Connect 4 Assignment

Stephen Rowe - ID 14319662

I have included with this document two versions of the Connect 4 program, one for a 2-player game, and one for 1 player vs. the computer. The differences between the two programs will be explained in the relevant subroutines.

Subroutines

initB

This subroutine initializes the board, by placing the character 'O' in every address that was designated to Board. *Note: I allocated an area of memory called MEM the same way as BOARD was allocated space in the template for this assignment that consists of only one byte. This was so that I didn't have to calculate the address in memory just after BOARD every time I wanted to read or write from memory.* initB takes no parameters and returns no values.

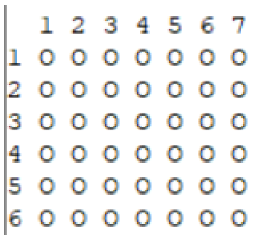
For any subroutines that require to find a particular element on the Board, I used the following instructions to find the given element:

```
;(row*rowsize)
;(row*rowsize)+column
;checked piece = BOARD+index
```

However with initB, this was unnecessary, as the array can simply be treated as a 1D array as each element is only 1 byte. Since the row and column of the given element doesn't actually matter for initB (and since we use MEM to know when to end the subroutine), simply incrementing the address of BOARD was enough to fill the board.

disp

disp reads from BOARD, and formats it into ASCII characters that are stored in memory, allowing us to create the exact shape of the board we want and print it all at once. It starts by reading the string new_board, which is just the numbered columns (1 2 3 4 5 6 7), which will be the same every time the board is printed. It then counts the current row, reads each element of BOARD, and appends space characters (0x20) and carriage return/new line characters (0xD, 0xA) where necessary. It then reads the entire string from memory and prints it to console with the puts subroutine. I print the entire board out like this at once as I will animate the piece falling down the board, and if each line is printed at once, this will ruin the animation effect.



```
  1 2 3 4 5 6 7
1 O O O O O O O
2 O O O O O O O
3 O O O O O O O
4 O O O O O O O
5 O O O O O O O
6 O O O O O O O
```

turn

the turn subroutine has four roles; (1) to print the appropriate message asking the player to take their turn using the strings choose_yellow and choose_red, (2) checking that the user has inputted a valid column for the board or if the user has asked to restart, (3) calls the place subroutine to place the current player's piece in the correct column, (4) determines if the placing of the piece was successful, and returns the final slot in which the piece is currently placed (row, column).

It accepts the current player character (Y or R), and if it is the computers turn, also accepts the column that the computer wants to place their piece.

It returns a single variable, successfulTurn, and is dictated by the place subroutine. This variable has three possible values:

- 0 – the turn was not successful, i.e. the player inputted an invalid command. If this is the case, then turn outputs an invalid message, and again asks the user to input their command.
- 1 – turn was successful, the piece has been placed in the board and the subroutine ends.
- 2 – the player has requested to restart the game, which will be dealt with in main. If 2 is returned, then the program restarts from the top line of main, reinitializing the board and resetting the player order.

place

The place subroutine accepts the current players piece (Y or R), and the column they want to make their move in R0 and R1 respectively. It returns the same player piece (for use in main), the variable successfulTurn, and the final row and final column of the piece when it reaches the lowest vacant spot in that column.

For this subroutine, I initialize the row as 0, then place the piece in this slot*, if the slot is vacant (i.e. contains character 'O'). The board is then printed as is. Then the next row (same column) is tested to see if it is also vacant, and if it is, the piece is removed from the top row, then placed in the next row, and the board is printed again. This is repeated until there are no more rows under then piece that are vacant. This is done to create the animation of the piece falling down. *Note: As the board will be printed too quickly for the animation to be seen if this is run unimpeded, I added a timer to the end of the disp subroutine. A register is assigned the value 0x5FFFF, and a counter is initialized at 0, and incremented by 1 until it is equal to 0x5FFFF. This slows the program down enough for the animation to be visible.*

*If there are no vacant rows in this column, then the piece isn't placed in the slot, and successfulTurn is returned as 0, else it is returned as 1.

There are two important branches in place: placeN and placeF. placeN will be used if the piece can be added to board and printed. placeF is used for the last stages of the subroutine, and checks that the current row incremented is above 0 (else the placement failed), and it places the correct values in the return registers.

chck

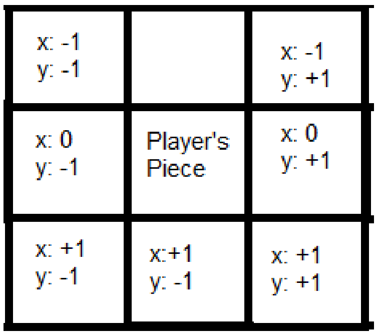
If the game has two human players, then the chck subroutine only checks that the last piece placed is a winning move. If the game is 1 player vs. the computer, then the chck subroutine also checks for eligible moves for the computer.

It accepts the current player piece, a binary value computerTurn to denote that the subroutine should be looking for an eligible move for the computer, the row of the last piece played, and the column of the last piece played (*Note: If playing against the computer, then these last two variables are the last piece played by the computer in their last go, not the piece played by the player. If playing two player, then these last three parameters are ignored.).*

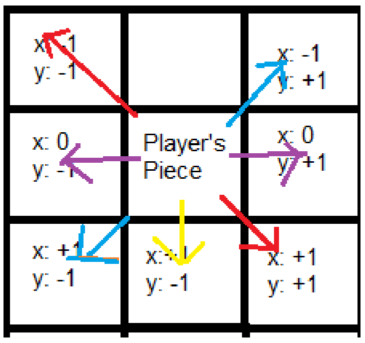
There are some registers that need to be explained to understand this subroutine:

- R9 – the highest streak that the computer can create by placing their next piece.
- R1 – streakSpot is used in conjunction with the subroutine curChk
  - If it is set to 1 or 2, the streak of pieces in a row may not be finished and the next adjacent piece should be checked.
  - If it is set to 0, then the next tested adjacent piece is either out-of-bounds of the board, or is the other players piece
  - If it is set to -1, then the streak has ended, but the next adjacent piece is vacant, meaning the computer may be able to place a piece here.
  - If it is set to 3, the player has won.
- R10 – the total streak of a given line of pieces. If this is set to 3, then the current player has won.

For this subroutine, the row and column of the place piece are saved in R4 and R5, while we manipulate these in R2 and R3. To check that a piece matches the adjacent piece, x and y values are subtracted from these base coordinates, and are tested to see if they match the players piece (also if they are vacant or invalid). The coordinates are derived from the base coordinates like so:



For example, if we want to check that the piece on the top left matches the players piece, we subtract 1 from the row and subtract 1 from the column, find this corresponding element in the array, and then compare it to the player's piece. If it is a match, same x and y values are subtracted from the coordinates of this piece, and again tested, until there are no more matches. The number of matches is counted in R10. Although this method seemed to work at first, I realized that if the last piece necessary to create four-in-a-row was not an end piece, (e.g. Y Y O Y and the player placed their final piece in the vacant slot), then this would not be recognized as a win. To fix this, I changed the program so that the pieces were checked in a line, and R10 would count for both ends of this line:



For example, the top left-hand line would be checked, then the bottom right-hand line would be checked, and their streak would be added together. Then the streak would be reset, then the left-middle and right-middle lines would be checked, and so on. I also initially thought that we could ignore the top-middle slot above the player's piece when checking (since it is impossible to place a piece under another piece), but then I realized that we needed to check this slot for the computer to recognize it as a potential winning move. The curChk (current check) subroutine would be called each time to denote if the adjacent piece matched the current piece.

As a subroutine that would check for the next possible computer move would be very similar to this subroutine, I made some changes to this subroutine to serve the same purpose. The branch chckCm (check computer) first checks if the binary computersTurn is set, and if not, simply branches to curChk. Otherwise, we use the registers R11 and R9 to denote if the current streak is the best possible move for the computer to make. If it is a potential option (i.e. the current streak is higher than max streak), then the column of the possible move is stored in R10, which will be returned when all the checks are complete. This branch also uses the streakSpot variable from above to determine when a streak has ended. However, as only the pieces adjacent to the last placed computer piece are being checked, the computer may miss a potential winning move on the other side of the board. If this subroutine cannot find an eligible place for the computer to place their piece (and create any kind of streak), it will fall back to the flBrd (full board) subroutine in main, which will give it the lowest vacant space on the board and place the piece there instead.

flBrd

The full board subroutine has two purposes, (1) to check that there are no empty slots on the board, (2) if the computer piece can't find a slot in chck to make its move, it gives column that contains the least pieces.

This subroutine has no parameters, and returns an available column in R1 (if possible) and a binary value isBoardFull in R2.

This subroutine initializes the row as 5 (lowest row) and the column as 0. In then increments the column, checking the equivalent character in the BOARD array to see if it is vacant ('O'), then increments the row and resets the column, until we reach the top right element on the board. If there are no more vacant slots, then it returns isBoardFull as true. Else, it returns isBoardFull as false and the first vacant column it finds as an available space.

Main Program

To start, the string "Let's Play Connect 4!" is printed. The board is then initialized and printed. Yellow goes first, so 'Y' is entered as a parameter for turn (and isComputersTurn is set to false, so it is not checking for potential moves). If 2 is returned from turn for successfulTurn, then the program restarts by branching to the first line in main. After the player has entered a valid move, chck accepts the row and column of their move and checks if the player has a row of 4.

Next, isComputersTurn is set to true and chck is run, finding a move for the computer. The last move by the computer will be stored in R4 and R5, and will be moved into the appropriate registers for the chck subroutine. If a space isn't found by chck, flBrd is run instead, and finally the piece is placed in the board. Again we have to run chck (this time with isComputerTurn set to false), in order to check if the computer has won.

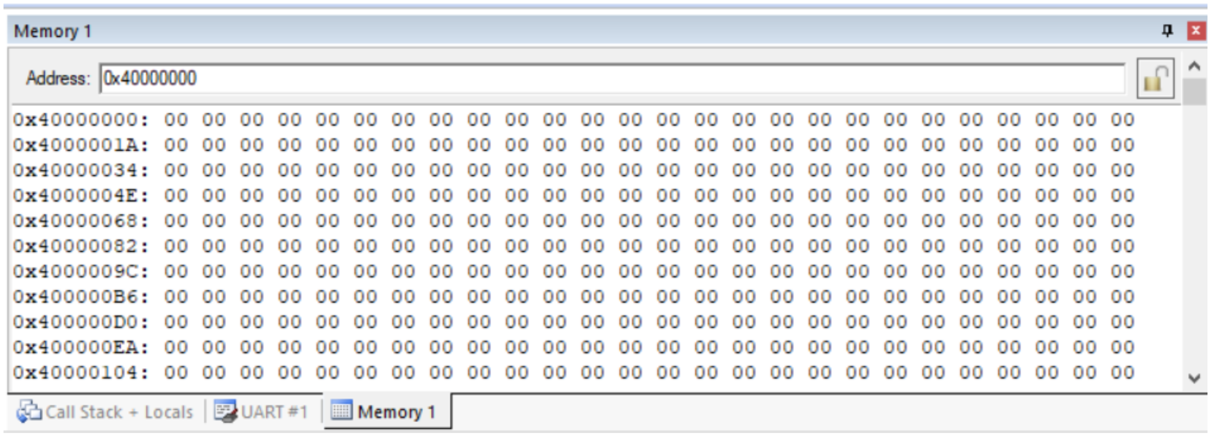
If it is a two player game, then the isComputerTurn binary variable is ignored, and the second payer is asked for their turn just like the first player.

 If successfulTurn is set to 3 after either players turn, then the player has won, and it branches to "winner", which prints an appropriate message to the console. If the board is full without either player winning, then "It's a draw!" is printed.

Tests for Subroutines

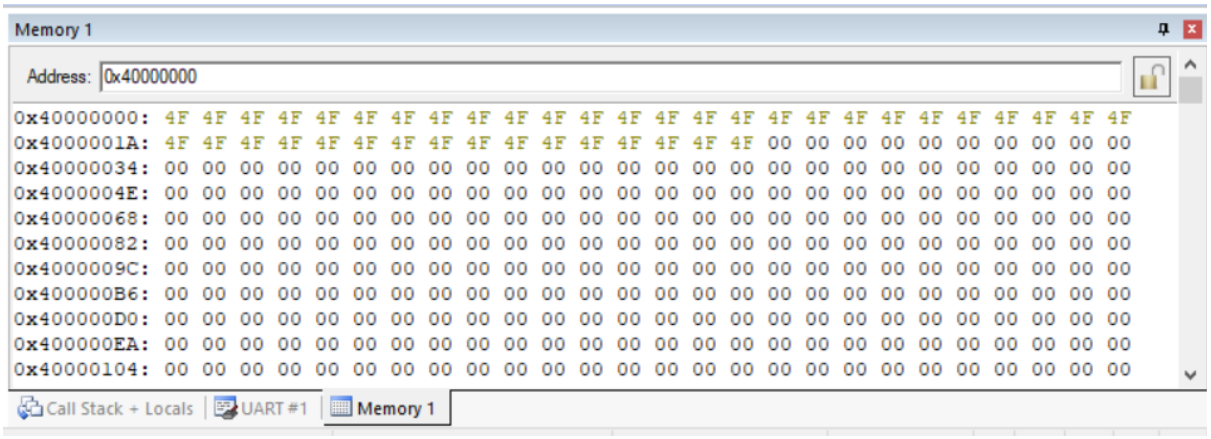
initB

To test initB, all I had to do was check the address 0x40000000(the address designated to BOARD) in the memory window in Keil *before* running initB, to see that every element was 0x00:



```
0x40000000
0x40000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x4000001A: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40000034: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x4000004E: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40000068: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40000082: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x4000009C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000B6: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000EA: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40000104: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

And then I checked memory *after* running initB to see that every element designated to BOARD was 0x4F(the ASCII code for the character 'O'):



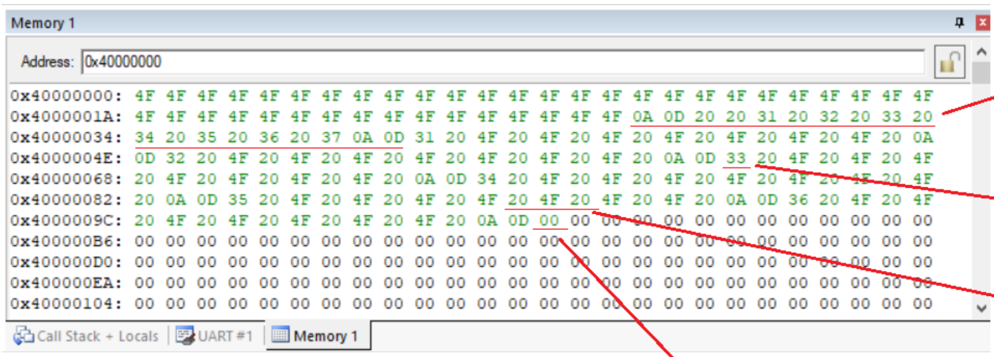
```
0x40000000
0x40000000: 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F
0x4000001A: 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 00 00 00 00 00 00 00 00 00 00
0x40000034: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x4000004E: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40000068: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40000082: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x4000009C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000B6: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000EA: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40000104: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

As initB does not take an parameters, we cannot influence it with inputs.

disp

Disp also does not accept any parameters, but we can see how it can be influenced by playing a turn in the game. The memory that will be used by disp will always be the same size and start immediately after BOARD, but the elements within this MEM will be manipulated during a game.

Before disp is run for the first time, the Memory window is identical to the previous image in the initB subroutine test. After running disp, we can see that a string, starting at MEM, has been created, and by printing this null-terminated string we will get a blank board:



```
0x40000000: 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F
0x4000001A: 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 0A 0D 20 20 31 20 32 20 33 20
0x40000034: 34 20 35 20 36 20 37 0A 0D 31 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 0A
0x4000004E: 0D 32 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 0A 0D 33 20 4F 20 4F 20 4F
0x40000068: 20 4F 20 4F 20 4F 20 4F 20 0A 0D 34 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F
0x40000082: 20 0A 0D 35 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 0A 0D 36 20 4F 20 4F
0x4000009C: 20 4F 20 4F 20 4F 20 4F 20 4F 20 0A 0D 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000B6: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000EA: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40000104: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

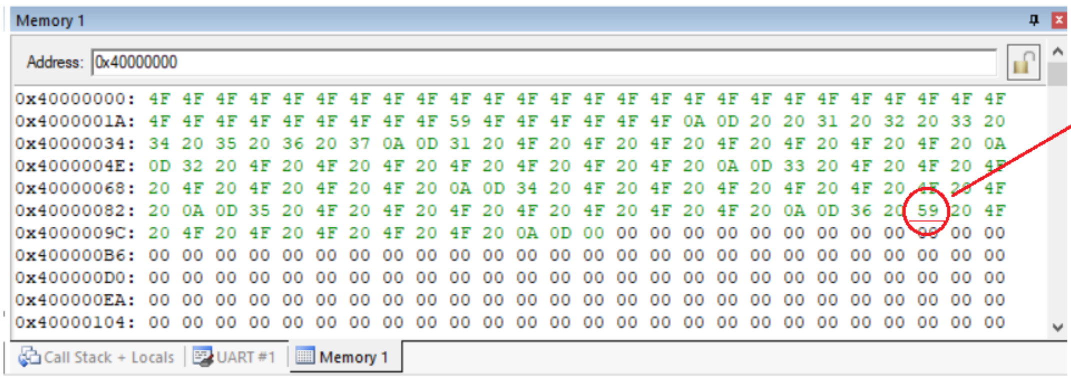
We can see this is the string new_board

We can see the Row numbers (this is 0x33, '3', and is printed for each new row of the board.

Each 'O' character in the board is padded with 2 space characters (0x20) for it to be displayed correctly.

null termination of string

I will now take a turn as the Yellow player, and place a piece in the first column:



```
0x40000000: 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F 4F
0x4000001A: 4F 4F 4F 4F 4F 4F 4F 4F 4F 59 4F 4F 4F 4F 4F 4F 0A 0D 20 20 31 20 32 20 33 20
0x40000034: 34 20 35 20 36 20 37 0A 0D 31 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 0A
0x4000004E: 0D 32 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 0A 0D 33 20 4F 20 4F 20 4F
0x40000068: 20 4F 20 4F 20 4F 20 4F 20 0A 0D 34 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F
0x40000082: 20 0A 0D 35 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 4F 20 0A 0D 36 20 59 20 4F
0x4000009C: 20 4F 20 4F 20 4F 20 4F 20 4F 20 0A 0D 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000B6: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x400000EA: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x40000104: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The element on the BOARD, and the element in MEM equivalent to the bottom row and first column on the board have been changed to 0x59 which is the ASCII code for 'Y'. This null terminated string in MEM will now be printed.

And this is the string printed:

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

turn

The current player is switched in main after each move, and is denoted in R0 with the char ‘Y’ or the char ‘R’. If the char is ‘Y’, then turn will print out the following message:

```
YELLOW: Choose a column for your next move (1-7, 'q' to restart):
```

If the char is ‘R’, *and this is a two player game*, then the following message is printed:

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

Else, if the second player is a computer, this RED message is never printed.

The user has 3 potential input types:

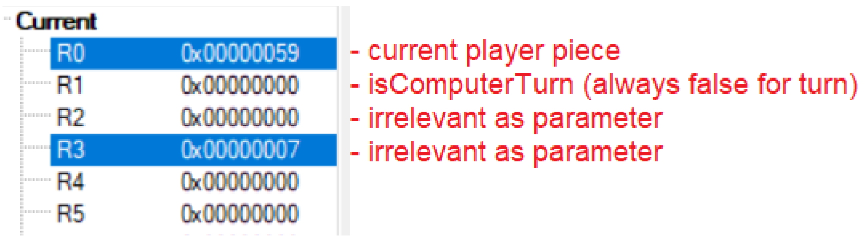
- A number between 1 and 7 (for column)

- The character ‘q’ for restart
- An invalid command (any character that is not included in the previous two points, or a column that is already full). In this case, the following is printed:

```
Invalid move, try again.
```

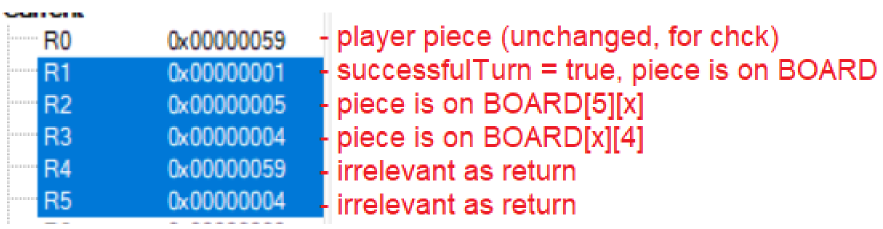
The second parameter, isComputersTurn, is not manipulated by turn, and since turn is only called when it’s a human player’s turn (when it is the computer’s turn, the turn subroutine is unnecessary, so place is just called instead), then this Boolean will always be 0 when it is passed into Turn, ensuring that place deals with the instructions appropriately.

For the player’s first turn, the input will be as follows:



```
Current
R0    0x00000059   - current player piece
R1    0x00000000   - isComputerTurn (always false for turn)
R2    0x00000000   - irrelevant as parameter
R3    0x00000007   - irrelevant as parameter
R4    0x00000000
R5    0x00000000
```

And when turn is finished, the output registers will be as follows:

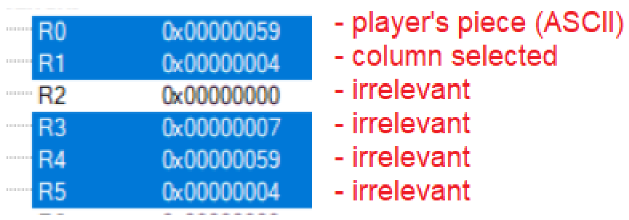


```
R0    0x00000059   - player piece (unchanged, for chck)
R1    0x00000001   - successfulTurn = true, piece is on BOARD
R2    0x00000005   - piece is on BOARD[5][x]
R3    0x00000004   - piece is on BOARD[x][4]
R4    0x00000059   - irrelevant as return
R5    0x00000004   - irrelevant as return
```

Above, we can see that the player selected column number 5 to place the piece on their board, and this is equivalent to the 4th column in the array (just subtract 0x31 for every time a number is entered into console to get it’s hex equivalent).

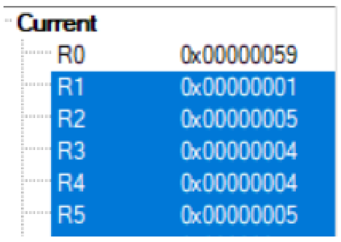
place

When place is called, the player piece is still in R0, and the column that the player or computer selected will be in R1 (this will be moved to R4 for protection).



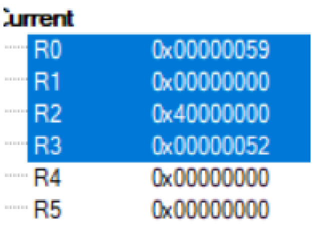
```
R0    0x00000059   - player's piece (ASCII)
R1    0x00000004   - column selected
R2    0x00000000   - irrelevant
R3    0x00000007   - irrelevant
R4    0x00000059   - irrelevant
R5    0x00000004   - irrelevant
```

After place, there are two possibilities: the place has been put on BOARD (column has vacant slots) and 1 is returned in R1 for successfulTurn, or the place is not on BOARD and 0 is returned for successfulTurn. If it is a case of the latter, then the row and column that will be returned in R2 and R3 don’t matter as they will be ignored and later overwritten. Otherwise, R2 and R3 will be important for checking the piece. Below is an example of a successful turn, with the piece placed in BOARD [5][4]:



```
Current
  R0    0x00000059    - current player
  R1    0x00000001    - turnSucessful = true
  R2    0x00000005    - row of the placed piece
  R3    0x00000004    - column of the placed piece
  R4    0x00000004    - irrelevant
  R5    0x00000005    - irrelevant
```

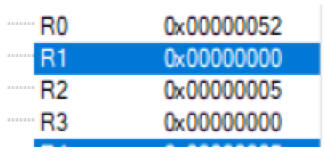
For this next image, I completely filled up the first column of the board with pieces, then placed another piece:



```
Current
  R0    0x00000059    - player piece
  R1    0x00000000    - sucessfulTurn is false, therefore...
  R2    0x40000000    - will not be used
  R3    0x00000052    - will not be used
  R4    0x00000000    - etc.
  R5    0x00000000
```

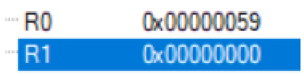
chck

For chck, the parameters will be as follows:



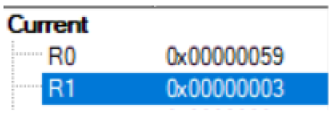
```
  R0    0x00000052    - player's piece
  R1    0x00000000    - isComputersTurn (false)
  R2    0x00000005    - if isComputersTurn is true, then R2
  R3    0x00000000    will have the row of the last R piece
                      played, and R3 will have the column
                      of the last R piece played. Else,
                      these registers are irrelevant.
```

After check (assuming this is not the computer's turn) the return values will look like this:



```
  R0    0x00000059    - current piece
  R1    0x00000000    - highest streak is 0
```

And here is the same, if the player has four-in-a-row:

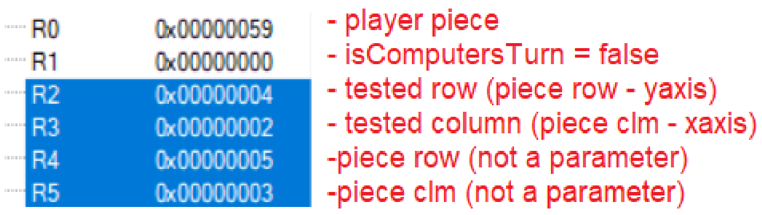


```
Current
  R0    0x00000059
  R1    0x00000003
```

If it is the computer’s turn, then R2 will be the suggested column they place their next piece.

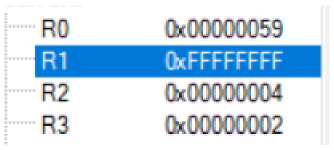
curChk

Here is an example of the relevant registers when entering curChk:



```
R0    0x00000059   - player piece
R1    0x00000000   - isComputersTurn = false
R2    0x00000004   - tested row (piece row - yaxis)
R3    0x00000002   - tested column (piece clm - xaxis)
R4    0x00000005   -piece row (not a parameter)
R5    0x00000003   -piece clm (not a parameter)
```

From above, we can see that the slot to the top left of the piece is being tested. As this slot is currently vacant and is not out of bounds, then -1 will be returned for R1 (this signifies for the case that isComputerTurn is true, we know that this is a vacant spot and potential move).



```
R0    0x00000059
R1    0xFFFFFFFF
R2    0x00000004
R3    0x00000002
```

If it is an invalid move then 0 will be returned in R1, and if it is a continued streak than 1 will be returned in R1, for use in chck.

flBrd, Animation, and Computer Player

The best way I have of demonstrating these three aspects of the program is by showing the console in the event that the board is full. I have created GIFs showing a playthrough of both versions of the game, available here: https://imgur.com/a/QRlKlax (you may need to refresh the page once or twice for the GIFs to appear properly).

In the first GIF in that link, we start out with an almost full board. When the board is completely filled by Player 1 and Player 2, then “It’s a draw!” is printed and the entire program resets (reinitializes board, order to player’s turn, etc.). Then you will see the case in which the Yellow player creates Four-In-A-Row with the final piece being placed in the middle (Y O Y Y), showing that this is recognized as a win regardless of the last piece being on the end of the 4 pieces.

In the second GIF, we start out with a partly filled board, and the user inputs “q” to restart the game. The board is then reinitialized. This time, the computer will try to get 4 in a row, and it will eventually win.

Please note: I wanted to demonstrate the animation of the pieces falling down, but it came out choppy in this GIF, likely due to the frame-rate of the GIF. If you run this program on your own computer, the falling down animation will be much smoother.