Rook, Ma, No Hands!

EE327
FINAL REPORT

Northwestern University

Adheik Dominic
Matthew Flais
Jason Paul
Louis Wong

June 8, 2023

TABLE OF CONTENTS:

I. Abstract

The proposed project, "Rook, Ma, No Hands!", introduces a smart chess board with an automated piece movement system, piece sensing subsystem, and the ability to play games against an AI opponent. This project aims to elevate the game of chess by automating the movement of pieces made by an AI opponent, bringing convenience and excitement to players, while allowing players to continue playing on a traditional board surface. The smart chess board incorporates two ESP32 microcontrollers: one controls the flow of game logic, manages communications with our external chess engine, and calculates piece movements, while the second controls the sensing subsystem in charge of determining piece positions when a piece is moved by the player. It features a user-friendly interface with minimal push buttons, making it accessible to players of all skill levels, as well as a traditional chess clock system.

The sensing subsystem utilizes Hall effect sensors and magnets embedded in each piece to detect their positions on the board, and sends move data to the main processor over a UART bus. The motion (gantry) system employs stepper motors, linear bearings, pulleys, and belts for smooth and precise movement of the pieces. The design ensures safety and reliability, preventing mechanical errors or collisions during gameplay.

The decision to pursue the development of Rook, Ma, No Hands! stemmed from the desire to enhance the overall chess playing experience of playing an AI opponent in real time and in person. By automating piece movement, players can focus more on strategy and gameplay. Furthermore, in future integration, it becomes easily possible to write additional firmware which allows for the extension of the automation feature, where two players with boards across the world would be able to play with each other. The project also provides a valuable learning experience in integrating technologies such as sensors, motion systems, and controllers. Ultimately, the smart chess board has the potential to revolutionize chess gameplay and introduce innovation to the game.

II. Introduction

Welcome to the world of "Rook, Ma, No Hands" - the smart chess board that redefines the way we play chess. This innovative project combines automation and advanced sensing technology to create a chess board that moves the pieces for you, revolutionizing the traditional chess-playing experience.

Our motivation behind developing the smart chess board was to introduce convenience, excitement, and accessibility to chess enthusiasts of all levels. By incorporating an automated piece movement system and a sophisticated sensing subsystem, players can now enjoy the game without the hassle of physically moving the pieces. With just a touch of a button, the board effortlessly detects and moves the pieces, leaving players free to focus on strategy and immersion in the game.

At the core of this intelligent system lies the ESP32, acting as the brain that orchestrates the coordination between the motion and sensing subsystems. This allows for accurate positioning and seamless movement of the chess pieces across the board. The user-friendly interface, equipped with intuitive pushbuttons and informative LED indicators, ensures that players of all skill levels can easily engage with the smart chess board.

Not only does "Rook, Ma, No Hands" prioritize convenience, but it also embraces cutting-edge features that enhance the overall playing experience. Through the utilization of Hall effect sensors and magnets embedded within each piece, the sensing subsystem ensures precise detection of piece positions on the board at all times. The motion system, driven by stepper motors, linear bearings, pulleys, and belts, enables smooth and accurate piece movement. Safety mechanisms have been incorporated to prevent mechanical errors or collisions, guaranteeing a secure and reliable gameplay environment.

Our decision to embark on the development of this smart chess board was driven by our passion for chess and a desire to introduce innovation to the game. By automating the movement of the pieces, we aim to elevate the chess-playing experience, freeing players from the mundane task of physically rearranging the pieces and allowing them to fully immerse themselves in the strategic intricacies of the game. Furthermore, this project provides us with an opportunity to explore and integrate various technologies, such as CNC systems and H-style gantries, while honing our skills and knowledge in sensor integration, motion control, and advanced algorithms.

With "Rook, Ma, No Hands," we envision a future where chess enthusiasts can engage with the game in a more immersive and convenient manner. Our smart chess board opens new possibilities, making chess accessible to a wider audience and paving the way for a transformative gaming experience. Join us on this exciting journey as we unveil the capabilities and potential of our innovative creation.

III. Design Description

III.a System Overview and Block Diagram

The design of the automated chess system contains the scheme as detailed in Figure 1. The main ESP32 processor oversees the entirety of the chessboard and is in charge of communications with the second ESP32 (sensing subsystem) and the AWS instance holding the chess engine. The ESP32 then factors this communication in with a structural game logic described in the Algorithms subsection below. The second ESP32 houses the firmware for determining the movement of the chess pieces and updates the main ESP32 when there is a confirmed movement of pieces on the board. Both of the processors are then packaged into a chessboard, with the hall effect sensors hidden by the top surface of the board. In the following subsections, the specific design details of each subsystem are explained.

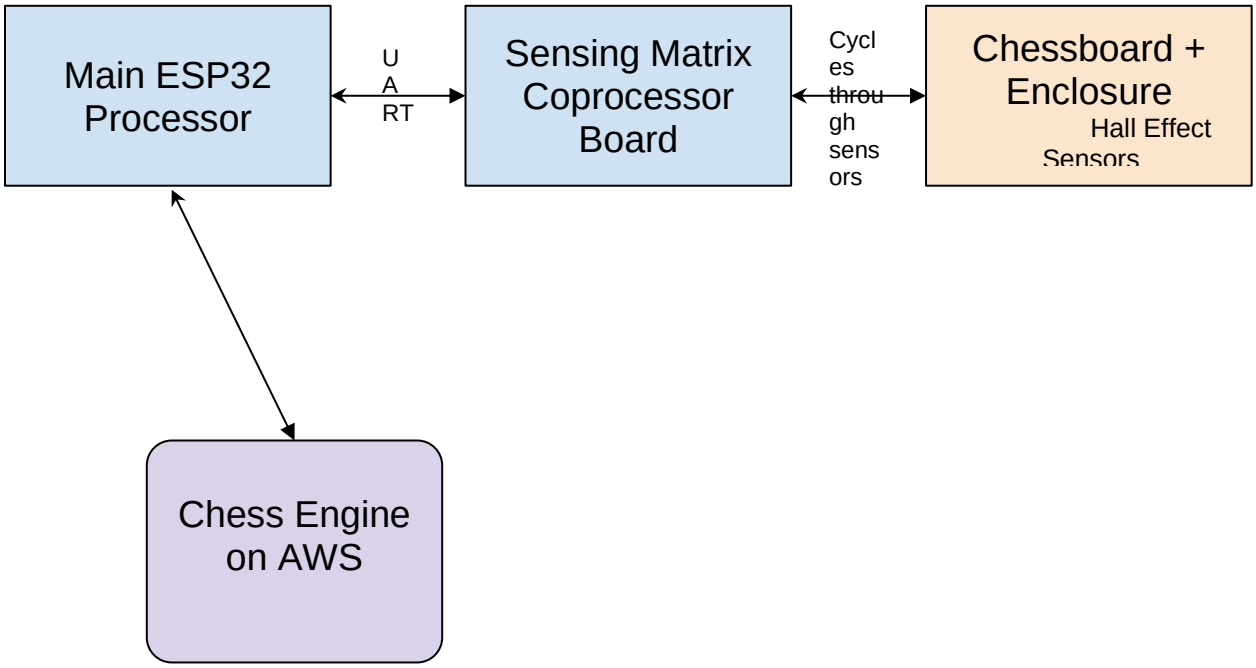

Fig. **1.** High-level block diagram of the system

III.b Sensing Subsystem

Our chessboard's sensing system has two main components: the ESP32 coprocessor board and the Hall effect sensor matrix.

The sensing matrix is composed of 16 identical PCBs, each of which houses four total omnipolar Hall effect sensors, enough to cover every board space. The VDD inputs of the horizontally aligned sensors are all connected, and the outputs of the vertically aligned sensors are all connected - with diode protection included. Each PCB has pads on each edge on both sides to allow the boards to be connected edge-to-edge, as seen below:

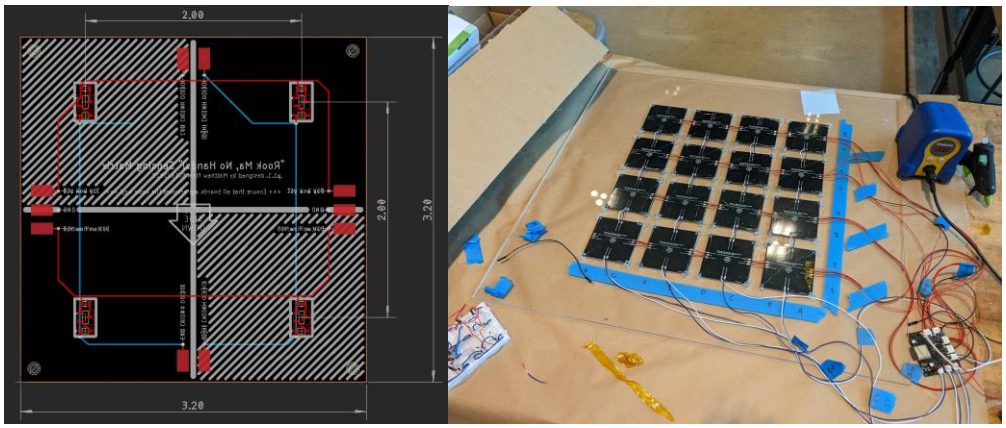

Fig. **2**. Sensor matrix PCB layout (left) and assembled sensor matrix (right)

With this configuration of sensors, each row of sensors can be powered one at a time, and each column's output can be read individually; by cycling through powering each row one at a time, an observing microcontroller can quickly calculate the status of each place on the board.

This microcontroller, an ESP32, is housed on its own PCB, which contains an 8-bit multiplexer, a voltage regulator to drop a 5V input to 3.0V to power each device, header pins that connect to GND, 5V, two UART buses, the EN pin of the ESP32, a COM pin that is set to low by the main board when it requests a move from the sensing system, and an ENDMOVE pin that is pulled to ground by a player-pressed button when they complete their move. The board contains buttons for both EN and ENDMOVE, as well as connectors to attach to the sensor matrix.

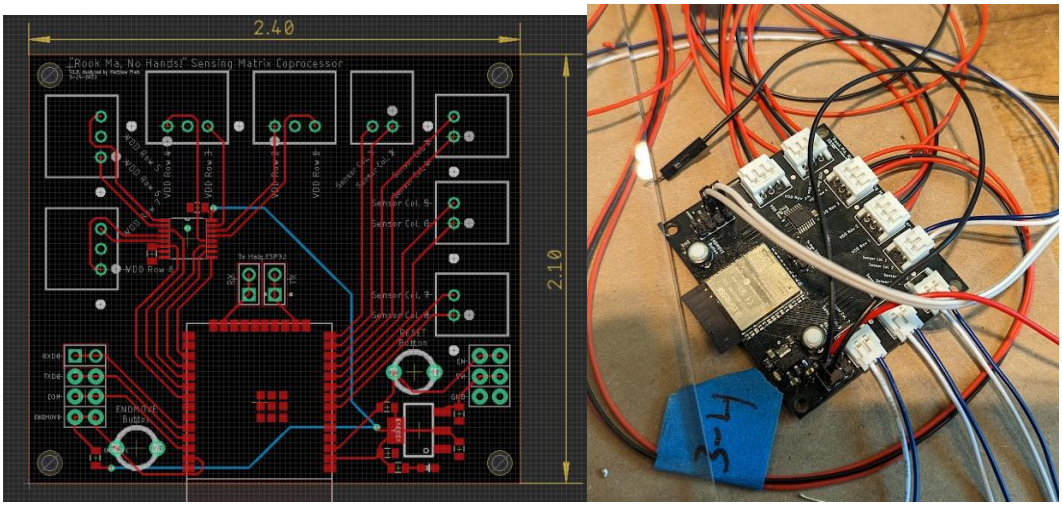

Fig. **3**. Sensing coprocessor PCB (left) and completed board (right)

The firmware for this board has a few simple steps. When it detects that the COM pin has been set to low, it begins cycling through supplying 3.3V to each row of sensors and reading the values outputted by each column of the board. If it detects that a position has changed in value since the last time it was read, it adds this change - noting the position and direction of the change

- to a list tracking all of these state changes. Once the ENDMOVE pin is set to low, this move list is taken, cleaned up to remove any unwanted lift-place indecision on the part of the player, and used to calculate what type of move was made. This move is then written to the main ESP32's UART bus.

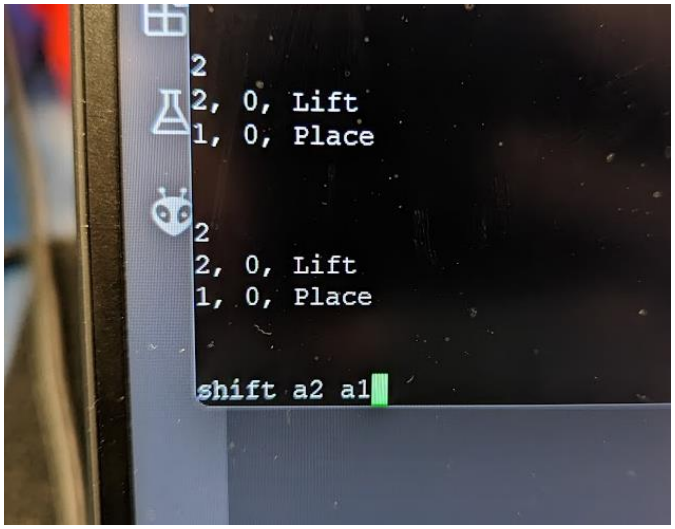

Fig. **4**. An example message sent for a piece moving from A2 to A1

After sending a move to the main board, the processor turns off the sensors to save power and waits for the main board to request another move by setting the COM pin to low.

III.c Gantry

The gantry is constructed using a H-bot style x/y mechanism. It carries the main electromagnet and communicates to the main processor using G-code over UART. The H-bot design allows both motors to remain stationary and keep from taxing the motors during operation by lightning the load. Nearly all of the gantry is hand built. The main parts were machined out of aluminum and the bearings machined in house as well. The ways were constructed from ground hardened shaft to keep the gantry moving square with itself. The DRV8825 stepper divers were selected to provide power to the steepers and a large 12V AC/DC converter provides power to both the motors and the electromagnet. The ESP32 processor runs FluidNC a open-source firmware for CNC machines. Custom kinematics and macros were written to suit our particular application.

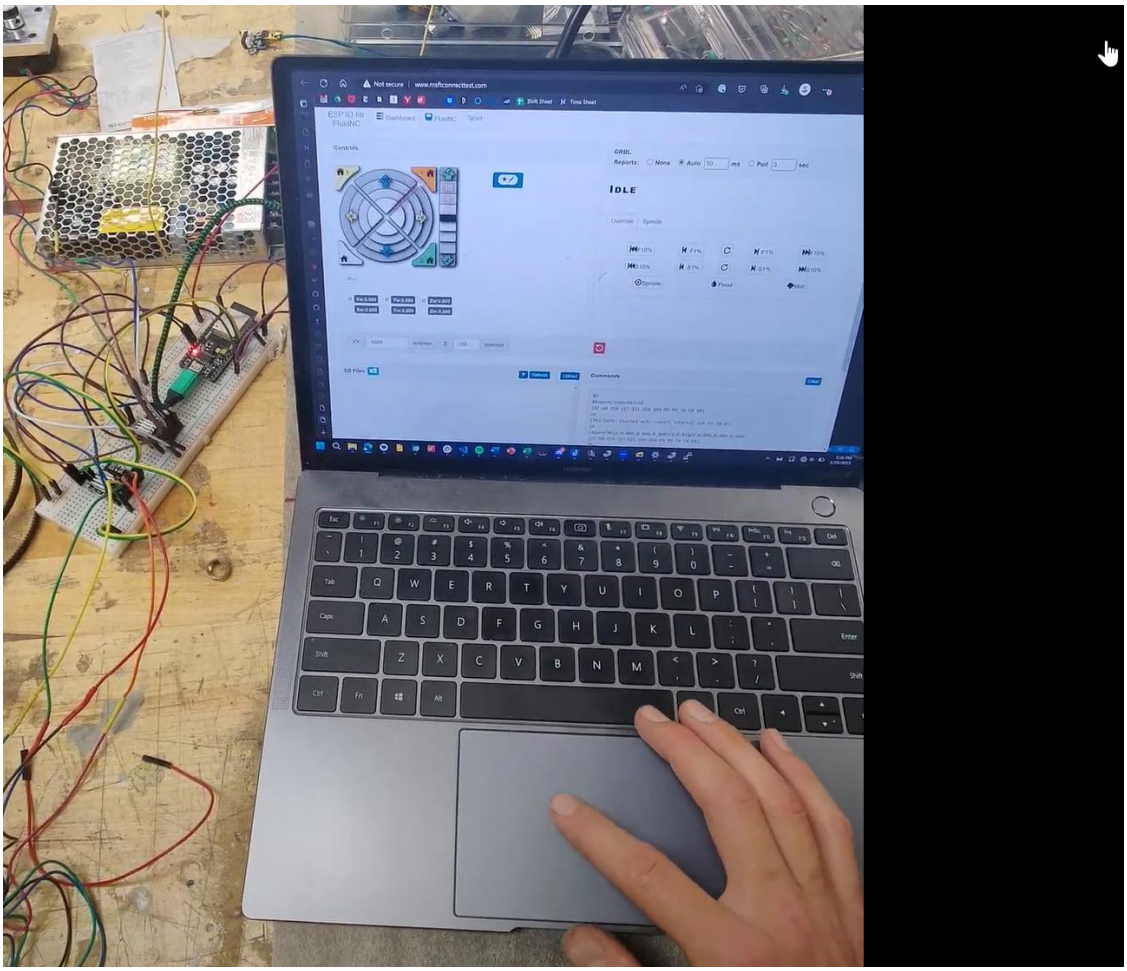

Fig. **5**: Web GUI and Control electronics for Gantry System

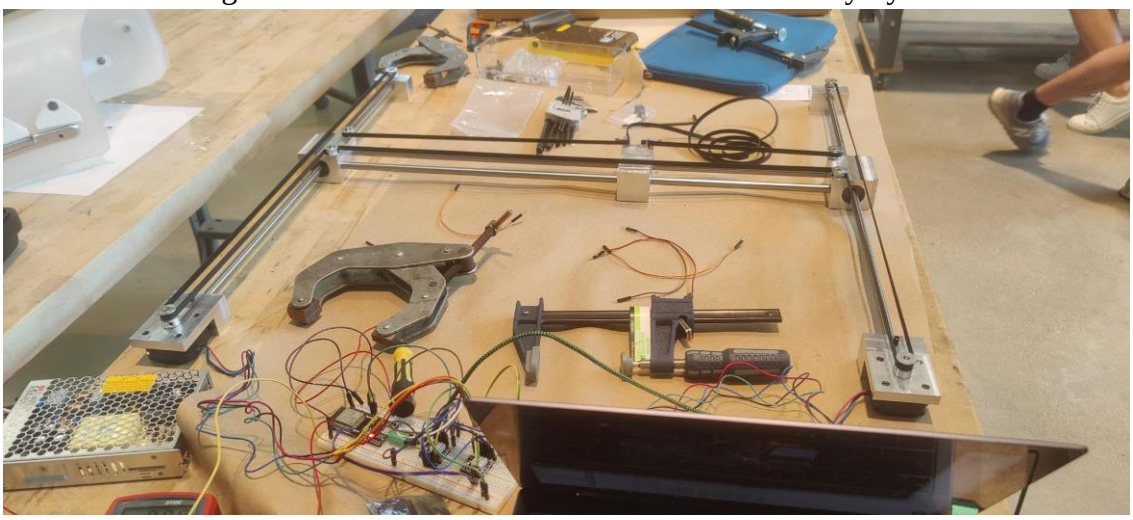

Fig. **6**: Gantry System assembled for testing.

III.d 3D Printing

The 3D printing component of this project consisted of the printing of the chess pieces used which were designed to hold a magnet at the bottom to allow the pieces to be moved by the electromagnet under the board. The figure below shows the final design of the chess pieces in the

CAD software Onshape. The CAD design only required six pieces to be designed, as the 64 chess pieces are only made of seven unique pieces, including the rook, king, queen, knight, bishops, and pawns. The right hand side of the figure then shows the adjustment made to the pieces which have a hole for gluing the magnets into the chess pieces. An example print of the pieces are shown in the accompanying figure below. The chess pieces were 3D printed using an AnkerMake FDM extrusion-based 3D printer, with the material being common PLA. The onshape link is additionally provided in Appendix A with the gantry system design.

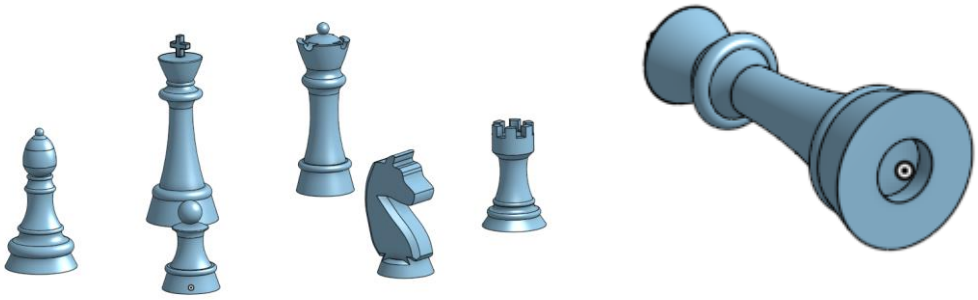

Fig. **7**. CAD of chess pieces (left) and the hole at the bottom of the piece (right)

Fig. **8**. Example set of 3D printed chess pieces

III.e Algorithms and Code

Our game logic code sets up a chess game using the Python Chess library and integrates it with G-code commands. It initializes the chessboard, display, and engine (e.g., Stockfish) for gameplay. The input pins are configured to receive commands for moving chess pieces.The gcode_move() function processes the G-code move commands by creating a move object, validating it, and executing it on the chessboard. It checks for game over conditions, updates the

display with the new board position, and allows the engine to calculate its move. Once again, it checks for game over conditions and updates the display accordingly. The process_gcode() function handles the G-code commands, distinguishing between move and set position commands. However, the specific implementation of extracting the source and target coordinates from the G-code commands which were written up in the g-code file.The main game loop is set to run indefinitely, allowing for continuous gameplay until interrupted. The engine.quit() statement ensures that the chess engine is properly closed and cleaned up when the game loop is exited. Overall, the code provides the basic structure for integrating chess gameplay with G-code commands. However, certain portions, such as the G-code command processing and the extraction of source and target coordinates, need to be completed for the code to function properly.

In order to create the AI bot for the board, the deployment of a chess engine becomes necessary. However, because the ESP32 does not house enough memory to hold programs like Stockfish and AlphaZero, a proxy is necessary for holding the software and code containing the chess engine. AWS was chosen as a proxy, as it allows for the facilitation of communication between a "serverless" instance and the main ESP32 used for logic. The instance has enough memory to hold the Stockfish executable which was used and the main python file used to subscribe the topic in AWS was implemented in a Ubuntu AWS EC2 instance. There are two main parts to the code in Figures 11 to 14: (1) connecting and subscribing to the topic for communication with the ESP32 (2) taking the input of the player and generating a corresponding move from a chess engine analysis. The scheme for AWS communication is as follows. AWS allows for a "Thing" to be created and a topic to be enabled within the "Thing." Both the EPS and AWS EC2 instance (essentially a virtual machine) subscribe to the same topic which allows them to read all updates sent to the topic and additionally are both given publishing permission, allowing for both the ESP and EC2 instance to write the moves by the player or the AI. The ESP32 will push player moves, which in turn is read by the EC2 instance, then the EC2 replies with the engine move. The implementation of the code used the python **chess** library, allowing the Stockfish executable to be loaded and ready for full compatibility with our code. The advantage of using a robust program like Stockfish allows for the player to play against a top level AI engine while also having the possibility of learning from Stockfish analysis if further implementation of the code is completed. A further implementation of accessing the code within the AWS instance is shown in Figure 14 below.

```
import chess
import chess.svg
import chess.engine
import chess.pgn
from machine import Pin
import chess.display

# Initialize the chessboard
board = chess.Board()

# Initialize the display
display = chess.display.Board()

# Initialize the engine (e.g., Stockfish)
engine = chess.engine.SimpleEngine.popen_uci("/path/to/stockfish")

# Set up the input pins for the chessboard (example: pins D1 to D8 for columns and A1 to A8 for rows)
columns = [Pin(Pin.D1, Pin.IN), Pin(Pin.D2, Pin.IN), Pin(Pin.D3, Pin.IN), Pin(Pin.D4, Pin.IN),
           Pin(Pin.D5, Pin.IN), Pin(Pin.D6, Pin.IN), Pin(Pin.D7, Pin.IN), Pin(Pin.D8, Pin.IN)]
rows = [Pin(Pin.A1, Pin.IN), Pin(Pin.A2, Pin.IN), Pin(Pin.A3, Pin.IN), Pin(Pin.A4, Pin.IN),
        Pin(Pin.A5, Pin.IN), Pin(Pin.A6, Pin.IN), Pin(Pin.A7, Pin.IN), Pin(Pin.A8, Pin.IN)]

# Define G-code commands and their actions
def gcode_move(source, target):
    # Create the move object
    move = chess.Move(source, target)

    # Validate and execute the move
    if move in board.legal_moves:
        board.push(move)

        # Check for game over conditions
        if board.is_game_over():
            result = board.result()
            print("Game over. Result: " + result)
            return
```

Fig. **9**. Game logic code

```
M62 P0 ; Magnet ON
G1 X15 Y15 ; Move to the center of square A1
M63 P0 ; Magnet OFF

M62 P0 ; Magnet ON
G1 X45 Y15 F100; Move to the center of square B1
M63 P0 ; Magnet OFF

M62 P0 ; Magnet ON
G1 X75 Y15 F100; Move to the center of square C1
M63 P0 ; Magnet OFF

M62 P0 ; Magnet ON
G0 X75  Y15 ; center of C1
G1 X105 Y15 F100; Move to the center of square D1
M63 P0 ; Magnet OFF
```

Fig. **10**. G-Code commands

```
# Import Libraries
import chess
import chess.engine
import boto3
import paramiko
import json
import os
```

Fig **11**. Imported Libraries for AWS instance

```
# Specify the EC2 instance ID
ec2_instance_id = 'lichess_server'

# Specify the AWS IoT Core topic to subscribe to
iot_topic = 'your/iot/topic'

# Retrieve the public IP address of the EC2 instance
ec2_client = boto3.client('ec2', region_name=region_name,
                          aws_access_key_id=aws_access_key_id,
                          aws_secret_access_key=aws_secret_access_key)

response = ec2_client.describe_instances(InstanceIds=[ec2_instance_id])
public_ip_address = response['Reservations'][0]['Instances'][0]['PublicIpAddress']

# Create an AWS IoT policy for the EC2 instance
policy_name = 'EC2IoTPolicy'
policy_document = '''
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:*",
            "Resource": "*"
        }
    ]
}
'''

iot_client.create_policy(policyName=policy_name, policyDocument=policy_document)
```

```
iot_client.create_policy(policyName=policy_name, policyDocument=policy_document)

# Attach the policy to the EC2 instance
iot_client.attach_policy(policyName=policy_name,
                         target=ec2_instance_id)

# Create an IoT Thing for the EC2 instance
thing_name = 'EC2InstanceThing'
iot_client.create_thing(thingName=thing_name)

# Create a certificate for the EC2 instance
certificate_response = iot_client.create_keys_and_certificate(setAsActive=True)

certificate_arn = certificate_response['certificateArn']
certificate_id = certificate_response['certificateId']
certificate_pem = certificate_response['certificatePem']
private_key_pem = certificate_response['keyPair']['PrivateKey']
public_key_pem = certificate_response['keyPair']['PublicKey']

# Attach the certificate to the IoT Thing
iot_client.attach_thing_principal(thingName=thing_name, principal=certificate_arn)

# Connect to the EC2 instance via SSH
# Install the AWS IoT SDK and dependencies on the EC2 instance

# On the EC2 instance, save the certificate, private key, and root CA certificate to files
cert_file_path = '/path/to/certificate.pem.crt'
key_file_path = '/path/to/private.pem.key'
root_ca_file_path = '/path/to/root-ca.crt'

with open(cert_file_path, 'w') as cert_file:
    cert_file.write(certificate_pem)

with open(key_file_path, 'w') as key_file:
    key_file.write(private_key_pem)

# Establish an SSH connection to the EC2 instance and transfer the root CA certificate
ssh_client = paramiko.SSHClient()
ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh_client.connect(public_ip_address, username='ec2-user', key_filename='/path/to/ec2-instance-key.pem')

sftp_client = ssh_client.open_sftp()
sftp_client.put(root_ca_file_path, 'root-ca.crt')
sftp_client.close()

ssh_client.close()
```

Fig. **12**. Subscribing and publishing to AWS IoT Core

```
def main():
    # create chess engine and response
    engine = chess.engine.SimpleEngine.popen_uci(os.getcwd()+"/stockfish.avx2")
    board = chess.Board()
    print(board, "\n")

    # Wait for incoming messages
    message = iot_client.get_thing_shadow(thingName='my_thing_name')

    # Check if a message is received
    if 'payload' in message:
        player_move = process_message(message['payload'].read().decode('utf-8'))


    player = input("Choose white or black: ")
    if player == "white":
        player_input_2(board,player_move)
        print(board, "\n")

    while not board.is_game_over():
        # Bot's turn
        result = engine.play(board, chess.engine.Limit(time=0.1))
        print("Bot's move: "+str(result.move))
        board.push(result.move)
        print(board, "\n")

        # Player's turn
        message = iot_client.get_thing_shadow(thingName='my_thing_name')
        if 'payload' in message:
            player_move = process_message(message['payload'].read().decode('utf-8'))
        player_input_2(board, player_move)
        print(board, "\n")

    print("Winner: "+ board.outcome().winner)
    engine.quit()
    return 0
```

Fig. **13**. Main Code for Instance

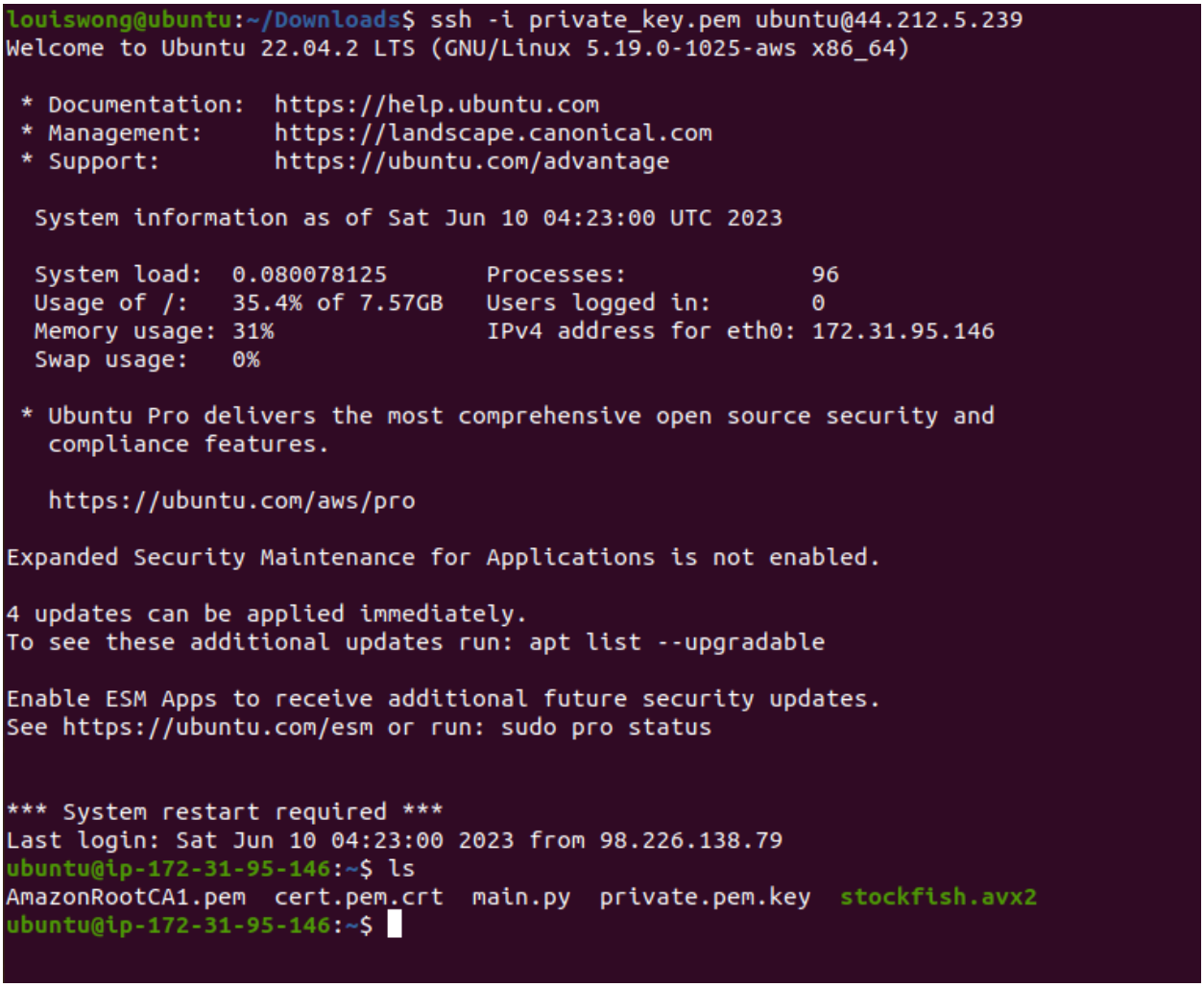

Fig. **14**. Accessing the EC2 instance

IV. Final Product

IV.a Initial Goal vs. Final Product

We set out to create this board knowing that it would be a challenge. There is a lot of hardware involved, and successful completion requires that many subsystems be integrated together successfully. The initial goal was to have a cohesive final product that could play a full game of chess. Due to the sheer size of the project and people's busy schedules during integration week, we fell short of our goal. Each subsystem is most of the way over the finish line, but compounding issues during integration and limited time together as a team due to external factors meant that we were unable to reach the initial goal. First, after getting the sensing system fully functional, we tried to solve a speed issue on the Hall sensor grid by changing out bypass capacitors which resulted in unstable outputs; repairing the capacitors would have resulted in too much time lost, and in some cases may not have been possible due to damage inflicted on the solder pads. Second, we fried the FET that turns the magnet on and off and had to resort to using the isolator

circuit to trigger the magnet by hand since new parts couldn't be located or ordered. The code is equally mature, but since we spent so much time chasing down hardware bugs, we didn't have a chance to have the subsystems commanded from a main processor. Likely another week or two would've been needed to debug the communications between subsystems and finish the goal we set out to complete. Considering that spring quarter has a missing week compared to other quarters, we were virtually able to accomplish around 85-90% of the target goal. Essentially, each subsystem in itself was virtually complete and required minimal debugging. The gantry subsystem itself was a great success and was an impeccably manufactured system and testing with the motors showed rather accurate movement. Integration with the main code would not have been too difficult with this subsystem. The bulk of the firmware code was completed from the logic of the game to subscribing and publishing to AWS to generating proper response from an engine, however as stated above we just ran out of time to complete integration of the project as a whole. Overall, we have excellent, robust designs that simply require some more fine tuning and integration headache to turn into a cohesive product.

IV.b Performance and Limitations

The gantry moves smoothly and can move as fast as the pieces can handle. It does have an issue with cross loading the bearings which comes from the belt design and lack of stiffness in the large cross-slide portion of the gantry. This makes simultaneous (diagonal) moves difficult due to the compound forces placed on the gantry. The magnet system works great and easily moves the pieces quickly along the surface of the board.

The sensing system by itself works great, outputting moves as expected over the UART bus, and the PCBs fit nicely into our board surface; unfortunately, due to the sensors' chosen bypass capacitors being of too large a value (0.1 μF), the sensors need to wait longer than expected to activate. Ultimately, it takes ~2 seconds to register each new piece lift or place from the player. An attempt to change the bypass capacitors in the final hour caused unstable outputs on almost all of our sensors, rendering the current hardware essentially useless, but the firmware and the hardware still performs as expected when assembled as planned, albeit slowly. Additionally, due to time constraints, as well as the fact that we cannot recognize what piece type is present on each space, the system is unable to recognize all possible moves a player could make in chess, such as castling, pawn promotion, or en passant.

In terms of code performance for the game logic, there are a few considerations. Firstly, it's important to ensure that the G-code processing and move execution steps are efficient to minimize delays during gameplay. As the code is incomplete in terms of handling G-code commands and extracting source and target coordinates, it's difficult to assess the performance of these specific parts.Additionally, it's worth considering the computational resources and memory usage of the chess engine (Stockfish) during gameplay. Running the engine in a separate process (as done with engine = chess.engine.SimpleEngine.popen_uci("/path/to/stockfish")) can help offload the

computation and prevent the game loop from becoming unresponsive due to lengthy calculations. Furthermore, it's important to handle potential exceptions or errors that may occur during the execution of moves, interaction with the engine, or other parts of the code. Proper error handling and error messages can enhance the user experience and help identify and resolve any issues that may arise.In terms of code performance, there are a few considerations. Firstly, it's important to ensure that the G-code processing and move execution steps are efficient to minimize delays during gameplay. As the code is incomplete in terms of handling G-code commands and extracting source and target coordinates, it's difficult to assess the performance of these specific parts.Additionally, it's worth considering the computational resources and memory usage of the chess engine (Stockfish) during gameplay. Running the engine in a separate process (as done with engine = chess.engine.SimpleEngine.popen_uci("/path/to/stockfish")) can help offload the computation and prevent the game loop from becoming unresponsive due to lengthy calculations.

V. Challenges Encountered

V.a Sensing Subsystem

When the design process for the sensing subsystem began, we noticed that a lot of other projects' sensing systems used multiple multiplexers, and had widely varying levels of complexity in terms of wiring. Given our ten-week timeframe, we hoped to have as simple and straightforward of a design as possible, and looking at the setbacks encountered here, we largely achieved that goal. Rather than issues with complex systems creating unnecessary confusion, our main problems encountered were simple oversights of electrical engineering fundamentals.

Our first setback was caused by a lack of diode protection on each Hall effect sensor. As we found out in the prototyping phase, any voltage applied to the output of a sensor is enough to power the device, as this voltage gets applied to the VDD pin. This essentially meant that for our initial tests of the 3x3 prototype, all sensors would get powered at once, regardless of which row was powered, and any activated sensors would pull their entire column to ground. Once this was discovered, it was quickly remedied by adding 1N4148 diodes to the outputs, though due to an eagerness to get ahead, an order of sensor PCBs was placed without the diodes implemented; this was quickly remedied and reordered, though it meant some time was lost.

Our second main oversight is the one that indirectly shut down our hardware version of the sensing matrix. When choosing bypass capacitors, we failed to consider that our faster switching speed for the sensor power would necessitate a faster charging capacitor, and ended up using the same value as for our ESP32 power supplies. While this still worked on our built version, the amount of time needed to wait between polling each row of sensors would be incredibly inconvenient for the player. We could have disassembled the wiring harnesses and resoldered 64 lower-value capacitors in place, but given that this problem was discovered in the final week, and that we were able to operate our 3x3 prototype chessboard without capacitors, we recklessly decided to remove the

capacitors. Unfortunately, this destabilized the sensor outputs and rendered their ability to record accurate piece movements useless.

V.b Gantry

The gantry is lovingly handmade and thus has some issues, but gets the job done with charm. For the design, we planned on making ways ourselves, but ended up using some ground shaft that we purchased to make the movement smoother. The first challenge was too much friction in the bearings which was resolved by ordering some low-friction bearing plastic which improved the bearings immensely. We also encountered a lot of challenges with the stiffness of the design. By adding some additional screws to fix everything in place and some strategic fitting in the enclosure we got rid of some of the issues. Special attention needed to be paid to how the gantry moved and how the servos were tuned to help prevent this stiffness issue causing crossloading in the bearings. Since the control scheme is open-loop and our steppers aren't particularly strong, any crossloading resulted in loss of location on the board and required the gantry to be rehomed. We also had lots of issues with the stepper drivers and electromagnet switch being damaged by overcurrent faults. An additional optoisolator was used to keep the 12V and 5V busses isolated to prevent damage to the power source (my laptop) by the big 12V supply.

V.c Algorithms and Code

Firstly, the initialization of the chessboard, display, and engine is necessary to set up the game environment. The input pins for the chessboard need to be properly configured to receive commands for moving chess pieces. The gcode_move() function handles the processing of G-code move commands. It validates the move, updates the board, and checks for game over conditions. Additionally, it allows the engine to calculate its move and updates the display accordingly.The process_gcode() function is responsible for parsing and executing G-code commands. It distinguishes between move and set position commands, but the code provided lacks the implementation details for extracting the necessary source and target coordinates. The main game loop ensures that the gameplay continues indefinitely until interrupted. However, the loop body is currently empty and needs to be populated with the necessary code for receiving and processing G-code commands.Lastly, the engine.quit() statement is essential for proper cleanup and closure of the chess engine when the game loop is exited.

A significant challenge in the code was the implementation of the communication scheme between AWS and the ESP32, as going into the project, none of our team members had great knowledge about AWS at all. Due to the breadth of the available resources on AWS, it becomes exceedingly difficult to determine which functionalities are relevant to us and moreover, the communication protocols for communicating within AWS's many functionalities. Eventually, an understanding of AWS functionalities were gained and the exact pieces identified were used for implementation of the code.

Additionally, the initial loading of MicroPython into the ESP32 presented interesting challenges, as our team did not have good experience with the ESP32 or MicroPython, however, the appeal of programming in Python pushed us to choose to take this initiative. Because a lot of the chess logic of the game is very structural and requires matrix representation of boards and moves require textual descriptions, it quickly becomes tedious to implement these schemes in C, as strings are almost necessary to preallocate and without a very defined scheme, the boards are not easily stored in matrices without a rigorous definition of memory to be allocated. Python ultimately offers a better interface for designing firmware with readable chess logic and allows for the use of robust libraries written in Python that are very powerful tools in dealing with chess.

VI. Planning and Organization

VI.a Gantt Chart

	Start Date	End Date	Timeline	Status
ESD2	Mar 28, 2023	Jun 9, 2023		
Milestone 1	Mar 28, 2023	Apr 7, 2023		Complete
Define project goals	Apr 3, 2023	Apr 7, 2023		Complete
Research components	Apr 4, 2023	Apr 7, 2023		Complete
Prepare order	Apr 6, 2023	Apr 7, 2023		Complete
Milestone 2	Apr 8, 2023	Apr 23, 2023		Complete
Create coding scheme for software and write initial code for each part of chessboard	Apr 8, 2023	Apr 20, 2023		Complete
Put together small hall sensor test breadboard	Apr 16, 2023	Apr 23, 2023		Complete
Interface components for each part of chessboard seperately	Apr 20, 2023	Apr 23, 2023		Complete
Manufacture and put together gantry	Apr 8, 2023	Apr 23, 2023		Complete
Design and print test chess pieces for Hall sensor testing	Apr 8, 2023	Apr 16, 2023		Complete
Milestone 3	Apr 24, 2023	May 7, 2023		Complete
Set up software architecture for driver, ESP, and WiFi communication	Apr 24, 2023	Apr 28, 2023		Complete
Research better components and order	Apr 28, 2023	May 7, 2023		Complete
Design and create order for PCBs	Apr 24, 2023	May 4, 2023		Complete
Manufacture chessboard encasing	Apr 24, 2023	May 7, 2023		Complete
Combine all hardware, firmware, and current software using for full funtionality	May 2, 2023	May 7, 2023		Active
Milestone 4	May 8, 2023	May 21, 2023		Complete
Revise and complete firmware	May 8, 2023	May 21, 2023		Complete
Integrate PCB components into chessboard	May 14, 2023	May 21, 2023		Complete
Finish 3D printing and manufacturing for all chess pieces	May 8, 2023	May 18, 2023		Active
Interface new components for new version	May 17, 2023	May 21, 2023		Active
Milestone 5	May 22, 2023	Jun 4, 2023		Active
Continue to research and improve code	May 22, 2023	May 24, 2023		Active
Implement revised code and firmware	May 24, 2023	Jun 2, 2023		Active
If possible, add chess engine capabilities	May 30, 2023	Jun 4, 2023		Complete
Revise and improve manufactured parts	May 28, 2023	Jun 4, 2023		Complete
Milestone 6	May 29, 2023	Jun 9, 2023		Complete
Prepare online project	May 29, 2023	Jun 7, 2023		Complete
Write report	Jun 1, 2023	Jun 9, 2023		Complete
		Burndown		

Fig. **15**. Current Snapshot of the Gantt Chart

VI.b Bill of Materials

Item Desc.	Mfg. Part #	Unit Price	1000 Unit Price	Quantity	URL	In Final Design	Total Unit Price	Total Bulk Price
Gantry Belt Kit	GT250	$14.190	$14.190	1	GT2 Timing Belt Pulley, 8pcs GT2	Yes	$168.03	$135.25
NEMA 17 Bipolar Steppe	17HS10-0704S	$11.990	$11.990	2	https://www.amazon.com/STEPPI	Yes		
6.35mm linear axis beari	SW4 LMB 4 UU	$1.400	$1.200	4	6.35mm Linear Bearing 1/4*1/2*3/	Yes		
Idler Pullies	QWE000015	$11.800	$11.800	1	BZ 3D GT2 Idler Pulley Aluminum	Yes		
Integrated Load Driver	BTS3125TFATMA	$1.260	$0.573	1	BTS3125TFATMA1 Infineon Techn	Yes		
12V Electromagnet	a16101700ux025	$13.490	$13.490	1	Amazon.com: uxcell 12VDC 200N	Yes		
ND magnets for pieces	TRYMAG Small	$7.640	$7.640	1	Neodymium Magnet, Magnetized	Yes		
SMD Hall Sensor	DRV5032FBDBZ	$0.405	$0.263	75	DRV5032FBDBZR Texas Instrume	Yes		
Through Hole Halls for Pr	DRV5032FALPG	$0.493	$0.254	15	DRV5032FALPGM Texas Instrume	No		
Stepper Drivers	DRV8825PWPR	$5.930	$3.164	2	DRV8825PWPR Texas Instrument	Yes		
Buttons	PS1023ABLK	$2.010	$1.320	3	https://www.digikey.com/en/pro	Yes		
Voltage Regulators	926-LM2940CT5(	$1.960	$0.965	7	https://www.mouser.com/Produ	Yes		
Limit Switches	ZD-0026	$10.990	$10.990	1	https://www.amazon.com/REIFEN	Yes		
3 to 8 Multiplexer	74HC4051PW,11	$0.450	$0.176	3	https://www.digikey.com/short/n3c	Yes		
MUX breakout board	74HC4051	$2.950	$2.950	1	https://www.sparkfun.com/product	No		
2 pin connector - board	S2B-EH(LF)(SN)	$0.141	$0.063	40	https://www.digikey.com/short/9q9	No		
3 pin connector - board	S3B-EH(LF)(SN)	$0.145	$0.062	40	https://www.digikey.com/short/z39	No		
2 pin connector - wire	EHR-2	$0.054	$0.032	40	https://www.digikey.com/short/nwp	No		
3 pin connector - wire	EHR-3	$0.060	$0.035	40	https://www.digikey.com/short/22c	No		
Crimps for JST conns	SEH-001T-P0.6	$0.029	$0.019	200	https://www.digikey.com/short/p3c	No		
28 AWG cable	28 AWG Silicone	$8.990	$8.990	1	https://a.co/d/j3qfQXJ	Yes		
1N4148 SMD Diodes	1N4148WX-TP	$0.090	$0.020	75	https://www.digikey.com/en/produ	Yes		

Fig. **16**. Current Proposed BOM for complete implementation of Rook, Ma, No Hands!

	A	B	C	D	E	F	G
	Item Desc.	Mfg. Part #	Unit Price	Quantity	URL	Total Order $	
	TRYMAG Small	------	$10.990	1	https://a.co/d/6N	$61.81	
	SMD Hall Senso	DRV5032FBDBZ	$0.405	75	DRV5032FBDBZ		
	2 pin connector -	S2B-EH(LF)(SN)	$0.141	10	https://www.digikey.com/short/9q9dzwmm		
	3 pin connector -	S3B-EH(LF)(SN)	$0.145	10	https://www.digikey.com/short/z39bhp5n		
	2 pin connector -	EHR-2	$0.054	10	https://www.digikey.com/short/nwptz2pq		
	3 pin connector -	EHR-3	$0.060	10	https://www.digikey.com/short/22dt91d5		
	Crimps for JST c	SEH-001T-P0.6	$0.029	50	https://www.digikey.com/short/p3d9dj4j		
	28 AWG, thread	CBAZY Hook up	$14.990	1	https://a.co/d/5re		
	1N4148 SMD Di	1N4148WX-TP	$0.090	75	https://www.digik	$6.75	

Fig. **17**. Second and Third Orders

VII. Results of Market Research

VII.a Manufacturing Advice and DFM Review

Due to the large size and scale of our project, it made sense to talk to professionals about how to construct the prototype and what types of manufacturing techniques would be suitable for the application and end product. Due to the prototype nature of this build, we will have to go heavier, but keeping the gantry simple and utilizing off the shelf hardware for the precision ground ways and motion system will aid a lot in both the prototype execution and the potential future lifetime of the product. Most of the machined aluminum could be remade on CNC machines or cast for higher quantities. Once the overbuilt prototype is built, we can also look at slimming down a lot of the components so we have a better idea of what kind of loads it needs to be subjected to.

VII.b Interviews

What was learned from experts

An Expert in Toolmaking, provided some feedback regarding the user experience. According to the Expert, the experience was still a little "crunchy," indicating that there might be some roughness or areas for improvement in the gameplay.

As for the feedback, the Expert suggested "Ream in place." Reaming is a machining operation that enlarges a hole to achieve a precise size and finish. This suggestion could imply the need for better fitting or alignment of components in the chessboard to enhance the overall user experience.

Additionally, the Expert recommended adding additional races to reduce play in the lateral direction. This likely refers to minimizing any unintended movement or play in a sideways or horizontal direction. By incorporating measures to enhance stability and reduce lateral play, the chessboard can offer a more reliable and controlled gameplay experience.

In summary, the Expert's feedback suggests the need for improved fitting and alignment of components through reaming, as well as reducing unintended lateral movement in order to enhance the user experience of the chessboard.

What was learned from non-experts

Based on the user experiences and feedback gathered from interviews, it is clear that users value a smooth and efficient gameplay experience. They appreciate indicators to determine whose turn it is and want to ensure that the server response time is fast. The LED display for keeping score and indicating the winner is also highly desired.

In terms of design, it is important to prevent collisions between chess pieces during movement and keep noise levels to a minimum. The chessboard should not be an obstacle to playing, and the

labeling of each board position should be clear. Additionally, users appreciate an aesthetically pleasing design that appears magical, with components hidden in the final product.

There were some specific concerns raised by, including the possibility of using alternative sensor approaches like Piezoelectric or IR sensors, addressing power issues, and considering the weight of the final chessboard. Additionally, a contingency plan for processor failure should be in place.

Overall, it is important to prioritize the user experience, ensuring a seamless and enjoyable gameplay while incorporating the desired features and addressing the concerns raised during the interviews. The firmware development process should be given sufficient time, and the final product should aim to create a sense of wonder and provide a premium experience for chess enthusiasts.

VII.c Proposed Retail Price

The proposed retail price of our product is $175. This is supported by the cost of the calculated bulk product price in the BOM. Of course, the price is only approximated by the BOM for the bulk, as there are additional quantities that have been listed for testing and possible failures, however assuming that we take these quantities out and perform failure testing for larger-scale manufacturing, the cost of manufacturing one board would be around $100 on the high side. Then if we consider the cost of manufacturing the boards themselves and marketing efforts to sell the board itself, we ultimately realize that the price will need to be increased for these efforts as well. Finally, to make a profit we further assume that we should have a $15 to $20 margin, which places the estimated price at around $175.

VIII. Conclusion

VIII.a What I learned

This was one of the best teams I have been a part of and also one of the most fun yet thrilling projects I have done. We had a mountainous task to complete as a making this chess board was definitely difficult. I come from a pure Electronics background so working with the mechanical aspect of the project was very interesting for me. Personally, I was fascinated by the CNC gantry and the way my teammates planned it and designed it out for it to execute chess moves properly. I have a particular interest in embedded programming so this class was rather insightful as me and one other teammate who were in charge of the firmware had a lot of fun playing with servers and integrating everything on the ESP32. One thing I enjoyed learning a lot about the gantry, CNC machines and writing up the code for the CNC gantry was very insightful.

VIII.b What I would do differently if I could start over

I would definitely probably test hardware as soon as we get something up and fixed. I would do a lot of initial testing and have a lot of contingency plans. One of the main problems with such projects is since we are doing a lot of testing some parts might rather not work or are highly capable of giving last minute surprises. So having more contingency plans and maybe integrating the hardware and firmware from the beginning and make sure they can work together would definitely be something I would follow.

VIII.c Possible next steps

We did not have much time to integrate both the hardware and the software a lot. We basically did not have a lot of time to play with the board, test different scenarios etc. So, for a start I would integrate everything see how well everything talks with each other and learn as much as I can to make it fully functional.

IX. References

[1] “Chess Board, Chess Games, Electronic Board Games Chess Set,” Square Off. [Online]. Available: https://squareoffnow.com/. [Accessed: 16-Apr-2023].

[2] “Automatic chessboard,” Hackaday.io. [Online]. Available: https://hackaday.io/project/179268-automatic-chessboard. [Accessed: 16-Apr-2023].

Appendix A: Gantry Design

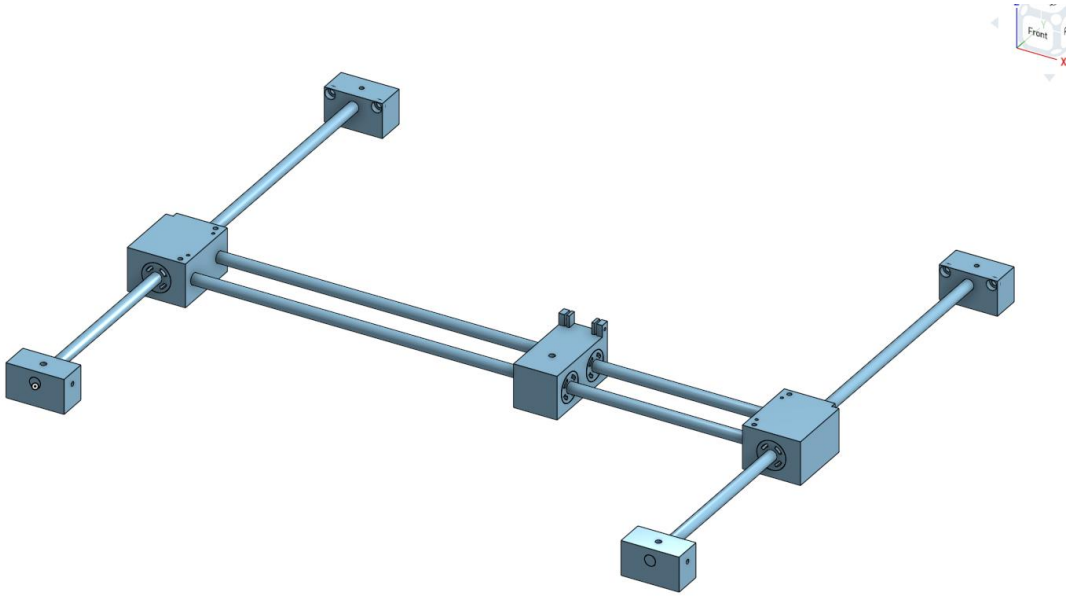

Fig. **A.1.** Complete Assembly for the Gantry System.

The proposed gantry is shown above in the completed CAD assembly. The style of gantry is an H-style or X-Y core system gantry. This allows the entire gantry to only use one belt and all x-y motion is only controlled by two stationary motors, versus one stationary and one moving motor. The entirety of the CAD files would constitute too much information and would not be too informative about the overall concept of the gantry system which is why they are not included. The gantry system vertical shafts are held at the ends by two end holders which stabilize the shafts by securing them with set screws. The motors sit behind one of these end holders. The block sliders contain linear ball bearings which move up and down these vertical shafts and allow for idlers to have a timing belt which the motors will use to control the movement of the sliders. The sliders themselves contain two shafts connecting the vertical rails and allow for the center slider, containing the electromagnet payload, to move to all possible positions in the board. The center block contains two linear ball bearings for the horizontal shafts and two clips for the timing belt.

Onshape Design Link:
(Chess pieces)
https://cad.onshape.com/documents/5c5ff7e0559bcd34a9efd177/w/76f350f08e0b6b70b9258d94/e/2d5451c39c4450f3d66b8feb
(Gantry)
https://cad.onshape.com/documents/f86f949d9b33f28a09a9f6c1/w/6fd4c9a5cbaacff90242ddaf/e/7757c1b23a39eded18ba117d

Appendix B: GitHub Repository

The following link is to our ESP32 firmware repository; it contains our code for the sensing subsystem, gantry commands, and game logic:

https://github.com/adheikd/Firmware--ESD-2.git

Class Feedback

The class was excellent and I had a lot of fun learning what I learnt in this class. No, the structure and format of this class was perfect. The progress reports helped us track our progress and motivated us to aim higher whereas the meetings were helpful as Professor Ilya had given us valuable tips and suggestions. I definitely enjoyed working with the ESP32. ESP32 is a really board to work with and has a vast functionality. However, I would like to work with some of the PIC kits like maybe the PIC32 MZ DA Curiosity Dev board.
This project has been an intriguing and engaging experience. The class provided a valuable platform for exploring new ideas and concepts while applying them to real-world scenarios. The chessboard project allowed for meaningful discussions and diverse perspectives, enhancing the overall learning experience. Interacting with my teammates and their unique perspectives broadened my understanding and enriched my knowledge. Overall, this project and class have been intellectually stimulating and rewarding, fostering growth and deepening my understanding of the subject matter.