Name: Ahmed Hamed Aly
Student Number: 18308279

Connect Four: Mid-Term Assignment

MAIN

My main subroutine is where everything in my program takes place. At the start of the program it prints out the “Let's play Connect4!!" string and initialises the board before going into a loop of:

Making a move, checking whether a move is a winning move and display the connect four board.

This is an infinite loop and wouldn’t stop unless the user has inputted ‘q’ which signifies that the user wishes to end the game or whenever either player has won the game.

INITIALISE BOARD

My initialise board subroutine is responsible for initialising the board at the start of the game. It loads both the board and the memory address where I’ll be storing the board.

I loaded the value of the board constant and stored it into the memory address where the board is initialised. I repeated the loop for all 42 slots on the board until it has initialised the whole board. I tested that it had initialised the board correctly by looking at the memory panel in keil uVision at the memory location I stored it at to check whether it had been initialised or not.

Pseudo Code:

```
MEM[BOARD] = 0x400000B0;

BOARD = BOARD;

While (counter < 42) {

        Load = BOARD[row][column];

        Store = MEM(BOARD[row][column]);

        Counter++;

}
```

DRAW BOARD

The draw board subroutine draws the connect four board. It prints out the board line by line starting from the rows and continues until it had drawn the whole board. The way I’ve implemented it is that it goes through the whole memory address of the board and prints out whichever piece (if any) onto the board. The following is the order in which my subroutine prints out the board.

```
Let's play Connect4!!

  1 2 3 4 5 6 7   <--- first line
1 0 0 0 0 0 0 0   <--- second line
2 0 0 0 0 0 0 0   <--- third line
3 0 0 0 0 0 0 0   <--- fourth line
4 0 0 0 0 0 0 0   <--- fifth line
5 0 0 0 0 0 0 0   <--- sixth line
6 0 0 0 0 0 0 0   <--- seventh line

RED: choose a column for your next move (1-7, q to restart):
```

Pseudo Code:

```
MEM[BOARD] = 0x400000B0

numberOfRows = 7;

While (counter < 42) {

        Column = counter2 * numberOfRows;

        columnNumber = counter2;

        If (counter > column) {

                [newline];

                Counter2++;

                If (columnNumber == 'whichever column it is') {

                        Print(whichever column it is);

                }

        }

        load = BOARD[row][column];

        if (blank-piece) {

                print(blank);

        }

        If (red-piece) {

                Print(red);

        }

        If (yellow-piece) {

                Print(yellow);

}
```

MAKE MOVE

My make move subroutine does more than just allow the user to make the move. It firstly checks which players turn it is and prints out either, “RED: choose a column for your next move (1-7, q to restart): " or "YELLOW: choose a column for your next move (1-7, q to restart): ". It then waits for the user to input a number using the get subroutine provided and the place whichever the user inputs using the put subroutine.

I firstly compare the user input to ‘q’ which if it’s true it will display an ending message then end the game. To make it more convenient for my program and more easy to understand, I subtracted 0x30 from the user input which will allow me to use the numbers 1 – 7 rather than 0x30 – 0x37. I checked whether the user input is between 1 to 7 and anything other than that would print an invalid input and allows the user to retake their turn. Otherwise it would place the piece in the column. If the column is full it would return with an error message and allow the user to retake their turn.

I tested this subroutine by checking whether the program would still work for every test trial. I started by testing whether the ‘q’ worked. Then I tested that it displayed an error message and would allow the user to retake their turn if its an invalid character or if the column is full.

Pseudo Code:

```
MEM[BOARD];
Column = 6;
if (redPlayersTurn) {
        print(redsTurn);
} else {
        print(yellowsTurn);
}
userInput.get();
print(userInput);
if (userInput.equals(‘q’)) {
        quit = true;
}
if (!(userInput > 0 && userInput < 7)) {
        invalidInput = true;
}

offset = 4;
userInput *= offset;
```

```
MEM[BOARD] += offset;
placeIsAvailable = currentAddress;
if (placeIsAvailable) {
        currentPlayer = place;
        savePosition;
        if (currentPlayer == redPlayer) {
                currentPlayer = yellowPlayer;
        } else {
                currentPlayer = redPlayer;
        }
} else {
        Column--;
        If (column ==  0) {
                invalidInput = true;
        }
}
```

CHECK WINNER

This subroutine branches out to check the board for a winner. There's four different ways that a winning move could be determined (pictured below). It's important that there were no errors with the check winner subroutines as if there's something wrong, it'll destroy the nature of the game. I've tested each subroutines with every outcome by playing the game close to 100 times and checking if there were any errors that may unexpectedly arise. Alongside that I also mapped out how it would check it by mapping it on paper and checking if it makes sense or if it'll run into any errors. I later added a subroutine to check if the board is full and if it is then it ends the game in a draw.

Draw:

```
  1 2 3 4 5 6 7
1 R Y R Y R Y Y
2 Y Y R R R Y R
3 R R Y Y Y R Y
4 Y Y R R R Y R
5 R R R Y Y R Y
6 R Y Y R Y Y R

THE GAME ENDED IN A DRAW!
```

Horizontally:

```
  1 2 3 4 5 6 7
1 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0
3 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0
5 0 0 Y 0 0 0 0
6 Y R R R R Y 0

RED PLAYER HAS WON!
```

Vertically:

```
  1 2 3 4 5 6 7
1 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0
3 0 0 R 0 0 0 0
4 0 0 R 0 0 0 0
5 R 0 R 0 Y 0 0
6 Y 0 R Y Y 0 0

RED PLAYER HAS WON!
```

Diagonal Forward:

```
  1 2 3 4 5 6 7
1 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0
3 0 R 0 0 0 0 0
4 0 Y R Y 0 0 0
5 Y R Y R 0 0 0
6 R Y R Y R 0 0

RED PLAYER HAS WON!
```

Diagonal Backward

```
  1 2 3 4 5 6 7
1 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0
3 0 0 0 Y R 0 0
4 0 0 Y R R 0 0
5 0 0 R R Y 0 0
6 0 R Y R Y Y 0

RED PLAYER HAS WON!
```

HORIZONTAL CHECK

This subroutine checks horizontally if there's 4 in a row. It does that by having a nested loop which checks the row and column and compares it to either 1 (red player) or 2 (yellow player) and repeats until it has checked the whole board.

I've tested this subroutine by getting a horizontal four in a row which works for every possible combination of horizontal win on the board. I checked for errors by trying to possibly cheat my system but nothing has worked so I came to the conclusion that it's a perfectly working subroutine.

Pseudo Code:

```
for (int i = 0; i < row - 3 ; i++ ){
        for (int j = 0; j < column; j++){
                if (board[i][j] == 1 && board[i][j+1] == 1 && board[i][j+2] == 1 && board[i][j+3] == 1)
                        redPlayerWin = true;

                if (board[i][j] == 2 && board[i][j+1] == 2 && board[i][j+2] == 2 && board[i][j+3] == 2)
                        yellowPlayerWin = true
        }
    }
  }
```

VERTICAL CHECK

This subroutine checks vertically if there's 4 in a row. It does that by having a nested loop which checks the row and column and compares it to either 1 (red player) or 2 (yellow player) and repeats until it has checked the whole board.

I've tested this subroutine by getting a vertical four in a row which works for every possible combination of vertical win on the board. I checked for errors by trying to possibly cheat my system but nothing has worked so I came to the conclusion that it's a perfectly working subroutine.

Pseudo Code:

```
for (int i = 0; i < column ; i++ ){
        for (int j = 0; j < row - 3; j++){
                if (board[i][j] == 1 && board[i][j+1] == 1 && board[i][j+2] == 1 && board[i][j+3] == 1)
                        redPlayerWin = true;

                if (board[i][j] == 2 && board[i][j+1] == 2 && board[i][j+2] == 2 && board[i][j+3] == 2)
                        yellowPlayerWin = true
        }
    }
  }
```

DIAGONAL FORWARD CHECK

This subroutine checks vertically if there's 4 in a row. It does that by having a nested loop which checks the row and column and compares it to either 1 (red player) or 2 (yellow player) and repeats until it has checked the whole board.

I've tested this subroutine by getting a diagonal forward four in a row which works for every possible combination of diagonal forward win on the board. I checked for errors by trying to possibly cheat my system but nothing has worked so I came to the conclusion that it's a perfectly working subroutine.

Pseudo Code:

```
for (int i = 3; i < column ; i++ ){
        for (int j = 3; j < row; j++){
                if (board[i][j] == 1 && board[i-1][j-1] == 1 && board[i-2][j-2] == 1 && board[i-3][j-3] == 1)
                        redPlayerWin = true;

                if (board[i][j] == 2 && board[i-1][j-1] == 2 && board[i-2][j-2] == 2 && board[i-3][j-3] == 2)
                        yellowPlayerWin = true
        }
    }
  }
```

DIAGONAL BACKWARD CHECK

This subroutine checks vertically if there's 4 in a row. It does that by having a nested loop which checks the row and column and compares it to either 1 (red player) or 2 (yellow player) and repeats until it has checked the whole board.

I've tested this subroutine by getting a diagonal backward four in a row which works for every possible combination of diagonal backward win on the board. I checked for errors by trying to possibly cheat my system but nothing has worked so I came to the conclusion that it's a perfectly working subroutine.

Pseudo Code:

```
for (int i = 3; i < column ; i++ ){
        for (int j = 0; j < row - 3; j++){
                if (board[i][j] == 1 && board[i-1][j-1] == 1 && board[i-2][j-2] == 1 && board[i-3][j-3] == 1)
                        redPlayerWin = true;

                if (board[i][j] == 2 && board[i-1][j-1] == 2 && board[i-2][j-2] == 2 && board[i-3][j-3] == 2)
                        yellowPlayerWin = true
        }
    }
  }
```

IS BOARD FULL CHECK

This subroutine checks if the board is full or not by looping through the board. If it finds a blank piece then it returns if not and it has reached the end of the board then it ends the game in a draw.

Pseudo Code:

```
MEM[BOARD]
While (counter < 42) {
        load = BOARD[row][column];
        if (load == 0) {
                return;
        }
}
boardIsFull = true;
```