ChessMate: A DIY Smart Chessboard

Parts list

Hardware:

128 **resistors**

Resistors are fundamental components in electronic circuits that limit the amount of current that flows through them. In this project, they're primarily used to protect the LEDs and switch sensors from receiving too much current, which could damage or destroy them. Each LED will typically require its own resistor connected in series.

64 **LEDs**

Light Emitting Diodes (LEDs) are used to visually indicate the status of each square on the chessboard. They may be used to display when a square is occupied, suggest possible moves, or highlight special conditions in the game (like check or checkmate). Each LED corresponds to a square on the chessboard, allowing for precise visual communication with the user.

64 **pairs socket wires**

These wires connect the LEDs to the IO extenders and the main circuit, allowing for communication between the different parts of the circuit. Socket wires help transmit electrical signals while providing flexibility and ease of assembly and maintenance. Otherwise, connecting the LEDs to wires would be tedious and require precise wiring.

16 **IO extenders**

The Arduino microcontroller has around 16 total pins that it can read or write information to. In our design, there are far too few to meet our needs. IO (Input/Output) extenders expand the number of pins available on the microcontroller, using the microcontroller’s SCL (Serial Clock Line) and SDA (Serial Data Line) pins (that make up the I2C channel) to communicate more complex information, allowing you to control ***many*** LEDs and switch sensors simultaneously. These become crucial in large projects, like this chessboard, where each LED needs to be independently controlled without overwhelming the limited IO capabilities of the Arduino.

1 **multiplexer**

A multiplexer allows multiple signals to share a single line or address. In this project, the IO extenders are writable from 8 possible addresses (0x20 - 0x27). This means that without the multiplexer, the Arduino can communicate with a maximum of 64 additional I/O’s. The multiplexer chip allows us to put addresses behind a maximum of 8 channels. This means that it can be used to efficiently manage the input from the LEDs or any other sensors you might include in the future, allowing for an essentially uncapped number of reachable I/O extenders.

1 **Arduino uno** w/ wifi shield

The main engine behind it all. The Arduino Uno serves as the main microcontroller, handling the control logic for the LEDs and any user input. The attached WiFi shield enables wireless connectivity, allowing the chessboard to communicate with other devices or a network, possibly for online gameplay or updates.

<u>Wires</u>

Wires are where we expect learners to face their biggest challenge. It is an organization problem at its core. Your board occupies a limited volume of space, the number of connections are so great, and the likelihood that any one wire might slip and fall out or cross and short circuit with another is near inevitable. Having some wires pre-soldered and a predetermined length helps mitigate these issues to some degree, but the learner will have to think not just into the future when layers will be placed on top of layers, making it difficult to go back and access older connections, but also hyper critically about the direction and placement of each set of wires as they relate to others and what parts they fit into.

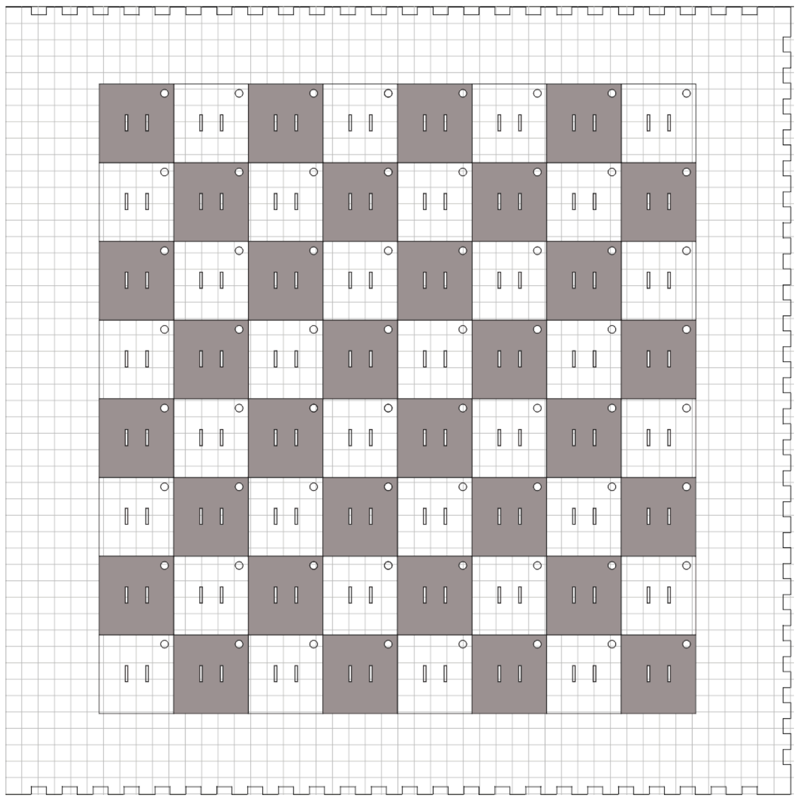
128 **wires** (resistors - ground)
64 **wires** (resistors - LEDs)
64 **wires** (LEDs - IO extender)
64 **wires** (IO extenders - resistor)
64 **wires** (resistor - switch sensor)
64 **wires** (switch sensor - power)
80 **wires** (IO extenders - setup (power, ground))
32 **wires** (IO extenders - multiplexor SCL, SDA)
16 **wires** (Arduino - IO extenders INT)
8 **wires** (IO extenders - INT pin)
6 **wires** (Multiplexor - setup (power, ground, RST)
2 **wires** (Arduino - multiplexor SCL, SDA)

<u>Additional wires</u> needed (These will vary based on your breadboard layout and space usage):
Wires (Arduino - power, negative)
Wires (Arduino - multiplexor SCL, SDA)

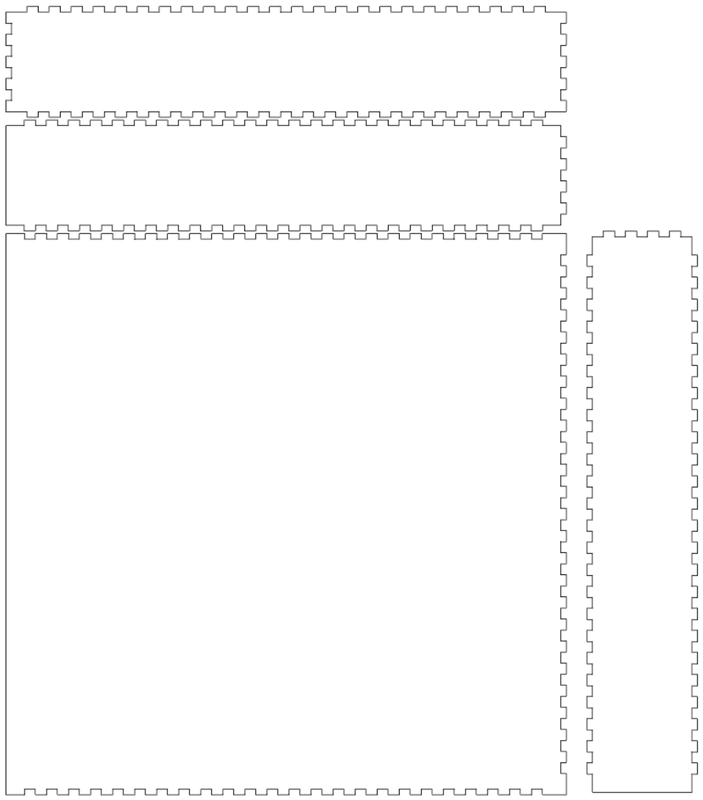
<u>Board pieces:</u>

The board is laser cut and assembled by hand with no glue or additional adhesives. These pieces will be provided but not assembled for learners. It is perfectly acceptable as a tinkerer to build and assemble their own board! Just remember that as you build, iterate, and debug, you're going to need to access the components easily and frequently. Because of this, the pieces provided are to build a drawer-type box, where an individual can slide out the interior box and access the circuits. Future iterations will have LED and switch sensor wires embedded into the board to cut back on the complexity of the final product and reduce both intrinsic (complexity of the problem) and extraneous (unnecessary and confusing information) cognitive load.

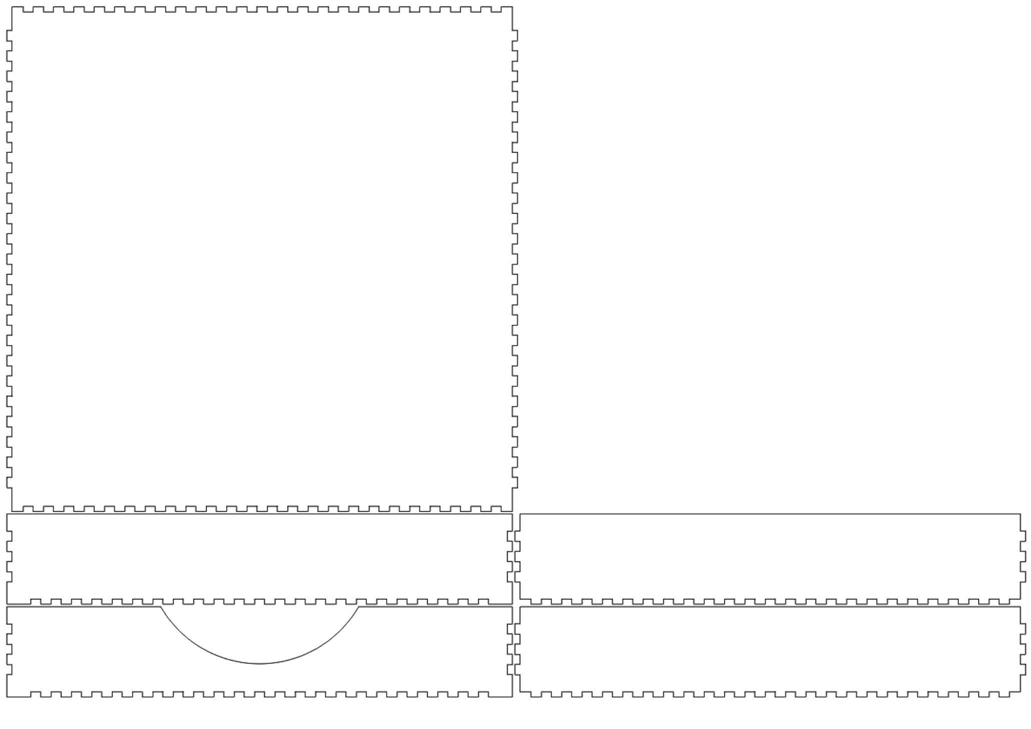
Exterior Board Top:



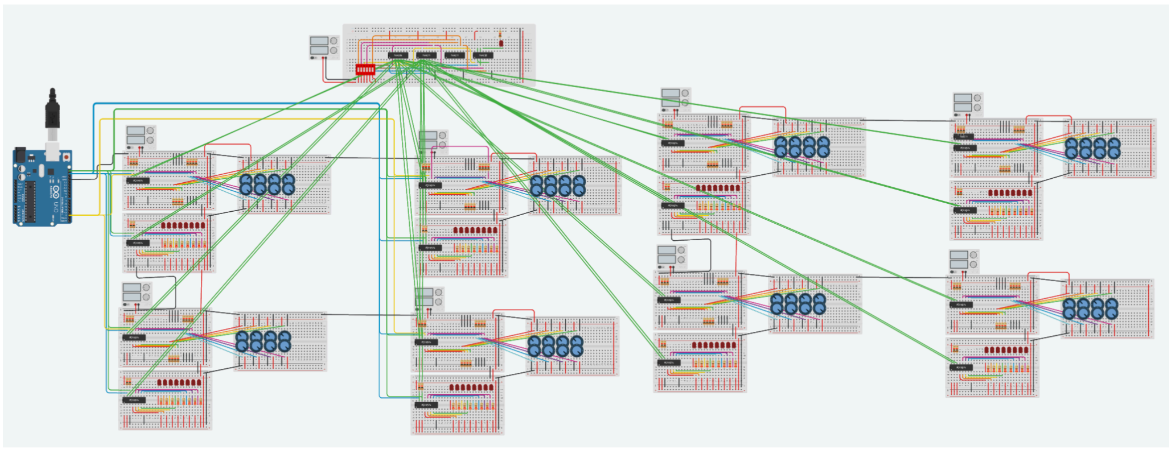
Exterior Board sides and bottom:



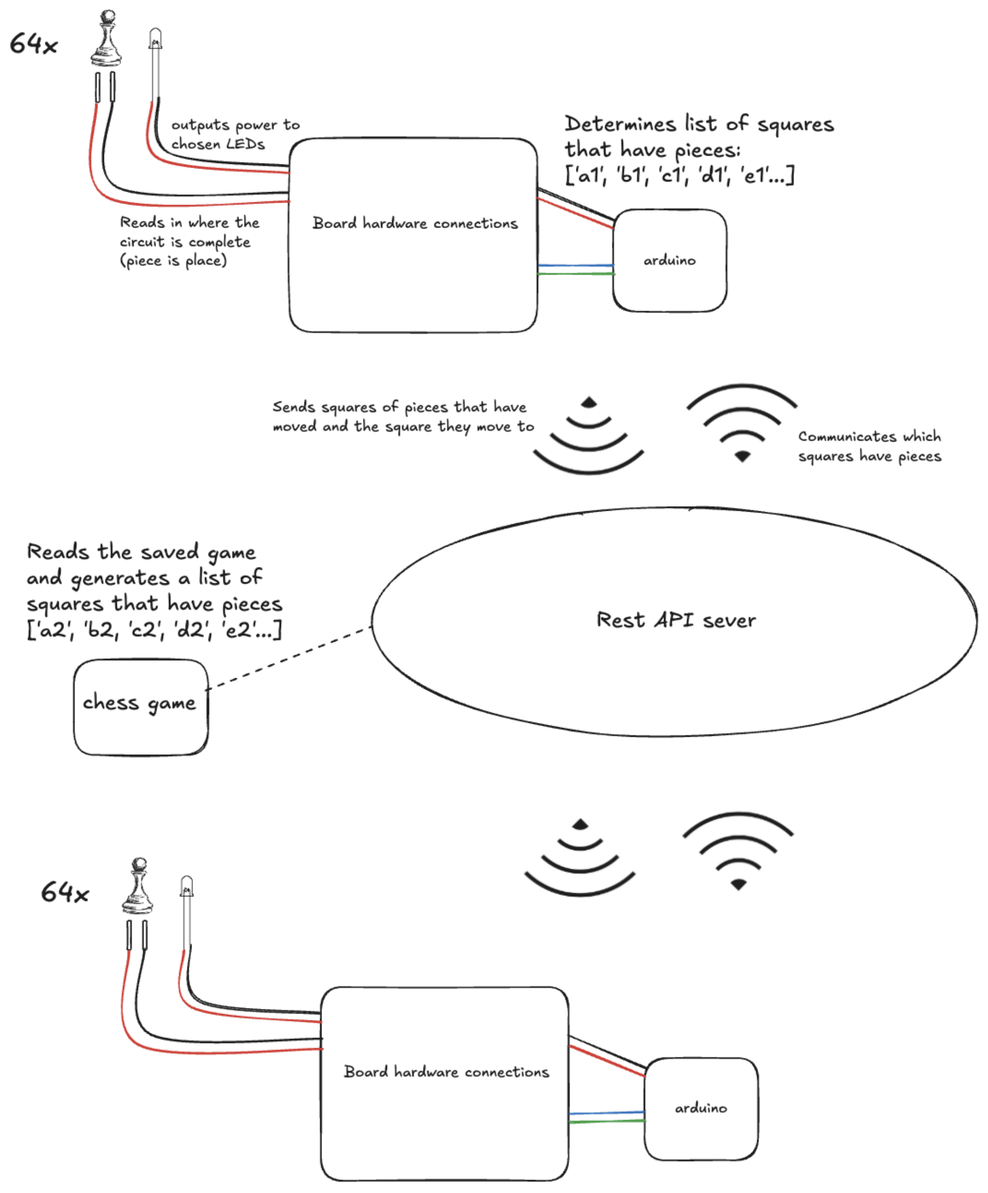
Interior board:



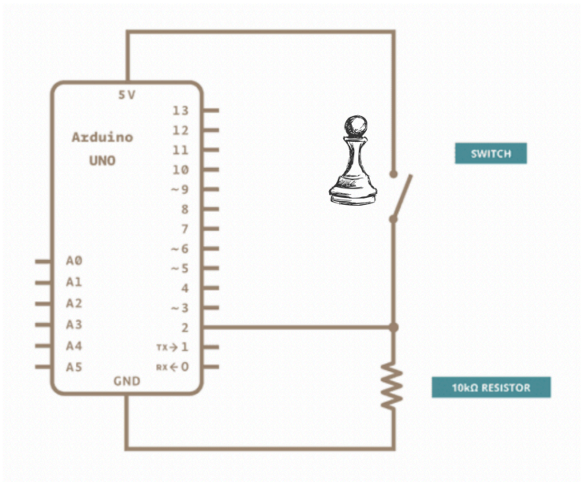
What does that look like?



Okay… but what does that look like at a higher level?


64x
outputs power to chosen LEDs
Reads in where the circuit is complete (piece is place)
Board hardware connections
Determines list of squares that have pieces: ['a1', 'b1', 'c1', 'd1', 'e1'...]
arduino
Sends squares of pieces that have moved and the square they move to
Communicates which squares have pieces
Reads the saved game and generates a list of squares that have pieces ['a2', 'b2, 'c2', 'd2', 'e2'...]
Rest API sever
chess game
64x
Board hardware connections
arduino

What is a ‘Switch sensor’?:



The switch sensor in our project is integrated into the board's circuitry, connecting to the IO extender (specifically the Input extenders) pin. The circuit is usually open (incomplete), meaning that no current flows through and no signal is sent. However, upon placing a piece on the square, the circuit is completed. This sends a signal to the IO extender pin, which communicates to the microcontroller that a piece is now occupying that square.

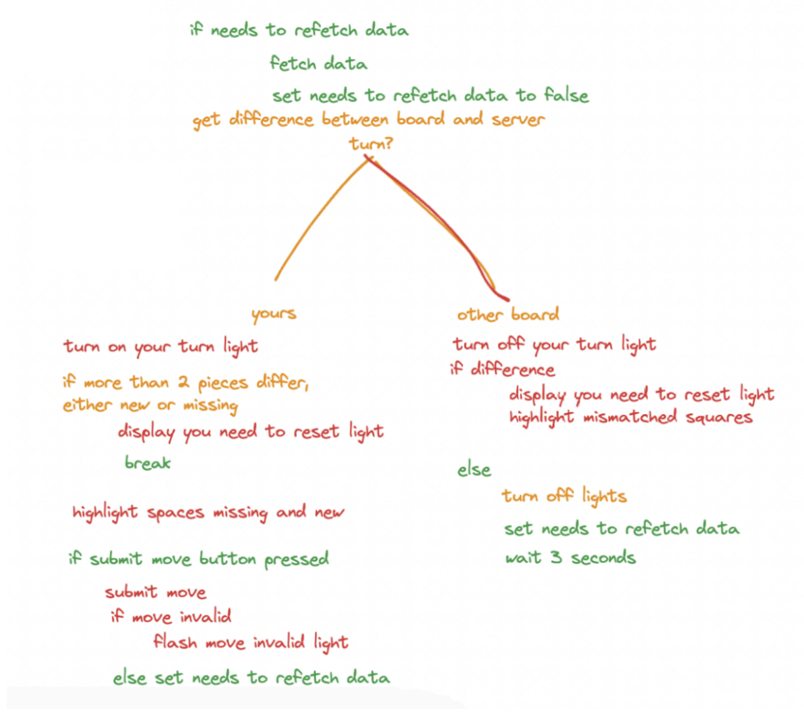
Advantages:
Switch sensors are highly valued for their simplicity and cost-effectiveness. They are robust components with few failure points, making them a reliable choice for a prototype. This straightforward mechanism ensures that the detection process is both quick and efficient, and requires minimal processing power from the microcontroller.

Challenges:
However, the simplicity of switch sensors comes with its own set of challenges. The primary downside is the difficulty in ensuring a steady connection, particularly in a grid as complex as a chessboard. Additionally, connecting wires to the board using a medium like connective tape is prone to failure points. Each sensor must be meticulously connected to its respective IO pin, power, and ground, which can make assembly labor-intensive. Additionally, any slight misalignment or loss of contact can result in unreliable performance, possibly missing a piece's placement or removal.

Overall, while using switch sensors presents certain challenges in terms of physical connectivity and reliability, their advantages of low cost and simplicity make them an excellent choice for early-stage prototypes (like the one we developed this semester).

Game Logic Pseudo-Code:

Your microcontroller will be responsible for organizing the data coming in from the inputs, communicating data with the server, and displaying outputs to the LEDs accordingly- depending on the state of the game and the placement of the pieces. It is our recommendation that your pseudo-code for this game logic should follow as such:



```
if needs to refetch data
        fetch data
        set needs to refetch data to false
get difference between board and server
                turn?

        yours                                   other board
turn on your turn light                     turn off your turn light
if more than 2 pieces differ,               if difference
either new or missing                           display you need to reset light
    display you need to reset light            highlight mismatched squares
    break
                                            else
highlight spaces missing and new                turn off lights
                                                set needs to refetch data
if submit move button pressed                   wait 3 seconds
    submit move
    if move invalid
        flash move invalid light
    else set needs to refetch data
```

Central Game Brain/Server:

In our smart chessboard system, a Python Flask RESTful API acts as the central server that orchestrates communication and data management between the chessboards. This server plays a crucial role in ensuring seamless gameplay and synchronization between multiple boards.

High-Level Description:

The central server is developed using Flask, a lightweight and versatile web framework for Python. It is designed to expose both GET and POST endpoints:

- GET Endpoints: These allow the boards to retrieve the current state of the game, including the placement of all pieces currently in the game and/or the difference between the read squares and actual squares occupied. This ensures that both players are fully informed about the game status, regardless of their physical location.
- POST Endpoints: These are used by the boards to communicate player moves back to the server. Once a move is received, the server updates the game state

and processes any necessary validations or rule checks, such as enforcing legal moves or recognizing check and checkmate conditions.

Centralized Orchestration:

The server functions as the centralized source of truth for the game, facilitating real-time data synchronization between the participating boards. By maintaining and consistently updating the game state, the server ensures that all players are interacting with the most recent game configuration. This orchestration allows for fluid, coordinated gameplay and minimizes discrepancies or errors in piece placement.

Learner Integration:

For learners building their custom boards, this server is made available as a hosted service. By directing their board's requests to our centralized server, learners can effortlessly participate in online gameplay without having to build or design their own. This setup provides an excellent opportunity for developers to focus on board design and interaction logic while leveraging a ready-made backend infrastructure.

This approach empowers users to integrate their designs into a broader ecosystem, enhancing the educational experience and encouraging innovation in smart board projects. That being said, there is nothing stopping a learner from taking our design, developing and customizing their own ‘game brain’.

Tips and tricks for troubleshooting:

Building your DIY smart chessboard is like solving a complex puzzle, and let's face it—sometimes, that puzzle just won't work the way you anticipated. But don't worry! We've all been there, and with a bit of perseverance and these handy tips, you'll have those LEDs lighting up in no time. Remember, troubleshooting is just another step in the exciting learning journey!

1. Keep Calm and Carry On

The first rule of troubleshooting is not to panic. It's perfectly normal for things not to work right the first time—it’s a key part of the learning and innovation process. Take a deep breath and get ready to dive in.

2. Break Down the Problem

Think like a detective. Start by isolating different sections of your circuit to determine which part might be causing issues. For instance:

- Check the power supply: Make sure your Arduino is receiving power and that all connections to the power and ground are secure.

- Verify individual components: Test each LED, sensor, and IO extender separately to ensure they're functioning correctly.

3. Follow the Trail

Carefully check the path of electricity through your system:

- Resistors and Connections: Confirm that all resistors are properly connected and soldered. A loose connection could be the culprit.
- Wire Paths: Ensure that your wires aren't tangled and are connected to the correct pins. Refer to your circuit diagram frequently.

4.Embrace Iteration

Each failed test is not a setback but a stepping stone towards your final product. Embrace that process:

- Document what you tried, what happened, and what you plan to tackle next. This helps avoid re-doing the same steps.
- Make incremental changes and test frequently. If you change too many things at once, it will be harder to identify what fixed or broke something.

5. Consult the Community

You're not alone! If you're stuck, consult online forums, reach out to classmates, or get in touch with others working on similar projects. Sometimes a fresh set of eyes can spot something you might have overlooked.

6. Give It a Break

When frustration kicks in, sometimes the best solution is a break. Step back, grab a cup of coffee, and return with fresh eyes. You'd be surprised how often the answer becomes clear after a little time away.

Remember, building something new is all about exploration, iteration, and, yes, a little bit of troubleshooting. Celebrate every small victory along the way and enjoy the learning process. Happy building!