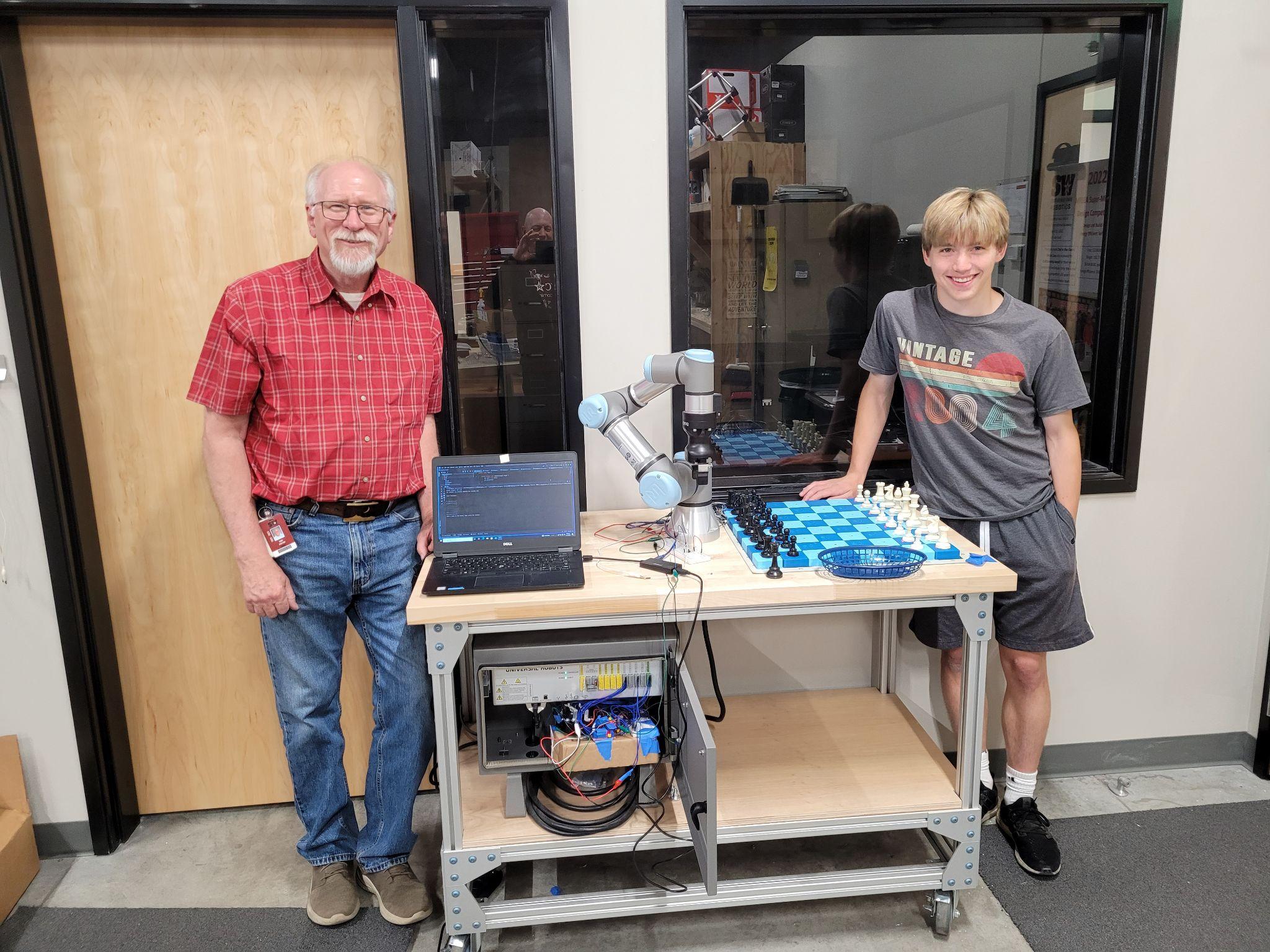
Chess Playing Robotic Arm

Southwest Christian High School

Nickolas Azanov

Gary Shelton, Faculty Advisor

Special Thanks To:

Wyatt Peterson

Caleb Hay

2021-2023

# Why This Project

As a whole, this is a project that connects my enjoyment of chess with my experience in circuits. For my entire childhood, I've played chess competitively by going to numerous tournaments. Even though I don't play as often as I did before, I still love the game. In addition to the game itself, I always found chess engines to be incredibly interesting. The idea of calculating the "best" move always interested me, so I always dreamed about being able to one day make my own chess engine.

During my sophomore year of high school, my school offered a class named "Automated System Controls" (taught by Gary Shelton, my advisor) where we learned about basic circuits and ladder programming (PLCs). That class gave me the intuition needed to create electronic control systems. During that same year, Mr. Shelton purchased a programmable robotic arm. One day, I randomly told him that I thought it was possible to make it play chess.

When I became his T.A. my junior year, he encouraged me to build on that idea, to make his robotic arm play chess. Since I've always wanted to combine programming with chess, I couldn't say no to this opportunity.

So, with my basic knowledge of Java, Arduino, and circuits, I began this project. After two years, this is my proof of a working prototype.



# 

# Descriptions

#### Summary

This is a project that uses a Universal Robots robotic arm to create a semi-automatic chess playing system. A player plays as white against the robotic arm that plays as black. The player makes his move and presses a button, and the robotic arm makes its move in response. This process continues until the game is finished.

#### Process of how it works

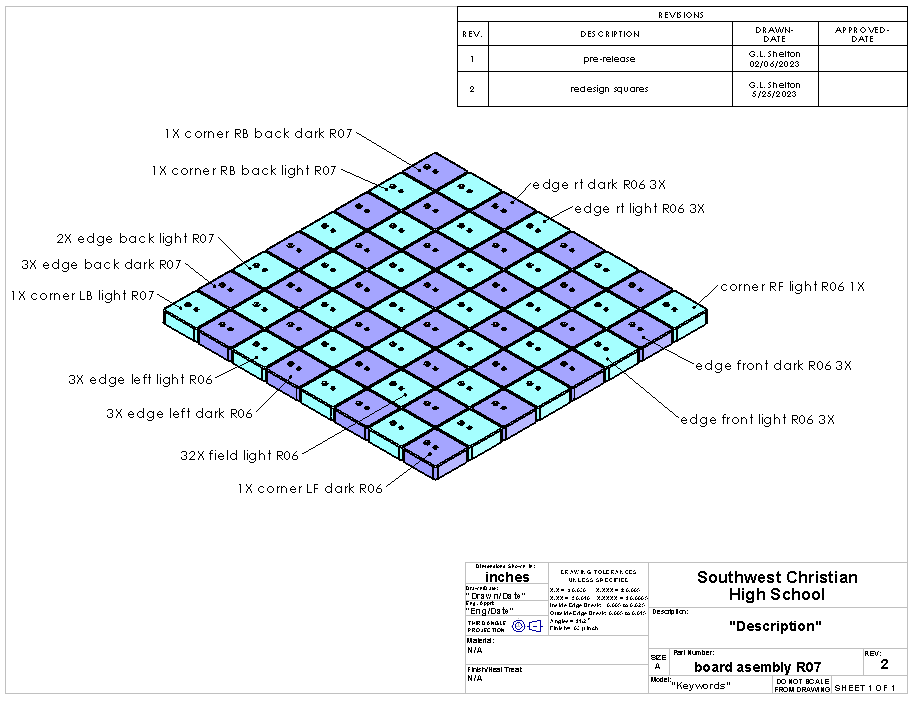
# While the player is making his move, the board position is being read. Underneath each square is a pcb circuit with a hall effect sensor. Underneath each piece is a magnet, and the hall effect switch is activated when a magnet is above it (hence, when a piece is on that square). An LED is activated to signal that a piece is on that square. Each circuit has a designated signal wire that is sent to an Arduino MEGA that records the board positions. During a player’s move, the starting position, the ending position, and all the squares that experienced change are recorded. After the player finishes the move and presses the button, that info is sent via serial communication to the java program on the laptop, and the program uses that info to determine what move the player made. Then, the java program performs a minimax search algorithm with alpha-beta pruning to calculate which move to make in response. Once a move has been calculated, all the details about the move (the start square, the end square, if it’s a capture, if the piece is tall, etc.) are sent to an Arduino UNO via serial communication. That Arduino UNO then translates that info into voltage signals sent to the robotic arm’s I/O box. Once the robotic arm reads those signals, it performs its move. Then, it’s the player’s move again. This cycle repeats until a checkmate or stalemate occurs.

#### Contributions

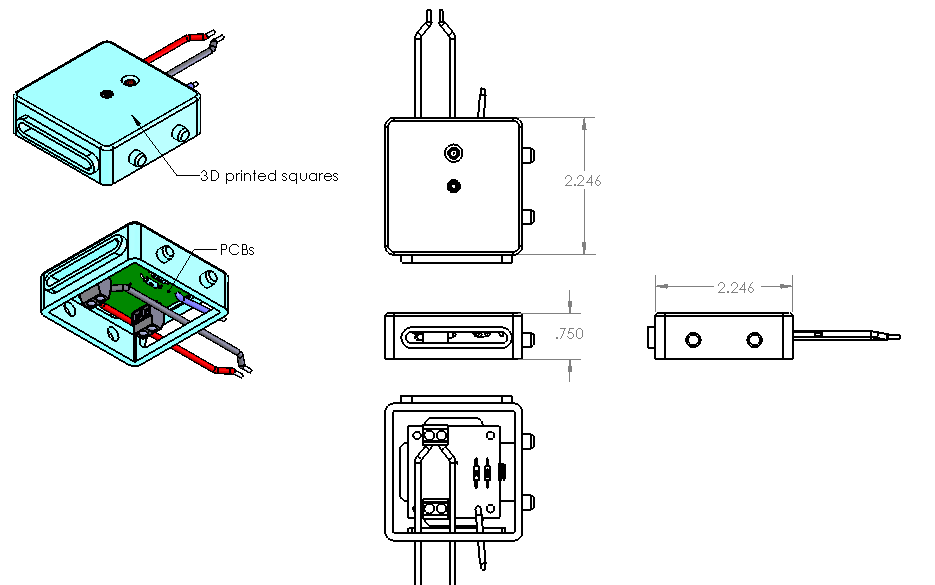
The physical board and physical printed circuit boards were designed by Gary Shelton.

The wiring schematics and the programs were designed and written by Nick Azanov.

# Board Drawing



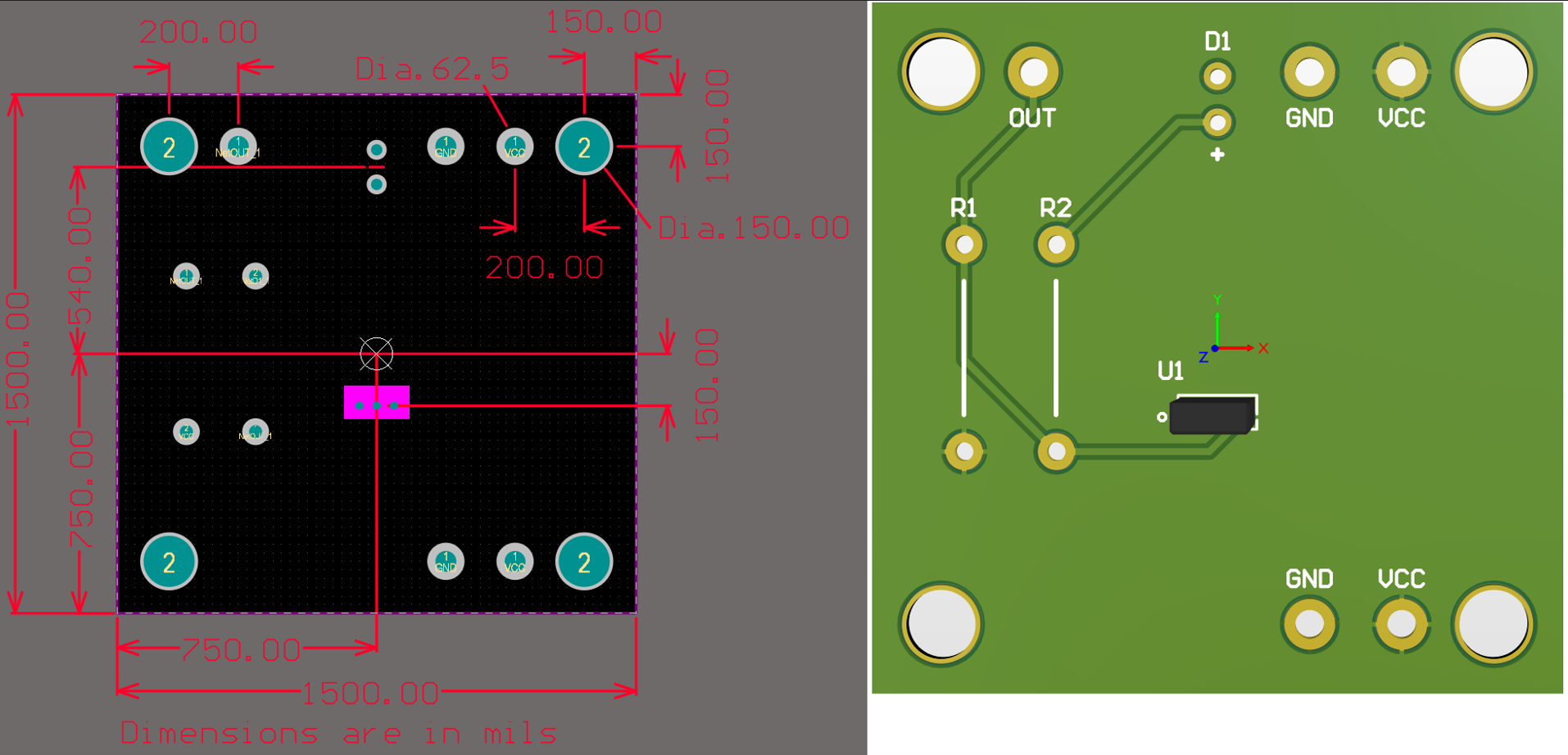
# Square Drawing



# 

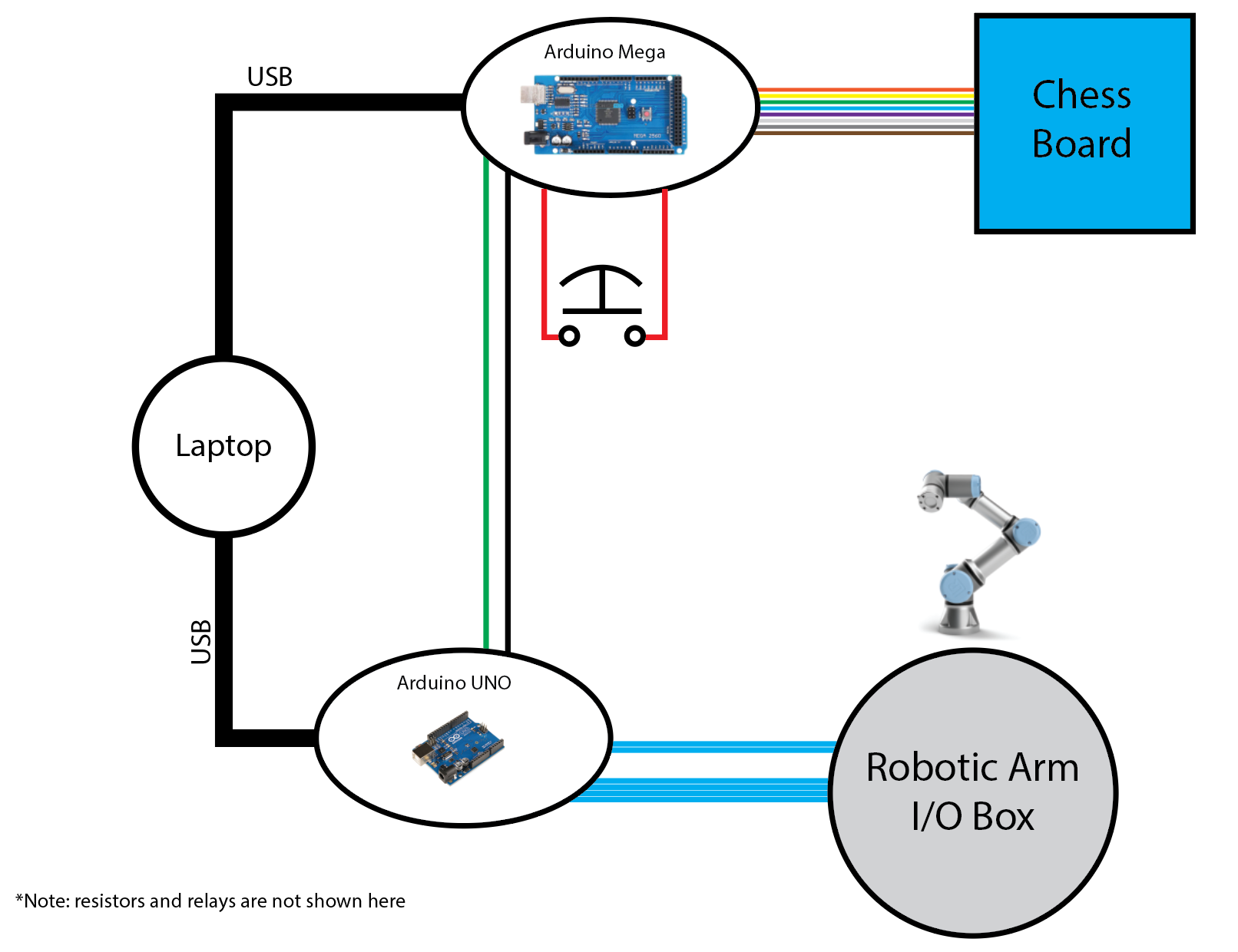
# PCB Circuit Drawing

# 



# 

# Circuitry Flow Chart

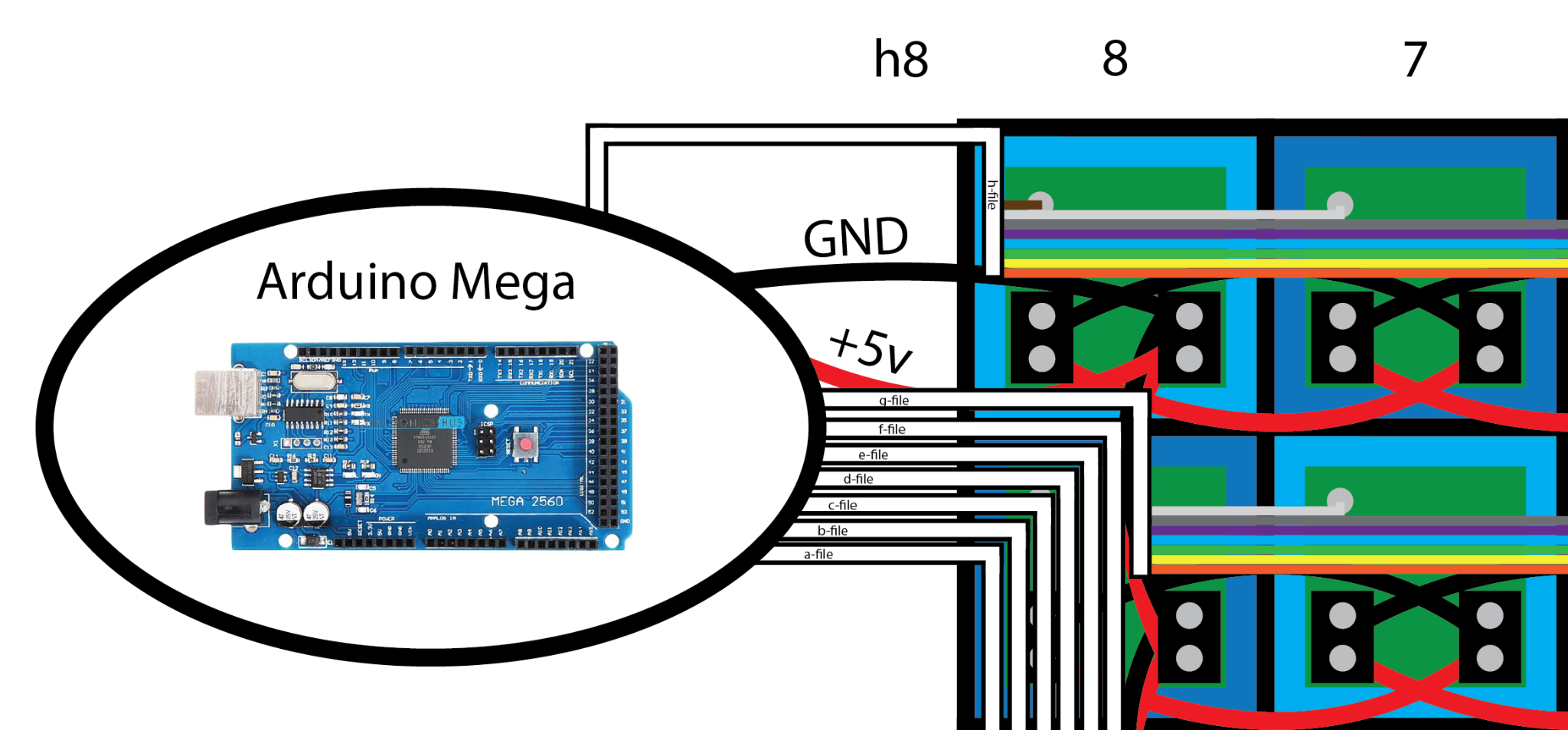


# Arduino UNO and URe3 I/O Box wiring diagram

# 

# Arduino Mega Circuitry

# Chess Board Wiring



# 

# Arduino UNO Code

### Chess\_project\_UNO\_code.ino

#define BUFFER\_SIZE 128

//these are the values that could be sent from the java program

#define NORMAL\_MOVE\_COMMAND 0

#define CAPTURE\_COMMAND 1

#define CASTLING\_COMMAND 2

#define ENPASSANT\_COMMAND 3

#define TRUE\_COMMAND 1

#define FALSE\_COMMAND 0

// pwm out pins to robot arm

const int YOutPin = 3;

const int XOutPin = 9;

// digital out pins to robot arm

const int isCapturePin = 12;

const int isEnPassantPin = 8;

const int isCastlingPin = 7;

const int isPromotionPin = 12;

const int isTallPin = 8;

const int isCaptureTallPin = 7;

//digital out pin to the arm indicating that this arduino’s output signals are ready

const int arduinoReadyPin = 4;

//digital out pin to arduinoMega indicating that the arm is ready

const int arduinoMegaPin = 5;

// digital in pin

const int armReadyPin = 2;

// initialization of temporary values (will constantly change)

int outputValue = 0;

// method that maps pwm output values from 0-8 to 0-255

int maps(int value) {

return map(value, 0, 8, 0, 255);

}

void setup() {

**Serial**.begin(9600);

//this pin reads when the robotic arm is ready

pinMode(armReadyPin, INPUT\_PULLUP);

//this tells the arm that this arduino UNO is ready

pinMode(arduinoReadyPin, OUTPUT);

//this tells the arduino MEGA that the arm is finished moving

pinMode(arduinoMegaPin, OUTPUT);

//these are the pins that go to the robot arm I/O box

pinMode(isCapturePin, OUTPUT);

pinMode(isEnPassantPin, OUTPUT);

pinMode(isCastlingPin, OUTPUT);

pinMode(isPromotionPin, OUTPUT);

pinMode(isTallPin, OUTPUT);

pinMode(isCaptureTallPin, OUTPUT);

//these pins output info on what squares the arm has to move to

pinMode(XOutPin, OUTPUT);

pinMode(YOutPin, OUTPUT);

}

//this is the buffer that reads commands from the java program

char readBuffer[BUFFER\_SIZE];

void loop() {

// waits for a command from the java program

while(**Serial**.available() < 8) {

delay(10);

}

**Serial**.readBytes(readBuffer, 8);

delay(1000);

//The information sent is translated into individual variables

int startX = (int) readBuffer[0]; //this is the x-coordinate of the starting square

int startY = (int) readBuffer[1]; //this is the y-coordinate of the starting square

int endX = (int) readBuffer[2]; //this is the x-coordinate of the ending square

int endY = (int) readBuffer[3]; //this is the y-coordinate of the ending square

//This contains the special move:

// 0 "NORMAL\_MOVE\_COMMAND" means the move doesn't require any crazy movements

// 1 "CAPTURE\_COMMAND" means the move is a capture

// 2 "CASTLING\_COMMAND" means the move is castling the king

// 3 "ENPASSANT\_COMMAND" means the move is an en passant capture

int specialMove = (int) readBuffer[4];

//for these three variables, the value will either be:

// 0 "FALSE\_COMMAND" or

// 1 "TRUE\_COMMAND"

int isPromotion = (int) readBuffer[5]; //does the move involve promoting a pawn

int isTall = (int) readBuffer[6]; // is the starting piece tall

int isCaptureTall = (int) readBuffer[7]; // is the piece that is being captured tall

// writes special moves

digitalWrite(isCapturePin, LOW);

digitalWrite(isCastlingPin, LOW);

digitalWrite(isEnPassantPin, LOW);

if (specialMove == CAPTURE\_COMMAND) {

digitalWrite(isCapturePin, HIGH);

}

if (specialMove == CASTLING\_COMMAND) {

digitalWrite(isCastlingPin, HIGH);

}

if (specialMove == ENPASSANT\_COMMAND) {

digitalWrite(isEnPassantPin, HIGH);

}

//writes the info about the starting square

analogWrite(XOutPin, maps(startX));

analogWrite(YOutPin, maps(startY));

digitalWrite(arduinoReadyPin, HIGH); // tells the arm that the first set of inputs are ready

delay(300);

digitalWrite(arduinoReadyPin, LOW);

// waits for the arm to be ready to accept extra inputs

// I do this so that it limits the number of inputs I have

// to physically build to the arm.

// Also, there are only two analog inputs to the arm anyway, so I have to do this.

while(true) {

if (digitalRead(armReadyPin) == 1)

break;

delay(10);

}

// is promotion

if(isPromotion == TRUE\_COMMAND) {

digitalWrite(isPromotionPin, HIGH);

} else {

digitalWrite(isPromotionPin, LOW);

}

// isTall

if(isTall == TRUE\_COMMAND) {

digitalWrite(isTallPin, HIGH);

} else {

digitalWrite(isTallPin, LOW);

}

// isCaptureTall

if(isCaptureTall == TRUE\_COMMAND) {

digitalWrite(isCaptureTallPin, HIGH);

} else {

digitalWrite(isCaptureTallPin, LOW);

}

//writes the info about the end square

analogWrite(XOutPin, maps(endX));

analogWrite(YOutPin, maps(endY));

//tells the arm that the second set of inputs are ready

//note: the delay times are fairly arbitrary.

//The time in total below should at least be 5 seconds to avoid any conflicts with the delay times in the arm's code

//If the time is set too short, the following lines of code in the while loop could be executed before the arm begins moving

//That would cause the whole system to softlock (halt) or crash

digitalWrite(arduinoReadyPin, HIGH);

delay(5000);

digitalWrite(arduinoReadyPin, LOW);

delay(3000);

// waits for the arm to finish its movement before accepting extra inputs

while(true) {

if (digitalRead(armReadyPin) == 1) {

delay (200);

if (digitalRead(armReadyPin) == 1)

break;

}

delay(10);

}

//Resets the pwm outputs

//note: these two lines aren't necessary for the program to work

//It is simply an organizes it so that pointless signals aren't being sent

analogWrite(XOutPin, 0);

analogWrite(YOutPin, 0);

**Serial**.write(3); // sends an arbitrary number. When the java program recieves this, it knows the arm is ready

//Indicates to the arduino MEGA that the arm has finished its move, and that it is free to start recording the board position

digitalWrite(arduinoMegaPin, HIGH);

delay (1000);

digitalWrite(arduinoMegaPin, LOW);

}

# 

# 

# Arduino MEGA code

### Chess\_project\_MEGA\_code.ino

#define PUSH\_BUTTON 50

#define ARM\_FINISHED 51

void setup() {

// put your setup code here, to run once:

**Serial**.begin(9600);

//a and b columns

for(int i = A0; i<=A15; i++)

pinMode(i, INPUT);

for(int i=2; i<=13; i++)

pinMode(i, INPUT);

for(int i=14; i<=49; i++)

pinMode(i, INPUT);

pinMode(PUSH\_BUTTON, INPUT); //button that player presses after finishing move

pinMode(ARM\_FINISHED, INPUT); // signal that will tell the mega (this arduino) that the arm has finished its move

}

void loop() {

byte startPos[64]; //stores the starting position

byte endPos[64]; //stores the ending position

byte changedSquares[64]; //stores the squares that changed states during the move

recordMove(startPos, endPos, changedSquares);

// waits for a Serial connection to be established between this arduino and the java program

while(!**Serial**) {

delay(10);

}

writeBytes(startPos, endPos, changedSquares);

//This delay time is arbitrary

// Five seconds works because it takes the arm over five seconds to move the piece

delay(5000);

//waits for Arm to finish its move before moving on to record

while (true) {

if (digitalRead(ARM\_FINISHED) == HIGH)

break;

delay(100);

}

}

//This is the function that writes the three byte arrays startPos, endPos, and changedSquares

//to the java program that decodes the move and moves the move in its stored position

void writeBytes(byte arr1[], byte arr2[], byte arr3[]) {

**Serial**.write(arr1, 64);

**Serial**.write(arr2, 64);

**Serial**.write(arr3, 64);

}

// Records the starting position, the ending position, and all the coordinates that experienced change

void recordMove(byte startPos[], byte endPos[], byte changedSquares[]) {

int i;

byte bufferBoard[64];

// clear the changedSquared buffer

for (i=0; i<64; i++)

changedSquares[i] = 0;

// reads the position before the move is made

readBoard(startPos);

//this will fill changedSquares[] with the squares that change until player presses the button

while (true) {

readBoard(bufferBoard);

compareBoards(startPos, bufferBoard, changedSquares);

if (digitalRead(PUSH\_BUTTON) == HIGH) {

readBoard(endPos);

break;

}

delay(30);

}

}

// compares the current position with the starting position, and records the changes in changedSquares[]

void compareBoards(byte startPos[], byte bufferBoard[], byte changedSquares[]) {

for (int i=0; i<64; i++) {

if (bufferBoard[i] != startPos[i])

changedSquares[i] = 1;

}

}

void readBoard(byte board[]) {

int i; // index variable for board[]

int j; // index variable representing the pins

// int board[64];

for (i=0; i<64; i++)

board[i] = 0;

// a column - [A1 to A7]

for (i=0, j=A0;j<=A7; i+=8, j++) {

board[i] = digitalRead(j);

}

//b column - [A8 to A15]

for (i=1, j=A8; i<64; i+=8, j++) {

board[i] = digitalRead(j);

}

//c column [13 to 6]

for (i=2, j=13; i<64; i+=8, j--) {

board[i] = digitalRead(j);

}

//d column [49 to 42]

for (i=3, j=49; i<64; i+=8, j--) {

board[i] = digitalRead(j);

}

//e column [18 to 25]

for (i=4, j=18; i<64; i+=8, j++) {

board[i] = digitalRead(j);

}

// f column [41 to 34]

for (i = 5, j=41; i<64; i+=8, j--) {

board[i] = digitalRead(j);

}

// g column [5 to 2 and then 14 to 17]

for (i=6, j=5; j>=2; i+=8, j--) {

board[i] = digitalRead(j);

}

for (i=i, j=14; j<=17; i+=8, j++) {

board[i] = digitalRead(j);

}

// h column [33 to 26]

for (i=7, j=33; i<= 64; i+=8, j--) {

board[i] = digitalRead(j);

}

}

// command used in testing

void printBoard(byte board[]) {

**Serial**.println("--------");

**Serial**.println();

int i,j;

for (i=7; i>=0; i--) {

for (j=i\*8; j< i\*8+8; j++)

**Serial**.print(board[j]);

**Serial**.println();

}

**Serial**.println();

**Serial**.println("--------");

}

# Java Code

### Main.java

public class Main {

public static void main(String[] args){

Arm arm = new Arm();

arm.playArm();

}

}

### Arm.java

public class Arm {

*/\*\**

*\* This class handles the interface while playing the robotic arm*

*\* Note: this class only works with the player playing white and the arm playing black*

*\*/*

public void playArm() {

Position position = new Position(); *//the board position will be stored in this variable*

Computer computer = new Computer(); *//this variable/class is used to determine what move the arm will make*

*//this class is used in communicating with the arduino uno, which facilitates voltage I/O communication with the robotic arm*

UnoSerialConnection arduinoUno = new UnoSerialConnection();

arduinoUno.openPort(); *//the port is opened at the beginning of the program and remains open so that the process of opening and closing it doesn't interfere with the Serial communication*

*// this class is used in communicating with the arduinoMega, which records the binary position of the physical board*

MegaSerialConnection arduinoMega = new MegaSerialConnection();

*// this class is essentially a function that determines what move the player made*

DecodeMove decodeMove = new DecodeMove();

*// these variable arrays are used with "decodeMove" to determine what move the player made*

byte[] startPos = new byte[64];

byte[] endPos = new byte[64];

byte[] changedSquares = new byte[64];

byte[][] bufferArrays;

*//these variables are used in printing what moves each player made*

int start;

int end;

System.*out*.println("You (white) are playing against the robotic arm. \n");

position.readFen(Const.*STARTING\_POSITION*); *//sets the starting position of a chess game*

*// this is the while loop that will loop for every move*

while (true) {

position.printBoard();

System.*out*.println("Make a move on the board, then press the button:");

*// this is section where the information about the board is read and decoded*

bufferArrays = arduinoMega.serialRead(); *//reads the three arrays from the arduino mega*

*// transfers the information into 1D arrays*

for (int i=0; i<64; i++) {

startPos[i] = bufferArrays[0][i];

endPos[i] = bufferArrays[1][i];

changedSquares[i] = bufferArrays[2][i];

}

*// determines the move made by the player*

int playerMove = decodeMove.determineMove(startPos, endPos, changedSquares, Const.*WHITE*);

*//prints out the move the player made along with the new updated position*

start = playerMove/100;

end = playerMove%100;

System.*out*.println("Player moved " + position.indexToCoordinate(start)

+ " to " + position.indexToCoordinate(end));

position.move(position,playerMove);

position.printBoard();

*// checks to see if the game is finished*

*// continues if game isn't over*

*//breaks the loop if it is*

if (computer.isGameOver(position)) {

computer.endingSequence(position);

break;

}

*//Now that the player made his move, the computer will compute a move in response*

double evaluation = computer.alphaBetaA(position, Const.*DEPTH*, Const.*NEGATIVE\_INFINITY*, Const.*POSITIVE\_INFINITY*, false);

*//after the alphaBeta evaluation, the best move is stored in "computer"'s private variable*

int computerMove = computer.getMove();

start = computerMove / 100;

end = computerMove % 100;

*// prints out the computer's move, moves the piece, and prints the new board position*

System.*out*.println("The computer determined the move: " + position.indexToCoordinate(start)

+ " to " + position.indexToCoordinate(end));

System.*out*.println("Computer evaluation: " + evaluation);

*// sends the move command to the arduino uno*

byte[] command = computer.determineArduinoCommand(position, computerMove);

position.move(position,computerMove);

position.printBoard();

System.*out*.println("Sending command: \n");

for (int i = 0; i < 8; i++) {

System.*out*.print(command[i]);

}

System.*out*.println();

arduinoUno.serialLoop(command);

*// This is simply down-time. The arm will be moving the move during this time*

try {

Thread.*sleep*(5000);

} catch (Exception e) {}

*//This waits for the confirmation that the arm finished its move before looping*

arduinoUno.serialRead();

}

arduinoUno.closePort();

}

}

### Pieces.java

*/\*\**

*\* This class represents each square on the chess board.*

*\* It has a piece and a color (or empty)*

*\*/*

public class Pieces {

*/\*\* stored values \*/*

int piece;

int color;

public Pieces(int piece, int color) {

this.piece = piece;

this.color = color;

}

}

### Position.java

import java.util.Arrays;

public class Position {

*// Array of Pieces of size 64*

private Pieces[] board;

*// whose turn it is*

private int turn;

*// values regarding whether white of black can castle*

private boolean whiteKingside;

private boolean blackKingside;

private boolean whiteQueenside;

private boolean blackQueenside;

*// how many moves have been played*

private int halfMoveClock;

private int fullMoveClock;

*// if there is a square that can be moved to via enPassant, that square will be stored here*

private int enPassant;

*// creates an empty board*

public Position() {

board = new Pieces[64];

for (int i=0; i<64; i++)

board[i]= new Pieces(Const.*EMPTY*, Const.*NO\_COLOR*);

}

public Pieces[] getBoard() {

return board;

}

public int getTurn() {

return turn;

}

public int getFullMoveClock() {

return fullMoveClock;

}

public int getHalfMoveClock() {

return halfMoveClock;

}

public int getEnPassant() {

return enPassant;

}

*// this function will take the piece on the starting square to the ending square*

*// it does not check whether the move is legal or not*

public void moveSimple(Position p, int start, int end) {

p.board[end].piece = p.board[start].piece;

p.board[end].color = p.board[start].color;

p.board[start].piece = Const.*EMPTY*;

p.board[start].color = Const.*NO\_COLOR*;

}

*/\*\**

*\* @param p*

*\* starting position*

*\* @param move*

*\* the move that is to be made on the starting position*

*\*/*

public Position getMovedBoard(Position p, int move) {

Position newPosition = new Position();

*// copies all the elements of the position onto the new moved position*

for (int i = 0; i<64; i++) {

newPosition.board[i].piece = p.board[i].piece;

newPosition.board[i].color = p.board[i].color;

newPosition.halfMoveClock = p.halfMoveClock;

newPosition.fullMoveClock = p.fullMoveClock;

newPosition.turn = p.turn;

newPosition.blackQueenside = p.blackQueenside;

newPosition.blackKingside = p.blackKingside;

newPosition.whiteQueenside = p.whiteQueenside;

newPosition.whiteKingside = p.whiteKingside;

newPosition.enPassant = p.enPassant;

}

return move(newPosition, move);

}

*/\*\**

*\* This method updates the board by moving the piece and updating the associated information about the position*

*\* @param p*

*\* The starting position*

*\* @param move*

*\* The move that is to be applied on the position*

*\* @return*

*\* Returns the moved position*

*\*/*

public Position move(Position p, int move) {

int start = move/100;

int startPiece = p.board[start].piece;

int color = p.board[start].color;

int end = move%100;

*/\*\* updates castling rules\*/*

if (startPiece == Const.*KING\_PIECE* && color == Const.*WHITE*) {

if (end == 6 && start == 4 && p.whiteKingside) {

moveSimple(p,7,5);

} else

if (end == 2 && start == 4 && p.whiteQueenside) {

moveSimple(p,0,3);

}

p.whiteKingside = p.whiteQueenside = false;

}

if (startPiece == Const.*KING\_PIECE* && color == Const.*BLACK*) {

if (end == 62 && start == 60 && p.blackKingside) {

moveSimple(p,63,61);

} else

if (end == 58 && start == 60 && p.blackQueenside) {

moveSimple(p,56,59);

}

p.blackQueenside = p.blackKingside = false;

}

if (startPiece == Const.*PAWN\_PIECE* && end == p.enPassant) {

if (end >= 40 && end <= 47) {

p.board[end-8].piece = Const.*EMPTY*;

p.board[end-8].color = Const.*NO\_COLOR*;

}

if (end >= 16 && end <= 23) {

p.board[end+8].piece = Const.*EMPTY*;

p.board[end+8].color = Const.*NO\_COLOR*;

}

}

*// if a piece on [corner square] moved or if a piece on [corner square] was captured, castling isn't allowed anymore*

if (start == 7 || end == 7) *// h1*

p.whiteKingside = false;

if (start == 0 ||end == 0) *// a1*

p.whiteQueenside = false;

if (start == 63 || end == 63) *// h8*

p.blackKingside = false;

if (start == 56 || end == 56) *// a8*

p.blackQueenside = false;

*/\*\* sets enpassant \*/*

if (startPiece == Const.*PAWN\_PIECE* && color == Const.*WHITE* && end-start == 16)

p.enPassant = end - 8;

else if (startPiece == Const.*PAWN\_PIECE* && color == Const.*BLACK* && end - start == -16)

p.enPassant = end + 8;

else

p.enPassant = Const.*EMPTY*;

*/\*\* updates halfmove clock\*/*

if (startPiece == Const.*PAWN\_PIECE* || p.board[end].piece != Const.*EMPTY*)

p.halfMoveClock = -1;

p.halfMoveClock += 1;

*/\*\* updates the actual move\*/*

moveSimple(p,start,end);

*/\*\* updates position if move was a promotion \*/*

if (p.board[end].piece == Const.*PAWN\_PIECE* && end >= 56 && end <= 63)

p.board[end].piece = Const.*QUEEN\_PIECE*;

if (p.board[end].piece == Const.*PAWN\_PIECE* && end <= 7)

p.board[end].piece = Const.*QUEEN\_PIECE*;

*/\*\* updates whose turn it is\*/*

if (p.turn == Const.*BLACK*) {

p.turn = Const.*WHITE*;

p.fullMoveClock += 1;

}

else

p.turn = Const.*BLACK*;

return p;

}

*//returns the common name of a square given the index*

*//e.g. returns "a2" when given index 8*

public String indexToCoordinate(int index) {

String coordinate = "";

char[] chars = {'a', 'b','c','d','e','f','g','h'};

int file = index % 8;

coordinate += chars[file];

int rank = index / 8;

rank++;

coordinate += Integer.*toString*(rank);

return coordinate;

}

*// returns the index form of a square from its common name*

*// e.g. returns 8 when given "a2"*

public int coordinateToIndex(String coordinate) {

int file;

int rank;

char fileChar = coordinate.charAt(0);

file = columnToNumber(fileChar);

rank = Character.*getNumericValue*(coordinate.charAt(1));

return (rank \* 8 - 8 + file);

}

*/\*\**

*\* This method finds all the legal moves in a given position.*

*\**

*\* @param p*

*\* Current position*

*\* @param color*

*\* Whose turn it is*

*\* @param threat*

*\* This is the buffer threat map that will be filled with which squares are under attack and discovered attack*

*\* @return*

*\* Returns an array with legal moves.*

*\* Note: The returned array will order it so that all the moves appear in the front.*

*\* Unused space will be in the back of the array*

*\*/*

public int[] findLegalMoves(Position p, int color, int[] threat) {

int[] legalMoves = new int[Const.*BUFFER\_SIZE*];

if (color == Const.*WHITE*)

threat = threatMap(p,Const.*BLACK*,threat);

else threat = threatMap(p,Const.*WHITE*, threat);

legalMoves = findPossibleMoves(p,color,legalMoves);

boolean isKingThreatened = false;

if (color == Const.*WHITE*)

*// if king is under direct or discovered attack, turn isKingThreatened true*

for (int i = 0; i<64; i++) {

if (p.board[i].piece == Const.*KING\_PIECE* && p.board[i].color == Const.*WHITE* && threat[i] != Const.*NO\_ATTACK*) {

isKingThreatened = true;

break;

}

}

else

*// if king is under direct or discovered attack*

for (int i = 0; i<64; i++)

if (p.board[i].piece == Const.*KING\_PIECE* && p.board[i].color == Const.*BLACK* && threat[i] != Const.*NO\_ATTACK*) {

isKingThreatened = true;

break;

}

int start,end;

for (int i = 0; i<Const.*BUFFER\_SIZE*; i++) {

start = legalMoves[i]/100;

end = legalMoves[i]%100;

*// remove castling moves where the king has to move over a threatened square*

*// and remove moves where the king walks into check.*

if (p.board[start].piece == Const.*KING\_PIECE*) {

if (start == 4 && end == 6 && (threat[5] == Const.*DIRECT\_ATTACK* || threat[4] == Const.*DIRECT\_ATTACK*))

legalMoves[i] = 0;

if (start == 4 && end == 2 && (threat[3] == Const.*DIRECT\_ATTACK* || threat[4] == Const.*DIRECT\_ATTACK*))

legalMoves[i] = 0;

if (start == 60 && end == 62 && (threat[61] == Const.*DIRECT\_ATTACK* || threat[60] == Const.*DIRECT\_ATTACK*))

legalMoves[i] = 0;

if (start == 60 && end == 58 && (threat[59] == Const.*DIRECT\_ATTACK* || threat[60] == Const.*DIRECT\_ATTACK*))

legalMoves[i] = 0;

if (threat[end] == Const.*DIRECT\_ATTACK*)

legalMoves[i] = 0;

}

}

*// if king is threatened, check every move to make sure the king doesn't end up in direct attack in following move*

if (isKingThreatened) {

Position tempPosition;

int[] tempThreatMap = new int[64];

for (int i = 0; i<Const.*BUFFER\_SIZE*; i++) {

if (legalMoves[i] == 0)

continue;

else

{

*// creates a new position for every move and makes sure the king is fine*

tempPosition = getMovedBoard(p,legalMoves[i]);

*//if is white's turn*

if (color == Const.*WHITE*) {

*// find black's threats*

tempThreatMap = threatMap(tempPosition,Const.*BLACK*, tempThreatMap);

for (int i1 = 0; i1<64; i1++) {

*// if white's king is under direct attack, romove the move*

if (tempPosition.board[i1].piece == Const.*KING\_PIECE*

&& tempPosition.board[i1].color == Const.*WHITE*

&& tempThreatMap[i1] == Const.*DIRECT\_ATTACK*) {

legalMoves[i] = 0; *// removes the move*

break;

}

}

}

else {

*// finds white's threats*

tempThreatMap = threatMap(tempPosition,Const.*WHITE*, tempThreatMap);

for (int i1 = 0; i1<64; i1++) {

*// if black's king is under direct attack, remove the move*

if (tempPosition.board[i1].piece == Const.*KING\_PIECE*

&& tempPosition.board[i1].color == Const.*BLACK*

&& tempThreatMap[i1] == Const.*DIRECT\_ATTACK*) {

legalMoves[i] = 0;*//removes the move*

break;

}

}

}

}

}

}

*//sorts the arrays in descending order so that the moves appear in front, and the empty space (integers of value 0) will fill the back*

for (int i = 0; i<64; i++)

legalMoves[i] = -1\*legalMoves[i];

Arrays.*sort*(legalMoves);

for (int i = 0; i<64; i++)

legalMoves[i] = -1\*legalMoves[i];

return legalMoves;

}

*/\*\**

*\* returns an array with every possible move*

*\* illegal moves that come from this function will be removed later on*

*\* note:*

*\* moves will be stored in this format*

*\* the start square index will be the first two digits of the integer*

*\* the end square index will be the final two digits of the integer*

*\* e.g the move "e2e4" will be stored with this integer: 1228*

*\* 12 is the index for "e2", and 28 is the index for "e4"*

*\**

*\* the index map of the board looks like this:*

*\**

*\* 8| 56 57 58 59 60 61 62 63*

*\* 7| 48 49 50 51 52 53 54 55*

*\* 6| 40 41 42 43 44 45 46 47*

*\* 5| 32 33 34 35 36 37 38 39*

*\* 4| 24 25 26 27 28 29 30 31*

*\* 3| 16 17 18 19 20 21 22 23*

*\* 2| 8 9 10 11 12 13 14 15*

*\* 1| 0 1 2 3 4 5 6 7*

*\* a b c d e f g h*

*\* \*/*

public int[] findPossibleMoves(Position p, int color, int[] buffer) {

Arrays.*fill*(buffer,0);

int count = 0; *//count for index in the buffer array*

*// this first loop goes through every square on the board to see where the pieces are*

for (int i = 0; i<64; i++) {

*// if the piece is the same color as whose turn it is*

if (p.board[i].color == color) {

*// sees which piece it is and checks its possible moves*

switch (p.board[i].piece) {

*// if it's a pawn*

case Const.*PAWN\_PIECE*:

*//if the pawn is white and the turn is white*

if (p.board[i].color == Const.*WHITE* && color == Const.*WHITE*) {

*// if the pawn isn't moving off the board and there isn't a piece directly in front of the piece,*

*// moving the pawn up one square is a possible move*

if (i+8<64 && p.board[i+8].piece == Const.*EMPTY*) {

buffer[count] = i\*100 + i+8;

count++;

}

*//if the pawn is on the 2nd rank and there are no pieces one and two squares in front of it, it can move two squares ahead*

if ((i >=8 && i<=15) && (p.board[i+8].piece == Const.*EMPTY* && p.board[i+16].piece == Const.*EMPTY*)) {

buffer[count] = i\*100 + i+16;

count++;

}

*// if the pawn is not the leftmost rank, and (if there's a black piece on it's left diagonal or if it can take enpassant), it can capture there*

if (i % 8 != 0 && (p.board[i+7].color == Const.*BLACK* || i+7 == p.enPassant)) {

buffer[count] = i\*100 + i+7;

count++;

}

*//if the pawn is not on the rightmost rank, and (if there's a black piece on its right diagonal or if it can take enpassant), it can capture it*

if ((i % 8 != 7) && (p.board[i+9].color == Const.*BLACK* || i+9 == p.enPassant)) {

buffer[count] = i\*100 + i+9;

count++;

}

}

if (p.board[i].color == Const.*BLACK* && color == Const.*BLACK*) {

*// if the pawn isn't moving off the board and there isn't a piece directly in front of the piece,*

*// moving the pawn down one square is a possible move*

if (i - 8 >= 0 && p.board[i-8].piece == Const.*EMPTY*) {

buffer[count] = i\*100 + i-8;

count++;

}

*//if the pawn is on the 7th rank and there are no pieces one and two squares in front of it, it can move two squares ahead*

if ((i >=49 && i<=55) && (p.board[i-8].piece == Const.*EMPTY* && p.board[i-16].piece == Const.*EMPTY*)) {

buffer[count] = i\*100 + i-16;

count++;

}

*// if the pawn is not the leftmost rank (from black's perspective), and (if there's a white piece on it's left diagonal or if it can take enpassant), it can capture there*

if (i % 8 != 7 && (p.board[i-7].color == Const.*WHITE* || i-7 == p.enPassant)) {

buffer[count] = i\*100 + i-7;

count++;

}

*//if the pawn is not on the rightmost rank (from black's perspective), and (if there's a white piece on its right diagonal or if it can take enpassant), it can capture it*

if ((i % 8 != 0) && (board[i-9].color == Const.*WHITE* || i-9 == p.enPassant)) {

buffer[count] = i\*100 + i-9;

count++;

}

}

break;

case Const.*KNIGHT\_PIECE*:

for (int i1 = 0; i1 < 8; i1++) {

if (i+Const.*KNIGHT\_MOVES*[i1] < 0

|| i+Const.*KNIGHT\_MOVES*[i1] >=64)

continue;

if (p.board[i+Const.*KNIGHT\_MOVES*[i1]].color != color *// if the proposed end square isn't contained by a piece of the same color*

&& (Math.*abs*( (i%8) - ((i+Const.*KNIGHT\_MOVES*[i1]) % 8)) <=2)) *// if the proposed end square doesn't jump across the board*

{

buffer[count] = i\*100 + (i + Const.*KNIGHT\_MOVES*[i1]);

count++;

}

}

break;

case Const.*BISHOP\_PIECE*:

for (int i1 = 0; i1 < 4; i1++) {

int direction = Const.*BISHOP\_MOVES*[i1];

for (int j = 1; j <9; j++) {

int test = linearMoveCheck(p,direction,color,i,j);

if (test == Const.*TERMINATE*)

break;

*//*

*//if piece of opposite color is on proposed end square, add it to the array of possible moves, and end the current loop on the direction*

if (test == Const.*CAPTURE*)

{

buffer[count] = i\*100 + (i + j\*direction);

count++;

break;

}

else {

buffer[count] = i\*100 + (i+ j\*direction);

count++;

}

}

}

break;

case Const.*ROOK\_PIECE*:

for (int i1 = 0; i1<4; i1++) {

int direction = Const.*ROOK\_MOVES*[i1];

for (int j = 1; j <9; j++) {

int test = linearMoveCheck(p,direction,color,i,j);

if (test == Const.*TERMINATE*)

break;

*//*

*//if piece of opposite color is on proposed end square, add it to the array of possible moves, and end the current loop on the direction*

if (test == Const.*CAPTURE*)

{

buffer[count] = i\*100 + (i + j\*direction);

count++;

break;

}

else {

buffer[count] = i\*100 + (i+ j\*direction);

count++;

}

}

}

break;

case Const.*QUEEN\_PIECE*:

for (int i1 = 0; i1<8; i1++) {

int direction = Const.*QUEEN\_MOVES*[i1];

for (int j = 1; j <9; j++) {

int test = linearMoveCheck(p,direction,color,i,j);

if (test == Const.*TERMINATE*)

break;

*//*

*//if piece of opposite color is on proposed end square, add it to the array of possible moves, and end the current loop on the direction*

if (test == Const.*CAPTURE*)

{

buffer[count] = i\*100 + (i + j\*direction);

count++;

break;

}

else {

buffer[count] = i\*100 + (i+ j\*direction);

count++;

}

}

}

break;

case Const.*KING\_PIECE*:

*// first checks adjacent moves*

for (int i1 = 0; i1<8; i1++) {

int direction = Const.*KING\_MOVES*[i1];

int test = linearMoveCheck(p,direction, color, i, 1);

if (test == Const.*CAPTURE* || test == Const.*CONTINUE*) {

buffer[count] = i\*100 + (i + direction);

count++;

}

}

*// next checks castling*

if (color == Const.*WHITE*

&& p.board[5].piece == Const.*EMPTY*

&& p.board[6].piece == Const.*EMPTY*

&& whiteKingside)

{

buffer[count] = i\*100 + 6; *// e1 to g1*

count++;

}

if (color == Const.*WHITE*

&& p.board[1].piece == Const.*EMPTY*

&& p.board[2].piece == Const.*EMPTY*

&& p.board[3].piece == Const.*EMPTY*

&& whiteQueenside)

{

buffer[count] = i\*100 + 2; *//e1 to c1*

count++;

}

if (color == Const.*BLACK*

&& p.board[61].piece == Const.*EMPTY*

&& p.board[62].piece == Const.*EMPTY*

&& blackKingside)

{

buffer[count] = i\*100 + i+2; *//e8 to g8*

count++;

}

if (color == Const.*BLACK*

&& p.board[57].piece == Const.*EMPTY*

&& p.board[58].piece == Const.*EMPTY*

&& p.board[59].piece == Const.*EMPTY*

&& blackQueenside)

{

buffer[count] = i\*100 + i-2; *//e8 to c8*

count++;

}

break;

}

}

}

return buffer;

}

*/\*\**

*\* This function is used in checking bishop, rook, queen, and king moves that extend over a length*

*\* returning 0 "Const.TERMINATE" means that the loop is being terminated*

*\* returning 1 "Const.CAPTURE" means that the move is a capture move, so it will be the last move in that loop*

*\* returning 2 "Const.CONTINUE" means that the move is possible and the loop can continue*

*\*/*

public int linearMoveCheck(Position p, int direction,int color, int i, int j) {

if (Math.*abs*(((i + (j-1)\*direction) % 8) - ((i + j\*direction) % 8)) > 1 *// if proposed end square wraps around the board horizontally*

|| i+ j\*direction >=64 *// if proposed end square goes out of bounds*

|| i+ j\*direction < 0 *// if the proposed end square goes out of bounds*

|| p.board[i+ j\*direction].color == color) *// if proposed end square is already contained by a piece of the same color*

return Const.*TERMINATE*;

*//if piece of opposite color is on proposed end square, add it to the array of possible moves, and end the current loop on the direction*

if ((color == Const.*WHITE* && p.board[i+j\*direction].color == Const.*BLACK*)

|| (color == Const.*BLACK* && p.board[i+j\*direction].color == Const.*WHITE*))

{

return Const.*CAPTURE*;

}

else {

return Const.*CONTINUE*;

}

}

*/\*\**

*\* modified version of LinearMoveCheck()*

*\* This function returns "Const.CAPTURE" even if it comes across its own colored piece*

*\* This is to say that the piece is defended\*/*

public int LinearThreatCheck (Position p,int direction,int color, int i, int j) {

if (Math.*abs*(((i + (j-1)\*direction) % 8) - ((i + j\*direction) % 8)) > 1 *// if proposed end square wraps around board horizontally*

|| i+ j\*direction >=64 *// if proposed end square goes out of bounds*

|| i+ j\*direction < 0) *//if the proposed end square goes out of bounds*

return Const.*TERMINATE*;

*//if piece of opposite color is on proposed end square, add it to the array of possible moves, and end the current loop on the direction*

if (p.board[i+j\*direction].piece != Const.*EMPTY*)

if ((color == Const.*WHITE* && p.board[i+j\*direction].color == Const.*BLACK*)

|| (color == Const.*BLACK* && p.board[i+j\*direction].color == Const.*WHITE*))

{

return Const.*DIFFERENT\_COLOR*;

}

if ((color == Const.*WHITE* && p.board[i+ j\*direction].color == Const.*WHITE*)

|| (color == Const.*BLACK* && p.board[i+ j\*direction].color == Const.*BLACK*))

{

return Const.*SAME\_COLOR*;

}

else {

return Const.*CONTINUE*;

}

}

*/\*\* returns an array with every square that the specified color is currently attacking.*

*\* Squares that are under direct attack have values of 1.*

*This is done so that finding illegal moves is more efficient*

*\*/*

public int[] threatMap(Position p, int color, int[] map) {

Arrays.*fill*(map,0);

for (int i = 0; i <64; i++) {

if (p.board[i].color == color) { *//if piece is the color we specified*

switch (p.board[i].piece) {

case Const.*PAWN\_PIECE*:

if (color == Const.*WHITE*) {

*// if the pawn isn't on the left edge (a-file), pawn is threatening its left diagonal*

if (i%8 != 0)

map[i+7] = Const.*DIRECT\_ATTACK*;

*//if the pawn isn't on the right edge (h-file), pawns is threatening its right diagonal*

if ((i%8) !=7)

map[i+9] = Const.*DIRECT\_ATTACK*;

}

else if (color == Const.*BLACK*) {

*//if the pawn isn't on the a-file, pawn is threatening its right diagonal*

if (i%8 != 0)

map[i-9] = Const.*DIRECT\_ATTACK*;

*// if the pawn isn't on the h-file, pawn is threatening its left diagonal*

if ((i%8) != 7)

map[i-7] = Const.*DIRECT\_ATTACK*;

}

break;

case Const.*KNIGHT\_PIECE*:

for (int i1 = 0; i1 <8; i1++) {

if (i+Const.*KNIGHT\_MOVES*[i1] < 0 *// if proposed square goes out of bounds*

|| i+Const.*KNIGHT\_MOVES*[i1] >=64 *// if proposed square goes out of bounds*

|| Math.*abs*( (i%8) - ((i+Const.*KNIGHT\_MOVES*[i1]) % 8)) > 2) *// if the proposed square wraps around the board horizontally*

continue;

map[i+Const.*KNIGHT\_MOVES*[i1]] = Const.*DIRECT\_ATTACK*; *// else, add the square to the threat map*

}

break;

case Const.*BISHOP\_PIECE*:

for (int i1 = 0; i1 <4; i1++) {

int direction = Const.*BISHOP\_MOVES*[i1];

int level = Const.*DIRECT\_ATTACK*; *//the default value is a direct attack. Going past a different colored piece reduces it to Const.DISCOVERED\_ATTACK*

for (int j = 1; j <9; j++) {

int test = LinearThreatCheck(p,direction,color,i,j);

*// if the proposed square goes out of bounds, end the loop now*

if (test == Const.*TERMINATE*)

break;

*// if the proposed square has a piece of the same color, and there is a direct line of sight to it (no obstructions),*

*// add that square to the threat map and end the loop*

if (test == Const.*SAME\_COLOR* && level == Const.*DIRECT\_ATTACK*) {

map[i+ j\*direction] = Const.*DIRECT\_ATTACK*;

break;

}

*// if the proposed square has a piece of the same color, but there isn't a direct line of sight to it (obstructed),*

*// end the loop*

else if (test == Const.*SAME\_COLOR* && level == Const.*DISCOVERED\_ATTACK*) {

break;

}

*//if the proposed square has a piece of a different color, and piece has a direct line of sight to it,*

*// add that square to the threat map, and continue the loop with a demoted "discovered attack" level*

else if (test == Const.*DIFFERENT\_COLOR* && level == Const.*DIRECT\_ATTACK*) {

map[i+ j\*direction] = Const.*DIRECT\_ATTACK*;

level = Const.*DISCOVERED\_ATTACK*;

}

*// if the proposed square has a piece of a different color, and the piece doesn't have a direct line of sight to it,*

*// add that square and end the loop*

else if (test == Const.*DIFFERENT\_COLOR* && level == Const.*DISCOVERED\_ATTACK*) {

map[i+ j\*direction] = Const.*DISCOVERED\_ATTACK*;

break;

}

*// else, add the square and continue the loop without demoting the level of threat*

else if (test == Const.*CONTINUE*) {

map[i+ j\*direction] = level;

}

}

}

break;

case Const.*ROOK\_PIECE*:

for (int i1 = 0; i1 <4; i1++) {

int direction = Const.*ROOK\_MOVES*[i1];

int level = Const.*DIRECT\_ATTACK*; *//the default value is a direct attack. Going past a different colored piece reduces it to Const.DISCOVERED\_ATTACK*

for (int j = 1; j <9; j++) {

int test = LinearThreatCheck(p,direction,color,i,j);

*// if the proposed square goes out of bounds, end the loop*

if (test == Const.*TERMINATE*)

break;

*// if the proposed square has a piece of the same color, and there is a direct line of sight to it (no obstructions),*

*// add that square to the threat map and end the loop*

if (test == Const.*SAME\_COLOR* && level == Const.*DIRECT\_ATTACK*) {

map[i+ j\*direction] = Const.*DIRECT\_ATTACK*;

break;

}

*// if the proposed square has a piece of the same color, but there isn't a direct line of sight to it (obstructed),*

*// end the loop*

else if (test == Const.*SAME\_COLOR* && level == Const.*DISCOVERED\_ATTACK*) {

break;

}

*//if the proposed square has a piece of a different color, and the piece has a direct line of sight to it,*

*// add that square to the threat map, and continue the loop with a demoted "discovered attack" level*

else if (test == Const.*DIFFERENT\_COLOR* && level == Const.*DIRECT\_ATTACK*) {

map[i+ j\*direction] = Const.*DIRECT\_ATTACK*;

level = Const.*DISCOVERED\_ATTACK*;

}

*// if the proposed square has a piece of a different color, and it doesn't have a direct line of sight to it,*

*// add that square and end the loop*

else if (test == Const.*DIFFERENT\_COLOR* && level == Const.*DISCOVERED\_ATTACK*) {

map[i+ j\*direction] = Const.*DISCOVERED\_ATTACK*;

break;

}

*// else, add the square and continue the loop without demoting the level of threat*

else if (test == Const.*CONTINUE*) {

map[i+ j\*direction] = level;

}

}

}

break;

case Const.*QUEEN\_PIECE*:

for (int i1 = 0; i1 <8; i1++) {

int direction = Const.*QUEEN\_MOVES*[i1];

int level = Const.*DIRECT\_ATTACK*; *//the default value is a direct attack. Going past a different colored piece reduces it to Const.DISCOVERED\_ATTACK*

for (int j = 1; j <9; j++) {

int test = LinearThreatCheck(p,direction,color,i,j);

*// if the proposed square goes out of bounds, end the loop*

if (test == Const.*TERMINATE*)

break;

*// if the proposed square has a piece of the same color, and there is a direct line of sight to it (no obstructions),*

*// add that square to the threat map and end the loop*

if (test == Const.*SAME\_COLOR* && level == Const.*DIRECT\_ATTACK*) {

map[i+ j\*direction] = Const.*DIRECT\_ATTACK*;

break;

}

*// if the proposed square has a piece of the same color, but there isn't a direct line of sight to it (obstructed),*

*// end the loop*

else if (test == Const.*SAME\_COLOR* && level == Const.*DISCOVERED\_ATTACK*) {

break;

}

*//if the proposed square has a piece of a different color, and the piece has a direct line of sight to it,*

*// add that square to the threat map, and continue the loop with a demoted "discovered attack" level*

else if (test == Const.*DIFFERENT\_COLOR* && level == Const.*DIRECT\_ATTACK*) {

map[i+ j\*direction] = Const.*DIRECT\_ATTACK*;

level = Const.*DISCOVERED\_ATTACK*;

}

*// if the proposed square has a piece of a different color, and it doesn't have a direct line of sight to it,*

*// add that square and end the loop*

else if (test == Const.*DIFFERENT\_COLOR* && level == Const.*DISCOVERED\_ATTACK*) {

map[i+ j\*direction] = Const.*DISCOVERED\_ATTACK*;

break;

}

*// else, add the square and continue the loop without demoting the level of threat*

else if (test == Const.*CONTINUE*) {

map[i+ j\*direction] = level;

}

}

}

break;

*// all the squares adjacent to the king are under direct attack*

case Const.*KING\_PIECE*:

for (int i1 = 0; i1 <8; i1++) {

int direction = Const.*KING\_MOVES*[i1];

if (i+direction < 0 || i+direction >=64) *//if the square is in-bounds*

continue;

map[i+direction] = Const.*DIRECT\_ATTACK*;

}

}

}

}

return map;

}

*// turns 'a' to 0, 'b' to 1, ... , 'h' to 7*

public int columnToNumber(char c) {

return c - 'a';

}

*// returns the associated piece given a character representation of the piece*

public Pieces characterToPiece(char c) {

Pieces piece = new Pieces(Const.*EMPTY*, Const.*NO\_COLOR*);

switch (Character.*toLowerCase*(c)) {

case 'p':

piece.piece = Const.*PAWN\_PIECE*;

break;

case 'n':

piece.piece = Const.*KNIGHT\_PIECE*;

break;

case 'b':

piece.piece = Const.*BISHOP\_PIECE*;

break;

case 'r':

piece.piece = Const.*ROOK\_PIECE*;

break;

case 'q':

piece.piece = Const.*QUEEN\_PIECE*;

break;

case 'k':

piece.piece = Const.*KING\_PIECE*;

break;

}

if (Character.*isUpperCase*(c))

piece.color = Const.*WHITE*;

else

piece.color = Const.*BLACK*;

return piece;

}

*// returns the character representation of a piece*

*// e.g. valueToSymbol(PAWN\_PIECE) returns PAWN\_SYMBOL ('p')*

public char pieceToSymbol(int piece, int color) {

char symbol;

switch (piece) {

case Const.*EMPTY*:

symbol = Const.*EMPTY\_SYMBOL*;

break;

case Const.*PAWN\_PIECE*:

symbol = Const.*PAWN\_SYMBOL*;

break;

case Const.*KNIGHT\_PIECE*:

symbol = Const.*KNIGHT\_SYMBOL*;

break;

case Const.*BISHOP\_PIECE*:

symbol = Const.*BISHOP\_SYMBOL*;

break;

case Const.*ROOK\_PIECE*:

symbol = Const.*ROOK\_SYMBOL*;

break;

case Const.*QUEEN\_PIECE*:

symbol = Const.*QUEEN\_SYMBOL*;

break;

case Const.*KING\_PIECE*:

symbol = Const.*KING\_SYMBOL*;

break;

default:

return (char) -1;

}

if (color == Const.*WHITE*)

return Character.*toUpperCase*(symbol);

else *//if color is black*

return Character.*toLowerCase*(symbol);

}

*//prints the current board position*

public void printBoard() {

Pieces currentPiece;

System.*out*.println("Board Position:\n\n");

for (int y = 7; y >=0; y--) {

for (int x = 0; x <= 7; x++) {

currentPiece = this.board[y\*8 + x];

System.*out*.print(pieceToSymbol(currentPiece.piece, currentPiece.color));

System.*out*.print(" ");

}

System.*out*.println();

}

}

*//reads a fen string and sets the board up according to that fen*

public void readFen(String fen) {

for (int i = 0; i<64; i++) {

this.board[i].color = Const.*NO\_COLOR*;

this.board[i].piece = Const.*EMPTY*;

}

int stringIndex = 0;

int x, y; */\*\* x represents the file, y represents the column \*/*

char currentChar;

*/\*\* rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1 \*/*

*// goes through the first part of the fen ("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR") and assigns each square on the board its piece*

for (y = 7; y >= 0; y--) {

for (x = 0; x <= 7; x++) {

currentChar = fen.charAt(stringIndex);

if (currentChar == '/') {

x--; *//this essentially "skips" this character*

}

else if (Character.*isDigit*(currentChar))

x += Character.*getNumericValue*(currentChar) - 1;

else if (Character.*isAlphabetic*(currentChar))

this.board[8\*y + x] = characterToPiece(currentChar);

stringIndex++;

}

}

stringIndex++;

currentChar = fen.charAt(stringIndex);

*/\*\* whose turn it is \*/*

if (currentChar == 'w')

this.turn = Const.*WHITE*;

else

this.turn = Const.*BLACK*;

*/\*\* castling parameters \*/*

stringIndex += 2;

currentChar = fen.charAt(stringIndex);

if (currentChar == '-') {

this.whiteKingside =

this.blackKingside =

this.whiteQueenside =

this.blackQueenside = false;

stringIndex++;

}

else

*// reads the castling part of the fen*

for (currentChar = fen.charAt(stringIndex); Character.*isAlphabetic*(currentChar); currentChar = fen.charAt(stringIndex+1), stringIndex++) {

switch (currentChar) {

case 'K':

whiteKingside = true;

break;

case 'Q':

whiteQueenside = true;

break;

case 'k':

blackKingside = true;

break;

case 'q':

blackQueenside = true;

break;

}

}

*// reads the enpassant square*

stringIndex++;

currentChar = fen.charAt(stringIndex);

if (currentChar == '-')

this.enPassant = Const.*EMPTY*;

else {

int xCoordinate = columnToNumber(currentChar);

stringIndex++;

currentChar = fen.charAt(stringIndex);

int yCoordinate = Character.*getNumericValue*(currentChar) - 1;

this.enPassant = xCoordinate + yCoordinate\*8;

}

*// reads the half move clock*

stringIndex += 2;

int len = 0;

while (true) {

currentChar = fen.charAt(stringIndex);

if (Character.*isDigit*(currentChar))

len++;

else

break;

stringIndex++;

}

this.halfMoveClock = 0;

stringIndex -= len;

for (int i = len; i > 0; i--, stringIndex++) {

currentChar = fen.charAt(stringIndex);

*// halfMoveClock = 3 \* 10^(2-1) + 1 \* 10^(1-1) = 31*

this.halfMoveClock += Character.*getNumericValue*(currentChar) \* Math.*pow*(10,len-1);

}

stringIndex += 1;

len = fen.length() - stringIndex;

this.fullMoveClock = 0;

for (int i = len; i > 0; i--, stringIndex++) {

currentChar = fen.charAt(stringIndex);

*// halfMoveClock = 3 \* 10^(2-1) + 1 \* 10^(1-1) = 31*

this.fullMoveClock += Character.*getNumericValue*(currentChar) \* Math.*pow*(10,i-1);

}

}

}

### Computer.java

import java.util.Locale;

import java.util.Scanner;

*/\*\**

*\* This class deals with determining what move to make in a given position.*

*\* Also known as the chess engine.*

*\*/*

public class Computer {

private int positionsExamined = 0; *//this counts how many positions were examined*

private int[] map = new int[64]; *// this is a temporary buffer threatMap that is used in determining the legal moves in a position*

private int move; *// this variable stores the computer's move after doing the alphaBeta search algorithm*

public int getMove() {

return move;

}

public int getPositionsExamined() {

return positionsExamined;

}

*/\*\**

*\* This is the search algorithm in determining what move to play in a certain position.*

*\* It uses ideas from minimax and alpha-beta pruning to find the move*

*\**

*\* @param p*

*\* This is the position that the computer is to search from.*

*\* @param depth*

*\* This is how many plies ("moves") deep the algorithm will search.*

*\* @param alpha*

*\* This is the alpha value. It is assigned as negative infinity, and it increases in value.*

*\* Its purpose is to indicate when it is pointless to continue searching a certain branch*

*\* as the minimizing player.*

*\* @param beta*

*\* This is the beta value. It is assigned positive infinity, and it decreases in value.*

*\* Its purpose is to indicate when it is pointless to continue searching a certain branch*

*\* as the maximizing player.*

*\* @param maximizingPlayer*

*\* This tells the computer whether it's trying to maximize its score (play for white),*

*\* or whether it is trying to minimize its score (play for black).*

*\* @return*

*\* Returns the evaluation of the position.*

*\* A positive score means the computer thinks white is ahead.*

*\* A negative score means the computer things black is ahead.*

*\* An equal score means the computer thinks the position is equal.*

*\* Returned values around +/- 10000 indicate that there's a forced mate.*

*\*/*

public double alphaBetaA(Position p, int depth, double alpha, double beta, boolean maximizingPlayer) {

this.positionsExamined++;

int[] legalMoves;

if (p.getTurn() == Const.*WHITE*)

legalMoves = p.findLegalMoves(p,Const.*WHITE*, p.threatMap(p,Const.*BLACK*,map));

else legalMoves = p.findLegalMoves(p,Const.*BLACK*, p.threatMap(p,Const.*WHITE*,map));

legalMoves = orderMoves(p,legalMoves);

*// if there are no legal moves, returns 10000-offset (White won), -10000+offset (Black won), or 0 (stalemate)*

if (legalMoves[0] == 0) {

return status(p);

}

*// if the end of the depth is reached, return the static evaluation of the position*

if (depth == 0)

return staticEvaluation(p);

*// is a draw if the 50-move rule is surpassed*

*// the rule is that 50-moves happen in a row where no pawn moves nor captures occur, someone can claim a draw*

*// Technically, players can keep playing if both agree, but here, I just assume that someone will claim a draw*

*// The number is 100 because the halfmove clock increases every ply, and 50 "moves" is equal to 100 "plies"*

if (p.getHalfMoveClock() >= 100)

return 0.0;

double value;

if (maximizingPlayer) {

value = Const.*NEGATIVE\_INFINITY*;

for (int i = 0; i<Const.*BUFFER\_SIZE*; i++) {

if (legalMoves[i] == 0)

break;

double eval = alphaBetaA(p.getMovedBoard(p,legalMoves[i]),depth-1,alpha, beta,false);

if (eval > value && depth == Const.*DEPTH*)

this.move = legalMoves[i];

value = Double.*max*(value, eval);

if (value > beta)

break;

alpha = Double.*max*(alpha, value);

}

}

else {

value = Const.*POSITIVE\_INFINITY*;

for (int i = 0; i<Const.*BUFFER\_SIZE*; i++) {

if (legalMoves[i] == 0)

break;

double eval = alphaBetaA(p.getMovedBoard(p,legalMoves[i]),depth-1,alpha, beta,true);

if (eval < value && depth == Const.*DEPTH*)

this.move = legalMoves[i];

value = Double.*min*(value,eval);

if (value < alpha)

break;

beta = Double.*min*(beta, value);

}

}

return value;

}

*/\*\**

*\* This function orders the moves so that more promising moves are evaluated first*

*\* Sets moves that are checks and captures first*

*\* Sets moves that are just checks second*

*\* Sets moves that are just captures third*

*\* Sets quiet moves last*

*\**

*\* @param p*

*\* The position that is being branched.*

*\* @param moves*

*\* These are the legal moves in the given position.*

*\* @return*

*\* Returns the legal move list in an order that is more likely to have better moves in the front*

*\* and worse moves in the back.*

*\*/*

public int[] orderMoves(Position p,int[] moves) {

*// This array associates each legal move with a classification as to what kind of move it is:*

*// quiet move, capture, check, or check+capture*

int[][] classifiedMoves = new int[Const.*BUFFER\_SIZE*][2];

for (int i = 0; i < Const.*BUFFER\_SIZE*; i++) {

int move = moves[i];

classifiedMoves[i][0] = move;

int start = move /100;

int end = move%100;

Pieces endPiece = p.getBoard()[end];

*// if the ending square contains a piece (is not empty)*

boolean isCapture = endPiece.piece != Const.*EMPTY*;

int defendingColor;

int attackingColor = p.getTurn();

if (attackingColor == Const.*WHITE*)

defendingColor = Const.*BLACK*;

else defendingColor = Const.*WHITE*;

int [] threatMap = new int[64];

threatMap = p.threatMap(p.getMovedBoard(p,move),attackingColor,threatMap);

boolean isKingThreatened = false;

for (int j = 0; j<64; j++) {

*// if the piece is a king, it's the defending color, and it's under direct attack, the king is threatened*

if (p.getBoard()[j].piece == Const.*KING\_PIECE* && p.getBoard()[j].color == defendingColor) {

if (threatMap[j] == Const.*DIRECT\_ATTACK*)

isKingThreatened = true;

break;

}

}

classifiedMoves[i][0] = move;

if (move == 0)

classifiedMoves[i][1] = 0;

else if (!isKingThreatened && !isCapture)

classifiedMoves[i][1] = Const.*QUIET\_MOVE*;

else if (isKingThreatened && !isCapture)

classifiedMoves[i][1] = Const.*CHECK\_MOVE*;

*//note: the value is multiplied/added by the value of the piece because moves that capture higher-valued pieces should be evaluated first*

*//note: it shouldn't matter whether the value of the piece is added or multiplied*

else if (!isKingThreatened && isCapture)

classifiedMoves[i][1] = Const.*CAPTURE\_MOVE* \* (int) Math.*abs*(getPieceValue(endPiece));

else if (isKingThreatened && isCapture)

classifiedMoves[i][1] = Const.*CHECK\_AND\_CAPTURE\_MOVE* + (int) Math.*abs*(getPieceValue(endPiece)) ;

}

return sortMoves(classifiedMoves,moves);

}

*// sorts the 2D array based on what its classifications are*

public int[] sortMoves(int[][] classifiedMoves, int[]moves) {

*// this is the sort algorithm that orders the move based on how strong its classification is*

*// 1. Checks+captures*

*// 2. Checks*

*// 3. Captures*

*// 4. Quiet moves*

for (int i = 0; i<Const.*BUFFER\_SIZE*; i++) {

int max=-1;

int index = 0;

for (int j = i; j<Const.*BUFFER\_SIZE*; j++) {

if (classifiedMoves[j][1] > max) {

index = j;

max = classifiedMoves[j][1];

}

}

int tempMove = classifiedMoves[index][0];

int tempValue = classifiedMoves[index][1];

classifiedMoves[index][0] = classifiedMoves[i][0];

classifiedMoves[index][1] = classifiedMoves[i][1];

classifiedMoves[i][0] = tempMove;

classifiedMoves[i][1] = tempValue;

}

*//converts the 2D array back to the 1D array with the part that contained the moves*

for (int i = 0; i< Const.*BUFFER\_SIZE*; i++) {

moves[i] = classifiedMoves[i][0];

}

return moves;

}

*// this function determines if the game is over*

public boolean isGameOver(Position p) {

int[] legalMoves;

int[] threatMap = new int[64];

legalMoves = p.findLegalMoves(p, p.getTurn(), threatMap);

return legalMoves[0] == 0;

}

*// this function determines the result of a given position*

*// is only called if the game is over*

public void endingSequence(Position p) {

double result = status(p);

if (result > 0)

System.*out*.println("White is victorious");

else if (result < 0)

System.*out*.println("Black is victorious");

else

System.*out*.println("It's a draw");

}

*/\*\**

*\* this function returns an array of bytes (size 8) that will be sent to the robotic arm via it's I/O box*

*\* the command will look something like this*

*\* {5,2,5,4,0,0,0,0}*

*\* the first byte represents the x coordinate of the start square (in this example, 5 is the e column)*

*\* the second byte represents the y coordinate of the start square (in this example, 2 is row 2*

*\* the third byte represents the x coordinate of the end square (in this example, 5 is the e column)*

*\* the fourth byte represent the y coordinate of the end square (in this example, 4 is row 4)*

*\* the fifth byte represents the "special move"*

*\* 0 means it is a normal move*

*\* 1 means it is a capture*

*\* 2 means it is a casting move*

*\* 3 means it is en passant*

*\* the sixth byte represents if the move is a promotion*

*\* 0 means no promotion*

*\* 1 means promotion*

*\* the seventh byte represents if the starting piece is tall*

*\* 0 for short piece*

*\* 1 for tall piece*

*\* (this is required so that the arm gripper doesn't crash into tall piece or come up short on grabbing short pieces)*

*\* the eighth byte represents if the ending/captured piece is tall*

*\* 0 for short piece*

*\* 1 for tall piece*

*\* (this is the same idea as the seventh byte, but for pieces that are getting captured)*

*\**

*\* note: the position is supposed to be in the position before the move is made*

*\*/*

public byte[] determineArduinoCommand(Position p, int move) {

byte[] command = new byte[8];

*//these are the variables, in order, that will be a part of the command*

int startX;

int startY;

int endX;

int endY;

int specialMove;

int isPromotion;

int isTall;

int isCaptureTall;

int start = move / 100;

int end = move % 100;

*// converts the indexes into coordinates*

startX = (start % 8) + 1; *// e.g. (16 % 8) + 1 = 0 + 1 = 1 which is the a-file*

startY = (start / 8) + 1; *// e.g. (16 / 8) + 1 = 2 + 1 = 3 which is the 3rd rank*

endX = (end % 8) + 1;

endY = (end / 8) + 1;

int startPiece = p.getBoard()[start].piece;

int endPiece = p.getBoard()[end].piece;

*// determines the special move*

*// if the end square is the enpassant square, and a pawn is moving there, the special move is an en passant*

if (end == p.getEnPassant() && startPiece == Const.*PAWN\_PIECE*)

specialMove = Const.*ENPASSANT\_COMMAND*;

*// if the starting piece is a king, and it is traveling two squares away, the special move is castling*

else if (startPiece == Const.*KING\_PIECE*

&& Math.*abs*(start-end) == 2)

specialMove = Const.*CASTLING\_COMMAND*;

*// if the ending square contains a piece, the move is a capture*

else if (endPiece != Const.*EMPTY*)

specialMove = Const.*CAPTURE\_COMMAND*;

else

specialMove = Const.*NORMAL\_MOVE\_COMMAND*;

*// determines if move is a promotion*

*// if the end square is on the first or eighth ranks, and the piece moved is a pawn, isPromotion is true*

if (((end / 8) + 1 == 1 || (end / 8) + 1 == 8)

&& startPiece == Const.*PAWN\_PIECE*)

isPromotion = Const.*TRUE\_COMMAND*;

else isPromotion = Const.*FALSE\_COMMAND*;

*// determines if the piece being moved is tall*

if (startPiece == Const.*QUEEN\_PIECE* || startPiece == Const.*KING\_PIECE*)

isTall = Const.*TRUE\_COMMAND*;

else isTall = Const.*FALSE\_COMMAND*;

*// determines if the piece being captured (if it is) is tall*

if (endPiece == Const.*QUEEN\_PIECE*) *//we don't need to check the king because capturing the king is an illegal move*

isCaptureTall = Const.*TRUE\_COMMAND*;

else isCaptureTall = Const.*FALSE\_COMMAND*;

command[0] = (byte) startX;

command[1] = (byte) startY;

command[2] = (byte) endX;

command[3] = (byte) endY;

command[4] = (byte) specialMove;

command[5] = (byte) isPromotion;

command[6] = (byte) isTall;

command[7] = (byte) isCaptureTall;

return command;

}

*// this function should only be called when there are no legal moves*

*// returns whether white won, black won, or is stalemate*

public double status(Position p) {

int turnColor = p.getTurn();

int attackingColor;

if (turnColor == Const.*WHITE*)

attackingColor = Const.*BLACK*;

else attackingColor = Const.*WHITE*;

int c; *// coefficient that will determine whether white or black won (won't matter if it is stalemate)*

if (attackingColor == Const.*BLACK*)

c = -1;

else c = 1;

*// this value is important in mating when there's few pieces on a board (e.g. KQ v k).*

*// Essentially, this value will make moves that go towards a mate slightly more valuable than moves that simply stall the mate count.*

*// In other words, it will cause a mate in 2 position to be more valuable than a mate in 3 position...*

*// ...because the mate in three position will have a higher half move clock value*

double halfMoveOffset = -c \* 0.05 \* p.getHalfMoveClock();

int[] map = new int[64];

int[] threatMap = p.threatMap(p,attackingColor,map);

boolean isKingUnderDirectAttack = false;

for (int i = 0; i <64; i++) {

*// if the piece is a king ,and it's the color of the player who is supossed to move*

if (p.getBoard()[i].piece == Const.*KING\_PIECE*

&& p.getBoard()[i].color == turnColor

&& threatMap[i] == Const.*DIRECT\_ATTACK*)

return c \* Const.*POSITIVE\_INFINITY* + halfMoveOffset; *//the halfmove offset is an arbitrarily small number that will make postions with quicker checkmate have higher value than postion with lower checkmates*

}

*// if the king isn't under direct attack, it's a stalemate.*

return 0.0;

}

*// This function evaluates the position statically*

*// The evaluation considers how many piece each side has,*

*// and the position of those pieces according based on the piece-square tables*

public double staticEvaluation(Position p) {

double eval = 0.0;

*//first, considers the material values of how many pieces each side has*

for (int i = 0; i<64; i++) {

Pieces currentPiece = p.getBoard()[i];

*// this coefficient determines will either make the values increase (better for white) or decrease (better for black) depending on the kind of piece it is*

int c;

if (currentPiece.color == Const.*WHITE*)

c = 1; *//positive direction (white)*

else

c = -1; *// negative direction (black)*

*//adds the material value of the piece times the direction (white/positive direction or black/negative direction)*

eval += c \* getPieceValue(currentPiece);

*// uses the piece-square tables to evaluate how effective the pieces' positions on the board are*

switch (currentPiece.piece) {

case Const.*EMPTY*:

break;

case Const.*PAWN\_PIECE*:

if (currentPiece.color == Const.*BLACK*) {

eval += c \* Const.*PAWN\_TABLE*[63-i];

}

else eval += c\* Const.*PAWN\_TABLE*[i];

break;

case Const.*KNIGHT\_PIECE*:

if (currentPiece.color == Const.*WHITE*) {

eval += c \* Const.*KNIGHT\_TABLE*[i];

}

else eval += c \* Const.*KNIGHT\_TABLE*[63-i];

break;

case Const.*BISHOP\_PIECE*:

if (currentPiece.color == Const.*WHITE*) {

eval += c \* Const.*BISHOP\_TABLE*[i];

}

else eval += c\* Const.*BISHOP\_TABLE*[63-i];

break;

case Const.*ROOK\_PIECE*:

if (currentPiece.color == Const.*WHITE*) {

eval += c \* Const.*ROOK\_TABLE*[i];

}

else eval += c\* Const.*ROOK\_TABLE*[63-i];

break;

*//note: the queen table requires two different tables according to the colors because*

*// queens tend to be best positioned on c2-b3 (white) and c7-b6 (black)*

*// those squares cause the tables to not be symmetric, so it requires two different tables*

case Const.*QUEEN\_PIECE*:

if (currentPiece.color == Const.*WHITE*) {

eval += c\* Const.*WHITE\_QUEEN\_TABLE*[i];

}

else eval += c \* Const.*BLACK\_QUEEN\_TABLE*[63-i];

break;

*// note: there are two different piece-square tables for the king because*

*// in the early stages of the game, the king is better protected,*

*// but in the later stages of the game, the king is better centralized*

case Const.*KING\_PIECE*:

if (p.getFullMoveClock() < 35) {

if (currentPiece.color == Const.*WHITE*)

eval += c\* Const.*KING\_MIDDLEGAME\_TABLE*[i];

else eval += c \* Const.*KING\_MIDDLEGAME\_TABLE*[63-i];

}

else {*// if fullmoveclock >=35*

if (currentPiece.color == Const.*WHITE*)

eval += c\* Const.*KING\_ENDGAME\_TABLE*[i];

else eval += c \* Const.*KING\_ENDGAME\_TABLE*[63-i];

}

}

}

return eval;

}

public double getPieceValue(Pieces piece) {

switch (piece.piece) {

case Const.*EMPTY*:

return 0;

case Const.*PAWN\_PIECE*:

return Const.*PAWN\_VALUE*;

case Const.*KNIGHT\_PIECE*:

return Const.*KNIGHT\_VALUE*;

case Const.*BISHOP\_PIECE*:

return Const.*BISHOP\_VALUE*;

case Const.*ROOK\_PIECE*:

return Const.*ROOK\_VALUE*;

case Const.*QUEEN\_PIECE*:

return Const.*QUEEN\_VALUE*;

case Const.*KING\_PIECE*:

return Const.*KING\_VALUE*;

}

return Const.*ERROR*;

}

}

### DecodeMove.java

import java.util.Arrays;

*/\*\**

*\* This class is dedicated to determining the move made by the player on the physical chess board*

*\* It takes the position before the move was made, the position after it was made,*

*\* and the squares that experienced change during the move.*

*\* With that information, it decodes what move must have taken place.*

*\* This class is a solution to the problem: "How can I determine the player's move if I'm only given a binary value of the position?"*

*\* In other words, "I only know whether a piece occupies a square or not. How do I determine the moves without knowing what the pieces are?"*

*\* Note: this a very unstable program and crashes if the player fumbles with the pieces.*

*\* For example, if the player initially makes one move, but then changes his mind and makes a capture move,*

*\* the program will crash because the "count" number will be wrong, and an IndexArrayOutOfBounds error occurs*

*\*/*

public class DecodeMove {

*// this returns the move that was played on the physical board by the player*

public int determineMove(byte[] startPos, byte[] endPos, byte[] changedSquares, int color) {

int start = -1; *//original square of piece's move*

int end = -1; *//destination square of piece's move*

*// first, checks to see if move was a simple move or promotion without capturing (no captures, enpassants or castling)*

int count =0;

*//counts how many differences are in the position*

for (int i=0; i<64; i++) {

if (startPos[i] != endPos[i])

count++;

}

if (count == 0) *//if no change was made (no move was made), return an error of -1*

return Const.*ERROR*;

if (count == 2) { *//simple move was made (one square turned off, one square turned on)*

for (int i=0; i<64; i++) {

if(startPos[i]==1 && endPos[i]==0) *//if the square was on before the move, and off after the move, that is the start square*

start = i;

else if(startPos[i] ==0 && endPos[i]==1) *//if square was off before the move, and on after the move, that is the end square*

end = i;

}

if (start == -1 || end == -1) *//if either one of the values doesn't update, return an error*

return Const.*ERROR*;

return start\*100 + end; *//return the move in the 4-digit format*

}

if (count == 1) { *// move was a capture (one square turned off, one square remains on)*

for (int i=0; i<64; i++) {

if (startPos[i] == 1 && endPos[i] == 0) *//if the square was on before the move, and off after the move, that is the start square*

start = i;

}

for (int i=0; i<64;i++) {

if (changedSquares[i] == 1 && i != start) { *//if the square experienced change during the move, and it wasn't the start square, that is the end square*

end = i;

break;

}

}

return start\*100 + end; *//return the move in the 4-digit format*

}

if (count==3) { *//move was enpassant (two squares turned off, one square turned on)*

int[] changedCoordinates = {-1, -1, -1}; *//this variable will hold the three squares that change occured*

for (int i = 0, j = 0; i<64; i++) { *//assigns changedCoordinates[] with the squares that changed*

if (changedSquares[i] == 1) {

changedCoordinates[j] = i;

j++;

}

}

*// if not all three squares were assigned, return an error*

Arrays.*sort*(changedCoordinates);

if (changedCoordinates[0] == -1)

return Const.*ERROR*;

*//this is a lengthy section that determines how the en passant happened*

*// note: a change of 7 and a change of 9 indicate diagonal captures*

*// those are the direction values that we check to determine pawn captures*

if (color == Const.*WHITE*) {

if (changedCoordinates[0] + 7 == changedCoordinates[2]) { *//enpassant move where white pawn captures leftwards*

start = changedCoordinates[0];

end = changedCoordinates[2];

} else

if (changedCoordinates[0] + 9 == changedCoordinates[2]) { *//enpassant move where white pawn captures rightwards*

start = changedCoordinates[0];

end = changedCoordinates[2];

} else

if (changedCoordinates[1] + 7 == changedCoordinates[2]) { *//enpassant move where white pawn captures leftwards*

start = changedCoordinates[1];

end = changedCoordinates[2];

} else

if (changedCoordinates[1] + 9 == changedCoordinates[2]) { *//enpassant move where white pawn captures leftwards*

start = changedCoordinates[1];

end = changedCoordinates[2];

} else

return Const.*ERROR*;

}

if (color == Const.*BLACK*) {

if (changedCoordinates[1] - 9 == changedCoordinates[0]) { *//enpassant move where black pawn captures leftwards relative to white perspective*

start = changedCoordinates[1];

end = changedCoordinates[0];

} else

if (changedCoordinates[1] - 7 == changedCoordinates[0]) { *//enpassant move where black pawn captures rightwards relative to white perspective*

start = changedCoordinates[1];

end = changedCoordinates[0];

} else

if (changedCoordinates[2] - 9 == changedCoordinates[0]) { *//enpassant move where black pawn captures leftwards relative to white perspective*

start = changedCoordinates[2];

end = changedCoordinates[0];

} else

if (changedCoordinates[2] - 7 == changedCoordinates[0]) { *//enpassant move where black pawn captures rightwards relative to white perspective*

start = changedCoordinates[2];

end = changedCoordinates[0];

} else

return Const.*ERROR*;

}

if (color != Const.*WHITE* && color != Const.*BLACK*) *// if we are evaluating an empty square*

return Const.*ERROR*;

return start\*100 + end; *//return move in 4-digit format*

}

if (count >= 4) { *// if is castling (two squares turn on, two squares turn off)*

int[] changedCoordinates = {-1, -1, -1, -1}; *//this array will hold the four squares that changed*

for (int i = 0, j = 0; i<64; i++) {

if (changedSquares[i] == 1) {

changedCoordinates[j] = i;

j++;

}

}

Arrays.*sort*(changedCoordinates);

if (changedCoordinates[0] == -1) *//if there aren't 4 changes, return an error*

return Const.*ERROR*;

if (color == Const.*WHITE* && changedCoordinates[1] == 2) { *//if it was white's turn and move involved c1 square*

start = 4; *// e1 square*

end = 2; *// c1 square meaning queenside castling*

} else

if (color == Const.*WHITE* && changedCoordinates[2] == 6) {*// if it was white's turn and move involved g1 square*

start = 4; *// e1 square*

end = 6; *// g1 square meaning kingside castling*

} else

if (color == Const.*BLACK* && changedCoordinates[1] == 58) { *// if it was black's turn and move involved c8 square*

start = 60; *// e8 square*

end = 58; *// c8 square meaning queenside castling*

} else

if (color == Const.*BLACK* && changedCoordinates[2] == 62) { *// if it was black's turn and move involved g8 square*

start = 60; *// e8 square*

end = 62; *// g8 square meaning kingside castling*

} else

return Const.*ERROR*;

return start\*100 + end; *//return move in 4-digit format*

}

return Const.*ERROR*; *//return an error if a move couldn't be determined*

}

}

### UnoSerialConnection.java

import com.fazecast.jSerialComm.SerialPort;

*/\*\**

*\* This class deals with Serial Communication with the Arduino UNO*

*\* Whenever a command needs to be sent to the arm, it is first sent to the Arduino UNO*

*\* This class is used to send that command to the arduino*

*\*/*

public class UnoSerialConnection {

private SerialPort port;

public UnoSerialConnection() {

port = SerialPort.*getCommPort*("COM3"); *// make sure this is the port used by the UNO*

}

*// this method has written with help from Caleb Hay, a software engineer who used to go to my high school*

*// This loop waits for the Arduino UNO to read the command for what move to make*

public void serialLoop(byte[] command) {

while(true) {

System.*out*.println("Sending to Arduino...");

int bytesWritten = port.writeBytes(command, 8);

port.flushIOBuffers();

if (bytesWritten > 0) {

break;

}

try { java.lang.Thread.*sleep*(200); }

catch(Exception e) {}

}

}

*// this method reads a byte from the Arduino UNO*

*// This is used to confirm that the arm finished its move, and that the board is ready to read the next move*

public byte[] serialRead() {

byte[] buffer = {0};

while (true) {

*//System.out.println("Looking for bytes to read...");*

if (this.port.bytesAvailable() > 0) {

port.readBytes(buffer,1);

}

if (buffer[0] != 0) {

break;

}

try { java.lang.Thread.*sleep*(200); }

catch(Exception e) {}

}

System.*out*.println("serialRead from arduino returned: " + buffer[0]);

return buffer;

}

public void openPort() {

if(!port.openPort()) {

System.*out*.println("Could not open Arduino UNO port");

return;

} else {

System.*out*.println("Opened Arduino UNO port");

}

}

public boolean closePort() {

return port.closePort();

}

}

### MegaSerialConnection.java

import com.fazecast.jSerialComm.SerialPort;

*/\*\**

*\* This class does the Serial communication between the Laptop and the Arduino MEGA*

*\* The Arduino MEGA is the controller that records the physical board position and any changes that are made*

*\* So, this class is used when the player's move is recorded*

*\*/*

public class MegaSerialConnection {

*// returns the three arrays that comes from the arduino mega*

*// those three arrays are the board positions*

public byte[][] serialRead() {

*// open the serial port*

SerialPort port = SerialPort.*getCommPort*("COM11"); *// make sure this is the correct port name before executing*

port.openPort();

port.setComPortTimeouts(SerialPort.*TIMEOUT\_READ\_BLOCKING*, 1000, 0);

*// // Flushes buffer so that old bytes don't mess with new bytes*

*// port.flushIOBuffers();*

*//this delay ensures that opening the port doesn't affect the Serial communication*

*// 2 seconds is about how long the time between opening the port and reading/writing data has to be in order to work*

try {

Thread.*sleep*(2000);

} catch (Exception e) {}

*//waits for bytes to be available*

*//this is finished when the player presses the button*

while (port.bytesAvailable() <=0) {

try {

Thread.*sleep*(10);

} catch (Exception e) {}

}

*// read the three arrays*

byte[] startPos = *readBytes*(port, 64);

byte[] endPos = *readBytes*(port, 64);

byte[] changedSquares = *readBytes*(port, 64);

*// print the arrays*

System.*out*.println("Start Position Array:");

*printBytes*(startPos);

System.*out*.println("End Position Array");

*printBytes*(endPos);

System.*out*.println("Changed Squares Array");

*printBytes*(changedSquares);

byte[][] arrays = new byte[3][64];

for (int i=0; i<64; i++) {

arrays[0][i] = startPos[i];

arrays[1][i] = endPos[i];

arrays[2][i] = changedSquares[i];

}

*// close the serial port*

port.closePort();

return arrays;

}

*// this method was written with help from ChatGPT*

*// reads bytes coming from the arduinoMega*

private static byte[] readBytes(SerialPort port, int numBytes) {

byte[] buffer = new byte[numBytes];

int bytesRead = 0;

while (bytesRead < numBytes) {

int numRead = port.readBytes(buffer, numBytes - bytesRead);

if (numRead == -1) {

throw new RuntimeException("Error reading from serial port");

}

bytesRead += numRead;

}

return buffer;

}

*// this is a method used in testing*

*// prints the arrays recorded from the Arduino MEGA*

private static void printBytes(byte[] arr) {

for (int i = 0; i < arr.length; i++) {

System.*out*.print(arr[i] + " ");

}

System.*out*.println();

System.*out*.println("--------");

System.*out*.println();

int i,j;

for (i=7; i>=0; i--) {

for (j=i\*8; j< i\*8+8; j++)

System.*out*.print(arr[j]);

System.*out*.println();

}

System.*out*.println();

System.*out*.println("--------");

}

}

### Const.java

*/\*\**

*\* This class holds all of this project's constant values*

*\*/*

public class Const {

*// error value*

public static final int *ERROR* = -1;

*// these are the values for the commands to send to the Arduino UNO*

*// also found on the code for the Arduino Uno*

public static final int *NORMAL\_MOVE\_COMMAND* = 0;

public static final int *CAPTURE\_COMMAND* = 1;

public static final int *CASTLING\_COMMAND* = 2;

public static final int *ENPASSANT\_COMMAND* = 3;

public static final int *TRUE\_COMMAND* = 1;

public static final int *FALSE\_COMMAND* = 0;

*// size of buffers such as the move list array*

public static final int *BUFFER\_SIZE* = 70;

*//constant for search depth in the alphaBeta minimax algorithm*

public static final int *DEPTH* = 5;

*//constant for the starting position fen*

public static final String *STARTING\_POSITION* = "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1";

*// values that represent the pieces*

public static final int *EMPTY* = 0;

public static final int *PAWN\_PIECE* = 1;

public static final int *KNIGHT\_PIECE* = 2;

public static final int *BISHOP\_PIECE* = 3;

public static final int *ROOK\_PIECE* = 4;

public static final int *QUEEN\_PIECE* = 5;

public static final int *KING\_PIECE* = 6;

*// values that represent the color*

public static final int *NO\_COLOR* = 0;

public static final int *WHITE* = 1;

public static final int *BLACK* = 2;

*// characters that represent the pieces*

public static final char *EMPTY\_SYMBOL* = '-';

public static final char *PAWN\_SYMBOL* = 'p';

public static final char *KNIGHT\_SYMBOL* = 'n';

public static final char *BISHOP\_SYMBOL* = 'b';

public static final char *ROOK\_SYMBOL* = 'r';

public static final char *QUEEN\_SYMBOL* = 'q';

public static final char *KING\_SYMBOL* = 'k';

*//unit directions for pieces*

*//note: the pawn moves are not here because they are not used*

public static final int[] *KNIGHT\_MOVES* =

{10, 17, 15, 6,

-10, -17, -15, -6};

*//note: the bishop, rook, and queen arrays are simply unit vectors.*

*// that means that these only contain the steps in one direction*

public static final int[] *BISHOP\_MOVES* =

{9,7,-9,-7};

public static final int[] *ROOK\_MOVES* =

{1,8,-1,-8};

public static final int[] *QUEEN\_MOVES* =

{1,9,8,7,-1,-9,-8,-7};

public static final int[] *KING\_MOVES* =

{1,9,8,7,-1,-9,-8,-7};

*// constants used with LinearMoveCheck*

public static final int *TERMINATE* = 0;

public static final int *CAPTURE* = 1;

public static final int *CONTINUE* = 2;

*// constants unique to LinearThreatCheck*

public static final int *SAME\_COLOR* = 3;

public static final int *DIFFERENT\_COLOR* = 4;

*// constants used with the threat map*

public static final int *NO\_ATTACK* = 0;

public static final int *DIRECT\_ATTACK* = 1;

public static final int *DISCOVERED\_ATTACK* = 2;

*//constants used in evaluating static positions*

*// these values were taken from this article*

*// https://www.chessprogramming.org/Simplified\_Evaluation\_Function*

public static final double *PAWN\_VALUE* = 100;

public static final double *KNIGHT\_VALUE* = 320;

public static final double *BISHOP\_VALUE* = 330;

public static final double *ROOK\_VALUE* = 500;

public static final double *QUEEN\_VALUE* = 900;

public static final double *KING\_VALUE* = 20000;

*//constants used in ordering moves*

public static final int *QUIET\_MOVE* = 1;

public static final int *CHECK\_MOVE* = 2000;

public static final int *CAPTURE\_MOVE* = 2;

public static final int *CHECK\_AND\_CAPTURE\_MOVE* = 2000;

*// these are the arbitrarily large numbers used in evaluating the position*

public static final double *NEGATIVE\_INFINITY* = -10000;

public static final double *POSITIVE\_INFINITY* = 10000;

*// piece-square tables that are based on the tables found at https://www.chessprogramming.org/Simplified\_Evaluation\_Function*

*//These tables are for White*

*//To make them for black, reverse the order of the for loop (63 to 0)*

public static final int[] *PAWN\_TABLE* =

*// These arrays are reflected over x-axis (flipped vertically) compared to a regular board orientation.*

*// this is done so that the array is compatible with the index form of the board*

*//top left represents a1, top right represents h1, bottom left represents a8, bottom right reps h8*

{

0, 0, 0, 0, 0, 0, 0, 0,

5, 10, 10,-20,-20, 10, 10, 5,

5, -5,-10, 0, 0,-10, -5, 5,

0, 0, 0, 20, 20, 0, 0, 0,

5, 5, 10, 25, 25, 10, 5, 5,

10, 10, 20, 30, 30, 20, 10, 10,

50, 50, 50, 50, 50, 50, 50, 50,

0, 0, 0, 0, 0, 0, 0, 0

};

public static final int[] *KNIGHT\_TABLE* =

{

-50,-40,-30,-30,-30,-30,-40,-50,

-40,-20, 0, 5, 5, 0,-20,-40,

-30, 5, 10, 15, 15, 10, 5,-30,

-30, 0, 15, 20, 20, 15, 0,-30,

-30, 5, 15, 20, 20, 15, 5,-30,

-30, 0, 10, 15, 15, 10, 0,-30,

-40,-20, 0, 0, 0, 0,-20,-40,

-50,-40,-30,-30,-30,-30,-40,-50

};

public static final int[] *BISHOP\_TABLE* =

{

-20,-10,-10,-10,-10,-10,-10,-20,

-10, 5, 0, 0, 0, 0, 5,-10,

-10, 10, 10, 10, 10, 10, 10,-10,

-10, 0, 10, 10, 10, 10, 0,-10,

-10, 5, 5, 10, 10, 5, 5,-10,

-10, 0, 5, 10, 10, 5, 0,-10,

-10, 0, 0, 0, 0, 0, 0,-10,

-20,-10,-10,-10,-10,-10,-10,-20

};

public static final int[] *ROOK\_TABLE* =

{

0, 0, 0, 5, 5, 0, 0, 0,

-5, 0, 0, 0, 0, 0, 0, -5,

-5, 0, 0, 0, 0, 0, 0, -5,

-5, 0, 0, 0, 0, 0, 0, -5,

-5, 0, 0, 0, 0, 0, 0, -5,

-5, 0, 0, 0, 0, 0, 0, -5,

5, 10, 10, 10, 10, 10, 10, 5,

0, 0, 0, 0, 0, 0, 0, 0

};

public static final int[] *WHITE\_QUEEN\_TABLE* =

{

-20,-10,-10, -5, -5,-10,-10,-20,

-10, 0, 5, 0, 0, 0, 0,-10,

-10, 5, 5, 5, 5, 5, 0,-10,

0, 0, 5, 5, 5, 5, 0, -5,

-5, 0, 5, 5, 5, 5, 0, -5,

-10, 0, 5, 5, 5, 5, 0,-10,

-10, 0, 0, 0, 0, 0, 0,-10,

-20,-10,-10, -5, -5,-10,-10,-20

};

*// This table is different from WHITE\_QUEEN\_TABLE[] in that the squares c7 and b6 are promoted rather than c2 and b3*

*// This will be read backwards, so the extra valued squares changed from indexes (10 and 17) to (20 and 29)*

public static final int[] *BLACK\_QUEEN\_TABLE* =

{

-20,-10,-10, -5, -5,-10,-10,-20,

-10, 0, 0, 0, 0, 5, 0,-10,

-10, 0, 5, 5, 5, 5, 5,-10,

0, 0, 5, 5, 5, 5, 0, -5,

-5, 0, 5, 5, 5, 5, 0, -5,

-10, 0, 5, 5, 5, 5, 0,-10,

-10, 0, 0, 0, 0, 0, 0,-10,

-20,-10,-10, -5, -5,-10,-10,-20

};

*//this table is used for positions early in the game because it's better to have a safe king at the start of the game*

public static final int[] *KING\_MIDDLEGAME\_TABLE* =

{

20, 30, 10, 0, 0, 10, 30, 20,

20, 20, 0, 0, 0, 0, 20, 20,

-10,-20,-20,-20,-20,-20,-20,-10,

-20,-30,-30,-40,-40,-30,-30,-20,

-30,-40,-40,-50,-50,-40,-40,-30,

-30,-40,-40,-50,-50,-40,-40,-30,

-30,-40,-40,-50,-50,-40,-40,-30,

-30,-40,-40,-50,-50,-40,-40,-30

};

*// this table is used for positions later in the game, because you want an active king during the endgame*

public static final int[] *KING\_ENDGAME\_TABLE* =

{

-50,-30,-30,-30,-30,-30,-30,-50,

-30,-30, 0, 0, 0, 0,-30,-30,

-30,-10, 20, 30, 30, 20,-10,-30,

-30,-10, 30, 40, 40, 30,-10,-30,

-30,-10, 30, 40, 40, 30,-10,-30,

-30,-10, 20, 30, 30, 20,-10,-30,

-30,-20,-10, 0, 0,-10,-20,-30,

-50,-40,-30,-20,-20,-30,-40,-50

};

}