ECE 109 **Program 3:** `tac.asm`
Spring 2021

This programming assignment must be completed individually. Do not share your code with or receive code from any other student. The only people who may see your code are the instructor and the ECE 109 TAs.

Evidence of copying or other unauthorized collaboration will be investigated as a potential academic integrity violation. **The minimum penalty for cheating on a programming assignment is a grade of -100 on the assignment.** If you are tempted to copy because you're running late, or don't know what you're doing, you will be better off missing the assignment and taking a zero. Providing your code to someone is cheating, just as much as copying someone else's work.

DO NOT copy code from the Internet, or use programs found online or in textbooks as a "starting point" for your code. Your job is to design and write this program from scratch, on your own. Evidence of using external code from any source will be investigated as a potential academic integrity violation.

This program will allow two users to play tic-tac-toe. (The computer will not be a player.) The playing board will be displayed on the graphics display, and the text console will be used for user prompts.

The learning objective for this assignment is:

- Use subroutines to modularize the program.

Program Specification

The program must start at address x3000.

The program will first clear the graphics display (making all pixels black). It will then draw two white vertical lines and two white horizontal lines, to show the playing board. The starting positions for the vertical lines are (30,0) and (60,0). The starting positions for the horizontal lines are (0,30) and (0,60).

This creates nine "blocks" into which X's and O's can be drawn. The blocks are numbered from zero to eight: 0,1,2 = first row (left to right); 3,4,5 = second row (left to right); 6,7,8 = third row (left to right). (See below for details on how and where to draw the X/O.)

The program will then prompt the users for their moves, starting with player X. The player will enter a block number (0-8), followed by Return/Enter. If the block is not occupied, then the appropriate marker (X or O) will be displayed in the chosen block. If the block is occupied, or if the user enters an illegal move, then the program prompts the same user for a move. (No error message is printed.)

When the user wants to quit the game, he/she enters "q" (followed by Return). When that happens, the program halts.

Details

Because the learning objective of this program is to use subroutines, I am requiring you to write at least **five (5)** subroutines, some are described in the next section. You are encouraged to use additional subroutines for the other parts of the program. For each subroutine, be sure to include a comment that describes the function and the interface.

IMPORTANT: Subroutines will be graded on their own. Make sure that the subroutine is self-contained. Don't use data that's stored outside the subroutine (unless a pointer to that data is passed in). Make sure that your subroutine does not rely on any of your code that is run before this subroutine is called.

Loading the Block Data

You are provided with files named **blockx.asm** and **blocko.asm**. This provides the data for the block used to draw the X and the O. Assemble this file. (You only need to do this once.)

Before running your program, load the **blockx.obj** and **blocko.obj** files. This puts the block data for X at memory location xA000, and the block data for O at memory location xA200.

Required Subroutines

`GETMOV`

This subroutine is used to get the next move from a player. The user input will always be terminated with the Return/Enter key (ASCII #10). Legal moves include the numbers between 0 and 8, or the letter 'q'. If the user enters a number between 0 and 8, then that number (not the character!) must be returned in R0. If the user types 'q', then the value 9 must be returned in R0. Otherwise, it's an illegal command, and the subroutine must return -1 in R0.

For this subroutine, you must echo every character typed by the user. (This means that each character typed by the user -- even Return! -- is also displayed to the console.)

Note that the command is not complete until the user types Return. So if the user types "11", it is not legal. Don't just get the first '1' and assume that it will be a legal command. It's only legal if the user types '1' and then Return.

Hint: You don't need to keep track of all of the characters typed by a user. If you see a legal character (e.g., '1') and then see anything other than Return, you can just throw away all characters until the Return is seen.

`DRAWH`

Draw a horizontal line. The line must be white, and it must be 90 pixels, starting at the leftmost boundary of the graphics display. The value in R0 specifies the y-coordinate of the starting position. (In other words, it's the row number of the graphics display.)

`DRAWV`

Draw a vertical line. The line must be white, and it must be 90 pixels, starting at the top boundary of the graphics display. The value in R0 specifies the x-coordinate of the starting position. (In other words, it's the column number of the graphics display.)

`DRAWB`

This subroutine draws a 20x20 block of pixels, using a pattern stored elsewhere in memory. This is what we will use to draw the X's and O's. (You will be given the data for the X and O blocks.)

The block data is stored as a 400-element array. The first 20 elements correspond to the top row. The next 20 elements are the second row, etc. Each element is either zero or non-zero; the specific non-zero value is not important, and your code should not look for a particular value.

The subroutine is given three inputs: (1) R0 is the starting address for drawing the block, corresponding to pixel in the top left corner of the desired location. (2) R1 is the starting address for the block data, described in the previous paragraph. (3) R2 is the color. When drawing the block, a zero element in the data means to use black, while a non-zero element specifies that the pixel should be the color in R2.

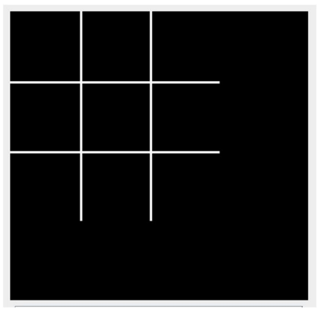
You may assume that the starting location is chosen such that the 20x20 block will be within the graphics display boundaries. You do not need to check for overflow or wraparound.

Main Program

The main program (together with any other subroutines you write) is responsible for the rest of the behavior. It prints the prompts for the user interface, and controls the flow of the program between player X and player O. It decides which block to print where, and which color to use.

First, the program must clear the graphics display. You did this in Program 2, so I recommend that you create a subroutine that uses the same code to accomplish this.

Next, the white lines are drawn. You can do this in any order. You will draw two vertical lines (30,0) and (60,0) and two horizontal lines (0,30) and (0,60). When this is done, the graphics display should look like this:



Now you are ready to interact with the user in the text console. You will alternate between X and O, with X going first.

First, print the following prompt string: "X move: ". (Note the space after the colon.) Then call GETMOV to get the user input.

When GETMOV returns, the move is in R0. If it's 9, then quit. If it's -1, then it's an illegal move, and you should simply print the same prompt and get another move. (Do this forever, as long as the user enters illegal moves.)

If the move is between 0 and 8, then you need to draw the X block into the correct square. Do this by calling DRAWB. As described earlier, by loading the block.obj file, you have placed that data into memory, starting at address xA000. The color for the X block is yellow (x7FED). The starting addresses for the various blocks are in the table below.

Block	*Starting Location*	*Starting Address*	*Description*
0	(5, 5)	`xC285`	Top left
1	(35, 5)	`xC2A3`	Top middle
2	(65, 5)	`xC2C1`	Top right
3	(5, 35)	`xD185`	Middle left
4	(35, 5)	`xD1A3`	Middle
5	(65, 35)	`xD1C1`	Middle right
6	(5, 65)	`xE085`	Bottom left
7	(35, 65)	`xE0A3`	Bottom middle
8	(65, 65)	`xE0C1`	Bottom right

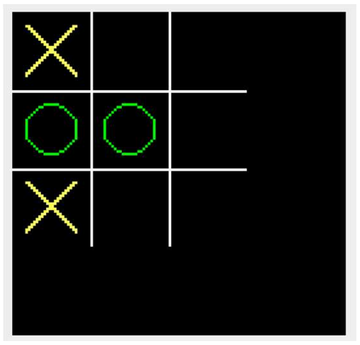
Hint: You can store the starting addresses in an array, and use the move number (0-8) as an index to load the address to draw the block.

Once the X move is done, do the same thing for player O.

1. Print the prompt: "O move: ".
2. Get the move.
3. Draw the O. The O block data starts at xA200. The color is green (x03E0).

However, before drawing the block, you need to make sure that the square is not already occupied. For example, if player X chooses block 4 (middle), and player O also selects block 4, you should treat that as an illegal move, and prompt for a new move. (Do not print an error message.)

Here's what the graphics display might look like after a couple of moves:



NOTE: You do not need to detect when a player wins; just keep playing until a player quits.

Hint: How do you know whether a square is occupied? One way is to keep an array of integers in memory, one per block. Initialize the block to zero. When a square is taken, write a 1 to that location. When getting a new move, check that array element to find out whether the block has been taken.

This can also be done using one word (instead of an array). A single bit can store whether a block is taken or not.

Important: DO NOT alter the block data to make the graphics look better. Although I'm sure some of you could make a better looking game, any change in what is actually drawn will break our grading scripts. Likewise, do not alter the user interface. This is very simple, and it is intentionally designed that way, so that you don't spend most of your time implementing the interface. If you want a more complete game, that's great -- just don't submit it.

Hints and Suggestions

- As always, **design before you code**! Draw a flowchart to organize and document your thinking before you start writing the program. Because we are breaking the program into modules (subroutines), you can separate the design of the subroutine from the design of the entire program. For example, one block can just say "get move" and you can figure out the details of that subroutine separately.
- **Work incrementally!** For example, implement one subroutine at a time. Write some special code to call the subroutine, and make sure it works before moving on to the next feature. This way, you always have working code.
- It's not a bad idea to submit each working version of your program to Wolfware. Then, if your machine crashes (it happens!), you haven't lost everything. Each time you submit, it overwrites the previous submission, so you can submit as many times as you like. But don't expect that we can recover some previous version of your code if you accidentally clobber it. (You should have some sort of backup system for your schoolwork, right?)

- *Test your program with a wide variety of inputs.* Make sure that you have tested the "corner cases," such as reaching the border of the display.
- Use the PennSim simulator and assembler. There are other simulators and assemblers out there, but there are some differences. Your program will be graded using PennSim, and no other tools will be used or considered.

Administrative Info

Any corrections or clarifications to this program spec will be posted on the **Discussion Forum**. It is important that you read these postings, so that your program will match the updated specification. (I recommend strongly that you subscribe to the forum, so that you will not miss any updates or corrections.)

What to turn in:

- Assembly Language Source file – it must be named **tac.asm**. Submit via **Moodle** to the Program 3 assignment.

  The program will be graded by one of the TAs, and your grade will be posted in Moodle. You will also get back some information about what you got wrong, and how points were deducted (if any).
- DO NOT submit .obj or .sym files. Do not submit a .txt file, a .doc file, or anything like that. It must be a simple text file with the proper .asm extension. If we are unable to open or interpret your file, you will get a zero for the assignment (even if it has the right name!).

Grading criteria:

There is NO flowchart due on Program 3. However you are expected to use one to create your program.

10 points: Correct type of file, submitted with the **proper name**. (No partial credit!! These are essentially FREE POINTS! Don't screw it up.)

10 points: **Program is complete** and **assembles with no warnings and no errors** using the PennSim assembler. To be "complete," there must be code that makes a reasonable attempt to meet the program specs. Programs that do not assemble will not be graded any further. (For warnings, points will be deducted, but the program will be tested for correctness.)

10 points: **Proper coding style, comments, and header**. Use indentation to easily distinguish labels from opcodes. Leave whitespace between sections of code. Include *useful* comments and *meaningful* labels to make your code more readable. Your file **must** include a header (comments) that includes your name and a description of the program. Don't cut-and-paste the description from the program spec – that's plagiarism. Describe the program in your own words. This category is somewhat subjective, but the goal is that your program should be easy to read and to understand. Minimum of 5 (five) subroutines.

40 points: Required subroutines -- tested independently:

(5 points each) DRAWH, DRAWV
(10 points) DRAWB
(10 points) GETMOV

30 points: Program features:

(5 points) Draw the playing area (two vertical and two horizontal lines).
(5 points) Alternate between X and O players.
(5 points) Draw X and O in the right spots, based on entered moves.
(5 points) Repeat move for the same player when illegal move is detected.
(5 points) Quit when “q” move is entered.
(5 points) Detect when square is already occupied, and repeat move for the same player.