score on the board given its current state. Minimax has five parameters: a pointer to the board at its current state, the current depth, alpha & beta for alpha-beta pruning, and whether the move is being maximized or minimized. Minimax checks if there is a terminal state and, if so, returns the final board position. Otherwise, minimax finds the score for each possible move on the board using recursion, evaluating the board to a terminal state, then collapsing to return the best value. It then takes the "best" score (depending on the maximized or minimized parameter input,) discarding branches that cannot possibly be better through alpha-beta pruning. The function returns the value of the board at a specific state.

5.2.7 bestRobotMove()

BestRobotMove applies minimax to give the robot the move it should select. The bestRobotMove function takes in a pointer to the board at its current state. bestRobotMove goes through each possible move, applying minimax to the resulting board to get the corresponding values and then comparing the values to get the index of the best move. The function returns the index of the best move given a board state.

5.2.8 robotMove()

The robotMove function determines the move selection for the robot, returning an integer value from zero to eight representing a space on the Tic-Tac-Toe board (Figure 11). It takes in two parameters; one is an integer representing the difficulty, with hard being a three, medium a two, and easy a one, and the other is an integer array that indicates the current state of play on each board space. This array contains a zero if the spot is not filled, and a one if it is. The function then iterates through the board array and updates a created openSpots array with the open positions on the board. To conclude, the function will look at the difficulty level selected and call bestRobotMove() if the difficulty is hard, make a completely random move if the difficulty is easy, or have a fifty-fifty chance at each if the difficulty is medium. The returns of the bestRobotMove() function or random of the open spots will be the integer value representing a playing tile returned by robotMove().

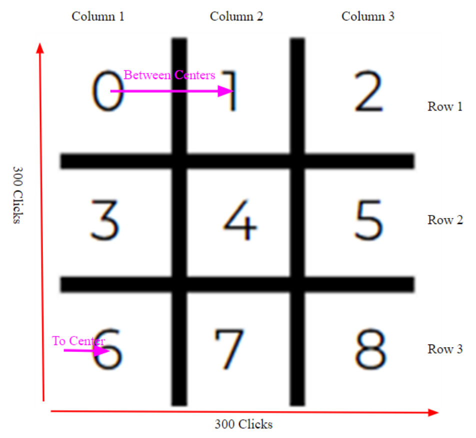


Figure 11. The Tic-Tac-Toe Board Position Numbers (With Robot set up Along Row 3)

5.2.9 trackReset()

The trackReset() function resets the arm of the robot to the default position, which is in the bottom-left position of square six seen in Figure 11. There are no parameters, and it is a void function. It initializes two Boolean-type variables xReset and yReset as true, resets the motor encoders, and starts a timer, then goes through a while loop while xReset or yReset is true. In this while loop, the function waits checks if the velocity of the arm (calculated by taking the absolute value of the motor encoder distance divided by the timer count) is below a threshold velocity that indicates if the arm has encountered resistance. If the velocity is at this point, it means that the rack and pinion system has reached its maximum movement distance, and will thus resist continued movement, and in turn change the reset variable to false, breaking the loop and ending the function. If there is no resistance detected in the loop, the timer and motor encoders will reset, wait one hundred milliseconds, and continue to loop through and calculate the velocity until resistance occurs.

5.2.10 moveMotor()

The move motor function takes in three parameters; two integers indicating the distance in motor encoder clicks and the speed of the motors, and then a motor port (using type “tMotor” so inputs can be “MotorA”, etc), and returns void. It moves the motor at the specified speed for the specified distance and then stops it again.

5.2.11 movePenSense()

The movePenSense() function is a void function that takes in two values, an integer representing the spot on the board (see Figure 11), and a Boolean which indicates whether the robot is moving the pen or the sensor. It begins by creating a constant indicating the playing length of the board, which is three hundred motor encoder clicks, and the pen adjustment distance, which is one hundred and seventy clicks. Then, it divides the playing length by three, which represents the distance between the centers of each position on the board (Figure 11), and by six, which represents the distance from a line to the center of a spot. It then determines the distances to each column/row from our reset position, by multiplying and adding these values together (see Table 2), and creates two parallel arrays with this data, with each index relating to a position’s column and row distance (for example, position six is index six and is in row three, column one). Finally, it will call the moveMotor() function with the index values from these arrays, and adjustments added if the pen is to be moved.

Table 2: Column and Row Distance Math For 300/300 Encoder Clicks Board.

Row/Column	Distance Math (Encoder Clicks)	Total Distance (Clicks)
Column 1	To Center (50)	50
Column 2	To Center (50) + Distance Between (100)	150
Column 3	To Center (50) + 2 * Distance Between (100)	250
Row 1	To Center (50) + 2 * Distance Between (100)	250
Row 2	To Center (50) + Distance Between (100)	150
Row 3	To Center (50)	50

5.2.12 penReset()

The penReset() function is a void function that moves the pen to the highest position within the pen holder. The function resets the encoder value for motor B, sets the power to negative, waits two hundred milliseconds, then begins then begins the ascent back up. The motor will continue running until it has reached its original position, tracked by calculating the velocity of the pen motor using the motor encoder and the time sensor through a calculated equation (Figure 2). This equation both allows the robot to continue moving until the pen encounters resistance and drops its velocity. After this while loop has finished, the motor will stop, and the encoder is reset.

```
while(abs(nMotorEncoder(motorB)/(time1(T2)/100)) > 7)
{}
```

Figure 12. Velocity Check in penReset()

5.2.13 penDown()

This small void function defines a constant PEN_DIST as thirty and calls upon the moveMotor() function to move motorB PEN_DIST encoder clicks for a set speed.

5.2.14 drawBoard()

The drawBoard function is a bigger function that also uses helper functions such as penReset and trackReset to optimize the code and draws the playing field. The function defines a constant PLAYING_LENGTH as three hundred, representing the board size in encoder clicks. It then cuts the board into three playing zones, defining the lines between adjacent playing zones. The robot will run through four chunks of codes, each visually like the last. This will allow the robot to draw the two vertical lines and the two horizontal lines. Each chunk of code first moves the motor to the starting position with the trackReset() function. MotorC will first move to the calculated position through the cutting of the playing board into zones, then it will run the penDown function. After waiting seven hundred and fifty milliseconds, the robot will run the moveMotor() function with motorA and will draw the entire span of the board defined by PLAYING_LENGTH. To conclude, the robot stops and runs the penReset() and trackReset() functions. This is repeated three more times for the other three calculated portions in order to draw the three-by-three playing field.

5.3 Software Design Decisions

Throughout the design of the software component of the robot, there were key decisions made to optimize and simplify the main code. The sections below explain each main decision, what it does, and the trade-offs required to implement it.

5.3.0 Sensing Moves using Only Filled-In Squares

When the robot scans the board after the user and itself played, the group decided that it should not scan over the board spaces it has already played on. This software design choice allows the scan to take much less time as it skips over spots that the robot knows are already full.

5.3.1 Removing Robot's Ability to Draw Shapes

Removing the need for precise movements and math needed to draw a shape on the board made the draw function significantly easier to code. For example, had the group decided to make the bot physically play an "X" or an "O" using calculated distances to draw, the markings would be unreliable and take more time. Although a dot on the board does not look like traditional Tic-Tac-Toe, it is much more reliable to draw.

5.3.2 Switching to Objective minimax()

By switching to an objective version of minimax, the function can evaluate the board to an end state, simplifying the code and removing the need for "magic" assigned point values (see Section 2.4).

5.3.3 Remove the timer shutdown condition.

By removing the timer end condition (see Section 2.4), the need for timer conditions in every loop and function loop was removed, significantly cleaning up the main code and making main much easier to test and debug. However, this means the robot will continue to wait for an input until there is a condition that will allow it to exit.

5.4 Data Storage

Data is stored in many different forms within the main code. Examples of the data storage types the group uses are in Table 3 below.

Table 3: Data Storage Types, Examples and Descriptions of use Within Code

Data Storage Type	Example	Description
1D Arrays	board[9]	The 1D array board is used to store the state of the board during the game with indices 0-8 matching up to board locations 0-8 (see Figure 11).
Integers	bestScore	The group uses integers all over the code. In this instance bestScore is used to hold the

		current best branch score found in minimax.
Booleans	LoopRun	The group uses booleans all over the code as well. In this instance LoopRun acts as a type of "break" command for the loops in main.

5.5 Testing

The program was tested through a multi-layered approach that used robot-C debugging features, individual function testing, and slow implementation into the main code. Each function after creation was individually tested with different parameter values and compared to the correct, calculated return or task. An example of this process is shown below, in Table 4, with the checkForWin() function and different parameters. A description of this function is found in Section 5.2, but a brief explanation is that the function returns a Boolean indicating whether the passed board state (a nine-value array shown in Figure 3) has a win by the indicated player (1: robot, 2: user).

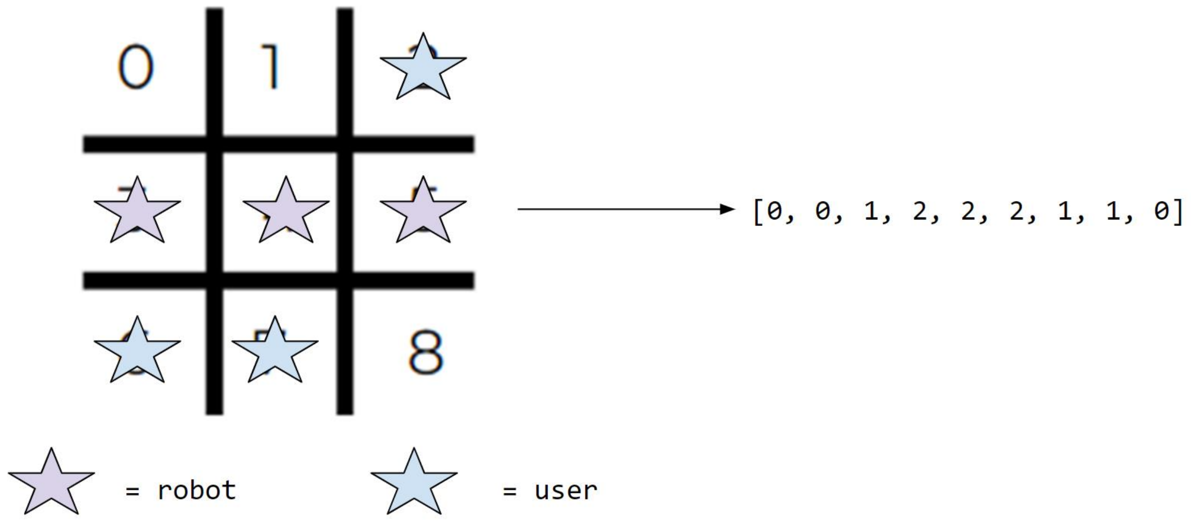


Figure 13. Example of Board State and Related Array (0: empty, 1: robot, 2: user)

Table 44: Function Calls and Expected/Actual Returns

Function Call	**Expected Return**	**Actual Return**
checkForWin([0, 0, 1, 2, 2, 2, 1, 1, 0], 2)	True	True
checkForWin([0, 0, 1, 2, 2, 2, 1, 1, 0], 1)	False	False
checkForWin([1, 1, 0, 1, 2, 2, 1, 2, 2], 1)	True	True
checkForWin([1, 1, 0, 1, 2, 2, 1, 2, 2,], 2)	False	False

After the individual testing of each function, which also included the void functions moving the mechanical portions of the robot, the working functions were implemented slowly into the main code. After each added function, the main code was compiled to check for errors, and tested similarly to that shown in Table 4. At this point, errors and bugs began to reveal themselves as functions combined into one source code (described further in Section 5.6). The testing procedure following these errors, involved the use of the robotC writeDebugStreamLine() function. An example is illustrated below (Figure 4) with the main use being to test when specific parts of the code are hit. This code was used in the main function found in Appendix A while testing. It writes out when the robot made a successful choice on where to play in the debug stream, and through this test indicated that the minimax() function was taking prolonged time to make move decisions (view Section 5.6 for more details on how this issue was resolved).

```
for(int i = 0; i < 9; i++)
{
        if(board[i] == 0)
        {
                board[i] = ROBOT;
                //calls itself with one higher depth and minimizing
                int score = minimax(board, depth+1, alpha, beta, false);
                board[i] = 0
                if(score > bestScore)
                        bestScore = score;
        }
        writeDebugStream("Score: %d", score);
}
*/ Debug Stream Output, the values take 15-30 sec to appear each and for a
beginning board state 8 spot values must be calculated. This output also
shows that the code is successfully running, as values are being calculated
and displayed.
96
-94
98
95
-97
*/
```

Figure 14. Example of writeDebugStreamLine() Use.

5.6 Significant Problems

One of the major problems encountered was the integration of the GUI into the Main code. Once the GUI was added, the robot would start as intended, but when the player selected a mode, the robot would draw the board, but the player would be taken back to the GUI. No matter how many conditions were added or removed from the parent loop the GUI would continue to loop. After using the robotC writeDebugStreamLine() function to display all the conditions, the issue was revealed. The GUI function was returning a value that made the terminal state of the board true no matter what. The solution was to create a variable called loopRun that would be false if the return value of GUI was negative. The GUI returns the difficulty level which will never be less than zero, so loopRun will never be false. This then became a condition for the game loop which solved the problem. Once loopRun became false the program would successfully end and the problem was resolved.

Another issue regarding the GUI occurred inside the function itself. The option for displaying the past high scores would not loop correctly back to the menu as intended. It would just end the function. The way this issue was resolved was by adding a variable called loopValue. While loopValue was greater than 0 the loop would run. This would only apply when the down button was pressed for display values because if the left, middle, or right buttons are pressed the difficulty is returned meaning the function has ended.

After discovering the long computation time for minimax on the EV3 brick (as discussed in Section 5.5), the group decided to optimize the algorithm through alpha-beta pruning. Alpha-beta pruning allows the algorithm to break out of a branch search when there is no way the branch could be "better" than a previously evaluated branch. In this application, alpha-beta pruning significantly decreases computation time, especially in cases with multiple equally rated spot values, such as a beginning board state. Through the implementation of the alpha-beta pruning, the group reduced the calculation time for a beginning board state from one-two minutes to about twenty seconds, worst case.

6.0 Verification

6.1 Constraints

The robot must be constrained within a few conditions in order to successfully play the game and complete all its tasks. The updated list of constraints for use in the final demo is below, in Table 5, with descriptions on how each were met.

Table 55: Met Constraints and Factor of Verification

Constraint	Verification
Tic-Tac-Toe board size has a max size of 300x300 motor encoder clicks, due to the size restrictions of the rack and pinion systems.	The robot successfully reaches each position with the light sensor and can reach each position with the pen.
The user's plays must be drawn in a darker colour and cover much of the desired spot in order to be sensed by the colour sensor.	Robot correctly senses the coloured spots on the board (can be confirmed with writeDebugStreamLine() when robot scans a filled in spot) and makes a move to an empty spot.
The paper provided must be white and blank so the colour sensor accurately senses moves.	Each filled spot is correctly scanned without error, as above.
The paper must be taped down to prevent movement when the robot is in use.	The paper does not move when the robot scans or plays on the board.
Robot must be placed on a level surface within reachable distance of the paper.	The pen fully extends to the board material and draws legibly.
Robot can tell when a player has inputted an invalid move (i.e., two turns in one).	The robot ends the game and outputs on the GUI that the user made an invalid play.

6.2 Dropped Constraints

All constraints that are initially listed in the preliminary report are met, except for a single constraint that is now updated due to a design change. Initially, a constraint for the robot was that the user must use an unchanging, dark coloured pen for their moves. The design of the robot at the time used the colour mode of the colour sensor, and thus needed a specific colour for every move in order to sense a filled spot.

Using a specific-coloured pen every game is not user-friendly, as it requires effort to get the same-coloured pen each game. Thus, the current design of the robot uses the colour sensor in ambient mode instead, measuring the reflective light intensity. Ambient mode allows the user to instead use any darker coloured pen, rather than a specific colour, and updated the constraints to no longer contain a colour restriction.

7.0 Project Plan

7.1 Tasks

The project management plan for the robot was created early in the design process, and includes tasks, deadlines, and work distributions for the entirety of the work timeline. The group goal was to split up the project into main steps, and then assign group members and a timeline to smaller, more specific tasks within them. A Gantt chart was used (Figure 5) to measure timelines, as well as a table splitting up tasks to specific group members (see Table 6). Each group member had different strengths, and the tasks were split to account for these. For example, one group member who had prior EV3 experience had a task to complete the main base portion of the robot in the same week where another was working on the pen holder, and the third, who had experience programming games was creating the algorithmic code. This plan played a significant role in the success of the Tic-Tac-Toe robot construction and meeting the deadline.

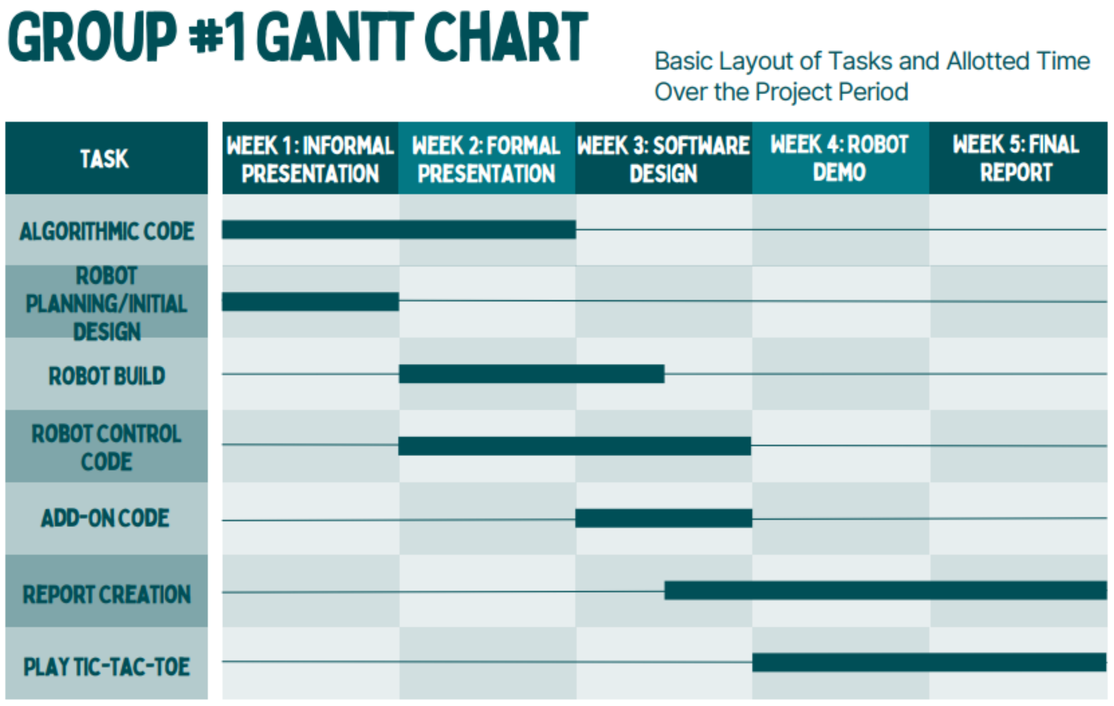


Figure 15. Gantt Chart with Main Tasks

Table 66: Main Tasks & Sub Tasks with Work Allotment

Main Tasks	Sub Tasks	Allocated Group Member
Build Algorithm Code	- Create functions required for robot to play intelligently (see Section 5.2 for more details).	- Aidan
Robot Planning/Initial Design	- Draw and brainstorm possible designs. - Create document for informal presentation questions to answer.	- Will, Winters - Will, Jeffrey

	- Research Tic-Tac-Toe robots for inspiration.	- Jeffrey, Aidan
Build Robot	- Build base of robot. - Build pen holder. - Build rack/pinion system.	- Winters - Will - Winters
Control Robot Through Code	- Create functions that move robot arm and pen, reset them, and other movement functions (see Section 5.2 for more details) - Make formal presentation and software presentation, describing individual functions made.	- Will, Jeffrey, Winters - Aidan, Will, Winters, Jeffrey
Add-on/Test Code	- Create GUI code. - Implement difficulty level/file input.	- Winters - Aidan, Winters
Report Created	- Introduction - Scope - Constrains and Criteria - Mechanical Design and Implementation - Software Design and Implementation - Verification - Project Plan - Conclusion - Recommendations	- Jeffrey - Jeffrey, Will - Jeffrey, Aidan, Will - Winters - Described by author (see Section 5.2) - Will - Will - Aidan, Jeffery, Will, Winters - Winters, Aidan
Play Tic-Tac-Toe	- Successfully demo robot	- Winters, Will, Aidan, Jeffrey

7.2 Revisions to Plan

Like many plans, the timeline and tasks layout required revisions and edits as the project went on. Some tasks took longer than expected, others took less time, and unexpected issues appeared that changed our project timelines.

The most significant change in the robot plan and timeline was the allocated time towards the software side of the robot. The algorithmic code and robot move code took longer than expected and was the main task that all group members worked on leading up to demo day. As each function got added to the main source code, bugs were revealed and tested, which took time and effort beyond what the Gantt chart described. Luckily, the physical building of the robot took less than a week rather than the allotted week and a half, so starting the coding earlier helped greatly. However, due to the time spent on getting a working robot, there was in turn less time to work on the final report, until after the demo.

8.0 Conclusion

8.1 Summary

The group's goal through the design and build of the Tic-Tac-Toe robot was to create an easy, efficient and fun way to play Tic-Tac-Toe without the need for a partner. With this goal in mind, the group built a robot that can successfully play Tic-Tac-Toe in a constrained environment (Section 6.1), while accomplishing the tasks required to play the game (Section 5.1). This was confirmed on demo day, when the robot completed multiple full games of Tic-Tac-Toe against varied opponents and with different difficulties, while storing the game results as intended. Overall, the project was a success, with the group accomplishing every goal set out in the project plan, and successfully making a robot to make being in engineering (a little) less lonely.

8.2 Important Features

The robot has a few significant features in both the software and mechanical design choices, as talked about throughout this report. In summary, mechanically the robot uses a rack and pinion system to move an arm holding a pen that can move up and down. The robot can move in both the x and y direction relative to itself to the physical extent of the long racks used within the system, and can draw the Tic-Tac-Toe Board, scan it, and play on it. On the software side, the robot can interact with the user through the EV3 Brick and receive input on difficulty level, changing the way that the robot plays dependant on what the user wants. It recognizes wins, losses and ties, outputs scores to a file, and even knows when invalid moves are made. All of these features have helped in creating a Tic-Tac-Toe playing robot that is unique and highly proficient in playing the game.

9.0 Recommendations

9.1 Mechanical

Based on the verification data and our experience building the robot, there are some changes that can be made that would improve the robot and increase the efficiency of the robot. One of the major things that would be changed would be the use of an acrylic gear rack instead of the laser cut one that was provided. After the robot program was demonstrated on Demo Day, the robot's gears began to slip when the program was run again. The issue found was that the laser cut gear rack was worn down and many of the teeth were broken due to the constant testing done on it. Since the laser cut wood is softer than the plastic Lego gears, the gears broke some of the teeth on the gear rack from the strain of moving the robot. If an acrylic gear rack was used, it is assumed that the rack would not wear down as much if not at all. This would increase the reliability and the life span of the robot.

A change that would improve the performance of the robot would be the use of a better colour sensor that would be able to sense coloured lines. This would allow the robot to read the player's move as an O or X instead of a large coloured in square. As was found during the testing of the robot, the EV3 colour sensor is not strong enough to pick up a line drawn by the user. This resulted in the constraint that the player must fill in the square for their move. With a better colour sensor, the board could also become bigger which would improve the ease of access for the user.

Another improvement to the design of the robot would be the use of a bearing ball on the extension arm. The robot's extension arm used two technic lift arms for support. These arms dragged on the paper which caused a lot of friction and made the arm difficult to move at times. If these arms were replaced by a bearing ball, the arm would still have support, but it would also have better mobility.

9.2 Software

A change the group would make on the software side if given more time is implementing a GUI selection that allows the player to select who goes first. The group would also try to implement some of our software changes, such as timer shutdown, robot shape drawing, and robot shape sensing (see section 5.3). Optimization-wise, the group would clean up main more, including more powerful break commands that can work with a timer, and create a set first move if the robot goes first to reduce calculation time. The group would also make the GUI more powerful: including better use of the files, more user options, and more advanced difficulty selections.

References

[1] M. Koulouras, "Lego Mindstorms NXT Robot - Tic Tac Toe (X play)," YouTube, 4 July 2012. [Online]. Available: https://www.youtube.com/watch?v=x1Q8h7qegjk. [Accessed 1 November 2022].

Appendices

Appendix A: Main Code

```
#include "PC_FileIO.c"

const int ROBOT = 1;
const int PLAYER = 2;

//CODE FOR DISPLAY SCORES
void displayScores()
{
	TFileHandle input;
	bool isOpen = openReadPC(input, "TTT_High_Scores.txt");
	if (!isOpen)
	{
		playSound(soundException);
		displayString(5, "FAILED TO OPEN FILE");
		wait1Msec(1000);
	}
	int easyHS = 0, mediumHS = 0, hardHS = 0, currentHS = 0;
	// reading input
	readIntPC(input, easyHS);
	readIntPC(input, mediumHS);
	readIntPC(input, hardHS);
	readIntPC(input, currentHS);
	// displaying input
	displayString(0, "Easy High Score: %d", easyHS);
	displayString(1, "Medium High Score: %d", mediumHS);
	displayString(2, "Hard High Score: %d", hardHS);
	displayString(3, "Current Score: %d", currentHS);
	closeFilePC(input);
}

//CODE TO WRITE TO FILE
bool writeOutput(long easy, long medium, long hard, long current)
{
	TFileHandle output;
	bool isOpen = openWritePC(output, "TTT_High_Scores.txt");
	if (!isOpen)
	{
		playSound(soundException);
		displayString(5, "FAILED TO OPEN FILE");
		wait1Msec(1000);
		return false;
	}
	writeLongPC(output, easy);
	writeEndlPC(output);
	writeLongPC(output, medium);
	writeEndlPC(output);
	writeLongPC(output, hard);
	writeEndlPC(output);
	writeLongPC(output, current);
	writeEndlPC(output);

	closeFilePC(output);
	return true;
}

//WRITES TO TTT_High_Scores.txt AFTER SCORE CALCULATIONS
void writeToHighscore(bool gameResult, int difficulty)
{
	TFileHandle input;
	bool isOpen = openReadPC(input, "TTT_High_Scores.txt");
	if (!isOpen)
	{
		playSound(soundException); // just plays a sound for failure
		displayString(5, "FAILED TO OPEN FILE");
		wait1Msec(1000);
	}
	int easyHS = 0, mediumHS = 0, hardHS = 0, currentHS = 0, lastHS;
	readIntPC(input, easyHS);
	readIntPC(input, mediumHS);
	readIntPC(input, hardHS);
	readIntPC(input, currentHS);
	if (gameResult)
	{
		currentHS+=difficulty;
	}
	if (difficulty == 1){
		if (currentHS > easyHS)
		{
			easyHS = currentHS;
		}
	}
	else if (difficulty == 2)
	{
		if (currentHS > mediumHS)
		{
			mediumHS = currentHS;
		}
	}
	else if (difficulty == 3)
	{
		if (currentHS > hardHS)
		{
			hardHS = currentHS;
		}
	}
	lastHS = currentHS;
	if (!gameResult) {
		currentHS = 0;
	}
	closeFilePC(input); // closes the input
	// writeOutput returns a true if it writes correctly
	if (writeOutput(easyHS, mediumHS, hardHS, currentHS))
	{
		} else {
	}
	wait1Msec(200);
}
```

```
//CODE FOR GUI
int GUI()
{
        int loopValue = 0;
        do {
                displayTextLine(2, "Welcome to your personal");
                displayTextLine(3, "Tic-Tic-Toe Friend!");
                displayTextLine(5, "Select Difficulty:");
                displayTextLine(7, "Easy   (left button)");
                displayTextLine(8, "Medium (enter button)");
                displayTextLine(9, "Hard   (right button)");
                displayTextLine(10, "Display Scores (down button)");

                while (!getButtonPress(buttonAny))
                {}


                if(getButtonPress(buttonLeft))
                {
                        loopValue = 0;
                        eraseDisplay();
                        displayTextLine(8, "Difficulty level: EASY");
                        wait1Msec(2000);
                        eraseDisplay();
                        displayTextLine(8, "Good Luck!");
                        wait1Msec(2000);
                        eraseDisplay();
                        wait1Msec(2000);
                        eraseDisplay();
                        return 1;
                }else if(getButtonPress(buttonEnter))
                {
                        loopValue = 0;
                        eraseDisplay();
                        displayTextLine(8, "Difficulty level: MEDIUM");
                        wait1Msec(2000);
                        eraseDisplay();
                        displayTextLine(8, "Good Luck!");
                        wait1Msec(2000);
                        eraseDisplay();
                        return 2;
                }else if (getButtonPress(buttonRight))
                {
                        loopValue = 0;
                        eraseDisplay();
                        displayTextLine(8, "Difficulty level: HARD");
                        wait1Msec(2000);
                        eraseDisplay();
                        displayTextLine(8, "Good Luck!");
                        wait1Msec(2000);
                        eraseDisplay();
                        wait1Msec(2000);
                        eraseDisplay();
                        return 3;
                }else if (getButtonPress(buttonDown))
                {
                        loopValue = 1;
                        eraseDisplay();
                        displayScores();
                        wait1Msec(6000);
                        eraseDisplay();
                }
                else if (getButtonPress(buttonUp))
                {
                        eraseDisplay();
                        return -1;
                }
        }
        while(loopValue > 0);
        return -1;
}

//Checks if a given player has won a game (any 3 in a row possibility)
bool checkForWin(int *board, int player)
{
        for(int i = 0; i < 3; i++)
        {
                if(board[i*3] == player && board[(i*3)+1] == player &&
                   board[(i*3)+2] == player)
                        return true;
                if(board[i] == player && board[i+3] == player &&
                   board[i+6] == player)
                        return true;
        }

        if(board[0] == player && board[4] == player && board[8] == player)
                return true;
        if(board[2] == player && board[4] == player && board[6] == player)
                return true;

        return false;
}

//CODE FOR ALGORITHM

//checks if the board is in a terminal state (a player has won/no more valid moves)
bool isTerminal(int *board)
{
        int openCount = 0;

        for(int i = 0; i < 9; i++)
        {
                if(board[i] == 0)
                        openCount++;
        }

        if(checkForWin(board, ROBOT) || checkForWin(board, PLAYER) ||
           openCount == 0)
                return true;
        return false;
}

//gets the scores of each possible move and returns the best
int minimax(int *board, int depth, long alpha, long beta, bool maxing)
{
        int bestScore = 0;

        //checks if there is a terminal state and returns the board value
        if(isTerminal(board))
        {
                if(checkForWin(board, ROBOT))
                        return 100-depth;
                else if(checkForWin(board, PLAYER))
                        return -100+depth;
                else
                        return 0;
        }

        /*
        for each valid move on the board find the score and returns the highest
        possible score
        */
        if(maxing)
        {
                bestScore = -1000;
                for(int i = 0; i < 9; i++)
                {
                        if(board[i] == 0)
                        {
                                board[i] = ROBOT;
                                //calls itself with one higher depth and minimizing
                                int score = minimax(board, depth+1, alpha,
                                                   beta, false);
                                board[i] = 0;
                                if(score > bestScore)
                                        bestScore = score;
                        }
                        /*break if the maximum score is greater than or equal to the
                        minimized score*/
                        if(bestScore >= beta)
                                i = 10;
                        if(bestScore > alpha)
                                alpha = bestScore;
                }
                return bestScore;
        }
        /*
        for each valid move on the board find the score and returns the lowest
        possible score
        */
        else
        {
                bestScore = 1000;
                for(int i = 0; i < 9; i++)
                {
                        if(board[i] == 0)
                        {
                                board[i] = PLAYER;
                                //calls itself with one higher depth and maximizing
                                int score = minimax(board, depth+1, alpha,
                                                    beta, true);
                                board[i] = 0;
                                if(score < bestScore)
                                        bestScore = score;
                        }
                        /*break if the minimum score is less than or equal to the
                        maximized score*/
                        if(bestScore <= alpha)
                                i = 10;
                        if(bestScore < beta)
                                beta = bestScore;
                }
                return bestScore;
        }

        return 0;
}

//gets the index of the best move using minimax
int bestRobotMove(int *board)
{
        int bestMove = -1;
        int bestScore = -100000;
        //gets the scores of each possible move and selects the highest
        for(int i = 0; i < 9; i++)
        {
                if(board[i] == 0)
                {
                        board[i] = ROBOT;
                        int score = minimax(board, 0, -1000000000,
                                            1000000000, false);
                        board[i] = 0;
                        //selects the index with the highest score
                        if(score > bestScore)
                        {
                                bestScore = score;
                                bestMove = i;
                        }
                }
        }
        return bestMove;
}

//determines the robots move given the difficulty
int robotMove(int *board, int diff)
{
        int moveSelection = -1;
        int openSpots[9];
        int openCounter = 0;

        //gets all valid moves
        for(int i = 0; i < 9; i++)
        {
                if(board[i] == 0)
                {
                        openSpots[openCounter] = i;
                        openCounter++;
                }
        }

        //always make the best move on hard mode
        if(diff == 3)
        {
                moveSelection = bestRobotMove(board);
        }
        //50/50 for best move or random move on medium mode
        else if(diff == 2)
        {
                if(random(1))
                        moveSelection = bestRobotMove(board);
                else
                        moveSelection = openSpots[random(openCounter-1)];
        }
        //always make a random move on easy mode
        else if(diff == 1)
        {
                moveSelection = openSpots[random(openCounter-1)];
        }

        return moveSelection;
}

//ROBOT MOVE FUNCTIONS

//brings the two arms to the bottom left most position relative to the robot
void trackReset()
{
        bool xReset = true;
        bool yReset = true;
        nMotorEncoder(motorA) = 0;
        nMotorEncoder(motorC) = 0;
        motor[motorA] = 25;
        motor[motorC] = 20;
        time1[T1] = 0;
        wait1Msec(200);

        while(xReset || yReset)
        {
                //stop resetting the x-arm if there is resistance
                if(abs(nMotorEncoder(motorA)/(time1(T1)/100)) < 9)
                {
                        xReset = false;
                        motor[motorA] = 0;
                }
                //stop resetting the y-arm if there is resistance
                if(abs(nMotorEncoder(motorC)/(time1(T1)/100)) < 14)
                {
                        yReset = false;
                        motor[motorC] = 0;
                }
                time1[T1] = 0;
                nMotorEncoder(motorA) = 0;
                nMotorEncoder(motorC) = 0;
                wait1Msec(100);
        }

        //zero the x and y arm positions
        nMotorEncoder(motorA) = 0;
        nMotorEncoder(motorC) = 0;
}

//brings the pen to its highest position
void penReset()
{
        nMotorEncoder(motorB) = 0;
        motor[motorB] = -30;
        time1[T2] = 0;
        wait1Msec(200);
        //stop resetting the pen if there is resistance
        while(abs(nMotorEncoder(motorB)/(time1(T2)/100)) > 7)
        {}
        motor[motorB] = 0;
        //zero the pen position
        nMotorEncoder(motorB) = 0;
}

//moves a given motor to a given distance at a given speed
void moveMotor(tMotor motorPort, int distance, int speed)
{
        motor[motorPort] = speed;
        while(abs(nMotorEncoder[motorPort]) < distance)
        {}
        motor[motorPort] = 0;
}


void movePenSense(int spot, bool isPen)
{
        /*
        Configurable distances for rows/columns, we are using 300/300 encoder
        Square
        */
        const int PLAYING_LENGTH = 300;
        int distBetweenCenters = 0;
        int distToCenter = 0;
        int firstColumnDist = 0;
        int secondColumnDist = 0;
        int thirdColumnDist = 0;
        int firstRowDist = 0;
        int secondRowDist = 0;
        int thirdRowDist = 0;

        //calculate the distance between the center of each spot
        distBetweenCenters = PLAYING_LENGTH / 3;
        //Calculate distance from an outside line to center of box
        distToCenter = PLAYING_LENGTH / 6;

        //Determine distances to each column/row starting with top left as 0
        firstColumnDist = distToCenter;
        secondColumnDist = distToCenter + distBetweenCenters;
        thirdColumnDist = distToCenter + 2*(distBetweenCenters);
        firstRowDist = distToCenter + 2*(distBetweenCenters);
        secondRowDist = distToCenter + distBetweenCenters;
        thirdRowDist = distToCenter;

        /*
        Create two parallel arrays, for column dist/row dist
        ie spot 1 would have second_column_dist and first_row_dist
        */
        int moveValuesColumn[9] = {
                firstColumnDist,
                secondColumnDist,
                thirdColumnDist,
                firstColumnDist,
                secondColumnDist,
                thirdColumnDist,
                firstColumnDist,
                secondColumnDist,
                thirdColumnDist
        };
        int moveValuesRow[9] = {
                firstRowDist,
                firstRowDist,
                firstRowDist,
                secondRowDist,
                secondRowDist,
                secondRowDist,
                thirdRowDist,
                thirdRowDist,
                thirdRowDist
        };
        //Adjustment distance for the light sensor
        const int HORZ_ADJUSTMENT = 170;
        const int VERT_ADJUSTMENT = 0;

        if(isPen)
        {
                //Move to column first
                moveMotor(motorC, moveValuesColumn[spot], -20);
                //Move to row second
                moveMotor(motorA, moveValuesRow[spot], -20);
        }
        else
```

```
        {
                moveMotor(motorC, moveValuesColumn[spot] + HORZ_ADJUSTMENT, -20);
                moveMotor(motorA, moveValuesRow[spot] + VERT_ADJUSTMENT, -20);
        }
}

void penDown()
{
        const int PEN_DIST = 30;
        moveMotor(motorB, PEN_DIST, 40);
}

/*
This function draws the playing board for a tic-tac-toe game.
A constant playing length in motor encoder clicks is assumed
to be 300.
*/
void drawBoard()
{
        penReset();
        trackReset();
        const int PLAYING_LENGTH = 300;
        int distBetweenLines = 0;
        distBetweenLines = PLAYING_LENGTH / 3;

        // move to first column line and then draw upwards
        moveMotor(motorC, distBetweenLines, -20);
        penDown();
        wait1Msec(750);
        moveMotor(motorA, PLAYING_LENGTH, -20);
        penReset();
        trackReset();

        //Move to second column and then draw upwards
        moveMotor(motorC, 2*distBetweenLines, -20);
        penDown();
        wait1Msec(750);
        moveMotor(motorA, PLAYING_LENGTH, -20);
        penReset();
        trackReset();

        //Move to first row, draw across column
        moveMotor(motorA, distBetweenLines, -20);
        penDown();
        wait1Msec(750);
        moveMotor(motorC, PLAYING_LENGTH, -20);
        penReset();
        trackReset();

        //Move to second row, draw across column
        moveMotor(motorA, 2*distBetweenLines, -20);
        penDown();
        wait1Msec(750);
        moveMotor(motorC, PLAYING_LENGTH, -20);
        penReset();
        trackReset();
}

task main()
{
        //initilizes sensors
        SensorType[S1] = sensorEV3_Color;
        wait1Msec(50);
        SensorMode[S1] = modeEV3Color_Reflected;
        wait1Msec(100);
        SensorType[S4] = sensorEV3_Touch;
        wait1Msec(100);


        bool loopRun = true;

        while(loopRun)
        {
                //initilizes the board
                int board[9] = {0,0,0,
                               0,0,0,
                               0,0,0};
                int difficulty = GUI();
                if(difficulty == -1)
                        loopRun = false;
                else
                        drawBoard();

                //loops through the turns
                while(!isTerminal(board) && loopRun)
                {
                        int changeCount = 0;
                        //player move
                        do
                                {
                                trackReset();
                                displayTextLine(8, "Please Make Your Move");
                                displayTextLine(9, "Then Press the Button");
                                while(SensorValue[S4] == 0)
                                {}
                                while(SensorValue[S4] == 1)
                                {}
                                eraseDisplay();
                                for(int i = 0; i < 9; i++)
                                {
                                        trackReset();
                                        if(board[i] == 0)
                                        {
                                                movePenSense(i, false);
                                                wait1Msec(500);
                                                if(SensorValue[S1] < 25)
                                                {
                                                        board[i] = PLAYER;
                                                        changeCount++;
                                                }
```

```
                    }
                }
            }
            while(changeCount < 1 && loopRun);

            //exits if the player has made too many moves
            if(changeCount > 1)
            {
                displayTextLine(8, "Too Many Moves!");
                displayTextLine(9, "The game will now restart");
                wait1Msec(5000);
                loopRun = false;
            }

            //robot move
            trackReset();
            if(!isTerminal(board) && loopRun)
            {
                int bestMove = robotMove(board, difficulty);
                writeDebugStreamLine("%d", bestMove);
                movePenSense(bestMove, true);
                board[bestMove] = ROBOT;
                penDown();
                wait1Msec(1000);
                penReset();

            }
        }
        //updates highscore file
        writeToHighscore(checkForWin(board, PLAYER), difficulty);
    }
}
```

Appendix B: Score File Creation/Reset Code (createEmptyFileTTT)

```
#include "PC_FileIO.c"


task main()
{
	TFileHandle output;
	bool isOpen = openWritePC(output, "TTT_High_Scores.txt");
	if (!isOpen)
	{
		playSound(soundException);
		displayString(5, "FAILED TO OPEN FILE");
		wait1Msec(1000);
	}
	for (int i = 0; i < 4; i ++)
	{
		writeLongPC(output, 0);
		writeEndlPC(output);
	}
	closeFilePC(output);
}
```

Appendix C: Flowcharts

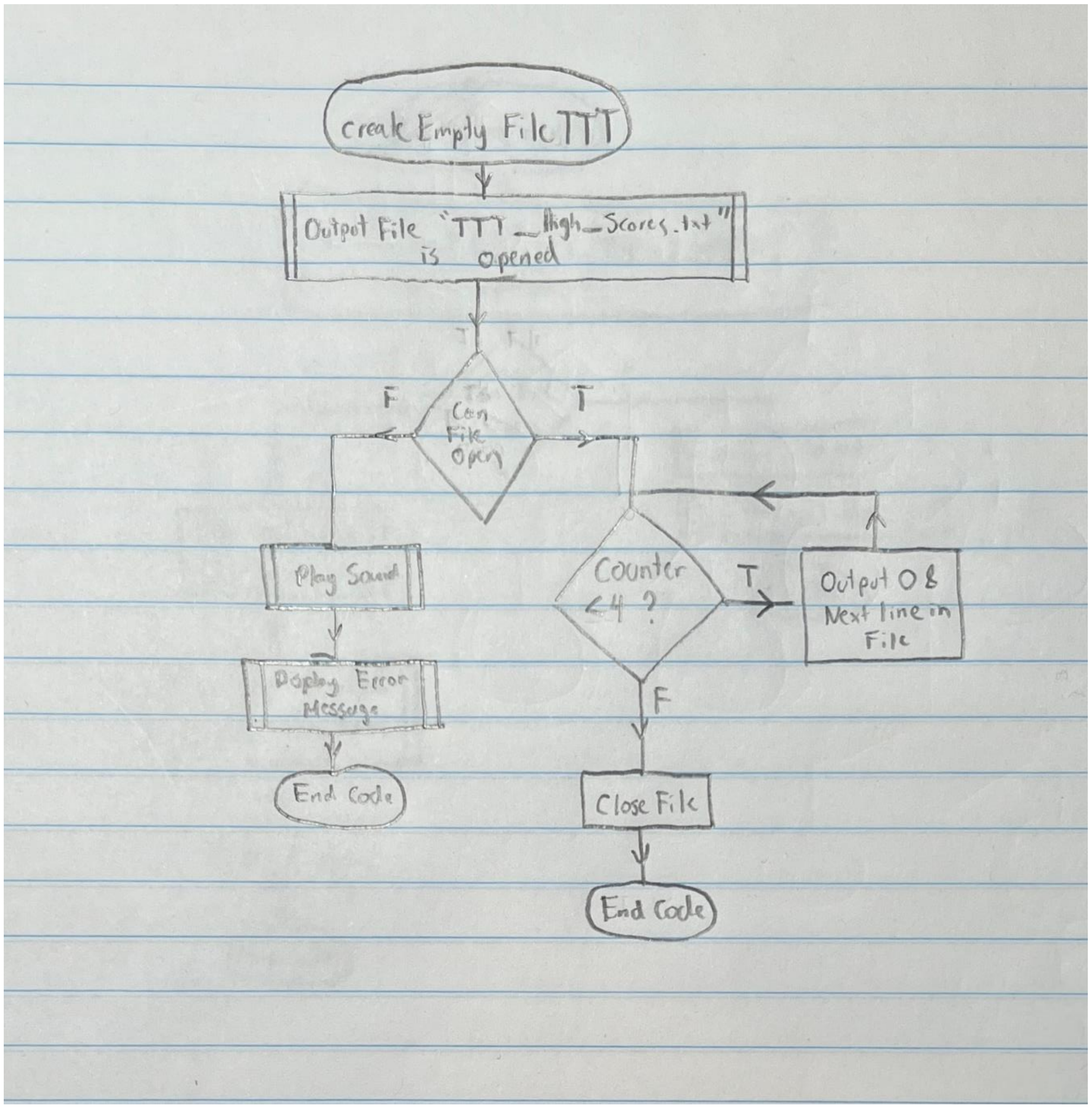


Figure 16. createEmptyFileTTT Flowchart

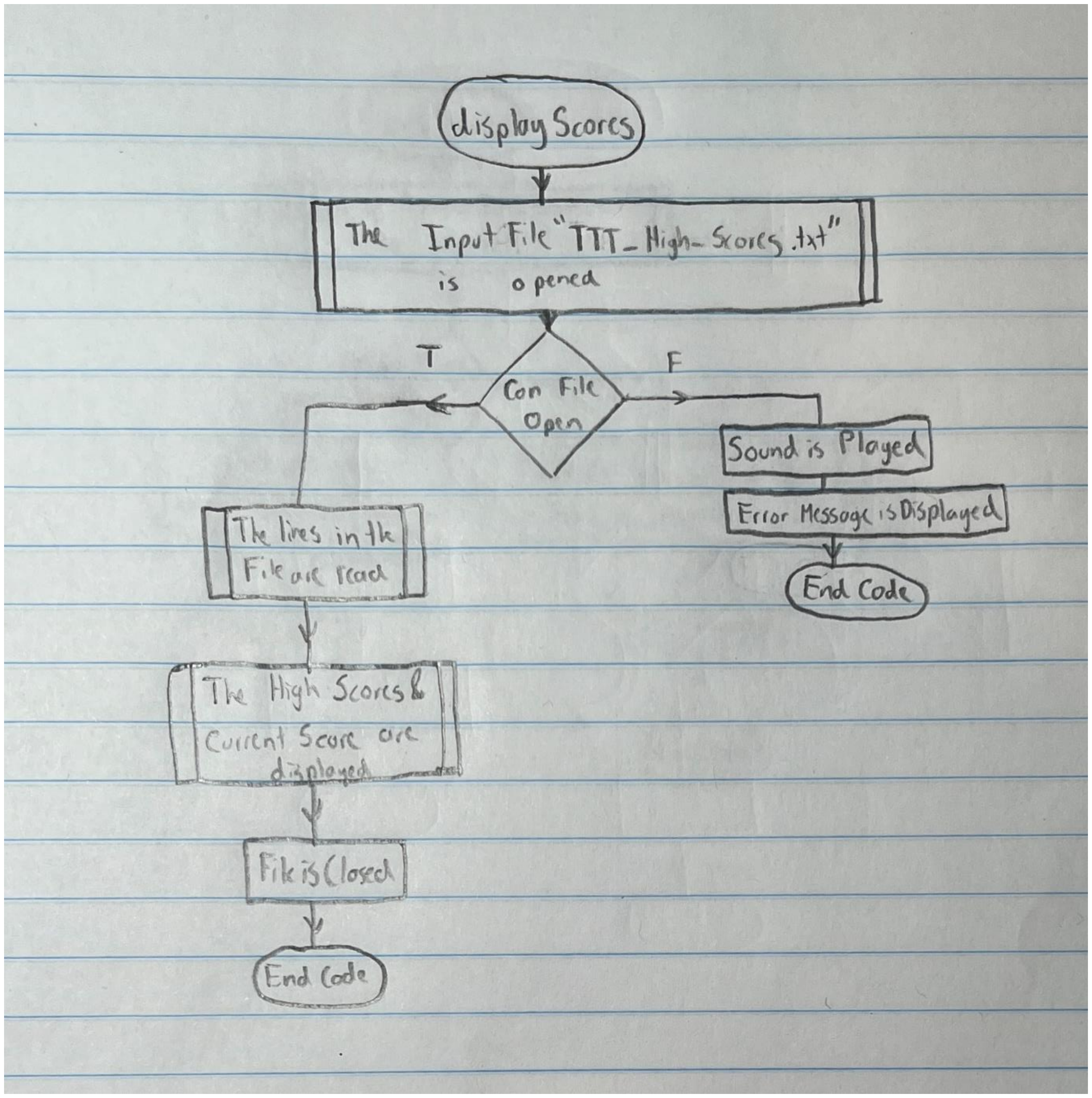

displayScores
The Input File "TTT_High_Scores.txt" is opened
Can File Open
T
F
Sound is Played
Error Message is Displayed
End Code
The lines in the File are read
The High Scores & Current Score are displayed
File is Closed
End Code

Figure 17. displayScores Flowchart

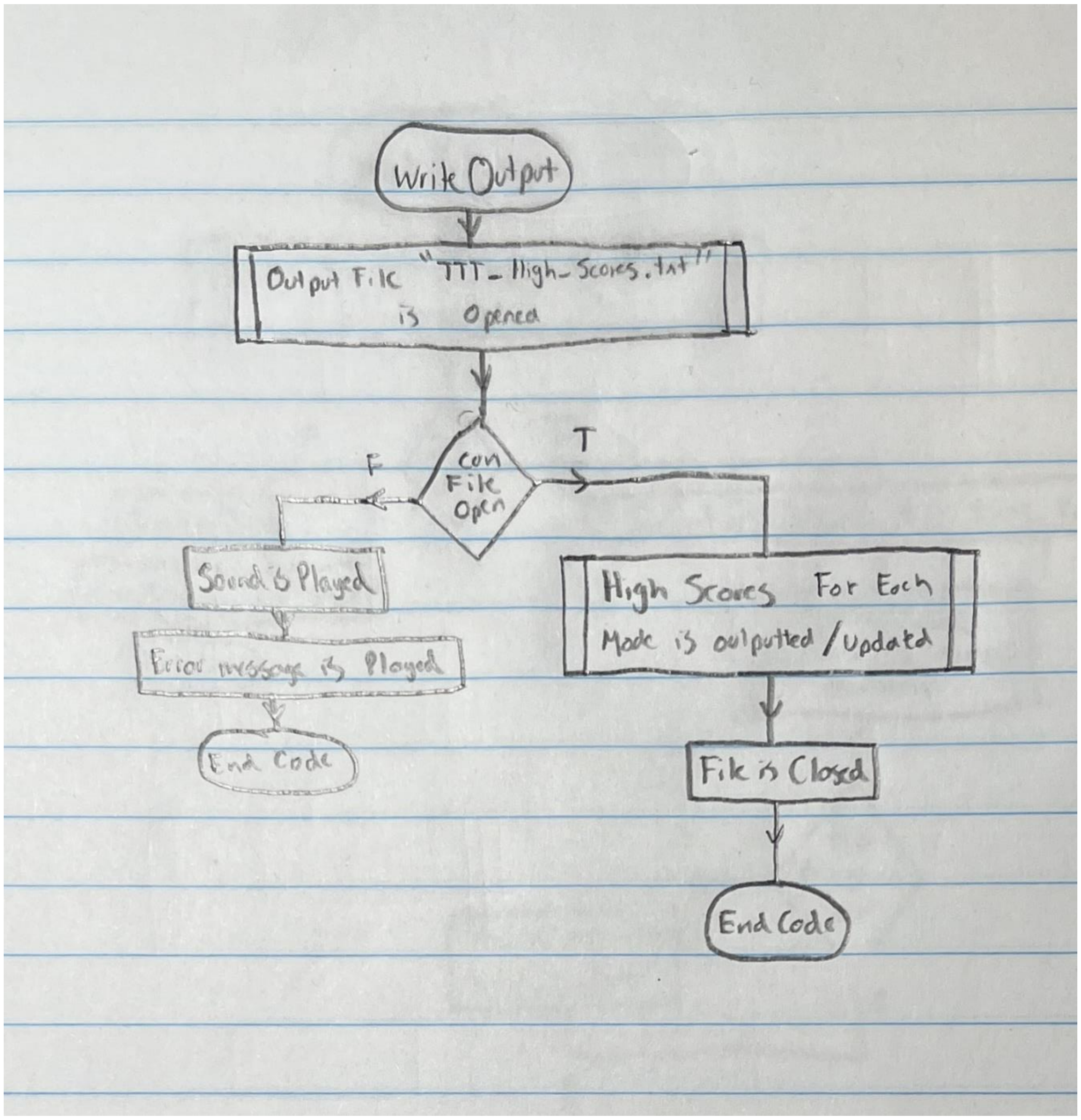
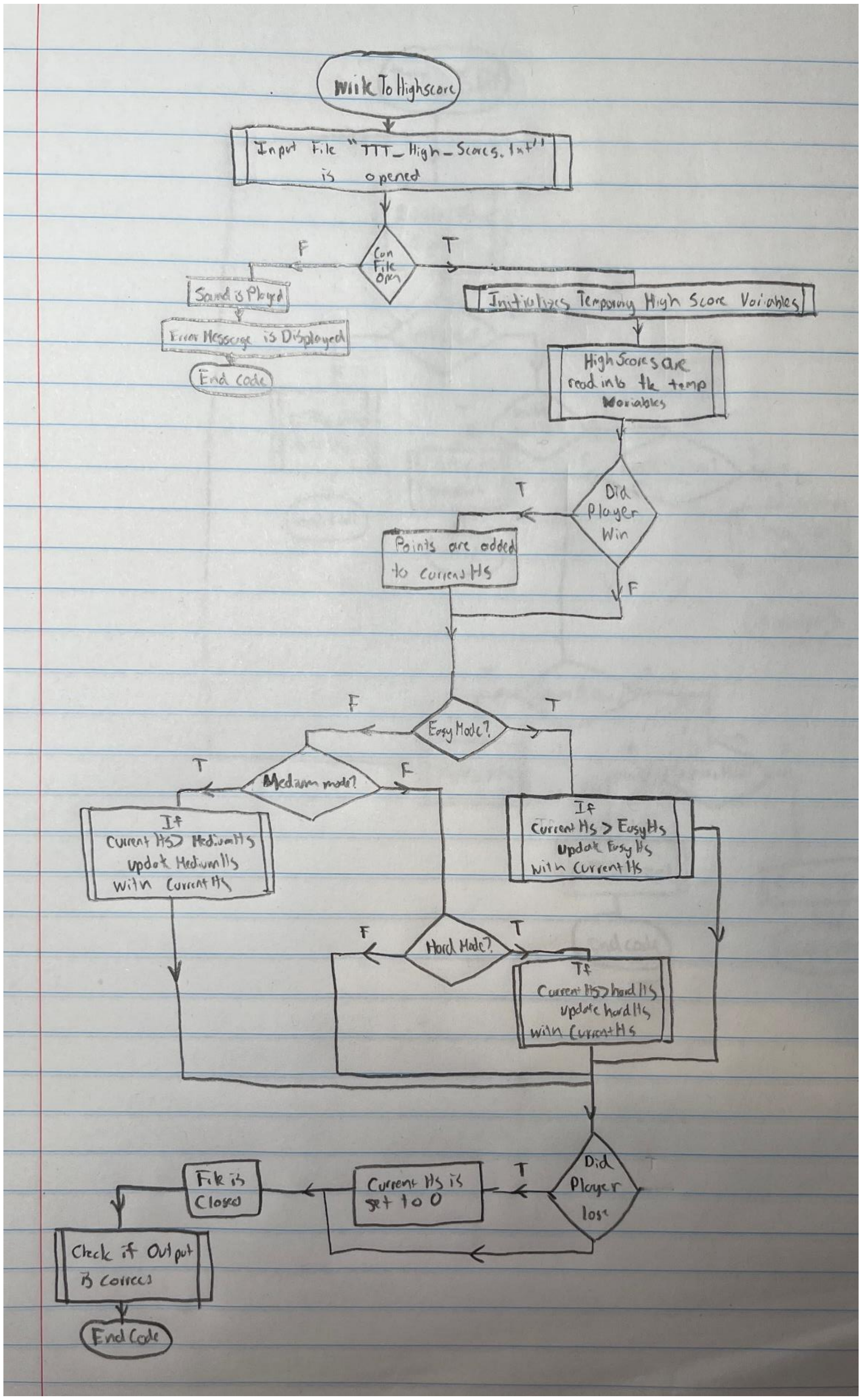


Figure 18. writeOutput Flowchart



write To Highscore
Input File "TTT_High_Scores.txt" is opened
Can File Open
F
T
Sound is Played
Error Message is Displayed
End Code
Initializes Temporary High Score Variables
High Scores are read into the temp Variables
Did Player Win
T
F
Points are added to Current HS
Easy Mode?
F
T
Medium mode?
T
F
If Current HS > MediumHS update MediumHS with Current HS
If Current HS > EasyHS Update Easy HS with Current HS
Hard Mode?
F
T
If Current HS > hard HS update hard HS with Current HS
Did Player lost
T
Current HS is set to 0
File is Closed
Check if Output is Correct
End Code

Figure 19. writeToHighscore Flowchart

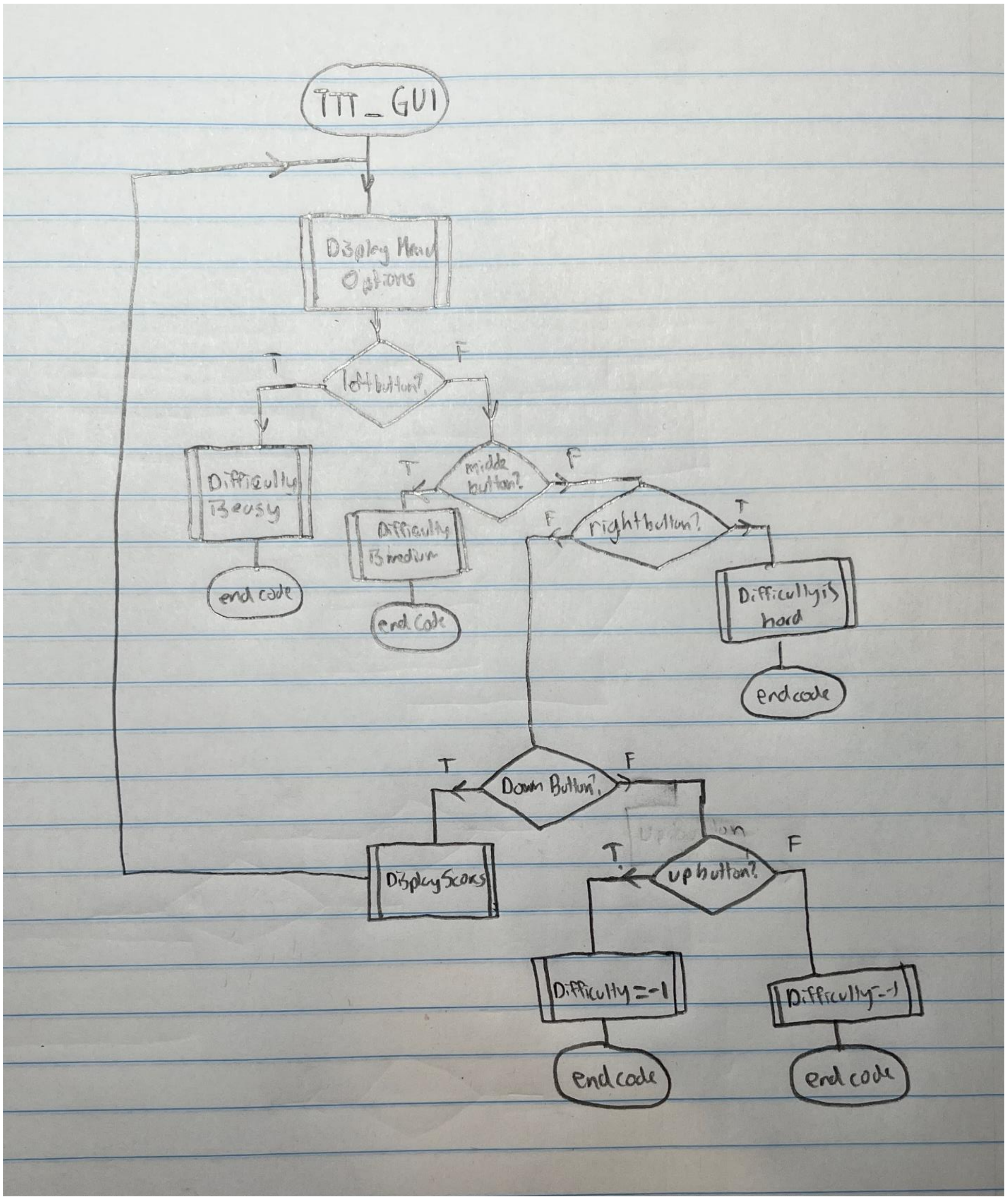


Figure 20. GUI Flowchart

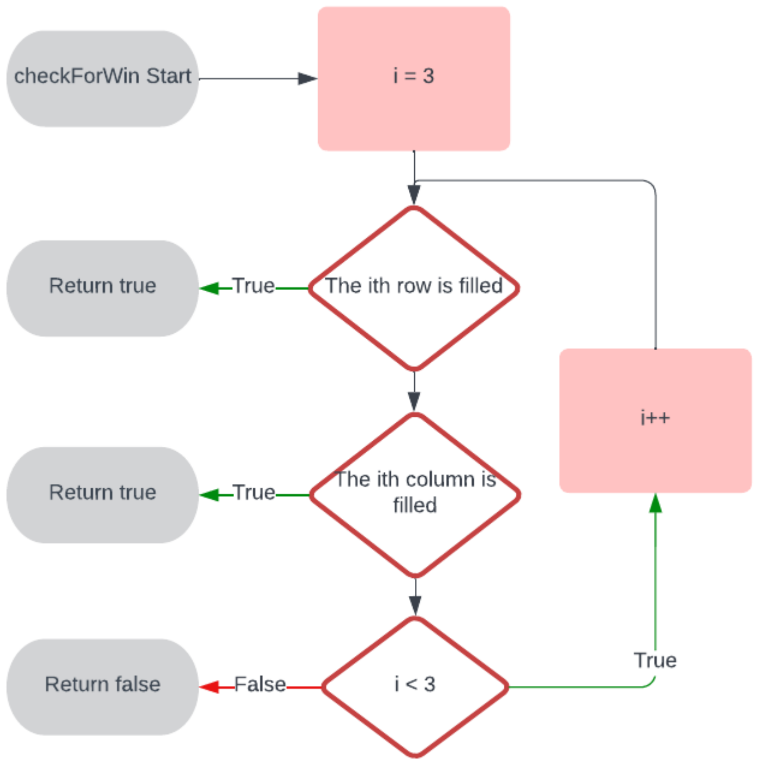


Figure 21. checkForWin Flowchart

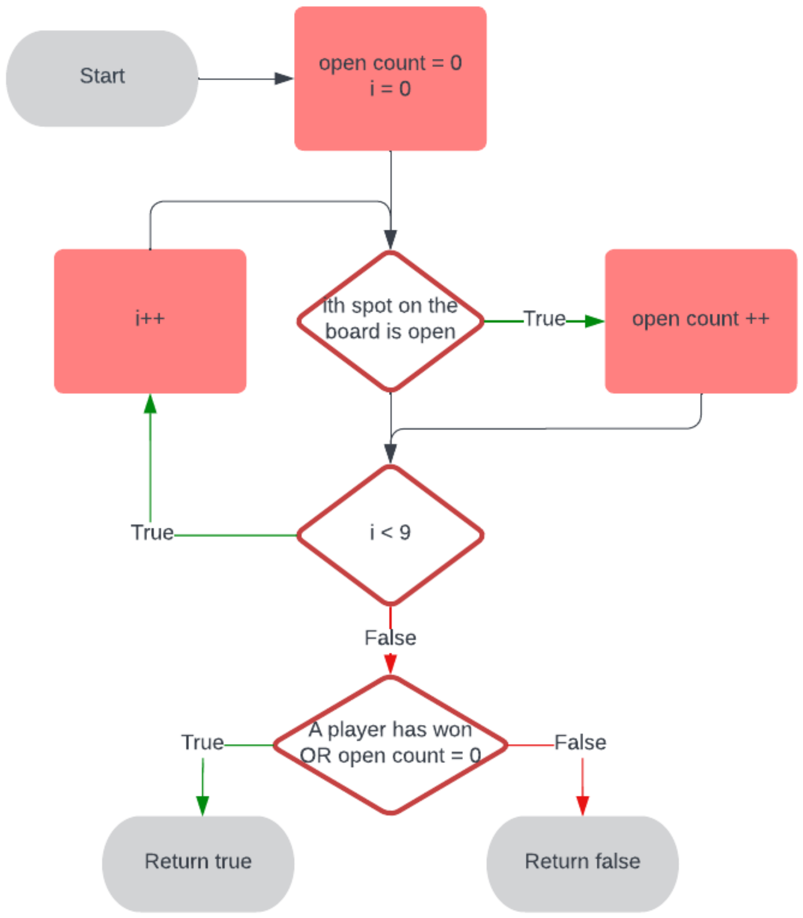


Figure 22. isTerminal Flowchart

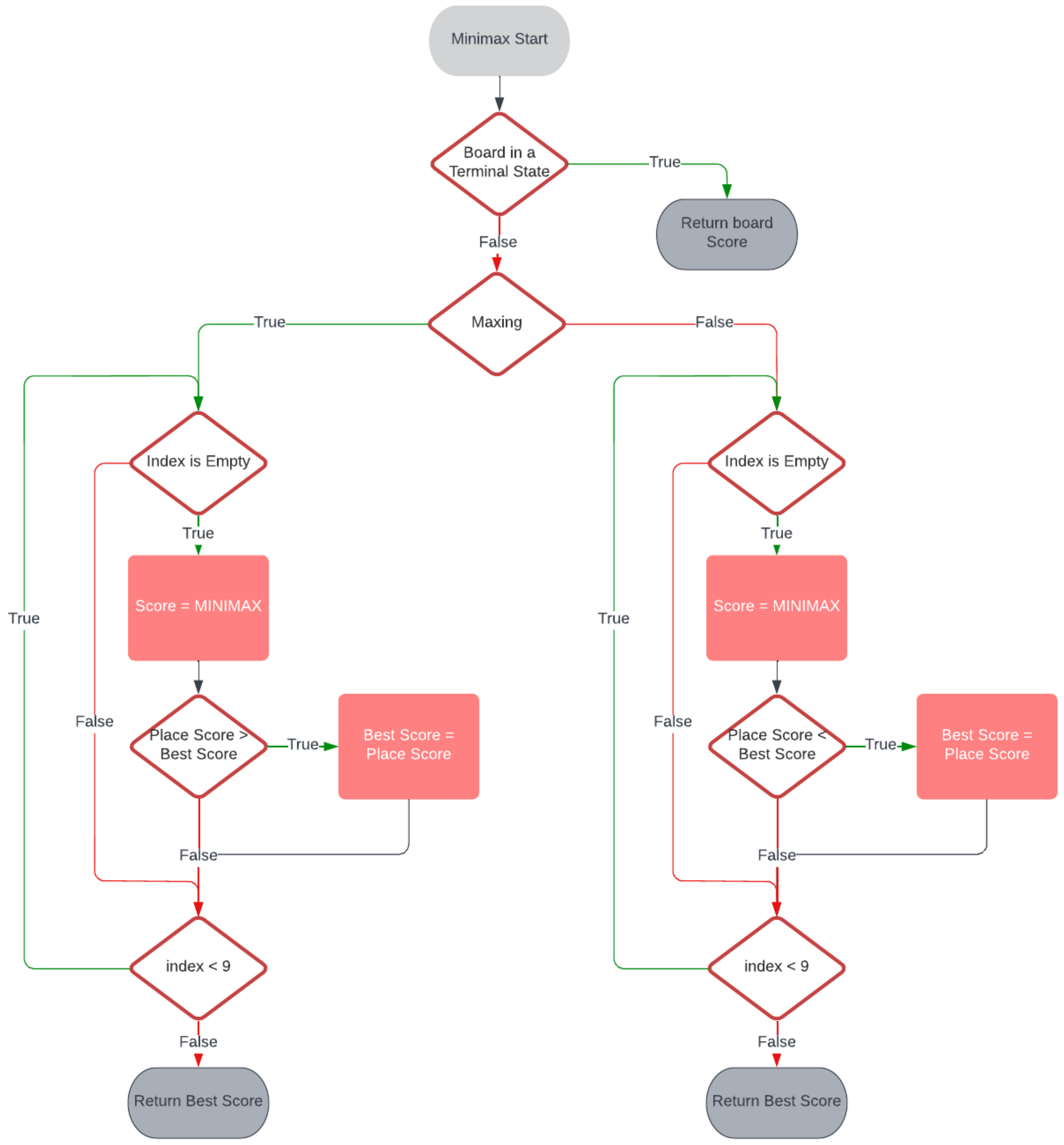


Figure 23. minimax Flowchart

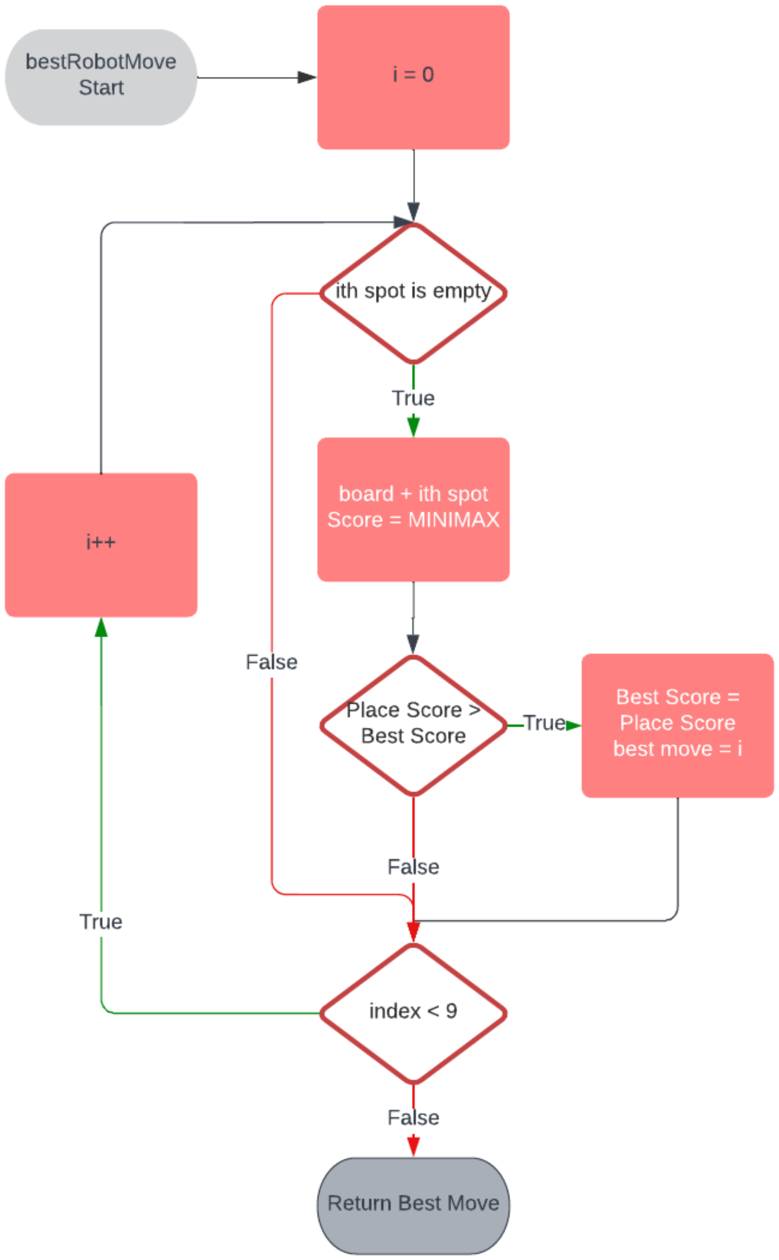


Figure 24. bestRobotMove Flowchart

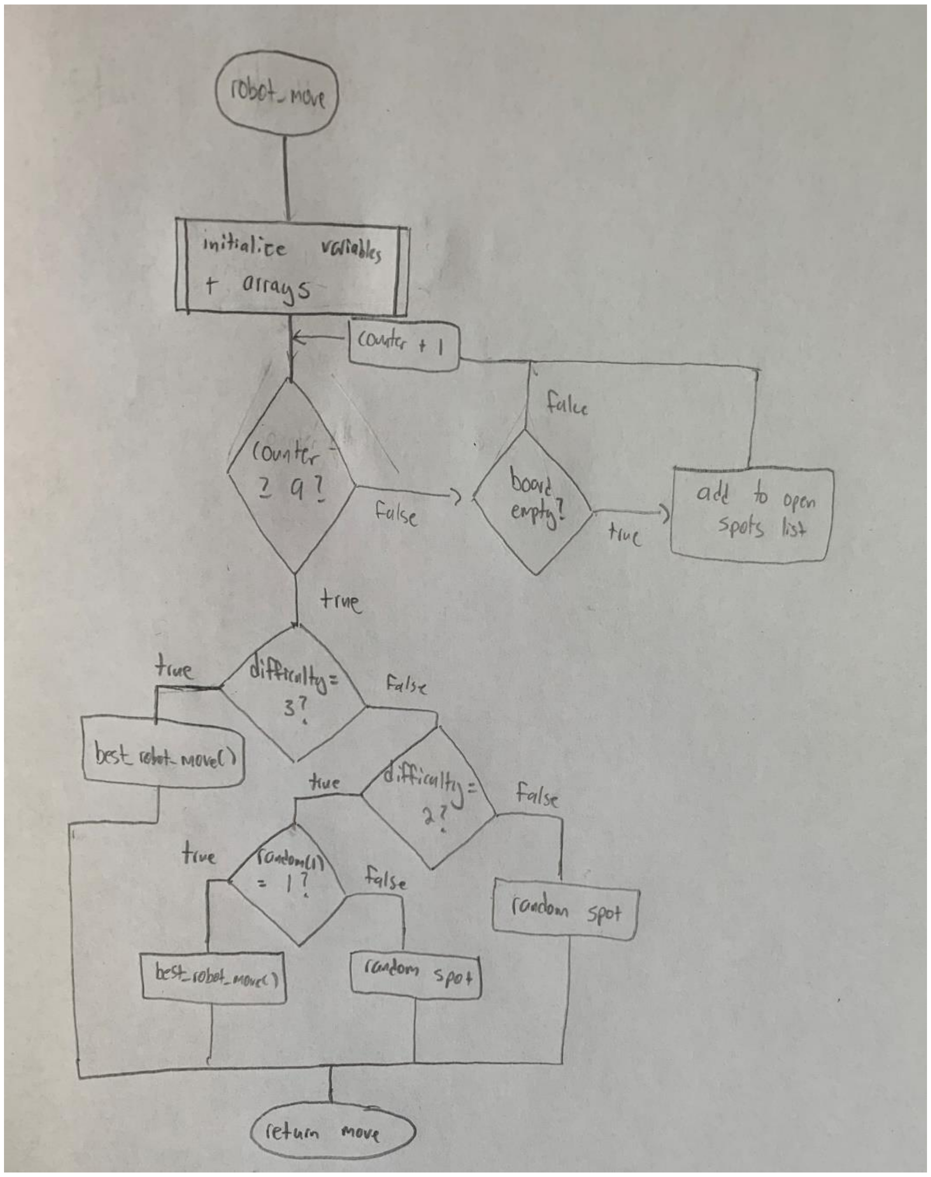


Figure 25. robot_move Flowchart

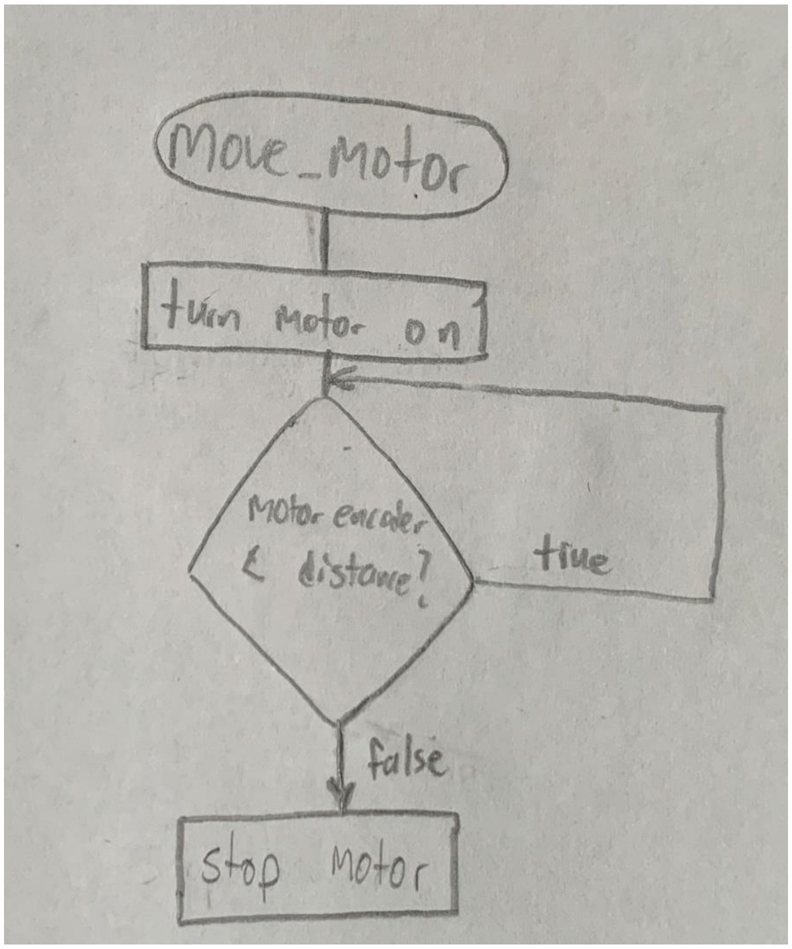


Figure 26. moveMotor Flowchart

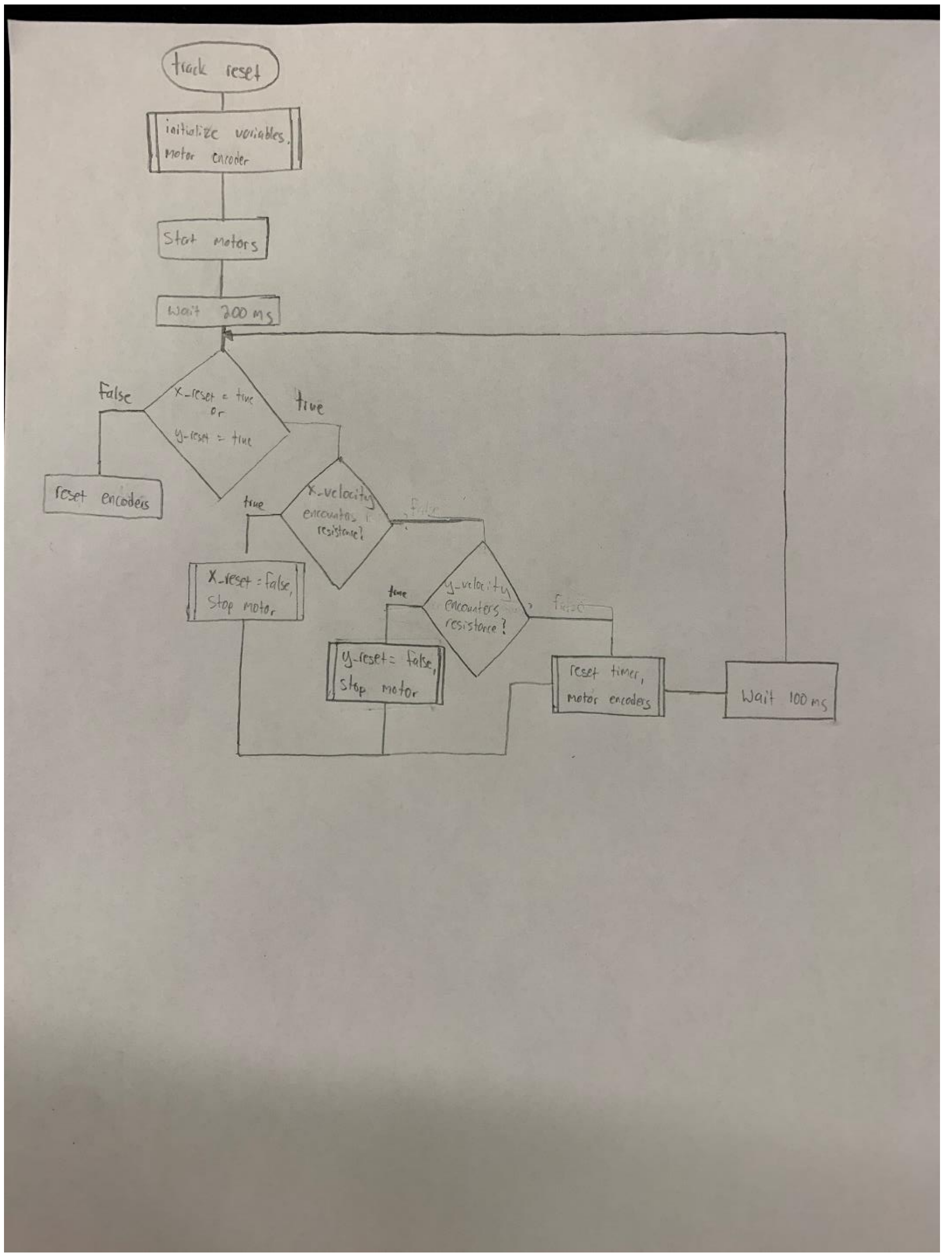

track reset
initialize variables, motor encoder
Start motors
Wait 200 ms
X_reset = true or y_reset = true
False
true
reset encoders
X_velocity encounters resistance?
true
False
X_reset = false, Stop motor
y_velocity encounters resistance?
true
false
y_reset = false, Stop motor
reset timer, motor encoders
Wait 100 ms

Figure 27. trackReset Flowchart

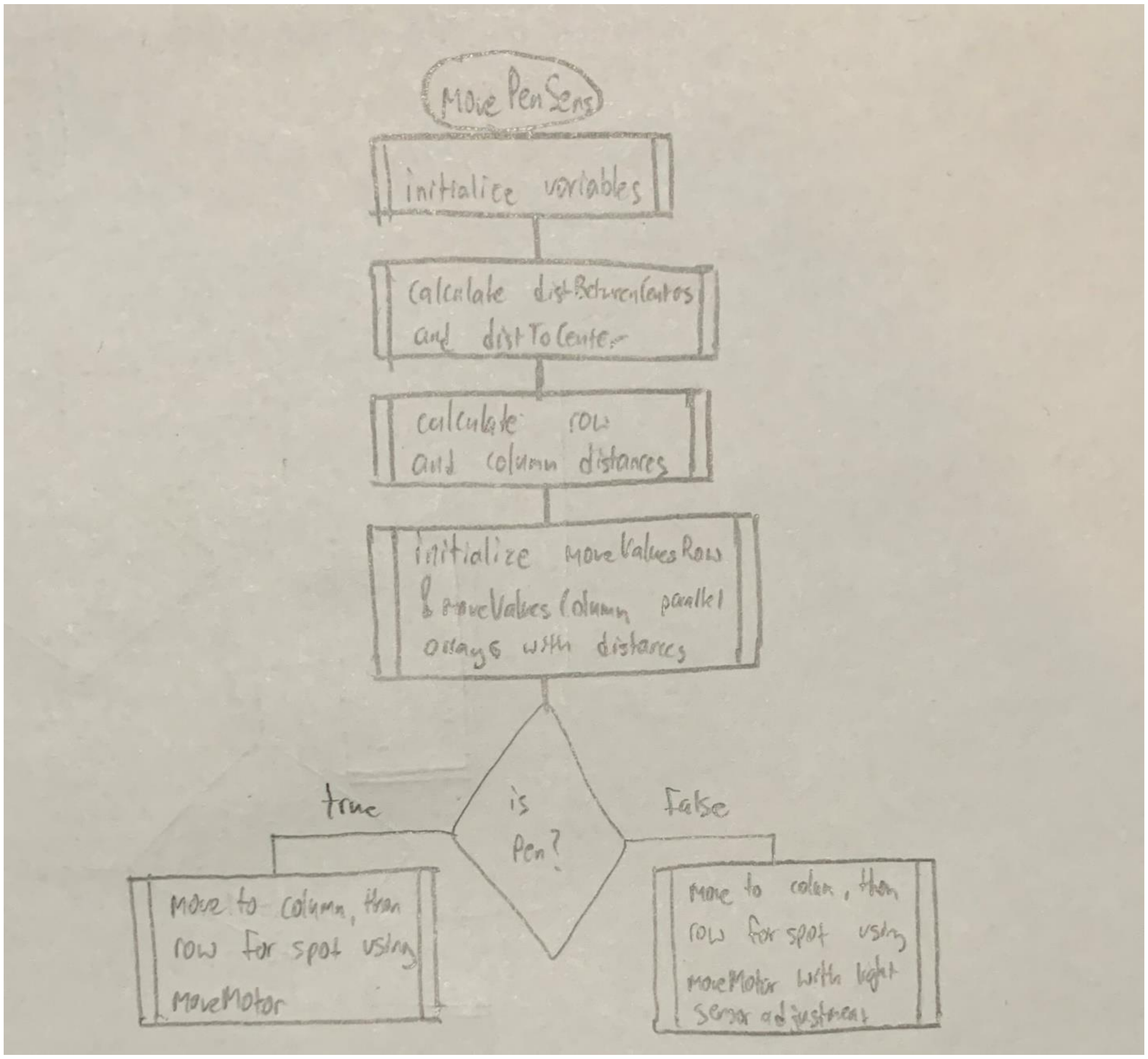

movePenSens
initialice variables
Calculate distBetweenCenters and distToCenter
calculate row and column distances
initialize moveValuesRow & moveValuesColumn parallel arrays with distances
is Pen?
true
False
move to column, then row for spot using moveMotor
move to colum, then row for spot using moveMotor with light sensor adjustment

Figure 28. movePenSens Flowchart

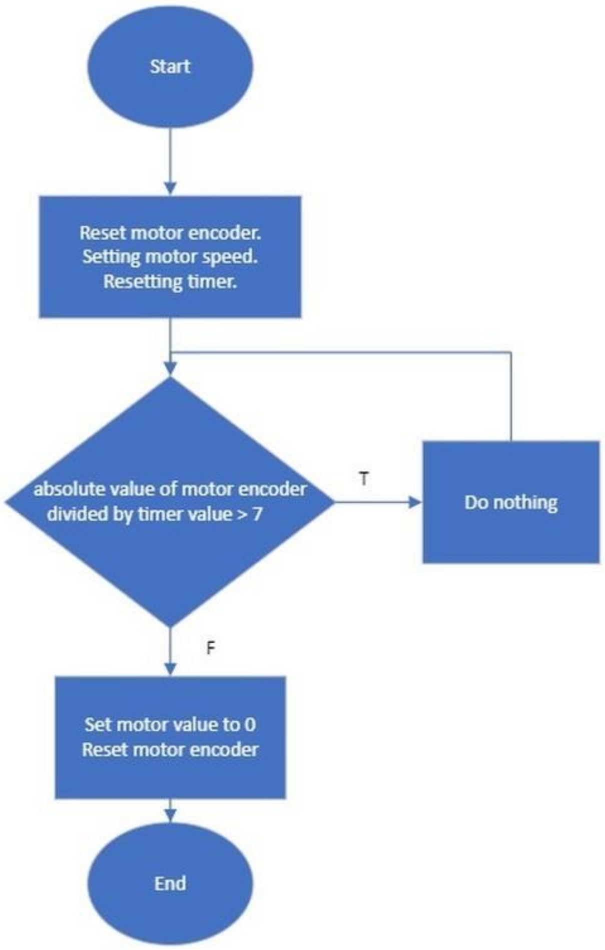


Figure 29. penReset Flowchart

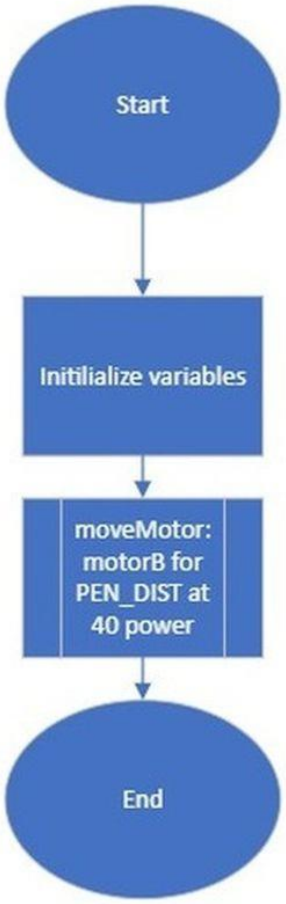


Figure 30. penDown FLowchart

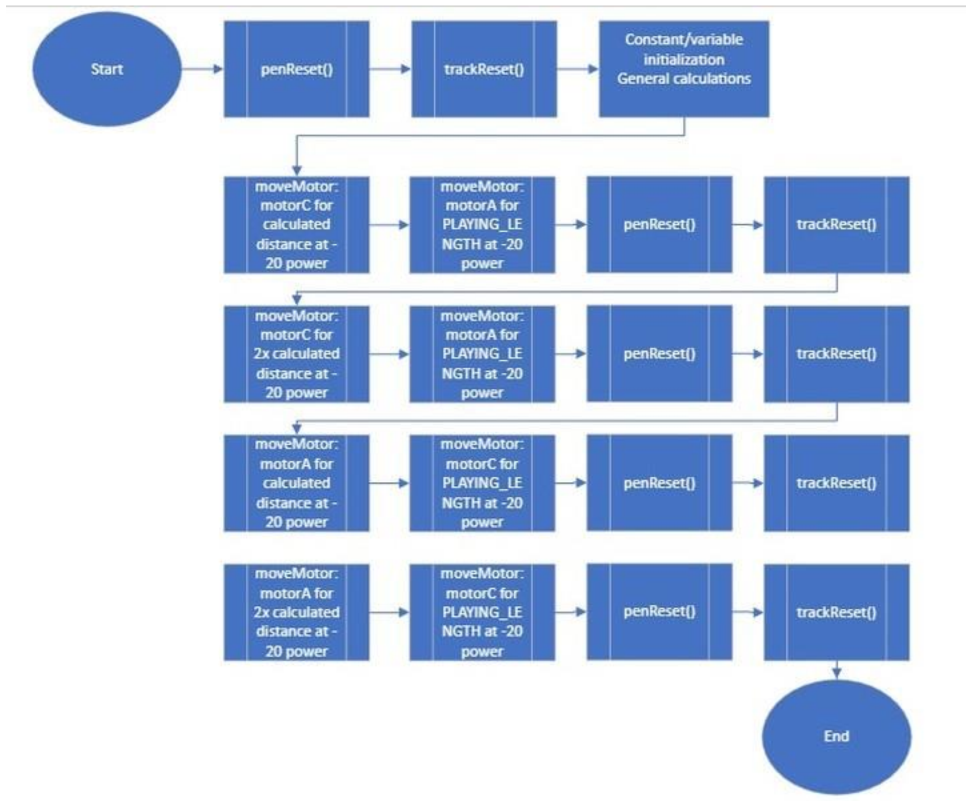

Start
penReset()
trackReset()
Constant/variable initialization General calculations
moveMotor: motorC for calculated distance at -20 power
moveMotor: motorA for PLAYING_LENGTH at -20 power
penReset()
trackReset()
moveMotor: motorC for 2x calculated distance at -20 power
moveMotor: motorA for PLAYING_LENGTH at -20 power
penReset()
trackReset()
moveMotor: motorA for calculated distance at -20 power
moveMotor: motorC for PLAYING_LENGTH at -20 power
penReset()
trackReset()
moveMotor: motorA for 2x calculated distance at -20 power
moveMotor: motorC for PLAYING_LENGTH at -20 power
penReset()
trackReset()
End

Figure 31. drawBoard Flowchart