Checkers – The Chess Playing Robot

Group 451

Adam Barroso #20834093

Cameron Kinsella #20820478

Allen Zhang #20846121

MTE 100

December 3, 2019

Summary

Checkers – “The Chess Playing Robot” is a robot which is great for lonely chess lovers and competitive chess players. This robot is programmed with a chess AI which allows it to see moves ahead and change the difficulty level of each game. The robot must be able to accurately scan and pick up pieces while still allowing for the user to easily move their own pieces too.

On the initial start-up of Checkers will ask the user to select a time duration for the game as well as a difficulty setting. After that Checkers will do an initial scan of the chess pieces to create a digital map of the board. During Checkers’ normal operation, after the user has done a move, Checkers will scan the board to detect what has been moved. Next it will check if a win condition or a stalemate has occurred. If not, then it will figure out the best move to make and play a piece accordingly. If a win condition or stalemate occurs then Checkers will output a statement about the results of the match, the arm will go back to origin and the C++ code will shut down.

Checkers has 2 primary constraint. One is that it must not detect empty spaces as an initialized chess piece. The other constraint is that the bottom of the claw must not be less than 15 cm above the board to ensure that it does not interfere with the user’s hand when moving pieces.

The original mechanical design had issues with the wooden dowels that were used to move the colour sensor movement block, so colour sensing from the bottom of the board to become impractical. To fix this, the colour sensor was moved to the top, the bottom movement block was removed, and the piece colours were moved to the piece tops. This fixed the mechanical issue but caused the colour sensing to be more susceptible to error. Additionally, cable management proved to be a challenge, but was able to be overcome.

Several test programs were created to validate each section of the overall program and ensure that it works as intended. There were no major changes to the software design, but a significant setback occurred when looking for a method to communicate between Robot C and C++.

The main mechanical improvements that Checkers would benefit from would be reducing the overall size, reducing the weight of all components in the movement block, and using more reliable materials. The main software improvements would be to conduct more thorough integration tests, and to modify the way file IO functions to prevent overlap with the C++ code writing to the file.

Acknowledgements

We would like to extend our thanks to the MME Clinic staff for allowing us to use the 3D printers in WATiMake for the manufacturing of our chess pieces.

Contents

Table of Figures

Table of Tables

Introduction

Our robot "Checkers" aims to solve the problem of chess relying on having two people to play. We aimed for checkers to be a chess playing robot which was able to use a real board and real pieces to play against a real person.

This would have been ideal for the elderly due to their struggles with computers and new technology, however Checkers should be simple and very intuitive. Also, the elderly are often quite lonely with their kids all grown up and moved out [1]. This robot can also help keep the elderly mentally active. With Canada's aging population this robot would be a great asset.

Commented [CK1]: I know it's nitpicky, but they might take a mark for not citing that. Maybe add citation or reword it?

Some other people who would benefit from a chess robot are competitive chess players. Checkers would allow them the freedom of honing their skills whenever they want without the need of others and with a real chess feel. Plus, there are different difficulty settings so as the player gets better the robot does as well, this way the player will always feel challenged.

This was the goal of our robot however; we were unable to reach this goal within the allowed time frame. What our robot could do was it was able to pick up pieces as well as scan the board to locate all the pieces. Our robot came close to being able to solve our problems however we were not able to implement the actual chess algorithm into our robot.

Background[1]

The game of chess is a two player turn based game. Each turn a player moves a piece and if your piece lands in the same spot as your opponent's piece then you "kill" their piece and take it off the board. Each piece type has a specific method of moving. The objective of the game is to "kill" the opponent's king piece, meaning to get them in "checkmate". "Checkmate" is when there is no move that a player can make that could prevent their king from being killed in their opponent's next move. "Check" is when the king can be taken in the opponent's next move, but there is a move available that can prevent it from happening.

There are also times when no one wins, called a "stalemate". This occurs when there is no possible way for either king to be put in checkmate. Another stalemate condition is if your opponent's king is not in check and any move, they make will put them in check. Additionally, a typical timed game of chess will end once one of the players run out of time, causing said player to lose.

Scope

Tasks

Checkers' primary goal is to pick up and place chess pieces to specific locations based on what the A.I. determines. This is accomplished through the following tasks:

[1] Retrieved from [1]

1. Setting game mode and difficulty settings through user input on the EV3 buttons
2. Conducting an initial scan of the chess board, using the colour sensor to scan the colour on each of the player's chess pieces and storing their initialized colour values
3. Beginning and displaying a timer, starting the player's turn. Begin waiting for touch sensor to be pressed.
4. After detecting the touch sensor being pressed (indicating the end of the player's turn), pausing the timer, conducting another scan of the board, comparing each tile to each initialized colour value to creating a digital map of the board with the new locations of the player's pieces
5. Sending the digital map of the board to the backend C# AI which determines Checkers' best move (difficulty dependant) and the claw movement details and sends it back to the Robot C program
6. Based on the received movement details, moving the claw to the desired location, picking it up, moving it to the desired location, placing it, then returning the claw to its default position (the front right corner from the player's perspective)
7. Resuming the timer and repeating tasks 3-6 until an end condition is met
8. Displaying a message depending on end condition
9. Ending the Robot C and C# programs

Measurements and Detections

On task 1, Checkers receives input from the player through, waiting to detect when one of the valid EV3 buttons are pressed.

The colour sensor is used to detect the red, blue, and green values from each tile on the board. This is used for initializing the colour values of the player's chess pieces and comparing the values to said initialized values for task 2 and 4 respectively.

On task 5 and 6, file IO is used by the Robot C program to detect changes in "chessmove.txt" for communicating between the C# and Robot C program. For more details, see Software Design and Implementation.

The touch sensor is used to detect when the player presses it, indicating the end of their turn.

Environment Interaction

Checkers uses 4 motors to physically interact with the environment. Two motors are used for lateral movements of the claw, one for X movement and one for Y movement. Another motor is used for vertical movement of the claw to allow the claw to reach low enough to pick up pieces. These 3 motors turn gears which are interlocked with gear racks, allowing the rotational movement to be converted to lateral movement. The final gear is directly attached to one of the gears which control the opening and closing movements of the claw.

Additionally, Checkers interacts with the player through the EV3 buttons and the touch sensor, as described in Measurements and Detections.

Task Completion and Shutdown Procedure

Other than task 1 and 3 which are recognized as complete once the buttons and touch sensor respectively are pressed, all tasks are recognized by the robot to be complete once the function being called for said task is complete.

A game is considered finished when the player's time runs out or when the Robot kills the King, at which point the screen will display a message gloating, or when the player kills the king, at which point the Robot will detect that it has no King and will display a sad message, accepting its defeat. If a stalemate is detected a tie message will be displayed. After the robot displays the ending message, the arm will move to its default position and both the Robot C and C# code will shut down.

Changes to Scope

The only changes to the scope were some minor modifications to the tasks. The number of options for time limits was reduced from 4 options (15 minutes, 30 minutes, 45 minutes, unlimited), to 3 options (removing the 45 minutes option). Additionally, the language of the A.I. was changed from C++ to C# for more versatility.

Criteria and Constraints

Criteria

The primary requirements which were chosen in the preliminary design report are as follows:

Table 1: List of Criteria and Justifications [1]

Criteria (from most to least important)	Justification	Method of Measurement
Must be able to pick up and place chess pieces without dropping them, tipping over other pieces, or knocking them over after placing	If pieces are knocked over, the state of the game will be ruined	Number of pieces knocked over and dropped per 25 turns.
Must be able to scan chess piece colour values without significant deviation from initialized values	If measured values vary greatly from initial values, the digital map of the board will have errors and Checkers will make illegal moves	Deviation between a scanned colour value and its initial colour value
Must be able to create a digital map of the board in under 1.5 minutes	to ensure that time between turns is reasonable for shorter game modes	Timing a scan of the board
Mechanism for picking up pieces must have minimum	Deadlines are strict and there is not much time for	Group survey ranking each design's complexity from 1-10.

complexity for quick prototyping and modifying since real-life tests may demonstrate flaws in the mechanism	troubleshooting, so reducing the risk of having problems is preferable	

Changes to Criteria

The number of tests turns for first criteria's method of measurement was reduced to allow for more reasonable impromptu testing. Also, the criteria for minimum mechanical complexity was removed since it is too vague to properly measure, and it has low importance. Additionally, the time criteria for creating digital map was increased from 1.5 minutes to 2 minutes. This is because even though the robot was capable of scanning under 1.5 minutes, slightly increasing wait times before movements would still allow time between turns to be reasonable while also allowing the colour sensor to scan each tile more accurately.

Constraints

The primary constraints on Checkers' design which were chosen in the preliminary design report are as follows:

1. Must not detect empty spaces as one of the initialized chess piece colours.
2. Bottom of claw resting position must not be under 25cm above the board to ensure that it does not impede the player from moving chess pieces with their hand.

Changes to Constraints

The minimum claw resting position was reduced from 25cm to 15cm. This allowed the overall size Checkers to be reduced while still achieving the goal of this constraint.

Criteria and Constraints Reflection

The most useful of the criteria and constraints was ensuring that the scanned colour values did not deviate significantly from the initialized values. The focus on that aspect resulted in more reliable scans and the accomplishment of constraint 2 by association.

The least useful of the criteria and constraints was ensuring that the digital map was created in under 2 minutes. The greater the effort put into accomplishing this criterion, the less accurate the scanning. It was not crucial and focusing on it only resulted in a negative impact on Checkers' reliability.

Some recommendations for changes include adding a constraint for total size since bending of long materials proved to be problematic, removing the criterion for scanning the board in under 2 minutes due to the reasons mentioned above, and removing constraint 2 since it did not prove to make a significant difference for the player.

Mechanical Design and Implementation

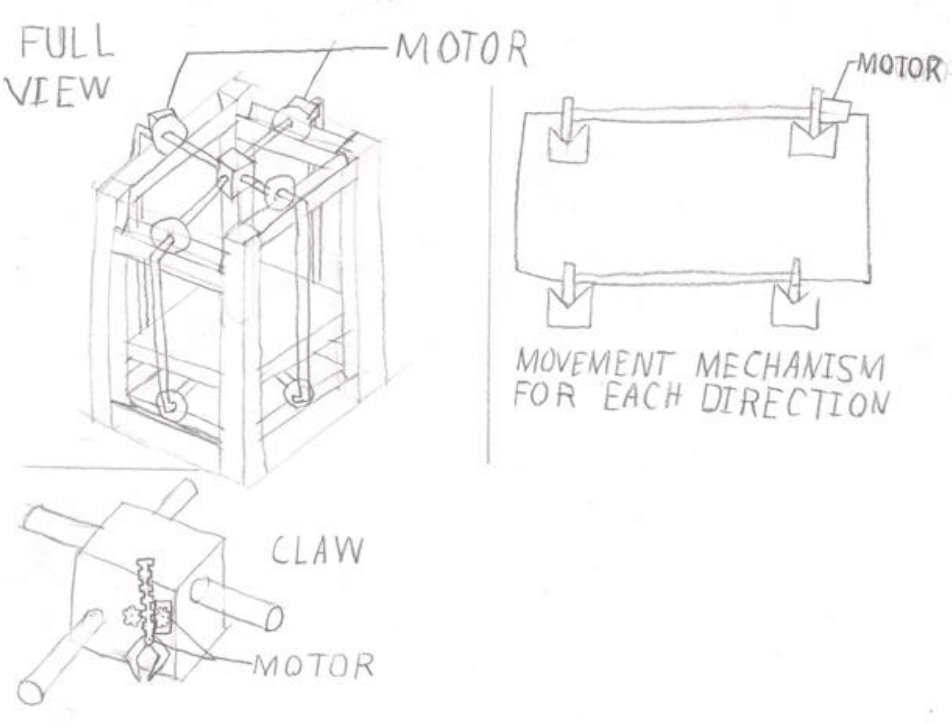


Figure 1: Initial Robot Design

This is the sketch of what we originally wanted our robot to look like. We wanted to have two planes of horizontal movement. This would have been achieved using two motors, one to move within the x axis of the top plane and the other to move within the y axis of the top plane. Then the top and bottom planes would be connected so that whatever movement would happen in the top plane would also happen in the bottom plane.

We originally had two movement blocks, one for the top plane and one for the bottom plane. These movement blocks were designed so that an attachment could be moved without the axes interfering with one another. As seen in the sketch the top plane has the movement block which hoists the claw. This claw moves vertically to pick up and put down pieces. The bottom movement block has a very similar system except that there is a stationary colour sensor rather than a claw. This colour sensor was meant to sense the bottoms of the chess pieces where the colours were going to be attached.

Unfortunately, this initial design did not work out. The main issue with the initial design was getting the top and bottom planes to move together. Rather than moving together the bottom axes would be lifted and dragged a few centimeters behind the top axes. We have hypotheses as to why this occurred. One being that the vertical dowels connecting the top and bottom planes could not fit exactly right to allow for coincident movement. Our other hypothesis is that the bottom plane did not have enough weight to make sure it was not lifted. This was a major design flaw that was not accounted for it had to be fixed for our board mapping to work. We attempted to fix this problem by adding more weights where possible and trying to secure the vertical dowels more, however neither approached worked. This resulted in us having to use a different design.

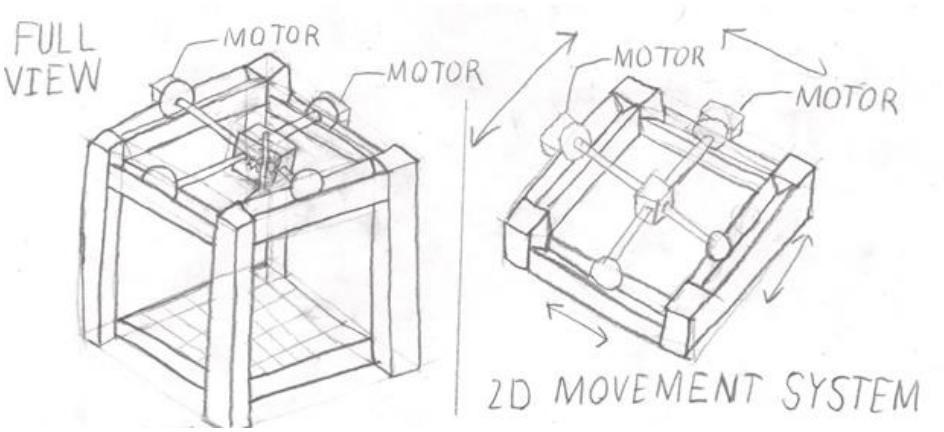


Figure 2: Backup Robot Design

This new design as depicted in the above figure was able to use one plane of horizontal movement to sense colours as well as pick up pieces. We opted out of using this design previously due to the inaccuracy that came with sensing colours from the top due to the amount of ambient light coming through, however this was our best choice.

Frame Design

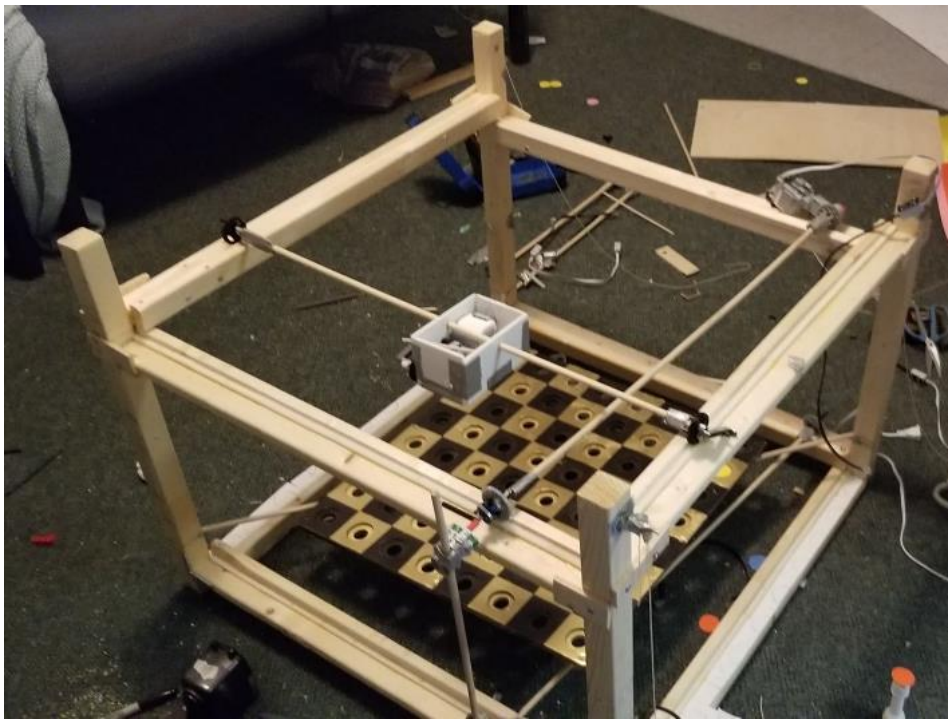


Figure 3: Frame of the Robot

The frame of the robot is a square based prism. The height is 60 centimeters and the width and length are 70 cm long. The decision for the large height was to try and meet the constraint of having 25 cm of space between the board and the bottom of the claw. This large base was to account for the space that the movement block would take up. Also, the board is quite big as it is 20 inches by 20 inches (or 50.8 cm by 50.8 cm). The reason we had such a large board was to allow for greater accuracy when picking up pieces and scanning, however in the conclusions section it will be explained how this had the reverse effect.

Lateral Movement Guides Design

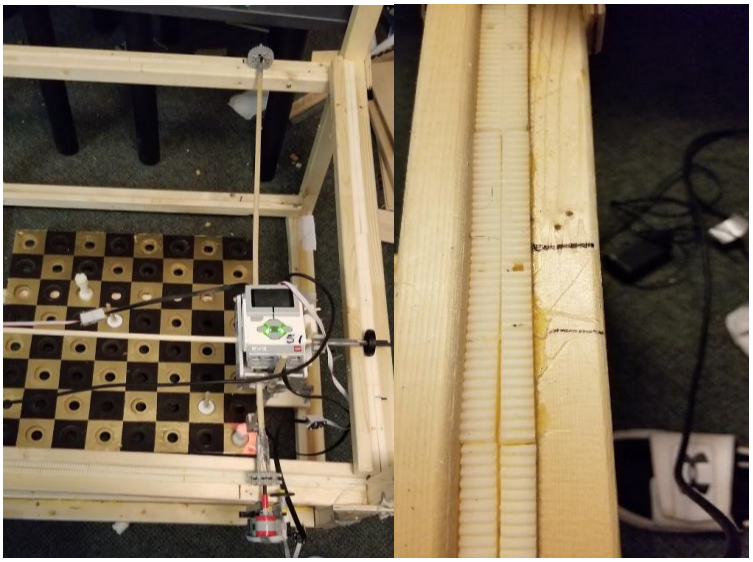


Figure 4: Top View of Lateral Movement and Gear Racks

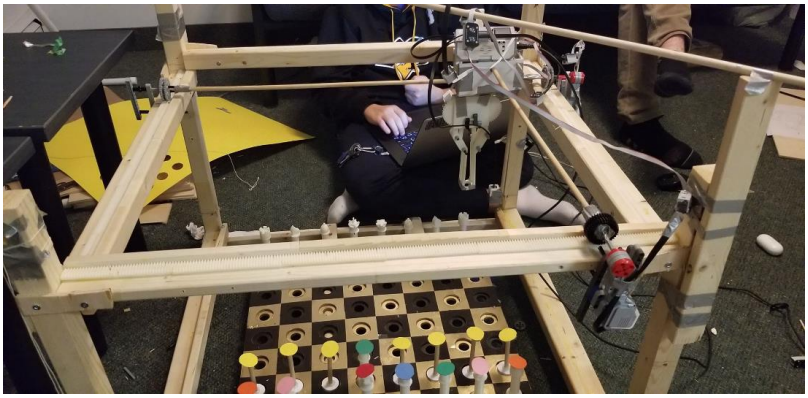


Figure 5: Side View of Lateral Movement

The above images show the mechanism that allows for lateral movement. Each of the surfaces which the gears turn on is lined with gear racks. These gear racks were printed using designs from Lego so that they would fit perfectly with the Lego gears which move along the rack. Gears were used rather than a wheel so that there would be more accuracy with the lateral movement.

In the original design two guide bars were to be used to make sure that the gear would not fall off the rack. However, we found that only one guide bar was necessary to ensure that the gears would not fall off the racks. This was due to the width of the gear racks as they were at least double the width of the actual gear. Also, since we were using gears it was rare that an axis would tilt in a way where the gear would come off. Occasionally the y axis movement

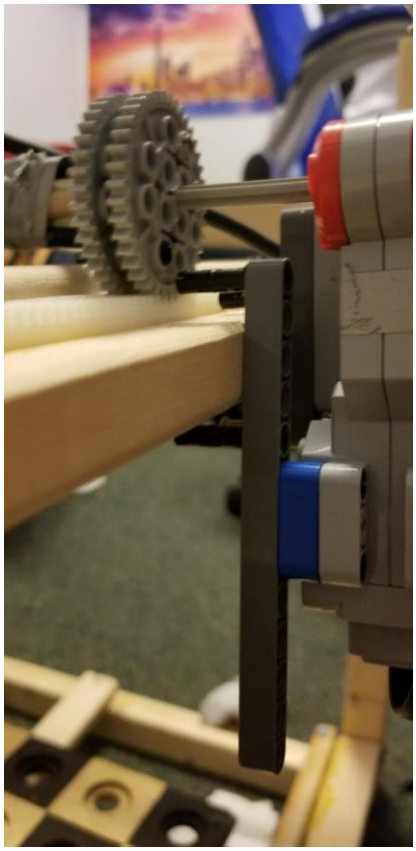


Figure 6: Motor Stabilizer

would shift the x axis dowel (and vice versa). To solve this issue, small Lego bars were placed onto the motors which would ensure no shifting of any axes would occur.

Claw and Chess Piece Design

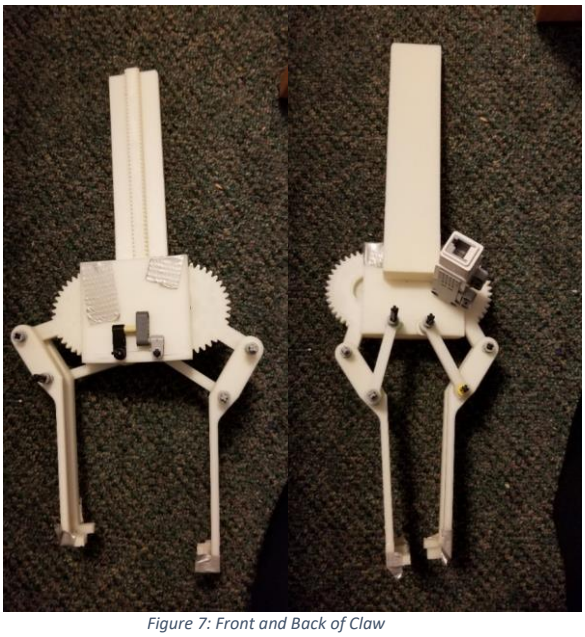


Figure 7: Front and Back of Claw

We needed a specific claw to pick up the chess pieces and not have any be knocked over, so we made our own chess pieces and claw. This claw and chess pieces were 3D modelled in SolidWorks and then 3D printed. The claw uses a medium motor which turns the gears to open and close. The claw is designed to pick up the pieces at a rim which all the pieces have at the same height. The rim functions as a consistent grabbing point for all the chess pieces to make programming for picking up pieces easier.

Figure 8: Example of Printed Chess Piece

Motor Drive Design

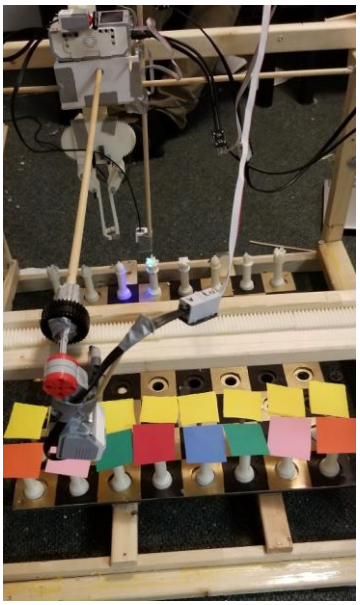


Figure 9: Motor Design

Rather than having 2 motors for each side of each axis we used one motor per axis. Although this decreased the efficiency and accuracy of our robot, this was our best option as we only had a set amount of motors.

The motor placement was more of a challenge than expected. We forgot to account for the torque of the motor, which caused it to rotate itself. This prevented the dowels from rotating properly. To fix this problem limits were placed onto the motor and the platform that the gear was rolling on (Figure 6: Motor Stabilizer).We found that placing these limits too tight would cause it to catch on the wood due to the uneven surface of the wood. The small pegs limit the movement of the motor just the right amount so that it would not catch, and it would allow for the axis to be turned.

Sensor Attachment Design

Figure 10: Colour Sensor Attachment

Due to the design change we had to now sense the colours from the top. This is a far less accurate method due to outside interreferences such as ambient light. We tested this by sensing from both the bottom and the top and compared the values. We observed that sensing from the bottom is much more accurate. When sensing from the bottom the values for a colour would only be off by 5-10, but when sensing from the top the values for the colours were off by more than 50.

To make sensing from the top to be accurate we required the colour sensor to be at a specific distance away from each color, as well as the area that is being scanned to be as big as possible. To accomplish this, we found the position at which the colour sensor would be most accurate, which was about 3.5 cm. Then we attached the colour sensor to a dowel and attached that to the movement block so that it would be at the right position to sense the colour. Large pieces of coloured paper were then attached to each piece.

These changes allowed for some decent colour measurements. However, they were still about 10-20 off so tolerances were needed. To find the right tolerance amount we scanned

each colour several times and found the range that the values would be in and set that to be the tolerance.

Cable Management

Figure 11: Rod for Cable Management

Cables were another unexpected issue to come up. Due to our large robot size, long cables were required. This posed a big issue of cable management. During the testing of the robot the cables had to be held up at the right position for them not to get caught and interrupt the robot's movement. This was fixed by attaching a long dowel diagonally across the top of the frame. Then the middles of the cables were attached to the middle of the dowel. With some fiddling around and testing we found adequate positions so that there would not have to be any cable adjustments during movement.

Movement Block

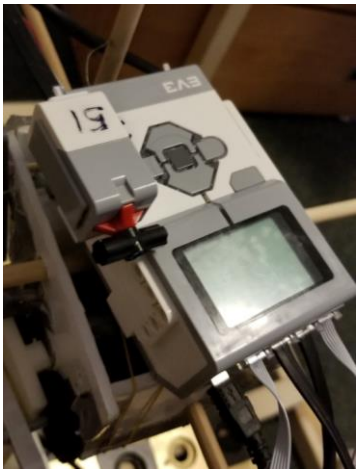


Figure 12: EV3 Mounted on Movement Block

The movement block is the most important component of the robot as it contains most of the key features. First, it allows for lateral movement, it has two slightly offset holes for each axis to move with no interference. Second, it holds the colour sensor and EV3 in place (Figure 10: Colour Sensor Attachment). Lastly it holds the claw and allows it to move vertically. The movement block contains a motor within it which turns a gear that moves the claw up and down.

Figure 13: Movement Block Holding the Claw

Initially the movement block was to be 3D printed, however this was not possible due to time constraints and our robot already requiring many 3D printed components. Our next best option was to make it out of foam board. The benefits of foam board are it is cheap and efficient. However, it has major drawbacks when it comes to accuracy and precision. This is due

to human error as everything had to be cut by hand. This meant not all holes were perfectly aligned and it was hard to place the gears surrounding the claw in the correct position.

In order to secure the claw within the gears, the gears had to be offset slightly, as finding the perfect distance between the two was very difficult. Even with this the gears were still slightly off so to fix this Lego bars and small wooden dowels were used to secure the claw in place.

Software Design and Implementation

Note the C# A.I. referenced is based upon [2]

Program Composition

For our program, we noticed our system had a lot of different sectors and as such we decided to break up our task lists based upon the different actions our robot had to do , and which parts we would need to call upon the most. We started with our main program, and we felt that this sector would just handle the general process of our robot to call upon the different functions to do procedures, and to deal with the process for initial setup and ending. As such we had the main function of our program work in such a way that someone with a very general view of our program would be able to look at our main and understand roughly the process that our robot went through. From there since we had a large amount of data that was going to be sent to and from different methods, in order to not be constantly passing a large amount of parameters, it made the most sense to put all the information within a struct. As such we decided to create the struct called “ColourValue” because we felt it aided the cleanliness of our program.

From there we moved on to our methods and our first major choice was to split up the different kinds of movement into different methods. Firstly we decided to split up our X and Y movement from both the sensing and from the picking up and placement of pieces as the movement would be done multiple times in both of those tasks and would also be done in both of those tasks which were separate from one another. As such it made sense to make the X and Y movement separate. In addition, there was a level of complexity into how the motors decided whether to move forwards and backwards and exactly how far to move and so when we tested that it just made sense to keep them separate from the rest of the program. In terms of separating the X and Y movement from each other, because they were separated physically and had different gears moving them and were at slightly different locations, we realized they would need different constants in order to make them work. Along with them powering different motors and so it made sense to keep them separate. In addition, in terms of readability and usability this way we would always quickly be able to tell which way the Robot was moving and would be able to only move the Robot in the Y direction for example if we so chose.

For our claw movement, we decided to instead of placing the process in main, to make a “movement” method that would deal with this entire process. We felt that since this process

was very specific, and there was a chance we could alter parts of it, that it made sense to place separately as its own method.

Within that claw movement we had a method called “pickup” that dealt with the picking up and placing the pieces. Since this had many different loops, we felt to place it separately from the movement class, as also that way we could test it specifically on its own. From there we also grouped together picking up and dropping the pieces, and this was largely due to the fact that there was some similarities in the picking up and dropping of the pieces, and their movement values are tied together and so we felt it made sense to group them into one method.

To move on from there we decided to separate the sensing from the rest of the system. This is mainly because sensing itself is fairly complex and integral to our program working, so in order to best test it and again in terms of cleanliness we felt it was best to break this up separately from the rest of the program. In terms of the Initial sense versus the Regular sense we felt since they had the same basic principles and worked off the same Idea we didn’t have a need to separate them within our main code and so we decided to keep them together as it didn’t necessarily benefit use by putting them separately.

In continuation, for our function isColor we felt since it has a large amount of if statements and is done multiple times for multiple different colours, it would be much cleaner and easier to understand if we put it as a separate method with a integer for the different piece types. This would also make it so any changes could all be done in one location.

For the method “countdown” although it is quite simple, since the function is done many times over our main, for cleanliness sake we decided just to call one method instead of writing it out multiple times.

For “checkButtonPress” we decided that if we put it all in one method we would be able to repurpose it for both getting the difficulty and the timer and as such we decided to separate it from the rest of the code.

Lastly, for FileIO since it is connected to the c++ code and as such deals with opening and closing a file repeatedly and has its own intricate setup that must be checked, we felt it was important to place it seperately from the rest of the code.

Demo Task List

Below follows the generic list of what our program showed on demo day.

1. EV3 prompts user for difficulty and game settings
2. Conducts initial board scan and display warning if empty space value is within tolerance of and piece value
3. Robot returns to origin
4. Display message for player to move
5. Robot begins scan when player presses touch sensor

6. Robot moves one of its pieces
7. Repeats operation tasks 3-6 until an end condition is detected, then continues to task 8
8. Displays message depending on end condition
9. Robot returns to origin (if not already there)
10. Robot C, C++, and C# code shut down

This list is starkly less then our task list for our full function as seen on "Tasks". For our demo, we had separate code that we ran (unit test files) for demonstrating our robot. We did this mainly because through testing we were not confident that our robot could reliabley do one move all at once and we did not want our robot breaking down completely during demonstration. Due to our lack of confidence and testing time we were not able to show off our File.IO communicating with our C# A.I. and as such we had to manually enter in the movement for the claw, and were only able to demonstrate the claw moving one specific move. Sadly, however as previously stated this wasn't all done through one program as we had all of our parts demonstrated (sensing, claw movement, timer, end game) through our separate test files, however we did not run the all together code. Ironically we did test it directly after our demo and it worked perfectly, however purely for demo we did not showcase all of our tasks that were mentioned earlier on in this report as a result of lack of testing and confidence in our movements being precise, not allowing us to even test our full movement together and so we didn't feel right to attempt to showcase it for the first time at that event.

Table 2: List of Functions and Their Descriptions

Name	Parameters	Type	Description	Written By
checkButtonPress	bool difficulty	int	Waits for a button to be pressed and then released. Return a different integer corresponding to the different buttons. Restrict which buttons are being tested for if it's a supposed to be checking difficulty.	Allen Zhang
xMovement	float oldX, float newX, bool reset	void	Takes the inputted oldX and new X, sets the motor to positive or negative if it moves forward or backwards in the X direction. Then loops until the motor has moved the arm to the location. The function resets the motor encoder if the move that is not made is not a successive one, which is determined using the reset bool.	Cameron Kinsella

yMovement	Float oldY, Float newY bool reset	void	Takes the inputted oldY and new Y, sets the motor to positive or negative if it moves forward or backwards in the Y direction. Then loops until the motor has moved the arm to the location. The function resets the motor encoder if the move that is not made is not a successive one, which is determined using the reset bool.	Cameron Kinsella
placePiece	bool pickup	void	This functions the sequence for the arm picking up a piece and dropping a piece. If it is picking it up, it will open the arm based on tested data, then drop the claw the required amount the close the claw then lifts the arm up the required amount. If it is not picking up, then the program will drop the arm then open the claw then raise the arm the required amount.	Adam Barroso
movement	int oldX, int oldY, int newX, int newY	void	This function runs through the entire process for picking up and moving a piece. It called the x and y movement to move the claw to the starting position. Then call the placePiece function to pick up the piece. From there call the x and y movement function to move the arm to the new location. Call the placePiece function to drop the piece. Lastly, call the movement functions to bring the arm back to the starting point.	Adam Barroso
isColor	Int Chosen, ColourValue & Piece (main class), int Red, int Green, int Blue	bool	This function checks if the colour values inputted match the stored constants for the colour values of the chosen piece type, and if it is then it returns true, otherwise it returns false.	Adam Barroso

scanBoard	Bool initial, ColourValue & Piece	void	This starts by bringing the colour sensor to the centre of the square. This then either does the initial scan of the board or does the regular scan. If the initial scan is chosen, the initial values for the locations for the white pieces are setup for the char 2-d array. From there the function will move to the specified locations and sense one of each type of white piece and store the colour value sensed in their appropriate arrays. If it is doing a regular sense, then the arm will snake along the board starting on the black side. It will sense the value of each square on the board and check to see if it matches any of the piece colour values. If not, it will set that location in the array to empty, otherwise it will set it to the appropriate character value for what piece was sensed and add one to the piece counter. The sensing will end if the correct number of pieces has been sensed or the sensing reaches the end of the board.	Adam Barroso
fileIO	ColourValue Values, int Difficulty, bool StaleMate, bool WhiteKing, bool BlackKing, int Attack, int CurX, int CurY, int NewX, int NewY	void	This function performs all of the fileIO tasks. It takes uploads the difficulty of the A.I. and then the state of the Board in Terms of White pieces to the text file “aistuff.txt”. From there it opens the text file “chessmove.txt” and wait for the A.I. to send in the appropriate information about the computers move. It then takes all that information and stores them in their values so that the Robot can use them. Then deletes all the information off both text files; resetting them for the next use	Adam Barroso

countdown	none	Void	This function very simply runs a timer for five seconds and displays a large countdown from 5 to zero on the screen	Allen Zhang

Flow Charts

Flowcharts were created to aid in the design and comprehension of the code. Refer to Appendix C for all software flowcharts.

Data Storage:

Within our program, all the important data is stored within our "ColourValue" struct as seen (Page link, line link) which we create one instance of in our main called "CurrentStuff" that gets used all throughout our program. For each of our specified pieces types Rook, Pawn, Knight, etc. We store their Red, Green, Blue colour values in a specific colour array with a size of 3. Index 0 being Red, 1 being blue, and 2 being Red. We decided to store the RGB values because it allowed us to more precisely tell what colour each of the pieces were and allowed us a larger range of colours for example, we used pink as one of our colours. The number of pieces was also important so it too was stored within this class with an integer as you cannot have a decimal value of pieces. This started with an initial value of 16 because each game the white team starts with 16 pieces. Last within this class we also store a 2d char array size 8 by 8 to match the board size, this contains the location of all the white pieces. Their x and y location match their location within this 2-d array, with each piece having a specific character value for example pawn has 'P' and knight has 'H' because 'K' is taken for King.

Outside of the struct, we have several constants for the different movements that are done. We have constants for how many rotations must be made to move exactly one space on the board in both the x and y direction and constants for what speed the motors will move at.

Additionally, we store bools within the main sector of the program for White King and Black King being alive, and the game being a stalemate, these Booleans are used to determine when the game should end, with the obvious case being if one of the kings is not alive or if it is a stalemate is alive the game ends.

We also store the settings I.e. difficulty and timer values within integers as that way the C# A.I. knows what difficulty to operate at, and we can set the white king to be dead if the player runs out of time.

In addition, for storage, our program handles storage via text files. On the RobotC end we send the difficulty of the A.I. and the current state of our White Piece 2D array to a text file on the Robot in the format that it would look like if one was to look at the board from the white side. From there the C++ code will go upload that file to our computer and run the A.I. The A.I. produces a text File on the Pc end that contains one integer/bool per line. These bools contain, whether or not the game will be a stalemate, if the White King/Black King will be alive after the

next turn and then what move the computer will make (old x, old y, new x, new y). These values are uploaded to the Robot and then the RobotC reads them into their correlated bools and integers and performs the appropriate actions. We chose to store the data on a file in the form of Integers and bools as we felt it was the easiest way to communicate different elements of the board state, and took a lot of the A.I. work off the RobotC code itself and instead allowed us to use and updated version of a C# Chess A.I. that was made before the start of this project, with C#. Since we could not move this A.I. on to the Robot then communicating via File IO ended up being the best option.

Lastly, we had a number of constants within our program that we used to basically better help the user understand what each value meant and that we also used when we needed to use a literal more than once.

Software Design Decisions and Trade-Offs

The first major software design choice was to not use an online A.I. and instead use a modified version of the C# that was made by one of our members in high school. This was because we wanted to make every little bit of our Robot in-house and we felt that it would allow us more freedom in what we could send back and forth between our robot and the A.I. and how it could be sent. This decision came with the trade-off that we had to figure out some sort of file transfer from our Robot to the computer, which took up a large chunk of our time as we went through different options including but not limited to Bluetooth, sending via Wi-Fi, debug stream, automatic file-Io, and sending data directly through the USB. However, we finally had to come to the decision the most logical method was manual file.IO sending. This was easy to code for the RobotC code; however, it added the extra manual aspect that the user had to upload the files themselves, so it was decided to make C++ code to do all of that automatically. While it was able to move the mouse and click in the right locations faster than a human, it had an added barrier to entry that u had to have all the windows in the right locations and that the RobotC file had to already be opened to the right folder otherwise it wouldn't work.

Moving onto our actual movement, we originally just had a simple movement where it determined the distance between the location the robot was currently at, and then move for that distance. However, this had the drawback that if we were off by even 0.01cm then each time that percentage off would be multiplied. This is because we were assuming that it perfectly reached the center of the square and basing our calculations off that wrong value, as such we decided to change the way our x and y movement worked so that all our values were based off the distance from (0,0). This generated another issue where backwards movement wouldn't work so we then had to update our movement so that for sequential moves we looked at the exact motor encoder value and then used the last motor encoder value and the distances based on zero to triangulate how far we had to move both forwards and backwards. While this solution worked the drawback was the large amount of time that came with making it this precise.

For our claw movement we decided to code it in such a way that all the opening and closing of the claw was based upon set values, this was because we felt we could keep them

consistent. This had the trade-off that when for example our motor would spin in place, we had no detection system to determine exactly whether or not the claw was open or closed, or if for example the restraints holding the claw became loose and gravity began to take affect we would not be able to detect that the claw has gone too low. This amounting in us performing large amounts of testing just to get the claw barely functional.

For our sensing, our first main decision was to take an initial sense of the colours of each of the specific pieces at the start of the game. This turned out to be an amazing decision as this allowed our robot to work in many different light areas and if the movement was done correctly, sense to a high degree of accuracy. Another sensing decision we made was to sense the board moving along all the points on the board and check if each position matched one of our colour values. This was a good decision as it was the most reliable way to make sure that all the white pieces got sensed. We could potentially have sensed the board using some algorithm and then determine where the pieces are likely to be and this would probably have reduced our sensing time, however this would introduce more variables and then would add on more time to our generation of code and testing which was already an issue. In addition, one drawback is since we started at the black side of the board then our robot took a large amount of time to sense at the beginning. In addition, since we didn't store the location of the black pieces then we had the potential to sense those as white pieces despite being low was still possible, and that added to our sensing time because we sensed 16 extra spaces.

Testing Procedure

For our testing procedure we used unit testing and broke up all our major functionalities into separate files which we then used to test constants and logic as followed.

Table 3: List of Functions for Testing

Tester Code Name	General Breakdown of what code does	How we used it to test
Claw Test (28)	This tester code would run the whole pickup and drop procedure for whatever piece to what location was specified within the main.	We used this code to test how much we needed to open and close the claw and how much we needed to drop and raise the claw. We had general numbers however we needed to tweak our constants and make sure the process was correct and so this test was perfect for testing accuracy. In addition, due to the nature of the pickup, this allowed us to make sure our X and Y movements were being made precisely and we could also alter those constants if need be.
Colour Tester (34)	This function would run three scans. During its first scan it would run all the regular initial scan getting colour values for each of the	We mainly used this to make sure that our sensor was moving correctly to all the correct locations and that it was more or less being placed in the centre of the squares. As such this testing caused us to look at the logic of our X and Y movement

	piece types, in addition to getting the colour values for a black and empty space and then checking to see if they're within the tolerance of any of the white pieces. During its second scan it would once again run the initial scan movements this time making sure that the pieces previously measured are still within the colour parameters that are just measured. The third time it would do a full scan of the board and then output how long it took.	and the constants that drive that movement. In addition, we had to make sure the colour sensor was properly being put in the centre of the squares and that our tolerances that we measure were good values. In addition, we measured to make sure all our data was stored correctly and that the logic of doing a full scan was done correctly, in how the sensors snakes along the board and that our speed fit our constraints.
Timer Test (48)	This was an extremely simplified version of our main for our full code that didn't do any scanning or moving but had a smaller timer value and was used to make sure the game ended when the timer ended and white lost.	This was mainly used to make sure our initial setup values were working correctly and that all timers and displays for different button presses were working correctly. We knew it worked if at the end of the timer the program stopped and outputted that white had won. We could also use this to test that after each move our value was being saved correctly and the timer started where it left off. As such the main purpose of this code was to make sure the logic of all the timers and button presses was working correctly on its own.
Robot File Io (53)	This code ran to act as a pseudo read and write code file. Where it would write the information to the files, then wait for information to read, then upload them to the correct locations and reset the files.	The main testing that came with this was making sure all the logic and the casting was done correctly and the code was correctly waiting for the files to be uploaded and only reading when the file was ready to be read and writing all the information in the correct format. For this we used tester files and tester white piece values just to make sure the situation worked. We knew it worked correctly based on whether the code was correctly outputting what was stored, and if the displayed values that it gathered from the file, matched what was put into the file manually. Lastly, we had to check it would work multiple times in a row which checked the file deletion.

C++ Screen Clicker (55)	This code ran through the whole process of taking the file from the Robot and uploading it to the Pc, via clicking and then running the C# to get the A.I values, the upload those values to the robot.	Due to the nature of this code, the main tests that we ran were to make sure all the pixel values were correct, that the code properly opened and read whether or not the file was empty, that it properly ran and waited for the C# code, and that it deleted the files off the PC and uploaded them to the Robot. As such all testing was just making sure the pixel values taken using paint were correct and the pauses were not too short or long and overall it was able to know when to upload and download files and do so successfully. When testing we would use manually created files.
Full Code (59)	This wasn't really tester code as it ran the whole procedure (wasn't tested until after demo day)	The purpose of this code was to make sure that when we put together all the other tester codes and all the other functions into one and set it up practically, we weren't getting unnecessary errors, and everything worked as expected.

Significant Problems

For us our main problems did not actually come on our software side, it was more the mechanical side that went wrong. We had significant setbacks when trying to look through the different types of ways we could get our RobotC to automatically run and send information to our C# A.I. This is because we expected that File IO would be able to send files to the computer when the USB was plugged in, or that we could use Bluetooth or debug stream in order to physically send information. Researching all these avenues took many days and so it set us very far back when we then had to go through the grueling process of testing out getting the C++ code to move the cursor to the correct pixels and click on the right things.

Another Major problem was our movement, we ended up solving it, however as previously stated our original design for movement did not account for the motor encoders missing the Centre of the squares by any amount and so the errors would compound. As such we had to completely rewrite our movement so it would account for the motor messing up and make sure that the errors did not compound, but this was a large amount of testing and that caused us much error.

In general the largest chunk of our time was spent taking our code and attempting to find the constant values that we needed to do each movement at a consistent level because our mechanical structure was so unreliable and so this was probably our biggest and our fatal problem that we sadly could not overcome to make the system reliable.

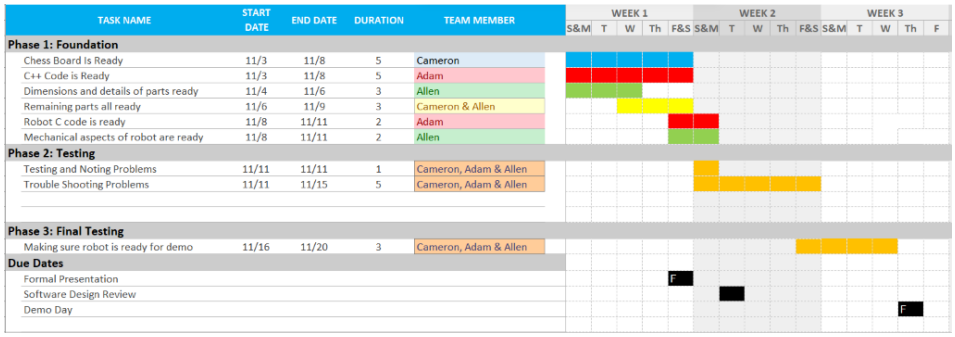
Verification

All of Checkers' constraints were met on demo day. The EV3 did not output any warnings during the empty space test, so the empty space scan was not detected as any of the

initialized pieces. Also, the measuring tape test revealed the bottom of the claw resting position to be approximately 20cm above the board. This did not meet the 25cm listed in the preliminary report, however the overall goal of not impeding the players hand was achieved nonetheless, as was determined from each group member being able to move chess pieces without issue.

Project Plan

Each aspect of the robot building was head by a group member. Allen lead the mechanical aspects of the robot. Cameron lead the 3D modelling and designing portion. Adam lead the software portion.



TASK NAME	START DATE	END DATE	DURATION	TEAM MEMBER
Phase 1: Foundation				
Chess Board Is Ready	11/3	11/8	5	Cameron
C++ Code is Ready	11/3	11/8	5	Adam
Dimensions and details of parts ready	11/4	11/6	3	Allen
Remaining parts all ready	11/6	11/9	3	Cameron & Allen
Robot C code is ready	11/8	11/11	2	Adam
Mechanical aspects of robot are ready	11/8	11/11	2	Allen
Phase 2: Testing				
Testing and Noting Problems	11/11	11/11	1	Cameron, Adam & Allen
Trouble Shooting Problems	11/11	11/15	5	Cameron, Adam & Allen
Phase 3: Final Testing				
Making sure robot is ready for demo	11/16	11/20	3	Cameron, Adam & Allen
Due Dates				
Formal Presentation				
Software Design Review				
Demo Day				

Figure 14: GANTT Chart for Original Schedule [1]

This GANTT chart shows the original schedule that was made for this project. Many things got delayed for various reasons. The “Mechanical aspects of robot are ready” was delayed due to not having all the necessary parts to begin building. This mostly came from 3D printing, since we underestimated the time it would take to get all parts 3D printed. Since the mechanical portion was delayed this delayed “Robot C code is ready” as we were not able to have measurements ready.

The actual schedule for this required a lot of work to be done a few nights before the due date. We had planned to have the full build ready by November 11, however due to the delays and having to change our entire design this was ready by November 18. Testing was planned to begin on November 16. Testing actually began on November 20. Two all-nighters had to be done in order to get our robot somewhat working.

Conclusions

Even though the scope, criteria, and constraints went largely untouched throughout the design of Checkers, the mechanical and software design had to overcome major revisions.

Due to the issue of the wooden dowels preventing proper movement of the under-board colour sensor, the under-board method of scanning the piece colours was rejected in favour of scanning the pieces from the top. This decision came with the consequence of less accurate colour scans due to the increase of external factors such as ambient light. Cable management also proved to be a challenge but was later able to be effectively handled.

As for software, there were very few changes from the original plan. The biggest change was the decision to code the A.I. in C# rather than C++ for simplicity and versatility. The only issue that occurred was a setback when looking for a method to transfer from C++ to Robot C. A simple solution was created in the end, but the process of finding said solution caused a significant delay in the development of the software.

Recommendations

Mechanical Design:

The main issue for the robot came with the accuracy and reliability. These issues stemmed from 3 things: size, weight and materials.

First the size of our robot was the stem to many of the problems. We designed the robot far too large. This caused things to warp and bend in ways that were not expected. As well it made it so that the Tetrix kit could not be used to build the robot.

The motors were too heavy for the materials that we had. We noticed that over time the two motors attached to the claw was causing the wooden dowels to bend and warp. This warping of the dowels caused the claw to rock back and forth during movement. This caused the accuracy of the robot to plummet as its movement were unpredictable.

Due to the size of the robot the Tetrix kit was too small to be used, therefore most of the robot was built with wood. Wood and Lego are not very compatible with each other. This was a problem when attaching the wooden dowels to the gears. The dowels had to be taped to the gears as they would not securely fit on the gears. This caused a lot of inaccuracy as the dowels would not be perfectly centered.

Software Design:

Doing more thorough tests of the full system rather than just separate units would have significantly increased reliability, since many unexpected occurrences were due to integration errors.

Additionally, it would be beneficial to change up the way our file IO checks the file in order for it to not overlap with the C++ trying to write to the file. This is because technically

there is a small chance that the file will get corrupted when two people try accessing it at once, and so we would modify it so it would only open the file if it wasn't currently under use.

References

[1] A. Zhang, A. Barroso and C. Kinsella, "Checkers - The Chess Playing Robot," University of Waterloo, Waterloo, 2019.

[2] A. Barroso, *C# A.I Code,* Mississauga, 2018.

[3] *c++ cursor(mouse) movement,* 2017.

[4] "cplusplus.com," 31 July 2010. [Online]. Available: http://www.cplusplus.com/forum/beginner/26798/. [Accessed 2019].

[5] E. Bachaalany, "Code Project," 14 June 2004. [Online]. Available: https://www.codeproject.com/Articles/6819/SendKeys-in-C. [Accessed 2019].

[6] Microsoft, "Microsoft- Virtual-Key Codes," 30 May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/inputdev/virtual-key-codes. [Accessed 2019].

[7] "stackoverflow," 21 2014 January. [Online]. Available: https://stackoverflow.com/questions/21257544/c-wait-for-user-input. [Accessed 2019].

[8] cpluscplus.com, "cplusplus.com," 2010-2019. [Online]. Available: http://www.cplusplus.com/reference/cstdio/remove/. [Accessed 2019].

[9] pajton, "stackoverflow," 6 March 2010. [Online]. Available: https://stackoverflow.com/questions/2390912/checking-for-an-empty-file-in-c. [Accessed 2019].

[10] Morpheus13, "stackoverflow," 17 September 2014. [Online]. Available: https://stackoverflow.com/questions/15435994/how-do-i-open-an-exe-from-another-c-exe. [Accessed 2019].

[11] I. Person, "stackoverflow," 14 Jan 2015. [Online]. Available: https://stackoverflow.com/questions/27937617/how-detect-spacebar. [Accessed 2019].

Things to check in revision:

- All tables and figures have captions which are **center justified**
- All body text is 12 size font, single spaced
- All formatter is in the section with roman numeral pages
- RobotC not Robot C
- References and citations are done correctly. Code is in proper font and uploaded correctly
- All sources are done correclty
- C# code is cited
- General readability is good, (quick skim)
- Consistent look
- Page numbers are good

(add more things like this if you notice them)

Appendix A

Commented [CK2]: Please look through this again to make sure that any in-line comments you made did not overflow to another line.

Claw Test:

```
#include "PC_FileIO.c"

const int BoardSize = 8;

const float oneSquare = 6.35;

const float xMovementconst = oneSquare*(360/(3.97*PI));
```

```
const float yMovementconst = oneSquare*(360/(3.55*PI));
const int XYPower = 15;
const int ClawPower = 100;
const int ClawDown = 35;

void countdown ()
{
    time1[T2] = 0;
    while(time1[T2] <5000)
    {
      displayBigTextLine (1, "%d SECONDS”, (5000-time1[T2])/1000);
    }
}

void xMovement (float oldX, float newX, bool reset)
{
    //for info see main code
    int Negative = 1;
    float XChange = newX-oldX;
    if (XChange<0)
    {
        Negative = -1;
        XChange *= Negative;
    }
    if(reset)
    {
        nMotorEncoder[motorD] = 0;
    }
    motor[motorD] = XYPower*Negative;
```

```
    if (Negative>0)
    {

        while(nMotorEncoder[motorD]<newX*xMovementconst)
        {
        }
    }
    else
    {
        while(nMotorEncoder[motorD]>newX*xMovementconst)
        {
        }
    }
    motor[motorD] = 0;

}
void yMovement (float oldY, float newY, bool reset)
{
    //for info see main code
    int Negative = 1;
    float YChange = newY-oldY;
    if (YChange<0)
    {
        Negative = -1;
        YChange *= Negative;
    }
    if(reset)
    {
        nMotorEncoder[motorA] = 0;
```

```
    }
    motor[motorA] = XYPower*Negative;
    if (Negative>0)
    {

        while(nMotorEncoder[motorA]<newY*yMovementconst)
        {
        }
    }
    else
    {
        while(nMotorEncoder[motorA]>newY*yMovementconst)
        {
        }
    }
    motor[motorA] = 0;
}
void placePiece (bool pickup)
{
    const int openDegrees = 40;
    const int upPower = 5;
    const int closeDegrees = -20;
    const float Height = 9.8;
    if(pickup)
    {
        //open claw
        nMotorEncoder[motorB] = 0;
        motor[motorB] = ClawPower;
        //wait time is because motor sometimes spins in place
```

```
    wait1Msec (250);
    while(nMotorEncoder[motorB] < openDegrees)
    {}
    motor[motorB] = 0;
}
//drop the arm
wait1Msec (500);
nMotorEncoder[motorC] = 0;
motor[motorC] = -ClawDown;
while(abs(nMotorEncoder[motorC])
< (Height)*(yMovementconst/BoardSize))
{}
motor[motorC] = 0;
wait1Msec (1000);

if(pickup)
{
    //close claw
    motor[motorB] = -ClawDown;
    //-20 degrees due to again motor spinning in place
    while(nMotorEncoder[motorB] > closeDegrees)
    {}
    motor[motorB] = 0;
}
else
{
    //open claw
    nMotorEncoder[motorB] = 0;
    motor[motorB] = ClawDown;
```

```
        wait1Msec (250);
        while(nMotorEncoder[motorB] < openDegrees)
        {}
        motor[motorB] = 0;
    }
    wait1Msec (1000);
    //move arm up
    nMotorEncoder[motorC] = 0;
    motor[motorC] = upPower;
    while(nMotorEncoder[motorC] < (Height*(yMovementconst/BoardSize)))
    {}
}

void movement (int oldX, int oldY, int newX, int newY)
{
    //general movement procedure
    xMovement(0, oldX,1);
    wait1Msec (250);
    yMovement(0, oldY,1);
    wait1Msec(500);
    placePiece(true);
    wait1Msec(250);
    xMovement(oldX, newX,0);
    wait1Msec(250);
    xMovement(oldY, newY,0);
    wait1Msec(500);
    placePiece(false);
    wait1Msec(250);
    xMovement(newX,0,0);
```

```
    wait1Msec(250);
    yMovement(newY,0,0);
    wait1Msec(250);
}

task main ()
{
    //small countdown then show movement of claw
    eraseDisplay();
    countdown();
    movement(0,0,4,0);
}
```

Colour Tester:

```
const int BoardSize = 8;
typedef struct
{
    int RookCol[3];
    int PawnCol[3];
    int KingCol[3];
    int QueenCol[3];
    int BishopCol[3];
    int KnightCol[3];
    int EmptyCol[3];
    int BlackCol[3];
    int WhitePiece;
    int CurColour[3];
    char whitecoords[BoardSize][BoardSize];
}ColourValue;
```

```
const int Tol = 5;
const float oneSquare = 6.35;
const float xMovementconst = oneSquare*(360/(3.97*PI));
const float yMovementconst = oneSquare*(360/(3.55*PI));
const int XYPower = 15;
void xMovement(float oldX, float newX, bool reset)
{
    int Negative = 1;
    float XChange = newX-oldX;
    if(XChange<0)
    {
        Negative = -1;
        XChange *= Negative;
    }
    if(reset)
    {
        nMotorEncoder[motorD] = 0;
    }
    motor[motorD] = XYPower*Negative;
    if(Negative>0)
    {
        while(nMotorEncoder[motorD]<newX*xMovementconst)
        {}
    }
    else
    {
        while(nMotorEncoder[motorD]>newX*xMovementconst)
        {}
```

```
    }
    motor[motorD] = 0;
}
void yMovement(float oldY, float newY, bool reset)
{
    int Negative = 1;
    float YChange = newY-oldY;
    if(YChange<0)
    {
        Negative = -1;
        YChange *= Negative;
    }
    if(reset)
    {
        nMotorEncoder[motorA] = 0;
    }
    motor[motorA] = XYPower*Negative;



    if(Negative>0)
    {
         while(nMotorEncoder[motorA]<newY*yMovementconst)
        {}
    }
    else
    {
        while(nMotorEncoder[motorA]>newY*yMovementconst)
        {}
```

```
    }
    motor[motorA] = 0;
}


bool isColor(int Chosen, ColourValue & Piece, int Red, int Green, int
Blue)
{
    // Pawn = 1, Rook = 2, Knight = 3, Bishop = 4, King = 5 , Queen =
6
    if(Chosen == 1)
    {
        return (abs(Piece.PawnCol[0]-Red)<Tol
        &&abs(Piece.PawnCol[1]-Green)<Tol
        &&abs(Piece.PawnCol[2]-Blue)<Tol);
    }
    else if(Chosen == 2)
    {
        return (abs(Piece.RookCol[0]-Red)<Tol
        &&abs(Piece.RookCol[1]-Green)<Tol
        && abs(Piece.RookCol[2]-Blue)<Tol);
    }
    else if(Chosen == 3)
    {
        return (abs(Piece.KnightCol[0]-Red)<Tol
        &&abs(Piece.KnightCol[1]-Green)<Tol
         && abs(Piece.KnightCol[2]-Blue)<Tol);
    }
    else if(Chosen == 4)
    {
        return (abs(Piece.BishopCol[0]-Red)<Tol
```

```
        &&abs(Piece.BishopCol[1]-Green)<Tol
        && abs(Piece.BishopCol[2]-Blue)<Tol);
    }
    else if(Chosen == 5)
    {
        return (abs(Piece.KingCol[0]-Red)<Tol
        &&abs(Piece.KingCol[1]-Green)<Tol
        && abs(Piece.KingCol[2]-Blue)<Tol);
    }
    else if(Chosen == 6)
    {
        return (abs(Piece.QueenCol[0]-Red)<Tol&&
        abs(Piece.QueenCol[1]-Green)<Tol &&
        abs(Piece.QueenCol[2]-Blue)<Tol);
    }
    return false;

}

void scanBoard(int initial, ColourValue & Piece)
{
    Piece.WhitePiece = 16;
    //if first initial sense
    if(initial==0)
    {
        yMovement(0,6.8/oneSquare,0);
        xMovement(0, -3/oneSquare,0);
        //inital filling up of hte array
        for(int Row = 0; Row< BoardSize-2; Row++)
```

```
{
    for(int Col = 0; Col<BoardSize; Col++)
    {
        Piece.whitecoords[Row][Col] = ' ';
    }
}
for(int Col = 0; Col<BoardSize; Col++)
{
    Piece.whitecoords[6][Col] = 'P';
}
Piece.whitecoords[7][0] = 'R';
Piece.whitecoords[7][1] = 'H';
Piece.whitecoords[7][2] = 'B';
Piece.whitecoords[7][3] = 'Q';
Piece.whitecoords[7][4] = 'K';
Piece.whitecoords[7][5] = 'B';
Piece.whitecoords[7][6] = 'H';
Piece.whitecoords[7][7] = 'R';
Piece.WhitePiece = 16;

//move to rook location
xMovement(0,7,1);
wait1Msec(1000);
getColorRGB(S3,Piece.RookCol[0],
Piece.RookCol[1],Piece.RookCol[2]);
//colour outputs
displayBigTextLine(1,"%d",Piece.RookCol[0]);
displayBigTextLine(3,"%d",Piece.RookCol[1]);
displayBigTextLine(5,"%d",Piece.RookCol[2]);
```

```
wait1Msec(1000);
//sense knight value
yMovement(0,1,1);
wait1Msec(1000);
getColorRGB(S3,Piece.KnightCol[0],
Piece.KnightCol[1], Piece.KnightCol[2]);
displayBigTextLine(1,"%d",Piece.KnightCol[0]);
displayBigTextLine(3,"%d",Piece.KnightCol[1]);
displayBigTextLine(5,"%d",Piece.KnightCol[2]);
wait1Msec(1000);
//sense bishop value
yMovement(1,2,0);
wait1Msec(1000);
getColorRGB(S3,Piece.BishopCol[0],
Piece.BishopCol[1],Piece.BishopCol[2]);
displayBigTextLine(1,"%d",Piece.BishopCol[0]);
displayBigTextLine(3,"%d",Piece.BishopCol[1]);
displayBigTextLine(5,"%d",Piece.BishopCol[2]);
wait1Msec(1000);
//sense Queen value
yMovement(2,3,0);
wait1Msec(1000);
getColorRGB(S3,Piece.QueenCol[0],
Piece.QueenCol[1],Piece.QueenCol[2]);
displayBigTextLine(1,"%d",Piece.QueenCol[0]);
displayBigTextLine(3,"%d",Piece.QueenCol[1]);
displayBigTextLine(5,"%d",Piece.QueenCol[2]);
wait1Msec(1000);
//sense king values
```

```
yMovement(3,4,0);
wait1Msec(1000);
getColorRGB(S3,Piece.KingCol[0],
Piece.KingCol[1],Piece.KingCol[2]);
displayBigTextLine(1,"%d",Piece.KingCol[0]);
displayBigTextLine(3,"%d",Piece.KingCol[1]);
displayBigTextLine(5,"%d",Piece.KingCol[2]);
wait1Msec(1000);
//sense pawn value
xMovement(7,6,0);
wait1Msec(1000);
getColorRGB(S3,Piece.PawnCol[0],
Piece.PawnCol[1],Piece.PawnCol[2]);
displayBigTextLine(1,"%d",Piece.PawnCol[0]);
displayBigTextLine(3,"%d",Piece.PawnCol[1]);
displayBigTextLine(5,"%d",Piece.PawnCol[2]);
wait1Msec(1000);
//empty square value check
xMovement(6,5,0);
wait1Msec(1000);
getColorRGB(S3,Piece.EmptyCol[0],
Piece.EmptyCol[1],Piece.EmptyCol[2]);
displayBigTextLine(1,"%d",Piece.EmptyCol[0]);
displayBigTextLine(3,"%d",Piece.EmptyCol[1]);
displayBigTextLine(5,"%d",Piece.EmptyCol[2]);
wait1Msec(3000);
//Black piece value check
xMovement(5,0,0);
getColorRGB(S3,Piece.EmptyCol[0],
```

```
 Piece.EmptyCol[1],Piece.EmptyCol[2]);
 displayBigTextLine(1,"%d",Piece.BlackCol[0]);
 displayBigTextLine(3,"%d",Piece.BlackCol[1]);
 displayBigTextLine(5,"%d",Piece.BlackCol[2]);
 wait1Msec(2000);
 yMovement(4,0,0);
 //runs through all the colour values and sees
 //if the empty spot is detected as any of them
 for(int Row=0;Row<=6;Row++)
 {
    if(isColor(Row,Piece,Piece.EmptyCol[0],
    Piece.EmptyCol[1],Piece.EmptyCol[2]))
    {
         eraseDisplay();
         displayTextLine(3,"empty spot detected as %d !",Row);
         wait1Msec(5000);
    }
}
 //runs through all colour values checks
 //if black is detected as any of them


  for(int Row=1;Row<=6;Row++)
  {
      if(isColor(Row,Piece,Piece.BlackCol[0],
      Piece.BlackCol[1],Piece.BlackCol[2]))
      {
          eraseDisplay();
          displayTextLine(3,"black piece detected as %d !",Row);
```

```
            wait1Msec(5000);
        }
    }
}
//if 2nd inital sense to test values
else if(initial==1)
{
    int Red = 0;
    int Green = 0;
    int Blue = 0;
    //move to rook location
    xMovement(0,7,1);
    wait1Msec(1000);
    getColorRGB(S3,Red,Green,Blue);
    //make sure its within colour tolerance
    if(!isColor(2,Piece, Red, Green, Blue))
    {
        eraseDisplay();
        displayTextLine(3, "Rook not accurate");
    }
    wait1Msec(3000);
    yMovement(0,1,1);
    wait1Msec(1000);
    getColorRGB(S3,Red,Green,Blue);
    if(!isColor(3,Piece, Red, Green, Blue))
    {
        eraseDisplay();
        displayTextLine(3, "Knight not accurate");
    }
```

```
wait1Msec(3000);
yMovement(1,2,0);
wait1Msec(1000);
getColorRGB(S3,Red,Green,Blue);
if(!isColor(4,Piece, Red, Green, Blue))
{
    eraseDisplay();
    displayTextLine(3, "Bishop not accurate");
}
wait1Msec(3000);
yMovement(2,3,0);
wait1Msec(1000);
getColorRGB(S3,Red,Green,Blue);
if(!isColor(6,Piece, Red, Green, Blue))
{
    eraseDisplay();
    displayTextLine(3, "Queen not accurate");
}
wait1Msec(3000);

yMovement(3,4,0);
wait1Msec(1000);
getColorRGB(S3,Red,Green,Blue);
if(!isColor(5,Piece, Red, Green, Blue))
{
    eraseDisplay();
    displayTextLine(3, "King not accurate");
}
wait1Msec(3000);
```

```
    xMovement(7,6,0);
    wait1Msec(1000);
    getColorRGB(S3,Red,Green,Blue);
    if(!isColor(1,Piece, Red, Green, Blue))
    {
        eraseDisplay();
        displayTextLine(3, "Pawn not accurate");
    }
    wait1Msec(3000);
    //return to 0
    xMovement(5,0,0);
    yMovement(4,0,0);
}

else//regular sensing, see our full code for more info
{
    nMotorEncoder[motorA] = 0;
    nMotorEncoder[motorD] = 0;
    int EndX = 0,EndY = 0;
    int TotalPiece = 0;
    int Negative = 1;
    for(int Row = 0;
    Row< BoardSize&&TotalPiece<Piece.WhitePiece; Row++)
    {
        if(Row > 0)
        {
            xMovement(Row-1, Row,0);
        }
        for(int Col = 0;
```

```
Col<BoardSize&&TotalPiece<Piece.WhitePiece; Col++)
{
    if(Col > 0)
    {
        if(Row%2==0)
        {
            yMovement(Col-1, Col, 0);
        }
        else
        {
            yMovement(BoardSize-Col,7-Col, 0);
        }

    }
    int Red = 0, Green = 0, Blue = 0;
    wait1Msec(300);
    getColorRGB(S3,Red,Green,Blue);
    if(isColor(2,Piece,Red,Green,Blue))
    {
        TotalPiece++;
        Piece.whitecoords[Row][Col] = 'R';
    }
    else if(isColor(1,Piece,Red,Green,Blue))
    {
        TotalPiece++;
        Piece.whitecoords[Row][Col] = 'P';
    }
    else if(isColor(3,Piece,Red,Green,Blue))
    {
```

```
            TotalPiece++;
            Piece.whitecoords[Row][Col] = 'H';
        }
        else if(isColor(4,Piece,Red,Green,Blue))
        {
            TotalPiece++;
            Piece.whitecoords[Row][Col] = 'B';
        }
        else if(isColor(6,Piece,Red,Green,Blue))
        {
            TotalPiece++;
            Piece.whitecoords[Row][Col] = 'Q';
        }
        else if(isColor(5,Piece,Red,Green,Blue))
        {
            TotalPiece++;
            Piece.whitecoords[Row][Col] = 'K';
        }


        else
        {
            Piece.whitecoords[Row][Col] = ' ';
        }
        if(TotalPiece == Piece.WhitePiece)
        {
            EndY = Col;
        }
    }
```

```
            if(TotalPiece == Piece.WhitePiece)
            {
                EndX = Row;
            }

        }
        xMovement(EndX,0,0);
        yMovement(EndY,0,0);
    }
}
task main()
{

    ColourValue testObj;
    //does an inital sense
    scanBoard(0, testObj);
    //checks if those values are good
    scanBoard(1, testObj);
    //times our full scan
    time1[T1] = 0;
    scanBoard(2,testObj);
    eraseDisplay();
    //displays the time
    displayBigTextLine(3, "%f s", time1[T1]/1000.0);
    wait1Msec(5000);
}
```

Timer Test:

```
int checkButtonPress(bool difficulty)
```

```
{
    while(!getButtonPress(buttonAny))
    {}
    int Check = 0;
    if(getButtonPress(buttonLeft))
    {
        Check = 1;
    }
    //only 2 difficulties compared to 3 timer settings
    else if(getButtonPress(buttonUp)&&! difficulty)
    {
        Check = 0;
    }

    else if(getButtonPress(buttonRight))
    {
        Check = 2;

    }
    else
    {
        Check = 4;
    }
    while(getButtonPress(buttonAny))
    {}
    return Check;
}
void countdown()
{
```

```
    time1[T2] = 0;
    while(time1[T2]<5000)
    {
        displayBigTextLine(1, "%d SECONDS",(5000-time1[T2])/1000);
    }

}

task main()
{
    SensorType[S1] = sensorEV3_Touch;
    wait1Msec(50);
    //timer constant has been shortened to 1 minute
    //to allow for quick demonstration of end game scenario
    const int Time = 60*1000;
    int Timer = 4;
    int Difficulty = 4;
    eraseDisplay();
    countdown();
    //gets timer value, timer will be 4 if invalid button pressed
    while(Timer == 4)
    {
        displayTextLine(1,"Enter the Timer Setting:");
        displayTextLine(2,"Up for no Limit");
        displayTextLine(3,"Left for 1 minute");
        displayTextLine(4,"Right for 2 minute");
        Timer = checkButtonPress(false);
    }
    eraseDisplay();
```

```
//set max timer
Timer *= Time;
//get difficulty value
while (Difficulty == 4)
{
    displayTextLine(2,"Enter the Difficulty");
    displayTextLine(3,"Left for beginner");
    displayTextLine(4,"Right for Regular");
    Difficulty = checkButtonPress(true);
}
eraseDisplay();
countdown();
eraseDisplay();
bool WhiteKing = true,BlackKing = true,StaleMate = false;
int LastTime = 0;
while(WhiteKing&&BlackKing&&!StaleMate)
{
    time1[T1] = 0;
    displayTextLine(1,"Please make your move");
    displayTextLine(3,"Tap touch sensor when done");
    wait1Msec(5000);
    eraseDisplay();
    countdown();
    eraseDisplay();
    //infinite time chosen
    if(Timer == 0)
    {
        while(SensorValue[S1] == 0)
        {}
```

```
    }
    else
    {
        //loops until sensor pressed or time ends
        while(SensorValue[S1]==0&&(time1[T1]+LastTime)<Timer)
        {
            //calculation for minutes
            displayBigTextLine(1,"%d minutes",
            ((Timer-(time1[T1]+LastTime))/60000));
            //calculation for seconds
            displayBigTextLine(3,"%d seconds",
            ( ( ( (Timer-(time1[T1]+LastTime) )/60000.0) –
            ( (Timer-(time1[T1]+LastTime) )/60000) ) *60));
        }
        //if timer up set white king to be dead
        if(time1[T1]+LastTime>=Timer)
        {
            WhiteKing = false;
        }

        else
        {
            //do nothing because this code was only used to
           //show our program ending when timer ran out
        }

    }
    eraseDisplay();
}
```

```
    //end game statements
    if(!WhiteKing)
    {
        displayBigTextLine(1,"You");
        displayBigTextLine(3,"Win");
        displayBigTextLine(5,":(");
    }
    else if(!BlackKing)
    {
        displayBigTextLine(1,"I");
        displayBigTextLine(3,"Win");
        displayBigTextLine(3,":P");
    }
    else
    {
        displayBigTextLine(1,"Tie :|");
        displayBigTextLine(3,"GG");
    }
    wait1Msec(5000);
}
```

RobotC File IO:

```
#include "PC_FileIO.c"
task main()
{
    //setting up files to be handled
    TFileHandle FIn;
    TFileHandle FOut;
    openReadPC(FIn,"chessmove.txt");
    int WhiteKing = true;
```

```
int BlackKing = true;
bool StaleMate = false;
int Attack = 0;
string Input = "";
//keep checking will the file is empty
while(Input == "")
{
    readTextPC(FIn, Input);
}
//display
displayString(1,"Success Exit");
//conversion as stalemate is a bool
StaleMate = atoi(Input);
readIntPC(FIn,WhiteKing);
readIntPC(FIn,BlackKing);
readIntPC(FIn,Attack);
//make sure all values are saved properly
displayString(2,"%d",StaleMate);
displayString(3,"%d",WhiteKing);
displayString(4,"%d",BlackKing);
displayString(5,"%d",Attack);
int CurX = 0,CurY = 0, NewX = 0, NewY = 0;
readIntPC(FIn,CurX);
readIntPC(FIn,CurY);
readIntPC(FIn,NewX);
readIntPC(FIn,NewY);
//make sure all values are saved properly
displayString(6,"%d",CurX);
displayString(7,"%d",CurY);
```

```
    displayString(8,"%d",NewX);
    displayString(9,"%d",NewY);
    //this is where movement woudl take place
    closeFilePC(FIn);
    //clear contents of chessmove
    openWritePC(FOut,"chessmove.txt");
    writeTextPC(FOut,"");
    closeFilePC(FOut);
    //clear contents of aistuff
    openWritePC(FOut,"aiStuff.txt");
    writeTextPC(FOut,"");
    closeFilePC(FOut);
    if(Attack==1)
    {
        //this is where the pc would wait for the piece to be moved
        displayString(1,"Attack Needed");
    }
    //so data stays on screen until button is pressed
    while(!getButtonPress(buttonAny))
    {}
}
```

C++ Clicker Code:

All background info taken from [2] [3] [4] [5] [6] [7] [8] [9] [10]

```
#include<windows.h>
#include<stdio.h>
#include <cmath>
#include <fstream>
#include <iostream>
```

```
using namespace std;
//This checks if the file is empty in a consistent manner
bool is_empty(ifstream& pFile)
{
    return pFile.peek() == ifstream::traits_type::eof();
}

void click(int x,int y)
{
    Sleep(20);
    SetCursorPos(x,y);
    Sleep(40);
    mouse_event(MOUSEEVENTF_LEFTDOWN |
     MOUSEEVENTF_LEFTUP, 0, 0, 0, 0);
}
int main(int argc, char** argv)
{
    //initial opening of the file management directory
    cin.get();//wait for enter to start
    click(139,36);
    Sleep(200);
    click(157,237);
    Sleep(200);
    click(443,260);
    Sleep(3000);
    //double click to go to rc-data
    click(366,375);
    mouse_event(MOUSEEVENTF_LEFTDOWN |
    MOUSEEVENTF_LEFTUP, 0, 0, 0, 0);
```

```
Sleep(200);
//click on the proper file to download
click(375,300);
//program runs until the user clicks the space key
//which they'll do at the end of the game
while(!(GetAsyncKeyState(VK_SPACE) & 0x80000000))
{
    //download the file to the computer
    Sleep(200);
    click(772,285);
    Sleep(500);
    click(983,683);
    Sleep(600);
    //check if the file is empty
    ifstream fin("aistuff.txt");
    if(!is_empty(fin))
    {
        fin.close();
        ShellExecute(NULL, "open", "chessai.exe",
        NULL, NULL, SW_SHOWDEFAULT);
        bool loop = false;
        while(!loop)
        {
            try
            {
                //wait for c# generated file to contain values
                fin.open("chessmove.txt");
                if(!is_empty(fin))
                {
```

```
            loop = true;
        }
        cout << "empty"<<endl;
        fin.close();
    }
    catch (exception& e)
    {
        fin.close();
        cout << "no file yet"<<endl;
    }
}
    /* deletes the robotc file from pc so we don't need to
     deal */
    // with overwriting the file and extra clicks
    remove("aistuff.txt");
    //uploads the c# file to the robot
    Sleep(200);
    click(873,277);
    Sleep(2000);
    click(550,345);
    Sleep(2000);
    click(1125,702);
    Sleep(5000);
    //remove the c# file from the pc as
    //to not deal with overwrite
    remove("chessmove.txt");
    //a turn will not take less than 2 minute
    //so wait atleast 2 minute, as per constraints
    Sleep(120000);
```

```
        }
        else
        {
            //wait 5 seconds then try opening the file again
            fin.close();
            remove("aistuff.txt");
            Sleep(5000);
        }
    }
    return 0;
}
```

Appendix B

Main Code:

```
#include "PC_FileIO.c"
const int BoardSize = 8;
typedef struct
{
    //decided to use arrays that way we didnt have 3
    //seperate ints for each type
    int RookCol[3];
    int PawnCol[3];
    int KingCol[3];
    int QueenCol[3];
    int BishopCol[3];
    int KnightCol[3];
    int WhitePiece;
    char whitecoords[BoardSize][BoardSize];
}ColourValue;
```

```
const int Tol = 5;
const float SquareSize = 6.35;
const float xMovementconst = SquareSize*(360/(3.97*PI));
const float yMovementconst = SquareSize*(360/(3.55*PI));
const int XYPower = 15;

int checkButtonPress(bool difficulty)
{
    while(!getButtonPress(buttonAny))
    {}
    int Check = 0;
    if(getButtonPress(buttonLeft))
    {
        Check = 1;
    }
    else if(getButtonPress(buttonUp)&&!difficulty)
    {
        Check = 0;
    }


    else if(getButtonPress(buttonRight))
    {
        Check = 2;
    }
    else
    {
        //invalid input
        Check = 4;
```

```
    }
    while(getButtonPress(buttonAny))
    {}
    return Check;
}

void countdown()
{
    time1[T2] = 0;
    while(time1[T2]<5000)
    {
        displayBigTextLine(1, "%d SECONDS",(5000-time1[T2])/1000);
    }
}

void xMovement(float oldX, float newX, bool reset)
{
    int Negative = 1;
    float XChange = newX-oldX;
    //the only purpose of XChange is to determine
   //if the movement is negative
    if(XChange<0)
    {
        Negative = -1;
        XChange *= Negative;
    }
    if(reset)
    {
        //this reset part basically happens only
```

```
    //if the move being made isnt a consecutive one
    nMotorEncoder[motorD] = 0;
}
motor[motorD] = XYPower*Negative;
/*
because newx is always positive it made sense to separate
our movement into
two separate while loops which handles movement
approaching zero from both ends
*/
if(Negative>0)
{
    /*
    as a general note we use newX as we
    take all our calculations from 0 to the X value
    as opposed to the last x value to the new X value
    */
    while(nMotorEncoder[motorD]<newX*xMovementconst)
    {
        //displayBigTextLine(1,"%d",nMotorEncoder[motorD]);
        //both above and below are testing code
        //displayBigTextLine(3,"%f",newX*xMovementconst);
    }
}
else
{
    while(nMotorEncoder[motorD]>newX*xMovementconst)
    {
        //displayBigTextLine(1,"%d",nMotorEncoder[motorD]);
```

```
            //tester code
            //displayBigTextLine(3,"%f",newX*xMovementconst);
        }
    }
    motor[motorD] = 0;
}
//same comments as the X value, just different constant and motor
void yMovement(float oldY, float newY, bool reset)
{
    int Negative = 1;
    float YChange = newY-oldY;
    if(YChange<0)
    {
        Negative = -1;
        YChange *= Negative;
    }
    if(reset)
    {
        nMotorEncoder[motorA] = 0;
    }
    motor[motorA] = XYPower*Negative;
    if(Negative>0)
    {
        while(nMotorEncoder[motorA]<newY*yMovementconst)
        {}
    }
    else
    {
        while(nMotorEncoder[motorA]>newY*yMovementconst)
```

```
        {}
    }
    motor[motorA] = 0;
}

void placePiece(bool pickup)
{
    const int ClawPower = 100;
    const int ClawDown = 35;
    const float Height = 9.8;
    const int ClawLiftPower = 5;
    const int OpenDegree = 40;
    const int CloseDegree = -30;
    //this opens the claw
    if(pickup)
    {
        nMotorEncoder[motorB] = 0;
        motor[motorB] = ClawPower;
        /*
        this wait here is mainly because we saw that otherwise
        our motors would not open the claw the full way
        as the motor would spin in its holding container and
        so, this pause dealt with that spin if needed
        however, its not so much that if the motor doesn't spin in
        place it will open it over 40 degrees
        */
        wait1Msec(250);
        while(nMotorEncoder[motorB] < OpenDegree)
        {}
```

```
    motor[motorB] = 0;
}
//this drops the claw down
wait1Msec(500);
nMotorEncoder[motorC] = 0;
motor[motorC] = -ClawDown;
while(abs(nMotorEncoder[motorC])
<(Height)*(yMovementconst/SquareSize))
{}
motor[motorC] = 0;
wait1Msec(1000);
//this closes the claw
if(pickup)
{
    motor[motorB] = -ClawDown;
    /*
    this was our experimentally determined value
    we needed to close the claw
    */
    while(nMotorEncoder[motorB] > CloseDegree)
    {}
    motor[motorB] = 0;
}
//this opens the claw
else
{
    nMotorEncoder[motorB] = 0;
    motor[motorB] = ClawDown;
    wait1Msec(250);
```

```
        while(nMotorEncoder[motorB] < OpenDegree)
        {}
        motor[motorB] = 0;
    }
    //this lifts the claw up
    wait1Msec(1000);
    nMotorEncoder[motorC] = 0;
    motor[motorC] = ClawLiftPower;
    while(nMotorEncoder[motorC]
    < (Height*(yMovementconst/SquareSize)))
    {}
}
void movement(int oldX, int oldY, int newX, int newY)
{
    /*
     pause times are just to allow everything to be more accurate
     move to starting location
     bool of one as we are beginning a consecutive move set and
     so we need to reset the motor encoders
    */
    xMovement(0,oldX,1);
    wait1Msec(250);
    yMovement(0,oldY,1);
    wait1Msec(500);
    //pickup the piece
    placePiece(true);
    //move to new location
    wait1Msec(250);
    xMovement(oldX,newX,0);
```

```
    wait1Msec(250);
    yMovement(oldY,newY,0);
    wait1Msec(500);
    //drop the piece
    placePiece(false);
    //move back to the origin
    wait1Msec(250);
    xMovement(newX,0,0);
    wait1Msec(250);
    yMovement(newY,0,0);
    wait1Msec(250);
}
bool isColor(int Chosen, ColourValue & Piece, int Red, int Green, int
Blue)
{
    // Pawn= 1, Rook= 2, Knight= 3, Bishop= 4, King= 5 , Queen= 6
    /*
    simple check if the given values are within the
    tolerance values for each of our systems
     */


     if(Chosen == 1)
    {
        return (abs(Piece.PawnCol[0]-Red)<Tol&&
        abs(Piece.PawnCol[1]-Green)<Tol &&
        abs(Piece.PawnCol[2]-Blue)<Tol);
    }
    else if(Chosen == 2)
    {
```

```
    return (abs(Piece.RookCol[0]-Red)<Tol&&
    abs(Piece.RookCol[1]-Green)<Tol &&
    abs(Piece.RookCol[2]-Blue)<Tol);
}
else if(Chosen == 3)
{
    return (abs(Piece.KnightCol[0]-Red)<Tol&&
    abs(Piece.KnightCol[1]-Green)<Tol &&
    abs(Piece.KnightCol[2]-Blue)<Tol);
}
else if(Chosen == 4)
{
    return (abs(Piece.BishopCol[0]-Red)<Tol&&
    abs(Piece.BishopCol[1]-Green)<Tol &&
    abs(Piece.BishopCol[2]-Blue)<Tol);
}
else if(Chosen == 5)
{
    return (abs(Piece.KingCol[0]-Red)<Tol&&
    abs(Piece.KingCol[1]-Green)<Tol &&
    abs(Piece.KingCol[2]-Blue)<Tol);
}
else if(Chosen == 6)
{
    return (abs(Piece.QueenCol[0]-Red)<Tol&&
    abs(Piece.QueenCol[1]-Green)<Tol &&
    abs(Piece.QueenCol[2]-Blue)<Tol);
}
//only reached if none of the if statements fit
```

```
    return false;
}

void scanBoard(bool initial, ColourValue & Piece)
{
    Piece.WhitePiece = 16;
    /* these constants are basically the precise distances that
    we need to move to place the colour sensor over the centre
    of the square */
    const int ColourX = 6.8;
    const int ColourY = -3;
    /* divide by Square Size to cancel it out within the
    X and Y movement constant
    this moves the colour sensor to the centre of the square
    */
    yMovement(0,ColourX/SquareSize,0);
    xMovement(0,ColourY/SquareSize,0);

    if(initial)
    {
        //fill up all the empty parts of the array
        for(int XValue = 0; XValue< BoardSize-2; XValue++)
        {
            for(int YValue = 0; YValue<BoardSize; YValue++)
            {
                Piece.whitecoords[XValue][YValue] = ' ';
            }
        }
        //fill in the pawn row in the array
```

```
for(int YValue = 0; YValue<BoardSize; YValue++)
{
    Piece.whitecoords[6][YValue] = 'P';
}
Piece.whitecoords[7][0] = 'R';
Piece.whitecoords[7][1] = 'H';
Piece.whitecoords[7][2] = 'B';
Piece.whitecoords[7][3] = 'Q';
Piece.whitecoords[7][4] = 'K';
Piece.whitecoords[7][5] = 'B';
Piece.whitecoords[7][6] = 'H';
Piece.whitecoords[7][7] = 'R';
Piece.WhitePiece = 16;

//initial rook sense
xMovement(0,7,1);
//bool of 1 signifies start of consecutive moves
wait1Msec(1000);
//pause time is just for consistent moving and sensing
getColorRGB(S3,Piece.RookCol[0],
Piece.RookCol[1],Piece.RookCol[2]);
displayBigTextLine(1,"%d",Piece.RookCol[0]);
displayBigTextLine(3,"%d",Piece.RookCol[1]);
displayBigTextLine(5,"%d",Piece.RookCol[2]);
wait1Msec(1000);

//initial knight sense
yMovement(0,1,1);
wait1Msec(1000);
```

```
getColorRGB(S3,Piece.KnightCol[0],
Piece.KnightCol[1], Piece.KnightCol[2]);
displayBigTextLine(1,"%d",Piece.KnightCol[0]);
displayBigTextLine(3,"%d",Piece.KnightCol[1]);
displayBigTextLine(5,"%d",Piece.KnightCol[2]);
wait1Msec(1000);

//initial bishop sense
yMovement(1,2,0);
wait1Msec(1000);
getColorRGB(S3,Piece.BishopCol[0],
Piece.BishopCol[1],Piece.BishopCol[2]);
displayBigTextLine(1,"%d",Piece.BishopCol[0]);
displayBigTextLine(3,"%d",Piece.BishopCol[1]);
displayBigTextLine(5,"%d",Piece.BishopCol[2]);
wait1Msec(1000);

//initial queen sense
yMovement(2,3,0);
wait1Msec(1000);
getColorRGB(S3,Piece.QueenCol[0],
Piece.QueenCol[1],Piece.QueenCol[2]);
displayBigTextLine(1,"%d",Piece.QueenCol[0]);
displayBigTextLine(3,"%d",Piece.QueenCol[1]);
displayBigTextLine(5,"%d",Piece.QueenCol[2]);
wait1Msec(1000);
//initial king sense
yMovement(3,4,0);
wait1Msec(1000);
```

```
    getColorRGB(S3,Piece.KingCol[0],
    Piece.KingCol[1],Piece.KingCol[2]);
    displayBigTextLine(1,"%d",Piece.KingCol[0]);
    displayBigTextLine(3,"%d",Piece.KingCol[1]);
    displayBigTextLine(5,"%d",Piece.KingCol[2]);
    wait1Msec(1000);
    //inital pawn sense
    xMovement(7,6,0);
    wait1Msec(1000);
    getColorRGB(S3,Piece.PawnCol[0],
    Piece.PawnCol[1],Piece.PawnCol[2]);
    displayBigTextLine(1,"%d",Piece.PawnCol[0]);
    displayBigTextLine(3,"%d",Piece.PawnCol[1]);
    displayBigTextLine(5,"%d",Piece.PawnCol[2]);
    wait1Msec(1000);

    //reset the arm back to zero
    xMovement(6,0,0);
    yMovement(4,0,0);
}

else
{
    /* this saves us an if statement to start
     with a bool of 1 for our movement on the very
     first x and y move
    */
    nMotorEncoder[motorA] = 0;
    nMotorEncoder[motorD] = 0;
```

```
int TotalPiece = 0;
int EndY = 0, EndX = 0;
for(int XValue = 0;
XValue< BoardSize&&TotalPiece<Piece.WhitePiece; XValue++)
{
    //moves the system up one in the x value
    if(XValue > 0)
    {
        xMovement(XValue-1, XValue,0);
    }
    for(int YValue = 0;
    YValue< BoardSize&&TotalPiece<Piece.WhitePiece; YValue++)
    {
        /*
        this basically covers the snaking so
        if we end on the right end of the board
        we move to the left and vise versa
       */
        if(YValue > 0)
        {

            if(XValue%2==0)
            {
                yMovement(YValue-1, YValue, 0);
            }
            else
            {
                yMovement(BoardSize-YValue,7-YValue, 0);
            }
```

```
}
//This deals with determining what the pieces are
int Red = 0, Green = 0, Blue = 0;
wait1Msec(300);
getColorRGB(S3,Red,Green,Blue);
if(isColor(2,Piece,Red,Green,Blue))
{
    TotalPiece++;
    Piece.whitecoords[XValue][YValue] = 'R';
}
else if(isColor(1,Piece,Red,Green,Blue))
{
    TotalPiece++;
    Piece.whitecoords[XValue][YValue] = 'P';
}
else if(isColor(3,Piece,Red,Green,Blue))
{
    TotalPiece++;
    Piece.whitecoords[XValue][YValue] = 'H';
}
else if(isColor(4,Piece,Red,Green,Blue))
{
    TotalPiece++;
    Piece.whitecoords[XValue][YValue] = 'B';
}
else if(isColor(6,Piece,Red,Green,Blue))
{
    TotalPiece++;
```

```
            Piece.whitecoords[XValue][YValue] = 'Q';
        }
        else if(isColor(5,Piece,Red,Green,Blue))
        {
            TotalPiece++;
            Piece.whitecoords[XValue][YValue] = 'K';
        }
        else
        {
            /* we set all non pieces to '  '
            that way if a piece has moved we
            aren't still storing its value */
            Piece.whitecoords[XValue][YValue] = ' ';
        }
        //this just stores where the arm ended sensing
        if(TotalPiece == Piece.WhitePiece)
        {
            EndY = YValue;
        }
    }
    if(TotalPiece == Piece.WhitePiece)
    {
        EndX = XValue;
    }

}
//reset the claw back to zero from wherever it left off
xMovement(EndX,0,0);
yMovement(EndY,0,0);
```

```
    }
    //moves the claw back to the centre of the claw
    yMovement(ColourX/SquareSize,0,0);
    xMovement(ColourY/SquareSize,0,0);
}


void fileIO(ColourValue & Values,int & Difficulty, bool & StaleMate,
bool & WhiteKing, bool & BlackKing, int & Attack,int & CurX, int &
CurY, int & NewX, int & NewY)
{
    TFileHandle FOut;
    TFileHandle FIn;
    openWritePC(FOut,"aistuff.txt");


    writeLongPC(FOut,Difficulty);
    writeEndlPC(FOut);
    //write out the white boardstate to the file
    for(int XValue = 0; XValue<BoardSize;XValue++)
    {
        for(int YValue = 0;YValue<BoardSize;YValue++)
        {
            /*
            this basically writes out a * for an empty
            location because it makes it easy for the C#
            code to read it in this way
            */
            if(Values.whitecoords[XValue][YValue] == ' ')
            {
                writeCharPC(FOut,'*');
            }
```

```
        else
        {
            writeCharPC(FOut,Values.whitecoords[XValue][YValue]);
        }
    }
    //formatting of the file
    writeEndlPC(FOut);
}

closeFilePC(FOut);
openReadPC(FIn,"chessmove.txt");
/* these values are here due to there not being a
proper read for bools so we first read into integers
and then convert them to bools
*/
int WhiteKingI = 0;
int BlackKingI = 0;
/* we use a string of input because our
first value is a bool so a 1 or a 0
so, when we read it will be unable to tell
if we have no value there at all or if the
bool is just zero. works the same way if we had an integer
and set it to -1
*/
string Input = "";
//loops while the file is empty
while(Input == "")
{
    readTextPC(FIn, Input);
```

```
    }
    /* converting our string to integer which can then be
    Implicitly converted to bool
    */
    StaleMate = atoi(Input);
    readIntPC(FIn,WhiteKingI);
    readIntPC(FIn,BlackKingI);
    readIntPC(FIn,Attack);
    readIntPC(FIn,CurX);
    readIntPC(FIn,CurY);
    readIntPC(FIn,NewX);
    readIntPC(FIn,NewY);
    closeFilePC(FIn);
    //empties the chessmove.txt
    openWritePC(FOut,"chessmove.txt");
    writeTextPC(FOut,"");
    closeFilePC(FOut);
    //empties the aistuff.txt
    openWritePC(FOut,"aistuff.txt");
    writeTextPC(FOut,"");
    closeFilePC(FOut);

    WhiteKing = WhiteKingI;
    BlackKing = BlackKingI;
}

task main()
{
    SensorType[S1] = sensorEV3_Touch;
```

```
ColourValue CurrentStuff;
/*
we don’t actually need to setup our colour sensor
because we use the getRGB values
*/
wait1Msec(50);
const int Time = 15*60*1000;
/*they both start as 4 because the checkbuttonpress
will return invalid as 4
as such we automatically account for invalid button presses
*/
int Timer = 4;
int Difficulty = 4;
eraseDisplay();
countdown();
while(Timer == 4)
{
    displayTextLine(1,"Enter the Timer Setting:");
    displayTextLine(2,"Up for no Limit");
    displayTextLine(3,"Left for 15 minute");
    displayTextLine(4,"Right for 30 minutes");
    Timer = checkButtonPress(false);
}
eraseDisplay();
Timer *= Time;

while (Difficulty == 4)
{
    displayTextLine(2,"Enter the Difficulty");
```

```
    displayTextLine(3,"Left for beginner");
    displayTextLine(4,"Right for Regular");
    Difficulty = checkButtonPress(true);
}
eraseDisplay();
countdown();
eraseDisplay();
//initial scan
scanBoard(true, CurrentStuff);

bool WhiteKing = true,BlackKing = true,StaleMate = false;
int LastTime = 0;
//main while loop, checks whether or not
//an end game state is reached
while(WhiteKing&&BlackKing&&!StaleMate)
{
    displayTextLine(1,"Please make your move");
    displayTextLine(3,"Tap touch sensor when done");
    wait1Msec(5000);
    eraseDisplay();
    countdown();
    time1[T1] = 0;
    eraseDisplay();
    //if our timer is set to infinite

     if(Timer == 0)
    {
        while(SensorValue[S1] == 0)
        {}
```

```
}
else
{
    /* we add on last time as that way
    we can start where the timer last left off
     */
    while(SensorValue[S1]==0&&(time1[T1]+LastTime)<Timer)
    {
        displayBigTextLine(1,"%d minutes",
        ((Timer-(time1[T1]+LastTime))/60000));
         //displaying exact number of minutes
        //seconds calculations
        displayBigTextLine(3,"%d seconds",
        ( ( ( (Timer-(time1[T1]+LastTime) )/60000.0)
         - ((Timer-(time1[T1]+LastTime) )/60000) ) *60));
    }
    //add on wherever the person ended their turn
    LastTime +=time1[T1];
}
eraseDisplay();
/*the greater then accounts for perhaps if
it takes the program 1 millisecond to add to
LastTime we handle that case
 !=0 is because it will not take the person less than
 a millisecond to make a move and so we ignore
 all the infinite time setups
*/
if(LastTime>=Timer&&LastTime!=0)
{
```

```
    /* the player loses if their timer runs out
   so, there's no point even doing the ai move
   */
    WhiteKing = false;
}
else
{
    //does a regular scan
    scanBoard(false,CurrentStuff);
    eraseDisplay();
    int Attack = 0, CurX = 0, CurY = 0, NewX = 0, NewY = 0;
    fileIO(CurrentStuff,Difficulty,StaleMate,
    WhiteKing,BlackKing,Attack,
    CurX,CurY,NewX,NewY);
    /* the arm won't move the pieces out
    of the way so we make the user do so
    */
    if(Attack==1)
    {
        displayTextLine(1,"Please remove pieces");
        displayTextLine(2,"At %d,%d",NewX,NewY);
        displayTextLine(3,"Then touch touch");
        while(SensorValue[S1]==0)
        {}
        //because attack = 1 we know a white piece will die so
       //we subtract one from their total pieces
        CurrentStuff.WhitePiece--;
        eraseDisplay();
    }
```

```
            //make the move
            movement(CurX,CurY,NewX,NewY);
        }
    }
    //little end statements before the end of the code
    if(!WhiteKing)
    {
        displayBigTextLine(1,"You");
        displayBigTextLine(3,"Win");
        displayBigTextLine(5,":(");
    }
    else if(!BlackKing)
    {
        displayBigTextLine(1,"I");
        displayBigTextLine(3,"Win");
        displayBigTextLine(3,":P");
    }
    else
    {
        displayBigTextLine(1,"Tie :|");
        displayBigTextLine(3,"GG");
    }
}
```

Appendix C

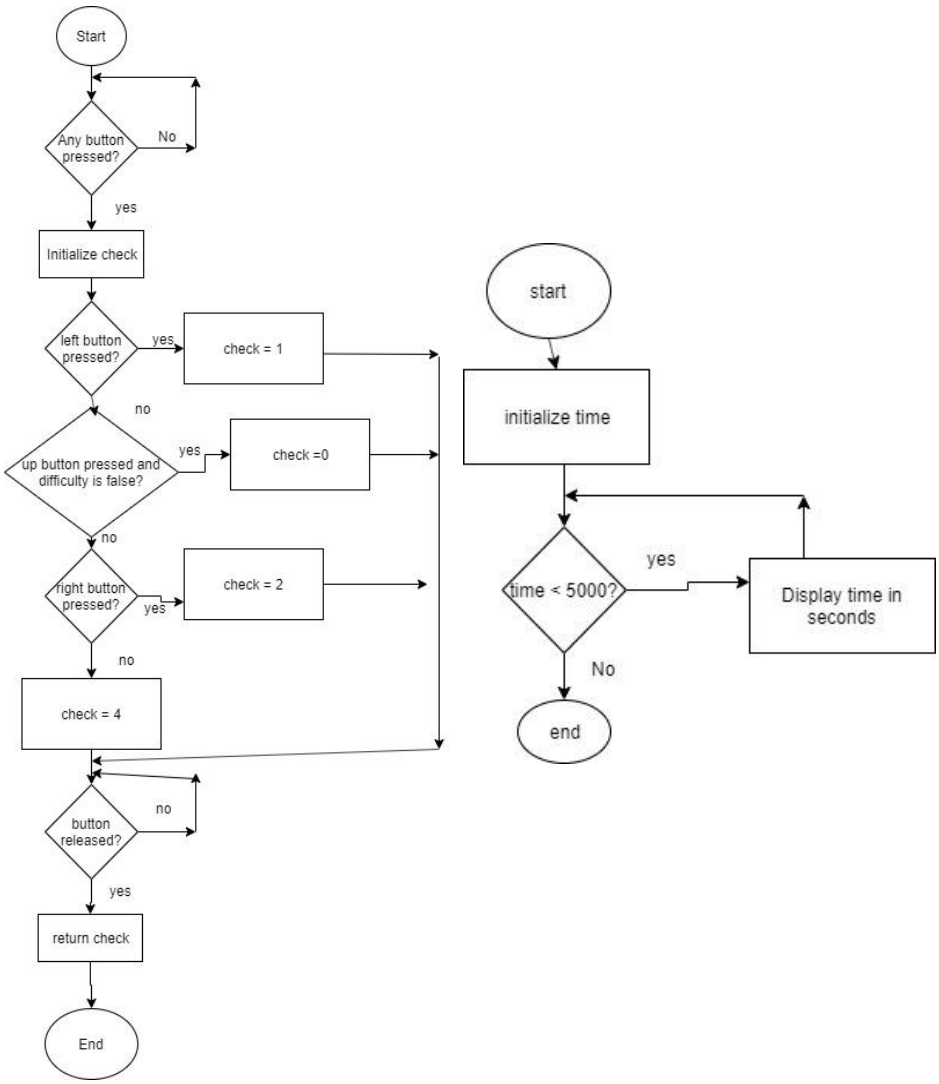


Figure 15:Button press function and countdown timer function (Robot C)

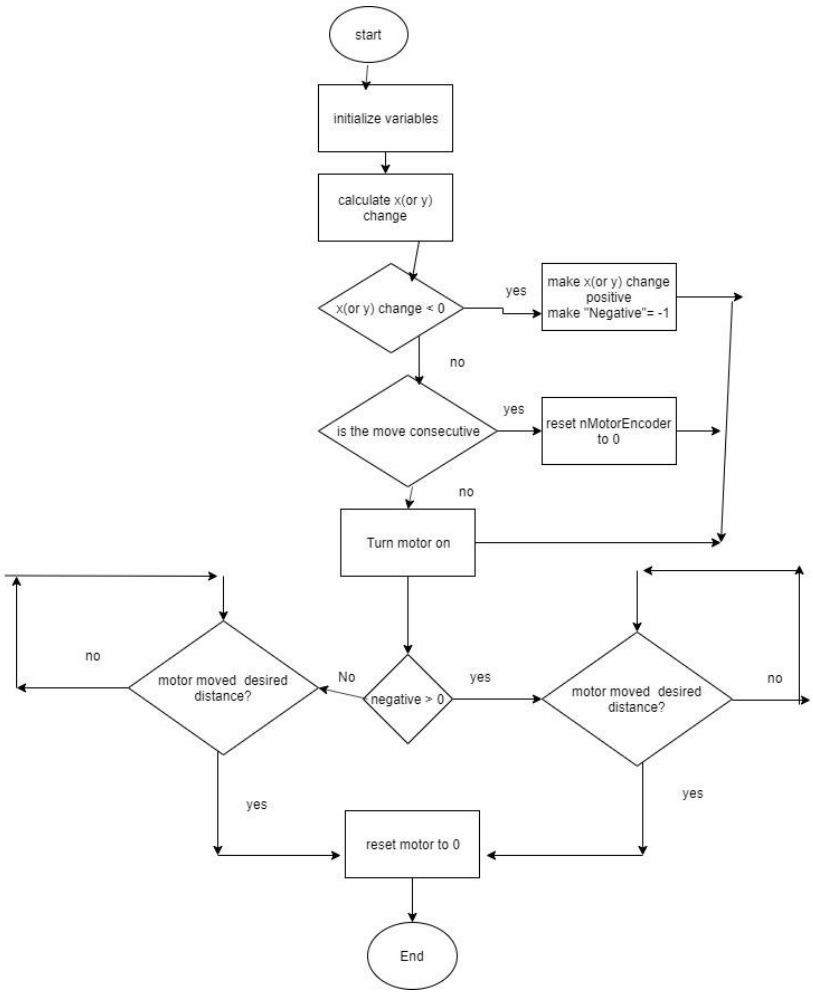


Figure 16: Function for either x or the y movement (Robot C)

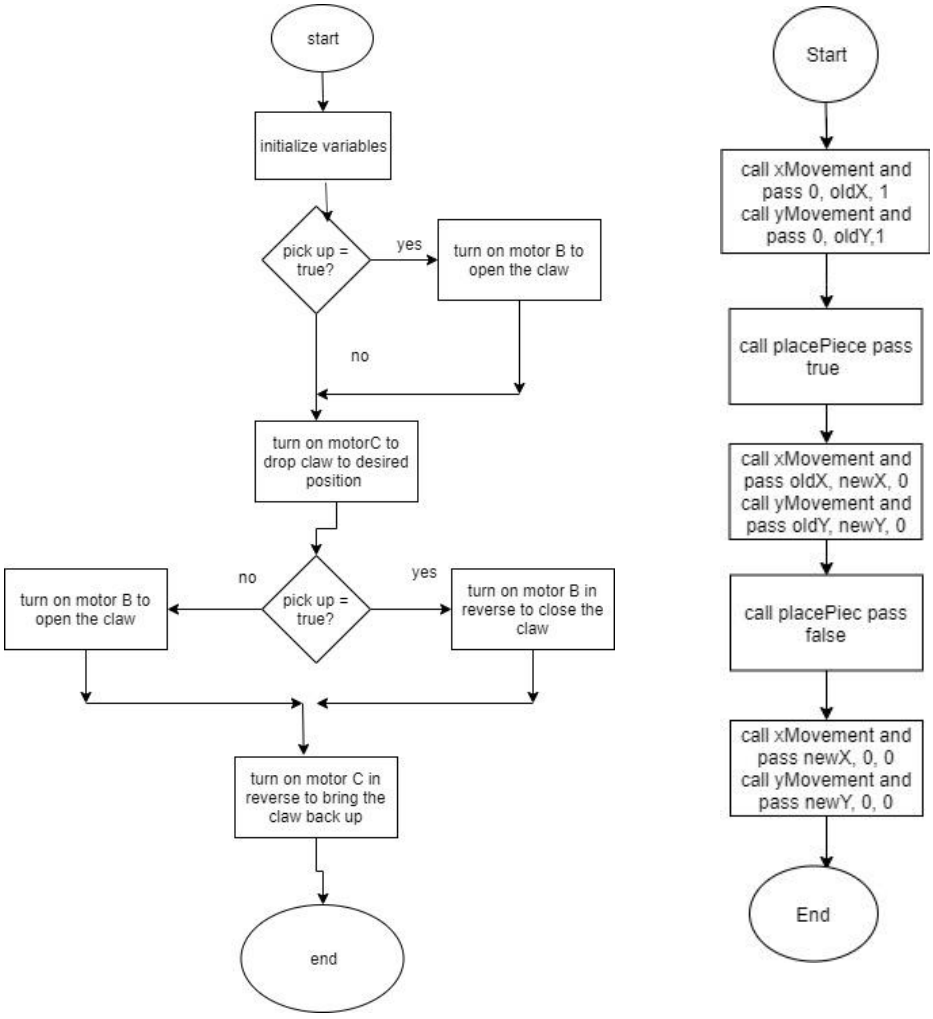


Figure 17: Function for picking up a piece and movement function (Robot C)

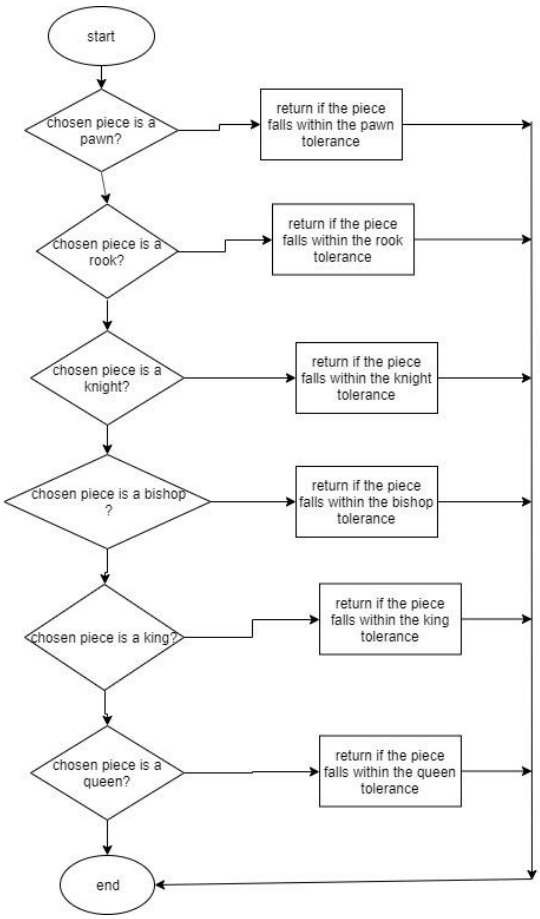


Figure 18: Function for checking pieces (Robot C)

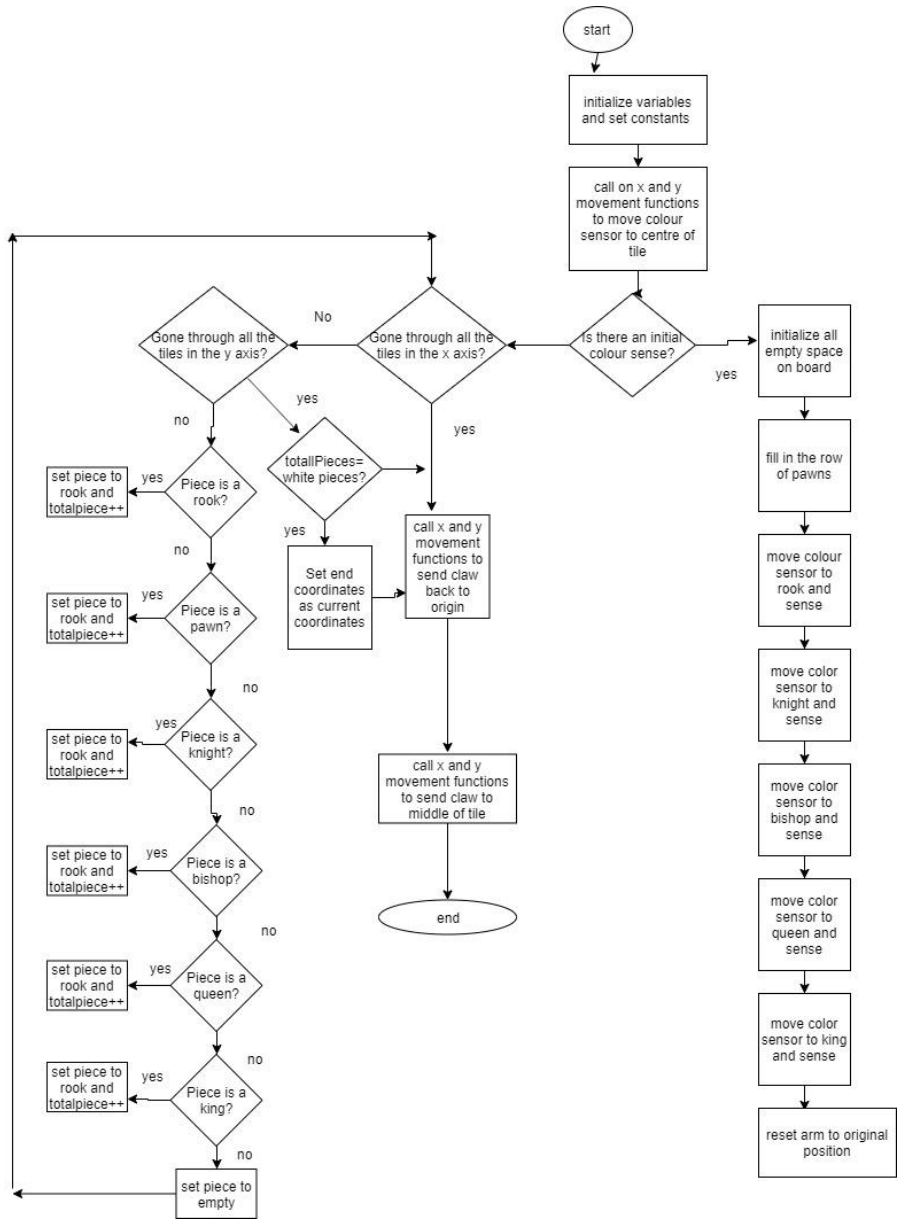


Figure 19: Function for scanning the board (Robot C)

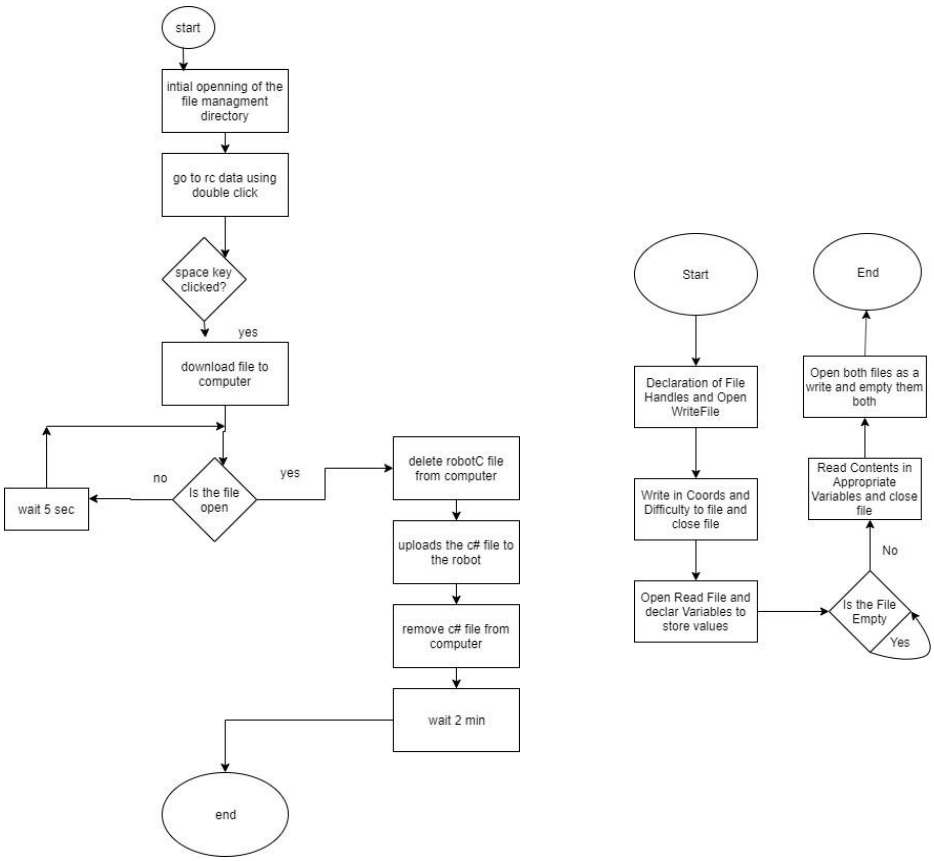


Figure 20: Main function of C++ and function for accessing and using file IO

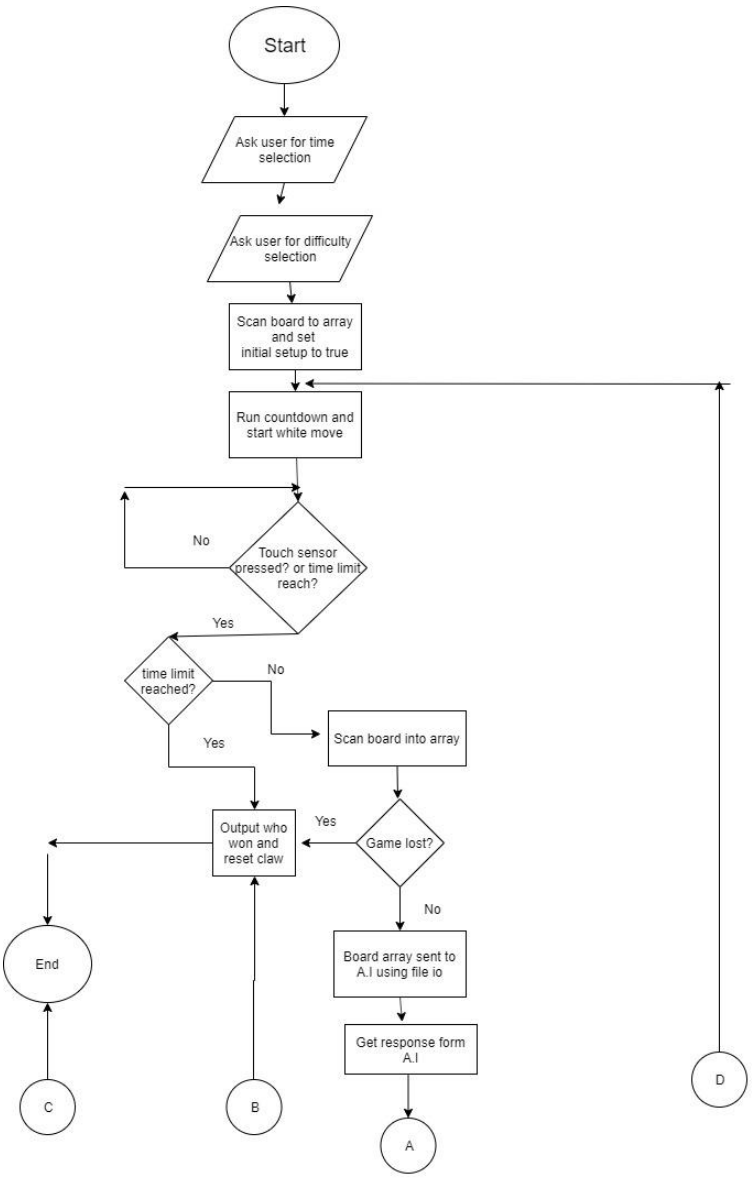


Figure 21: Main function (Robot C) part 1

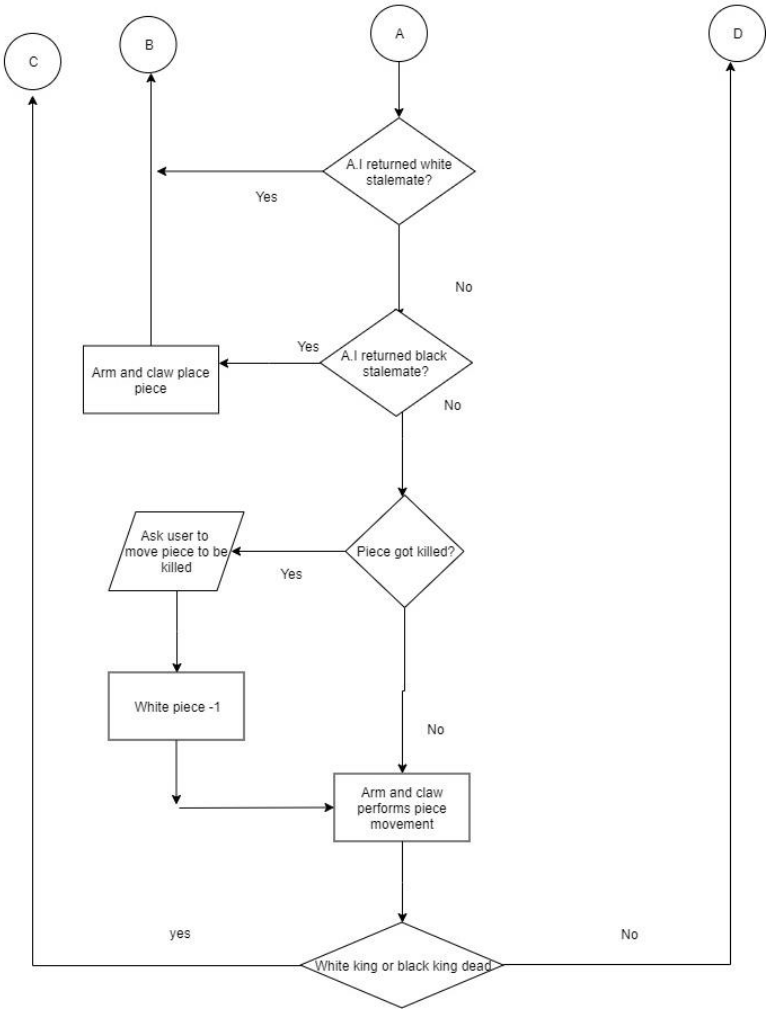


Figure 22: Main function (Robot C) part 2