Carl-Bot

The Chess Grandmaster

Eidan Erlich (21020751), Owen Moogk (21006256),

Richard Wang (21007917), Shabd Gupta (21000124)

December 6, 2022

Table of Contents

Table of Figures

Acknowledgements

The team would like to thank the MTE 100 & 121 TA team for assistance with this project. Throughout working, their assistance and feedback greatly improved the project outcome, and helped us overcome roadblocks and challenges along the way. The team would like to specifically thank Parker Lawrence-Valeriani and Sabiha Akter for their continued constructive feedback that greatly improved our project. Additionally, the team would like to acknowledge Simran Sivagnanasegaram for her support in supplying us with the necessary parts.

Introduction

As a group of chess enthusiasts, the team decided that an interesting application of a modified EV3 robot would be to play a game of chess. As such, the team set out to build a robot that can execute chess moves. Although specific goals with this project have shifted through the development of the robot, the main goal was always the same; to play a game of chess interfaced only by the robot.

The team believe this solution has plenty of potential as well, as it could assist people with poor sight or mobility issues to play the game that many people love. This robot could also be integrated with an AI, providing an intelligent way to train and improve, while playing a physical game of 'over the board' chess.

Scope

The main functionality that the team was looking to complete with this robot was to be able to play a game of chess. A two-axis system was wanted that would be able to pick up and move pieces around on the board, according to a user input.

In earlier stages of the project, the team wanted functionality to input and output files containing game states, which would allow users to replay past games, as well as play certain chess puzzles. Although this is still an intriguing possibility, it was proven to be beyond the scope of the project and not crucial to our minimum viable product, so the team decided to put the idea on the shelf during this development phase.

This robot has the functionality to take inputs from its environment to obtain information about the state of the system, as well as get input from the user to further the gameplay. Inputs include a touch sensor, color sensor, EV3 button input, and motor encoders. The buttons are used as the interface between the user and the robot, receiving commands to be executed by the robot. The touch sensor is used to 'zero' the system. This will bring the robot to a known location, from which moves can be completed precisely and repeatably. The color sensor and motor encoders are used to measure the position of the system on the X-axis and Y-axis respectively. Although our original plan was to use motor encoders for both axes, it was found that the X-axis was slipping, and the motor encoder values were inconsistent, hence the switch to color sensors at incremental locations across the board.

The robot will interact with the environment with multiple motors connected into a 2-axis system. This system can move above the board in 2 dimensions, able to cover all the cells on the board. The two large motors are used to control the two axes, and the medium motor is used to move the claw up and down. The claw is the default Tetrix claw, modified with rubber and tape to increase grip to the variety of chess pieces. When a player requests a move, the robot will navigate to the correct cell, lower the claw, pick up the piece, raise the claw, and then move to the destination location. It will place the piece in a similar manner, and then will 'zero' the system again before prompting for the next move.

The robot will recognize when tasks are complete by using the corresponding color and touch sensors, and the motor encoder values. It will include error checking, so if any system failures occur, the system can react accordingly.

When the game is over, the user can press a button. This will prompt a shut-down procedure, where the robot will re-zero itself and terminate the program.

Mechanical Design and Implementation

The mechanical assembly of the robot is composed of a base and 3 driven systems. The foundation provides structure and support to the rail system, pulley system, and claw system.

Base

The foundation comprises of Tetrix pieces arranged in a square that covers the entire chess board. The dimensions are 0.65 meters in length and 0.58 meters in width, this spans the chess board with additional space on all 4 sides. It includes extra space around the board to keep the captured pieces and accounts for space occupied by the pulley mechanism and EV3 brick. The base has a pillar in each corner, raised in approximately 150 mm up.

Due to the limited supply of resources and materials, each corner pillar is designed differently. Two of the pillars are made using the Tetrix extrusions, a traditional use of pieces. However, the other two pillars are uniquely designed. One of the pillars is made using Tetrix internal connecters rather than the extrusions. The last corner is held up using a wheel, although it makes the structure less stable, it serves its purpose in raising the base.

Additionally, the 4 pillars were secured down to the desk, ensuring that they would not move or shift during operation, leading to error in precision of movement. The overall structure of the base and robot is shown here.

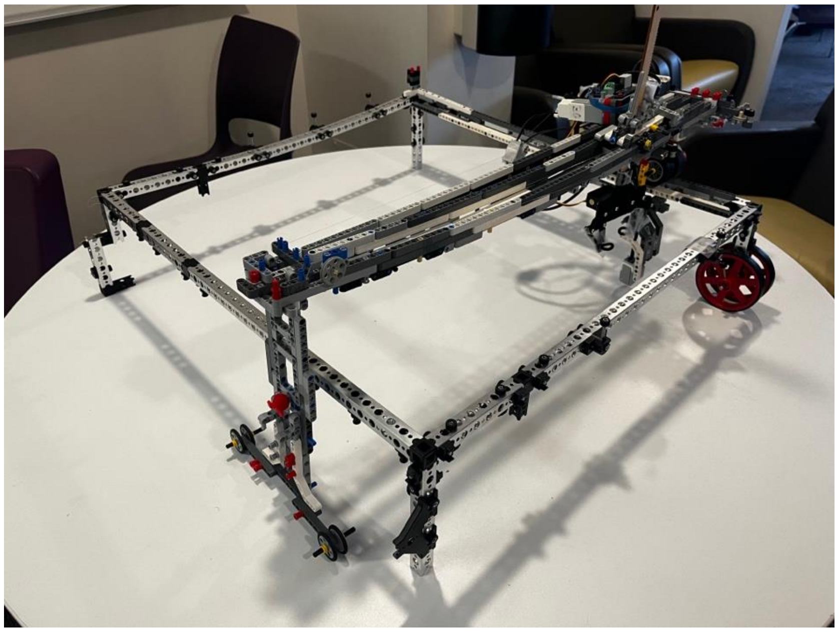

Figure 1: Robot Frame

The purpose of raising the base was to provide clearance for the claw so that it would not knock over chessmen in the board when moving around and moving pieces. The height of the pillars was chosen by identifying the tallest and shortest piece, 96 mm and 48 mm respectively (king and pawn), and analysing the types of moves that will be made. For example, the king (tallest piece), will never move over another piece (due to the rules of chess), so its clearance when being picked up did not need to be as high as other pieces.

The axial movement sits on 2 sets of rails. Both of which are connected to the base. The rails are made of Lego beams and are connected to the aluminum extrusions using Lego pegs. One set is used as a base for the system to slide on while the other acts as guide rails. The base also contains a “wall” which acts as a control, it uses a touch sensor so the robot can recognize it has reached the end of the rail. The rail system is integrated to the Tetrix Extrusions as shown.

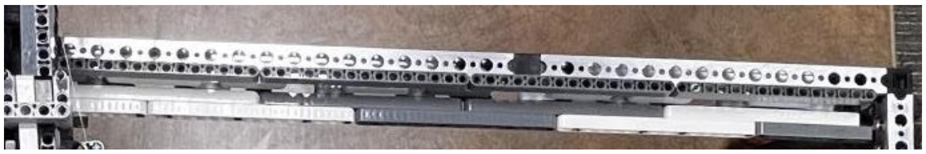

Figure 2: Rail System

Rail System

The control unit of the robot is connected to the rail system through a set of two tires which roll overtop the rail and clap onto the centre guide. Additionally, the system uses a clamp underneath the rails to prevent the housing from rolling off the rail and ensures the tires maintain contact with the rail.

The control unit houses the EV3 Brick and all the drive systems of the robot, as shown.

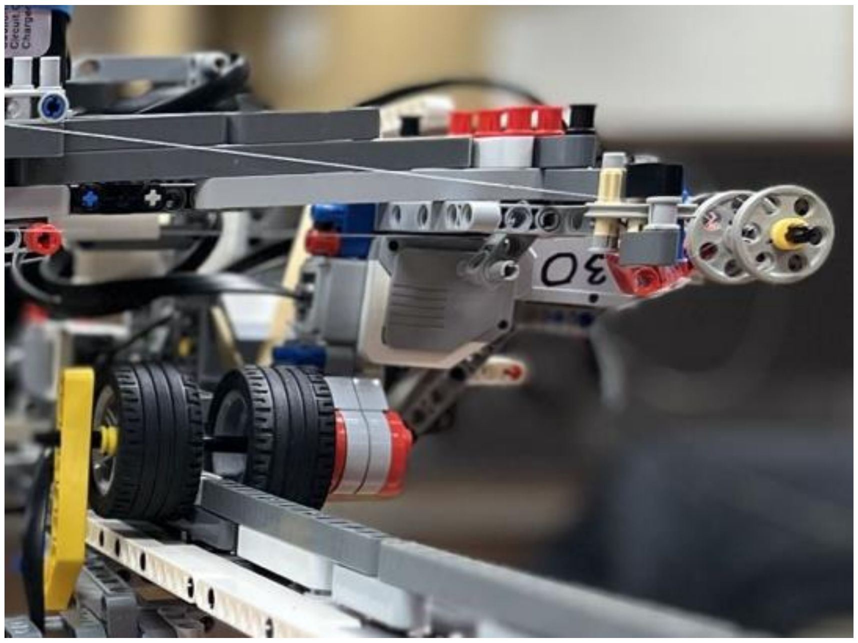

Figure 3: Drivetrain

Lego EV3 Brick

The EV3 Brick is placed on top of the large motor ensuring the centre of mass is directly on top of the drive motor. This helps distribute the weight of the brick and ensures the tires are in contact with the rail. An elastic band is used to squish the brick into the housing, as the movements of the assembly may cause the brick to shift out of its place. The elastic band makes sure the brick does not fall off and provides force opposing the forces due to motion. The mounting and securing of the brick is as shown.

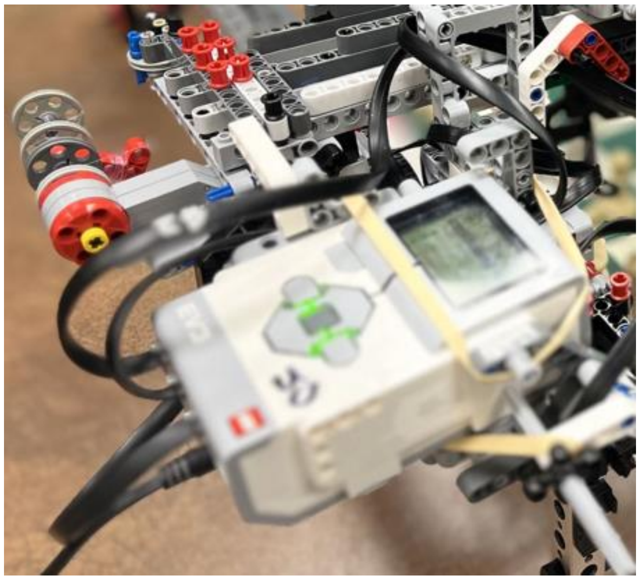

Figure 4: EV3 Placement

Large Motor and Control Unit

The assembly is moved across the rail system through the large motors which drives the clamping tires. The motor itself is lifted, so it does not cause more friction due to contact with the drive surface, so the drive wheels, guide rails and carriage itself provide enough support to balance the assembly.

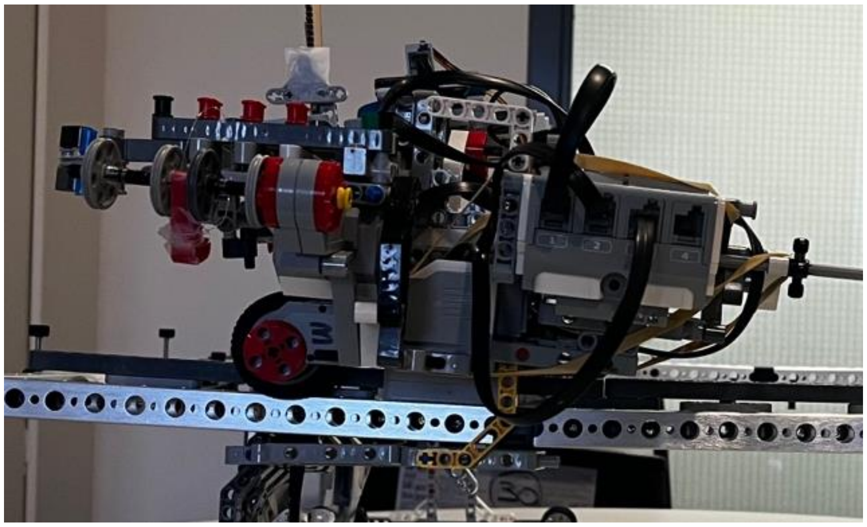

Figure 5: Drivetrain System

The weight distribution, use of tension, and position of the motor ensure the control unit does not lean and experiences little to no tangential force. The control unit rests on top of the tires and has a ski mechanism that holds the unit. The ski system clamps on to the bottom of the base to help secure the system and provide better movement without much friction. Moreover, the base of the pulley system is attached to the control unit. The base includes another large motor which rests on top of the original large motor using Lego pieces, perpendicular to the position of the drive motor. It is placed on top of the drive wheels, ensuring the weight distribution is as close to the axle of the wheels as possible.

Sensors

The rail system houses two sensors: a touch sensor and a colour sensor.

The touch sensor is places at a specific height, such that it activates when the control unit reaches the end of the rail by hitting the "wall" of the base. It is used to zero the system during initialization and between movement.

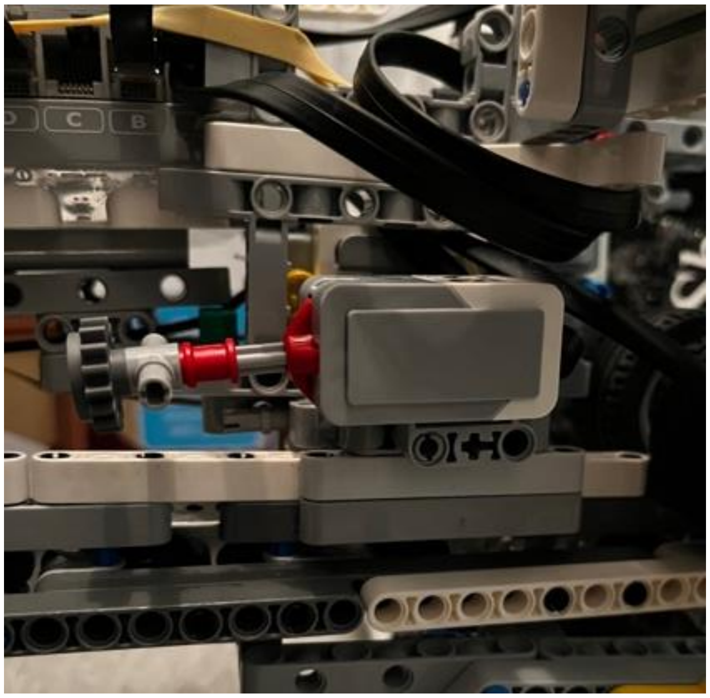

Figure 6: Zero Touch Sensor

The colour sensor is connected to the clamp of the control unit using such that it is ~7.6 mm away from the chess board, as it was observed that this distance provided the most precise colour reading. The sensor was placed in the centre of the pulley system at a predetermined distanced from the base, so that it could identify the red markers that are attached to the centre of each column, used to locate the cell.

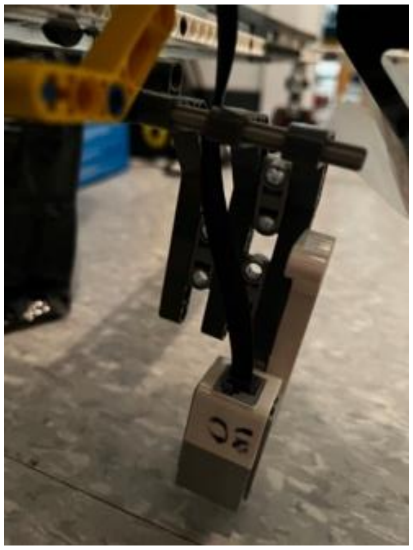

Figure 7: Color Sensor

Pulley System

The pulley system in driven by a large motor as aforementioned. The system uses a fishing line to move the claw, with an axle on the other side to allow for rotational freedom. A fishing line was used as it could withstand substantial tensile forces while having a small profile, allowing for it to be integrated into the system. The fishing line was connected to the large motor through an axle, acting as a driver for the carriage system. The line was pulled tight and tied off to the claw carriage, providing the most precise movement possible.

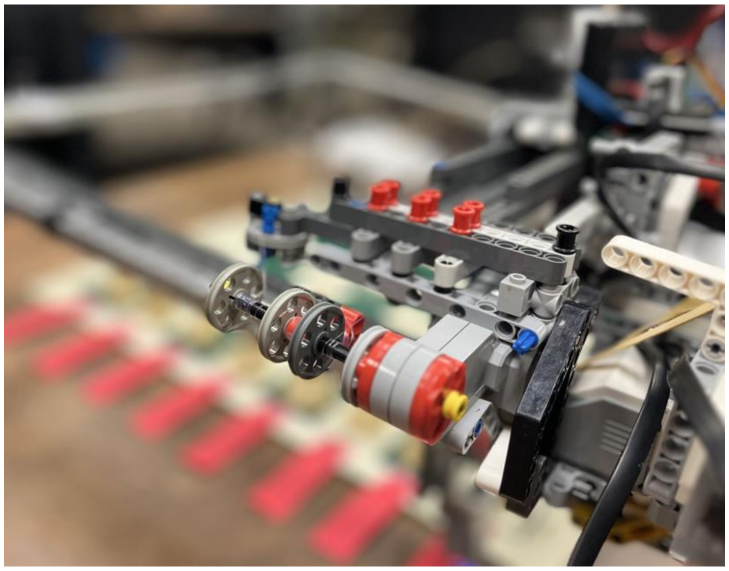

Figure 8: Pulley System

The axle is divided into two sections, one of which has excess line. As the axle rotates the fishing line unwraps from one side and wraps itself to the other section. Simultaneously, the reverse process occurs to the other side of the axle. When the pulley is driven in the reverse direction, the fishing line is spooled up to maintain tension. The line is tied to the claw system such that it does not slip and maintains tension, and the system moves with the fishing line as the pulley rotates. The axial movement on the y-axis is dependent on a raised ski system. This system is composed of three different beams, each of which serve a unique purpose in moving the carriage and dispersing the load.

Initially, a two-beam system was being used, however, it was noticed that when under full operational load it began to buckle. This caused the movement to be inaccurate and imprecise, but also interfered with the carriage movement. Therefore, the team made the change to the fishing line. An added advantage that was noticed was the high strength fishing line provided additional support to the bridge and helped the beams by providing additional tension. It was determined that a third, high load beam, should be integrated into the pully system. Transitioning the 2-beam pulley system into a 3-beam system with each beam having a unique purpose. The high load beam was used to support the medium motor and battery, as they caused the most complications due to their weight. The middle beam was used to hold the claw system with the third beam acting as a counterbalance for the carriage. The ski system was still on the 2 original beams, with an added support on the high load beam.

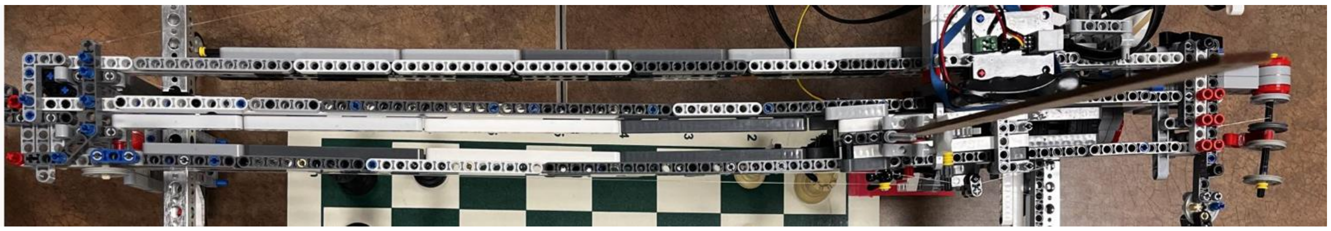

Figure 9: Y-Axis Ski System

The skis were made such that they clamp the beams from top and bottom to help support the weight of the clay system, reduce coupling, and prevent the carriage from moving off the beams. They were spaced such that the claw and the rack and pinion gears were sandwiched between the 2 beams. Placing the claw in this position provided the most precise movement with the least pitch and yaw while the claw moved vertically.

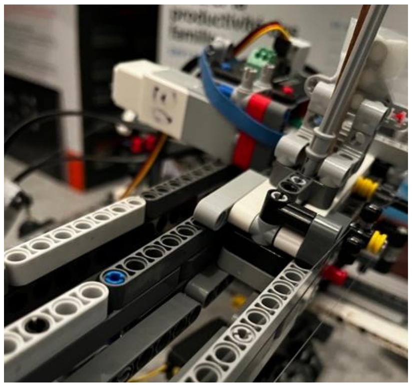

Figure 10: Ski System (Close Up)

The beams were fixed to the control unit on one side and to a support structure on the other side. The support structure had wheels on the bottom that moved along with the rail system, parallel to the x-axis movement. The structure had a wheel carrier on the bottom to ensure it has freedom of motion on the x-axis but due to the tension of the bridge remained in line with the control unit.

The wheel carrier system was implemented instead of a rail system as it was noticed that it improved precision and reduced errors due to sway in the bridge. It was noted that without a drive motor, the rail system experienced a lot of friction that prevented the system from moving properly. Although the wheel carrier still provided some inconsistency and reduced the portability of the robot, it was significantly more precise than a rail system and provided much needed support to the beams. The wheel carrier system is shown below.

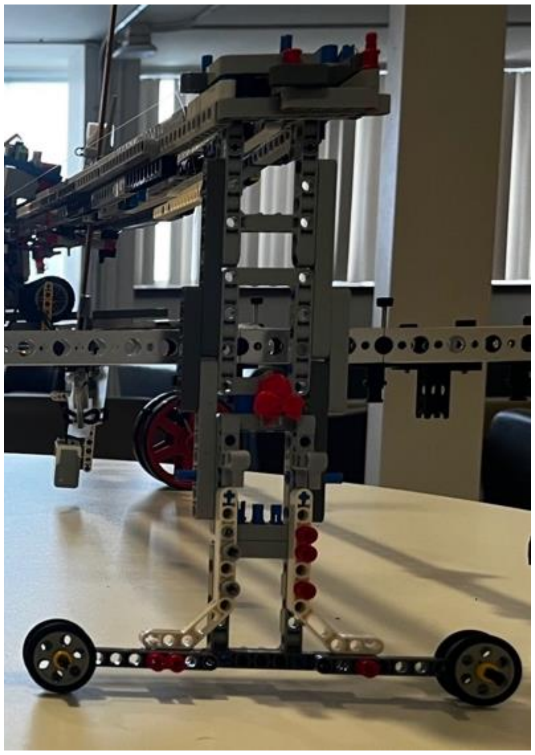

Figure 11: Wheel Carrier System

Claw System

The claw system itself was attached to the skis through a carriage as mentioned above. It consists of a rack and pinion gear set, when powered by the medium motor moves the claw up and down. The claw itself is a Tetrix claw which uses a servo motor to open and close the jaws. The pinion is driven by a medium motor that is fixed in place on top of the skis. The claw itself is mounted onto the rack and thus moves with it.

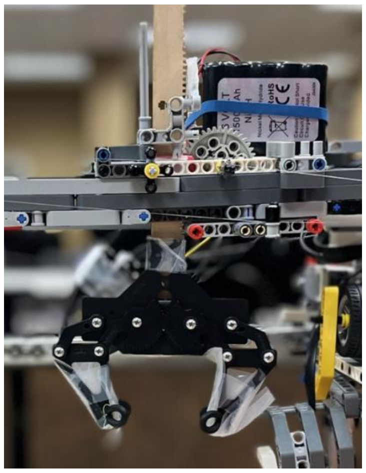

Figure 12: Claw System

The claw is in the standard Tetrix configuration, with a modification to the end of the jaws. To help grab and hold pieces, rubber grippers were added, such that they form a semi-circle. When the jaws shut, they securely hold the piece desired piece. Rubber was chosen so that it is able pick pieces with different shapes due its ability to compress and its malleability. The grippers are also positioned such that when a piece is picked up, it is not at an angle, reducing error when placing a piece down.

Software Design and Implementation

During the software design process, the main priority the team had is modularity. All main functionality was separated into functions, and repetitive code was further separated to decrease debugging work and ease testing. The code was broken apart into the following sub-tasks (functions):

- Configure sensors
- Initialize the board
- Zero the robot
- Get input from users
- Move Piece
  - Move the robot to a cell
  - Pick up a piece
  - Put down a piece
- Shut down

As the move piece function is rather complex, the team split it up into sub-tasks, and allowed the move piece to call those functions. This would not only simplify the code, but also allow for additional logic and flexibility. A good example of this is if a piece needs to be taken. The move piece functions can account for this and call the sub-functions accordingly.

As a sidenote, all of our code was stored on GitHub, and accessed using revision control. This made for easy access across our group and allowed for modifications to be controlled and stored. In the end, our software was made up of over 50 different commits, all representing different versions of code. It proved very useful as there were multiple occasions that the team realized that a change didn't work as expected and used GitHub to revert to a previous working copy. The code and GitHub can be found in the

Main Flowchart

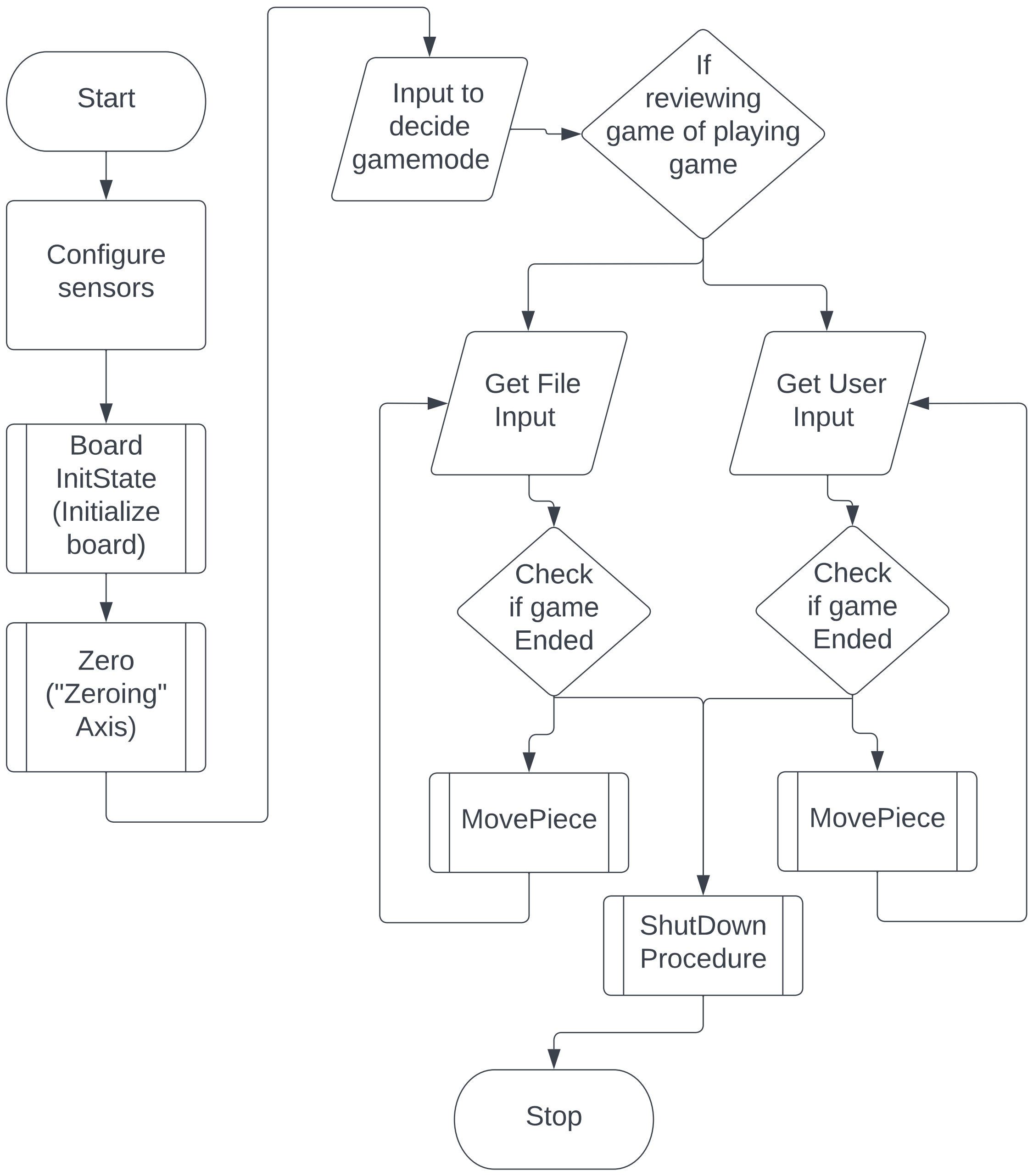

Figure 13: Main Structure

Move Piece Structure

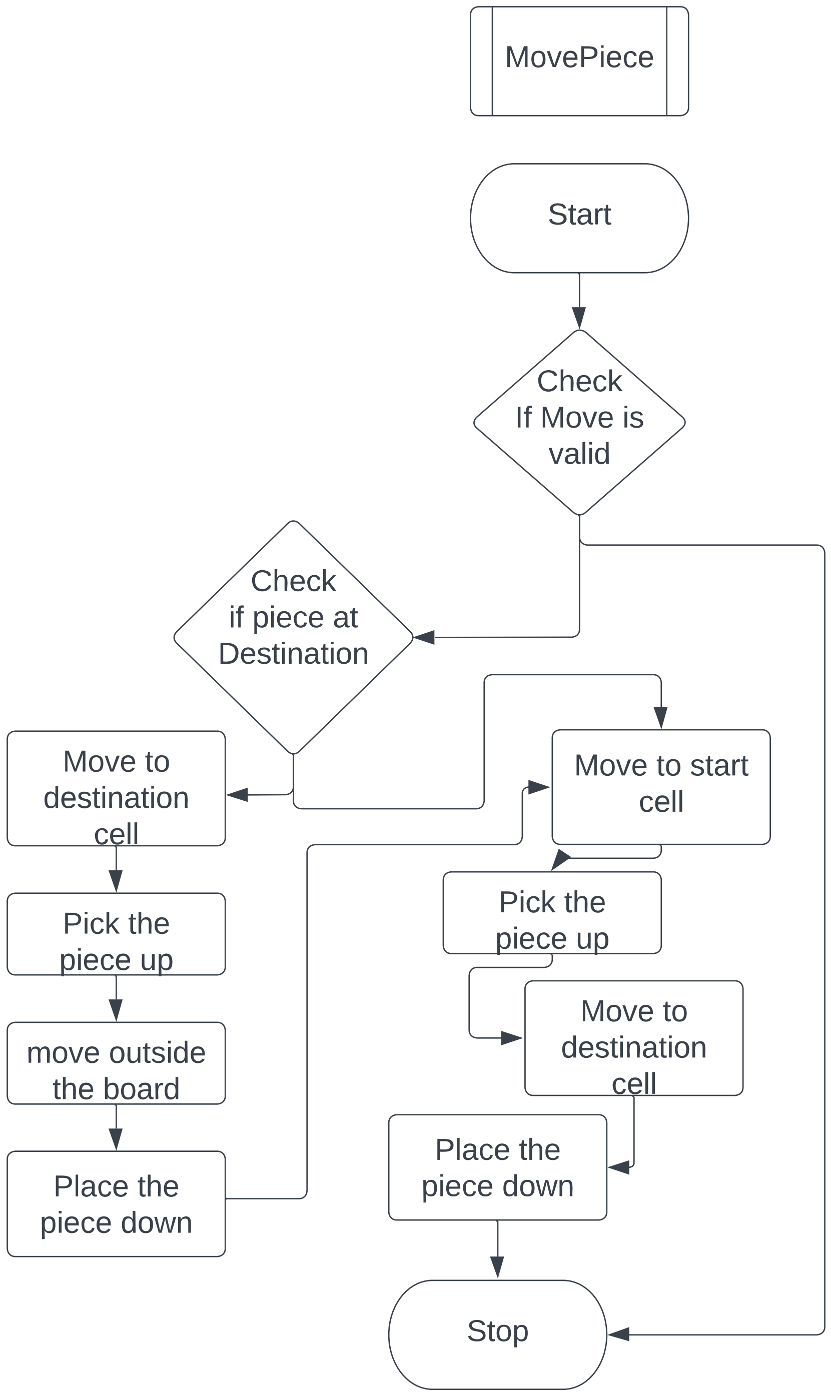

Figure 14: Move Piece Structure

Demo Tasks

During the demo, the team aimed to achieve the functionality to play a short game. This functionality includes:

- Startup (start the program)
- Zeroing procedure (Zero Axes)
  - Return both X and Y axis to the 'zero' corner cell
- Display time remaining before move is made
- Take User Input
  - Interpret cell input and translate it to a move on the board.
    - Be able to input any cell with the EV3 buttons.
- Move Piece
  - Pick up the desired piece and move it to the selected cell on the board
- Capture Piece
  - When the selected cell has an opposing piece, it will:
    1) Go to the opposing chess piece and pick it up
    2) Bring the piece to the "captured" section, outside the bounds of the board
    3) Return to zero
    4) Move the desired piece to the newfound open cell
- Ignores Input while moving and making a move
- Zeroes after every move
- Shutdown
  - Will display who won the game (by resignation of a player)
  - Will zero the axes for quick initialization if a rematch is wanted
  - Will terminate the program
- Time permitting, play a game with TA available
  - If TA loses, course project mark will be changed to 100 :)

Function Description

Sensor Configuration	Input: Void	Output: Void	Programmer: Owen
The sensor configuration function will be used to initialize all the robot sensors. This includes the color sensor, and touch sensor. This function will also be used to zero the motor encoder on the y-axis, so that the robot always knows the correct location of the y-axis carriage. Because of this, it is crucial to the robot's operation that on start up, the y-axis be at it's zero point. If it is somewhere else, the robot will not be able to move to the required locations. This was a decision that the team made during the design, as although it could be fixed with another touch sensor, the team decided that it was simpler to just ensure the proper position on start up.			

Board Initialization	Input: Void	Output: Void	Programmer: Shabd

The board initialization function receives the 2D array representing the board (which is a global variable due to the RobotC constraints) and populates it with the starting board in a chess game. Each piece is represented by a string that is two characters long, the first representing the color, ‘W’ or ‘B’ for white or black. The second character represents the piece type, ‘N’ for knight, or ‘K’ for king. This will be accessed by all the functions and be able to keep track of the location of all the pieces. This is critical for the operation of the robot, because it will allow the robot to know if a piece is being captured and can trigger alternate movements. This function modifies the board array, which is stored as a global variable due to restrictions in RobotC.

Zero	Input: Void	Output: Void	Programmer: Eidan
This function is used to bring the robot to a set zero location. This will ensure consistent and reliable movement of the axes during the player movements. The function will zero the x-axis with a touch sensor and will drive until the touch sensor is triggered. The y-axis will then be zeroed with the motor encoder, so the claw will be at a set location.			

Get User Input	Input: int currentLetter, int currentNumber, int moveToLetter, int moveToNumber	Output: boolean	Programmer: Owen
This function will get the user input, getting the row and column of the piece and the destination. It will take an input of all 4 integers (initial X, initial Y, destination X, destination Y), and will modify them. Because the inputs will be pass by reference, the modified variables will be modified in the main function as well, so the functions is effectively returning all 4 values. Additionally, this function returns a boolean values, representing whether the player has resigned. If the player has resigned, it returns false, otherwise it returns true, having modified the coordinate variables.			

Move Piece	Input: int x1, int y1, int x2, int y2	Output: boolean	Programmer: Richard
This function is the high-level function, that performs all piece movement. It calls sub-functions, such as move to cell, pick up piece, and put down piece. There is also logic in this function to check if there is a piece at a destination that can be captured, and if there is, it will also pick up and remove that piece. It returns false if the move is invalid, and true if the move has been performed.			

Sub-functions:

Move To Cell	Input: int currX, int currY, int x, int y, bool firstMove	Output: void	Programmer: Richard
This function is called to move the robot to a particular cell. It takes the current cell and destination cell as an input and performs the math to determine the movement necessary to			

navigate to the destination. It will navigate using the color sensor values for the x-axis, which are positioned to allow the robot to know which cell it is located at. It uses motor encoder values for the y-axis, which, along with the knowledge of the diameter of the axle, can be used to calculate the distance to travel.

Pick Up Piece	Input: int x1, int y1	Output: Void	Programmer: Owen
This function will pick up a piece given its coordinates. When this function is called, the robot is already directly above the piece. This function will ensure the claw is open, lower the claw, close the claw to secure the piece, and raise the claw back up. Raising the claw back up the proper height is important because it allows the pieces to be moved over other pieces and will prevent the pieces from knocking other pieces over. It will use the coordinates that are passed to the function (by value) to know which piece is being picked up. If the piece is a king or queen, the claw lower distance will be a different value. This is to grab the piece at the correct location and have different movements for pieces of different heights.			

Put Down Piece	Input: int x1, int y1	Output: Void	Programmer: Owen
This function will put down a piece that is in the claw. This function is very similar to the pick up piece function and performs the operations in almost opposite order. When this function is called, the robot is already directly above the piece. This function will lower the claw, open the claw to release the piece, and raise the claw back up. As with the pick up piece function, it will use the coordinates that are passed to the function (by value) to know which piece is being put down. If the piece is a king or queen, the claw lower distance will be a different value. This is to release the piece at the correct location and have different movements for pieces of different heights.			

Capture Piece	Input: int x2, int y2	Output: Void	Programmer: Owen
This function will be called when the move piece function recognizes that a piece needs to be captured. It will pass the coordinates of the piece that needs to be captured. The function will navigate to this piece and move it off the board. It will re-zero the robot, and then return. This function will also utilize the move to cell function, and the pick up and put down piece function.			

Shut Down	Input: Void	Output: Void	Programmer: Eidan
This function will be used to shut down the robot. It will zero the axis before the robot shuts down, as it will ensure next time on start-up it is at the correct position. It will then prompt the user to start a new game, or shut down the program, and react accordingly.			

Unit Testing

These functions were tested throughout the assembly process, ensuring everything was put together correctly and that integration would work as expected. All functions were first tested individually, to make sure everything accomplished the appropriate task. This helped eliminate many errors that likely would have plagued us for a long time, had the team not found them at that stage. These mostly include many small things, such as getting a button press and not waiting for it to be released or running motors in the opposite direction as intended. These issues were easily fixed, and all of the functions could run smoothly.

Next, the team started the integration testing, by testing our main function as well as the interfacing between functions. Again, the team ran into many errors, however most of them were easily fixed. This included passing the wrong variables to functions or passing them in the wrong order. One large issue that the team came across was the transposing of the 2D array to the board. The board on the table had the letters and numbers in a different orientation than what was expected, which led to our 'zero' point not being nearest to the cell [0,0]. This was a large issue that took a fair bit of logic, as the coordinates were converted to reflect that in the hardware. This was an integration issue that the team did not expect, however once solved it was no longer a worry.

Data Storage

The data about the chess board is stored in a 2D array, which is 8x8. This stores strings that represent the color, and piece. For example, "WN" represents a white knight, and "BK" represents a black king. This is very important to the functionality of the program because it allows the robot to adjust the operation depending on the piece it is picking up, as well as know when a piece is being captured.

If a piece is moved to a location where a piece of the opposing color is, the robot will first pick up that piece and place it outside the bounds of the board, then execute the original move 'capturing' that piece.

The robot checks this array for many things, including determining:

a) Whether the destination location is the same as the starting location (invalid move)
b) The destination location has a piece that is the same color as the one selected (invalid move)
c) If the destination location has a piece of different color (capturing a piece)
d) If the initial location contains a king or a queen, which will actuate the claw differently to ensure possession of the piece

Apart from this array storing the position of the board, there are other data that the robot stores to ensure its proper operation. This includes which player's turn it is (black or white), so that at the end of the game it will correctly display the winner of the game. It also stores the amount of time that each player has left (subtracted from 5 minutes), which is used to implement the chess clock functionality. Additionally, there are many more variables that are used to store locations and coordinates, that are passed between functions, however, are much less notable then those mentioned above.

Constant data storage has been crucial in allowing easy modification of the program, which has proved to be useful throughout the integration process. Due to RobotC's allowance of motor and sensor ports to be stored in integer variables/constants, there are many pre-set constants used. This includes storing the motor ports for each axis, as well as the default motor powers. It also includes storing the sensor ports, motor encoder constants, claw movement constants, and location constants.

Constraints and Criteria

The success criteria of our project, Carl-bot, is determined by evaluating the purpose the robot serves. The objective of the project is to support a game of chess through mechanical movement, taking user input and computing the logic autonomously.

Our project's success criteria are in addition to the overall success criteria of the course project.

Criteria

The project criteria were established by working backwards from the project deadline, ensuring it would be complete by the target date. It was concluded that agile development would be used to ensure the project met the minimum criteria before adding complexity. The minimum criteria were identified by isolating the fundamental tasks that the robot would need to meet.

The criteria are split into two subsections, mechanical and software. Each subsection criteria would need to be independently met before they could be integrated.

Mechanical

The mechanical criteria were developed by analyzing the resources available. It was identified that out mechanical assembly must:

- Span the entire tournament sized chess board with generous tolerances on all extremes
- Have enough height to clear all pieces during piece movement
- Have precise and smooth movement across both axes
- Have a robust bridge to hold the carriage and all accompanying accessories
- Have a complete assembly to be able to execute an entire game of chess
- Be able to pick up each piece
- Be able to go to any cell on the board
- Successfully integrate the Tetrix claw for claw actuation

Software

The software criteria were developed by going through a chess game and breaking it down into smaller components. Furthermore, each "type" of move was broken down into the fundamental functions required. The "skeleton" of the operation had 8 aspects: initialization, calibration, game state, input, movement, piece pickup, shutdown, and case handling.

- Initialization
  - Upon start-up, the EV3 must:
    - Configure all sensors
    - Initialize the board and starting locations of all pieces
- Calibration
  - After every move, the EV3 must calibrate itself to ensure unintentional offsets do not compound
- Game state
  - The EV3 must ensure the game is still active before prompting for user input
  - The EV3 must check the desired cell's status and act accordingly based on if the cell is empty or if piece is being captured.
  - The EV3 must implement File I/O by writing game moves to a file and then playing the game based on file input.

- Input
    - The EV3 must take in user input and execute movements accordingly
    - The EV3 must only allow predetermined inputs
    - The EV3 must disregard illegal inputs and/or inputs made during movement
    - The ability to have a user select a cell, interpret the selected piece in the cell, and only offer legal moves available to that piece.
- Movement
    - The EV3 must be able to interpret the desired cell and move the carriage accordingly
    - The EV3 must decide upon the appropriate movement
    - The EV3 must be able to track its current position on the board
- Piece pickup
    - The EV3 must identify the desired piece and actuate the claw the required distance
    - The claw must always be actuated when holding a piece
- Shutdown
    - The EV3 must return to the initialization position upon shutdown
    - The EV3 must only initiate the "end-game" state upon resignation or timeout
- Case handling
    - The EV3 must retry a move if it gets stuck during movement
    - The EV3 must identify that a piece is being captured and "take" the piece before moving

Constraints

The constraints of the project were identified by considering the project timeline, expected mechanical assembly, and our team's software experience. Our goal was to use a tournament sized chess set, guaranteeing that the board size and pieces would be standard across the board.

Due to the tight time constraint, the team decided to use agile development. This entailed identifying our required constraints and desired constraints. Complexity would be added only after the initial criteria was reached. This ensued the team would complete the project before the deadline.

Mechanical

- A tournament sized board and pieces
    - Tournament sized chess boards are larger than most casual sets, resulting in a large mechanical assembly, bringing forth additional challenges.
    - Additionally, tournament chessmen are tall in relation to the board, resulting in a higher than anticipated mechanical assembly.
- Lego EV3 components
    - The Lego EV3 components are unreliable, occasionally having unpredictable movements/readings, requiring redundancy to be built in
    - The Lego beams are not very structurally sound, causing bends in the bridge and large tolerances
- Tetrix integration
    - There is no "official" way to integrate Lego pieces with Tetrix components, so a workaround method was used
    - The Tetrix and Lego components did not have similar length: width: height ratios, resulting in components being held using compression, causing additional friction and tension in the components

Software

- RobotC Language / IDE
  - The program was constrained to be done in RobotC, as RobotC connects to the EV3
- Movements
  - The software must only be able to make predetermined movements, limited to discrete movements within the board

Implementation

The primary guiding constraints were the board size and the restriction to using RobotC to program the EV3. These restraints guided the development & implementation of the mechanical assembly and accompanying software. During development, it was observed that 2 chessmen, the queen and king, needed to have exceptional cases for their movement as their height was not compatible with the default claw movement.

Furthermore, the guiding criteria for the project were the Tetrix integration, axial movement, ongoing calibration, and controlled movement. The Tetrix integration is necessary for the project as the assembly was build using Tetrix components and it provided the foundation of the robot. The Lego components on their own would be structurally sound to form a base for the project. The controlled axial movement is a fundamental aspect of the project as it is necessary to play a game of chess. Achieving controlled axial movement was paramount to the success of the project, once it was achieved, ongoing calibration was implemented to maintain precise axial movement. These criteria were essential in the success of the project, as they are the fundamental traits that the project required.

During development of the project, the guiding criteria changed to further focus on precision. It was recognized that without repeatable movements, the robot would not be able to execute an entire chess game. Ongoing calibration become a priority and extra case handling supressed as a result. Moreover, our initial design for user input became inessential as the implemented design choices to increase accuracy met the criteria in another manner. It is only necessary for the robot to compute basic game logic during user input, thus a complex user interface with game logic and piece detection became irrelevant. Additionally, storing the game states and piece movement in a file became unimportant to the project. Since all the chessmen were stored in an 2D array and basic game logic was being checked throughout the game, writing the game states to a file became redundant. Moreover, the team prioritized user input to conduct a player v. player game over replaying a game from file.

Throughout the integration and implementation phase of the project, the success criteria of the project evolved to align with our progress. However, our primary criteria, controlled axial movement and Tetrix integration, remained left unchanged. These criteria set the foundation of the project and the remaining criteria became secondary.

Verification

Throughout project development, the project constraints evolved to fit the timeline. The updated success criteria and constraints for the project demo were as follows:

Success Criteria

- Board mapping
- User input
- 2 Axis movement
- Piece movement
- Basic game logic

Constraints

- Tournament sized chess set
- Board size and position
- Predetermined movements
- Small tolerances for captured piece location

The success criteria were all met as the robot successfully executed a game of chess, taking in user input and conducting the appropriate moves.

During the demo, the robot conducted its tasks with the full-sized tournament chess set. Furthermore, it accurately located and identified each chess piece in the board and was able to move the selected piece precisely. It correctly executed the predetermined movements, and correctly translated user input into mechanical movement on the chess board. Lastly, upon piece capturing, the robot correctly executing the appropriate logic.

The constraints from the preliminary report that were not met were the File I/O integration, complex game logic, and an intricate GUI. The preliminary report stated that our team would implement agile development, ensuring that the primary criteria and constraints were met before additionally, “secondary”, goals were pursued.

During the project development, it was determined that the File I/O would not be pursued because it was a redundant feature that did not take precedence over other criteria. It was identified that the board mapping and game status would be stored within the source code, eliminating the need to use File I/O to read and write the game state.

Additionally, complex game logic and an intricate GUI was not met in the final design. Our group implemented a standard GUI allowing the user to select the initial cell and the final cell. The program would determine the piece being moved, if a piece was being captured, and then would execute the appropriate move. In the preliminary report, the team had planned to integrate game logic within the move selection. The move selection was planned to only let legal moves be played, as well as piece selection instead of cell selection. However, these features were not implemented as the game logic was out of the scope of our abilities within the timeframe. The updated design to account for this was the initial & final cell selection, with basic game logic such as only moving to a free space or capturing an opposing piece, ensuring a different initial and final position, and only capturing opposing pieces. The implemented logic allowed for a full game to be played with the appropriate logic to ensure moves would not “break” our design.

Throughout development, the team faced many failed ideas and unsuccessful tests. However, our initial plan of action accounted for this with the implementation of agile development, ensuring the project would meet the necessary fundamental criteria before adding complexity to the design.

Project Plan

A well-organized project plan was key to successful project development. The team made a schedule and an outline to ensure that all the work would be completed with proper buffer time. Throughout this process, the team maintained consistent communications through regular meetings and through virtual means. Our deliverable tasks were assigned and completed in a timely manner and follow-up meetings were conducted to review tasks and outputs. This form of organization ensured all team members were informed and prepared for all due dates and responsibilities.

At the start of our project, the team laid out due dates for different aspects of the robot to ensure all required criteria were met before the final deadline. This included planning, hardware development, software development, integration testing, and presentation rehearsal and development. Knowing software integration would take a lot of time, the team planned to allow for a few extra days for integration and testing. This was shown to be accurate, as it was probably the greatest challenge of the project.

During our development, the team made sure to split our tasks among group members to progress towards our goals. For example, when preparing for presentations the team divided the presentation into sections with each member taking a part. Then, the team met before the presentation to assemble the presentation. This improved efficiency and allowed everyone to meet the deadlines set out.

The hardware and software development were also divided in a similar manner to ensure efficiency and consistent progress. During the hardware development, some members focused on the chassis of the robot while others worked on the frame. This allowed all members to work productively and ensured there were not too many hands working on one task. This allowed everyone to accomplish some of the goals the team set out.

Similarly, software development was divided up by assigning each team member function(s) to write. Everyone would complete their work, and then the team could all come together with our parts and work on integration as a team. One other notable planning and execution tool from the software perspective was the use of GitHub to track and implement changes. This allowed for all members to be informed of changes, as well as store working copies so that code that did not work as intended could be reverted.

While every member took on a different role in the development of the robot, consistent communication was maintained, and all members were aware of all developments in other areas.

After development began, a few revisions were made to this plan. Software development began at the same time as chassis and frame development to test the moving parts. Testing and troubleshooting were moved up and given more time as many issues presented themselves early, requiring more time to be allocated for this.

To conclude, the manner in which the team completed the project closely matched the projected timeline that was outlined in the preliminary report. The team's effective communication and productivity through this process ensured that deadlines were met, and all deliverables were completed on time.

Recommendations

After developing the chess robot, the team reflected on our design decisions, and identified areas where the robot could be improved. The team identified many things that could have been improved upon, and if given another chance, would have implemented. This includes a more robust mechanical assembly and increased useability through improved software functionality.

Mechanical

Axis Movement

To improve the precision and accuracy, gears could be used instead of tires. A rack and pinion system would make the system easier to control since a motor encoder input would directly correlate with the position the board. Additionally, y-axis zeroing would greatly increase the accuracy of the system. This would allow for more precision movements and would greatly decrease the error in relation to the y position. Furthermore, during initialization, the y position could be zeroed instead of "assuming" that it is at the zero position. This would decrease error that the team experienced greatly.

Another improvement that could be made is to implement a motor rail system on both sides of the board. This would be much more consistent than having a wheel carrier on one side which relies on the system opposite the board. This would make the movement much more reliable, especially when the robot needs to make a move on far side of the board.

Lastly, instead of using a pulley system a motor driven system could be implemented for y-axis movement, this would make the maintenance easier since the fishing line made it very hard to make changes to the system once it was locked in place on the pulley.

Claw Gearing

An improvement to the claw system could be to use a "made to fit" acrylic rack and pinion set. This would make the system more robust and more resistant to malfunctions. During our testing, it was observed that the Lego pinion began to chip at the wooden rack, decreasing the precision of the system. Additionally, a zeroing system could be implemented to precisely track the position of the claw during actuation.

Weight Reduction / Size Reduction

An additional improvement to the claw system would be to reduce the overall weight of the system. A heavy carriage and claw system applied a lot of force on the beams, causing them to flex unnaturally. Overtime the flex in the beams will weaken them until they broke. Another solution to make the assembly more robust is to use a smaller board instead of a tournament sized board. By using a tournament sized board, a lot of complications and constraints were introduced that made the assembly hard to develop and maintain.

Software

Chess Engine

One large change to our project that would vastly improve the applications and usefulness of our project would be the integration of a chess engine. A chess engine is a computer that has pre-programmed knowledge and will be able to make "intelligent" moves. This would allow users to play by themselves, against the robot powered by a computer. Although the constraints of the project limited this, specifically the computing power of the EV3 and the time allocated to the project, an alternate design of the system that could integrate a chess engine would be a large improvement.

Computer Vision

Utilizing offboard computer vision for this project would be a large improvement to reliability and accuracy. Mounting a camera above the board, the computer would be able to always know the location of the claw assembly in relation to the board. This can be accomplished with any specific and recognizable visual indicator on the corners of the board and the robot, which can be captured with the camera and recognized by the computer. A common form of this is a sticker that looks comparable to a QR code, from which the computer will be able to determine the relative position of the robot. Following this, the computer can adjust for error and ensure precise movements without offsets. If the robot were to position itself slightly inaccurately (a frequent problem the team faced), the computer could recognize the offset and compensate for the error, increasing reliability and precision.

File/IO

Lastly, something that the team had hoped to integrate into our solution, but did not have the time to implement, is a File I/O system. This was in our preliminary project plans, and the team wanted to be able to import and export game states. This would allow users to play different challenges and save game states for later. This would be a small improvement that would vastly increase the usability and application of our robot and would be a great addition.

Conclusions

After reflecting on our learnings and outcomes, the team feels very proud of the resulting robot and system while maintaining that our project has plenty of room for improvement. In recap of our goals, the problem the team wanted to solve was to build a robot capable of executing chess moves. Our final goal was to play a game of chess interfaced by a robot, while being useable in many different playing circumstances.

For our mechanical system, the EV3 brick moved in the “x-axis”, and the claw chassis would move in the “y-axis”, allowing the robot to reach any spot on the board. Tetrix pieces were to be used to construct the frame, as they would provide the strongest foundation possible. The team integrated the Tetrix claw to the axis system to enable capture and movement of all the pieces on the board.

The initial software design involved storing the entire chess board and piece positions in a 2D array. The robot would travel to any given spot on the board by first calculating how far it would need to move in both axes, then moving that distance with motor encoders. The software also included plenty of error handling, for both illegal moves and for capturing of pieces. This allowed for flexibility in the programming and the improved user interfacing.

The criteria of this project were established by working backwards from the deadline. Due to the complex nature of the project, it was decided that minimum criteria would be met before moving to more complex goals. This would ensure the project would hit the initial goals before the deadline. The mechanical and software criteria were identified, and the guiding criteria items were established. The guiding criteria were identified as the fundamental criteria that were necessary for project success.

A major mechanical constraint was the use of Lego pieces for construction. Lego is flexible, causing bending in the support beams and buckling on load bearing pieces. Thus, compounding tolerances causing imprecision and inaccuracies. This constraint guided the hardware development and was a primary concern during software development as well. It was concluded that without repeatable movements, it would be impossible to play an entire chess match. Thus, the focus shifted to precision in both mechanical and software design.

The data storage of all the chessman information as stored in a 2D array, with each cell in the array correlated to a cell on the board. Each piece has a colour designation, “B” or “W”, and then the piece letter. This let the robot track the location of all the pieces throughout the game. Additionally, the claw moved a different vertical distance depending on the piece, requiring accurate locations of all the pieces. The location of the pieces was also used in determining the validity of a move and contributed to the logic when capturing a piece.

The testing and troubleshooting procedure involved segmenting the code and testing each function individually. By testing their individual functionality before integration, it made the debugging process less strenuous. Thus, the debugging process mostly consisted of working through the function interaction rather then the individual functions themselves. As previously mentioned, this process took a lot of time with results coming slowly, however once completed was functional without issue.

Lastly, the team spent some time to reflect on our robotic system. It was recognized that the project met all the minimum success criteria that were initially set out, however also left plenty of room for improvement. The robot could interface a game of over the board chess, which the team later realized could be purposed to help individuals afflicted with certain disabilities to play a game of chess.

The team is quite happy with the result of the robot and enjoyed the engineering process that was worked through. The team learned plenty about the design process and look forward to applying this knowledge in future studies.

Backmatter

References

No references cited. All referenced documentation / information was from MTE 100 or MTE 121, which is not needed to be referenced as per Professor Consell’s instruction.

The team would like to specifically thank Professor Consell and Professor Nassar for their assistance and imparted knowledge through the term, as well as the teaching team for both MTE 100 and MTE 121 who the team learned from and referenced throughout the development of this project. The team could not have accomplished what has been done without the aforementioned support.

Appendix A – Source Code

```
// Can also be found at https://github.com/owenmoogk/carl-bot

// main.c

/*
      Carl-Bot

      Owen Moogk
      Eidan Erlich
      Shabd Gupta
      Richard Wang

      November 25, 2022
      Version 58.4

      This code is used to operate the 'carl-bot' chess robot.
      We poured lots of effort into this project and are very proud of the
      result.

      Assumptions:
            - On startup, the program was run previously.
          This is so the y-axis is at the zero point
              and the claw lowering is at the zero point as well.
*/

// include the claw library
#include "EV3Servo-lib-UW.c"

// HUGE ASSUMPTION:
// on startup, the y-axis (pulley) MUST be at the zero point

// sensor constants
const int TOUCH = S4;
const int COLOR = S3;
const int X_ZERO_TOUCH = S2;
const int RED = colorRed;
const int SENSOR_WAIT_TIME = 50; // ms

// motor constants
const int X_MOTOR = motorD;
const int X_MOTOR_POWER = 30;
const int X_AXIS_OFFSET = 250; // ms
const int X_H_OFFSET = 150; // ms, offset error in the 'h' cell
const int CAPTURE_WAIT_TIME = 900; // ms

const int Y_MOTOR = motorA;
const int Y_MOTOR_POWER = 100;
const int Y_CELL_CLICKS = 1175;
const int Y_ENCODER_OFFSET = 110; // mss

// claw constants
```

```
const int CLAW_ACTUATION_MOTOR = motorB;
const int CLAW_MOTOR = S1;
const int CLAW_CLOSE = 10;
const int CLAW_OPEN = 70;
const int CLAW_WAIT_TIME = 500;
const int CLAW_LOWER_CLICKS = 280;
const int CLAW_LOWER_CLICKS_TALL = 185;
const int CLAW_ACTUATION_POWER = 10;
const int CLAW_ACTUATION_OFFSET = 30;
const int SV_GRIPPER = 4;

// display constants
const int ASCII_START = 65;
const int ASCII_END = 72;

// where the taken peices go
const int END_X = 7;
const int END_Y = 0;

// the board is 8x8
const int BOARD_SIZE = 8;
const int CHESS_CLOCK_INIT = 300; // seconds

// 2d array with the board location
// bk, wk, k, q, b, r, p, n for knight
string board[BOARD_SIZE][BOARD_SIZE];

// sensor configuration
void configureSensors()
{
      SensorType[TOUCH] = sensorEV3_Touch;
      wait1Msec(SENSOR_WAIT_TIME);
      SensorType[COLOR] = sensorEV3_Color;
      wait1Msec(SENSOR_WAIT_TIME);
      SensorMode[COLOR] = modeEV3Color_Color;
      wait1Msec(SENSOR_WAIT_TIME);
      nMotorEncoder[Y_MOTOR] = 0;
}

// get the input for a cell
void getCellInput(int &currentLetter, int &currentNumber, bool firstLine)
{
      while (true)
      {
            // the displays cast the current letter to a character via the
        // ascii values
            if (firstLine)
                  displayBigTextLine(2, "Current: %c%d", currentLetter,
                               currentNumber);
            else
                  displayBigTextLine(4, "Final: %c%d", currentLetter,
                               currentNumber);
            // wait until the button is pressed
            while (!getButtonPress(buttonAny))
            { }

            // if the user hits enter, we return from the function
            if (getButtonPress(buttonEnter))
            {
                  while(getButtonPress(buttonEnter))
                  { }
                  return;
            }

            // adjust the numbers and letters
            if (getButtonPress(buttonLeft))
                  currentLetter -= 1;
            if (getButtonPress(buttonRight))
                  currentLetter += 1;
            if (getButtonPress(buttonUp))
                  currentNumber += 1;
            if (getButtonPress(buttonDown))
                  currentNumber -= 1;
            // wait until the button is released
            while (getButtonPress(buttonAny))
            { }

            // wrapping around the board
            // ie, if you start at 1 and hit back, it will go to position 8
            // this makes it easier, instead of scrolling through everything
            if (currentLetter < ASCII_START)
                  currentLetter = ASCII_END;
            if (currentLetter > ASCII_END)
                  currentLetter = ASCII_START;
            if (currentNumber < 1)
                  currentNumber = BOARD_SIZE;
            if (currentNumber > BOARD_SIZE)
                  currentNumber = 1;
      }
}

// getting the user input (pbr)
bool getInput(int &currentLetter, int &currentNumber, int &moveToLetter,
              int &moveToNumber)
{
      // first ask the user if they want to continue (or resign)
      displayBigTextLine(1,"Continue?");

      bool exit = false;
      bool doContinue = true;

      // while the user is entering a move
      while(!exit)
```

```
  {
        // if the user hits enter, then exit out of the loop
        if (getButtonPress(buttonEnter))
              exit = true;

        // if they hit left or right switch from resign / continue
        else if (getButtonPress(buttonRight) || getButtonPress(buttonLeft))
        {
              doContinue = !doContinue;
              while(getButtonPress(buttonRight)||getButtonPress(buttonLeft))
              { }
        }

        // if we are continuing, display yes, otherwise no
        if (doContinue)
              displayBigTextLine(3,"Yes");
        else
              displayBigTextLine(3,"No");
  }

  // wait until enter is released to continue
  while(getButtonPress(buttonEnter))
  { }

  eraseDisplay();

  // if we resign, return false
  if (!doContinue)
        return false;

  // letter stored in ascii, number stored as int
  currentLetter = ASCII_START;
  currentNumber = 1;
  moveToLetter = ASCII_START;
  moveToNumber = 1;

  // get the input for both the current and destination cells
  getCellInput(currentLetter, currentNumber, true);
  getCellInput(moveToLetter, moveToNumber, false);

  // after we get the input, we need to rectify the numbers
  // so that they can be used as indicies of the array
  currentLetter -= ASCII_START;
  moveToLetter -= ASCII_START;
  currentNumber -= 1;
  moveToNumber -= 1;

  // we also need to transpose this so it corresponds correctly with the
// board
  currentLetter = (BOARD_SIZE-1)-currentLetter;
  moveToLetter = (BOARD_SIZE-1)-moveToLetter;
```

```
	eraseDisplay();
	return true;
}

// zeroing function
void zero()
{
	// zero the x axis (until the touch sensor is triggered)
	motor[X_MOTOR] = X_MOTOR_POWER;
	while (!SensorValue[X_ZERO_TOUCH])
	{     }
	motor[X_MOTOR] = 0;

	// zeroing the y axis (until motor encoder is 0)
	motor[Y_MOTOR] = Y_MOTOR_POWER;
	while (nMotorEncoder[Y_MOTOR] < 0)
	{ }
	motor[Y_MOTOR] = 0;

	// default to setting the gripper to open state
	setGripperPosition(CLAW_MOTOR,SV_GRIPPER,CLAW_OPEN);
}

// move to cell
void moveToCell(int currX, int currY, int x, int y, bool firstMove)
{
	// calculate the change in y and x
	int travelX = currX - x;
	int travelY = currY - y;

	// we need these to determine the direction of travel
	int directionX = -1;
	int directionY = 1;

	// if we are going backwards, multiply by -1
	if (travelX < 0)
		directionX *= -1;
	if (travelY < 0)
		directionY *= -1;

	// if we are moving on the x axis
	if (travelX != 0)
	{
		// move using the color sensor, a specified number of times
		for (int count = 0; count < abs(travelX) + 1; count++)
		{
			// turn the motor on
			motor[X_MOTOR] = X_MOTOR_POWER * directionX;
			wait1Msec(50);
			// while the color isn't red
			while(SensorValue(COLOR) != RED)
			{ }
```

```
				// if we are not on the last pass
				if (count != abs(travelX))
				{
					// while the color is red
					while(SensorValue(COLOR) == RED)
					{ }
				}
			}

			// this is tuned for color sensor positioning,
		  // need to delay to center
			wait1Msec(X_AXIS_OFFSET);
			// turn off the motor
			motor[X_MOTOR] = 0;
		}

		// if we are not moving in the X, and it is picking up the piece, we
	  // need to offset the hardware
		else if (firstMove)
		{
			motor[X_MOTOR] = X_MOTOR_POWER;
			wait1Msec(X_H_OFFSET);
			motor[X_MOTOR] = 0;
		}

		// move the y axis forward slightly to prevent motor encoder errors
		motor[Y_MOTOR] = -Y_MOTOR_POWER;
		wait1Msec(Y_ENCODER_OFFSET);

		// move the motor encoder to the correct location
		motor[Y_MOTOR] = Y_MOTOR_POWER * directionY;
		if (directionY == 1)
		{
			while(abs(nMotorEncoder(Y_MOTOR)) > abs(Y_CELL_CLICKS * y)
			  && nMotorEncoder(Y_MOTOR) < 0)
			{ }
		}
		if (directionY == -1)
		{
			while(abs(nMotorEncoder(Y_MOTOR)) < abs(Y_CELL_CLICKS * y)
			  && nMotorEncoder(Y_MOTOR) < 0)
			{ }
		}

		motor[Y_MOTOR] = 0;
}

// picking up the peice when the claw is in place
void pickUpPiece(int x2, int y2)
{
		// if we have a king or queen, we need to lower the claw less
		int lowerDistance = CLAW_LOWER_CLICKS;
```

```
      if (stringFind(board[y2][x2], "K") != -1
        || stringFind(board[y2][x2], "Q") != -1)
          lowerDistance = CLAW_LOWER_CLICKS_TALL;

      // make sure the claw is open
      setGripperPosition(CLAW_MOTOR,SV_GRIPPER,CLAW_OPEN);

      // lower the claw
      nMotorEncoder[CLAW_ACTUATION_MOTOR] = 0;
      motor[CLAW_ACTUATION_MOTOR] = -CLAW_ACTUATION_POWER;
      while(abs(nMotorEncoder[CLAW_ACTUATION_MOTOR]) < lowerDistance)
      { }
      motor[CLAW_ACTUATION_MOTOR] = 0;

      // close the claw
      setGripperPosition(CLAW_MOTOR,SV_GRIPPER,CLAW_CLOSE);
      wait1Msec(CLAW_WAIT_TIME);

      // raise the claw back up
      nMotorEncoder[CLAW_ACTUATION_MOTOR] = 0;
      motor[CLAW_ACTUATION_MOTOR] = CLAW_ACTUATION_POWER;
      while(abs(nMotorEncoder[CLAW_ACTUATION_MOTOR]) < lowerDistance)
      { }
      motor[CLAW_ACTUATION_MOTOR] = 0;
}

// putting down the peice when the claw is in place
void putDownPiece(int x2, int y2)
{
      // change lower distance depending on the peice (king / queen)
      int lowerDistance = CLAW_LOWER_CLICKS;
      if (stringFind(board[y2][x2], "K") != -1
        || stringFind(board[y2][x2], "Q") != -1)
          lowerDistance = CLAW_LOWER_CLICKS_TALL;

      // lower the claw
      nMotorEncoder[CLAW_ACTUATION_MOTOR] = 0;
      motor[CLAW_ACTUATION_MOTOR] = -CLAW_ACTUATION_POWER;
      while(abs(nMotorEncoder[CLAW_ACTUATION_MOTOR])
          < lowerDistance-CLAW_ACTUATION_OFFSET)
      { }
      motor[CLAW_ACTUATION_MOTOR] = 0;

      // open the claw
      wait1Msec(CLAW_WAIT_TIME);
      setGripperPosition(CLAW_MOTOR, SV_GRIPPER, CLAW_OPEN);
      wait1Msec(CLAW_WAIT_TIME);

      // raise the claw back up
      nMotorEncoder[CLAW_ACTUATION_MOTOR] = 0;
      motor[CLAW_ACTUATION_MOTOR] = CLAW_ACTUATION_POWER;
      while(abs(nMotorEncoder[CLAW_ACTUATION_MOTOR])
        < lowerDistance-CLAW_ACTUATION_OFFSET)
      { }
      motor[CLAW_ACTUATION_MOTOR] = 0;
}

// function to capture a piece at a location
void capturePiece(int x2, int y2)
{
      // move to the cell with the captured piece
      moveToCell(0,0,x2,y2, true);

      // pick up the piece
      pickUpPiece(x2,y2);

      // move to the capture location
      moveToCell(x2,y2,END_X,END_Y, false);

      // drive a bit further, off the board
      motor[X_MOTOR] = X_MOTOR_POWER;
      wait1Msec(CAPTURE_WAIT_TIME);
      motor[X_MOTOR] = 0;

      // put the piece back down
      putDownPiece(x2,y2);
      return;
}

// execute move fucntion
bool movePiece(int x1, int y1, int x2, int y2)
{
      // if the start location and end location are the same, invalid move
      if ((x1 == x2 && y1 == y2) )
      {
            displayBigTextLine(1, "Invalid Move");
            displayBigTextLine(3, "Try again");
            wait1Msec(3000);
            return false;
      }

      // if the board has another piece here, capture it
      if (board[y2][x2] != "")
      {
            capturePiece(x2,y2);
            zero();
            wait1Msec(300);
      }

      // move to the location of the piece
      moveToCell(0,0,x1,y1, true);

      // pick it up
      pickUpPiece(x1,y1);
      // move to the destination cell
      moveToCell(x1,y1,x2,y2, false);

      // put down the piece
      putDownPiece(x1,y1);

      // update the board with the new location of the piece
      board[y2][x2] = board[y1][x1];
      board[y1][x1] = "";

      // zero the system again
      zero();

      return true;
}

// initializes the board with the starting location of all the peices
void boardInitState()
{

      // loops through every index in the board array
  for (int row = 0; row < BOARD_SIZE; row++)
  {
    for (int col = 0; col < BOARD_SIZE; col++)
    {

      // start with nothing, append to string
      string value = "";

      // row 0 and 1 are white, 6 and 7 are black
      if (row <= 1)
        value = "W";
      else if (row >= 6)
        value = "B";

      // if row 1 or 6, it is a pawn
      if (row == 1 || row == 6)
        value = value + "P";

      // otherwise...
      else if (row == 7 || row == 0)

            // rook in corners
        if (col == 0 || col == 7)
          value = value + "R";

        // then knight
        else if (col == 1 || col == 6)
          value = value + "N";

        // then bishop
        else if (col == 2 || col == 5)
          value = value + "B";

        // queen
        else if (col == 3)
          value = value + "Q";

        // king
        else if (col == 4)
          value = value + "K";

      // update the board with these values
      board[row][col] = value;
    }
  }
}

// when the user wants to shut down
void shutDownProcedure(bool whiteLoses, int endCode)
{
      // display the winner
      if (whiteLoses)
            displayBigTextLine(2,"Black Wins!");
      else
            displayBigTextLine(2,"White Wins!");

      // display how they won (different codes)
      if (endCode == 0)
            displayBigTextLine(4,"By Resignation");
      if (endCode == 1)
            displayBigTextLine(4, "On Time");

      // zero the system
      zero();

      // wait 5 seconds before the program ends
      wait1Msec(5000);
}

// main function
task main()
{

      // configure all the sensors
      configureSensors();

      // initalize the board with the starting values
      boardInitState();

      // chess timers, seconds left
      int whiteTime = CHESS_CLOCK_INIT;
      int blackTime = CHESS_CLOCK_INIT;
```

```
// keep track of the players turn
bool whiteTurn = true;

// while loop to continue making moves
bool playing = true;
while(playing)
{

      // reset the timer
      clearTimer(T1);

      // display how much time the user has left
      if (whiteTurn)
            displayBigTextLine(7,"Time Left:%d:%d", whiteTime / 60,
                         whiteTime % 60);
      else
            displayBigTextLine(7,"Time Left:%d:%d", blackTime / 60,
                         blackTime % 60);

      // initalize the input variables
      int x1,y1,x2,y2;

      // get the input, determine if the user wants to continue
      playing = getInput(x1,y1,x2,y2);

      // update the chess clock
      if (whiteTurn)
            whiteTime -= time1[T1] / 1000;
      else
            blackTime -= time1[T1] / 1000;

      // if the user is out of time, go to shut down procedure
      if (whiteTime == 0 || blackTime == 0)
      {
            playing = false;
            shutDownProcedure(whiteTurn, 1);
      }
      else
      {
            // if the user wants to stop, shut down
            if (!playing)
                  shutDownProcedure(whiteTurn, 0);
      }

      // if we are continuing to play...
      if (playing)
      {

            // zero the board
            zero();
```

```
                // move the piece to the desired location
                movePiece(x1,y1,x2,y2);

                // zero the board again
                zero();

                // update who's turn it is
                whiteTurn = !whiteTurn;
            }
        }
}
```

Appendix B – Robot Photos

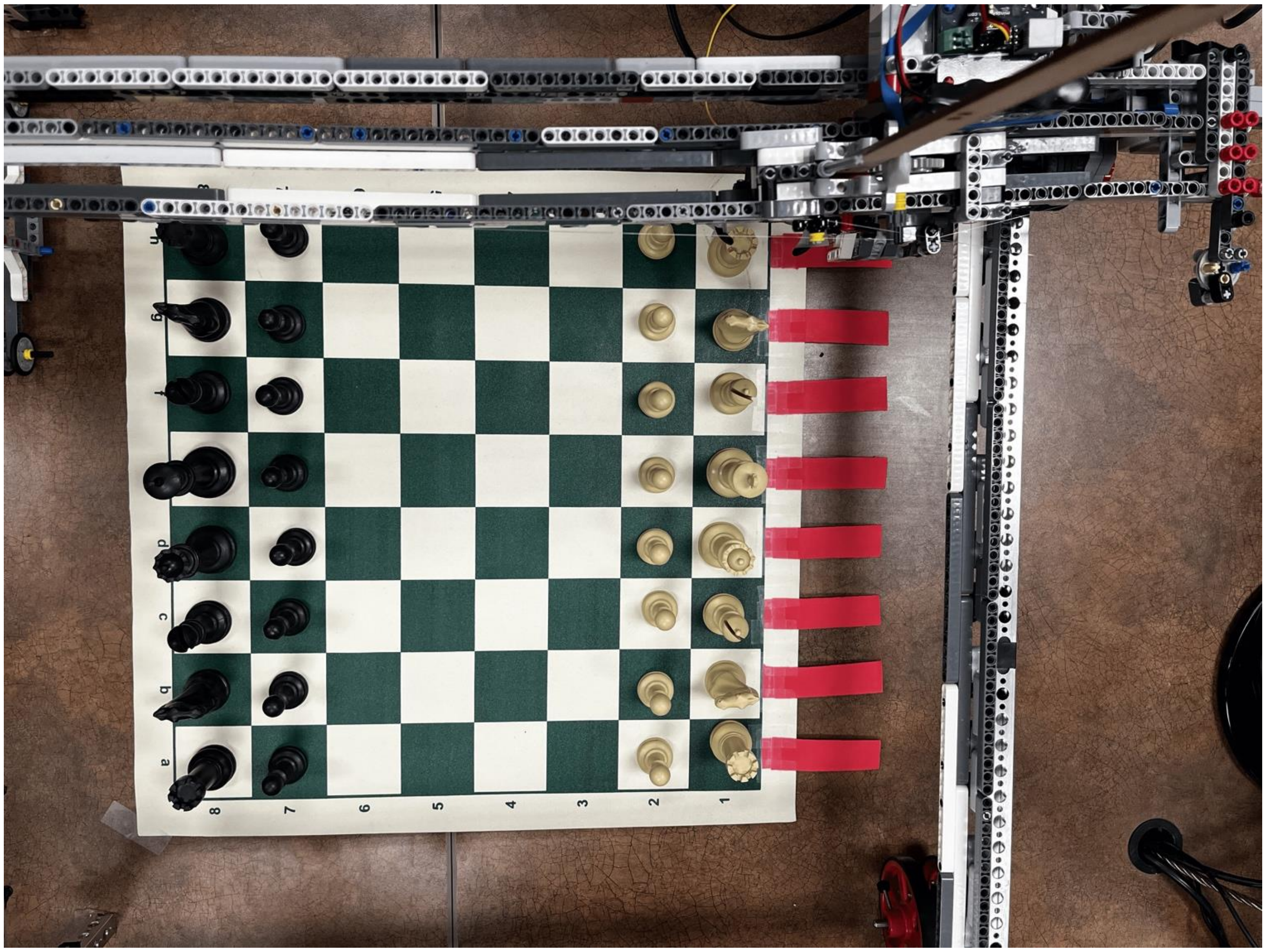

Figure 15: Overhead View

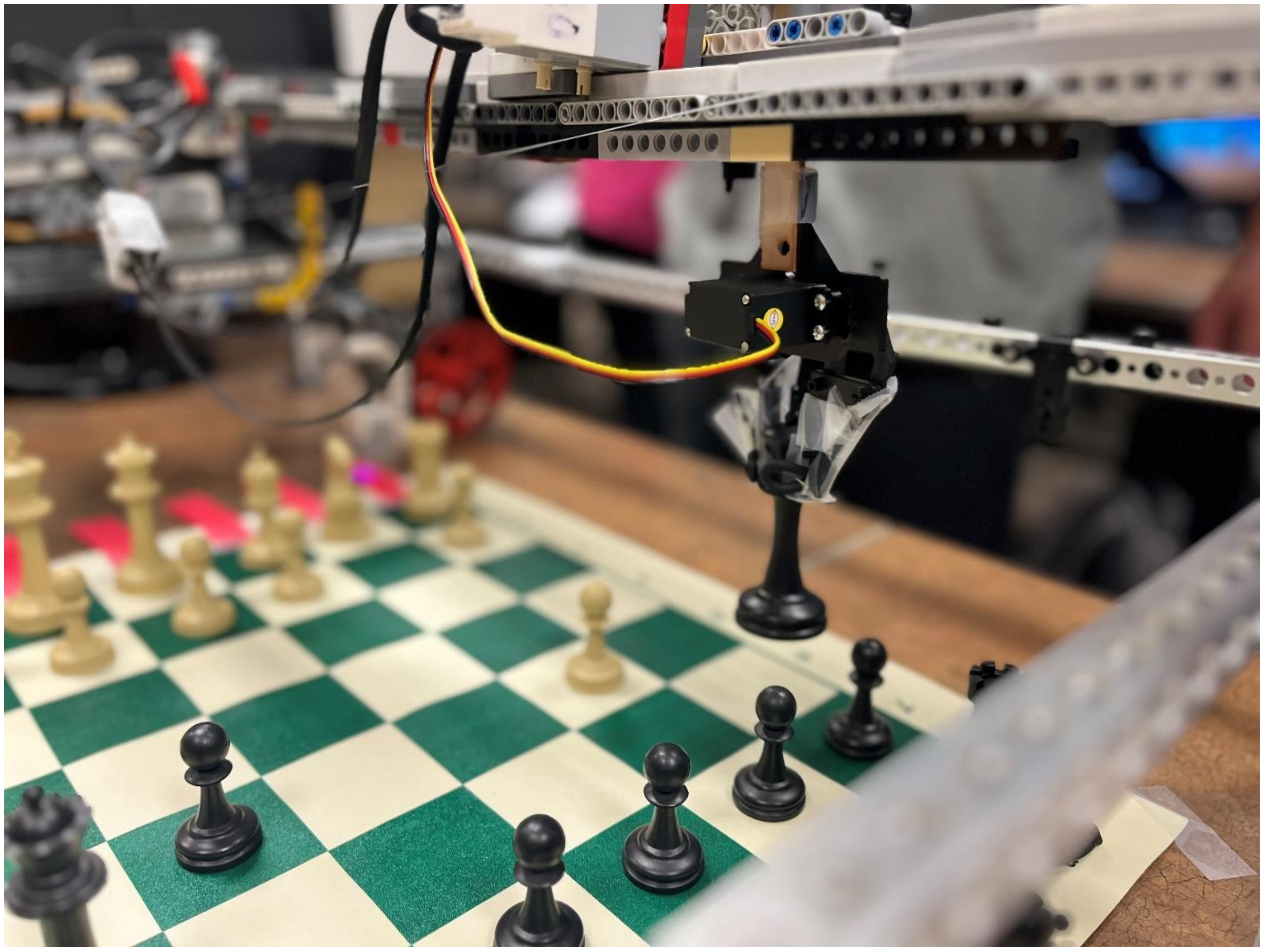
Figure 16: Claw Raised

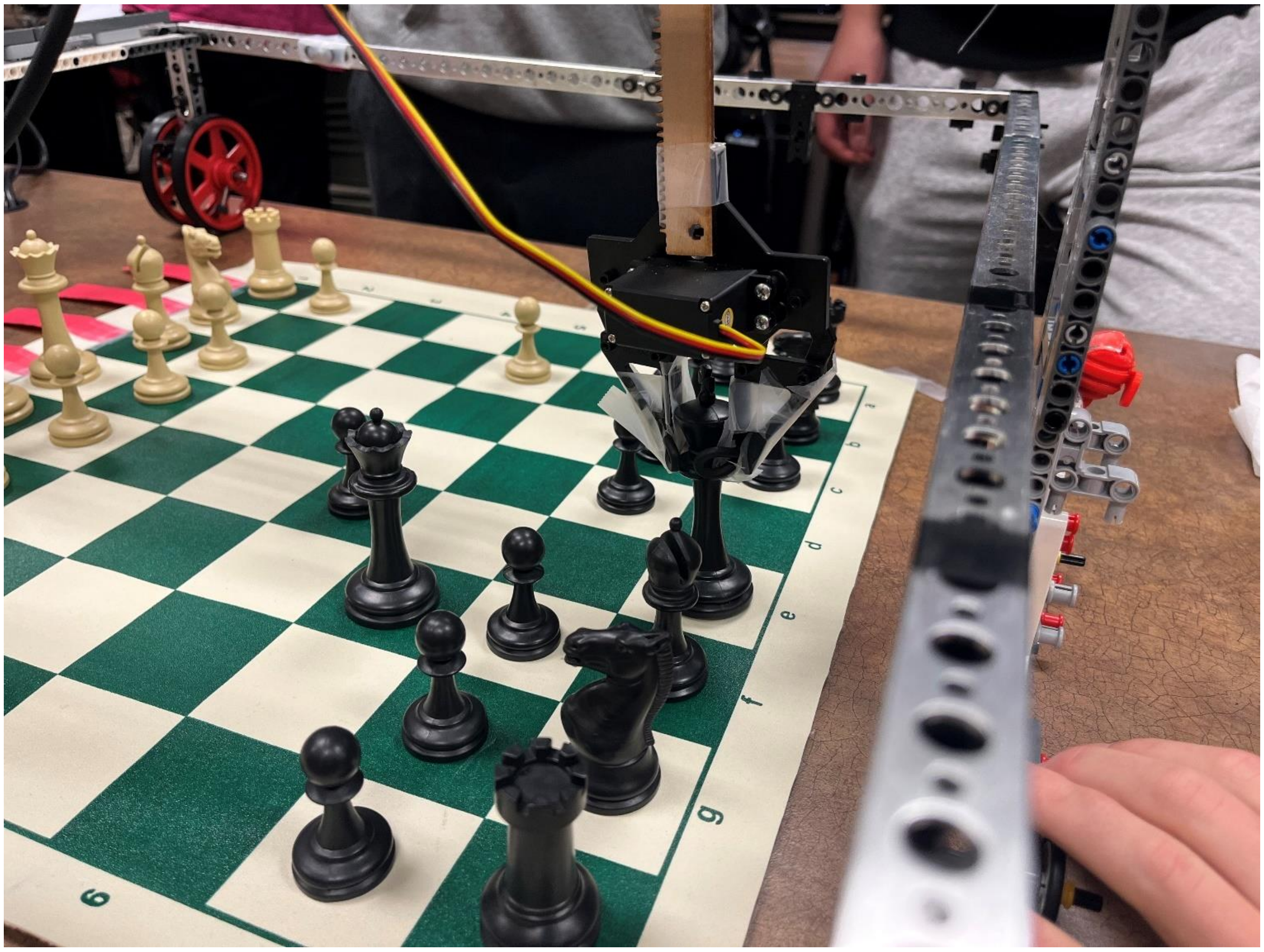

Figure 17: Claw Lowered

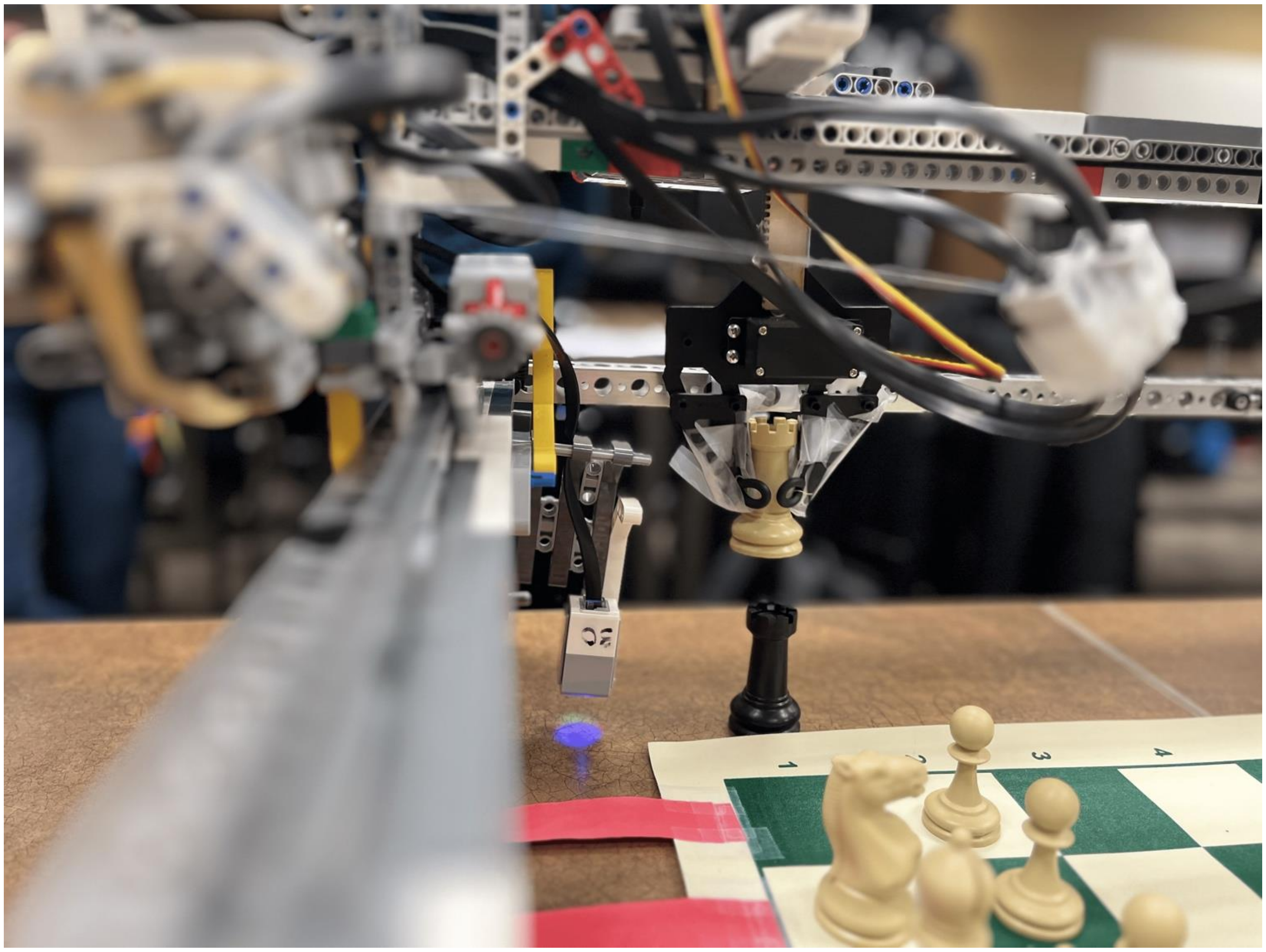

Figure 18: Capturing a Piece

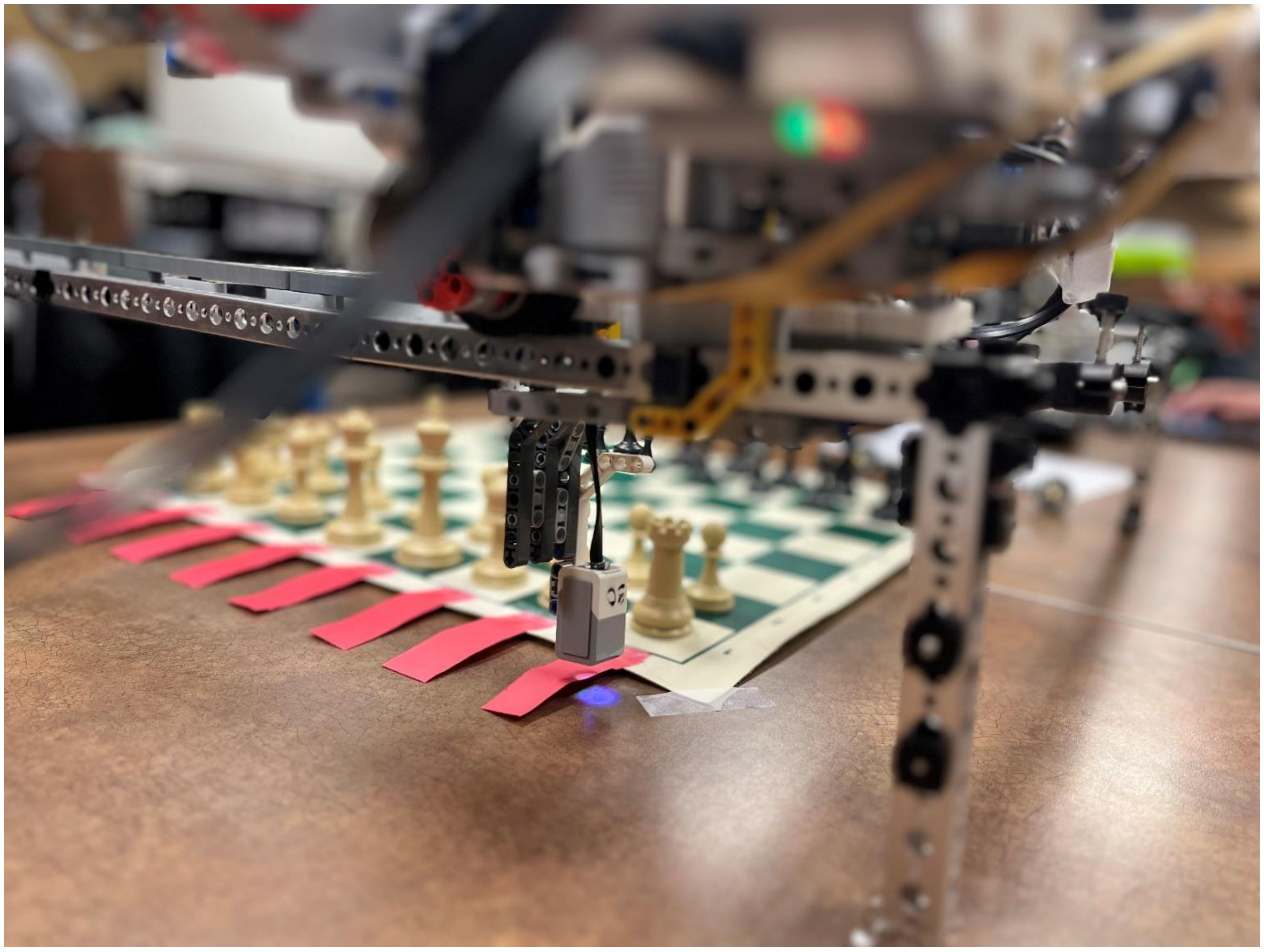

Figure 19: Color Sensor Integration

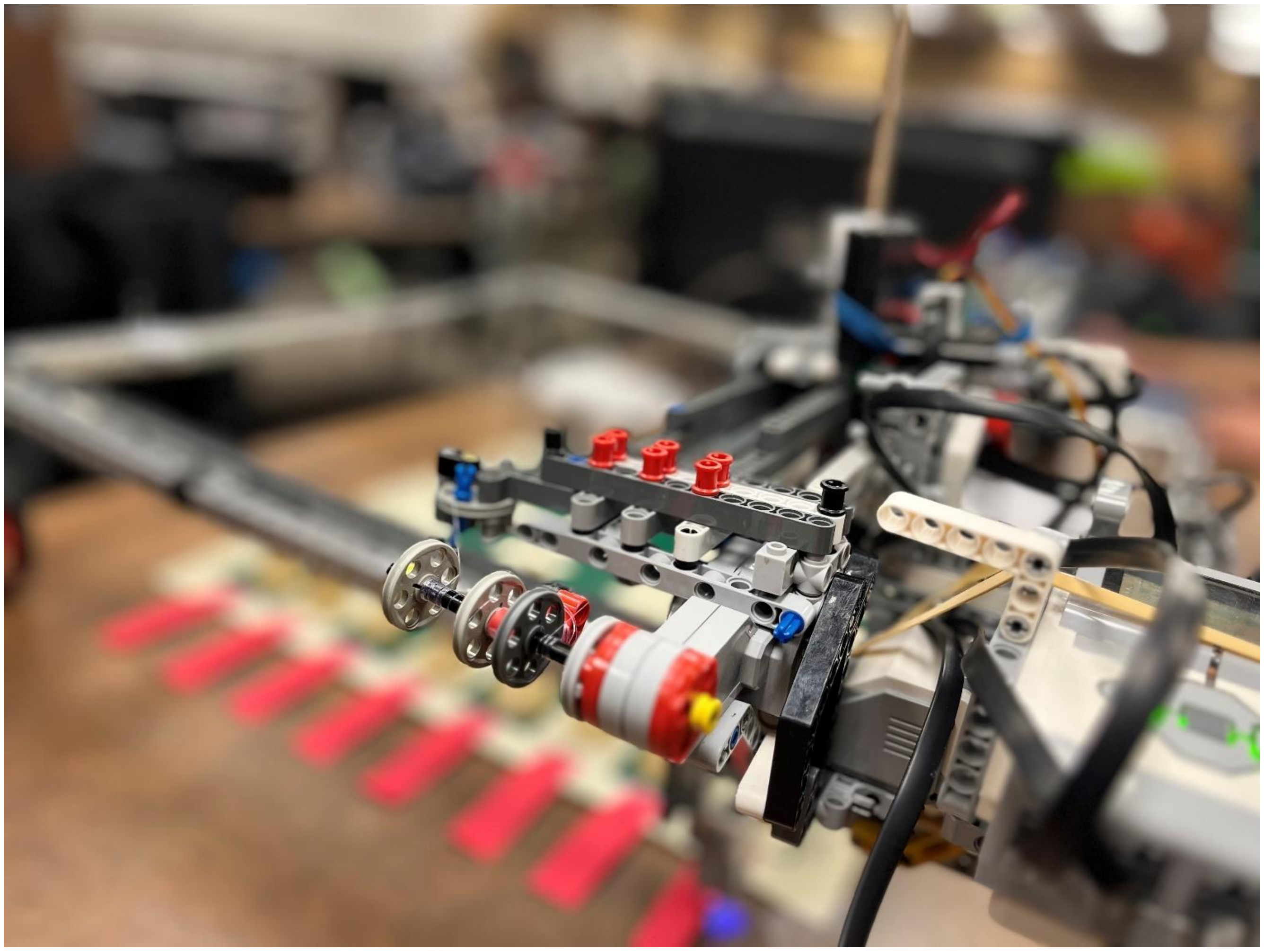

Figure 20: Pulley System

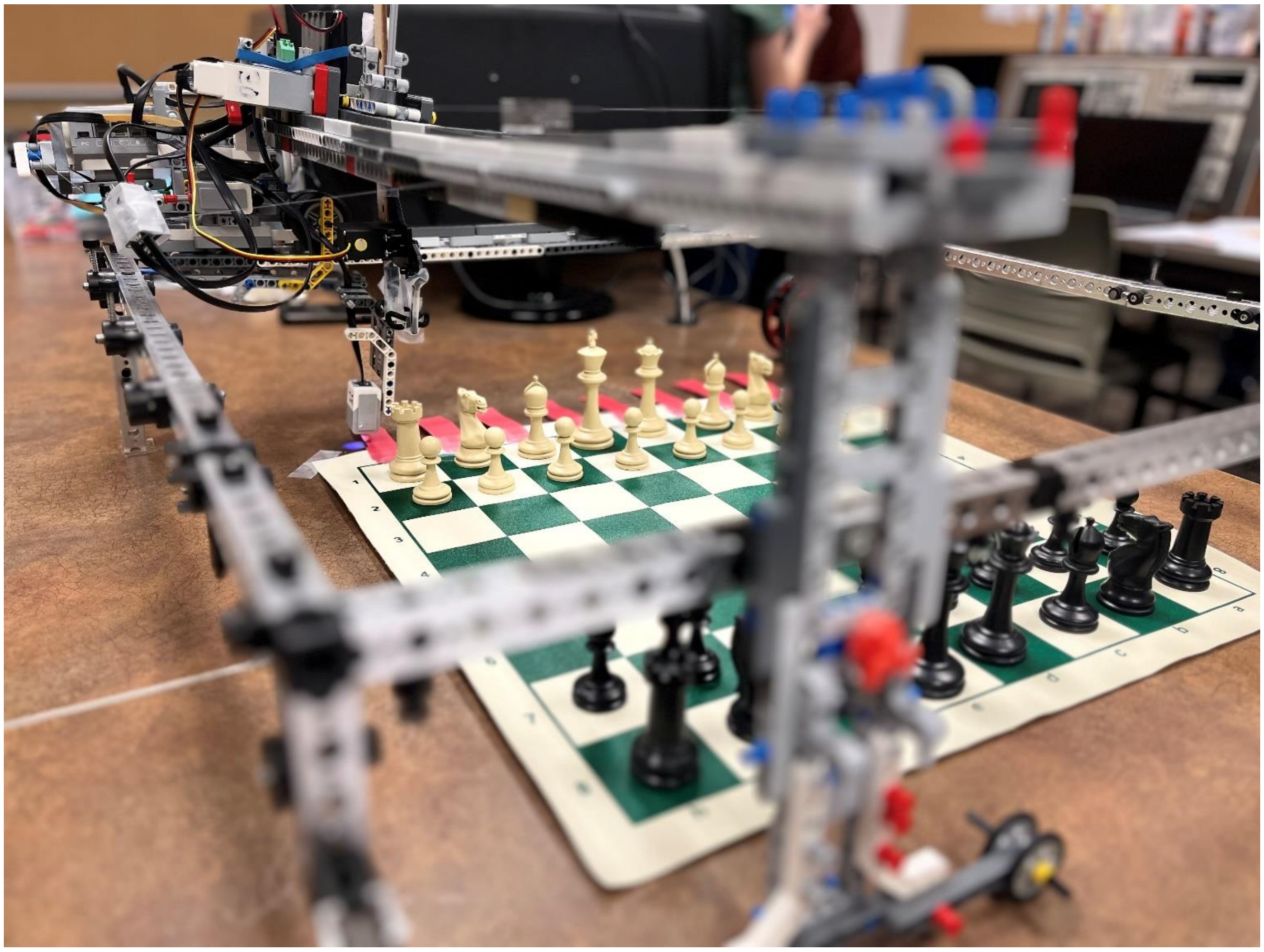

Figure 21: Main Assembly

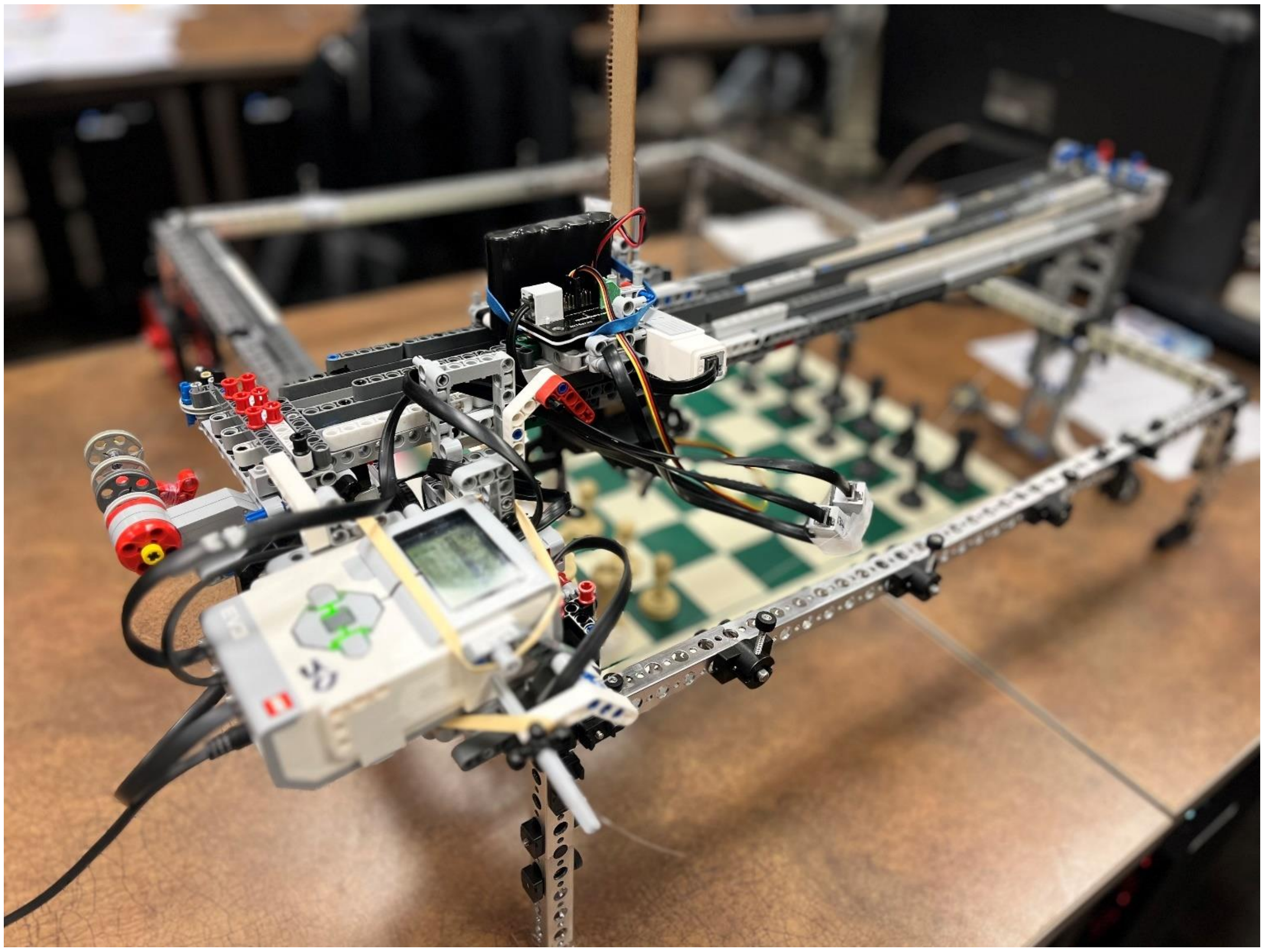

Figure 22: Main Assembly (2)

Figure 23: Team Photo!