IMPERIAL

A Smart Chessboard for Technology Demonstration Outreach

Author

Ahmed Elkouny

CID: 01902185

Supervised by

Dr Edward Stott

Second Marker

Prof. Thomas J W Clarke

A Thesis submitted in fulfillment of requirements for the degree of
Master of Engineering in Electronic and Information Engineering

Department of Electrical and Electronic Engineering
Imperial College London

2025

Abstract

This project details the design and implementation of an autonomous smart chessboard aimed at merging the tactile experience of physical gameplay with the features of modern digital chess. The primary objectives were to create a board that can automatically track piece positions, provide intuitive visual feedback to the user, and physically actuate an opponent's moves for remote or AI-driven games.

The system architecture was designed to separate complex game logic from real-time hardware control. An embedded system manages all physical board operations, including piece detection and movement, while a companion mobile application handles the user interface, manages the authoritative game state, and interfaces with an online chess platform.

The project resulted in a functional prototype that meets the core objectives. The final system tracks local games, facilitates online play against human opponents and AI, and autonomously moves pieces on the board in response to remote moves. This outcome shows the viability of the chosen system architecture. The project also serves as a technology demonstrator, showcasing the integration of mechanical, electronic, and software engineering disciplines required to build a complex, interactive electro-mechanical system.

Declaration of Originality

I hereby declare that the work presented in this thesis is my own unless otherwise stated. To the best of my knowledge the work is original and ideas developed in collaboration with others have been appropriately referenced. Large language models (LLMs) were utilized to improve the quality of the English language in this report.

Copyright Declaration

Acknowledgments

I would like to dedicate this project to my family. Their support from far away has helped me through this whole project. I am sincerely grateful to my father for his generosity in making my education at this institution possible, and for consistently providing me with the best opportunities throughout my life. Thank you to my mother, for guiding my education from the very beginning, for always supporting my interests, and for pushing me to pursue this degree even when I had doubts. To my older sister, thank you for constantly checking in on me and for your care, which made the distance from home feel significantly smaller. And finally, to my younger sister. I hope this work serves as an inspiration and encourages you in your own pursuits.

I would like to thank my supervisor, Dr. Ed Stott. I am truly grateful for your guidance and for the time you dedicated to this project from start to finish. Your support was invaluable.

I also wish to thank Imperial College London for these past four years , providing me with knowledge and skills I could have never previously imagined. In particular, I would like to express my gratitude to the Imperial College Robotics Society and its student volunteers. Their incredible facilities—including 3D printers, laser cutters, and a constant supply of tools and components available seven days a week—were instrumental in the completion of this project. Without their support, a hardware project of this scale would not have been possible.

Finally, a special thanks and shoutout to the DD.

Contents

List of Figures

1 Introduction

Contents

1.1 Introduction

Chess, a timeless game of strategy, has fascinated players for over a thousand years. Originating in 6th-century India as chaturanga, it evolved through Persia and Europe into the modern form we know today [1]. The game's objective is to outmaneuver an opponent by controlling the board, protecting one's pieces, and delivering checkmate—a position where the king is threatened and escape is impossible. Beyond checkmate, chess challenges players to balance strategy, tactics, offense, and defense. Chess has traditionally been limited to two players needing to be physically present to engage in the game. This limitation began to erode with the advent of digital computers in the 20th century, allowing for single-player games against computers. The development of programs like IBM's Deep Blue, which famously defeated world champion Garry Kasparov in 1997, further solidified the integration of technology into chess [2]. These innovations expanded chess's accessibility and popularized the game among millions worldwide. However, despite these advances, many players yearn for the unique experience of a physical chessboard—a quality digital

interfaces struggle to replicate.

The Smart Chess Board project aims combines the feel of a traditional chessboard with the convenience of modern technology. The goal of the system is to accurately digitizes a physical chessboard, offering the advantages of digital chess, such as legal move indicators, single-player mode, automatic game logging, and more. In addition to its practical goals, this project serves as a powerful technology demonstrator for prospective engineering undergraduates. By integrating concepts from fields such as robotics, system-design, embedded systems, and software development. It provides a tangible example of how engineering principles can be applied to create innovative solutions that enhance everyday experiences.

1.2 Requirements

The Smart Chess Board must digitize the game by scanning the full board within 1 seconds — a sampling rate of 1 scan per second. The scanning system must be robust, error-free, and capable of maintaining an accurate game state at all times. Given that the average number of moves in a chess game is 40 [3] and the fastest chess mode is bullet chess, which averages 93.51 seconds per game [4], the fastest moves occur at a rate of approximately 0.43 moves per second. Since $1 \gg 0.43 * 2$, this fulfills the Nyquist sampling rate criterion.

The following must be kept track of :

- **Piece Positions**: The location of every piece on the board, updated after each move.
- **Captured Pieces**: A record of pieces that have been removed from play.
- **Turn Tracker**: Information on whose turn it is (White or Black).
- **Move Log**: A chronological record of all moves made during the game.
- **Castling Rights**: Whether either player can castle and in which direction (kingside or queenside).
- **En Passant Status**: Information about any available en passant capture opportunity.
- **Check Status**: Whether either king is in check.
- **Game Result**: If applicable, the current result (ongoing, checkmate, stalemate, draw by repetition, etc.).

- **Move Timer**: The time remaining for each player .
- **Promotion Tracking**: Status of pawns that have reached the last rank and their promoted pieces.
- **Board State History**: Previous states of the board to detect threefold repetition or for undo functionality.

1.3 Features

1.3.1 Game Modes

The smart chessboard is required to support several distinct modes of play, catering to both solo and multiplayer experiences. The core functionality for each mode is outlined below.

- **Single Player (Player vs. AI):** The user plays a game against an integrated Artificial Intelligence opponent. After the user makes a move on the physical board, the system's will respond and physically move the corresponding piece on the board.
- **Local Multiplayer (Player vs. Player):** Two users play a standard game against each other on the same physical board. The system's role in this mode is to track all moves digitally in real-time. This allows the full game to be saved and exported (e.g., in PGN format) for later review and analysis.
- **Online Multiplayer:** The user plays on their physical board against a human opponent over the internet via an online chess service (e.g., Lichess). The system must transmit the user's physical moves to the online service and, in turn, receive the opponent's moves to be physically actuated on the board. This mode must support playing against both specific friends and randomly matched opponents.
- **Board-to-Board Play:** As an extension of the Online Multiplayer mode, the system must support games where both players are using a compatible smart chessboard. Each player's physical move on their own board will be mirrored on their opponent's board.

1.3.2 Lighting Features

The Smart Chess Board incorporates lighting system beneath the board to enhance the user experience, bringing one of the most popular features of digital chess boards into the physical world.

- **Legal Moves**: When a player selects a piece, the squares that the piece can legally move to are illuminated. This helps the player quickly identify valid moves, making the game more intuitive and reducing the chances of making illegal moves.
- **Best Moves**: In single-player mode against the mechanical system, the lighting system can highlight the best move suggested by the AI. The square of the best move is lit up, allowing the player to learn from the system's strategy and improve their own game play.
- **Capturing a Piece**: In single-player or online mode against the mechanical system, the lighting system highlights the piece that the system intends to capture in red. The player then removes the captured piece, and the system automatically moves the attacking piece to its new position.

2 Background

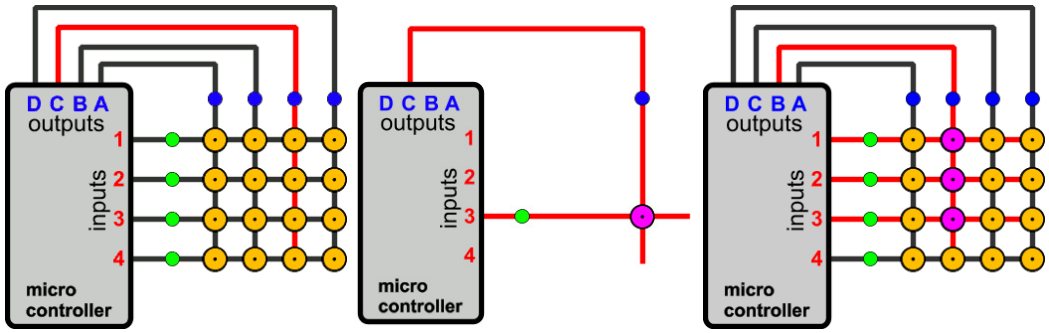
Contents

In this section of the report, we will examine existing systems that incorporate features similar to those planned for the Smart Chess Board. Currently, the Miko Chess - Grand is the only system that implements the two main features I aim to integrate which are tracking and automatic movement of chess pieces. However, it appears that the Miko Chess Board has been discontinued. Many companies claim to offer chess boards that move and track chess pieces, yet most of these products

are currently on a waitlist or available only for pre-order. Therefore, we will also explore systems that independently track chess pieces or move them automatically, examining their technology and implementation to better understand how these systems operate.

2.1 Tracking

2.1.1 Key matrix



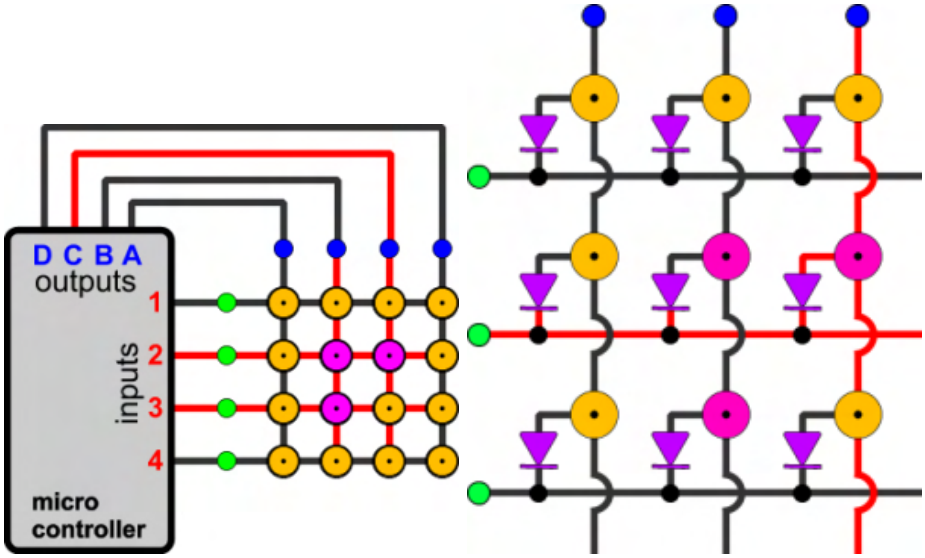
(a) Sweeping of the key matrix (b) C3 key being pressed (c) B1, B2, B3 keys being pressed

Figure 2.1: Visualization of a key matrix operation Source: [5]

This section introduces the concept of a key matrix, which is particularly relevant due to the nature of the project—managing a large number of inputs. In this case, the 64 cells of the chessboard act as individual ”keys” that need to be scanned and tracked.

A key matrix is a technique for organizing and wiring multiple input switches in a grid-like layout. It allows for the detection of many inputs while significantly reducing the number of microcontroller pins required. Instead of assigning a dedicated pin to each input, the switches are arranged at the intersections of rows and columns. By scanning these intersections, the system can determine which specific key (or cell) is active.

Taking Figure 2.2 as an example, the key matrix consists of 4 output pins (A, B, C, D) and 4 input pins (1, 2, 3, 4). There are two key stages in the process. The first stage is the sweep, shown in part (a), where the output pins are sequentially set to high. In the image, output pin C is high, and the input pins are checking for a high signal. Since no keys are pressed, no key is detected. The second stage is the press shown in part (b), key C3 is pressed, which connects column rail C to row rail 3, driving input 3 high. Since the microcontroller knows that only output pin C is high, and input 3 is now high, it deduces that key C3 is pressed. The same process applies if any of the other input pins (1, 2, or 4) are pressed, with the system detecting the specific key by identifying the active output and input pair. This method allows us to read 16 keys using only 8 pins (instead of 16). For a 64-key matrix, it would require just 16 pins, significantly reducing the number of microcontroller pins needed.



(a) Aliasing of the key matrix (b) Aliasing solution

Figure 2.2: Aliasing of a key matrix and diode solution Source: [5]

A common issue that arises in key matrices is aliasing. As shown in Figure 2.2a, in part (a), when rail C is high, keys C2, B2, and B3 are pressed, causing inputs 2 and 3 (via C2 → B2 → B3) to be high. However, although key C3 is not pressed, its rail is high, leading to an aliasing problem where the microcontroller mistakenly detects key C3 as being pressed. A simple and effective solution to this issue is to connect all the keys to the rails through diodes, as shown in part (b) of the figure. This prevents aliasing by ensuring that only the relevant key press is detected.

2.1.2 Miko Chess Board

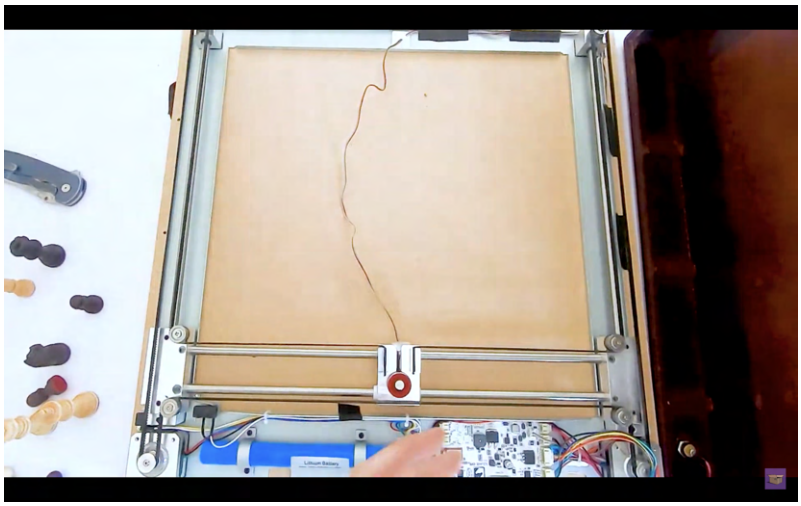
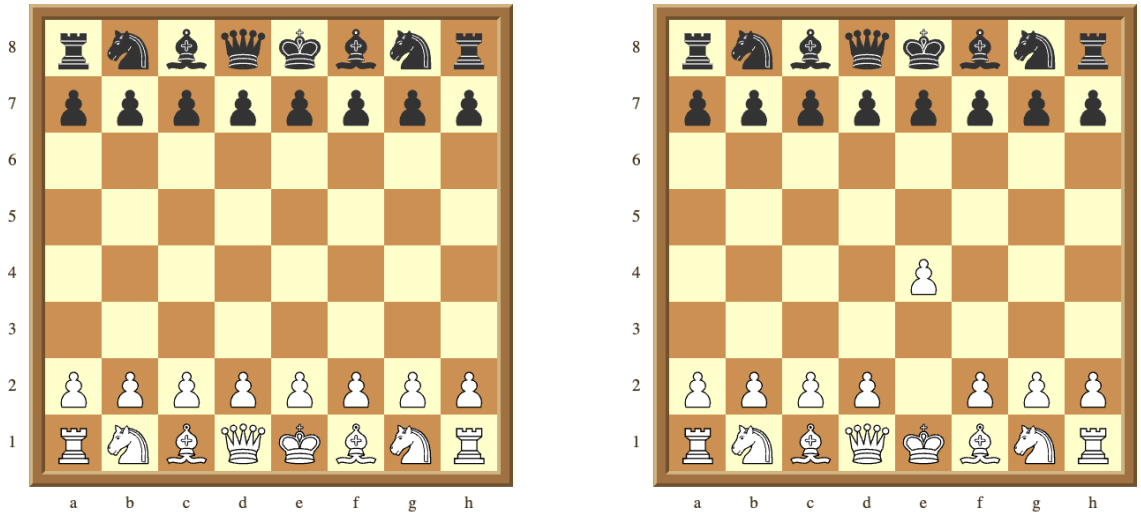


Figure 2.3: Inside of the Miko Chess board. Source: [6]

The Miko Chess Board uses tactile switches arranged in a key matrix beneath the board. When

a player moves a piece, the pressure from the piece activates the corresponding switch, signaling the move. The smart chess board assumes that the user has set up the chess board correctly, relying on the known initial state of the board and the incoming signals from the switches to update the game state. While this method is simple and cost-effective, it has limitations. It does not track pieces uniquely and assumes the board is set up correctly by the player.



(a) Initial state of the chess board.

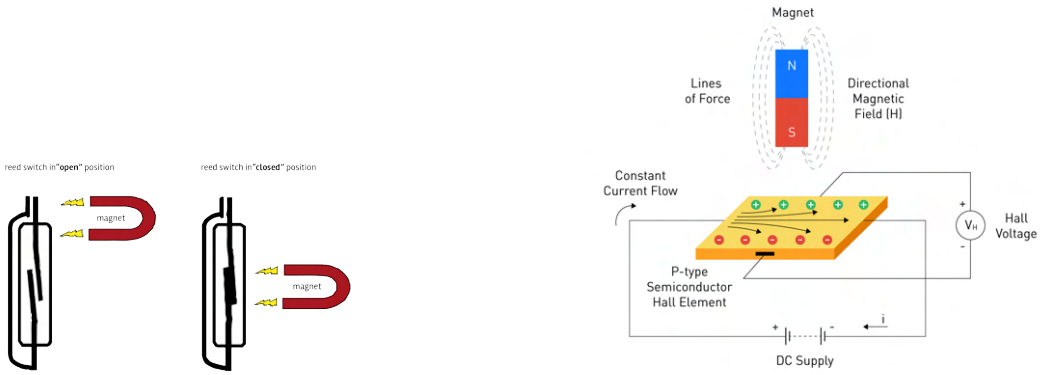
(b) A common opening move: white pawn to e4.

Figure 2.4: Comparison of the initial chessboard state and after an opening move.

Taking Figure 2.4 as an example, the key matrix captures the board's state using a 64x64 2D array, assigning squares with pieces as active (1) and empty squares as inactive (0). When the move "white pawn to e4" is made, the key matrix detects this and update the array to reflect that square e2 is now inactive, while e4 becomes active. By comparing the previous state of the board with the new state, the system can infer that the pawn originally at e2 has moved to e4. This process allows the board to track moves dynamically and update the game state accordingly. The system has no concept of what type of piece is on each square, only detecting the presence or absence of a piece. If a player starts the game with the king and queen swapped, the board would interpret their positions as the correct initial setup. Consequently, the roles of the king and queen would effectively swap, with the system treating the piece in the king's position as the king and the piece in the queen's position as the queen. The Miko Chess Board features a Cartesian stepper motor that moves freely along the 2D axis, with an attached electromagnet used to move the opponent's pieces. The electromagnet positions itself beneath the piece to be moved, activates to hold the piece, glides it into position, and then deactivates. The board does not include a built-in user interface, requiring customers to download the Miko Chess app to play. The app allows users to replay their games, choose a promotion piece during gameplay, and access three playing

modes: Player vs. Board (computer), Player vs. Player (online), and Player vs. Player (offline). For online play, both players must have the Miko Chess board and the app properly set up, it doesn't allow the users to play on popular platforms like Lichess. The schematic of the system is not publicly available, and the details of its design are based on videos of people playing and opening the board.

2.1.3 Interactive Blitz Chess ECE470



(a) Reed Switeches

(b) Hall effect sensor

Figure 2.5: Operation of Reed switches and Hall effect sensors

This project, conducted at Cornell University, presents a solution for tracking chess pieces using a key matrix with reed switches as buttons. Instead of requiring players to physically press the square to indicate a move, players simply move the chess piece, which contains a magnet in its base. This approach is both cost-effective and widely used as having magnetic chess pieces is very common [7] [8]. Additionally, it abstracts the chess tracking system from the user, creating a more seamless experience. Like the Miko Chess Board, this system assumes the board is set up correctly and updates the game state based on the initial setup and incoming signals. Similar projects have been done with hall effect sensors which detects changes in magnetic fields. When a magnet is placed near the sensor, it causes a small voltage to appear in the sensor. The direction of this voltage changes depending on the magnet's polarity. If the north pole of the magnet faces the sensor, the voltage will point one way. If the south pole is facing the sensor, the voltage will point the other way. This change in voltage direction helps the system understand not just that a magnet is present, but also which way the magnet is oriented.

2.1.4 DGT Chess boards

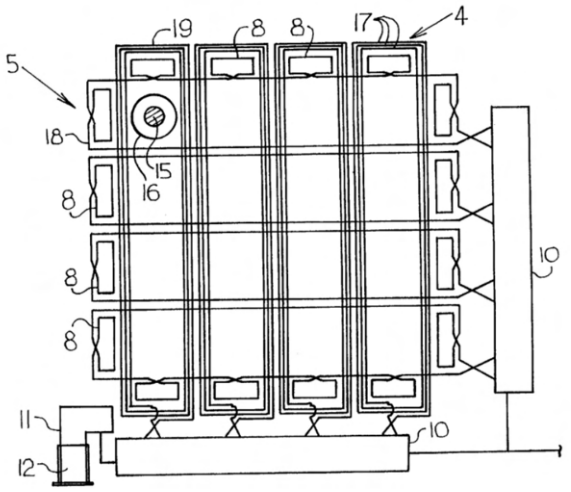
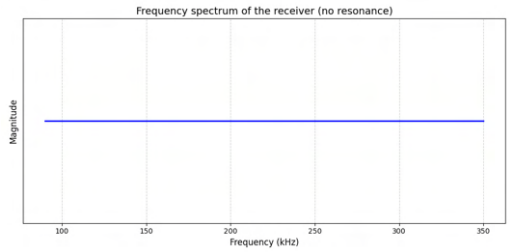


Figure 2.6: Inside of the DGT Chess board. Source: [9]

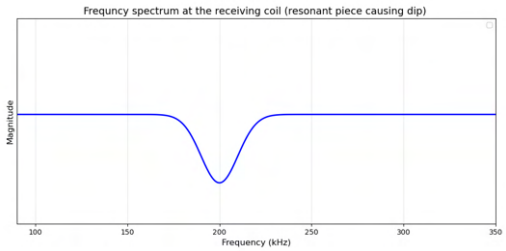
Tracking chess pieces on physical boards has become an essential component of international chess competitions, allowing fans to watch games remotely and view the board more clearly [10]. Additionally, it enables the storage of game logs, replacing the traditional paper-based recording system. The technology is created by DGT [11] . Each piece on the board contains a passive LC circuit, which is tuned to have a specific resonant frequency between 90 kHz and 350 kHz to uniquely identify it. The resonant frequency f of the LC circuit is given by the following equation:

$$f = \frac{1}{2\pi\sqrt{L \cdot C}}$$

where L is the inductance and C is the capacitance of the circuit.



(a) Frequency at receiver with no piece on square (same as transmitter)



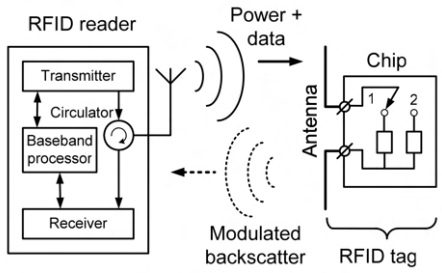
(b) Frequency spectrum with dip at 200KHz due to a piece tuned to that frequency

Figure 2.7: Frequency spectrum at the receiving coil

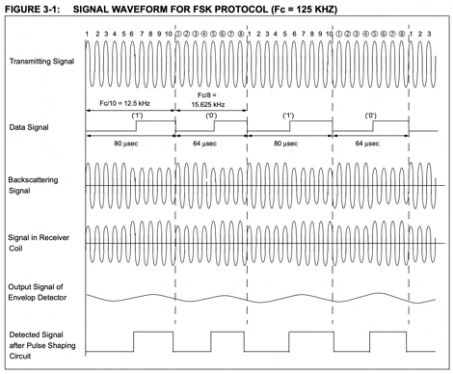
Multiplexers (10) enable the isolation of individual squares on the chessboard. A frequency

sweep ranging from 90 kHz to 350 kHz is transmitted through the transmitting coil (18), while the receiving coil (19) detects a dip caused by the resonance of the piece, which absorbs energy at that magnitude. This dip indicates the presence of a piece on the corresponding square with an example shown in 2.8 of a piece tuned to 200KHz [9]. A drawback of this system is its susceptibility to noise. For example, nearby electronic devices such as smartphones or Wi-Fi routers can emit electromagnetic interference, potentially disrupting the resonance detection. Additionally, the receiver coil, essentially an inductor, exhibits parasitic capacitance, resulting in its own inherent resonant frequency. This self-resonance can distort the signal, complicating accurate detection of the chess pieces.

2.1.5 RFID Solutions



(a) Passive RFID System overview. Source : [12]

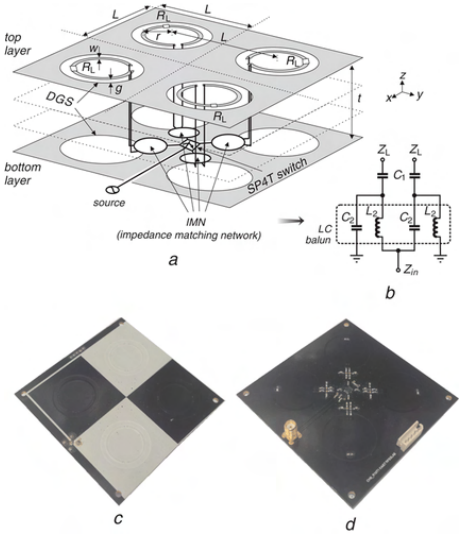


(b) An example of signals involved in RFID communication at 125kHz with frequency shifting key (FSK) modulation. Source : [13]

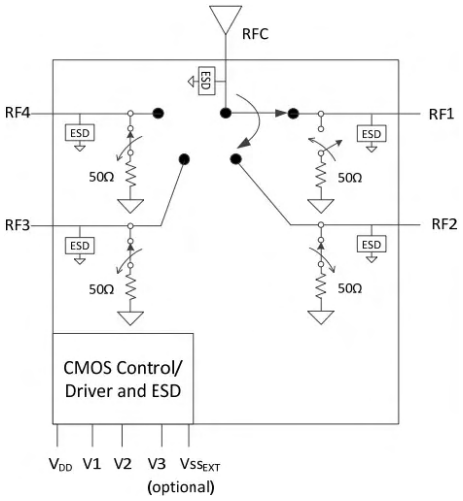
Figure 2.8: Overview of RFID technology

Radio Frequency Identification (RFID) is a wireless method used to identify objects over a short distance without requiring a power supply for the tag. The system consists of a reader and a tag. Taking 2.8b as an example, the reader transmits a radio frequency signal at 125 kHz. This signal induces a current—and consequently a voltage—in the tag's antenna, powering its chip. Once powered, the chip's sole function is to identify itself by sending a series of bits that uniquely represent the tag. It does so using Frequency-Shift Keying (FSK) via backscattered waves modulated at 125 kHz. The reader's antenna detects these backscattered signals, which are then processed through an envelope detector and a pulse-shaping circuit to create a digital signal and recover the tag's unique ID. This technology is widely used in contactless payment systems, where tags embedded in cards or devices allow secure transactions. Its application could extend to chess, where each piece is equipped with a unique RFID tag.

This method of tracking has been successfully implemented using Ultra High Frequency (UHF) at 915 MHz [14]. Researchers connected a single UHF RFID reader to 64 squares of a chessboard by dividing the board into 2 × 2 antenna array modules. The RF signal was routed to the appropriate module using a specialized RF switch 2.9b. This switch includes Electrostatic Discharge (ESD) circuitry to protect its internal components from damage caused by high-frequency signals. Additionally, 50 Ω resistors were used to match the impedance of the signal, ensuring maximum power transfer. The output of the switch is fed into an LC balun circuit, which produces a differential signal across the inductor antenna. In RF systems, differential signals are preferred because they are less susceptible to noise and interference. Any external noise that affects both lines of



(a) Antenna array module for chess board. Source : [14]



(b) SP4T switch in the antenna array.

Figure 2.9: UHRFID chess tracking

the differential pair equally will be rejected, making the differential signal more robust. The LC balun achieves this conversion by using an inductive network that balances the single-ended input (from the switch) into two opposing but equal signals. These two signals are then applied to the antenna in a differential configuration, enabling efficient transmission by the antenna.

2.1.6 Tracking summarised

Method	Advantages	Disadvantages
Miko Chess Board (Switch)	Simple and easy to implement. Detects moves based on pressure from the chess pieces.	Depends on correct board setup. Assumes pieces are placed correctly (e.g., no swapped pieces). Cannot track pieces uniquely. Requires user to physically press down on square after moving piece.
Reed Switches / Hall Effect Sensors	Simple and low-cost solution. Magnetic pieces are widely available. No need for user interaction beyond moving pieces.	Relies on correct board setup and piece placement. No tracking of individual pieces, just the presence of magnets.
DGT Chess Board (LC Circuit)	Uses passive components in pieces. Unique identification of each piece.	Expensive and complicated to recreate. Sensitive to electromagnetic interference. Potential issues with parasitic capacitance in receiver coils.
UHF RFID	Can uniquely track each piece. Works with a variety of piece types and setups. Reliable even if pieces are moved across the board.	Complex setup with antenna arrays and RF switches. High cost and complexity compared to other methods. Sensitive to signal interference and requires precise calibration.

Table 2.1: Comparison of Chess Piece Tracking Methods

2.2 Game State Representation

To digitally manage and record chess games, standardized data representations are essential. The two standard formats are Forsyth-Edwards Notation for individual board positions and Portable Game Notation for the sequence of moves in an entire game.

2.2.1 Forsyth-Edwards Notation (FEN)

FEN is a standard for describing a particular board position using a single line of ASCII text. It acts as a ”snapshot” of the game, capturing all information needed to restart a game from that specific point. To understand its structure, we can dissect the FEN for the standard starting position:

`rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1`

A FEN string is composed of six distinct fields, separated by spaces:

1. **Piece Placement** This describes the position of all pieces, rank by rank, starting from the 8th rank down to the 1st, with each rank separated by a forward slash (‘/‘).
   - White pieces are uppercase (`P, N, B, R, Q, K`); black pieces are lowercase (`p, n, b, r, q, k`).
   - A number from 1 to 8 represents that many consecutive empty squares on a rank.

   In the example, `rnbqkbnr` represents the 8th rank, and each `8` represents an entirely empty rank.

2. **Active Color** A single character indicating whose turn it is to move: `w` for White or `b` for Black.

3. **Castling Availability** Specifies which castling moves are legally possible. `K` and `Q` represent White’s kingside and queenside rights, while `k` and `q` represent Black’s. A hyphen (`-`) is used if no castling moves are possible. In the starting position, `KQkq` indicates all options are available.

4. **En Passant Target Square** Notes a square where an *en passant* capture is possible. If a pawn has just advanced two squares, this field gives the coordinate of the square *behind* that pawn (e.g., `e3`). If there is no en passant opportunity, this field is a hyphen (`-`).

5. **Halfmove Clock** An integer representing the number of plies (half-moves) since the last capture or pawn advance. This is used to enforce the 50-move rule.

6. **Fullmove Number** An integer that starts at 1 and is incremented after Black completes a move.

2.2.2 Portable Game Notation (PGN)

PGN is a plain text format designed for recording the moves of an entire chess game. While FEN captures a static position, PGN records the game's full history, making it the universal standard for storing and sharing games. A PGN file consists of two main parts: header tags and the movetext.

Header Tags This section provides metadata about the game in '[TagName "Value"]' format. While many tags exist, the "Seven Tag Roster" is standard for formal games:

- `Event`: The name of the tournament or match.
- `Site`: The location of the game.
- `Date`: The starting date of the game, in `YYYY.MM.DD` format.
- `Round`: The round number of the game.
- `White`: The name of the player with the white pieces.
- `Black`: The name of the player with the black pieces.
- `Result`: The result of the game (`1-0`, `0-1`, `1/2-1/2`, or `*` for an ongoing game).

Movetext This is the core of the PGN, recording the sequence of moves in **Standard Algebraic Notation (SAN)**.

- **Piece Representation:** SAN uses the same uppercase English letters as FEN to represent the pieces: `K` (King), `Q` (Queen), `R` (Rook), `B` (Bishop), and `N` (Knight). Uniquely, pawns are not identified by a letter.
- **Move Description:** A move is described by the piece's letter followed by its destination square's coordinate. For example, `Nf3` means a knight moves to square `f3`. For a pawn move, only the destination square is given, such as `e4`.
- **Captures and Special Moves:** A capture is denoted by an `x` (e.g., `Bxf7`). When two identical pieces can move to the same square, their starting file or rank is added to disambiguate the move (e.g., `Nbd2`). Special symbols include `+` for check, `#` for checkmate, `O-O` for kingside castling, and `O-O-O` for queenside castling.

- **Game Termination:** The movetext must always end with the game's final result, which matches the Result tag (e.g., `1-0`).

A complete PGN example for a short game is as follows:

```
[Event "Fictitious Game"]
[Site "London, England"]
[Date "2025.05.27"]
[Round "1"]
[White "Player A"]
[Black "Player B"]
[Result "1-0"]

1. e4 e5 2. Bc4 Nc6 3. Qh5 Nf6 4. Qxf7# 1-0
```

2.3 User Interface

Platforms have refined a standard set of features that effectively communicate game information to the user. By analyzing a typical game screen from Chess.com (Figure 2.10), we can identify a set of best practices and key components.

Figure 2.10: An annotated screenshot of the Chess.com user interface.

The essential UI elements, as referenced by the labels in the figure, are as follows:

Label 1: Player Information Panel This area consolidates key player details, including their username, profile picture, rating, and country flag. Below this, it often displays a running tally of pieces captured from the opponent and their corresponding point value. Adjacent to this panel is the player's individual game clock, showing their remaining time.

Label 2: Board Coordinates Standard algebraic notation coordinates are displayed along the edges of the board (files 'a' through 'h' and ranks '1' through '8') to provide clear orientation.

Label 3: Selected Piece Highlight When a player selects a piece to move, its square is highlighted (typically in yellow or green) to provide clear confirmation of the selection.

Label 4: Valid Move Indication All legal destination squares for the selected piece are visually marked, often with a subtle dot or circle, providing immediate feedback on possible moves.

Label 5: Capture Indication To distinguish from a simple move, a legal move that results in a capture is highlighted differently. This is commonly shown with a distinct ring around the destination square.

Labels 6 & 7: Last Move Indication To help players track the game flow, the opponent's most recent move is clearly indicated by highlighting both the origin square (Label 7) and the destination square (Label 6) of the piece that was just moved.

Label 8: Move Log / Game History A scrollable move log displays the game's entire history in algebraic notation, allowing players to review the sequence of moves at any point during the game.

Label 9: Game Action Buttons A set of controls provides access to essential game actions, such as offering a draw, resigning the game, or requesting a takeback.

2.4 Online Chess Services

2.4.1 Online Playing Platforms

These are comprehensive web services that provide the entire ecosystem for playing chess, including user accounts, player matching, game databases, and learning tools.

Lichess.org

Lichess is a free, open-source internet chess server funded entirely by donations. It has become a favorite among developers and serious players for its commitment to open access and its powerful feature set.

- **Capabilities:** Lichess offers a full suite of features, including various time controls, chess variants, puzzles, opening analysis, and user-run tournaments. Crucially for this project, it provides an extensive and well-documented Application Programming Interface (API). The Lichess Board API is specifically designed to allow third-party clients (such as custom websites or physical smart boards) to connect, authenticate, and play rated games on the platform by streaming game events and sending moves.
- **Limitations:** As a non-profit, its user base, while massive, is smaller than its main commercial competitor. From a developer's perspective, however, the limitations are minimal due to the permissive and powerful nature of its open-source API.

Chess.com

Chess.com is the largest commercial chess platform in the world, boasting the largest user base. It operates on a freemium model, where basic play is free, but many advanced learning features and analytics are behind a subscription paywall.

- **Capabilities:** It offers a vast array of content, including professional lessons, videos, news articles, and a massive community. It also has a public API, which is excellent for retrieving data such as player profiles, game archives, and tournament results.
- **Limitations:** The primary limitation for a project like this is that the Chess.com API is primarily designed for data retrieval, not for real-time gameplay integration by third-party

clients. It does not offer the same level of support for creating an external interface to play live games on its server as Lichess does, making it a less suitable choice for this project's core online play feature.

2.4.2 Chess Engines

A chess engine is fundamentally different from a playing platform. It is a pure calculation program that, given a board position, determines the best possible move. Engines do not have user interfaces, player accounts, or matchmaking capabilities on their own.

Stockfish

Stockfish is an open-source chess engine that is widely regarded as the strongest traditional engine in the world. It serves as the foundation for the analysis tools on many platforms, including Lichess.

- **Capabilities:** Stockfish's sole function is to analyze chess positions with incredible depth and speed.
- **Limitations:** The critical limitation of these Stockfish APIs is that they do not manage a persistent game. They cannot be used to "run" a continuous game, track move history, handle turns, or validate a sequence of moves. Each API call is an independent, one-off analysis of a single position. Therefore, while a Stockfish API is excellent for creating a feature like a "puzzle solver" or getting a hint for a specific position, it is not a suitable backend for facilitating a complete, interactive game session.

2.5 Lighting Systems

2.5.1 Classic LED Matrix (Row/Column Scanning)

This is a traditional and cost-effective method for controlling a large number of LEDs arranged in a grid.

- **How it Works:** The LEDs are wired in a matrix of rows and columns. For example, in an 8x8 grid, all the anodes of the LEDs in a row are connected together, and all the cathodes in a column are connected together. To light a specific LED at coordinate (row, column), the microcontroller applies power to that specific row and grounds that specific column. To give the illusion of multiple LEDs being lit simultaneously, the microcontroller rapidly scans through all the rows and column.
- **Capabilities:** It can light up any combination of LEDs in the grid using a relatively simple wiring scheme.
- **Limitations:**
  - **Brightness Issues:** Since each LED is only turned on for a fraction of the time (e.g., 1/8th of the time for an 8x8 matrix), the perceived brightness is significantly lower than if it were continuously powered.
  - **Pin Count:** This method requires a significant number of microcontroller I/O pins (e.g., 8 for rows + 8 for columns = 16 pins for a basic 8x8 matrix), which can be a major constraint.
  - **Single Color:** A standard matrix is typically limited to a single color. Implementing full RGB color would require three separate matrices and 3x the control logic, making it highly complex.
  - **Constant Refresh Required:** The microcontroller must dedicate processing time to constantly refresh the display, which can interfere with other time-sensitive tasks.

2.5.2 Standard LEDs with Shift Registers

This method uses integrated circuits (ICs) called shift registers to expand the number of output pins available from a microcontroller.

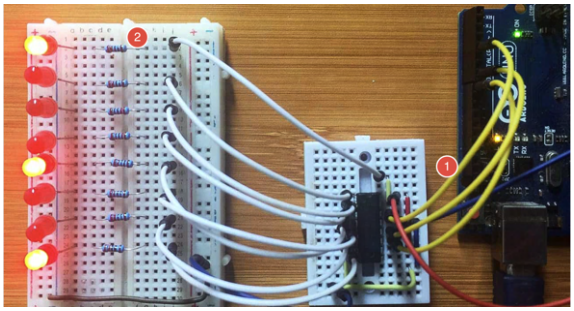


Figure 2.11: An illustration of a shift register expanding a microcontroller's outputs. Three control pins send serial data to the register, which converts it to drive eight parallel LEDs.

- **How it Works:** A shift register takes a serial data stream from the microcontroller using just a few pins (data, clock, and latch), as illustrated in Figure 2.11. It then converts this serial data into a parallel output (typically 8 bits), allowing the microcontroller to control 8 separate LEDs. Multiple shift registers can be daisy-chained together, so with the same three pins, one could control 16, 24, or all 64 LEDs.
- **Capabilities:** It dramatically reduces the number of microcontroller pins needed to control many individual LEDs. Unlike a matrix, a shift register holds its state, so the LEDs remain on or off without needing a constant refresh from the microcontroller.
- **Limitations:** This method is most practical for single-color LEDs. To achieve full RGB control for 64 squares, one would need 3 outputs per square, requiring a total of $64 \times 3 = 192$ outputs. This would necessitate a complex chain of 24 shift registers, which increases board complexity and wiring.

2.5.3 Individually Addressable RGB LEDs (e.g., NeoPixels/WS2812B)

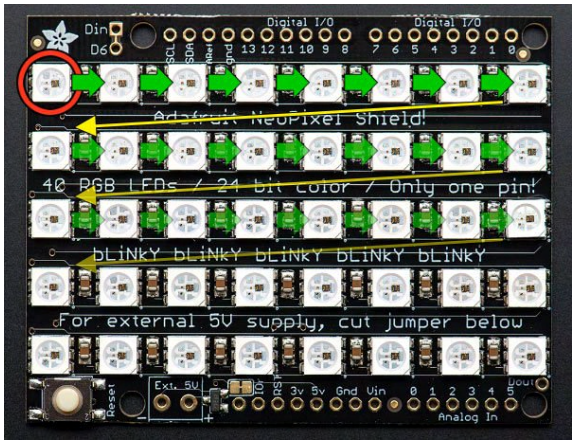


Figure 2.12: Neopixel lights data flow source : [15]

This modern technology integrates a small microcontroller into each LED package, allowing for sophisticated control with minimal wiring.

- **How it Works:** All the LEDs are connected in a single daisy-chain. The microcontroller sends a precisely timed serial data stream down a single data pin. This stream contains the color information for every LED in the chain. The first LED in the chain reads the first packet of color data (typically 24 bits for RGB), sets its own color accordingly, and then re-transmits the rest of the data stream to the next LED. This process repeats down the entire chain.
- **Capabilities:**
  - **Full, Independent Color Control:** Every single LED can be set to any one of millions of colors, completely independent of all other LEDs.
  - **Minimal Wiring:** The entire grid of 64 LEDs can be controlled with just one data pin from the microcontroller.
  - **High Brightness:** Each LED is powered continuously, resulting in high brightness.
- **Limitations:**
  - **Power Consumption:** A chain of 64 RGB LEDs can draw a significant amount of current (several amps if all are on full brightness white), necessitating a separate, robust power supply.
  - **Cost:** Per-unit cost is generally higher than for standard LEDs.

2.6 Movement Systems

2

In this section, we explore different mechanical configurations of XY robots designed to move a magnet or electromagnet underneath a chessboard. This mechanism would allow the system to physically latch onto chess pieces from below and move them to new positions, effectively playing out the moves of an online opponent or an AI. Several designs common in CNC machines, 3D printers, and plotters are suitable for this planar X-Y motion.

2.6.1 Cartesian CNC

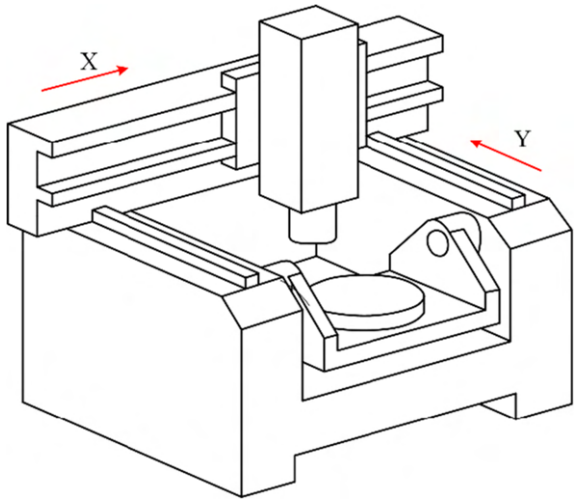


Figure 2.13: CNC Machine kinematics

Kinematics

Each axis is driven independently by a lead- or ball-screw. The mapping from screw displacement ΔL_i to linear travel along axis i is direct:

$$\Delta X = \Delta L_X, \quad \Delta Y = \Delta L_Y$$

Pros

- **Rigidity:** High stiffness under cutting loads, ideal for precise positioning.
- **Precision:** Very accurate positioning, within ±0.01,mm.
- **Load Capacity:** Handles heavy end-effectors or cutting tools without frame flex.

Cons

- **Cost:** Ballscrews and rigid frames drive up cost.
- **Footprint:** Large space required for stable mounting and clearance.
- **Speed Limitation:** Screw critical speed limits safe rapid moves (usually $\leq 200\,\text{mm/s}$).

2.6.2 H-Bot

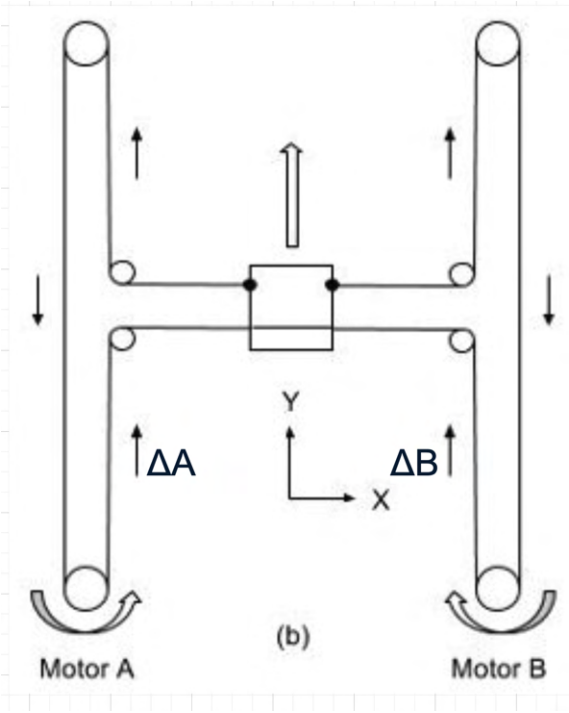


Figure 2.14: H-Bot Kinematics

Kinematics

A single continuous belt forms an "H" shape. Motors A and B drive the belt ends. The linear displacements $\Delta X, \Delta Y$ relate to belt motions $\Delta L_A, \Delta L_B$ by:

$$\begin{bmatrix} \Delta X \\ \Delta Y \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \Delta L_A \\ \Delta L_B \end{bmatrix}.$$

2

Pros

- **Simplicity:** Single belt loop, minimal idlers and pulleys.
- **Cost-Effective:** Lower cost requires 3 linear rails with carriages along with belt.
- **Lightweight and space efficient:** Reduced moving mass enables higher accelerations.

Cons

- **Twist Sensitivity:** Belt stretch or unequal tension induces torsion/skew.

2.6.3 CoreXY

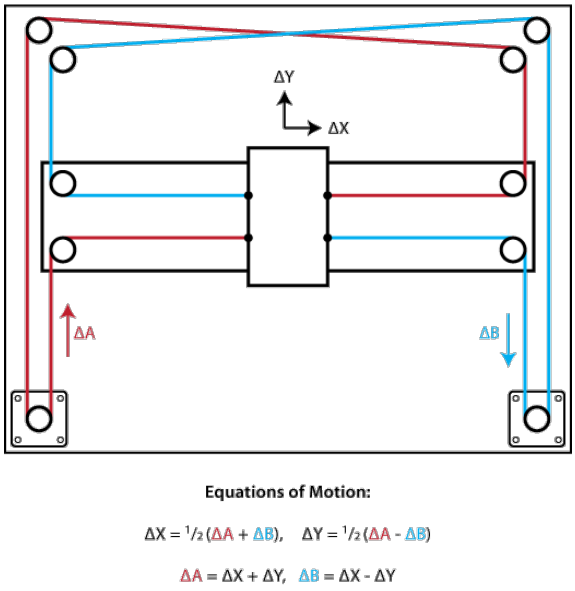


Figure 2.15: CoreXY Kinematics

Kinematics

Two independent belts cross between motors. Each motor affects both axes:

$$\begin{aligned} \Delta L_A &= \Delta X + \Delta Y, \\ \Delta L_B &= \Delta X - \Delta Y, \end{aligned} \quad \Longrightarrow \quad \begin{bmatrix} \Delta X \\ \Delta Y \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \Delta L_A \\ \Delta L_B \end{bmatrix}.$$

Pros

- **Balanced Forces:** Torque vectors cancel, reducing frame bending.
- **Speed & Dynamics:** Can achieve $\geq 500\,\text{mm/s}$ with good acceleration.

Cons

- **Belt Routing:** More idlers and careful tensioning required.
- **Coupling Errors:** Unequal belt tension can induce slight axis coupling.
- **Cost:** Slightly higher cost versus H-Bot (extra pulleys, belts).

2.6.4 G-code for CNC Control

G-code is the most widely used programming language for controlling Computer Numerical Control (CNC) machines, including 3D printers, mills, lathes, and the custom-built movement system in this project. It is a plain text language that provides line-by-line instructions to the machine's controller, telling it how to move, what speed to use, and how to operate its tools.

Structure of G-code Commands

Taking the following as an example `G1 X50.5 Y100.0 F3000`

- **G1 (Preparatory Command):** This word sets the machine's mode to **Linear Interpolation**. It instructs the controller to move the tool (the magnet) in a straight line from its current position to the specified target coordinates at a controlled speed.
- **X50.5 (X-Axis Coordinate):** This is a parameter that defines the target coordinate for the **X-axis**. The machine will move to the position 50.5 millimeters (or the configured unit) along its X-axis.
- **Y100.0 (Y-Axis Coordinate):** This defines the target coordinate for the **Y-axis**. The controller will move both the X and Y axes simultaneously to create a straight-line path to the point (50.5, 100.0).
- **F3000 (Feed Rate):** This parameter sets the **speed** at which the machine will execute the move. The 'F' stands for Feed Rate, and the value is typically specified in units per minute (e.g., 3000 mm/minute).

3 Analysis & Design

Contents

3.1 System Design

3.1.1 System Architecture

The architecture of the smart chessboard system integrates embedded hardware for board interaction, a microcontroller for processing and communication, and a mobile application for user interface and online connectivity. Figure 3.1 provides a high-level overview of these interconnected components.

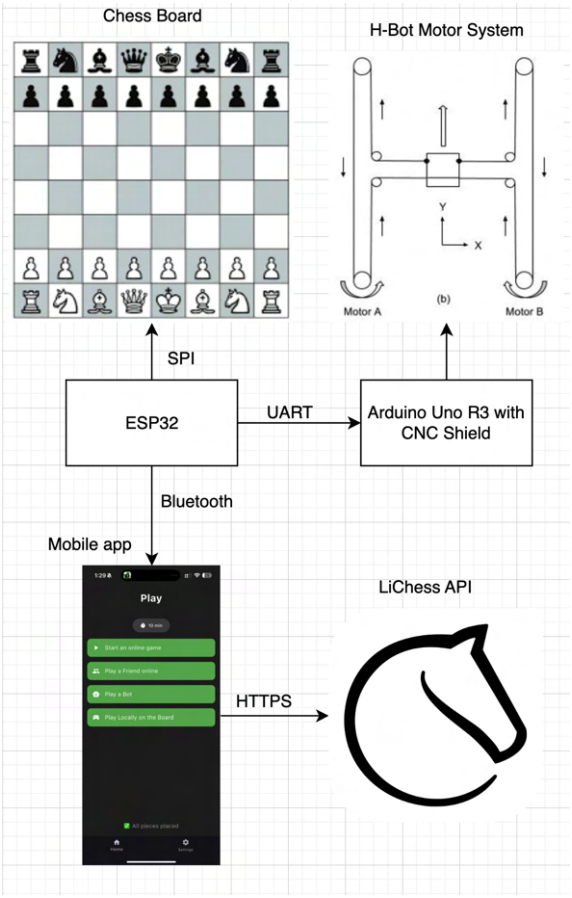


Figure 3.1: High-Level System Architecture Diagram illustrating the key components and their communication pathways.

The core components and their interactions are as follows:

- **Chessboard and ESP32:** The physical chessboard, equipped with RFID readers for piece detection, interfaces directly with the **ESP32** microcontroller. This connection is primarily managed via the **Serial Peripheral Interface (SPI)** bus, allowing for fast data exchange related to the physical state of the board.
- **ESP32 and Motor Control (Arduino Uno R3 with CNC Shield):** For physical piece movement, the **ESP32** communicates with an **Arduino Uno R3** microcontroller, which is augmented with a **CNC shield**. This communication is handled via **UART (Universal Asynchronous Receiver-Transmitter)**. Instead of abstract messages, the ESP32 is responsible for translating a desired move (e.g., from square ‘e2‘ to ‘e4‘) into a sequence of standard **G-code commands**. For example, it sends commands like `$H` to home the machine or `G1 X50.0 Y100.0 F3000` to move the magnet to specific coordinates at a set speed. The Arduino, running GRBL firmware, simply executes these incoming G-code commands, using

the CNC shield to generate the precise step and direction signals for the motors.

- **ESP32 and Mobile Application:** The **ESP32** also establishes a wireless link with a companion **mobile application** using **Bluetooth Low Energy (BLE)**. This channel is used for transmitting game state information, proposed moves detected on the physical board, and receiving commands or acknowledgments from the app.
- **Mobile Application and Lichess API:** The mobile application is responsible for all external network communication. It interacts with the **Lichess API** using **HTTPS** requests to facilitate online gameplay, fetch game data, submit moves, and handle user authentication.

This layered architecture allows each component to specialize in its designated tasks: the chessboard hardware captures physical interactions, the **ESP32** acts as the central embedded processor and communication hub, the Arduino/CNC shield handles motor control, and the mobile application provides the user interface and manages online interactions.

3.1.2 System State

The board operates in one of three primary states, each defining its behavior at different stages of use. As shown in Figure 3.2.

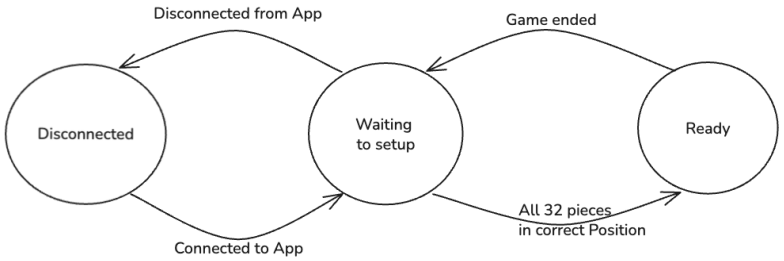


Figure 3.2: A state machine diagram illustrating the three main operational states of the smart chessboard.

Disconnected State This is the initial or default state of the board. In this state, the system's functions are minimal; it will do nothing other than advertise its presence via Bluetooth, awaiting a connection from the companion mobile application.

Waiting for Setup State The system transitions to this state immediately after a successful Bluetooth connection is established. In this phase, the board's primary function is to guide and validate the initial placement of the 32 chess pieces according to the rules of chess. No game event processing will occur.

Ready State The system enters this state only after it has confirmed that all 32 pieces have been correctly placed on the board. This is the main operational state for gameplay. In the Ready State, the board will actively scan for piece movements, stream these events to the companion application, and receive and execute commands for lighting and, if applicable, for physical piece movement.

3.2 Design decisions

3.2.1 Choice of Microcontrollers

The project's architecture leverages two distinct microcontrollers, each selected for its specific strengths and suitability for different aspects of the system: the **Arduino Nano ESP32** for main processing and communication, and an **Arduino Uno R3** paired with a **CNC shield** for dedicated motor control.

Arduino Nano ESP32

The **Arduino Nano ESP32** was chosen as the primary microcontroller for several key reasons:

- **Wireless Capabilities:** It features integrated **Wi-Fi** and **Bluetooth Low Energy**, which are essential for communication with the companion mobile application.
- **Processing Power:** The ESP32 is able manage the array of RFID readers, handle the real-time detection of piece movements, run the Bluetooth stack, and coordinate with the secondary microcontroller for motor actions.
- **Peripheral Interfaces:** It offers a rich set of peripherals, including SPI for interfacing with the RFID readers and UART for communication with the motor control subsystem.
- **Role:** In this system, the Nano ESP32 acts as the central brain for the physical board, responsible for reading sensor data, processing board events, and relaying information to and from the mobile app and the motor controller.

Arduino Uno R3 with CNC Shield

For the task of physically moving the chess pieces (via an under-board magnet system), an **Arduino Uno R3** in conjunction with a **CNC (Computer Numerical Control) shield** was selected.

- **Established Solution for Motor Control:** The Arduino Uno, when paired with a CNC shield (often running GRBL firmware or similar), is a widely adopted and well-documented solution for controlling stepper motors in 3D printers, small CNC mills, and other robotic systems. This familiarity and extensive community support make it a reliable choice.

- **Dedicated Tasking:** Offloading motor control to a separate microcontroller allows the ESP32 to focus on its primary tasks without being burdened by the precise timing and step generation required for smooth motor operation.
- **CNC Shield Functionality:** The CNC shield provides a convenient interface for connecting stepper motor drivers (like A4988 or DRV8825 modules), power inputs, and potentially limit switches.
- **Role:** The Arduino Uno R3 receives high-level movement commands from the ESP32 (e.g., ”move magnet to square e4”) via a UART connection. It then translates these commands into the necessary step and direction signals for the motors connected to the CNC shield.

3.2.2 Communication between ESP32 and App

Several wireless communication methods were considered for establishing a link between the ESP32 and the companion mobile application. These alternatives included:

- **Wi-Fi:** This would typically involve both the ESP32 and the mobile app connecting to the same local area network (LAN). Messages could then be exchanged directly using socket programming (e.g., TCP/IP or UDP streams between the devices' IP addresses) or, if a web server component was part of the architecture, via HTTP/HTTPS requests.
- **MQTT (Message Queuing Telemetry Transport):** This is a lightweight publish-subscribe messaging protocol. In this model, both the ESP32 and the app would connect to a central MQTT broker (server). The ESP32 would publish messages (like board events) to specific "topics," and the app would subscribe to these topics to receive them, and vice-versa for commands to the ESP32.

Given the operational context where the ESP32 and the mobile app are typically in close proximity, and with a high priority placed on responsive, real-time interaction, **Bluetooth Low Energy (BLE)** was ultimately selected as the most suitable technology.

BLE offers several advantages for this application:

- **Low Latency:** For the small, frequent data packets involved in this project (e.g., sending move coordinates like `"e2e4"`, hover events like `"hover:e2"`, or acknowledgments like `"ACK"`), BLE provides low-latency communication. This is crucial for ensuring that the physical board interactions and the app's UI feel instantaneous and synchronized, contributing to a "fast" user experience.
- **Sufficient Throughput for Application Needs:** While Wi-Fi can offer higher raw data throughput, BLE's data rates (up to 2 Mbps at the physical layer for BLE 5) [16] are more than adequate for the concise messages exchanged in this system. The "speed" prioritized here refers to responsiveness rather than bulk data transfer.
- **Direct Connection and Simplicity:** BLE allows for a direct peer-to-peer connection between the ESP32 and the mobile app without the need for intermediary network infrastructure like Wi-Fi routers or MQTT brokers, simplifying the setup and connection process for the end-user.

The communication over BLE is structured using the Generic Attribute Profile (GATT). In this model, the ESP32 acts as the GATT server, defining specific services and characteristics. These characteristics are then used by the mobile app (the GATT client) to write commands to the ESP32 (e.g., lighting instructions) and to receive notifications or indications from the ESP32 (e.g., detected moves or board events).

3.2.3 Storing game state

Initially, the simplest way to store the chess state was to use an external chess engine library and run it on the ESP32. However, a review of available libraries showed that most are designed for computers and are too large to run on a microcontroller. The reason is that they are often built around highly efficient but complex data structures, such as bitboards.

A **bitboard** uses a 64-bit integer where each bit maps to a square, allowing for extremely fast move generation using bitwise CPU operations. While the bitboard data structure itself is memory-efficient, the libraries are large because they include resource-intensive features for playing AI opponents, such as:

- Complex search algorithms (e.g., alpha-beta pruning).
- Large, pre-computed lookup tables for move generation.

This surrounding infrastructure, not just the board representation, makes a full engine too heavy for the ESP32.

This presented a significant design challenge. A basic FEN/PGN parser could track the board state, but the project required more complex, dynamic logic. For instance, every move made by a player needs to be **validated** to ensure it is legal according to the rules of chess. Furthermore, a key feature is the lighting system, which must illuminate all possible legal moves for a selected piece.

An attempt to implement the logic natively on the ESP32 was made. While this proved to be an interesting and educational "sidequest," it became apparent that it was an inefficient use of time. The effort to create a bug-free move generator from scratch felt like attempting to reinvent a solution that has already been perfected in numerous existing chess libraries. The time required to correctly handle all of chess's complex edge cases—such as en passant, castling restrictions, and illegal moves due to pins—was better spent on the project's unique hardware challenge. Therefore, the final and most robust architectural solution was to **offload all chess logic to a companion application**. In this model, the ESP32's sole responsibility is to act as a hardware controller, communicating board events via Bluetooth to an app on a host device like a smartphone. This approach leverages the power of a proven, full-featured chess engine running on the host, while keeping the ESP32's firmware simple, responsive, and focused on its primary tasks of scanning the board and controlling the lights.

3.2.4 Online Interfacing

For enabling online gameplay, **Lichess.com** was selected as the platform of choice. Lichess is an open-source, free-to-play internet chess server with a large global user base and a comprehensive Application Programming Interface (API).

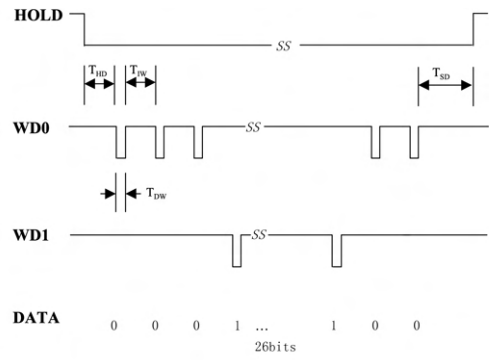
The decision to integrate with Lichess, rather than developing a proprietary online platform, was based on several key considerations:

- **Extensive and Open API:** The Lichess API is well-documented and provides all the necessary endpoints for third-party applications to manage user authentication, create and accept challenges, send and receive moves for online games, and stream game events. This allows for deep integration of Lichess functionality directly into the custom application and physical board.
- **Established User Base:** Utilizing Lichess grants users immediate access to a vast and active community of chess players. This significantly enhances the online play experience, as users can easily find opponents or play with existing friends who are already on the platform.
- **Reduced Onboarding Friction:** By leveraging Lichess, users can use their existing accounts. Developing a custom platform would necessitate a separate registration process for both the user and anyone they wish to play against, creating a significant barrier to adoption.
- **Broader Opponent Pool:** A proprietary system would limit online play to only other users of that specific service. In contrast, Lichess offers a much larger and more diverse pool of potential opponents.
- **Focus on Core Project Goals:** Integrating with an established platform like Lichess allows development efforts to be focused on the unique aspects of the smart chessboard and its interface, rather than on the monumental task of building, maintaining, and popularizing a new online chess service.

3.2.5 Tracking pieces

RFID technology was selected for tracking chess pieces primarily due to its ability to uniquely identify each piece through its embedded tag and associated UID. This allows the system to accurately verify the correct initial board setup by ensuring each unique ID is assigned to the correct piece type and starting square, a crucial step for game integrity.

125kHz



Symbol	Specification	Representative value
T_{HD}	Send data to active time extension	2ms
T_{SD}	Send data to finish time extension	2ms
T_{DW}	Data impulse width	80 μ s
T_{IW}	Data impulse interval width	1ms

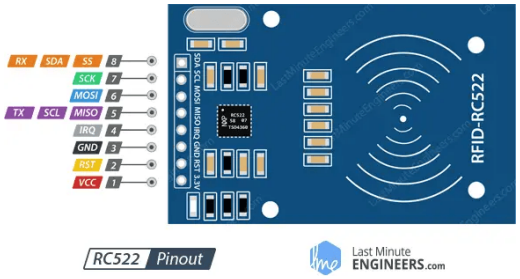
Figure 3.3: Weigand data protocol for RDM6300. Source: [17]

125 kHz is the standard low frequency for RFID chips. Since it operates at such a low frequency, handling it is much simpler—for example, reflections in the circuit are minimal, meaning impedance matching is not a concern. Cards that support 125 kHz can only store the unique ID of the card and nothing else. They do not have storage capabilities and cannot be rewritten. I began running preliminary tests with the RDM6000, a common RFID reader that supports 125 kHz. Its wiring was straightforward, and it used UART communication with the ESP-32. The reader utilizes the Wiegand data protocol, which consists of 26 bits: the first and last bits are parity bits, followed by 8 bits for the facility code, and 16 bits for the unique ID. During initial testing, I observed that the scan time took 40 ms, which was too slow. Upon further investigation, I found that the Wiegand data protocol requires three lines: WD0 for sending 0s, WD1 for sending 1s, and a HOLD line for data transmission. It takes 2 ms to initiate the protocol, 1 ms per bit to transmit the 26 bits

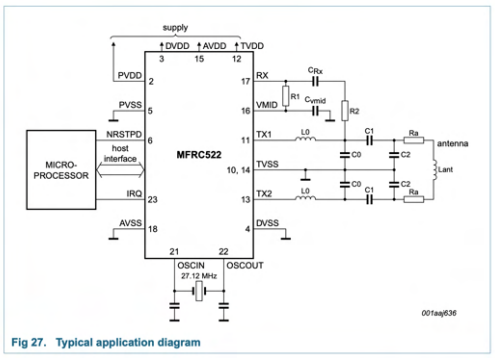
(totaling 26 ms), and another 2 ms to end the transmission, resulting in a best-case total of 30 ms. For a chessboard with 64 squares, this means scanning all squares would take 1.92 seconds in the best-case scenario, which is far too slow. As a result, an alternative solution is needed.

3

13.56 MHz - Final Choice



(a) RC522 pin layout Source : [18]



(b) Schematic of the RC522 connected to the MFRC522 IC Source : [19]

Figure 3.4: Overview of the RC522, a common RFID reader

Since 125 kHz was deemed too slow, higher RFID frequencies needed to be explored. The next option, 13.56 MHz, uses the MIFARE Classic 1K communication protocol, which offers additional features such as anti-collision mechanisms and encryption. Tags supporting this frequency can store more than just a unique ID; they allow up to 1 KB of data, divided into 16 sectors, each containing 4 blocks. Of these, 3 blocks are allocated for user data, while 1 block is reserved for the sector key to control access [19]. However, operating at a higher frequency introduces new challenges. The circuit becomes more susceptible to signal reflectance, requiring impedance matching to minimize signal loss. Additionally, EMC (Electromagnetic Compatibility) filters are necessary to ensure the system meets regulatory standards and operates efficiently - these can be seen in 4.1b with the circuit coming out of TX1 , TVSS and TX2 . Furthermore, the advanced features of the 13.56 MHz communication protocol add complexity to the interaction between the sender and receiver. Unlike the simpler process of measuring backscattered waves and demodulating them at lower frequencies, this protocol involves more sophisticated data exchange and processing.

The RC522 module supports both I^2C and SPI communication between the board and the microcontroller. Given that speed is a priority, SPI was chosen as it supports data transfer rates of up to 10 Mbps, compared to I^2C, which is limited to 5 Mbps in ultra-fast mode [20]. This difference is due to SPI being a full-duplex communication protocol, allowing simultaneous data transmission

and reception, whereas I^2C is half-duplex. While SPI requires more wiring, the benefit of faster communication justifies the trade-off. Of course. Here are the two brief subsections detailing the design choices for the lighting and movement systems, formatted in LaTeX.

3.2.6 Lighting System

For the board's lighting system, Individually Addressable RGB LEDs (NeoPixel WS2812B) were selected as the final design choice. While other methods like LED matrices were considered, the NeoPixel system offers unparalleled advantages for this application. The ability to control the precise color and brightness of all 64 LEDs independently using a single data pin from the microcontroller is a critical feature. This allows for a rich and intuitive user experience—such as indicating valid moves, captures, and checks with different colors simultaneously—which would be prohibitively complex to implement with a traditional LED matrix or a large array of shift registers.

3.2.7 Magnets

Electromagnets

Initially, electromagnets were considered the ideal choice due to their primary advantage: a controllable magnetic field. The ability to turn the grip on and off via software would, in theory, solve the piece release problem with elegant simplicity.

However, this approach was found to be impractical due to a significant limitation: **limited effective range**. During analysis and prototyping, it was determined that an electromagnet of a reasonable size and power consumption could not produce a magnetic field strong enough to reliably grip and move the pieces through the full thickness of the chessboard material. Overcoming this would require a much larger and more powerful electromagnet, which would in turn introduce prohibitive new problems, including:

- **Excessive Heat Generation:** A more powerful coil would generate significant heat, risking damage to the gantry mechanism and the board itself.
- **High Power Consumption:** The power requirements would complicate the system's power supply design.
- **Physical Size:** A larger electromagnet would be more difficult to integrate into a compact under-board gantry.

Due to these strength and range limitations, electromagnets were deemed unsuitable for this project.

Permanent Magnets (Neodymium)

In contrast, Neodymium (NdFeB) permanent magnets were analyzed as the alternative.

- **Superior Magnetic Strength and Range:** The key advantage of Neodymium magnets is their exceptionally high magnetic field strength for a given size. A small Neodymium magnet provides more than enough force to securely grip the chess pieces through the board, ensuring reliable movement without risk of slippage.
- **The Release Challenge:** The obvious drawback of a permanent magnet is that it is "always on," which introduces the challenge of releasing the piece. Unlike the fundamental strength

issue of the electromagnet, this was identified as a solvable **mechanical engineering problem**.

4 Implementation

Contents

4.1 RFID Piece Detection

In this section, we explore methods for tracking chess pieces using RFID technology, drawing inspiration from the key matrix. The goal of this system is to power each square of the chessboard individually using a reader module or antenna and then read the corresponding values. Since the microcontroller knows which square has been activated and what values are being read, the system can map out the chessboard. This process cannot be parallelized because, while it is possible to read all the values that identify the chess pieces simultaneously, the system would lack the context to determine which square each value corresponds to. Therefore, sequential activation and reading of each square are necessary for mapping.

4.1.1 Using the RC522 Module and reducing scan time

Despite its capabilities, the RC522 module is complex to work with directly. To simplify development, the MFRC522 library was used. To illustrate the module's complexity, the following steps outline how to read the ID from a card:

1. **Reset baud rates**: Configure the transmission and reception rates by writing to the `TxModeReg` and `RxModeReg`.
2. **Set modulation width**: Adjust the transmitted data's modulation width via the `ModWidthReg`.
3. **Clear collision settings**: Reset potential collision detection by modifying the `CollReg` bitmask.
4. **Stop active commands**: Set the `CommandReg` to idle mode to stop any ongoing operations.
5. **Clear interrupt requests**: Reset the seven interrupt request bits in the `ComIrqReg`.
6. **Flush the FIFO buffer**: Clear the buffer using the `FIFOLevelReg`.
7. **Send request command**: Write the *Request* command to `CommandReg`, initiating a 36 ms timer. During this time, the reader attempts to detect a card and updates the `ComIrqReg`. If no card is detected, the module sends a `STATUS_TIMEOUT` signal.

During testing, the scanning time with the card present took approximately 5–7 ms, while the scanning time with no card present was up to 37 ms. This delay was caused by the timer set in the

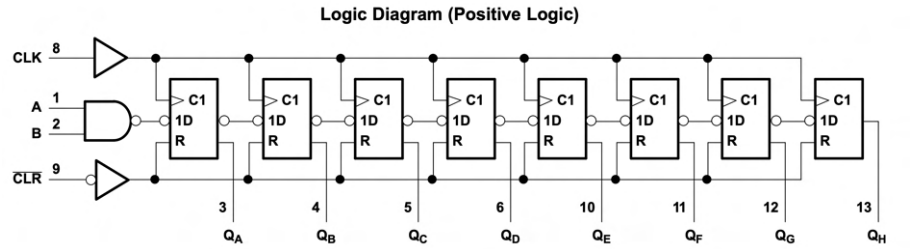
MFRC522 library (as shown on line 7), where the reader was waiting for a timeout when no card was detected. After further testing, this timeout period was reduced to 10 ms, which resulted in the worst-case scenario for scan time being 11 ms. This modification successfully met the target of a 1-second scan time goal.

After discovering the short scan time, the next step was to wire multiple RC522 modules and test whether they could switch between readers to scan at different positions. The SPI protocol uses a Slave Select (SS) pin that is active low. When a low signal is sent to it, it activates the selected slave. Initially, this approach was tested, but it failed to read any cards. Upon investigating with an oscilloscope, I discovered that the MFRC522 device does not set the MISO (Master In Slave Out) output pin to high impedance when powered and not selected. As a result, the MISO pin ties the line to ground, even when the chip select is inactive. This behavior necessitated finding an alternative method. After consulting the RC522 and MFRC522 datasheets, I found that the RST (reset) pin could be used for this purpose. When the RST pin is low, the device powers down and disables the antenna - which will reduce interference if the devices are close together. By sending a high signal to the RST pin, the device powers back up and resets, clearing any issues from an unexpected shutdown. This method successfully allowed switching between multiple readers without interference.

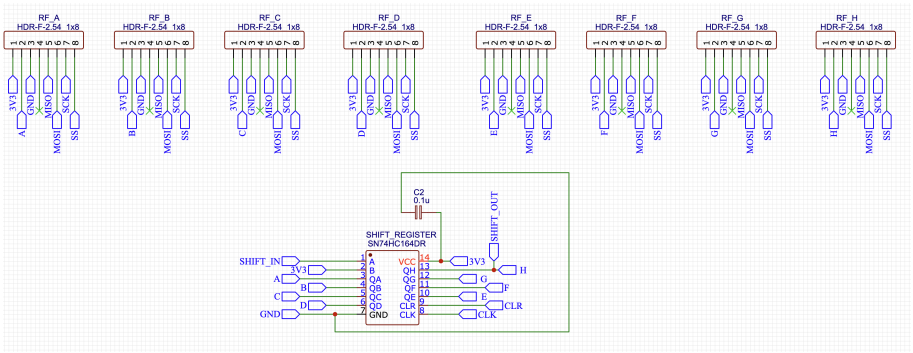
4.1.2 Wiring multiple RC522s together

To manage the 64 `RC522` modules, a shared bus architecture is employed. The primary SPI communication lines (`MOSI`, `MISO`, and `SCK`) as well as the Slave Select (`SS`) line are common among all modules. A shift register is then used to individually control the `RST` pin of each module, which allows us to ensure only one reader is active and communicating on the bus at any given time. Since dedicating an individual GPIO pin from the `ESP32` to the `RST` pin of each module would be impractical, the shift register provides a more efficient solution. With this approach, we can control all 64 modules using just a few GPIO pins. The simplified control scheme relies on a shift register with three control lines:

- **`CLK` (Clock):** On each pulse, the register shifts its current state by one position.
- **`CLR` (Clear):** When triggered, this asynchronously resets all outputs of the register to ‘0‘.
- **`A` (Data Input):** This is the serial input used to feed data into the first position of the register.



(a) Shift register diagram source: [21]



(b) Schematic of multiple RC522s connected

Figure 4.1: Overview of wiring multiple RC522s

(A second input, `B`, is tied to VCC, effectively keeping the data input stage permanently enabled for receiving data from input `A`). To sequentially activate one `RC522` module at a time, we use a "walking bit" method:

1. First, the `CLR` pin is pulsed. This sets all 64 outputs to '0', ensuring all `RC522` modules are in a deactivated state.
2. To activate the first module, the data input `A` is set to '1' and the `CLK` is pulsed once. This loads a '1' into the first stage of the register (`10000000...`), activating the corresponding `RC522`.
3. The data input `A` is then set back to '0'.
4. For every subsequent `CLK` pulse, the single '1' is shifted one position down the register (`01000000...`, `00100000...`, etc.). This effectively "walks" the active signal across the outputs, enabling one `RC522` at a time for scanning.

This method is highly scalable, as multiple shift registers can be daisy-chained without requiring additional GPIO pins.

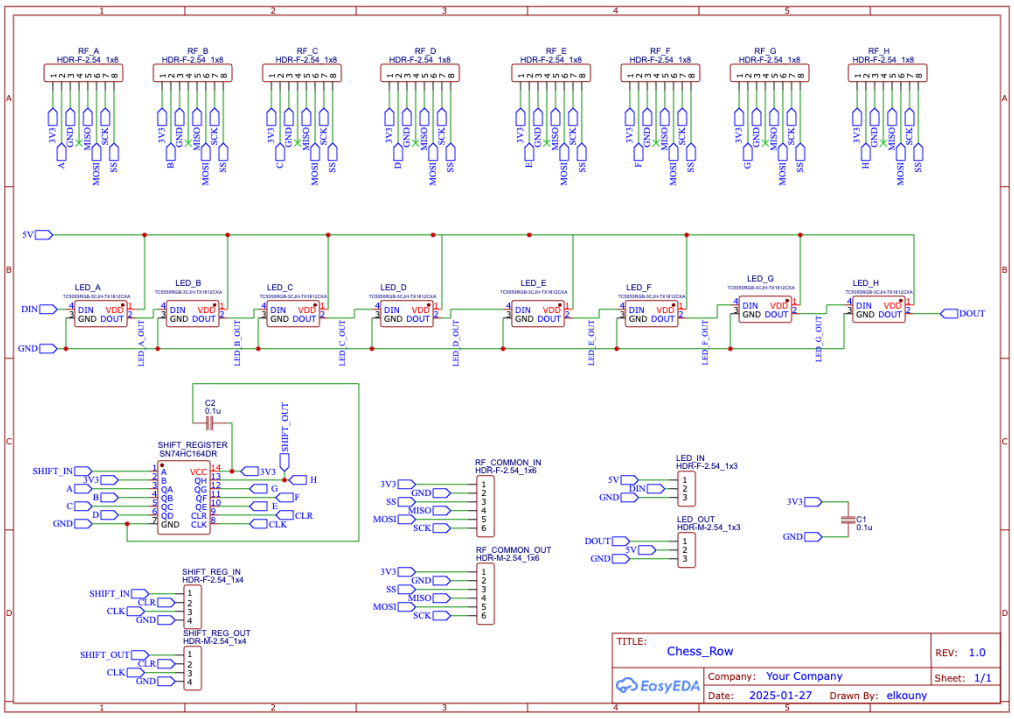
4.2 PCB Design

The project involves 64 individual `RC522` modules, along with 64 lights presents a significant wiring challenge. Each of the 64 modules requires seven distinct connections:

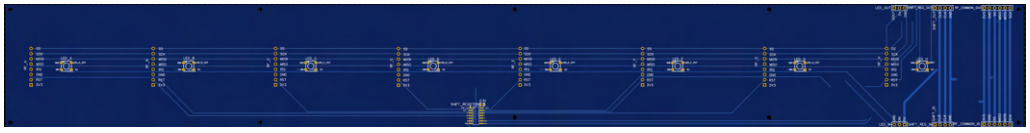
- Power (`3.3V`)
- Ground (`GND`)
- SPI bus lines (`MOSI`, `MISO`, `SCK`, `SS`)
- A reset line (`RST`) connected to a shift register output

This results in 448 wires for the readers alone. Additionally, the eight daisy-chained shift registers require their own control lines, adding further complexity. The total number of connections, exceeding 470, makes manual point-to-point wiring not only impractical but also highly susceptible to connection failures and noise. To address this complexity and ensure system reliability, a custom Printed Circuit Board (PCB) was designed.

4.2.1 Modular design



(a) The electronic schematic for a single row, detailing the connections for the eight RC522 modules, the controlling shift register, and the input/output headers for daisy-chaining.



(b) The corresponding PCB layout, illustrating the physical placement of the RFID readers, LEDs, and board-to-board connectors on the 48 cm × 6 cm board.

Figure 4.2: Design overview for a single chessboard row, from the electronic schematic (a) to the final physical PCB layout (b).

Rather than creating a single, large PCB for the entire 8x8 grid, a modular design approach was adopted. A dedicated PCB was designed to accommodate a single row of eight `RC522` modules. This modular strategy offers several key advantages. It simplifies the manufacturing and assembly process and creates a logical mapping between the physical layout and the electronic control system, as each row of eight readers corresponds perfectly to a single 8-bit shift register. More importantly, this design enhances the system's maintainability. In the event of a component failure, an entire row can be quickly and easily replaced without disturbing the rest of the board.

To facilitate the modular, row-by-row design, each PCB is equipped with standardized input and output headers, allowing the boards to be daisy-chained using board-to-board connectors. The

inputs are located at the bottom edge of the PCB, while the outputs are at the top. This creates a clear signal flow across the entire chessboard, encompassing three main bus systems:

- **Shift Register Bus:** The primary control signals for the shift registers (`CLK`, `CLR`, and `GND`) are passed directly through from the input header to the output header. To daisy-chain the registers' data lines, the serial output from the last register on a board (`Shift_out`) is routed to the output header, ready to connect to the `Shift_in` of the subsequent board.
- **NeoPixel Bus:** The NeoPixel LEDs are also connected in a daisy-chain fashion. Each board receives power (`5V`, `GND`) and a data signal (`Din`). The data passes through the LEDs on the board and exits via a `Dout` pin, which connects to the next board's `Din`. This mechanism operates similarly to a shift register, a concept that will be detailed further in a later section.
- **RFID SPI Bus:** The entire SPI bus required for the `RC522` readers is passed directly through each board. This includes power (`3.3V`, `GND`) and all data and control lines (`MOSI`, `MISO`, `SCK`, and `SS`). The input and output headers have an identical pinout for this bus.

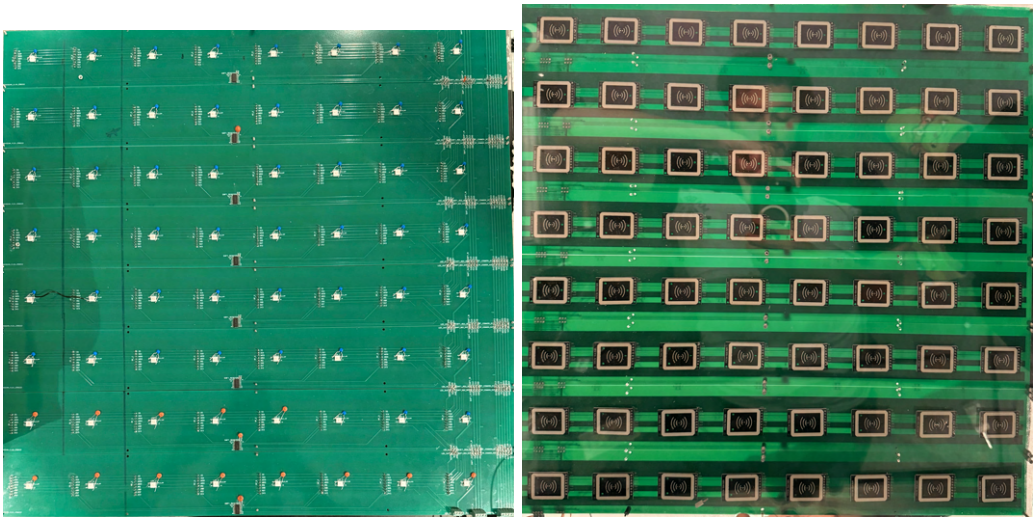
In the final assembly, the `ESP32` controller connects to the input header of the first row. Each subsequent row connects to the output of the one before it, and the output header of the final (eighth) row is left unconnected, or "floating."

4.2.2 PCB Layers and Power Plane

The board was fabricated as a standard **two-layer PCB**. A key feature of this design is the strategic use of the bottom layer for power distribution. Instead of routing power with thin traces, the bottom layer is predominantly covered by three distinct **copper pours**, creating dedicated planes for **5V**, **3.3V**, and **Ground** . This approach offers several critical advantages for system stability and performance:

- **Stable Power Distribution:** The large copper planes provide a low-impedance path for current. This minimizes voltage drop across the board, ensuring that all components receive a clean and stable power supply.
- **Noise Reduction:** The extensive ground plane acts as a shield, reducing susceptibility to electromagnetic interference .

4.2.3 Layout and Dimensional Considerations of PCB



(a) Top view of the assembled board, showing the 8x8 grid of indicator LEDs.

(b) Bottom view of the assembled board, revealing the matrix of 64 RC522 RFID readers.

Figure 4.3: Top and bottom views of the fully assembled 8x8 chessboard PCB.

The physical layout of the PCB was primarily dictated by the dimensional constraints of a standard tournament chessboard. Adhering to these standards, which specify a square size of approximately **6 cm** × **6 cm**, required a space-efficient method to integrate both an RFID reader and a visible indicator LED within each cell. To achieve this, a **double-sided component mounting** strategy was adopted. For each square:

- The **indicator LED** was placed on the **top surface** of the PCB, ensuring it was clearly visible to the player.
- The corresponding **RFID reader** was mounted on the **bottom surface** of the PCB, directly underneath the LED's position.

This two-sided assembly is highly space-efficient, allowing both components to occupy the same footprint within the square while being on different layers. This design maintains the required 6 cm horizontal spacing for each of the eight cells, resulting in a modular row PCB with final dimensions of **48 cm in width** (8 cells × 6 cm per cell) and **6 cm in height**.

4.3 Game State

4.3.1 Board Setup and Validation

Before a game can begin, the system enters a ”waiting for setup” state to ensure all pieces are placed correctly on the board. This validation process is crucial for establishing the correct initial game state.

The setup logic proceeds as follows:

1. **Monitoring Valid Squares:** The system actively monitors only the first two and last two ranks of the board, as these are the designated starting rows for all pieces. Any piece placed outside these areas during the setup phase would be flagged as an error.

2. **Unique ID (UID) Verification:** As described in the RFID technology section, each physical chess piece has a unique, known UID. When a player places a piece on a valid starting square, the system reads its UID.

3. **Piece Set Validation:** The system then cross-references the detected UID with a predefined set of valid UIDs for that specific piece type. For example, if a player places a piece on a square designated for a white pawn, the system checks if the detected UID belongs to the set of eight UIDs assigned to the white pawns. This ensures that a player cannot, for instance, place a rook where a pawn should be.

4. **Ready State Trigger:** The system maintains a count of correctly placed pieces. Once all 32 pieces have been placed on their correct starting squares and validated, the board transitions from the ”waiting for setup” state to the ”ready” state, signaling that the game can now begin.

4.3.2 Determining Board Events on the ESP32

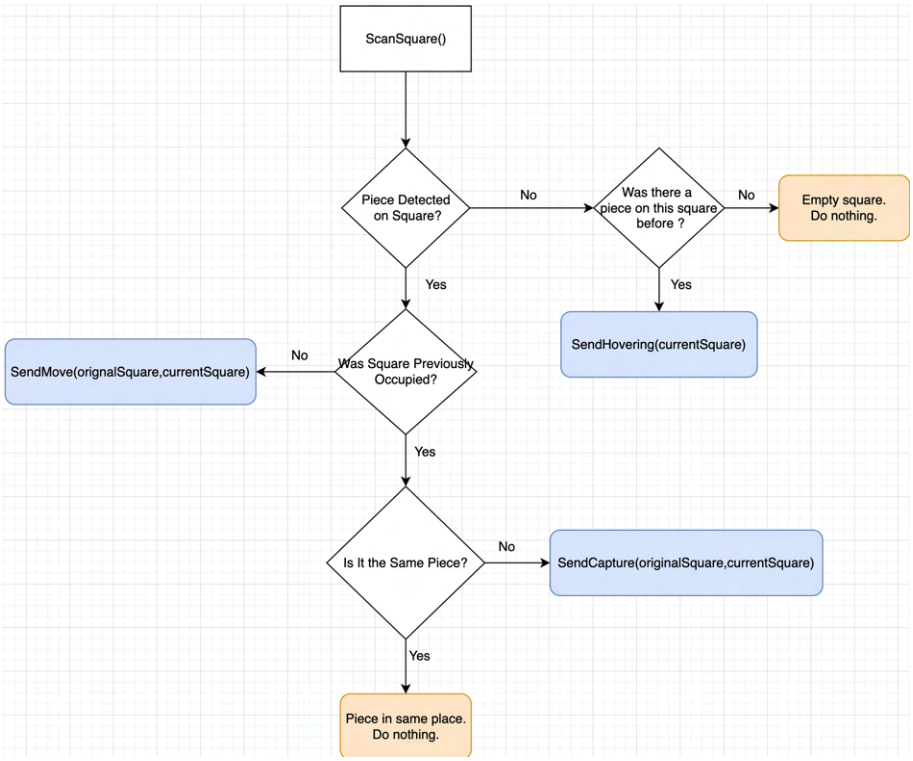


Figure 4.4: A high-level flowchart of the board's event detection logic, showing the paths to identify a hover, move, or capture.

As established in the system architecture, the ESP32 is dedicated to monitoring the physical state of the board and reporting events to the companion app, which maintains the official game logic. The firmware is programmed to recognize three main events.

- **Hovering** : This occurs whenever a player lifts a piece from a square. When this happens, the board illuminates all valid destination squares to provide instant visual feedback to the player.
- **Normal move** : This event is detected when a piece is moved from its starting square to a previously empty destination square.
- **Capture** : A capture is identified when a piece is moved to a destination square that was occupied by an opponent's piece.

To track the real-time physical locations of the pieces, the ESP32 utilizes a custom-made **BiMap** class, which functions as a two-way hash map. This data structure is designed to maintain

a one-to-one mapping between the unique ID of each piece's RFID tag and its alphanumeric position on the board (e.g., `e3`, `e4`, etc.). It achieves this by internally using two standard hash maps:

1. A "forward" map that stores: `unique_id` → `position`
2. An "inverse" map that stores: `position` → `unique_id`

The selection of this data structure is critical for the system's real-time performance due to its impressive **time complexity**. Because the BiMap is built on hash maps, it provides an average time complexity of $O(1)$ **(constant time)** for both lookups and updates. This efficiency is essential for the two primary operations required by the event-detection logic:

- **Inverse Lookup (Position → ID):** When a square is scanned, the system can instantly check if a piece was previously recorded at that position and retrieve its unique ID in $O(1)$ time. This is fundamental for detecting hover and capture events.
- **Forward Lookup (ID → Position):** When a piece appears on a new square, the system can retrieve its original position in $O(1)$ time, which is necessary for constructing a complete move message.

If a less efficient data structure like an array or a list were used, finding a piece by its position would require iterating through the entire collection, resulting in a time complexity of $O(n)$. This would introduce significant processing delays, making the BiMap's constant-time performance essential for a responsive user experience.

4.3.3 Synchronising State Between the Mobile App and ESP32

To ensure the game state remains consistent between the physical board and the app engine, a handshake protocol was implemented. The core principle of this protocol is that the **mobile app holds the authoritative or "master" game state**. The ESP32 monitors the physical state and proposes changes, but it will not update its own internal state representation until the app validates and approves the move.

Valid Move Protocol

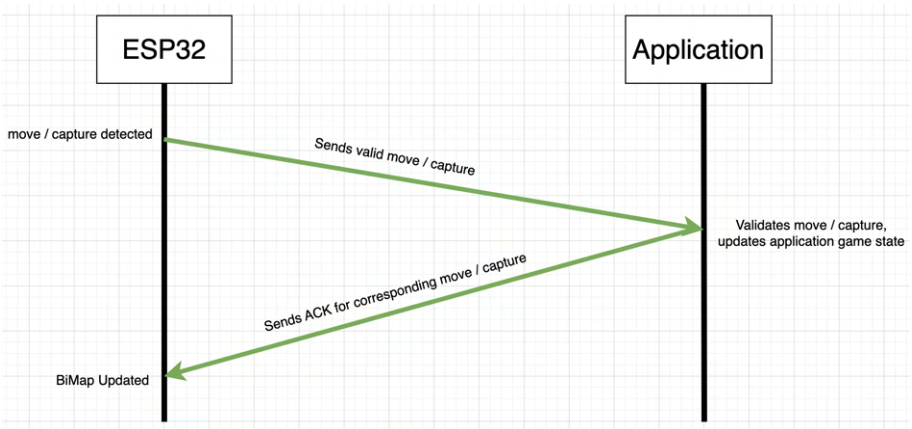


Figure 4.5: The handshake protocol for a valid move, which successfully concludes with an acknowledgment (ACK) from the companion app.

When a player makes a valid move, the synchronization follows this sequence:

1. **Detection (ESP32):** The ESP32 detects a potential move or capture (e.g., a piece moving from `e2` to `e4`).
2. **Proposal (ESP32 → App):** Crucially, the ESP32 **does not** immediately update its own BiMap. Instead, it sends the proposed move as a message (e.g., `"move:e2e4"`) to the application via Bluetooth.
3. **Validation (App):** The application receives the message and checks the move against the master game state. It validates the move based on two conditions: is the move legal by the rules of chess, and is it the correct player's turn?
4. **Confirmation (App → ESP32):** Once the move is confirmed as valid, the application sends an acknowledgment (**ACK**) message back to the ESP32. It then updates its own game state.

5. **Synchronization (ESP32):** Only after receiving the ACK does the ESP32 update its internal BiMap. At this point, the physical board, the ESP32's state, and the app's state are all synchronized.

Handling Invalid Moves

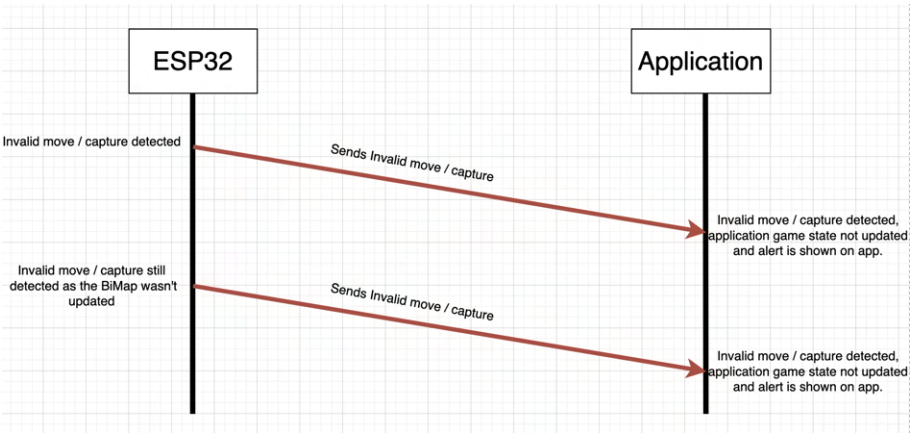


Figure 4.6: The protocol for handling an invalid move. No ACK is returned, ensuring the ESP32's internal state is preserved and remains correct.The invalid move will continue to be sent until the user moves the piece to a valid square or back to its original position

If a player attempts an illegal move, the protocol ensures the game state is not corrupted:

- The ESP32 detects the illegal move and sends it to the app, just as with a valid move.
- The app's engine validates the move and finds it to be invalid. It **does not send an ACK**. An error message will appear on the app's interface.
- Because the ESP32 never receives an ACK, its BiMap remains unchanged. On every subsequent scan, it will detect the same physical discrepancy (the piece is on a square where it shouldn't be) and will continue to send the same invalid move message.
- This loop is only broken when the player corrects the physical state of the board. If the piece is returned to its original square, the board will again match the ESP32's BiMap, and the messages will cease. If the piece is moved to a new, *valid* square, the valid move protocol will be initiated and the states will successfully synchronize.

4.3.4 Handling Special Moves

Beyond standard moves and captures, the system is designed to handle the three special moves in chess: en passant, castling, and pawn promotion. Each requires a unique interaction between the user, the ESP32, and the companion app.

En Passant

When an *en passant* capture is performed, the app's engine is responsible for deconstructing the event into two distinct actions.

1. The player physically moves their pawn to the empty capture square (e.g., from `d5` to `e6`).
2. The ESP32 detects this as a standard move (`"d5e6"`) and sends it to the app for validation.
3. The app's engine recognizes the move as a valid en passant. It then sends back two separate **ACKs** to the ESP32: the first confirms the **capture** of the opponent's pawn (at `e5`), and the second confirms the **move** of the player's pawn to its destination (`e6`).
4. Upon receiving both ACKs, the ESP32 updates its BiMap accordingly.
5. **Error Handling:** If the player fails to physically remove the captured pawn from the board, the ESP32 will detect a piece on a the board that shouldnt be there. This discrepancy will be flagged as an error until the board's physical state is corrected.

Castling

Castling is initiated by the king's movement and is confirmed with two separate move **ACKs** from the app.

1. In accordance with chess rules, the player first moves the king two squares (e.g., from `e1` to `g1`).
2. The ESP32 detects this king-only move and sends it to the app.
3. The app's engine validates this as a castling move. It then sends two distinct move **ACKs** to the ESP32: one to authorize the **king's move** (to `g1`), and a second to authorize the **rook's corresponding move** (to `f1`).

4. The ESP32 processes both ACKs to update its BiMap for the new positions of both the king and the rook.

5. **Error Handling:** The player is now expected to physically move the rook to its correct castled position. If they fail to do so, the ESP32 will detect that the rook is on the wrong square, and the system will report an error state until the rook is moved.

Pawn Promotion

Pawn promotion is handled primarily by the application, as the ESP32 is agnostic about piece types.

1. The player moves a pawn to the final rank (e.g., from `e7` to `e8`).

2. The ESP32 detects and reports this as a normal move. The app validates it and sends a standard **ACK**.

3. The ESP32 updates its BiMap to show the original pawn's RFID tag is now at square `e8`.

4. Simultaneously, the companion app prompts the user to choose a piece to promote to (e.g., a Queen).

5. The app's **master game state** is updated to replace the pawn with the new piece (e.g., a Queen at `e8`). The ESP32's physical state tracker, however, does not need to be updated with the new piece's type, as it only cares about the presence and location of a piece via its unique ID.

4.4 Lighting System

The NeoPixel lighting system is a core component of the user experience, providing intuitive visual feedback during all phases of the game. The behavior of the lights is tailored to the specific context, from initial setup to visualizing online play.

Game Setup Phase

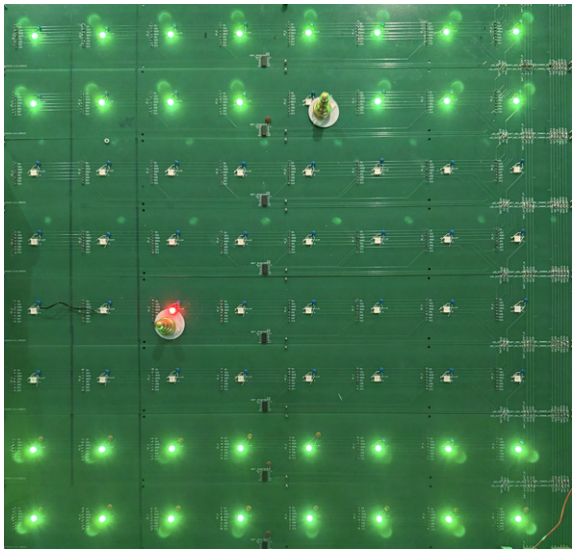


Figure 4.7: Lights for game setup phase

To guide the initial placement of pieces, the system enters a setup mode.

- **Placement Guidance:** The LEDs on the first two and last two ranks are lit **green**, indicating where the white and black pieces should be placed.
- **Error Indication:** If a piece is placed on any other row during this phase, the corresponding cell immediately turns **red** to signal an incorrect placement.
- **Ready Confirmation:** Once all pieces are correctly positioned, the board displays a ”firework” animation, providing a clear and satisfying signal that the game is set up and ready to begin.

In-Game Player Feedback

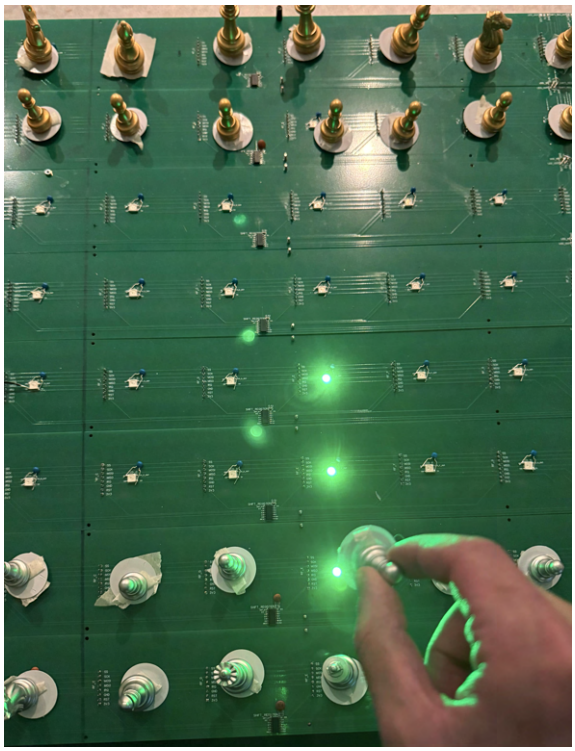


Figure 4.8: Lights indicating all valid moves for hovering pawn

Drawing inspiration from modern chess applications, the lighting system provides real-time feedback to the player.

- **Legal Move Indication:** When a player lifts a piece (triggering a ”hover” event), the companion app calculates all valid moves and sends a message back to the ESP32 containing the coordinates of all possible destination squares (e.g., `light_up:e3e4f5`). The ESP32 firmware then parses this string, and for each alphanumeric coordinate, it calculates the corresponding NeoPixel index on the 64-light strip. Finally, the system illuminates these specific LEDs: **green** for empty squares and **red** for squares occupied by an opponent's piece, clearly indicating a potential capture.
- **Check Warning:** If a player's king is in check, the square underneath the king is lit with a constant **red** glow, serving as an immediate visual warning.

Visualizing Online Moves

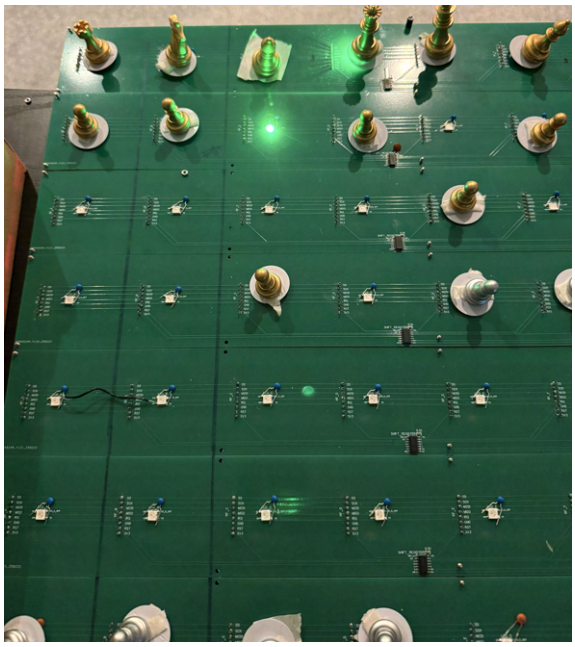


Figure 4.9: Visualizing an opponent's online Queen move by illuminating the path from its start to end square.

When playing an online match, the lighting system is used to clearly animate the opponent's moves. Since the system knows both the start and end squares, it can illuminate the path the piece takes.

- **Knight Moves:** As a knight's move is non-linear, the system simply illuminates the start and end squares.
- **Sliding Moves:** For sliding pieces (Rooks, Bishops, and Queens) and pawns, the system lights up all the intermediate squares between the start and end position along their line of movement (i.e., along the rank, file, or diagonal). This creates a clear visual animation of the move as it happens.

4.5 Application Development

The design philosophy for the companion application centered on three core principles: simplicity, an intuitive user experience (UX), and seamless integration with the physical board. The goal was to create a single, cohesive product where the hardware and software feel completely interconnected.

4.5.1 Lichess Authentication

To enable online play, the application must interact with the Lichess API, which requires user authentication. This is handled through a secure, one-time login process to provide a smooth user experience.

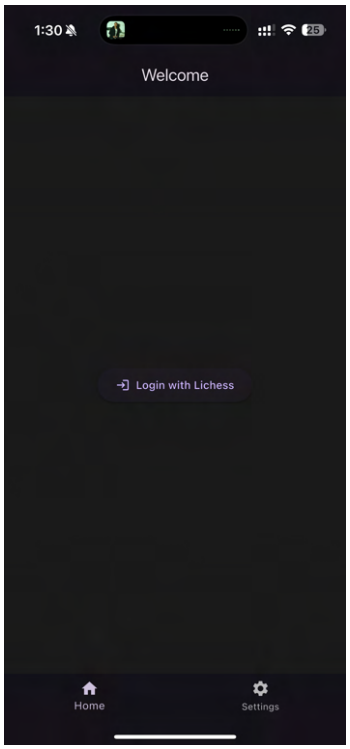
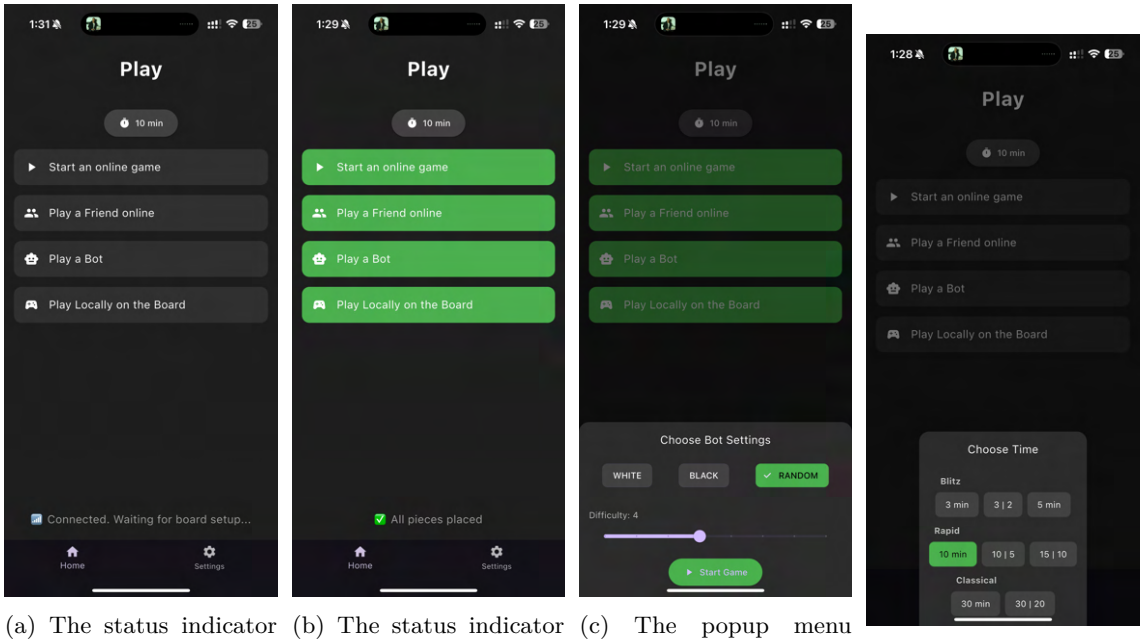


Figure 4.10: The application's login screen, which facilitates a one-time authentication with Lichess to acquire an API access token.

Upon first use, the user must log in to their Lichess account via this screen. The application then uses these credentials to request a secure API access token from Lichess. This token, not the user's actual password, is then stored locally on the device. For all subsequent sessions, the app uses the stored token to authenticate with the Lichess API automatically, meaning the user does not need to log in again.

4.5.2 Home Page

The home page serves as the central hub for the application, providing access to all game modes and displaying the board's connection status. The interface is designed to be clean and intuitive, ensuring the user can quickly start a game.



(a) The status indicator when the board is connected but awaiting piece setup. Buttons are disabled.

(b) The status indicator showing the board is connected and ready. Game mode buttons are now enabled.

(c) The popup menu for configuring a game against a bot, allowing the user to select difficulty and color.

(d) The popup for selecting time controls, offering standard Blitz, Rapid, and Classical formats.

Figure 4.11: An overview of the application's home page UI, showing various game option popups and board connection status indicators.

Game Modes

The main screen presents the user with four distinct game modes:

- **Play Online (Random):** Searches for a random opponent on Lichess.
- **Play Online (Friend):** Allows the user to challenge a specific Lichess friend.
- **Play vs Bot:** Initiates a game against the Stockfish engine.
- **Play Locally:** Enables a two-player game on the physical board.

Board Connection Status

A status bar at the bottom of the screen displays the real-time connection state of the chessboard. This is crucial for user feedback and has several states, including ”Scanning for board,” ”Disconnected,” ”Connected, waiting for setup” (Figure 4.11a), and ”Connected and Ready” (Figure 4.11b). To prevent errors, all game mode buttons are greyed out and disabled until the board reports a ”Connected and Ready” status.

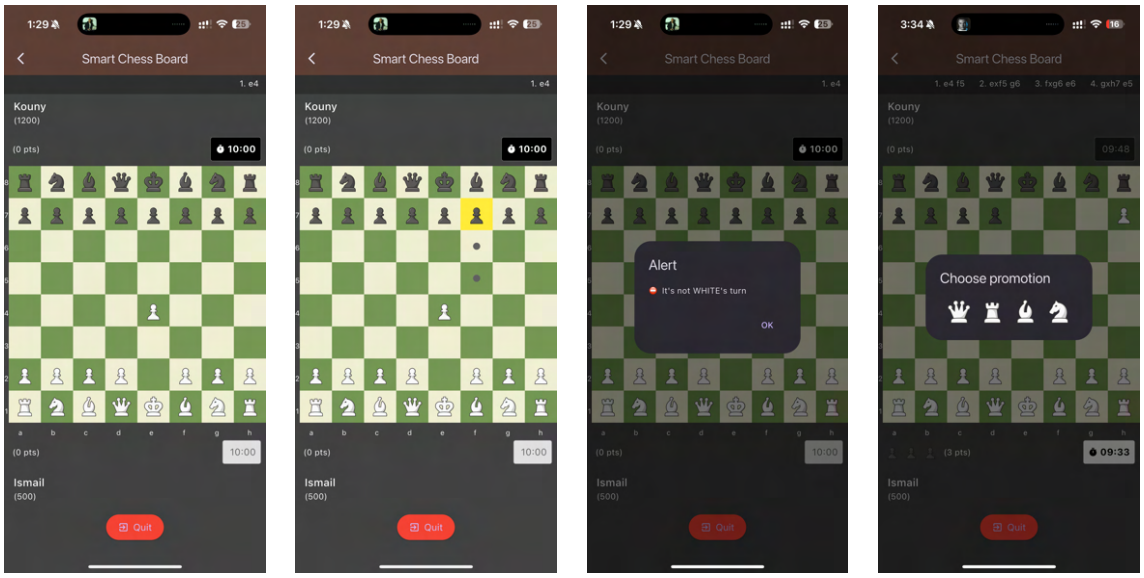
Game Options

Before starting a game, users can configure specific options via popup menus.

- **Time Control:** A timer option brings up a menu to select from Blitz, Rapid, and Classical time formats, each with pre-set base times and increments (Figure 4.11d).
- **Bot Configuration:** When playing against a bot, the user can choose a difficulty level from 1 to 8 and select their preferred piece color (White, Black, or Random), as shown in Figure 4.11c.

4.5.3 Game Screen UI

The user interface for the game screen is built using Flutter's declarative widget system, arranged primarily in a vertical `Column`. The layout is designed to present all necessary game information clearly, as shown in Figure 4.12. The `_game` object, an instance from the Flutter chess library, stored and managed the master game state.



(a) The screen after a move, with updated log.

(b) Highlighting valid moves during a hover.

(c) An alert for an invalid move.

(d) The pawn promotion dialog.

Figure 4.12: The main user interface of the application's game screen, showing different states of user interaction.

- **Player Information:** The top and bottom of the screen are dedicated to displaying player information, including names, ratings, a list of captured pieces, and the active game timer for each side.
- **Chessboard Rendering:** The board is rendered by the `_buildBoard` method, which uses a `GridView.builder` to create the 8x8 grid. For each of the 64 squares, the `_buildSquare` function is called, which:
  - Converts the grid index to an algebraic coordinate (e.g., "e4").
  - Checks the `_game` object to see if a piece exists on that square.
  - Uses a `Stack` to layer UI elements. The base layer is the square's color. A highlight is then added to indicate a valid move destination—a grey circle for an empty square or a red highlight for a capture (Figure 4.12b).

  - The top layer is the piece's `.png` image asset, if present.
  - The entire board's orientation can be flipped based on the `isBoardFlipped` getter, which checks the player's assigned color.

- **Move Log:** A horizontal, auto-scrolling `ListView` at the top of the screen displays the move history fetched from the `_game` object (Figure 4.12a).

- **User Feedback:** The application provides immediate feedback for invalid actions by displaying a modal alert dialog (Figure 4.12c). It also handles special game rules by pausing the game to display a pawn promotion dialog when required (Figure 4.12d).

4.6 Lichess Integration

4.6.1 Local Play

In the ”Local Play” mode, two users can play a game against each other directly on the physical chessboard. The application tracks the moves, and upon the game's conclusion (either through a standard end condition like checkmate or if a player quits), it facilitates the archiving of the game to Lichess for later review and analysis.

The process for uploading the locally played game is as follows:

1. **Game Conclusion:** Once the local game has ended, the application extracts the complete game record in Portable Game Notation (PGN) format.
2. **API Request Preparation:** An HTTP POST request is prepared, targeting the Lichess API endpoint: `https://lichess.org/api/import`.
3. **Authentication:** The user's Lichess API token, previously obtained during the login process, is included in the header of this request for authorization.
4. **Data Transmission:** The PGN string of the played game is sent as the body of the POST request.
5. **Lichess API Response:** If the import is successful, the Lichess API responds with a status code `200 OK` and a JSON object. This object contains the unique ID assigned to the imported game and a direct URL to access it on Lichess. For example:

```
{
  "id": "R6iLjwz5",
  "url": "https://lichess.org/R6iLjwz5"
}
```

6. **Access and Review:** The user can then use the provided URL to view and analyze their fully played local game on the Lichess platform, including using Lichess's powerful analysis tools.

4.6.2 Initiating Online Games

The application supports three distinct online game modes, all facilitated through integration with the Lichess API: playing against a bot, issuing an open challenge to a random online opponent, or challenging a specific Lichess friend.

Playing Against a Bot

To start a game against Lichess's AI, the application sends an HTTP POST request to the `https://lichess.org/api/challenge/ai` endpoint. This request includes parameters for the bot's difficulty level (1 to 8), clock limit, clock increment, and the player's desired color (white, black, or random). A successful request returns a `201 Created` status code and a JSON body containing the game details. For example:

```
{
  "id": "FLuFXFzs",
  "variant": {"key": "standard", "name": "Standard", "short": "Std"},
  "speed": "correspondence",
  "rated": false,
  "player": "white",
  "status": {"id": 20, "name": "started"}
  // ... other fields omitted for brevity
}
```

From this response, the application extracts the **game ID** (e.g., `FLuFXFzs`) and the assigned **player color**. The player color is passed to the game UI to orient the board correctly, and the game ID is used for subsequent API calls to stream game events and send moves.

Playing Against a Random Opponent

To play against any available online opponent, the application sends an HTTP POST request to the `https://lichess.org/api/board/seek` endpoint. This request includes the desired color, time control, and increment. Lichess then attempts to match the player, and the API response primarily returns the **game ID** if a match is found:

```
{
  "id": "gwkzmEBY"
}
```

Once the game ID is obtained, the application listens to the game's event stream to determine the assigned player color and other game details.

Challenging a Friend

To challenge a specific Lichess user, the application sends an HTTP POST request to the `https://lichess.org/api/challenge/{username}` endpoint. This request requires the target username in the path, along with parameters for time control, increment, and color. A successful challenge creation returns a `200 OK` status and a JSON response detailing the challenge, for example:

```
{
  "id": "1jqMGJOv",
  "url": "https://lichess.org/1jqMGJOv",
  "status": "created",
  "challenger": {"id": "bobby", "name": "Bobby", "rating": 1612},
  "destUser": {"id": "mary", "name": "Mary", "rating": 1064},
  "variant": {"key": "standard", "name": "Standard"},
  "rated": false,
  "color": "random",
  "finalColor": "white"
  // ... other fields omitted for brevity
}
```

The crucial information extracted here includes the **game ID** and the **player's final assigned color** (`finalColor`).

4.6.3 Game Flow

Once an online game is set up and the game ID is obtained, the general flow of interaction with the Lichess API is as follows:

1. **Player's Turn :** If it is the application user's turn to move (e.g., they are playing as White and it is White's move), they make the move on the physical board. The app then sends this move to Lichess via an HTTP POST request to the `https://lichess.org/api/board/game/{gameId}/move/{move}` endpoint. The `{gameId}` is replaced with the actual game ID, and `{move}` is the move in UCI format (e.g., `e2e4`). A successful response is typically a `200 OK`.

2. **Listening for Opponent's Moves:** The application listens for incoming game events. This is done by making an HTTP GET request to `https://lichess.org/api/board/game/stream/{gameId}`. This endpoint streams events as newline-delimited JSON (NDJSON). The app monitors this stream for `gameState` type events, which provide the current move list, remaining time for both players, and other status updates. An example `gameState` event might look like:

```
{
  "type": "gameState",
  "moves": "e2e4 c7c5 f2f4 d7d6 g1f3",
  "wtime": 7598040, "btime": 8395220,
  "winc": 10000, "binc": 10000,
  "status": "started"
  // ... other fields
}
```

3. **State Synchronization:** The application compares the list of moves received from the Lichess stream (`event['moves']`) with its local game state. If the Lichess stream contains moves not yet present in the local state, it signifies an opponent's move. The app then updates its board representation on the app and sends a message to the ESP32 to light up the board and move the piece.

4. **Repeating the Process:** This cycle of making a move, sending it to Lichess, and then listening for the opponent's response (or listening first if it's the opponent's turn) repeats until the game concludes.

4.7 Design and Assembly

4.7.1 H-BOT

The designs for the 3D printed components, such as the motor mounts and carriage assemblies, were adapted from the open-source plans for a Kinetic Sand Art Table by DIY Machines [22].

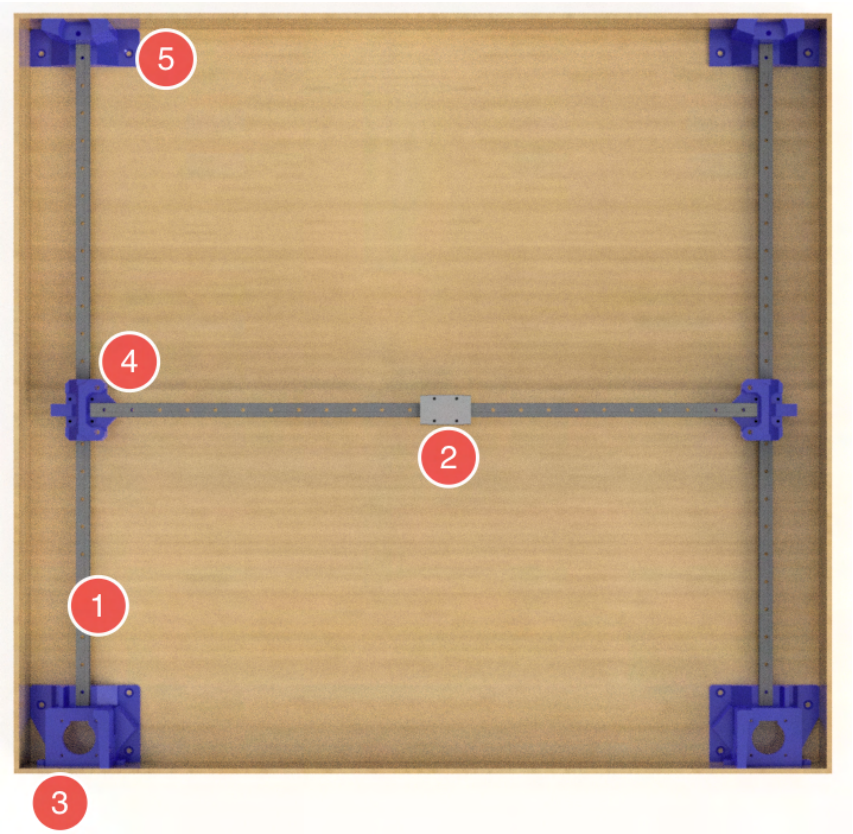


Figure 4.13: A CAD rendering of the assembled H-Bot gantry, showing the primary mechanical components and their layout on the baseboard.

Label 1: Main Linear Rails Two parallel linear rails, each 600 mm in length, define the primary (Y-axis) movement space. This length was chosen to comfortably exceed the 500 mm board dimension, accounting for any ”dead zones” at the ends of the rails that the carriages cannot reach.

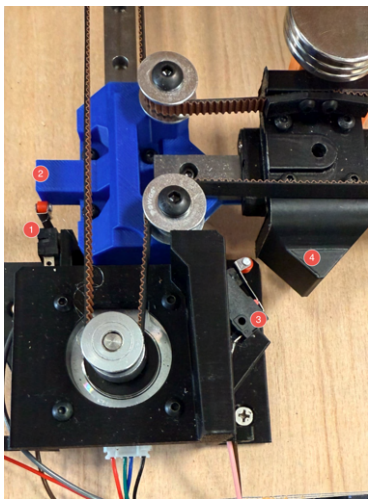
Label 2: Central Carriage Assembly This assembly moves along the horizontal rail (X-axis). It serves as the mounting platform for the servo-actuated magnet holder mechanism. This mechanism is positioned on top of the carriage but is not depicted in this particular diagram. The assembly also features an integrated, pointed extrusion that functions as an actuator

tab, designed to trigger the X-axis limit switch during the homing sequence.

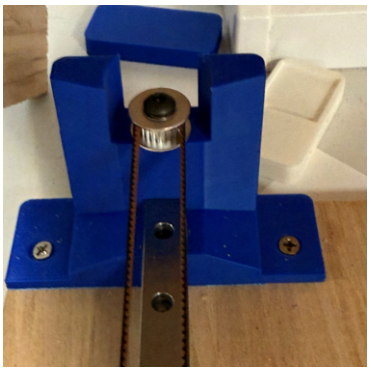
Label 3: Motor Mounts and Rail Supports These brackets securely hold the two stepper motors and also act as the end supports for the main linear rails. Each stepper motor is fitted with a **toothed timing pulley** to drive the belt. The brackets also include provisions for mounting the Y-axis limit switches.

Label 4: Horizontal Carriage Mounts These components connect the horizontal rail (X-axis) to the linear bearing carriages on the main rails. They are fitted with **smooth (non-teethed) idler pulleys**, which guide the back (smooth) side of the timing belt. Each mount also features an integrated "tail" that extends outwards to physically trigger the Y-axis limit switches.

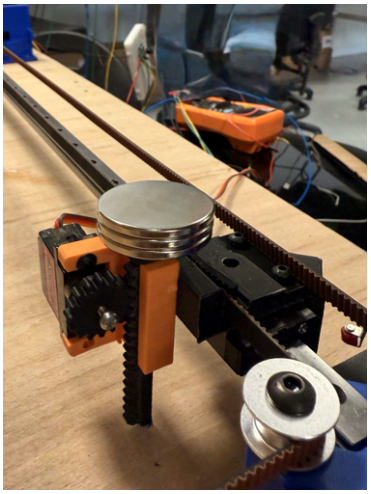
Label 5: Idler Supports Positioned opposite the motor mounts, these brackets hold the **toothed idler pulleys**. These pulleys guide the toothed side of the timing belt, completing the "H" shaped path and ensuring proper tension and movement.



(a) The 3D printed motor mount. The Y-axis homing switch (1) is positioned to be triggered by the carriage mount's tail (2). The X-axis switch (3) is triggered by the extrusion on the central magnet mount (4).



(b) The idler support bracket, designed to hold a toothed pulley at the corner opposite the drive motor.



(c) The central carriage assembly, which houses the servo motor and the magnet holder for piece actuation.

Figure 4.14: The three primary 3D printed components of the H-Bot gantry system, adapted from an open-source design.

4.7.2 PCB Case

Base Plate

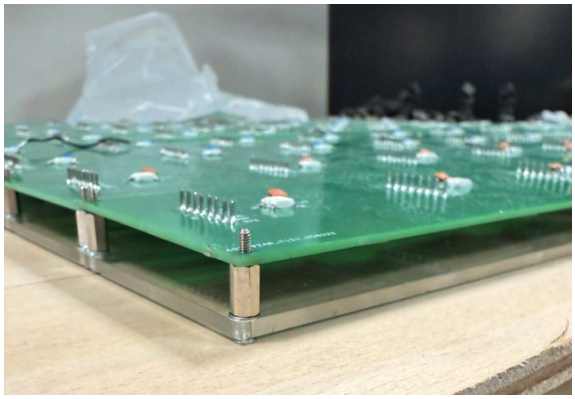


Figure 4.15: The 3mm acrylic base plate, featuring mounting holes for the PCB standoffs.

The foundation of the enclosure is a 500,mm x 500,mm sheet of 3mm thick acrylic, shown in Figure 4.15. This base plate includes precisely located holes, which were cut using a laser cutter, for mounting the eight PCB rows on standoffs. The standoffs serve the dual purpose of securely holding the PCBs in place while also creating a necessary gap between the base and the electronics.

Top Case

(a) The top acrylic sheet after laser engraving, creating the frosted white squares.

(b) The completed playing surface, created from two types of acrylic.

(c) An external L-bracket (1) used for assembling the side walls along with the 3D printed rail (2) to hold the top acrylic.

Figure 4.16: Components of the top case assembly.

- **Playing Surface:** The top is a 500 mm x 500 mm assembly, 1mm thick, composed of two different types of acrylic that were precisely laser-cut and fitted together.
  - The **white squares** are made from a transparent acrylic that was laser-engraved to create a "frosty" finish, which effectively diffuses light from the LEDs underneath (Figure 4.16a).
  - The **black squares** are made from a special **black light-diffusing acrylic**, which provides the dark color while still allowing the LED light to pass through.
- **Assembly Method:** The top playing surface assembly slides into a custom 3D-printed rail, which is attached to the inside of four long acrylic side walls. This design choice ensures the playing surface is completely flush. These four walls are then joined at the corners using external L-brackets (Figure 4.16c). Mounting the brackets externally maximizes the internal volume, ensuring sufficient space for the PCBs and wiring.

4.7.3 Mounting the PCB Case onto the H-Bot

To integrate the two major sub-assemblies of the project, a method was designed to securely mount the finished PCB case on top of the H-Bot gantry system. The H-Bot mechanism is covered wth a large wooden sheet which serves as the mounting surface for the final chessboard enclosure.

(a) An overview of the 3D printed mounting brackets in position.

(b) A close-up view showing how the bracket's step design grips the corner of the case.

Figure 4.17: The custom-designed 3D printed brackets used to mount the PCB case onto the H-Bot base.

The mounting is achieved using four custom-designed, 3D printed brackets, as shown in Figure 4.17. These parts are designed with a "step" shape that allows them to grip the corner edges of the PCB case assembly. Each bracket has two screw holes on its base, allowing it to be securely fastened to the wooden plate.

4.7.4 Chess Pieces

The chess pieces for this project were 3D printed using a design adapted from an online source [23].

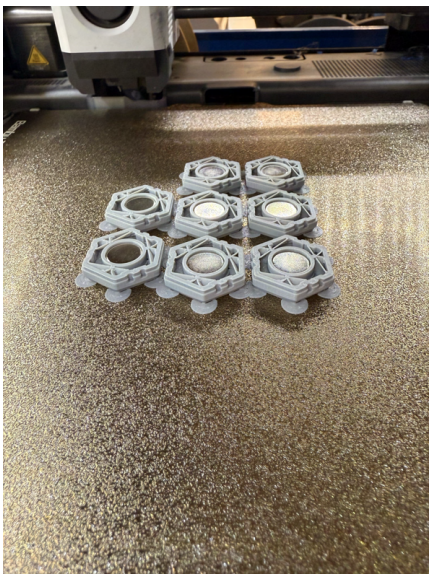


Figure 4.18: A view of the 3D print paused mid-process, showing the three Neodymium magnets placed inside the base cavity before being sealed.

1. **Model Modification:** Each piece's 3D model was edited to include a cylindrical cavity in its base. This hollow space was designed with dimensions of 12.2 mm in diameter and 6.2 mm in height to accommodate three stacked 12 mm x 2 mm circular Neodymium magnets.

2. **Embedded Magnet Printing Technique:** A specific 3D printing technique was used to embed the magnets seamlessly within the pieces. The printing process was configured to pause automatically just before the layer that would seal the top of the internal cavity was printed.

3. **Magnet Insertion:** During this programmed pause, the three Neodymium magnets were manually inserted into the printed cavity.

4. **Resuming the Print:** Once the magnets were in place, the 3D print was resumed. The printer then continued to build the subsequent layers on top of the magnets, effectively sealing them inside the base of the finished piece. This method ensures a clean finish with no visible signs of the embedded components.

4.8 Movement System

4.8.1 Homing and Calibration

Before the movement system can accurately position the magnet under the board, it must first be calibrated to establish a known reference point. This process, known as ”homing,” is performed once during the initial setup of the board.

The homing sequence is initiated by sending the system command `$H` to the GRBL controller via a UART serial connection. Upon receiving this command, the controller executes the following steps:

1. It moves the entire gantry along the Y-axis towards one end until the corresponding Y-axis limit switch is triggered. This establishes the machine’s zero position for the Y-axis.
2. It then moves the central carriage along the X-axis towards one end until the X-axis limit switch is triggered, establishing the zero position for the X-axis.

Once both switches have been triggered, the system has established its home position at coordinate `(0, 0)`. The physical chessboard enclosure has been mounted onto the H-Bot in such a way that the bottom-left corner of the playing area aligns precisely with this homing point. This ensures that the controller’s coordinate system directly maps to the chessboard grid, allowing for accurate movement across the entire board. After the homing cycle is complete, the machine is calibrated and waits for subsequent G-code commands to move the magnet to specific squares.

4.8.2 Executing an Online Move

When the companion app receives an opponent's move from the Lichess game stream, it forwards the move's start and end coordinates to the ESP32. The ESP32 then executes a sequence of actions to move the corresponding piece on the physical board.

Coordinate Conversion

The first step is for the ESP32 to convert the algebraic notation (e.g., "a1", "h8") received from the app into physical G-code coordinates. The system is calibrated such that the center of the first square, 'a1', corresponds to the machine coordinate `(X30, Y30)`. With each square being 60 mm wide, the center of any other square can be calculated with a simple formula. For example, the square 'c2' would be converted to `X(30 + 2*60)` and `Y(30 + 1*60)`, resulting in the G-code target `X150.0 Y90.0`.

Movement and Actuation Sequence

The ESP32 sends a series of commands to the GRBL controller on the Arduino to perform the move:

1. **Move to Start Position:** A G-code command (e.g., `G1 X30.0 Y30.0 F3000`) is sent to move the magnet to the center of the starting square.
2. **Wait for Arrival:** The ESP32's firmware pauses and monitors the serial data coming back from the GRBL controller. It waits for GRBL to send a status message (e.g., an `ok` response and its real-time status report changing to `Idle`), which confirms that the move is complete and the machine is stationary.
3. **Engage Magnet:** A command is sent to the servo to raise the magnet holder, bringing the magnet into contact with the piece.
4. **Move to End Position:** A new G-code command is sent with the coordinates of the destination square. The ESP32 again waits for GRBL's confirmation that this move has been completed.
5. **Disengage Magnet:** Finally, a command is sent to the servo to lower the magnet holder, releasing the piece on its new square.

Error Handling and State Lock

During this online move execution, the system enters a locked state where it will **only accept the expected move from Lichess**. The application's BLE message handler is configured to ignore any other physical board events (like a user manually moving a piece). If the physical actuation fails for any reason (e.g., a piece is dropped mid-move), the ESP32's board scan will detect a discrepancy between the expected physical state and the actual state. This will trigger an error message to be sent to the app, which can then alert the user to the problem, allowing them to manually correct the piece's position.

4.8.3 Handling special moves

The under-board H-Bot movement system must handle several special move cases that go beyond simple point-to-point travel. The two primary challenges are moving a Knight when its path is blocked, and executing a castling move.

Blocked Knight Moves

A Knight is the only piece that can "jump" over other pieces. Since the under-board magnet moves in straight lines, it cannot physically jump. Therefore, a special protocol involving user assistance is required when the magnet's path is obstructed.

1. **Path Obstruction Detected:** Before executing a Knight's move, the system's pathfinding algorithm checks if any squares between the start and end points are occupied.
2. **User Prompt:** If the path is blocked, the system alerts the user. The LEDs under the obstructing piece(s) flash yellow, and a message appears on the companion app prompting the user to temporarily remove the blocking piece(s).
3. **Knight Actuation:** Once the system confirms via RFID scan that the path is clear, the H-Bot gantry moves the Knight to its destination square.
4. **Piece Replacement Prompt:** The system then prompts the user, again via lights and the app, to return the piece(s) they had removed to their original squares. The game continues once the board state is correct.

Castling

Castling is a unique move where two pieces, the King and a Rook, move simultaneously. When the system needs to perform an opponent's castling move, it executes it as two sequential automated movements.

1. **King Movement:** The actuation system first moves the King two squares towards the Rook. For example, it moves the King from square `e8` to `g8`.
2. **Rook Movement:** Immediately after the King's move is complete, the system moves the corresponding Rook to the square the King just crossed. For example, it moves the Rook from `h8` to `f8`.

5

Testing & Evaluation

Contents

5.1 Testing Criteria

The complete chessboard system—encompassing RFID-based piece detection, NeoPixel lighting, ESP32-to-app Bluetooth communication, and H-Bot piece movement has been verified to support full game tracking, hover highlighting, Lichess integration, local player-vs-Stockfish matches, and online 1v1 play. What follows are targeted performance testing of its critical subsystems.

We test our system along four key dimensions:

- **Detection Accuracy and Range:** Correctly reading each piece's UID on the targeted square.
- **Scan Latency:** Time to activate one reader, detect (or timeout), and deactivate.

- **Movement Performance:** Total time and positioning precision when executing a move.
- **Feedback Responsiveness:** Latency of Bluetooth transmission.

5.1.1 Detection Accuracy and Range

Testing Method

- *Setup:* Position a tagged chess piece directly above the reader at a reader at vertical offsets of 0 mm, 2 mm, 4 mm, 6 mm, 8mm, 10 mm, and 12 mm.
- *Trials:* At each distance, perform 100 consecutive read attempts (powering on the reader, issuing a request, and checking for UID response).
- *Metrics:* Record the number of successful UID reads versus total attempts to compute a per-distance detection success rate.

Results

Distance (mm)	Success Rate (%)
0	95.2
2	99.8
4	99.5
6	99.2
8	98.9
10	85.0
12	0.0

Table 5.1: RFID detection success as a function of distance from the reader

At 0 mm, the lower success rate (95.2 %) is likely due to the inherent “dead zone” of the RC522 antenna when a tag is pressed directly against it. In our design, we use an 8 mm separation (PCB plus diffusing acrylic), which yields a 98.9 % read rate. This distance provides sufficient clearance to avoid the dead zone, allows the NeoPixel light to diffuse evenly through the playing surface, and prevents accidental detections when a player’s hand briefly hovers over a square. The 98.9 % reliability at 8 mm therefore represents a strong compromise between optical diffusion and robust piece detection.

5.1.2 Sequential Scan with Shift Register

Testing Method

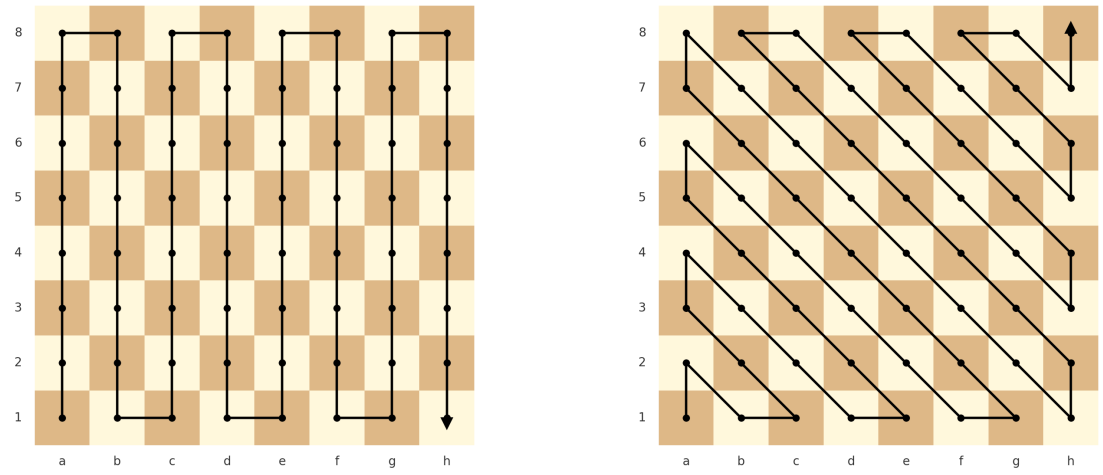
- *Setup:* Eight RC522 readers were daisy-chained via the shift register on a single row. A tagged chess piece was placed on each of the eight reader locations (and, in a separate trial, no pieces were present).
- *Procedure:* The ESP32 firmware walked the "1" bit through the shift register to activate each reader in turn, timing each read from the moment the shift register was cleared and the next bit loaded until a UID response or timeout was received.
- *Trials:* 100 full-row scans for each condition (800 activations per condition).
- *Metrics:* Recorded per-reader latency (shift + read) and computed full-row scan time.

Results

- **No tags:** Mean per-reader latency 8.0 ms (exact same across all 800 activations)
- **Tags on all 8 readers:** Mean per-reader latency 4.4 ms (SD 0.5 ms)
- **Full-row scan time:**
  - No tags: 8 readers × 8.0 ms = 64.0 ms
  - With tags: 8 readers × 4.4 ms = 35.2 ms
- **Error rate:** 0.1875 % (3 failures out of 1600 activations)

Even when cycling through eight readers, the system completes a full-row scan in under 65 ms without tags and under 40 ms with tags—well within our 1 s full-board requirement—and achieves very high reliability in both scenarios.

5.1.3 Movement Performance



(a) Linear movement path along the first rank: squares `a1` through `h1`.

(b) Zigzag traversal path from `a1` to `h8`, visiting each square in turn.

Figure 5.1: Movement paths used for performance testing: (a) linear along the first rank; (b) zigzag across the board.

Testing Method

- *Paths:*
  1. **Linear:** Move sequentially from `a1` to `h1`, one square at a time.
  2. **Zigzag:** Traverse the board in a raster-like zigzag pattern from `a1` up to `h8`, moving to each adjacent square.
- *Procedure:* For each square-to-square transition along both paths, the ESP32 firmware recorded timestamps at magnet engage and at piece release. A total of 128 moves were logged across each path.
- *Metrics:* Computed the average time per square-to-square move and its standard deviation for each path.
- *Positioning Accuracy:* After each move in both tests, measured final piece center vs. target center with digital calipers.

Results

- **Linear path (a1 to h1):**

  - Average time per square: 1.1 s
  - Standard deviation: 0.1 s

- **Zigzag path (a1 to h8):**
  - Average time per square: 1.2 s
  - Standard deviation: 0.2 s

- **Positioning error:** Mean 0.5 mm (± 0.2 mm) for both tests.

5.1.4 Feedback Responsiveness

Testing Method

- *Bluetooth Transmission Test:* Two separate 100-message streams were sent:
  - **ESP32 to App:** Messages sent at 0.5 s, 1.5 s, …, 99.5 s.
  - **App to ESP32:** Messages sent at 1.0 s, 2.0 s, …, 100.0 s.

  Each message included a millisecond-resolution timestamp. Both endpoints logged local send and receive times.

- *Metrics:* For each stream, one-way latency was computed as:

$$\text{latency} = t_{\text{receive}} - t_{\text{send}}$$

  The number of messages successfully received was also recorded.

Results

- **ESP32 to App:** 100 / 100 messages received; average latency 26 ms (SD 4 ms), computed as `app_receive_timestamp - esp32_send_timestamp`.
- **App to ESP32:** 100 / 100 messages received; average latency 24 ms (SD 3 ms), computed as `esp32_receive_timestamp - app_send_timestamp`.

5.2 Evaluation

5.2.1 Evaluation on Project requirments

Performance Requirements

- **Scanning Speed:** The requirement to scan the full 64-square board in under one second was successfully met. The RFID scanning loop implemented on the ESP32 is highly efficient, allowing it to iterate through all readers well within this time frame. This performance satisfies the Nyquist sampling rate criterion outlined in the initial requirements, ensuring that even the fastest physical moves can be reliably detected.
- **Robustness and Accuracy:** The requirement for an error-free and accurate game state was a primary focus. This was achieved through the implementation of the request/ACK handshake protocol between the ESP32 and the companion app. By making the app's chess engine the "master" authority on the game state and preventing the ESP32 from updating its own state until a move is validated, the system is highly robust against desynchronization from illegal moves or hardware misreads.

State Tracking Requirements

All of the detailed state tracking requirements were fully met. This was a direct result of the key architectural decision to offload all chess logic to the companion application. The integrated `chess.dart` library capably manages:

- Piece positions, captured pieces, turn tracking, and the move log.
- Complex rule-based states, including castling rights, en passant status, check status, threefold repetition history, and promotion tracking.
- Game timers and final game results (checkmate, stalemate, etc.).

By leveraging a dedicated, well-tested chess library, the project successfully fulfilled all state-tracking requirements without needing to implement this complex logic from scratch on the resource-constrained ESP32.

Feature Evaluation: Game Modes

The implementation successfully delivered on all specified game modes:

- **Single Player (Player vs. AI):** Met. This functionality is provided through the Lichess API integration, allowing the user to play against Lichess's AI bots. The system successfully receives the bot's moves and actuates them on the physical board.
- **Local Multiplayer (Player vs. Player):** Met. The system accurately tracks a two-player game on the board and successfully implements the feature to upload the final PGN to Lichess for analysis.
- **Online Multiplayer:** Met. Full integration with the Lichess API allows users to play against friends or random opponents online.
- **Board-to-Board Play:** Met. This feature is inherently supported through the Lichess integration. If two users who both own the smart chessboard play each other on Lichess, their physical boards will seamlessly mirror each other's moves as a standard online game.

Feature Evaluation: Lighting System

The lighting system was a key component of the user experience, and most of its specified features were successfully implemented.

- **Legal Moves Indication:** Met. When a player lifts a piece, the board correctly illuminates all valid destination squares, providing intuitive feedback.
- **Capture Indication:** Met. The requirement to highlight a capture was implemented by illuminating the destination square in red during a hover event, which clearly distinguishes a capture from a regular move.

5.2.2 Project as a Technology Demonstrator

Beyond its primary function as a smart chessboard, this project serves as an effective **technology demonstrator** for prospective undergraduate students. It provides a tangible example of how various theoretical and practical disciplines within engineering and computer science converge to create a complex, real-world product. It showcases how individual courses contribute to a larger, integrated system.

Software, Programming, Embedded and AI

The companion application and online integration demonstrate key software engineering principles.

- **Programming for Engineers:** This foundational course, covering programming fundamentals and data structures, is the basis for the entire project. The smart chessboard is a massive software engineering undertaking, showcasing programming at two distinct levels:
  - **Embedded Firmware:** The logic on the ESP32 is written in C++, involving low-level hardware control, real-time data processing, and custom data structure implementation (such as the BiMap).
  - **Application Development:** The companion mobile app is a large-scale project built in Dart using the Flutter framework, involving complex UI/UX design, state management, asynchronous programming, and interfacing with external web services.
- **Software Systems:** This course, which covers the internet and networking stacks, is directly demonstrated by the application's interaction with the Lichess web service. The system uses **HTTPS requests**, built upon the TCP/IP stack, to communicate with the Lichess API for user authentication, game creation, and state synchronization. This is a practical implementation of a client-server architecture over the internet.
- **Embedded Systems:** It involves the integration of sensors (RFID readers), actuators (motors, LEDs), and communication interfaces (BLE, UART) with a microcontroller. The firmware development had to manage real-time constraints, such as the precise timing required for the NeoPixel data protocol, while simultaneously handling multiple concurrent tasks like board scanning and serial communication. The use of two distinct microcontrollers to delegate tasks (main logic vs. motor control) is a common design pattern in more complex embedded systems.
- **Machine Learning, Deep Learning:** While not implemented locally, the integration with Lichess allows users to play against powerful AI bots, which are themselves prime examples of machine learning and deep learning algorithms applied to game theory. The board provides a physical interface to interact with these advanced systems.

Electronics and Circuit Design

- **Analysis and Design of Circuits, Electronics Design Project:** The design of the custom PCBs for the 8x8 grid of RFID readers and NeoPixel LEDs is a hands-on electronics design project. It required consideration of power distribution, signal integrity, and component layout.

Control, Robotics, and Communication Networks

The physical movement of the pieces is a direct application of robotics and control theory.

- **Control Systems, Robotic Manipulation:** The H-Bot gantry is a 2-axis robotic system. The process of sending G-code commands and having the GRBL firmware translate them into precise stepper motor movements is a core concept in control engineering and robotic manipulation.
- **Communication Networks:** This course is demonstrated by the project's wireless hardware. Both the **RFID readers** and the **Bluetooth Low Energy (BLE)** module operate using radio frequency (RF) waves. Their performance relies on core communication network concepts such as signal modulation (how data is encoded onto the waves), antenna design, signal-to-noise ratio (SNR), and mitigating interference—all fundamental topics in this field.

6

Conclusion

In conclusion, this project successfully achieved its goal of creating a fully functional smart chessboard that seamlessly integrates physical and digital gameplay. Through a robust architecture that offloads complex chess logic to a companion mobile application, the system provides a feature-rich user experience, including local and online multiplayer via the Lichess API, as well as physical piece actuation. The final prototype stands as a successful demonstration of how mechanical, embedded, and software engineering disciplines can be combined to create a complex, interactive product. The project delivered on all primary requirements, validating the core design decisions made throughout its development. The full source code and design files are available at https://github.com/elkouny/FYP.

7

Reflection & Further Work

Contents

7.1 Self Reflection

This project has been an extensive and deeply rewarding journey, providing practical experience that extends far beyond any single academic course. The greatest lesson learned was in the challenge of **system integration**. While individual courses provide deep knowledge in specific domains, this project demanded that mechanical design, embedded firmware, mobile application development, and web API communication be woven into a single, functional system. Seeing a physical move on the board trigger a validated API call, or an online move manifest as a physical piece moving across the board, was a powerful demonstration of full-stack product development in action.

The process also highlighted the importance of design flexibility. Initial ambitions, such as running a full chess engine on the ESP32, were tempered by the reality of hardware limitations.

The decision to offload this logic to a companion app was a critical lesson in architectural trade-offs. Similarly, the choice to use a powerful Neodymium magnet combined with a mechanical release mechanism—rather than an electromagnet with insufficient range—was a practical solution to a real-world physics problem.

Ultimately, this project was a transformative experience. It required the hands-on application of a wide range of skills, from CAD modeling and 3D printing to low-level embedded programming in C++ and high-level mobile development in Dart. It served as a tangible bridge between the theoretical concepts learned in courses like Control Systems, Embedded Systems, and Software Systems and the discipline required to bring a complex engineering vision to life. I emerged from this project not only with a diverse new skill set but with a much deeper appreciation for the iterative process of design, testing, and creative problem-solving.

7

7.2 Further Work and Future Improvements

While the project successfully meets its primary requirements, the current prototype exhibits some intermittent reliability issues that could be addressed in future revisions. Further work would focus on fine-tuning the system's performance and robustness.

7.2.1 Improving Sensor Reading Reliability

A key area for improvement is the consistency of RFID piece detection.

- **The Issue:** Occasionally, a stationary piece is momentarily not detected by its RFID reader during a scan cycle. The system incorrectly interprets this "blip" as a `hover` event and sends a corresponding message to the app. This can cause a flickering effect on the lighting system, as a false hover event may interfere with a legitimate one when the user is actually moving a piece.
- **Proposed Solution:** A software-based debouncing or filtering algorithm could be implemented in the ESP32 firmware. Instead of triggering a hover event on a single missed read, the system would require the piece to be undetected for two or three consecutive scan cycles before confirming the event. This would filter out momentary sensor misreads while still being fast enough to respond to genuine user actions. Further hardware tuning of the RFID reader's antenna gain could also improve detection consistency.

7.2.2 Implementing Autonomous Knight Movement

A significant enhancement would be to fully automate the movement of a Knight, even when its path is physically blocked by other pieces.

- **The Challenge:** The current implementation prompts the user to manually clear the path for a blocked Knight. Since the under-board gantry cannot ”jump” over pieces, a fully autonomous solution requires a sophisticated pathfinding and piece-shuffling algorithm.
- **Proposed Solution:** An algorithm needs to be developed that can:
  1. Identify the specific pieces obstructing the Knight's path.
  2. Find a nearby empty ”parking” square for each obstructing piece.
  3. Execute a sequence of G-code commands to temporarily move the obstructing piece(s) to these parking squares.
  4. Move the Knight to its final destination.
  5. Finally, move the obstructing piece(s) from their parking squares back to their original positions.

  Implementing this multi-step process would represent a major step towards a fully autonomous system.

7.2.3 Refining Mechanical Actuation

The physical movement of the pieces via the H-Bot is functional but could be made more reliable with further tuning.

- **The Issues:** Two issues are occasionally observed:
  1. The movement systems magnetic field sometimes latches onto an adjacent piece in addition to the target piece.
  2. The grip is sometimes insufficient to move a piece cleanly across its full path, causing it to be dropped or left short of its destination square.
- **Proposed Solutions:** These issues could be addressed through a combination of mechanical and software fine-tuning:

- **Magnetic Field Shielding:** Experimenting with a small ”mu-metal” or soft iron shield around the magnet could help focus its magnetic field directly upwards, reducing its horizontal spread and minimizing the chance of attracting nearby pieces.
- **Motion Profile Tuning:** The G-code generation could be refined. Adjusting the acceleration and feed rate (`F`) parameters for moves might provide a smoother motion. Additionally, programming a small ”settling” delay or a slight ”wiggle” motion after the magnet engages could ensure a more secure grip before the main travel begins.

7.3 IET Engineering Competencies Reflection

7.3.1 Sustainability and Environmental Considerations

The project design incorporated principles of sustainability. The choice of modular, row-based PCBs enhances repairability and reduces waste, as a single faulty row can be replaced without discarding the entire board. Furthermore, the selection of Bluetooth Low Energy (BLE) for communication was a conscious decision to minimize the power consumption of the companion mobile app, a key consideration for sustainable design in battery-powered devices.

7.3.2 Communication and Interpersonal Skills

While an individual project, the competency of communication was demonstrated through technical documentation and the design of system interfaces.

- **Technical Documentation:** The creation of this report, complete with structured explanations, diagrams (state machines, sequence diagrams), and clear justifications for design choices, demonstrates the ability to communicate complex technical information effectively.
- **System Communication Design:** A significant part of the project was designing the communication protocols between the system’s components. This included defining the structure of the BLE messages between the ESP32 and the app, implementing the request/ACK handshake protocol for state synchronization, and using the established G-code standard for communication between the ESP32 and the GRBL controller.

Bibliography

[1] H. J. R. Murray, "A history of chess," *Oxford University Press*, 1913.

[2] F.-h. Hsu, "Behind deep blue: Building the computer that defeated the world chess champion," *Princeton University Press*, 2002.

[3] ChessGame.com, "https://www.chessgames.com/chessstats.html", Statistics of chess games collected from the year 2000 - 2025, over 1,770,817 games recorded Accessed: 2025-01-13.

[4] chess.stackexchange.com, https://chess.stackexchange.com/questions/39194/whats-the-average-total-time-for-a-game-of-bullet-chess, Average total time for bullet chess calculated by L.Scott Johnson based on a sample of 27,398,824 games played on LiChess.com during January of 2022 Accessed: 2025-01-13.

[5] G. Lazaridis, https://pcbheaven.com/wikipages/How_Key_Matrices_Works/, What are the key matrices? By Giorgos Lazaridis. Accessed: 2025-01-13.

[6] @thingsfromtheinternet4430, https://www.youtube.com/watch?v=j7Iq3HkebhM, Video of a customer that bought the Miko chess board, it broke so he opened it to see whats inside it Accessed: 2025-01-13.

[7] O. Mercier, https://www.youtube.com/watch?v=nGR9oMMSW_o, A chess board with an effective tracking system utilizing hall effect sensors. Accessed: 2025-01-13.

[8] asdfjkl, https://www.youtube.com/watch?v=WxEr-5x00cQ, A chess board with an effective tracking system reed switches. Accessed: 2025-01-13.

[9] B. J. Bulsink, https://patents.google.com/patent/US6168158/en, Device for detecting playing pieces on a board. Accessed: 2025-01-13.

[10] Chessify, https://chessify.me/blog/new-technology-for-3d-chess-board-digitalization, New Technology For 3D Chess Board Digitalization. Accessed: 2025-01-13.

[11] DGT, https://digitalgametechnology.com/about-us, DGT story. Accessed: 2025-01-13.

[12] Microchip, https://ww1.microchip.com/downloads/en/devicedoc/51115f.pdf, microID® 125 kHz RFID System Design Guide. Accessed: 2025-01-13.

[13] Microchip, https://ww1.microchip.com/downloads/en/devicedoc/51115f.pdf, microID® 125 kHz RFID System Design Guide. Accessed: 2025-01-13.

[14] W.-S. L. H.-S. J. K.-S. O. J.-W. Yu, https://ietresearch.onlinelibrary.wiley.com/doi/full/10.1049/el.2013.2213, Near-field switched loop antenna array with circular defected ground structure for precise positioning systems. Accessed: 2025-01-13.

[15] E. Easy, https://espeasy.readthedocs.io/en/latest/Plugin/P131.html, Display - NeoPixel Matrix Accessed: 2025-05-25.

[16] Arduino, https://docs.arduino.cc/resources/datasheets/ABX00083-datasheet.pdf, Arduino® Nano ESP32 Datasheet, see section 6.3 on BLE Accessed: 2025-05-25.

[17] etly, https://elty.pl/upload/download/RFID/RDM630-Spec.pdf, RDM6300 Datasheet. Accessed: 2025-01-13.

[18] L. M. Engineers, https://lastminuteengineers.com/how-rfid-works-rc522-arduino-tutorial/, How rfid works rc522 arduino tutorial Accessed: 2025-01-13.

[19] NXP, https://www.nxp.com/docs/en/data-sheet/MF1S50YYX_V1.pdf, MIFARE Classic EV1 1K Datasheet Accessed: 2025-01-13.

[20] ProdigyTechno, https://www.prodigytechno.com/i2c-vs-spi, I2C vs SPI Accessed: 2025-01-13.

[21] T. Intrsuments, https://www.ti.com/lit/ds/symlink/sn54hc164.pdf?ts=1748366386342, SNx4HC164 8-Bit Parallel-Out Serial Shift Registers Accessed: 2025-05-25.

[22] DIYMachines, https://www.diymachines.co.uk/kinetic-sand-art-coffee-table-self-drawing, Kinetic sand art coffee table self drawing DIY tutorial. Accessed: 2025-05-25.

[23] ParasitKegel, https://www.printables.com/model/417138-chess-pieces/files, Chess Pieces 3D models. Accessed: 2025-05-25.