MINISTRY OF EDUCATION AND RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

ROBOTIZED CHESS TRAINING SYSTEM

LICENSE THESIS

Graduate: Andrei Fărcaș
Supervisor: Șl. dr. ing. Tassos Natsakis

2024

MINISTRY OF EDUCATION AND RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DEAN,
Prof. dr. eng. Mihaela DINSORENU

HEAD OF DEPARTMENT,
Prof. dr. eng. Honoriu VALEAN

Graduate: **Andrei Fărcaș**

ROBOTIZED CHESS TRAINING SYSTEM

1. **Project proposal:** *Robotized Chess System consisting of an automated chessboard with an integrated Android companion application*
2. **Project contents:** *Introduction-Project Context, Bibliographic Research, Analysis and Theoretical Foundation, Detailed Design and Implementation, Testing and Validation, User's Manual, Conclusions, Bibliography*
3. **Place of documentation:** *Technical University of Cluj-Napoca, Automation Department*
4. **Consultants:**
5. **Date of issue of the proposal:** December 19, 2023
6. **Date of delivery:** July 12, 2024

Graduate: _____

Supervisor: _____

MINISTRY OF EDUCATION AND RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

Declaraţie pe proprie răspundere privind autenticitatea lucrării de licenţă

Subsemnatul(a) Fărcaș Andrei, legitimat(ă) cu CI seria CJ nr. 717528 CNP 5010816125795, autorul lucrării "ROBOTIZED CHESS TRAINING SYSTEM" elaborată în vederea susţinerii examenului de finalizare a studiilor de licenţă la Facultatea de Automatică şi Calculatoare, Specializarea Automatică si Informatică Aplicată din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea de vară a anului universitar 2023-2024, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

În cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă.*

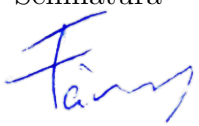
Data

11.07.2024

Nume, Prenume

Fărcaș Andrei

Semnătura



Contents

Chapter 1

Introduction - Project Context

Widely regarded as the greatest board game ever created, chess dates back to the 7th century and remains one of the most popular games globally. Its origins trace back to ancient games out of which the closest relative is chaturanga, an ancient Indian war game, which featured different pieces, each with special powers, laying the groundwork for the roles seen in modern chess.

The rules and appearance of chess pieces evolved gradually, with significant milestones such as the introduction of pawn promotion and the transformation of the counselor into the powerful queen around 1475. The game's rules have largely remained the same since then, with only minor refinements made up until the 19th century. [1]

These enduring rules create a game with an almost unlimited number of possible situations, allowing chess to develop into a sport where the best players in the world train for a lifetime to compete at the highest level.

However, as with all domains of human life, technology is slowly finding its way into the game of chess. Chess engines, complex computer programs designed to analyze and choose optimal chess moves, have revolutionized the way players study and prepare for games. These engines can evaluate millions of positions per second, providing strategies that were previously extremely difficult to find just by human analysis. Additionally, chess-playing robots have recently emerged, bringing a physical dimension to AI-powered chess and allowing for a combination of traditional gameplay and the advantages brought by technology.

It was around 1967 when the first computers started to compete against humans. One of the first computer programs to be tested in a U.S. Chess Federation tournament was MachHack VI, a program written by Richard Greenblatt, which has obtained a USCF rating of over 1200 in February 1967. By 1970, the first official computer chess championship had already been held in New York City. [1]

In 1980, the first commercially available chess robot, named Boris HANDroid, was introduced (Figure 1.1). This robot was capable of playing a chess game by independently moving pieces with a special grasping arm. However, its high price and production delays made it commercially unsuccessful, with only a few units ever produced. Despite this, the

Boris HANDroid represents a significant milestone in the evolution of automated chess. [2]

By 1997 it was clear for most that humans can no longer compete with computers when the then-reigning World Chess Champion Garry Kasparov, considered one of the greatest chess players of all time, has lost a six-game match against the Deep Blue chess-playing computer developed by IBM.[3]

Figure 1.1: HANDroid Chess Robot, 1980
Source:
`https://en.chessbase.com/portals/all/2022/07/robots/boris-handroid.jpg`

Nowadays, chess playing robots are becoming increasingly popular. This project aims to analyze existing solutions and develop a new automated chessboard, achieving the optimal balance between affordability and performance.

This thesis project is about designing and implementation of an automated chess system which consists of a smart chessboard and an integrated android companion application. The project is structured as following: Chapter 2, the bibliographic research, provides a comprehensive analysis of current solutions and existing robotic systems. Chapter 3 presents the objectives set forth at the project's inception, while Chapters 4 and 5 are dedicated to the project's implementation. Chapter 4 analyzes various preferred solutions from a theoretical point of view. Chapter 5 details the practical execution of the final project. The final chapters assess the project's performance, include a user manual, and present the final conclusions and closing remarks.

Chapter 2

Bibliographic research

2.1 Chess Engines

A chess engine is a computational software designed to evaluate chess positions and determine optimal moves. These engines are indispensable for the functioning of automated chess applications and typically operate via a command line interface. Therefore, for practical application, they are typically integrated with a graphical user interface or a physical interface such as a robotic arm or an automated chessboard.

Some of the fundamental techniques chess engines use are:[4]

1. **Search algorithms:** Chess engines use complex search algorithms that allow the engine to explore potential moves by evaluating future positions. The probability of winning is maximized by minimizing potential losses and improving the position as much as possible.

2. **Evaluation functions:** These functions are responsible for evaluating the strength of a certain position. This is achieved by giving ratings for different aspects of the game such as material balance, king safety, control of central squares and pawns placement.

3. **Endgame tablebases:** An endgame position has limited pieces and limited possibilities, so many chess engines use a databases to choose the optimal move effectively.

4. **Opening Book Moves:** Contrary to the endgame, at the opening there are infinite possibilities. However, similar to the endgame, databases with ideal moves for the opening phase are used to play the first moves efficiently.

5. **Machine learning techniques:** These methods are used to improve the evaluation functions of chess engines. By analyzing previous games and outcome of different moves, they can adjust certain parameters in the evaluation functions and provide better results.

A large number of chess engines are available. Some of the most performant and popular options are presented in the following subsections.

2.1.1 Stockfish

Stockfish is an open-source chess engine that is widely regarded as being one of the strongest in the world. The engine is written in C++ and is using a combination of search algorithms to examine and evaluate positions. These algorithms include the Alpha-Beta Pruning which reduces the number of nodes in the search tree by eliminating the branches which seem unlikely to contain the best move. It also uses an algorithm designed to take advantage of multi-core processors by allocating a separate portion of the search tree to each core of the processor.[4]

Besides being extremely performant and open-source, the popularity of Stockfish is also due to it being compatible with UCI (Universal Chess Interface), a protocol used by many graphical user interfaces to communicate with the engines in a standardized way.

2.1.2 AlphaZero

AlphaZero is a chess engine developed by DeepMind, an AI company owned by Google. It is written in C++ and Python and is not available for public use. However, it is famous for managing to defeat the Stockfish chess engine after only four hours of training. In a series of 100 games, the AlphaZero managed to get 28 wins, 72 draws and 0 losses.[5] The AlphaZero has been discontinued shortly after and the exact code that stays at the base of the chess engine has been kept a secret so far.

However, some concepts that are used by AlphaZero are well known. The main difference compared to Stockfish is in the way they evaluate positions. Stockfish uses a rule-based evaluation function while AlphaZero uses a deep neural network that is trained to predict the outcome of the game from different positions.[4]

2.1.3 Leela Chess Zero

Leela Chess Zero is also an AI based chess engine, which was inspired by the AlphaZero. However, being launched in 2018, Leela Chess Zero is open source, making it one of the most popular and strongest of the modern chess engines. Lc0 has also managed to win games against Stockfish after only days of training in which the chess engine has played millions of games against itself. What sets this engine apart from others is that it uses multiple Neural Networks, each with its own strengths and weaknesses, and enabling players to choose a network that aligns best with their specific chess training objective. [6]

Just as Stockfish, Leela Chess Zero is also available on GitHub and uses the UCI command line interface to communicate its moves to graphical user interfaces. Stockfish and Leela Chess Zero are often placed against each other and for the majority of matches

the latest version of Stockfish is still winning, making Stockfish the best chess engine publicly available to this date acoording to most chess engines ratings.

2.2 Chess Playing Robots

In the following subsections, some of the existing solutions for chess playing robots are presented. These showcase the large panel of options engineers can choose from when deciding how to build their task of automating the chess game.

2.2.1 MarineBlue Chess Robot

The MarineBlue chess robot [7] is a low-cost, configurable robotic system designed to play chess autonomously. The implementation of this robot involves several key components that must work together in order to achieve the project's purpose: the robotic arm, the vision system, the chess engine, and the control algorithms that integrate these systems.

The MarineBlue robot utilizes the Robix RC-6 robotic arm, which is composed of multiple segments that can move independently using servos. For this specific application, three segments of the arm move horizontally in the plane parallel to the chessboard, while a fourth segment moves vertically to lift and place chess pieces, by using a special gripper attached to the last segment of the arm.

A critical component of the MarineBlue system is the vision system, which uses a high-quality Sony DFW-VL500 camera mounted approximately one meter above the chessboard. This positioning minimizes perspective distortion and allows the camera to capture clear images of the entire board.

The control of the robot arm is managed through a series of algorithms that translate high-level commands into low-level servo movements. The inverse kinematics algorithm plays a vital role in this process, being responsible for calculating the necessary angles for each servo based on the desired position of the gripper.

The system's software architecture is modular, consisting of three primary modules: vision, chess engine, and robot control. This modular design allows for easier maintenance and potential upgrades, such as incorporating more precise robotics or enhanced vision capabilities.

The authors of paper [7] have chosen to use the camera only for detecting the squares occupied by white or black pieces and not the exact type of piece. The reasoning behind their choice is that an image captured from above of the chessboard only depicts circular shapes for all the pieces, making it impossible to develop a reliable algorithm for piece type detection. The method of keeping track of the chess position should provide good results but does not justify entirely the cost of adding a dedicated camera for this process, as there are other methods of detection used by projects presented in the following subsections.

2.2.2 Reed Sensor Based Detection on Chess-Playing Robotic Systems

The chess-playing robotic system designed by Al-Saedi and Mohammed is intended to autonomously compete against human opponents[8] . The core of this system is the Lab-Volt 5150 robotic manipulator, a five degrees of freedom robotic arm, which is controlled through a networked control system and can be seen in figure **??**.

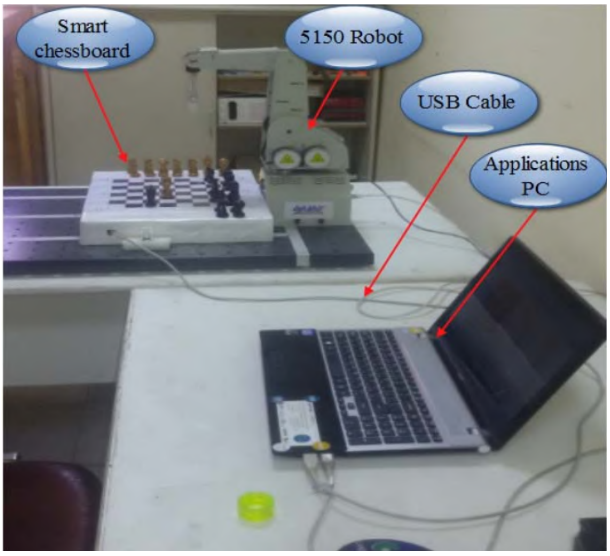


Figure 2.1: Chess playing robot presented in [8]

A key technology used by this system is the "SharpChess" application, a free and open-source chess program modified to meet the project's requirements. SharpChess provides the board representation and search techniques needed for the robot to understand and respond to the game state. It operates in a Visual C environment, enabling seamless integration with the robotic arm's control system.

The results obtained are similar to those presented in the paper from presented in subsection 2.2.1, but the main difference is in the choice of how to detect chess pieces placement. For this project, the chessboard itself features reed switches to track the opponent's movements, acting as an input device that feeds data into the chess engine. This setup requires a reed switch below each chess square and magnets strong enough to trigger them placed at the bottom of the chess pieces. Considering the affordability of reed switches, this approach improves the costs by eliminating the need of a dedicated camera.

2.2.3 Chess Game Tracking using Computer Vision

In [9], Maciej et al propose a more versatile method of detecting chess positions by using Computer Vision. The main advantage compared to the previous methods is the capability to detect the chess position directly, without the need for starting from the initial position and tracking all the moves one by one.

The method developed by the authors is capable of detecting the chessboard and the positions and typed of chess pieces with an accuracy of 95 percent. This was achieved by using computer vision methods such as an improved chessboard lattice point detector and using advanced line detection with segment merging to detect and process chessboards captured at different conditions and angles. Additionally, machine learning techniques were incorporated to enhance classifier accuracy at various stages of the algorithm.

In [10] Jialin Ding et al. is showcasing how using descriptors such as Scale-Invariant Feature Transform (SIFT) and Histogram of Oriented Gradients (HOG), the chess pieces and their positions can be identified. Support Vector Machines were used to train binary one-vs-rest classifiers for each piece type. Piece classification was successfully implemented by using a modified sliding window technique, where classifiers produced probability estimates for each board square, and determining the piece type based on the highest probability.

The conclusion is that computer vision can be used for detecting chess positions without the need of tracking every move from initial positions. The build is also simpler than in the case of detection by using classical electric circuits because it only requires a camera of sufficient quality. However, the processing power needed is considerably larger and the accuracy of 95 percent for the most performant models might not be satisfactory for all projects.

2.2.4 Using an Electromagnet for Moving Pieces

The solution that closest matches the vision of the project presented by this paper is the one presented in document [11]. The documentation presents an Automated Chessboard that is defined by the following elements: move detection implemented by using Reed Switches placed under the board; Moves executed by a Cartesian structure and an electromagnet; A chess engine implemented in C++ that is executed on the Arduino Development board.

The system implemented has managed to successfully keep track of chess positions and execute moves. However, even though remarkable performances have been obtained, some negatives can be observed. Firstly, the chess engine choices are limited by the Arduino's computational power, causing poor move choices from the chessboard. The user interface is provided by a small LCD screen and two buttons, which is also directly controlled by the Arduino. Secondly, captured pieces do not have a designated place, meaning the human player will have to observe the capture attempt and remove manually the piece from the edge of the board.

The above mentioned drawbacks can be solved by running the chess engine and the user interface on an external device and designing the chessboard to provide dedicated places for captured pieces. Also, improvements of the reliability of the reading process might be possible with the replacement of the reed switches by Hall Effect Sensors, which have no moving parts and may be more suitable for this application.

Chapter 3

Project Objectives and Specifications

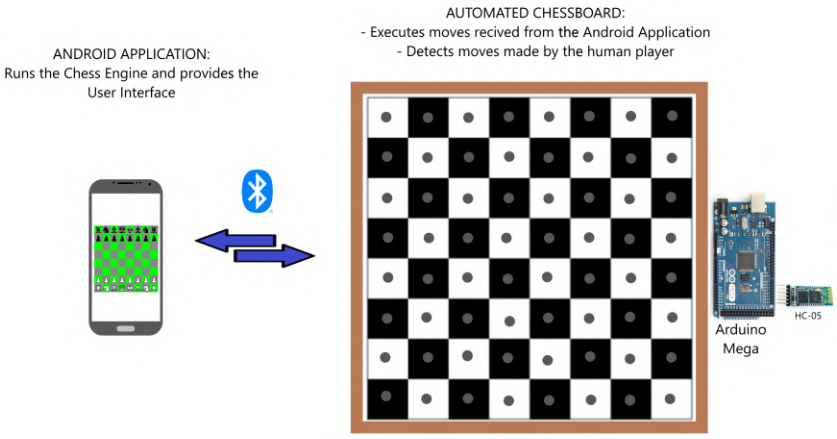


Figure 3.1: Project Objective

The aim of this project is to design and develop an automated chessboard that autonomously moves chess pieces using an electromagnet mechanism located beneath the board. This system will be integrated with a mobile application built for Android devices, where user interactions and game logic will be handled. Figure 3.1 depicts the two main components that together constitute the objective of this project. A comprehensive list of objectives and specifications can be found in the following subsections.

3.1 Objectives

1. **Design and Development of the Automated Chessboard:**
    - Create a physical chessboard with integrated sensors to detect the positions of chess pieces.

- Implement an electromagnet mechanism capable of accurately moving chess pieces based on commands.
- Ensure smooth and precise movement of chess pieces, replicating human-like interactions.

2. **Design and Development of Embedded Software for the Automated Chessboard**
   - Develop firmware to manage the sensors and control the electromagnets.
   - Implement communication protocols for data exchange between the chessboard and the Android application.
   - Optimize the embedded software for real-time performance and reliability.
3. **Design and Development of an Arduino Chess application with a Chess Engine support**
   - Develop a user-friendly Android application that provides a seamless interface for playing chess.
   - Integrate the Stockfish chess engine to generate legal moves, provide hints, and allow game play at different levels.
   - Include features such as move validation, game history tracking, and move suggestions.
4. **Integration of the Android Application with the Chessboard**
   - Ensure real-time communication and synchronization between the chessboard and the Android application.
   - Provide a robust and secure connection to enhance user experience.

3.2 Hardware Specifications

1. **Chessboard**
   - The chessboard should provide a standard 8x8 layout with integrated Hall Sensors under each square for pieces presence detection.
   - On the left and right sides of the board, an extra 32 spaces should be available to allow captured pieces to be stored. These places should also have sensors for magnetic presence detection.
2. **Pieces Movement Mechanism**
   - An electromagnet of sufficient strength and precision should be included such that precise control over the pieces can be obtained

- The chessboard should implement a CoreXY system actuated by two stepper motors that ensure precise positioning of the electromagnet on a sufficiently large working envelope

3. **Arduino Mega**
   - First role of the Arduino Mega is to manage sensor inputs. Due to the limited number of pins, six 16:1 multiplexers will be used.
   - A wireless communication module will be used for connecting with the Android Application
   - A transistor will be used to control the electromagnet.
   - Two motor drivers will also be connected. One for each stepper.

3.3 Software Specifications

3.3.1 Android Application Specifications

The Android application will serve as the primary interface for users to interact with the automated chessboard. The application will be designed with a focus on user experience, providing a seamless and intuitive way to manage Bluetooth connections, set up games, practice, and record games. Key features and functionalities of the application will include:

- A menu with options for Bluetooth connection management, Automated Chessboard game setup, free practice or game recording.
- Difficulty level selection available before starting each game.
- A user-friendly interface, including an interactive virtual chessboard that allows users to move pieces and provides move suggestions for each piece when selected.
- Communication via Bluetooth with an external device must be supported. A dedicated game mode for the Automated Chessboard will be implemented. Moves made by the Stockfish chess engine will be sent to the chessboard, and moves made by the human player on the chessboard will be received via Bluetooth and automatically updated in the application.

3.3.2 Arduino Software Specifications

The Arduino software will serve as the core of the Automated Chessboard, managing sensor data, controlling the electromagnets, and facilitating communication with the Android application. The software will be designed to ensure precise, real-time performance and reliable operation. Key features and functionalities of the Arduino software will include:

- Firmware to manage the sensors and accurately detect the positions of the chess pieces.
- Control algorithms for the electromagnets must be implemented, ensuring smooth and precise movement of the chess pieces.
- Implementation of communication protocols to enable data exchange between the chessboard and the Android application.
- Real-time performance optimization to ensure prompt and accurate execution of commands.
- Synchronization protocols to maintain the integrity of the game state between the chessboard and the Android application.

Chapter 4

Analysis and Theoretical Foundation

4.1 Hardware Analysis

The development of this project demanded extensive research and analysis on both hardware and software. This section begins with a detailed examination of hardware solutions followed by software considerations. The chessboard must offer several functionalities: it needs to detect moves made by a human player, analyze the state of the game, determine an appropriate move, and execute it by moving a piece on the physical board. Each of these functions is described in the following subsections.

4.1.1 Pieces Detection

Firstly, the project required a way to detect the position of the chess pieces on the board. Multiple practical solutions had to be evaluated. One was to include a camera and use Machine Vision for this specific task. Although promising, I decided against this due to the additional costs, more complex setup and potential disruption in the playing area caused by the presence of the camera.

The other possibility to provide this functionality is based on the idea that every chess game starts from the same initial position. Therefore, it is not needed to detect all the pieces after each move, the automated chessboard can just keep track of the move that was made and update the previous position. This can be achieved by detecting the presence of a piece on a specific square. Since only one piece will be on one square at a certain moment and only one piece can be moved at a time, it is possible to know exactly where each piece is located by updating the starting position from each point on. By following this idea, different solutions have been analyzed.

The detection can be made by using capacitive sensors, an open contact which can be closed by adding a conductive material on the bottom of each piece or by using magnetism. These options were tested, and even though the results were satisfactory, making capacitive sensors that are triggered when a human touches the piece had the big downside of needing conductive pieces. On the other hand, exposed contacts had the

downside of being exposed to corrosion and mechanical damage, while also being difficult to manufacture.

Considering the testing and the analysis that was made, another solution was tested. A magnet was added on the bottom of the piece and the presence of magnetic field from below was detected. This was deemed as the best solution and two options for magnetic field detection were evaluated: Reed switches and Hall sensors.

Reed switches (Figure 4.1) are small sensors that consist of two metallic contacts, one is free, and the other is fixed. When a magnetic field is present, the mobile contact is attracted, and the contact is closed.

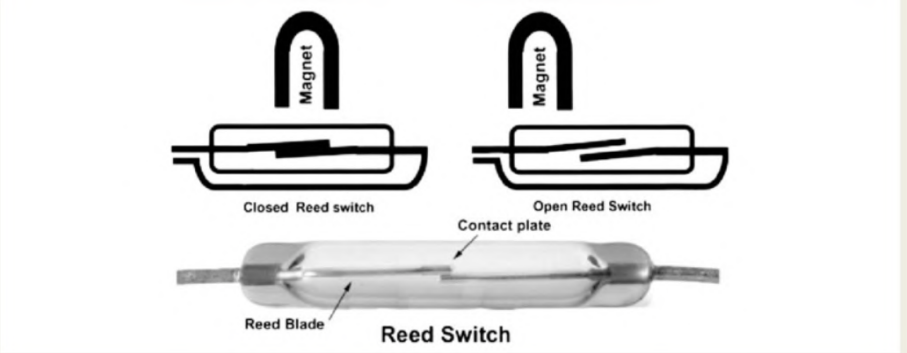


Figure 4.1: Reed Switch
Source: `https://www.researchgate.net/publication/51169357_Clinical_applications_of_magnets_on_cardiac_rhythm_management_devices`

Hall sensors are just as small, and they operate on the principle of the Hall effect: The presence of a magnetic field is detected by measuring the voltage difference generated when electric current is applied to a conductor near the presence of the perpendicular magnetic field that is being measured. This voltage can be measured to determine the strength and direction of the magnetic field around the sensor. These sensors are extremely common in multiple applications that include detecting of moving parts, including measuring the rotational speed of a motor, proximity sensing for detecting objects without contact in industrial applications or home appliances and even MRI (Magnetic Resonance Imaging) for measuring and mapping the magnetic fields.

Hall sensors can be split into two main categories: mono-polar and bipolar. Mono-polar hall sensors only detect the presence of a specific magnetic field polarity (north or south). They switch the state when the magnetic field is strong enough to exceed the threshold, and they switch back to the initial state when the magnetic field is removed. Bipolar Hall sensors can discriminate between north and south magnetic poles but require the polarity to change to trigger the toggle of the output. They are therefore suitable for detecting the change in magnetic field polarity and mainly used in sensing rotational movements where magnets of alternative polarity pass by the sensor.

Taking all this information into consideration, digital mono-polar Hall sensors are

suitable for this application because their non-latching behavior and digital properties are needed. They have been preferred over the other solutions due to having more reliable performance and lacking moving or exposed parts, which can improve the lifetime of the final product.

4.1.2 Pieces Movement

CoreXY

In order to move the pieces, an electromagnet was placed under the board. Therefore, a system that is capable of precise positioning on two axes was necessary. The most simple and obvious form of implementation would be to place a motor on each of the axes and move the center trolley as it can be seen in the Cartesian system presented in the first image of the figure 4.2.

The main advantage is the simplicity of the design, but also the simple mathematical kinematics model, since movements on X axis depend exclusively on one motor and movements on Y axis depend only on the other one. This approach works well in theory, but it has the main disadvantage that the second motor must be placed on a horizontal linear rail which takes space, adds extra weight and inertia, and also requires wires that are constantly moving to supply the motor of the horizontal axis with electricity.

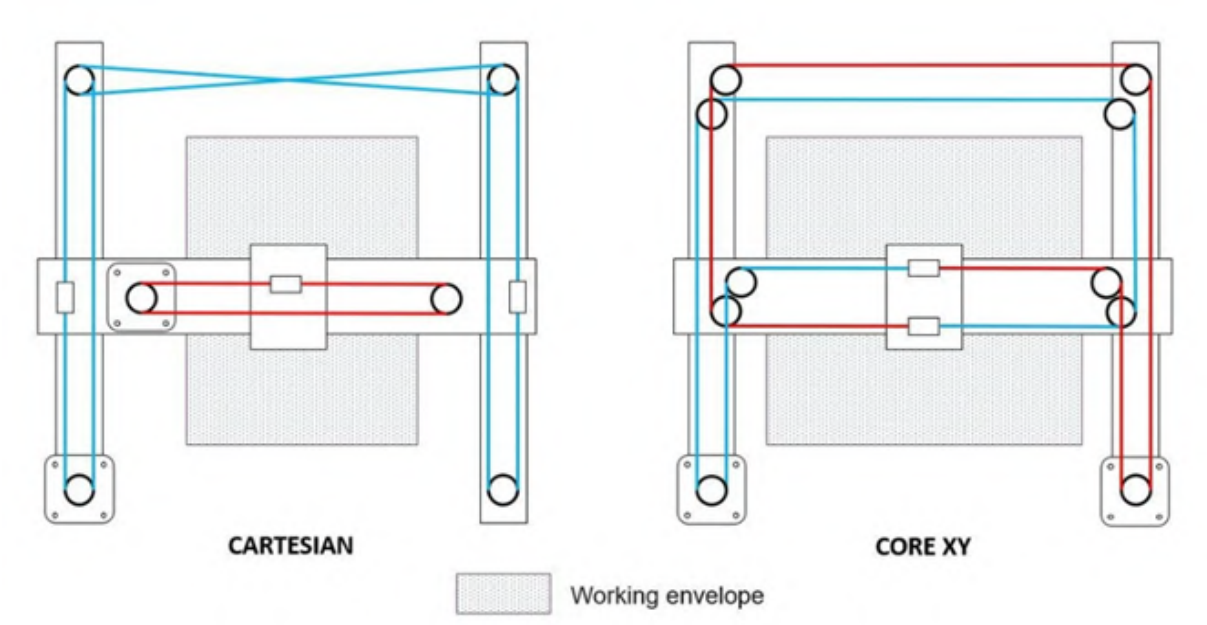


Figure 4.2: Cartesian and CoreXY Comparison
Source: `https://www.instructables.com/Automated-Chessboard/`

The CoreXY concept solves these problems. It can be seen in the second image from

figure 4.2, and it shows that the motors can be placed on a single side and achieve the same functionality. By positioning the gears and pulleys appropriately, we can determine that each motor is responsible for diagonal motions, which means that if horizontal or vertical movements are needed, both motors need to move at the same time.

To add some more details, the end effector can be moved on the main diagonal by rotating only the left motor (Clockwise for moving away from the motor and anticlockwise for moving closer to the motor). It can also move on the secondary diagonal by rotating only the right motor (Clockwise for moving away and anticlockwise for moving closer to the motor). By combining two of these movements, it is possible to create horizontal movements by using simultaneously both motors as follows. By rotating both motors in the same direction and velocity, horizontal movement is created and by rotating the motors in opposite directions but same velocity, vertical movements are made.

A detailed explanation of the implemented CoreXY system and the equations that describe the forward and inverse kinematics for this specific project can be found in the Implementation Chapter in subsection 5.2.3.

Electromagnet

One solution for gripping the chess pieces would have been to use a regular gripper placed on a robotic arm or a Cartesian robot placed outside the board.This approach would be sufficient and performant from a functional point of view. It would however protrude the playing surface and might disturb some players by changing the classic visual aspect of the game. Another downside would be the apprehension certain players might have to engage with such a system, especially in light of a recent incident in which a robotic arm broke a 7 year old child's finger in an international chess competition.[12]

The improvement then comes by moving the Cartesian structure inside the board and moving the pieces without direct contact between the element of execution and the targeted piece. This can be done by using an electromagnet placed very close to the playing surface.

An electromagnet is a type of magnet in which the magnetic field is produced by an electric current. When the current flows through a coil of wire, it generates a magnetic field which can be easily switched on or off as needed. This behavior of an electromagnet makes it a suitable choice for picking up and dropping pieces.

Also, by keeping it under the board and there will be no visible mechanisms on the playing surface, the traditional look and feel of the chessboard will be maintained, making the automation unobtrusive. The movement of the pieces without direct contact from a robotic gripper introduces an interesting and novel element to the chess game, increasing the likelihood of acceptance and use by regular chess players.

4.1.3 Development Board

Up to this point, the fundamentals of sensors and movement within the Cartesian space have been covered. However, all this information needs to be processed and a control logic has to be implemented. For this purpose a microcontroller is the most obvious choice.

Among the various development boards available, many of which could be used for this specific application, I have considered some of the ESP32 boards, such as the ones using the ESP32 Pico D4 microcontroller and the widely popular Arduino Mega development board, which is based on the ATmega2560 microcontroller.

The ESP32 Pico D4 microcontroller offers notable advantages, particularly its built-in Bluetooth and Wi-Fi modules, which facilitate wireless communication. However, its limited number of pins presents a significant drawback given our project's requirements that include 96 sensor inputs, two motor drivers, and electromagnet control operations.

Ultimately, the development board chosen for the project is the Arduino Mega, which is based on the ATmega2560 microcontroller, mainly because cost reduction was an important part in building the project. Also, there are relatively cheap options to add an external Bluetooth module and the large number of pins ensured that they were sufficient to cover the needs of the project. Moreover, the most complicated part of the system from a mathematical point of view is the Stockfish algorithm, which will be integrated in the application and run on an external device. Consequently, the Arduino only does light tasks like reading the sensors, computing paths based on simple mathematical equations and controlling the motors so it should have no issues with processing power.

4.1.4 Communication

As mentioned in the previous section, the Arduino Mega lacks wireless communication capabilities. Therefore, I had to choose a module that is compatible with the Arduino Mega and allows connection to an Android application and bidirectional communication.

For this task, the HC-05 Bluetooth Module HC-05 was chosen. This module is designed for serial communication and can be used either as "master" or "slave" device. For this project the device will be used in its "slave" mode and wait for a connection from the Android Application. Once connected, the Bluetooth module is used both for transmitting and receiving information on a turn based manner.

Serial communication is a fundamental part of interfacing the Arduino Mega with external devices. This involves sending data one bit at a time over a single communication channel. The whole process happens at a certain frequency called baud rate, which refers to the number of bits transmitted per second. It is also important to make sure both the transmitter and receiver are set to the same baud rate, which will be 9600 for this application. Otherwise, the data transmission will not work. The Arduino platform provides built-in support for serial communication via the Serial library.

4.1.5 Transistors

The project uses an electromagnet that requires 12V power supply and can draw close to 700mA of current at one time. The Arduino pins operate at 5V and can only provide a maximum of 40mA of current. This makes it obvious that the pins cannot be directly connected to the electromagnet and a separate power supply is needed, as well as a switch that can control when the electromagnet is connected and when it is disconnected from the power supply.

Transistors are semiconductor devices used to amplify electronic signals. The two main types of transistors that are generally used in electronics are PNP and NPN transistors which can be seen in Figure **??**. Both have three terminals: Collector, Base and Emitter, but their operation is different.

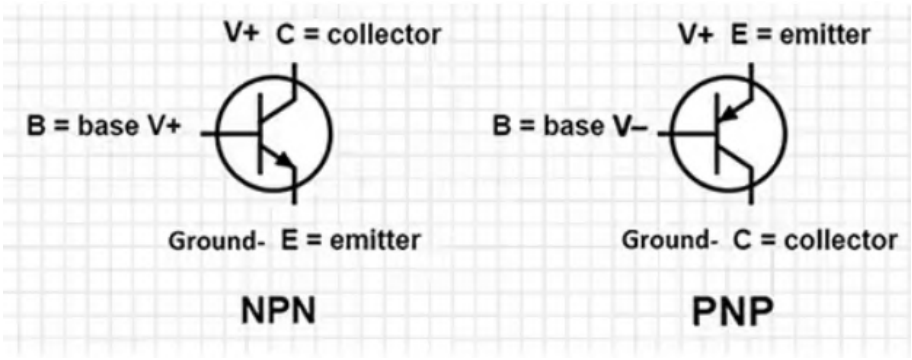


Figure 4.3: Schematic Symbols for Transistors
Source: `https://www.digikey.com/en/articles/transistor-basics`

An NPN transistor is made of two n-type (negative) semiconductor layers with a p-type (positive) layer in between them. To turn on an NPN transistor, a positive voltage is applied to the base relative to the emitter, allowing current to flow from the collector to the emitter. When the NPN transistor is on, the voltage drop from collector to emitter is minimal.

A PNP transistor, on the other hand, has two p-type layers with an n-type layer in between. To turn on a PNP transistor, a negative voltage is applied to the base relative to the emitter. When the PNP transistor is on, the voltage drop from emitter to collector is minimal. For both these transistors, when there is no current flow in the base of the transistor, the transistor is off and current flow is also restricted between emitter and collector. That is how both of these devices can act as a switch for a circuit of higher power than the control circuit.

For specific use of transistors for this application, NPN transistors are recommended because their operation matches the positive pins of the Arduino, making it easy to connect the emitter to ground, and keep the circuit as simple as possible. The 2N3904 NPN transistor was deemed a suitable choice because of its widespread availability, low cost,

and reliable performance in low-power switching applications. The implementation and circuit diagram for this part can be found in section 5.1.2

4.2 Software Analysis

4.2.1 Android development

A major component of this project involves creating an Android application to serve as a companion for the Automated Chessboard. Given that the Android operating system is widely used on smartphones and tablets, it is an ideal platform for the companion app.

Technologies Overview

The primary programming languages for Android development are Java and Kotlin. Since more than 60 percent of professional Android developers prefer Kotlin, this application is also being developed using Kotlin. Kotlin is an open source, modern, statically-typed programming language that is designed to improve coding efficiency and reduce common programming errors. [13]

Considering the recommendations from the official Android Developers website, the Jetpack Compose toolkit is used to develop the user interface for the application. Unlike the traditional Android development methods that use XML files to develop the UI, Jetpack Compose allows the programmer to describe the UI directly in the Kotlin code.

Jetpack compose provides access to UI components known as composables that are used throughout the application and have proved to be sufficient to deliver all the functionalities expected from the application. Examples of such elements are Box, Button, Text, Row, Column, Spacer and many others.

Jetpack Compose also offers exceptional tools for state management, such as "MutableState" classes, which are essential for observing and managing UI interactions such as button presses, options selection or interactions with the digital chessboard in practice mode. [14]

Architectural Analysis

In figure 4.4, the general structure of an Android application built with Jetpack Compose can be seen. The main architectural pattern used in development of Android Applications is the Model-View-View Model. [15] A similar approach is recommended for the chess application. The Data Layer in general is responsible to store, retrieve and expose app data. In the case of this project, the Data Layer holds objects such as the chess pieces images and longer pieces of text, as well as provide storage for chess positions and different options selected by the user while navigating the application.

Furthermore, in the Data Layer, an integration with the chess engine needs to be realized. Although communication in the general structure from Figure 4.4 is typically

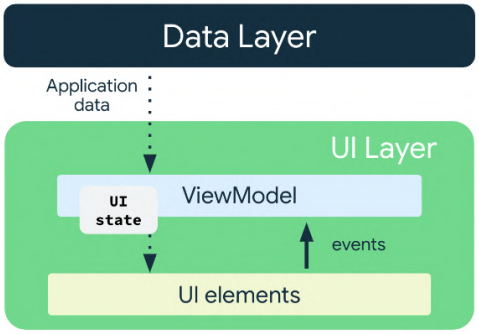


Figure 4.4: General Android Application Architecture
Source: `https://developer.android.com/codelabs/basic-android-kotlin-compose-viewmodel-and-state#3`

unidirectional, this project requires bidirectional communication. By following user inputs, whenever required, the UI should be able to send a chess position to the chess engine and receive information about the move that the chess engine suggests.

The UI Layer consists of two components. The ViewModel which in our case is the brain of the chess game. It is responsible for keeping track of the moves and also allow the user to make only legal moves. It keeps track of all the UI elements and reacts according to the users commands. The UI elements layer will contain the Jetpack Compose elements that represent the chess board, buttons, menus and options that the user will see and interact with.

A detailed documentation of the specific implementation of the application can be found in section 5.3.

4.2.2 Bluetooth Communication (Arduino-Android)

Chess moves have to be sent and received constantly between the two devices: the application and the chessboard represented by the Arduino. The implementation for the Arduino communication is straightforward. It requires correct wiring of the HC-05 module as described in section 5.1.3 and the use of Serial libraries provided by the Arduino platform.

However, connection via Bluetooth on the Android application is much more difficult. Firstly, Android mandates explicit user permission for apps to utilize Bluetooth functionalities, and recent Android versions also require location permissions for Bluetooth operation. This additional requirement aims to enhance user privacy and security, but induces significant complexity for the application development process.

Secondly, for connecting and communicating with a Bluetooth device, there are two main methods proposed by the Android Developers community. The first one is to use the BluetoothAdapter class provided by the Android SDK, that should provide enough tools

to enable Bluetooth, pair with devices and establish connections. Unfortunately, during testing conducted for this specific application and connection with HC-05, it proved to be unreliable and an alternative approach was preferred.[16]

The alternative solution is to use the CompanionDeviceManager system that simplifies the process of pairing and connecting to Bluetooth devices. Even though this method does not provide the programmer with the same level of control over the process, during testing on this specific project has proved to be more reliable and easier to implement. Another element that is used with this method is the BluetoothManager class that provides tools for checking permissions, searching for specific devices (in this case only "HC-05" devices will be listed in the Bluetooth connection menu) and managing connections and sockets. [17]

Chapter 5

Detailed Design and Implementation

5.1 Hardware Implementation

5.1.1 Pieces Detection

Pieces detection mechanism is based on the idea that all chess games start from the same initial position. Therefore, as previously mentioned, a Hall Sensor must be added under each square in order to detect when a piece is present in that position. For this task, the sensors A3144 (Figure 5.1) have been tested and used successfully. They have been found to be suitable for this application because they are digital mono-polar sensors whose magnetic field required to trigger a response has a value that matches the size of the magnets that have been attached to the bottom of the pieces.

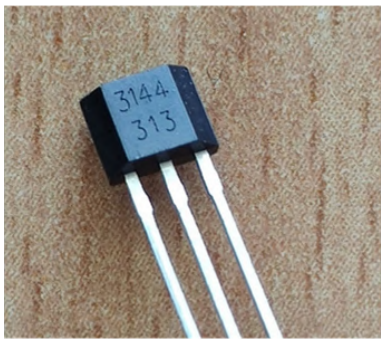


Figure 5.1: A3144 Hall Sensor

The number of the sensors matches the number of squares of the board. The board has 64 squares that represent the 8x8 grid needed to play the game, plus two groups of 16 squares each which represent the places where the captured pieces are taken. Therefore, the project has a total of 96 sensors.

The A3144 has an open drain output pin, which means that it connects the output pin to 0 when a magnetic field is detected, and it leaves the pin disconnected in all other

cases. Therefore, in order to read the digital value logic 1 from the Arduino, a pull up resistor between the output of the sensor and +5V must be connected.

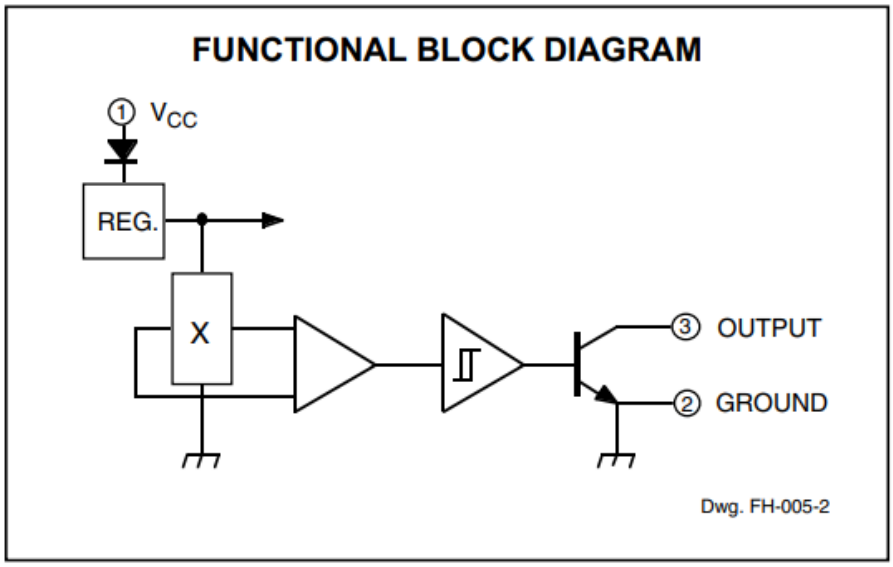


Figure 5.2: A3144 functional diagram; Source: A3144EU Datasheet

From the Datasheet [18], the functional block diagram from figure 5.2 can be found. It gives a high level overview of the functionality of the sensor and provides information about the purpose of each pin, information that allows us to connect the sensors correctly.

Because the project has such a high number of sensors that need to be managed by the Arduino board, it is necessary to add multiplexers that drastically reduce the number of digital inputs that are needed to read the state of the board.

For this purpose, the choice was to get 6 multiplexers with 16 channels which reduce the number of digital pins needed from 92 to 30 (4 for command signal and one for reading the selected signal for each of the 6 multiplexers). Normally, the external pull up resistance mentioned earlier can be replaced by the internal pull up resistance offered by the Arduino board, but because the sensors are connected to the multiplexers first, it cannot be done for this specific application. Therefore, 96 resistors have been added to the build, each having a value of 10k, which proved to be a good compromise between current consumption, responsiveness, and resistance to disturbances. The resistors are visible next to the sensors in the final build presented in figure 5.3. What cannot be seen in the figure is the fact that each wire from the bottom part leads to one input from a multiplexer.

Figure 5.3: All Hall Sensors mounted on the board

5.1.2 Pieces Movement

CoreXY

The CoreXY system, which was presented in Chapter 4, requires a great number of hardware components to work together. For building the system, V-Slot linear rails have been used for guiding the electromagnet on the two axes. The system needs 3 linear rails and 3 groups of wheels that allow movement on the two degrees of freedom as required. Two of the rails were placed parallel, on the sides, and one will be placed transversely in between them. The montage can be seen in the figure 5.4. In order to make it possible for the system to work, a minimum of 2 motors are needed. They work together to move the trolley in any point in the 2D working envelope.

In order to build the CoreXY system, the dimensions for the working envelope were calculated taking into consideration the dimensions of the pieces. Some automated movements (for example when a Knight "jumps" over two other pieces) require paths that are on the edge between two occupied squares. Also, the pieces of the chess set were not all the same size, so the worst-case scenario of the Queens should be taken into consideration,

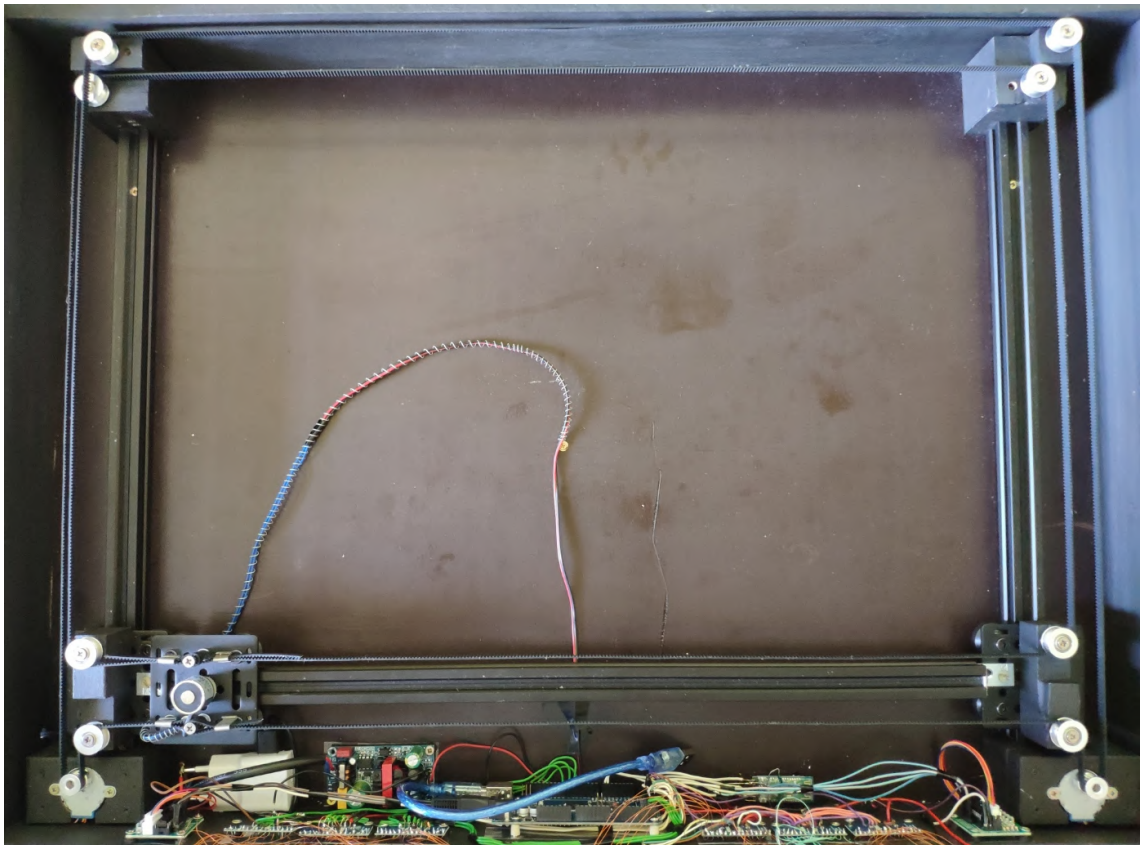


Figure 5.4: CoreXY Build

since they are the largest pieces. After some simple calculations considering the 12x8 grid and keeping a small margin for error, the minimal work envelope was determined to be 495x320 mm. For this, the linear rails included in the final build had to be two of 350mm and one of 500mm.

The linear rails are just the base of the main structure of the system. A set of supporting elements has been designed. These elements have to hold the gears and guide the belts as needed. Besides the mandatory functional requirements, the design had to make sure the dimensions on the horizontal axis are kept as small as possible because of limiting the final dimensions of the chessboard as much as possible. It was very important for them to offer precise positions on X and Y axes for the gears and also describe two planes with almost no tolerance on the vertical axis. This is required because the two components of the CoreXY (each actuated by one motor) are moving independently and there are 4 places where the belts would intersect, had they been in the same plane. That is why the structural elements were designed to provide exactly 8mm between the positions on the Z axis of the belts connected to each motor.

Due to easier design process, a 3D modeling software was used to conceive the pieces shapes and dimensions. Therefore, all the pieces have a design that is ready for 3D

printing figure but due to unavailability of such a device, the elements had to be built by hand. The result of the work can be seen in the figure 5.5. They were manufactured out of wood, by using basic tools such as a circular saw and a hand operated milling machine. For aesthetic purposes and for the longevity of the final product, the pieces were painted mate black once they were finished.

Figure 5.5: Structural elements made out of wood

For the actual connection between the motors and the rest of the structure, the choice was to go for 6mm GT2 belts which were connected to the motors shaft by using GT2 also 6mm width gears that can be seen in the second image from figure 5.6.

For the rest of the 8 pivot points of the system, simple wheels had to be used in order to allow movement of the trolley even when one of the motors is stationary (for diagonal movements). These wheels were separated from the structural elements to allow easy rotation by using dented washers. Also in the first image from figure 5.6 you can see the way the horizontal rail is connected to the vertical ones by using a set of 4 V-Slot wheels and an aluminum board that holds the structural element necessary for the belts and gears.

Electromagnet

As has already been mentioned, the pieces have a magnet attached to their bottom side. An electromagnet is positioned under the playing surface in a two-dimensional space. This electromagnet should produce a magnetic field strong enough to pull the piece to the desired position. The KK-P20/15 electromagnet (Figure 5.7) which needs to be supplied with 12V DC and has a pulling force of 3kg has proved to be sufficient for moving the pieces through the Plexiglas board.

Because of the 12 V supply voltage, the electromagnet cannot be controlled directly from the digital pins of the Arduino, so a power transistor is needed for this task. A suitable choice is the 2N3904 NPN transistor, which has been connected to the Arduino and the electromagnet as can be seen in the diagram from Figure 5.8.

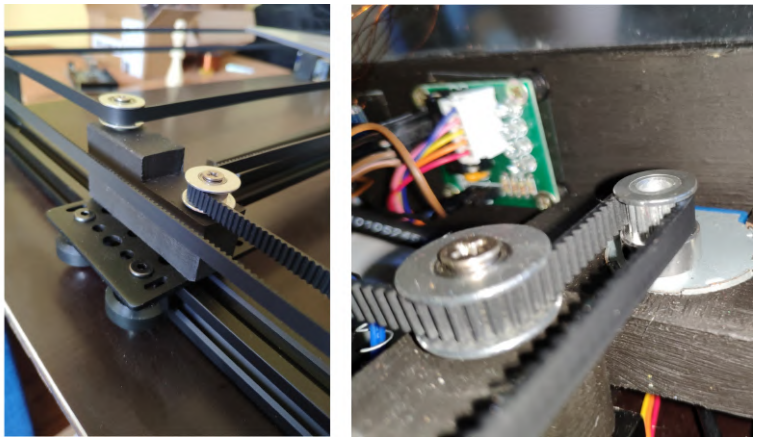


Figure 5.6: Gears and Belts

Figure 5.7: KK-P20/15 - Electromagnet

While deciding how to wire the transistor it is important to take into consideration the operating rules for this type of electronic component. Being a NPN transistor, the pin number 1, which is the emitter has been wired directly to ground, the pin number 2, which is the base has been wired through a resistance of 1kOhm (in order to limit the current that is supplied by the output pin of the Arduino) to a pin from the Arduino. Finally, the collector pin number 3 has been connected to one of the wires of the electromagnet.

The other wire of the electromagnet has been connected directly to +12V. The polarity of the electromagnet depends directly on the wire that is supplied with 12V, so I have chosen the wires experimentally to be opposite of the polarity of the magnets that are placed on the bottom side of the pieces.

Ensuring stable movement for the pieces was probably one of the hardest parts of this project because of the very small tolerances that it has. The electromagnet is barely strong enough to move the pieces through the board. For this I have tested multiple materials and Plexiglas was chosen. Pieces could be moved through the board only if the electromagnet is 4mm away from the chessboard or less.

This means that the whole CoreXY structure has to be built with such a small tolerance that the electromagnet is not too far from the board. Also, space for the whole

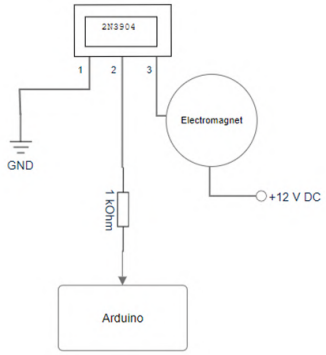


Figure 5.8: Transistor and Electromagnet Connection

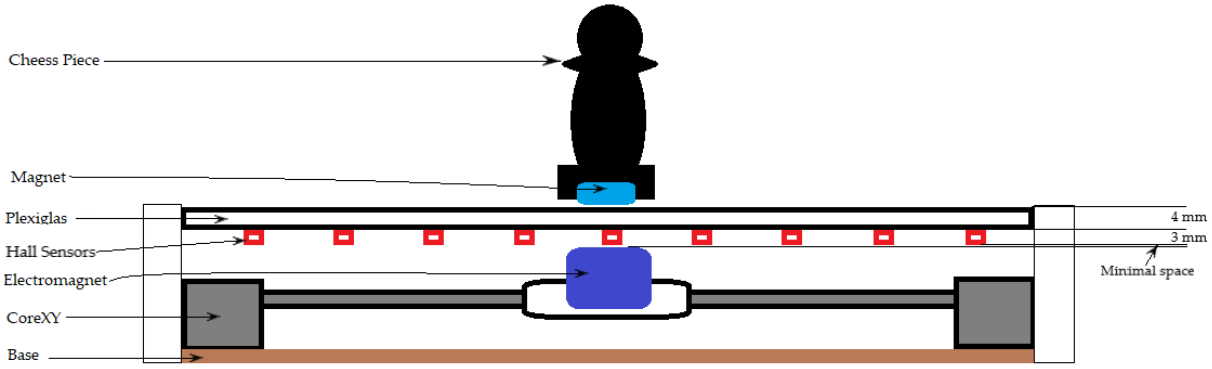


Figure 5.9: Design of the layers - Side View

montage of the pieces detection which includes all the sensors, resistances and wires is needed. The thickness of the sensor is 2mm and the diameter of the resistance is almost 3mm, which gives us only 1mm of tolerance in the whole built. Considering the fact that everything was built by hand, this tolerance has proven to be problematic and very difficult to respect. However, the results are satisfactory, and pieces can be moved, but in some isolated cases problems still arise and pieces movement is still imperfect.

Stepper Motors

The previous subsections have already mentioned all the necessary parts that are needed for the project. This section will present how the connection of the motors and the Bluetooth module was made.

A stepper motor is a type of brushless DC motor that divides a full rotation in a number of equal steps. Stepper motors are controlled by electrical pulses, with each pulse corresponding to a step in movement. The motor's position is known at all times,

eliminating the need for feedback systems found in other types of motors. This type of motor is suitable for the project because it satisfies the need for precise positioning.

The stepper motor that was chosen is the 28BYJ-48 (Figure 5.10). This stepper is a unipolar stepper motor with a 5.625° step angle, translating to 64 steps per revolution in its full-step mode. However, due to its internal gear reduction (which is 1:64), the effective number of steps per output shaft revolution increases to 4096.

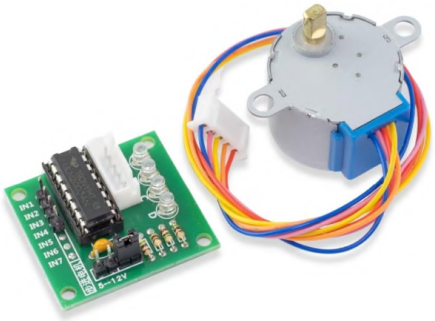


Figure 5.10: 28BYJ-48 Motor and ULN2003AN Driver

The motor has four coils, which are activated in a predefined order to achieve smooth rotation. By altering the sequence and duration of the pulses sent to the coils, the motor can be made to rotate clockwise or counterclockwise, at various speeds.

In order to operate the motors, the use of a driver is essential for the following reasons. Firstly, it provides the necessary current amplification considering that the Arduino Mega cannot provide directly sufficient current to drive the motor directly. Secondly, the driver protects the microcontroller from high voltages and inductive spikes generated by the motor's coils. For this project, the driver that was chosen was the ULN2003AN (Figure 5.10) which is compatible with the 28BYJ-48 motors and is often sold in the same package.

5.1.3 Bluetooth Module

As was described in subsection 4.1.4, the hardware build requires a Bluetooth module that allows for wireless serial communication. The module that was chosen is the HC-05 which can be seen in the right hand side of figure 5.11.

The connection diagram from figure 5.11 depicts a voltage divider between the RX (Recieve) pin of the HC-05. This is needed because the pin operates at 3.3V which cannot be exceeded. This pin needs to be connected to the Arduino Mega TX pin which operates at 5V. By using two resistances one of 1k and the other of 2k, we can scale the 5V signal to a 3.3V as follows:

$$V_{hc-05_{RX}} = \frac{2\,\text{k}\Omega}{1\,\text{k}\Omega + 2\,\text{k}\Omega} \times V_{arduino_{TX}}$$

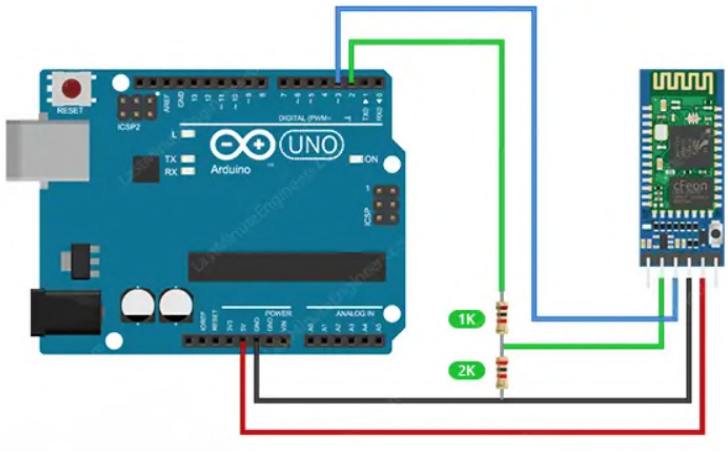


Figure 5.11: HC-05 Bluetooth Module
Source: `https://lastminuteengineers.com/hc05-bluetooth-arduino-tutorial/`

$$V_{hc-05_{RX}} = \frac{2\,\text{k}\Omega}{1\,\text{k}\Omega + 2\,\text{k}\Omega} \times 5\,\text{V}$$

$$V_{hc-05_{RX}} \approx 3.33\,\text{V}$$

5.1.4 Power Supply

Once all sensors and actuators are identified, solutions for addressing the power supply issue can be explored. Considering the components mentioned earlier, two separate voltages are required: 5V for motors, Hall Sensors, and multiplexers, and 12V for the Arduino and the electromagnet.

To meet these requirements, one option is to use a single 12V DC power supply and derive 5V using a DC to DC Buck Converter. Alternatively, for economic reasons, I have opted for two separate power supplies—one for 12V and another for 5V. It is crucial to ensure that the total current drawn by all electronics remains within the limits of the power supplies.

For the components connected to the 5V supply, the requirements are as follows:

1. **Hall Sensors A3144**:

$$\begin{aligned} \text{Current per sensor} &= 4\,\text{mA} \\ \text{Number of sensors} &= 96 \\ \text{Total current for sensors} &= 96 \times 4\,\text{mA} = 384\,\text{mA} \end{aligned}$$

2. **Pull-Up Resistors**:

$$\begin{aligned}
\text{Resistance per pull-up} &= 10\,\text{k}\Omega \\
\text{Voltage} &= 5\,\text{V} \\
\text{Current through each pull-up resistor} &= \frac{V}{R} = \frac{5\,\text{V}}{10\,\text{k}\Omega} = 0.5\,\text{mA} \\
\text{Number of resistors} &= 96 \\
\text{Total current for pull-up resistors} &= 96 \times 0.5\,\text{mA} = 48\,\text{mA}
\end{aligned}$$

3. **HC-05 Bluetooth Module**:

$$\text{Current during transmission} = 50\,\text{mA}$$

4. **28BYJ-48 Stepper Motors**:

$$\begin{aligned}
\text{Current per phase} &= 240\,\text{mA} \\
\text{Number of phases energized} &= 2 \\
\text{Current for one stepper motor} &= 2 \times 240\,\text{mA} = 480\,\text{mA} \\
\text{Number of stepper motors} &= 2 \\
\text{Total current for stepper motors} &= 2 \times 480\,\text{mA} = 960\,\text{mA}
\end{aligned}$$

5. **Multiplexers**:

$$\begin{aligned}
\text{Current per multiplexer} &= 1,\text{mA} \\
\text{Number of multiplexers} &= 6 \\
\text{Total current for multiplexers} &= 6 \times 1\text{mA} = 6\text{mA}
\end{aligned}$$

6. **Total Current Calculation**

$$\begin{aligned}
I_{\text{total}} &= I_{\text{sensors}} + I_{\text{total pull-up}} + I_{\text{HC-05}} + I_{\text{total steppers}} \\
I_{\text{total}} &= 384\,\text{mA} + 48\,\text{mA} + 50\,\text{mA} + 960\,\text{mA} \\
I_{\text{total}} &= 1442\,\text{mA} \\
I_{\text{total}} &= 1.442\,\text{A}
\end{aligned}$$

Consequently, in the worst case scenario, the 5V power supply will be required to provide 1.44 A of current, which for this project will be provided by an old Samsung charger rated at 5V and 2A, which can be seen in the right hand side of Figure 5.12.

For the electronics that are connected to the 12V power supply, the current requirement is computed as follows:

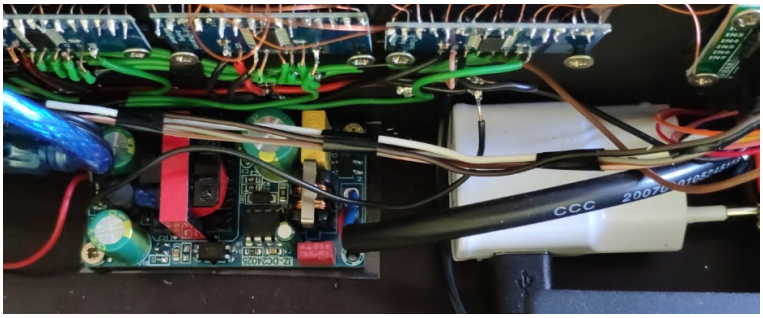


Figure 5.12: Power Supplies

1. **Arduino Mega**:

$$\text{Operating voltage} = 12\,\text{V}$$
$$\text{Typical current draw} = 250\,\text{mA}$$

2. **Electromagnet KK-P20/15**:

$$\text{Operating voltage} = 12\,\text{V}$$
$$\text{Current draw} = 0.36\,\text{A} = 360\,\text{mA}$$

3. **Total Current Calculation**

$$I_{\text{total}} = I_{\text{Arduino Mega}} + I_{\text{Electromagnet}}$$
$$I_{\text{total}} = 250\,\text{mA} + 360\,\text{mA}$$
$$I_{\text{total}} = 610\,\text{mA}$$
$$I_{\text{total}} = 0.61\,\text{A}$$

Considering the fact that the Arduino is connected to multiple other elements, as well as market availability, a power supply of 2A was chosen. The 12V power supply OKN428-32 can be seen in the same figure 5.12 on the left hand side.

5.2 Arduino Software Implementation

5.2.1 Architecture Overview

The code uploaded on the Arduino is responsible for all of the functionalities of the chessboard. It has to communicate via Bluetooth with the Android application and also, based on the moment of the game, it should fulfill one of the following roles: read the state of the board in order to detect the moves of the human player or receive instructions from the Android Application and execute the move.

Being an Arduino project, the main language used to develop this part of the software was C++ and it was built by using the Arduino IDE. The highest level overview of the architecture can be seen in the diagram from figure 5.13.

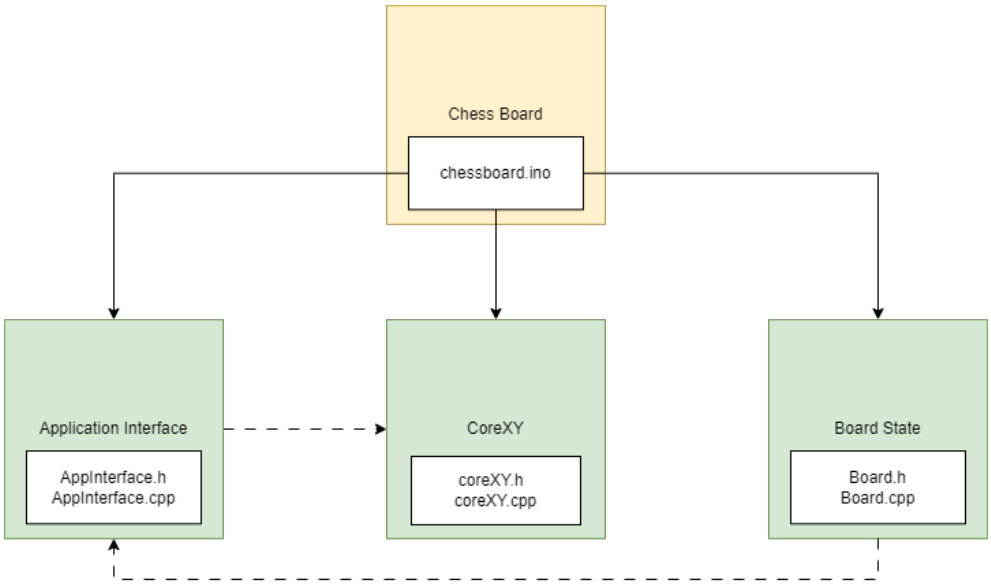


Figure 5.13: High Level Software Architecture

The four components are the main file "chessboard.ino", the "Application Interface" which is responsible with all the communications between the Arduino and the Android application, the "CoreXY" which is responsible for calculating and controlling all movements of the CoreXY structure and finally the "Board State" that should facilitate tracking of all the moves that happen on the chessboard.

Figure 5.13 also illustrates the relationship between these components. The solid lines suggest that the main file "chessboard.ino" is responsible for instantiating all the other objects. There are two dependencies represented with dotted lines. One suggests that the "Application Interface" is responsible for giving instructions to the "CoreXY" component, while the other suggests that the "Board State" is responsible for providing information about moves and pieces positions to the "Application Interface".

A detailed documentation for each of the four components is presented individually in the following subsections.

5.2.2 Main File "chessboard.ino"

Being the main file of the sketch, chessboard.ino follows the general structure of an Arduino sketch. As can be seen in figure 5.14, the file only has two global functions: setup() and loop().

The **setup()** function is only run once, when the Arduino code is first executed. It is responsible for initiating the serial communication using the Serial library provided by

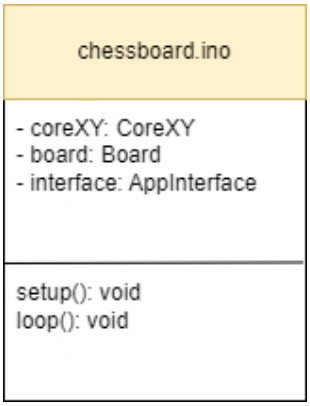


Figure 5.14: Structure of chessboard.ino

the Arduino package, as well as initializing the other three important components of the application.

The **loop()** function, as its name suggests, is executed in a loop for as long as the Arduino is running. The loop is fairly simple. It is a conditional construction that determines whether its the human player's turn or the chessboard's turn. If it is the human player's turn, then a function from the "Board State" component is called and the program waits until a move is detected and acts accordingly. If it is the chessboard's turn, then a function from the "App Interface" component is called which is responsible for processing the data received via Bluetooth and command the CoreXY system to execute the move.

5.2.3 CoreXY

The inverse and forward kinematics are fundamental concepts in controlling the movement of robotic systems. Forward kinematics refers to a set of equations that allows calculation of the position and orientation of an end effector given the joint parameters. Inverse kinematics, on the other hand, involves a set of equations that provides the necessary relations to compute the joint parameters needed to achieve a desired position and orientation of the end effector.

The purpose of this part of the implementation is to enable the "CoreXY" component to accept a set of coordinates of the XY system defined as the one in figure 5.15 and control the motor positions so that the electromagnet reaches the specified position. Therefore, the first step in developing a control system for the CoreXY is to obtain the Inverse Kinematics model.

Firstly, an experimental measurement was made in order to determine the equations of the forward kinematics. By logical analysis, considering that a composed action of both motors is required to move either vertically or horizontally, the general form of the forward kinematics can be seen in the following equations:

Figure 5.15: Cartesian space definition on the board

$$x = a \cdot q_1 + b \cdot q_2 \tag{5.1}$$
$$y = c \cdot q_1 + d \cdot q_2 \tag{5.2}$$

The purpose of the experiment was to determine the values for the variables 'a', 'b', 'c' and 'd'. For this, two measurements were recorded. Firstly, a complete rotation of the first motor (q_1) was made and the displacement caused by this movement on the X and Y axes was recorded in the variables 'a' and 'c'. Similarly, the displacement caused by the second motor (q_2) was recorded in the 'b' and 'd' variables. The measurements have been recorded in millimeters and these numbers shaped the forward kinematics in its final form, which can be seen in the following equations:

$$x = 16 \cdot q_1 - 16 \cdot q_2 \tag{5.3}$$
$$y = 16 \cdot q_1 + 16 \cdot q_2 \tag{5.4}$$

Once the forward kinematics is known, we can simply consider the x and y known and solve the system of equations for q_1 and q_2. The result of the computations which is the final form of the inverse kinematics is presented below:

$$q_1 = \frac{1}{32} \cdot (x + y) \tag{5.5}$$

$$q_2 = \frac{1}{32} \cdot (x - y) \tag{5.6}$$

The structure of the CoreXY component can be seen in figure 5.16. It is a collection of methods whose purpose is to satisfy requests of movement for the CoreXY system. It can be observed that it only has 3 parameters: leftMotor, rightMotor and isMoving. All are private so this is the only component that has control over the motors.

The motors are controlled using an external Arduino library named AccelStepper that has been preferred over the standard Arduino Stepper Library for multiple reasons. One of the key advantages of the AccelStepper library is its non-blocking control feature, allowing the main program to execute other tasks while managing the motors in the background. The library also simplifies the simultaneous control of multiple motors, which is essential for this specific CoreXY configuration. It provides an easy-to-use interface for synchronizing the movements of both motors, ensuring accurate positioning of the end effector. [19]

The main method offered by the CoreXY class is the **moveTo(int x, int y)** one which takes x and y coordinates in millimeters and uses the other methods of the class to satisfy the task.

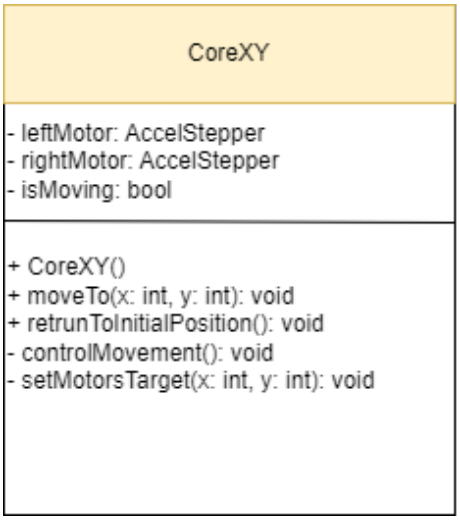


Figure 5.16: Structure of CoreXY class

The **setMotorsTarget(int x, int y)** is responsible for taking the coordinates in millimeters and setting the target positions for both motors by applying the inverse kinematics from equations 5.5 and 5.6. The **controlMovement()** function is responsible that the motors get to the target specified by the setMotorsTarget() by using the leftMotor and

rightMotor objects. Both of these functions are used by the moveTo(x, y) to fulfill its tasks.

The other methods from figure 5.16 are used by the main file (chessboard.ino) to instantiate the CoreXY object, to setup the performances and to make sure the CoreXY system will always start a game and eventually return to the same place, called "Initial Position" and represented by the origin of the XY coordinate system in figure 5.15. The code developed for controlling the CoreXY system can be seen in appendix A.2.

5.2.4 Board State

The "Board State" component is represented by the Board class. The purpose of this class is to use the 6 multiplexers and the sensors to keep track of all the moves. The attributes and methods that belong to this class can be seen in figure 5.17 and will be described below.

The two attributes represent the board in matrix form. The **state** attribute is of type char and holds the type of piece that is present on each square. Each letter used is specific to the piece it represents, for example 'P' for pawns and 'B' for bishop. White pieces are represented by lowercase, while black pieces are represented by uppercase letters.

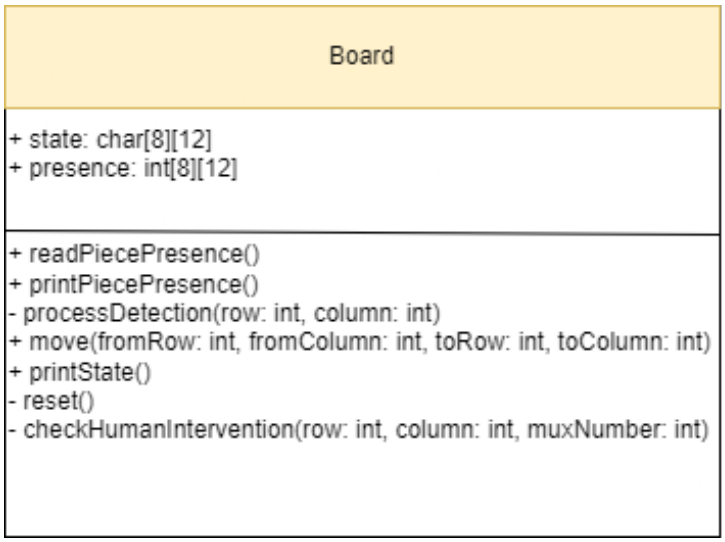


Figure 5.17: Structure of Board class

The **presence** attribute is similar to the state, but it only holds values of 0 and 1. For every square, the respective location in the presence matrix is 1 when the sensor detects magnetic presence and 0 when the sensor does not detect anything. Both of these matrices are initialized in the **Board()** constructor and can be reset when a new game is desired by using the **reset()** function.

The main loop() of the Arduino is responsible for calling the **readPiecePresence()** method whenever a move is expected from the human player. This function is responsible

for commanding the multiplexers such that all the sensors are read. Each multiplexer is responsible for two columns of the whole matrix and we have 6 multiplexers. Therefore, 6 sensors are read at a time and then the value of the multiplexer's command is changed.

The move tracking process continues by calling of the **checkHumanIntervention()** which compares the value in the presence matrix with the value that is given by each sensor. If the values are different, we know that a human has interfered with the state of the board so the **processDetection(row, column)** method is called. These three important methods can be seen in the appendix A.1.

There are two types of interventions that can be detected. The first one is when a player lifts a piece and a change in presence from 1 to 0 is detected, and the second one is when a player places down a piece when the change in presence is from 0 to 1. The processDetection() checks which type of detection was made and acts accordingly. It also is responsible for checking what color the piece moved was in case the player is in a capturing action and registers the move in the state[][] matrix only once the human action is complete. When a move is registered, the function sends the data to the Bluetooth device via Serial communication in the form of a string of coordinates.

5.2.5 Application Interface

This component is responsible for receiving and processing moves that are sent by the Android application. Its structure can be seen in figure 5.18.

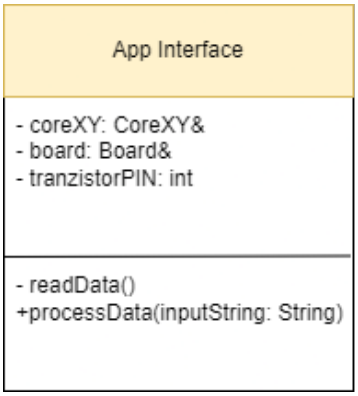


Figure 5.18: Structure of Application Interface

The **readData()** method is called by the loop in the chessboard.ino file and it is responsible for reading each character from a message until receiving the '\n' character, which symbolizes the end of a message.

The message is received in the form of a string of 4 numeric characters that represent the row and column of the initial position and final position of the moved piece. The **processData(String inputString)** function is converting this data into a matrix of 4 integers and ensures the move is executed.

For a simple move, the process is as follows: the electromagnet is moved to the pick up position, the tranzistorPIN which is responsible for switching on the power for the electromagnet is switched to value 'HIGH' and the electromagnet is moved to the drop down coordinates in a simple linear movement, where the transitorPIN is switched back to value "LOW".

In special cases the move is more complicated and requires multiple steps. A complex example is presented in figure 5.19.

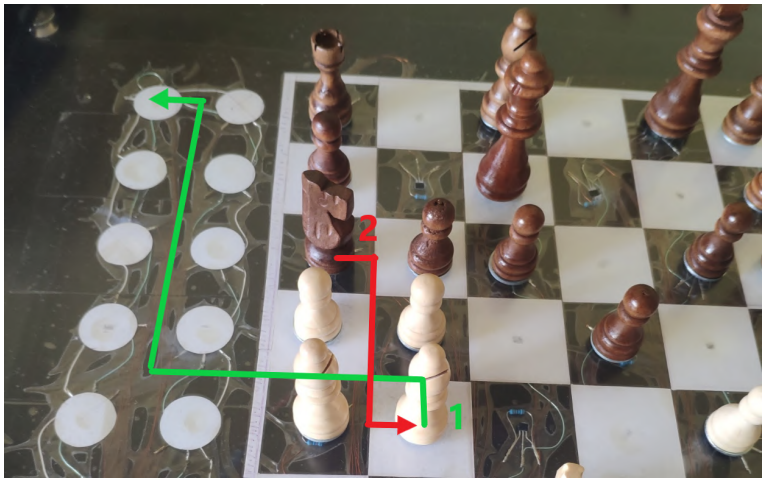


Figure 5.19: Special Move Execution

For this example, the processData() method only receives the coordinates that describe the initial and the final position of the black knight. It is then responsible for checking the destination square by using the board state and finding that a white piece is present there. Here the first "special move" condition is checked because a piece capture is taking place. The method then looks for an empty place in the "graveyard" which can be seen in the image represented as the white round places. In this case, the empty place is the first one in the upper left corner so the white bishop is moved there, while making sure that it only travels on the edges to avoid any contact with the other pieces on the board. The moves are all made by using the coreXY.moveTo(x, y) methods. This move is represented by the green line from figure 5.19.

Once the captured piece is out of the way, the system can execute the move for the black knight. Here we have the second special case because the piece that moves is a knight. As it is well known, the knight is the only chess piece that can "jump" over occupied squares. Because a piece cannot be lifted by our system, it computes a path that only travels on the edges, as it can be seen in the red line from figure 5.19.

5.3 Android Software Implementation

5.3.1 Application Architecture

The application mostly follows a View - View Model - Model architectural pattern as the one recommended by the Android development team and presented in subsection 4.2.1. The application of those principles for the android part of the project has resulted in the architecture from figure 5.20 which is presented in detail in this section.

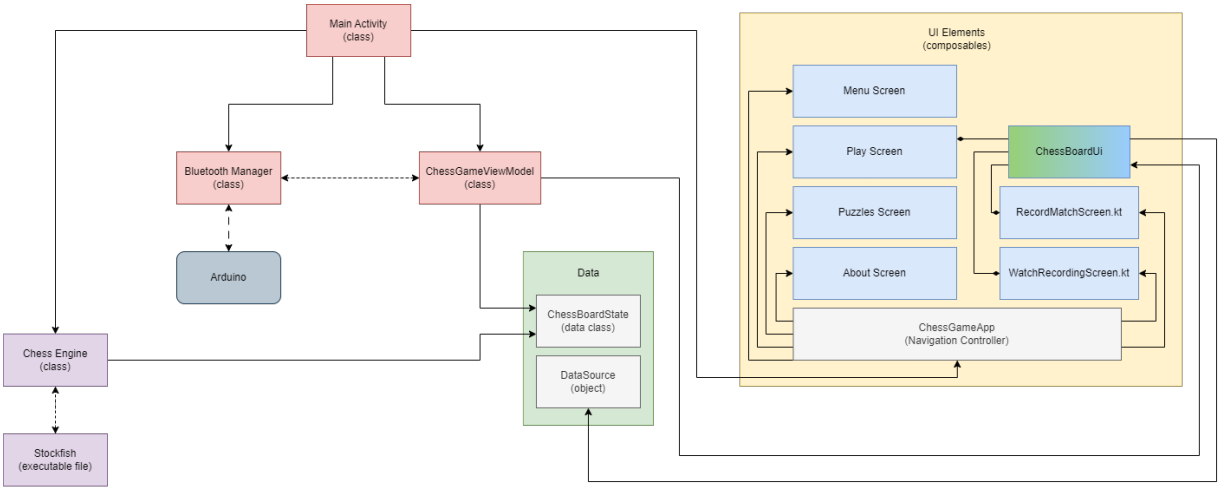


Figure 5.20: Android Application Architecture

The starting point for understanding the schematic is the Main Activity component. It is the activity component that is started when the application is launched and is responsible for starting all the main processes, by instantiating the Chess Engine, Bluetooth Manager, and Chess Game View Model components, each having its fundamental role in achieving the desired functionality. It is also setting the content to the Chess Game App component, which is the starting point for the UI and is responsible for managing navigation around the different screens of the application. Each of these components will be described in more detail in the following subsections.

5.3.2 User Interface and Application Navigation

The first component of the UI is the **ChessGameApp.kt** file which holds a class that takes as a parameter a NavHostController that allows the user to select between all the screens of the application. It is just a collection of composables that constructs an object of each screen type based on predefined ”routes” that the user can select from through different actions inside the application. The screens that can be seen directly connected to the ChessGameApp are all composables that include elements such as Text, Box, Button, and others.

The next essential component of the user interface is **ChessBoardUi.kt**. This element, like all others displayed by the application, is a composable. It is designed to be integrated into various screens of the application, making it adaptable for both current and future game modes. As a key component of the Play Screen, RecordMatchScreen and WatchRecordingScreen, the ChessBoardUi.kt provides a graphical representation of the chessboard and its pieces.

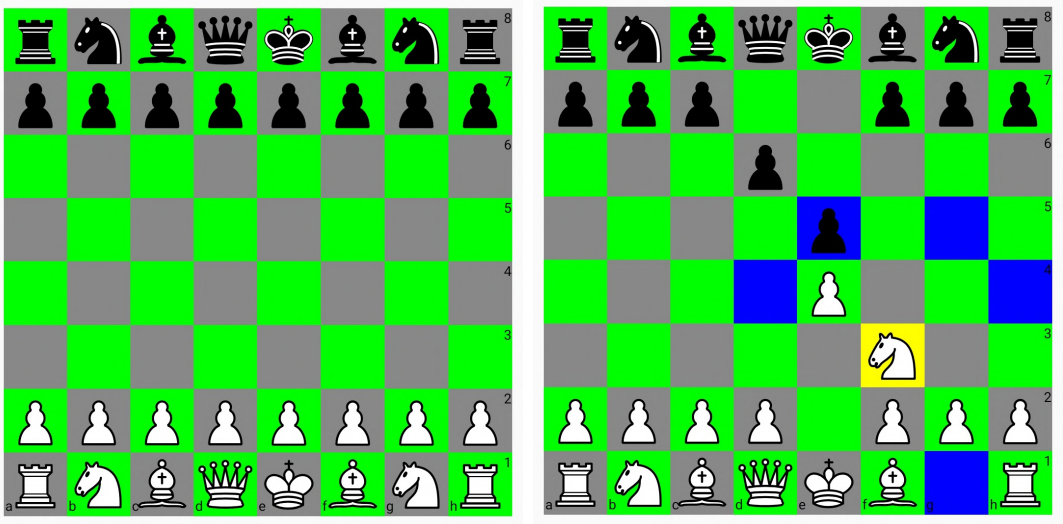


Figure 5.21: Print-screen of ChessBoardUi composable

The ChessBoardUi.kt is actually made out of two composable functions, one for the board itself and one for the ChessSquareUi. Each square build out of a colored Box() and optionally attributes that set an image for the piece present there or letters and numbers for the squares on bottom or right edges. The ChessSquareUi is capable to decide how each square should look and color the last clicked square in yellow and all possible moves for the selected piece in blue.

The chessboard is then build by using 64 ChessSquareUi elements that always have to reflect the state of the board as it is stored into the ChessBoardState objects and is transmitted from the Play Screen through the constructor. Also, every time a square is clicked by the user, the information is transmitted to the ChessGameViewModel through the chessGameViewModel.onClickSquare(Pair(i, j)) function. The results can be seen in figure 5.21, where the first image depicts the initial state and the second one illustrates how the board changes during user interaction.

5.3.3 Data Layer

As can be observed in the architecture from figure 5.3.1, the data layer of the application is represented by two files: ChessBoardState and DataSource.

The **DataSource** is an object that maps the resources used by the application to a certain label. For instance, the image resources of chess pieces mapped to the corresponding code utilized by both the Board State and ChessBoardUi. This approach is regarded as a best practice in Android Development, as it enhances code organization and maintainability.

The **ChessBoardState** is a data class extensively utilized, either directly or indirectly, by most components of the application. It encapsulates all the essential information related to the game of chess, including:

- A matrix representing the positions of the pieces
- The clicked square coordinates
- A list of possible moves
- The current turn (white or black)
- The move counter
- Option selection elements, such as whether the user is practicing or playing against Stockfish and the difficulty level

5.3.4 Bluetooth Connection

The Bluetooth functionality of the application is ensured through the **BluetoothManager.kt** class. The main methods that are provided by this class are:

- checkPermissions()
- handleActivityResult(requestCode: Int, resultCode: Int, data: Intent?)
- searchForDevices()
- connectDevice(device: BluetoothDevice)
- changeConnectionStatus(connection: Boolean)
- disconnectDevice()
- sendDataToDevice(dataString: String)
- receiveDataFromDevice(): String

The first step for implementing Bluetooth communication inside an android application should be to ensure that the application has been granted with the necessary permissions. These permissions have to be included into the AndroidManifest.xml file and they are the following: BLUETOOTH, BLUETOOTH_ADMIN, BLUETOOTH_CONNECT

and BLUETOOTH_SCAN. If the application is intended for use on Android versions higher than Android 12, then ACCESS_FINE_LOCATION permission must also be obtained. Without these permissions, the use of Bluetooth is not allowed by the Android System.[20]

Consequently, the Bluetooth Manager provides the method **checkPermissions()** which checks if the required permissions have been granted by the user and also requests them in the case in which they have not.

The **searchForDevices()** method is responsible for allowing the user to select the Bluetooth device he desires to connect to. This uses a CompanionDeviceManager object to access the list of all the paired devices which are in functioning range. It also uses a BluetoothDeviceFilter object that filters the available devices such that only the ones named "HC-05" will be available for selection.

The **handleActivityResult()** method is called from the MainActivity class when a user selects the device and is eventually providing the **connectDevice(device: BluetoothDevice)** method the required information to complete the task. These functions use methods provided by the Companion Device Manager framework.

Once all these steps have been successfully completed, the connectionStatus is changed and the ChessGameViewModel can communicate with the Bluetooth device via the methods: **sendDataToDevice(dataString: String)** and receiveDataFromDevice(): String. These methods are using BluetoothSocket objects to allow sending and receiving strings of information with the Chessboard. Because it is not recommended to freeze the main thread of the application during processes related to Bluetooth, the functions are launching a separate threads that are attempting to connect to a device or transmit the String of information.

5.3.5 Stockfish Integration

Being open source and one of the most popular chess engines, Stockfish is chosen as the chess engine responsible for providing this project with high quality moves for a given position.

Stockfish provides an executable that uses UCI for communication. UCI stands for Universal Chess Interface and it is a protocol that standardizes the way engines and UI's interact. Examples of commands used are: *ucinewgame, position, go, stop* and many others.

Besides UCI, a protocol is also needed to communicate effectively a position for the game. For this purpose Forsyth-Edwards Notation (FEN) is used. This protocol is exemplified directly on a situation captured in the practice mode of our application (Figure 5.22).

FEN describes a chess position by using a string that contains six fields separated by spaces. The first field represents the position of the pieces on the board as follows. Rows are separated by '/' character and they contain all the pieces in the respective row represented by a letter. Empty places are represented by the number of consecutive empty

Figure 5.22: Print-screen of Practice Mode

squares written as a digit (1 to 8). The second field represents the color of the pieces for the player that has to move (in figure 5.22 is white's turn) and the third field represents castling availability. The last three fields are the en passant target square (if any), the "half-move clock" (for the fifty-move rule) and finally the "full-move number" (starts at 1 and increments after each black move).

For simplicity while working with the UI, the application holds the chess position in a matrix form. Therefore, the first method from the ChessEngine class is **getFenFromChessBoardState(boardState: ChessBoardState): String**. This function follows the rules presented in the previous paragraph and transforms the board state to a String that can be used for communication with Stockfish. The FEN that has been computed for a chess position can be seen under the "Reset Board" button in figure 5.22.

In order to communicate with the Stockfish executable, it has to be launched in the background of the application. This can only be done if the executable is placed in the right

location of the internal memory of the device. This location is allocated by the Android System and it allows the application to store and execute files like the Stockfish compiled binary. The executable file is copied from the resources folder to the right location from the internal memory directly from the MainActivity.kt when the application is opened. Then, this file is executed from the **init** method of the ChessEngine class (which is executed automatically right after the primary constructor in Kotlin).

The last method from the ChessEngine class is the **getBestMove(fen: String, difficulty: Int): String?**. This method uses input and output streams to communicate with the executable file by using the UCI protocol, the difficulty and the position which is received as a parameter from the calls inside the ChessGameViewModel. The function gives Stockfish a command and checks the output until it finds the line which starts with "bestmove". It extracts the move from that line and returns it as a String with 4 characters. The methods presented in this subsection can be seen in the full implementation of the ChessEngine class in the appendix A.3.

5.3.6 Chess Game View Model

The Chess Game View Model is the largest and most important class for this application. It is responsible for managing all the chess logic implemented in the application, all user interactions with the game screens and it guides the communication between human player, chess engine and the physical chessboard.

The class has a large collection of methods, so that only the most important are briefly presented in this documentation:

- sendMoveToBluetooth(fromRow: Int, fromColumn: Int, toRow: Int, toColumn: Int)
- findBestMoveByStockfish()
- setDifficultyLevelStockfish(difficulty: String)
- moveByStockfish()
- onClickSquare(square: Pair<Int, Int>)
- resetBoard()
- movePiece(fromRow: Int, fromColumn: Int, toRow: Int, toColumn: Int)
- moveFromChessboard()
- saveRecording(context: Context, recordingName: String)
- saveMovesToFile(context: Context, filename: String)
- loadMovesFromFile(filename: String)

- moveForward() and moveBack()
- markPossibleMoves(selectedSquare: Pair<Int, Int>, currentBoardState: ChessBoardState, addOnOldPossibleMoves: Boolean)

The first method from the list is **sendMoveToBluetooth**, that uses the bluetoothManager object to call the bluetoothManager.sendDataToDevice() function. This is required in order to respect the architecture presented in subsection 5.3.1.

The following three methods are all related to communication with the chess engine. The function **setDifficultyLevelStockfish** is called when the Play Screen object is constructed based on the option that the user has selected. This function sets a parameter from the chessBoardState that is used by the ChessEngine class during communication with the executable.

The **moveByStockfish()** method has a crucial role for this project. It uses the ChessEngine object to convert the current ChessBoardState into FEN and gets the best move by using the chessEngine.getBestMove function. It is then responsible for converting this move from the chess specific letter notation (ex. b1a3) to a set of coordinates used by the rest of the application. It also calls the movePiece() function to make the move on the virtual chessboard and the sendMoveToBluetooth() to make the move on the physical chessboard as well.

The **movePiece()** method is called whenever a move is made, irrespective of who makes it. It takes as parameters the coordinates for the initial square and target square and updates the board state accordingly. It is also responsible for updating the move counter, keeping track of the turn (white or black) and deciding based on values stored in the board state if the next move should be executed by the chess engine or if the application has to wait for a move from the physical chessboard.

Another important method is the **moveFromChessboard()**. Called from the movePiece() method presented above, this function uses the BluetoothManager object to receive data from the chessboard, it transforms the data from a String to a set of coordinates and calls again the movePiece() method for the move that is received via Bluetooth.

The methods containing the keyword ”file” are used to implement the functionalities of game recording and replay. The methods **saveRecording()** and **saveMovesToFile** are using data provided by the user through the RecordGameScreen UI element to save all the moves made by the players on the chessboard during the recording process to a file with their desired name. These files can then be accessed through a method **getAllRecordingFiles** and the user can select from the WatchRecordingScreen one of the records and load it into the application. Once the moves are loaded via a call of the **loadMovesFromFile()** method, the user has the ability to re-watch and analyze the game by using two buttons which directly call the methods: **MoveForward()** and **moveBack()**.

The functionality presented in the previous paragraph has been developed after the initial structure of the application and therefore had to be included in the architecture and reuse as many pre-existing components. For this, extra parameters were added to the

ChessBoardState: recordingGame: Boolean, moves: MutableList(List(Int)), currentMove: Int, watchRecording: Boolean, captures: List(Pair(Int, String)). The boolean flags are used by other methods to ensure they do not interfere with the recording or watching of a record processes, while the moves list is used to keep track of all the moves made by the players. The other list is necessary because previously, the captured pieces were no longer of interest, so they were simply discarded, but in order to allow the player to move back in a replay, some of the captured pieces need to be revived, so the captures provides this information with a string representing the piece and an integer representing the move where it was captured.

The last method from the list, the **markPossibleMoves()**, takes a chess square as a parameter and uses the chessboard state to check if there is a piece on the square or not. Then, by respecting the rules of chess, it is checking all the possible moves for the respective piece and adds all the squares that piece can move to legally to a list which is used by the UI. Also, when the user is playing on the application directly, he can only move a piece to one of the squares marked as a possible move, making sure only legal moves are played.

Chapter 6

Testing and Validation

6.1 Bluetooth Communication

The Bluetooth communication has proved to be reliable and has caused almost no issues during testing. In some isolated cases, the HC-05 module loses connection unexpectedly which causes problems for the current activity, but it can be reconnected directly from the application making this a minor inconvenient.

The communication protocol and turn based exchange of information has proved to be sufficient for this application and stable, causing no problems during testing.

6.2 Pieces Detection

Pieces detection is crucial for the well functioning of all game modes provided by the chessboard. The Hall Sensors performance is satisfactory and they are able to detect the presence of the magnets through the Plexiglas surface.

The multiplexing of the outputs, the algorithms for reading the board state by detecting human interventions and tracking all moves have been successfully implemented. These allow for reliable game recording and move tracking functionalities. The results validate that Hall sensors are an effective solution for detecting chess pieces and confirm that the hardware connections have been constructed correctly.

Isolated problems may arise due to imperfect manipulation of the pieces by human players that can trigger the sensors without intention and cause unwanted detections that alter the state of the game.

6.3 Moves Execution

The CoreXY system has been successfully implemented and controlled. Precise positioning of the electromagnet can be obtained anywhere on the two dimensional working envelope by providing coordinates in millimeters. The hardware build has been realized

according to the designs and the gap between the electromagnet and the electronics has been minimized to under 2mm.

Algorithms for path computation of the electromagnet that allows all pieces to move according to the specifications have been successfully implemented. Therefore, the path taken is always correct and has shown no errors during testing.

The motors performance has been unsatisfactory. The motor has to perform 2048 steps/revolution and the ULN2003 driver can provide a control signal of maximum of 15 revolutions/minute. After simple computations it results that the highest performance we can achieve is 512 steps/second, meaning it takes approximately 4 seconds to complete a rotation. Considering the forward kinematics, a full rotation causes a 32mm displacement on the CoreXY system. Considering these numbers and the working envelope of 320mm x 496mm, we can compute that it can take up to 40 seconds for the electromagnet to travel the whole height of the chessboard and 60 seconds for it to travel the width. Therefore, the project is functional but the time it takes for the electromagnet to travel is recommended to be improved by an upgrade to more capable stepper motors.

The magnetic connection between the pieces and the electromagnet is not strong enough to ensure adequate performance. The system is able to ensure the piece follows the path in only about 75 percent of the cases in the current version. The main issue has been identified as the inconsistency of the magnetic field of the permanent magnets attached at the base of the pieces. This means that some magnets are stronger than others and some pieces provide better performances than others, and for solving this issue, higher quality magnets are recommended. This can be improved further with an upgrade to a stronger electromagnet or by reducing the friction between the pieces and the playing surface, either by reducing pieces weight or changing the material of the contact surface between the two.

6.4 Stockfish Integration

Stockfish has been successfully integrated in the Android application. The chess engine is executed successfully and communication with the methods provided by the implementation is reliable. The Stockfish chess engine is receiving the chess position generated as a FEN String and the move suggestions are collected and either displayed on the UI or executed both on the UI and on the chessboard depending on the game mode. Also, the difficulty level parameter can be set successfully, and overall the Stockfish functionality is stable.

6.5 Android Application Performance

The Android application offers access and control to all the functionalities provided by the project. Management of Bluetooth connection is functioning as expected and the menus provide access to all game modes provided. The current limitations are the lack

of possibility to select the color (User always plays white) and the lack of the puzzles functionality which will be developed in the future.

The ”Play” function correctly displays the chessboard UI and allows the player to play chess against the Stockfish chess engine. The limitation is related to the special moves of castling and ”En passsant”. The record game functionality correctly illustrates and records all moves received from the chessboard and allows the user to save the recording in the internal memory. The watch record functionality is working as expected and allows the user to load (or delete) recordings and to analyze the game by displaying the board state on demand after every move as well as a list of moves and captures.

Chapter 7

User's manual

7.1 Automated Chessboard

7.1.1 Powering the device

The Automated Chessboard is designed to function from the initial chess position. Therefore, the user must place the pieces in the initial position and then plug in the power cord, which can be found on one of the short sides of the chessboard.

It is forbidden for the power to be unplugged during the functioning of the system and electromagnet movement. In all scenarios, the user must obey the following instruction for powering off the chessboard to avoid decalibration of the chessboard's mechanical systems.

When the chessboard is no longer in use, the user must make sure to disconnect from the chessboard using the Android application, and once the motors inside the board stop moving, the chessboard can be unplugged and stored for later use. Under no circumstance must the chessboard come into contact with liquids or extreme moisture.

7.1.2 Capturing a piece

It is very important to respect the correct order of moving the pieces when making a capture. First, lift the captured piece and place it on a free circular spot on the sides of the playing surface. Then, only after placing down the captured piece, the piece that is capturing can be lifted from the initial square and placed in the new location.

7.2 Android Application

7.2.1 Start up and Board Connection

The Android Application opens up with the Menu from the first image from figure 7.1. In the top left hand side, a "Connect Chessboard" button can be observed. This button

must be pressed in order to connect the application to the chessboard via Bluetooth. Once the button is pressed, the pop-up from the second image shows up and the ”HC-05” device must be selected from the list.

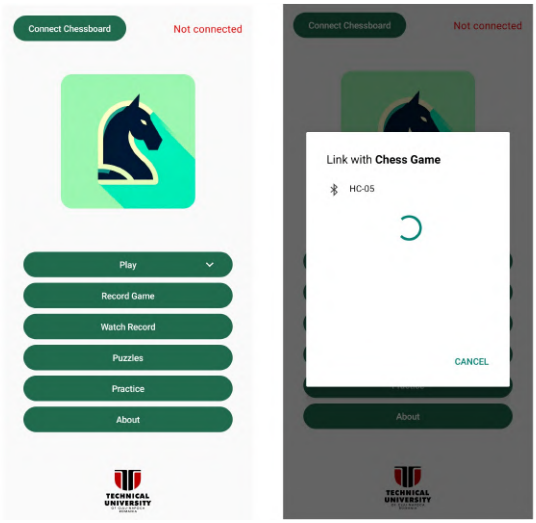


Figure 7.1: Android Application Menu and Bluetooth Connection

If the board is connected, the ”Connected” text will on the right side of the button. If the user desires to disconnect from the chessboard, the ”Disconnect Chessboard” button which appears in the same location is accessed.

7.2.2 Game Recording

Only after the Chessboard is connected, the user can access the functionalities of game recording or playing against Stockfish on the chessboard. In order to record a game of chess, the ”Record Game” button must be pressed which will lead the player to the screen from figure 7.2. The user must place all the pieces in their initial position and then press the start recording button. While the recording is on, all the moves made on the chessboard will be visible on the digital representation from the display and listed at the bottom of the display, as can be seen in the middle representation of the same figure.

Once the game is over, the ”Stop Recording” button is pressed and the user is directed to the pop-up from the last illustration from figure 7.2. Here, the user can either discard the recording or enter a desired name for it and saving.

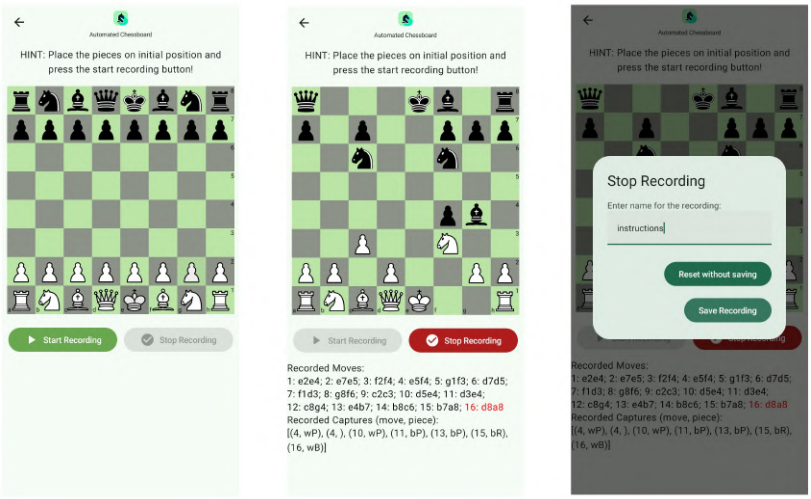


Figure 7.2: Recording a Game

Once a recording is saved, it can be accessed from the "Watch Recording" section which can be seen in figure 7.3. Here, the user must select one of the files from the top button and load it by pressing the "Load" button. Then, by using the left and right arrows, the user can navigate the list of moves and see them on the graphical representation. In order to facilitate the analysis, the user can also see the current move highlighted with red in the list of moves. Once analysis is complete and a record is not needed anymore, it can be deleted by using the red button near the "Load". The Game Recording implementation is listed in the appendix A.4.

Figure 7.3: Watching a Record

7.2.3 Play Against Stockfish

If the user desires to play a game against Stockfish, the ”Play” button should be accessed from the main menu. A drop down that allows the user to select the desired difficulty is then visible and once the play button is pressed, the game can start (Figure 7.4). The user has to place the pieces on their initial positions and the game can be played normally from the chessboard. A representation of all the moves can be seen on the display, and the board is able to execute the Stockfish moves independently.

Figure 7.4: Play against Stockfish

Chapter 8

Conclusions

In addressing the challenge of integrating automation and technological advancements into the game of chess, this thesis has successfully developed an Automated Chessboard. This solution enables users to play chess on a physical board against one of today's most powerful AI chess engines, while also providing functionalities for recording games played by human players. These functionalities not only enhance the gaming experience by offering a formidable AI opponent on a physical board but also serve as a valuable tool for players seeking to improve their skills through detailed analysis of automatically recorded games.

8.1 Critical Analysis of Results

The successful development process can be assessed by comparing the results with the initial objectives and specifications. The results obtained are exceptionally good and can be divided into software and hardware.

8.1.1 Hardware

For the hardware part, an automated chessboard has been constructed. The pieces detection using Hall Sensors is functioning as expected and represents a reliable solution for this application. The CoreXY system is also operates correctly and allows for precise positioning of the end effector at any position on the working envelope. Utilizing V-Slot linear rails with the well designed supporting elements manufactured out of wood have provided a solid foundation, minimizing dimensions while ensuring smooth and durable performance.

However, the stepper motors used are not an optimal choice due to the limitations in torque and speed. Another downside of the hardware implementation is the performance of the electromagnet in providing magnetic connection with the pieces during movement. Cases when the electromagnet cannot provide sufficient force to slide the pieces are enough to consider the current implementation unreliable for the pieces movement part.

Overall, considering the limited budget allocated for this prototype, the hardware implementation is deemed successful as it meets the initial requirements. While there is ample room for improvement in future iterations, the concepts presented throughout the paper have been effectively demonstrated.

8.1.2 Software

The software developed for this thesis project is composed of two main parts. The Arduino software has successfully been implemented with different components that provide the functionalities stated by the requirements. Detection of human moves by using the 6 multiplexers and the Hall Sensors is working with no major issues and the electromagnet can be positioned accurately at any place in the working envelope by using the stepper motors and the inverse kinematics of the CoreXY system. Furthermore, communication with the Android application via Bluetooth has also been implemented and allows for synchronization and cooperation between the software components such that the end goal is achieved.

The Android application software has also been implemented successfully to provide a user interface for the chessboard and solve the problem of limited resources of the embedded system on the hardware part. Integrating the Stockfish chess engine as a component of the application enables high-quality move suggestions. The engine runs directly on the Android device, ensuring independence from any external server.

Moreover, as outlined in the objectives, the Android application offers functionalities for game recording, analysis, practice, and playing against Stockfish through a user-friendly interface. The application communicates with the Arduino via Bluetooth, ensuring correct information flow based on the selected game type.

In conclusion, the software components fulfill all the requirements stated in the specifications and are considered a major success. Best practices have been followed wherever possible, allowing for development of additional features in the future.

8.2 Future Development

Even though significant results have been achieved, this thesis has only developed a prototype, and substantial room for improvement remains, particularly in the hardware component.

The main issue that needs addressing is the improvement of the reliability of the magnetic connection between the pieces and the electromagnet. For this, testing of different materials for the board needs to be conducted and determine if a better choice than Plexiglas is available. Also, pieces weight could be reduced and the friction coefficient between the material of the base of the pieces and the board should also be minimized.

Another area for hardware improvement is acquiring more performant stepper motors and developing a method to automatically calibrate the position of the end effector

during initialization. This can be achieved by adding sensors to the origin of the XY coordinate system, moving the electromagnet until the sensors are triggered, and setting the motors' reference position at that point.

A strong argument for separating the UI from the chessboard is that they can be developed independently. Therefore, upgrades can be made to the chessboard without altering the Android application and vice versa. If any of the previous developments are added to a following version of the chessboard, the Android application in its current version can be used.

In terms of future development for the Android application, there are a number of features that can be added. The first would be to add a check and checkmate verification method that only allows the user to make legal moves when the king is involved and end the game in case the king has no escape. Additionally, the color selection and puzzles sections, which are currently not implemented, should be developed to offer a more complete experience to the user.

Bibliography

[1] A. E. Soltis, "History of chess," 2024, accessed: 2024-06-21. [Online]. Available: https://www.britannica.com/topic/chess/History

[2] R. van Sol, "Handroid: A revolutionary chess evaluation system," 2015, accessed: 2024-07-02. [Online]. Available: http://www.chesseval.com/ChessEvalJournal/Handroid.htm

[3] M. Campbell, A. J. H. Jr., and F. hsiung Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, 2002.

[4] R. Degni, "The ultimate checkmate: Ai and chess engines," 2023, accessed: 2024-07-02. [Online]. Available: https://www.codemotion.com/magazine/ai-ml/the-ultimate-checkmate-ai-and-chess-engines/

[5] D. Georgyan, "Alphazero vs stockfish 8: A landmark battle of human and artificial intelligence in chess," 2023, accessed: 2024-07-10. [Online]. Available: https://medium.com/@david_georgyan/alphazero-vs-stockfish-8-a-landmark-battle-of-human-and-artificial-intelligence-in-chess-ac625

[6] "Leela chess zero networks," 2024, accessed: 2024-07-02. [Online]. Available: https://chessify.me/blog/leela-chess-zero-networks

[7] D. Urting and Y. Berbers, "Marineblue: A low-cost chess robot." Leuven, Belgium: KU Leuven, Department of Computer Science, 2003. [Online]. Available: https://www.researchgate.net/publication/220785821_MarineBlue_A_Low-cost_Chess_Robot

[8] F. A. T. Al-Saedi and A. H. Mohammed, "Design and implementation of chess-playing robotic system," *International Journal of Computer Science Engineering and Technology (IJCSET)*, vol. 5, no. 5, pp. 90–98, 2015. [Online]. Available: http://www.ijcset.net/docs/Volumes/volume5issue5/ijcset2015050501.pdf

[9] S. W. Maciej A. Czyzewski and A. Laskowski, "Chessboard and chess piece recognition with the support of neural networks," in *FOUNDATIONS OF COMPUTING AND DECISION SCIENCES*, vol. 45, 2020, pp. 257–278. [Online]. Available: https://intapi.sciendo.com/pdf/10.2478/fcds-2020-0014

[10] S. U. Jialin Ding, "Chessvision: Chess board and piece recognition," 2016. [Online]. Available: https://web.stanford.edu/class/cs231a/prev_projects_2016/CS_231A_Final_Report.pdf

[11] Greg06, "Automated chessboard," 2023, accessed: 2024-02-14. [Online]. Available: https://www.instructables.com/Automated-Chessboard/

[12] Jul 2022. [Online]. Available: https://www.theguardian.com/sport/2022/jul/24/chess-robot-grabs-and-breaks-finger-of-seven-year-old-opponent-moscow

[13] Google, "Kotlin on android documentation," 2024, accessed: 2024-01-19. [Online]. Available: https://developer.android.com/kotlin

[14] ——, "Jetpack compose documentation," 2024, accessed: 2024-01-19. [Online]. Available: https://developer.android.com/develop/ui/compose

[15] ——, "Android application architecture," 2024, accessed: 2024-04-01. [Online]. Available: https://developer.android.com/topic/architecture#common-principles

[16] ——, "Android bluetoothadapter documentation," 2024, accessed: 2024-06-13. [Online]. Available: https://developer.android.com/reference/android/bluetooth/BluetoothAdapter

[17] ——, "Android bluetoothmanager documentation," 2024, accessed: 2024-05-19. [Online]. Available: https://developer.android.com/reference/android/bluetooth/BluetoothManager

[18] Allegro MicroSystems, "A3141-2-3-4 datasheet," https://www.elecrow.com/download/A3141-2-3-4-Datasheet.pdf, 2005, accessed: 2024-07-01.

[19] M. McCauley, "Accelstepper library documentation," 2024, accessed: 2024-06-13. [Online]. Available: https://www.airspayce.com/mikem/arduino/AccelStepper/

[20] Google, "Android bluetooth permissions documentation," 2024, accessed: 2024-06-28. [Online]. Available: https://developer.android.com/develop/connectivity/bluetooth/bt-permissions

Appendix A

Relevant Code

A.1 Human Moves Detection - Board.cpp

```
// Called when human intervention detected and checks whether a piece was placed on the square or removed and acts accordingly
void Board::processDetection(int row, int column){
  // Human intervention detected -> check whether a piece was placed on the square or removed
  if(presence[row][column] == 1){
    // Intervention was to remove a piece, memorize its original place
    liftedPiece[0] = row;
    liftedPiece[1] = column;
  } else if(presence[row][column] == 0){
    // Intervention was to place down a previously lifted piece (assumes lifted piece exists in the memory)
    char piece = state[liftedPiece[0]][liftedPiece[1]];

    // Check if we have moved a black piece, we are in process of making a capture so we do not send that to Bluetooth yet
    if ((piece == 'B' || piece == 'K' || piece == 'R' || piece == 'P' || piece == 'Q' || piece == 'N') && (turn != 2)) {
      move(liftedPiece[0], liftedPiece[1], row, column); // Updates the board state
    }else{
      move(liftedPiece[0], liftedPiece[1], row, column); // Updates the board state

      // Sends the data to Bluetooth
      String dataToSend = String(liftedPiece[0]) + " " + String(liftedPiece[1]) + " " + String(row) + " " +
```

```
          String(column);

      // Send the string to the HC-05 via Serial3
      Serial3.println(dataToSend);

      // Signals its Stockfish turn
      if(turn == 1)
        turn = 0;

      // Print the data to Serial for debugging
      Serial.println("Sent to Bluetooth: " + dataToSend);
    }
  }
}

// Reads all sensors and detects any intervention on the pieces
void Board::readPiecePresence(){
  Serial.println("readPiecePresence() called");
  // Read sequentially 6 squares at a time
  for(int i = 0; i < 16; i++){
    // Converts the value of i to binary and sends each bit to
       mux on pins S0, S1, S2 and S3
    digitalWrite(S0, (i >> 0) & 1); // Shifts i by 0 positions to
       the right and extracts the last digit
    digitalWrite(S1, (i >> 1) & 1);
    digitalWrite(S2, (i >> 2) & 1);
    digitalWrite(S3, (i >> 3) & 1);
    digitalWrite(S0_1, (i >> 0) & 1);
    digitalWrite(S1_1, (i >> 1) & 1);
    digitalWrite(S2_1, (i >> 2) & 1);
    digitalWrite(S3_1, (i >> 3) & 1);

    // Read the value of each mux output for a certain square on
       the board and update the array
    int row = (i/8 + 1) * 7 + i/8 - i;

    // For each mux we check the multiplexor outputs to detect
       changes in pieces presence
    checkHumanIntervention(row, 1 - i/8, mux_o1);
    checkHumanIntervention(row, 3 - i/8, mux_o2);
    checkHumanIntervention(row, 5 - i/8, mux_o3);
    checkHumanIntervention(row, 7 - i/8, mux_o4);
    checkHumanIntervention(row, 9 - i/8, mux_o5);
    checkHumanIntervention(row, 11 - i/8, mux_o6);
```

```
59
60      delay(10);
61    }
62  }
63
64  void Board::checkHumanIntervention(int row, int column, int
       muxNumber){
65    // For each square we check human intervention
66    if (presence[row][column] != !digitalRead(muxNumber)) {
67      // Human intervention detected
68      // Added filtering to try and eliminate false readings and
          disturbances
69      bool falseReading = false;
70      for(int k = 0; k < 5; k++){
71        if(presence[row][column] == !digitalRead(muxNumber))
72          falseReading = true;
73        delay(50);
74      }
75
76      if(!falseReading){
77        // Good reading so we continue the process
78        processDetection(row, column);
79        presence[row][column] = !digitalRead(muxNumber); // Update
            the presence matrix
80        printPiecePresence();
81      }
82    }
83  }
```

A.2 CoreXY implementation - CoreXY.cpp

```
1  const int stepsPerRevolution = 2038; // Number of steps per
       complete rotation
2  bool isMoving = false; // Sets movement status for the motors
3
4  CoreXY::CoreXY() {
5    // Pins entered in sequence IN1-IN3-IN2-IN4 for proper step
       sequence
6    leftMotor = AccelStepper(AccelStepper::FULL4WIRE, 44, 46, 45,
       47);
7    rightMotor = AccelStepper(AccelStepper::FULL4WIRE, 30, 33, 29,
       32);
8  }
```

```
9
10 void CoreXY::initialize() {
11   // Setup motor performances
12   leftMotor.setMaxSpeed(625);
13   leftMotor.setAcceleration(900);
14   rightMotor.setMaxSpeed(625);
15   rightMotor.setAcceleration(900);
16
17   // Set initial position as origin
18   leftMotor.setCurrentPosition(0);
19   rightMotor.setCurrentPosition(0);
20 }
21
22 void CoreXY::moveTo(int x, int y){
23     if (!isMoving) { // Check again in case state changed
24       setMotorsTarget(x, y);
25       isMoving = true;
26       while(isMoving){
27         controlMovement();
28       }
29   }
30 }
31
32 void CoreXY::controlMovement() {
33   if (isMoving) {
34     leftMotor.run();
35     rightMotor.run();
36     if (leftMotor.distanceToGo() == 0 &&
          rightMotor.distanceToGo() == 0) {
37       leftMotor.stop();
38       rightMotor.stop();
39       isMoving = false;
40     }
41   }
42 }
43
44 void CoreXY::setMotorsTarget(int x, int y) {
45   float q1 = 1.0/32.0 * (x + y);
46   float q2 = 1.0/32.0 * (x - y);
47   Serial.print("x: ");
48   Serial.print(x);
49   Serial.print("; y: ");
50   Serial.print(y);
51   Serial.print("; q1: ");
```

```
  Serial.print(q1);
  Serial.print("; q2: ");
  Serial.println(q2);

  leftMotor.moveTo(q1 * stepsPerRevolution);
  rightMotor.moveTo(q2 * stepsPerRevolution);
}

void CoreXY::returnToInitialPosition() {
  Serial.println("Returning to initial position.");

  leftMotor.moveTo(0); // Move to initial position (0)
  rightMotor.moveTo(0); // Move to initial position (0)

  isMoving = true;

  while (leftMotor.distanceToGo() != 0 ||
     rightMotor.distanceToGo() != 0) {
    leftMotor.run();
    rightMotor.run();
  }

  Serial.println("Returned to initial position.");
  isMoving = false; // Reset movement status
}
```

A.3 Stockfish integration - ChessEngine.kt

```
/*
Class that creates an instance of Stockfish and provides
   functions that facilitate communication
between the chess engine and our app.
*/
class ChessEngine(context: Context) {
    private var process: Process? = null

    init {
        // Obtain the path to the Stockfish executable within the
           application's internal storage
        val stockfishPath = File(context.filesDir,
           "stockfish").absolutePath
        // Start Stockfish process
        process = Runtime.getRuntime().exec(stockfishPath)
```

```
13      }
14
15      fun getBestMove(fen: String, difficulty: Int): String? {
16          val outputStream =
                OutputStreamWriter(process!!.outputStream)
17          val inputStream =
                BufferedReader(InputStreamReader(process!!.inputStream))
18
19          outputStream.write("setoption name Skill Level value
                $difficulty\n")
20          outputStream.write("position fen $fen\n")
21          outputStream.write("go movetime 2000\n")
22          outputStream.flush() // make sure we send the commands
                immediately
23
24          var line: String?
25          var bestMove: String? = null
26          while (inputStream.readLine().also { line = it } != null)
                {
27              line?.let { Log.d("Stockfish Output", it) } // Log
                    everything Stockfish outputs
28              // If line starts with "bestmove" extract the response
29              if (line?.startsWith("bestmove") == true) {
30                  bestMove = line!!.split(" ")[1]
31                  break
32              }
33          }
34
35          return bestMove
36      }
37
38      fun close() {
39          process?.destroy()
40      }
41
42      // Interface between our position storage and FEN (used by
            Stockfish)
43      fun getFenFromChessBoardState(boardState: ChessBoardState):
            String {
44          val fenStringBuilder = StringBuilder()
45          boardState.piecesState.forEach { row ->
46              var emptySquares = 0
47              row.forEach { square ->
48                  if (square.isEmpty()) {
```

```
49                      emptySquares++
50                  } else {
51                      if (emptySquares > 0) {
52                          fenStringBuilder.append(emptySquares)
53                          // we found a piece and append the fen
54                          emptySquares = 0
55                      }
56                      // Convert our notation to FEN notation
57                      if(square[0] == 'b')
58                          // FEN black pieces are lowercase
59                          fenStringBuilder.append(square[1]
60                              .lowercaseChar())
61                      else
62                          // FEN white pieces remain uppercase
63                          fenStringBuilder.append(square[1])
64                  }
65              }
66              if (emptySquares > 0) {
67                  fenStringBuilder.append(emptySquares)
68              }
69              fenStringBuilder.append('/')
70          }
71          fenStringBuilder.removeSuffix("/")
72
73          // Add turn black/white
74          if(boardState.whiteTurn)
75              fenStringBuilder.append(" w")
76          else
77              fenStringBuilder.append(" b")
78
79          // We have to add white and black castling possibility,
               en passant and move counter
80          fenStringBuilder.append(" KQkq - 0
               ${boardState.moveCounter}")
81
82          return fenStringBuilder.toString()
83      }
84  }
```

A.4 Game Recording implementation - ChessGameViewModel.kt

```
1
2  // Updates the gameRecording status
3      fun startRecording(){
4          bluetoothManager.sendDataToDevice("s\n")
5          _chessBoardUiState.update {
               ChessBoardState(playVsStockfish = false) }   //
               Rebuilds the board state
6          val currentBoardState = _chessBoardUiState.value
7          // Send start Recording Mode to the board
8          _chessBoardUiState.value =
               currentBoardState.copy(recordingGame = true, moves =
               mutableListOf())
9
10         Log.d("Recording", "Start Recording sent to chessboard
               and recordingGame flag set
               ${_chessBoardUiState.value.recordingGame}")
11     }
12
13     // Saves recording to a file
14     fun saveRecording(context: Context, recordingName: String){
15         val currentBoardState = _chessBoardUiState.value
16
17         saveMovesToFile(context, recordingName) // Saves all
               moves to the specified file
18
19         _chessBoardUiState.value =
               currentBoardState.copy(recordingGame = false, moves =
               mutableListOf())
20         resetBoard()
21     }
22
23     // Resets board and recording status
24     fun resetRecording(){
25         val currentBoardState = _chessBoardUiState.value
26         _chessBoardUiState.value =
               currentBoardState.copy(recordingGame = false, moves =
               mutableListOf())
27
28         resetBoard()
29     }
30
```

```
31      // Function that waits for data to be recieved from
            chessboard and executes the move
32      fun moveFromChessboard(){
33          Log.d("bluetooth", "moveFromChessboard() called")
34
35          val receivedMove =
                bluetoothManager.receiveDataFromDevice()
36
37          Log.d("bluetooth", "Recived move: $receivedMove")
38
39          // Data received in format (fromRow, fromColumn, toRow,
                toColumn)
40          val trimmedMessage = receivedMove.trim().split(" ")
41
42          if (trimmedMessage.size == 4) {
43              val fromRow = trimmedMessage[0].toInt()
44              val fromColumn = trimmedMessage[1].toInt() - 2
45              val toRow = trimmedMessage[2].toInt()
46              val toColumn = trimmedMessage[3].toInt() - 2
47
48              // If we are recording the game, save the move in the
                    temporary list
49              val currentBoardState = _chessBoardUiState.value
50
51              Log.d("Record", "current flag:
                    ${currentBoardState.recordingGame}")
52
53              if(currentBoardState.recordingGame == false){
54                  // Try to filter false readings if it asks to
                        move a piece from a empty position
55                  if(currentBoardState.piecesState[fromRow]
56                  [fromColumn] != "" && (fromColumn in 0..7))
57                      movePiece(fromRow, fromColumn, toRow,
                            toColumn)
58                  else
59                      moveFromChessboard()
60              } else{     // we are recording
61                  // Create a copy of the current moves list and
                        add the new move
62                  val updatedMoves =
                        currentBoardState.moves.toMutableList()
63
64                  if(toColumn < 8 && toColumn > -1){      // Move
                        happened on the chessboard (8x8)
```

```
65                      val move = listOf<Int>(fromRow, fromColumn,
                            toRow, toColumn)
66
67                      updatedMoves.add(move)
68                      Log.d("Record", "Move
                            ${currentBoardState.currentMove + 1}
                            added: $fromRow-$fromColumn to
                            $toRow-$toColumn")
69
70                      // Confirm to chessboard we are still
                            recording:
71                      bluetoothManager.sendDataToDevice("r\n")
72                      Log.d("Recording", "r sent to bluetooth")
73
74                      // Update the UI state with the new moves list
75                      _chessBoardUiState.value =
                            currentBoardState.copy(moves =
                            updatedMoves, currentMove =
                            currentBoardState.currentMove + 1)
76                      movePiece(fromRow, fromColumn, toRow,
                            toColumn)
77
78                  } else{     // Piece was moved outside the 8x8
79                      val capturedPiece = currentBoardState
80                      .piecesState[fromRow][fromColumn]
81                      val captureMoveIndex =
                            currentBoardState.currentMove + 1 //
                            Capture is part of last executed move + 1
82                      val capture = Pair<Int,
                            String>(captureMoveIndex, capturedPiece)
83
84                      val updatedCaptures =
                            currentBoardState.captures.toMutableList()
85                      updatedCaptures.add(capture)
86
87                      Log.d("Record", "Capture of piece added at
                            move: ${captureMoveIndex}, for piece
                            ${capturedPiece}")
88                      // Update the UI state with the captured Piece
89                      _chessBoardUiState.value =
                            currentBoardState.copy(captures =
                            updatedCaptures)
90
```

```
91                      // Confirm to chessboard we are still
                           recording:
92                      bluetoothManager.sendDataToDevice("r\n")
93                      Log.d("Recording", "r sent to bluetooth")
94
95                      movePiece(fromRow, fromColumn, toRow,
                           toColumn)
96                  }
97              }
98          }
99      }
100
101     // Executes current move
102     fun moveForward(){
103         val currentBoardState = _chessBoardUiState.value
104         val currentMove = currentBoardState.currentMove
105         val movesList = currentBoardState.moves
106
107         // Execute a forward move
108         if (currentMove < movesList.size) {
109             val toExecute = movesList[currentMove]
110             Log.d("movesNavigation", "Attempting move
                   ${currentMove + 1}: ${toExecute[0]},
                   ${toExecute[1]}, ${toExecute[2]}, ${toExecute[3]}")
111             movePiece(toExecute[0], toExecute[1], toExecute[2],
                   toExecute[3])
112             _chessBoardUiState.value =
                   _chessBoardUiState.value.copy(currentMove =
                   currentBoardState.currentMove + 1)
113         }
114     }
115
116     // Executes previous move
117     fun moveBack() {
118         val currentBoardState = _chessBoardUiState.value
119         val currentMove = currentBoardState.currentMove
120         val movesList = currentBoardState.moves
121         val captures = currentBoardState.captures
122
123         if (currentMove > 0) {
124             // Execute the reverse of the move
125             val toExecute = movesList[currentMove - 1]
126             Log.d("movesNavigation", "Attempting move
                   ${currentMove}: ${toExecute[0]}, ${toExecute[1]},
```

```
                ${toExecute[2]}, ${toExecute[3]}")
127             movePiece(toExecute[2], toExecute[3], toExecute[0],
                    toExecute[1])
128
129
130             val newPiecesState =
                    _chessBoardUiState.value.piecesState.map {
                    it.toMutableList() }.toMutableList() // Mutable
                    copy of piecesState
131             // Check if we had a capture in this move so we
                    "revive" the piece that was captured
132             val capture = captures.find { it.first == currentMove
                    }
133             capture?.let {
134                 val capturedPiece = it.second
135                 // add to the copy of piecesState matrix the
                        revived piece
136                 newPiecesState[toExecute[2]][toExecute[3]] =
                        capturedPiece
137             }
138
139             // Update the chess board state
140             _chessBoardUiState.value =
                    _chessBoardUiState.value.copy(
141                 currentMove = currentMove - 1,
142                 piecesState = newPiecesState
143             )
144         }
145     }
146
147     // Function used to save moves
148     fun saveMovesToFile(context: Context, filename: String) {
149         // Define the folder for recordings
150         val recordingsFolder = File(context.filesDir,
                "recordings")
151
152         // Create the folder if it doesn't exist - for first time
                saving only
153         if (!recordingsFolder.exists()) {
154             recordingsFolder.mkdir()
155         }
156
157         // Save all moves and captures from ChessBoardState to a
                file
```

```
158         val filename = "$filename.txt"
159         val file = File(recordingsFolder, filename)
160         file.printWriter().use { out ->
161             _chessBoardUiState.value.moves.forEach { move ->
162                 out.println(move.joinToString(","))
163                 Log.d("Recording", "Move Saved to $filename:
                        ${move.joinToString(",")}")
164             }
165             // Write a separator line
166             out.println("===")
167             Log.d("Recording", "Separator Saved to $filename")
168             _chessBoardUiState.value.captures.forEach { capture ->
169                 out.println("${capture.first},${capture.second}")
170                 Log.d("Recording", "Capture Saved to $filename:
                        ${capture.first},${capture.second}")
171             }
172         }
173     }
174
175     fun getAllRecordingFiles(context: Context): List<File> {
176         val recordingsFolder = File(context.filesDir,
                "recordings")
177         if (!recordingsFolder.exists()) {
178             return emptyList()
179         }
180         return recordingsFolder.listFiles()?.toList() ?:
                emptyList()
181     }
182
183     // Function to load moves from a file
184     fun loadMovesFromFile(filename: String) {
185         Log.d("FileLoad", "loadMovesFromFile called for file:
                $filename")
186         val file = File(filename)
187
188         if (file.exists()) {
189             Log.d("FileLoad", "File Exists")
190
191             // Read and log all lines for debug purposes
192             val fileContent = file.readText()
193             Log.d("FileLoad", "File Content:\n$fileContent")
194
195             val lines = file.readLines()
196             val separatorIndex = lines.indexOf("===")
```

```
197
198             // Parse moves
199             val loadedMoves = lines.take(separatorIndex).map {
                    line ->
200                 line.split(",").map { it.toInt() }
201             }
202
203             // Parse captures
204             val loadedCaptures = lines.drop(separatorIndex +
                    1).map { line ->
205                 val (moveIndex, piece) = line.split(",")
206                 moveIndex.toInt() to piece
207             }
208
209             // Logs for debug only
210             loadedMoves.forEachIndexed { index, move ->
211                 val (fromRow, fromColumn, toRow, toColumn) = move
212                 Log.d("FileLoad", "Move ${index + 1}:
                        $fromRow-$fromColumn to $toRow-$toColumn")
213             }
214
215             loadedCaptures.forEachIndexed { index, capture ->
216                 val (moveIndex, piece) = capture
217                 Log.d("FileLoad", "Capture ${index + 1}:
                        MoveIndex=$moveIndex, Piece=$piece")
218             }
219
220             // Reset board
221             resetBoard()
222
223             // Update ChessBoardState with the moves and captures
                    list
224             _chessBoardUiState.value =
                    _chessBoardUiState.value.copy(
225                 moves = loadedMoves.toMutableList(),
226                 captures = loadedCaptures.toMutableList(),
227                 watchRecording = true
228             )
229         } else {
230             Log.d("FileLoad", "File does not exist")
231         }
232     }
233
234     fun deleteFile(fileName: String):Boolean {
```

```
235         val file = File(fileName)
236         return try {
237             if (file.exists()) {
238                 file.delete()
239             } else {
240                 false
241             }
242         } catch (e: Exception) {
243             e.printStackTrace()
244             false
245         }
246     }
```

Appendix B

Published Papers

Published a paper at the Automation and Computer Science Conference (ACSC) 2023, entitled ”Sigma Soccer”.