```
////////////////////////////////////Appendix 1////////////////////////////////////
//
//  TicTacToeC
//
//  Created on 6/26/18.
//  Copyright ? 2018 Victor, Kristoff, Hanut, Kamin. All rights reserved.
//

/*Sensors and Motors setup guide
    motorA  large motor for board rack and pinion
    sensor1 touch sensor at the end of the rack and pinion

    motorB  large motor driving the arm rack
    sensor2 touch sensor at the end of the arm

    motorC  motor to drive the ball dispenser
    sensor3 is color sensor

    sensor4 is touch sensor for user*/

typedef struct {
    //Struct to hold a 3x3 game board/state
    char array[3][3];
} structarray;

//Written by Victor
int max(int a,int b){
    /*  Helper function to find max
            Arguments: two ints, a and b
            Returns: larger int
    */
    if(a > b){
        return a;
    }
    return b;
}

//Written by Victor
int min(int a,int b){
    /*  Helper function to find min
            Arguments: two ints, a and b
            Returns:smaller int
    */
    if(a < b){
        return a;
    }
    return b;
}
```

```
////////////////////////////////Hardware

//ME101 Assignmnet Function
TEV3Buttons get_button_number(){
    //Returns button press
    while (!getButtonPress(buttonAny)){}
    TEV3Buttons buttonSave = buttonNone;
    for (TEV3Buttons button = buttonUp;button <= buttonLeft && buttonSave ==
     buttonNone; button++)
        if (getButtonPress(button))
        buttonSave = button;
    while(getButtonPress(buttonAny)){}
    return buttonSave;
}

//Written by Hanut
void zeroArm(){
    //Zeros the arm
    motor[motorB] = -28; //fastest speed with no grind
    while (SensorValue[S2] == 0){
    }
    motor[motorB] = 0;
    nMotorEncoder[motorB] = 0;
}

//Written by Hanut
void zeroBoard(){
    //Zeros the board
    //negative moves board "forward"
    motor[motorA] = -50;
    while (SensorValue[S1] == 0){
    }
    motor[motorA] = 0;
    nMotorEncoder[motorA] = 0;
}

//Written by Kamin
void zeroBoth(){
    //Helper function to zero both the board and the arm
    zeroBoard();
    zeroArm();
}

//Written by Kamin
void moveBoard(int row, int ballOrSensor){
    /*
            Colour sensor is 2cm wide
            large motor for board rack and pinion is motorA
            touch sensor at the end of the rack and pinion is sensor1
            gear driving the rack is 2cm diameter, radius is 1
```

```
            Arguments:  row          -- row position
                        ballOrSensor -- position to release ball or use colour
                         sensor
    */

    float rowPosition[3] = {0, 0, 0};
    if (ballOrSensor == 0){////////////////////////////Ball position
        rowPosition[0] = 0.0;
        rowPosition[1] = 3.3;//check
        rowPosition[2] = 6.6;//check
    } else {/////////////////////////////////////////Sensor position
        rowPosition[0] = 2.5;
        rowPosition[1] = 5.6;
        rowPosition[2] = 8.8; //this could be greater
    }
    const float ENC_LIMIT = rowPosition[row] * (360 / (2 * PI * 1));
    motor[motorA] = 50;
    while (nMotorEncoder[motorA] < ENC_LIMIT){}
    motor[motorA] = 0;
}

//Written by Kamin
void moveArm(int col, int ballOrSensor){
    /*
            large motor driving the arm rack is motorB
            touch sensor at the end of the arm is sensor2
            large gear diamter is 4cm, radius is 2cm
            ball is 0, sensor is 1

            Arguments:  col           -- col position
                        ballOrSensor  -- position to release ball or use colour
                         sensor
    */
    float colPosition[3] = {0,0,0};
    if (ballOrSensor == 0){////////////////////////////Ball position
        colPosition[0] = 0.0;
        colPosition[1] = 4.4;
        colPosition[2] = 8.4;
    } else {/////////////////////////////////////////Sensor position
        colPosition[0] = 0.5;
        colPosition[1] = 4.7;
        colPosition[2] = 8.4;
    }

    const int ENC_LIMIT = colPosition[col] * (360 / (2 * PI * 2));

    motor[motorB] = 50;
    while (nMotorEncoder[motorB] < ENC_LIMIT){}
    motor[motorB] = 0;
}
```

```
//Written by Hanut
void ballDispenser(){
    /*
            Helper function releases one ball onto the arm
            motor to drive the ball dispenser is motorC
            color sensor is S3
    */
    nMotorEncoder[motorC] = 0;
    motor[motorC] = 20;
    while(nMotorEncoder[motorC] < 120){}
    motor[motorC] = 0;
}

//written by Kamin
void computerBall(int row, int col){
    /*
            Function moves arm and board to desired position and releases ball
            Arguments:  row    -- row position
                        col    -- col position

    */
    zeroBoth();
    moveArm(col, 0);
    moveBoard(row, 0);
    ballDispenser();
    wait1Msec(1600); //Time for ball to roll down arm
}

//Written by Kristoff
void scanBoard(structarray & returnState){
    /*
            Function scans the current game states and stores it in the returnState
             array
            Argument:   returnState     --array to store scanned state
    */
    SensorType[S3] = sensorEV3_Color;
    wait1Msec(50);
    SensorMode[S3] = modeEV3Color_Color;
    wait1Msec(50);

    zeroBoth();

    for (int row = 0; row < 3; row++){
        zeroArm();
        for (int col = 0; col < 3; col++){
            moveBoard(row, 1);
            wait1Msec(50);
            moveArm(col,1);
            wait1Msec(50);
            //white is 2?, green is 3, yellow is 7
            //Sensor values determined by testing
```

```
            if(SensorValue[S3] == 3 || SensorValue[S3] == 1){
                returnState.array[col][row] = 'o';
            }else if (SensorValue[S3] == 4 || SensorValue[S3] == 7){
                returnState.array[col][row] = 'x';
            }else {
                returnState.array[col][row] = '_';
            }
        }
    }
}

//Written by Kristoff
void moveBoardToPlayer(){
    /*
        Move the board and arm for player to access the board
    */
    zeroBoth();
    moveArm(0, 0);
    moveBoard(2, 1); //Can't move board more
}

//Written by Victor
byte checkStatus(structarray previousState, structarray returnState, byte & rowMove,
 byte & colMove){
    /* Function checks if human move on physical board was valid
    Arguments:  checkStatus struct  -- current game state
                returnState struc   -- state after human move
                rowMove             -- variable passed by reference to store human
                 move row index
                colMove             -- variable passed by reference to store human
                 move col index

    Returns:    status              -- board status after move
                an integer value from 1-3
                1. player move was valid
                2. player did not make a move
                3. cheating was detected (illegal move(s) made))
    */
    byte playerMove = 0;
    byte cheating = 0;
    for(int row = 0; row < 3; row++){
        for (int col = 0; col < 3; col++){
            if (previousState.array[row][col] == '_' && returnState.array[row][col]
             == 'o'){
                rowMove = row;
                colMove = col;
                playerMove += 1;
            }
            else if (previousState.array[row][col] != returnState.array[row][col]){
                cheating += 1;
            }
```

```
            //redundant check used for testing purposes
            if (previousState.array[row][col] == 'x' && returnState.array[row]
             [col] != 'x'){
                cheating += 10;
            }
        }
    }

    byte status = 0;
    if (playerMove == 1){ //one change and the change was the player's move
        status = 1;
    }
    if (playerMove == 0){ //no move made
        status = 2;
    }
    if (playerMove > 1 || cheating > 0){ //more than 1 move, and/or there was
     cheating
        status = 3;
    }
    return status;
}

////////////////////////////////////////Gameplay
//Written by Victor
void printBoard(structarray b){
        /*
            Helper function to print board
            Argument: structarray b -- 3x3 board struc
        */
    eraseDisplay();
        displayBigTextLine(1, " %c | %c | %c\n",b.array[0][0],b.array[0]
         [1],b.array[0][2]);
        displayBigTextLine(3, "---+---+---\n");
        displayBigTextLine(5, " %c | %c | %c\n",b.array[1][0],b.array[1]
         [1],b.array[1][2]);
        displayBigTextLine(7, "---+---+---\n");
        displayBigTextLine(9, " %c | %c | %c\n",b.array[2][0],b.array[2]
         [1],b.array[2][2]);
}

//Written by Victor
byte evaluate(structarray &board){
    /*Static Evaluation Function
            Argument: 3x3 board struct
    */
    for(byte i=0;i < 3;i++){ //checks horizontal wins
        if((board.array[i][0] != '_') && (board.array[i][0]==board.array[i][1]) &&
         (board.array[i][2]==board.array[i][1])){
            if(board.array[i][0] == 'x'){
                return 10;
            }else if(board.array[i][0] =='o'){
```

```
                return -10;
            }
        }
    }
    for(byte i=0;i < 3;i++){ //checks vertical wins
        if((board.array[0][i] != '_') && (board.array[0][i]==board.array[1][i]) &&
         (board.array[2][i]==board.array[1][i])){
            if(board.array[0][i] == 'x'){
                return 10;
            }else if(board.array[0][i] =='o'){
                return -10;
            }
        }
    }
        //check first diagonal
    if((board.array[0][0] != '_') && (board.array[0][0]==board.array[1][1]) &&
     (board.array[2][2]==board.array[1][1])){
        if(board.array[0][0] == 'x'){
            return 10;
        }else if(board.array[0][0] =='o'){
            return -10;
        }
    }
        //check second diagonal
    if((board.array[0][2] != '_') && (board.array[0][2]==board.array[1][1]) &&
     (board.array[2][0]==board.array[1][1])){
        if(board.array[0][2] == 'x'){
            return 10;
        }else if(board.array[0][2] =='o'){
            return -10;
        }
    }
    return 0;
}

//Written by Victor
byte noMovesLeft(structarray &board){
/*
        Helper function to determine if there are any moves left in the current game
        Argument: board struct  -- game state to evaluate
*/
    for(byte i = 0;i < 3;i++){
        for(byte j = 0;j < 3;j++){
            if(board.array[i][j] == '_'){
                return 0;
            }
        }
    }
    return 1;
}
```

```
//Written by Victor
int alphaBeta(structarray &board, byte depth, byte isMax, int alpha, int beta){
        /* Alpha Beta Pruned Minimax function
        Arguments:  board struct    -- game state to evaluate
                                depth               -- depth to evaluate
                                isMax                   -- Maximizing or minimizing
                                 player score
                                alpha               -- Lower limit
                                beta                    -- Upper limit
                                Initially alpha is negative infinity and beta is
                                 positive infinity,
                                i.e. both players start with their worst possible
                                 score.
        Returns:        bestValue           -- best score
        */
    byte score = evaluate(board);
    if(score == 10){
        return score - depth;
    }
    if(score == -10){
        return score + depth;
    }
    if(noMovesLeft(board) != 0){
        return 0;
    }
    if(isMax != 0){
        int bestValue = -1000;
        for(byte i = 0; i < 3;i++){
            for(byte j = 0;j < 3;j++){
                if(board.array[i][j] == '_'){
                    board.array[i][j] = 'x';
                    int val = alphaBeta(board,depth + 1 ,0,alpha,beta);
                    bestValue = max(bestValue,val);
                    alpha = max(bestValue,alpha); // minimizing player, alpha
                     pruning
                    //cout<<"In Max "<<bestValue<<endl;
                    //printBoard(board);
                    board.array[i][j] = '_';
                    if(beta <= alpha)
                        break;
                }
                if(beta <= alpha)
                    break;
            }
        }
        return bestValue;
    }else{
        int bestValue = 1000;
        for(byte i = 0; i < 3;i++){
            for(byte j = 0;j < 3;j++){
                if(board.array[i][j] == '_'){
```

```
                    board.array[i][j] = 'o';
                    int val = alphaBeta(board,depth + 1 ,1,alpha,beta);
                    bestValue = min(bestValue,val);
                    beta = min(bestValue,beta); //maximizing player, beta pruning
                    //cout<<"In Min "<<bestValue<<endl;
                    //printBoard(board);
                    board.array[i][j] = '_';
                    if(beta <= alpha)
                        break;
                }
                if(beta <= alpha)
                    break;
            }
        }
        return bestValue;
    }
}

//Written by Victor
void findBestMove(structarray & board, byte & rowReturn, byte & colReturn){
        /*
        Function finds best move for robot for the given game state
        Arguments:  board struct        -- current game state
                                rowReturn       -- Pass by reference to get best row
                                 move
                                colReturn       -- Pass by reference to get best col
                                 move
        */
    int bestValue = -1000;
    int alpha = -1000;
    int beta = 1000;
    for(byte i = 0;i < 3; i++){
        for(byte j =0;j < 3;j++){
            if(board.array[i][j] == '_'){
                board.array[i][j] = 'x';
                int moveValue = alphaBeta(board,0,0,alpha,beta);
                board.array[i][j] = '_';
                //cout <<i<<" "<<j<<" "<<moveValue<<endl;
                if(moveValue > bestValue){
                    rowReturn = i;
                    colReturn = j;
                    bestValue = moveValue;
                }
            }
        }
    }
}
```

```
//Written by Victor
void copyBoard(structarray & board, structarray & returnState){
    /*
        Helper function to copy a 3 by 3 array
    */
    for (int row = 0; row < 3; row++){
        for (int col = 0; col < 3; col++){
            board.array[row][col] = returnState.array[row][col]; //copies
             returnState to board
        }
    }
}

//Written by Victor
void initializeBoard(structarray & board){
    /*
        Helper function to initialize an empty game board
    */
    for(byte i =0 ;i < 3;i++){
            for(byte j =0;j< 3;j++){
                    board.array[i][j]='_';
            }
    }
}

//Written by Victor
byte gamePlay(){
        /*
        Function drives tic tac toe gameplay
        Returns:    byte    -- integer value that indicates to main function the
         status of the game
                    1. sucessful game was completed (1 returned)
                    2. no player move was detected in the alloted time (2 returned)
                    3. cheating was detected at some point in the gameplay (0
                     returned)
        */
        structarray board;
    initializeBoard(board);
    byte move = 0; //keeps track of how many total moves have been played on the
     board
    byte row = 0, col = 0;
    //printBoard(board);
    byte counter = 0;
    bool reminder = true;
    //variable to keep track of if there should be a reminder if no move if made
    ////////////////////////////////////////Get if user wants to go first
        displayBigTextLine(1, "Computer: O");
        displayBigTextLine(3, "You: X");
        displayBigTextLine(5, "Play (1)st");
```

```
    displayBigTextLine(7, "or (2)nd ?");
    displayBigTextLine(9, "Left (1)");
    displayBigTextLine(11, "Right (2)");
    TEV3Buttons one = get_button_number();

    if (one == buttonLeft){
        counter = 0;
    }
    if (one == buttonRight){
        counter = 1;
    }
    eraseDisplay();
    printBoard(board);
    ////////////////////////////////////////////////////////////////////////////
     ////
while(!noMovesLeft(board)&& move < 9 ){
    counter++;
    if(counter%2 == 1){
                    displayBigTextLine(11, "Please make");
                    displayBigTextLine(13, "a move");
            moveBoardToPlayer();

            time1[T1] = 0;
                    //wait for player to make move
                    while(SensorValue[S4] == 0 && time1[T1] < 60000){ //one
                     minute is 60000
                    };
                    //if player doesn't respond the second time, end game
                    if (time1[T1] >= 60000 && reminder == false){
                        return 2;
                    }

                    structarray returnState;
                    initializeBoard(returnState);
                    displayBigTextLine(11, "
                                ");//This is to erase individual lines
                    displayBigTextLine(13, "
                                ");
                    displayBigTextLine(11, "Scanning");
                    scanBoard(returnState);

                    byte status = 0;
                    status = checkStatus(board, returnState, row, col);
                    // 0  is error?
                    if (status == 1){
                        // valid move
                        returnState.array[row][col] = 'o';
                        displayTextLine(12,"

                         ");
```

```
                    displayBigTextLine(12, "Valid Move!");
                    reminder = true;
                    wait1Msec(2000);
                    copyBoard(board, returnState);
                } else if (status == 2){
                //invalid
                    displayTextLine(12,"

                     ");
                    displayBigTextLine(12, "Make a move.");
                    reminder = false;
                    setSoundVolume(90);
                    playSound(soundBeepBeep);
                    wait1Msec(2000);
      counter--;
                } else if (status == 3) {
                //cheating
                    displayTextLine(12,"

                     ");
                    displayTextLine(1,"

                     ");
                    displayBigTextLine(12, "Cheating");
                    displayBigTextLine(14, "Detected.");
                    wait1Msec(2000);
                    return 0;
                }
} else {////////////////////////////////////////////////////////Computer
 move
        displayBigTextLine(11, "Computing");
        displayBigTextLine(13, "Bot Move");
    if (move == 0){
            //Preprogrammed move if computer goes first. Minimax takes too
             long to find the first move
        row = 1; col = 1;
        board.array[row][col] = 'x';
                        computerBall(1, 1);
    } else {
    byte rowReturn = 0;
    byte colReturn = 0;
    findBestMove(board,rowReturn, colReturn);
    board.array[rowReturn][colReturn] = 'x';
                computerBall(colReturn, rowReturn); ////////////////////Is
                 this flipped correct? Yes
    }
}
move += 1;
eraseDisplay();
printBoard(board);
if(evaluate(board) == 10){
```

```
            eraseDisplay();
            displayBigTextLine(5, " The Bot Won!");
            displayBigTextLine(8, "-----GG!-----");
            wait1Msec(2000);
            break;
        }else if(evaluate(board) == -10){
            eraseDisplay();
            displayBigTextLine(5, "  YOU WON!");
            displayBigTextLine(8, "-----GJ!-----");
            wait1Msec(2000);
            break;
        }
    }
    if(noMovesLeft(board)){
        eraseDisplay();
        displayBigTextLine(5, "Ties are boring");
        displayBigTextLine(8, "ZZZZZzzzzz.....");
      wait1Msec(2000);
    }
    return 1;
}

//Team effort
task main(){
        //Computer is X, user is O
    // Game Play Controls the entire play
        //while user wants to keep playing, run gamePlay();
        displayBigTextLine(5, "  Welcome to ");
        displayBigTextLine(8, " Tic Tac Toe ");
        wait1Msec(3000);
        eraseDisplay();

        byte play = 0;
        byte valid = 0;
        do{ //do while loop insures that at least one game is attempted
            time1[T2] = 0 ;
            valid = gamePlay();
            int time = time1[T2];
            eraseDisplay();
            if (valid == 1){
                eraseDisplay();
                wait1Msec(50);
                displayBigTextLine(1, "Game Time:");
                displayBigTextLine(3, "%.2f", time/1000);
                displayBigTextLine(5, "Would you like");
                wait1Msec(50); //Text won't display properly
                displayBigTextLine(7, "to play again?");
                displayBigTextLine(9,"Left (No), Right (Yes)");
                wait1Msec(25);
                TEV3Buttons keepPlaying = get_button_number();
```

```
            if (keepPlaying == buttonRight){
                play = 1;
            }
            if (keepPlaying == buttonLeft){
                play = 0;
            }
            if (play == 1){
                eraseDisplay();
                displayBigTextLine(3, "Please reset");
                displayBigTextLine(5, "the Board");
                moveBoardToPlayer();
                while(SensorValue[S4] == 0){}
                eraseDisplay();
            }
        } else if (valid == 2) {
                eraseDisplay();
                displayBigTextLine(2, "Game Time:");
                displayBigTextLine(4, "%.2f seconds", time/1000);
                displayBigTextLine(6, "No move made");
                wait1Msec(5000);
                play = 0;
        } else {
                eraseDisplay();
                displayBigTextLine(5, "     Don't       ");
                displayBigTextLine(7, "  CHEAT!!      ");
                wait1Msec(5000);
                play = 0;
        }
    } while (play == 1);

    eraseDisplay();
    displayBigTextLine(5, " Thanks for  ");
    displayBigTextLine(7, " Playing ");

    zeroBoth();
    wait1Msec(3000);
}
```