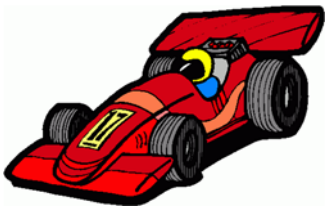
ECE 109-001
Spring 2017

Program 3: `monaco.asm`

Due April 27, 2017 @ 11:45pm



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.

For this assignment, you will use your previous program that draws and moves blocks on PennSim's graphic display, and add more function to it. The program user will be able to direct their block through a race course, checking for crashes, and will be able to change the color of the block which is being moved.

The learning objectives for this assignment are:

- Use load and store instructions to manipulate the content of memory.
- Use I/O routines to allow a user to interact with the program.
- Subroutines

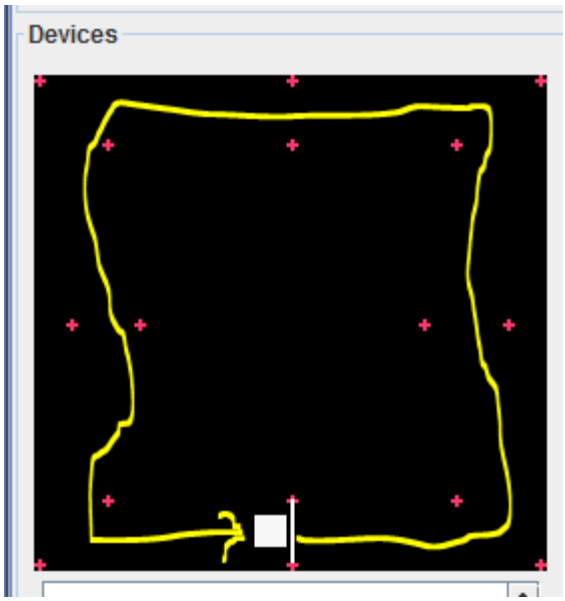
Program Specification

The program must start at address x3000.

The program will manipulate the location of a box on the screen, which we will call the Box. The Box has a color and a location and is 8x8 pixels in size. The screen pixels around the Box's current location will take on the Box's color. When the Box moves, the pixels at the previous location return to the background color black. We must erase the old box, move to the new coordinates, and draw the new box.

The PennSim graphics display (the "screen") is 128 by 124 pixels. We use an (x, y) coordinate system to describe a location on the screen. Location (0, 0) is the top left corner. The x coordinate increases as we move to the right, and the y coordinate increases as we move down. In other words, (1, 0) is one pixel to the right of (0, 0), and location (0, 1) is one pixel below (0, 0). Location (127, 123) is the bottom right corner of the screen.

The program begins by drawing a race course as shown:



The coordinates for the "**pylons**" and the start/finish line are given here in a table:

X	Y	Memory Decimal	Memory Hex
1	1	49281	C081
9	62	57097	DF09
1	122	64769	FD01
18	17	51346	C892
26	62	57114	DF1A
18	106	62738	F512
64	1	49344	C0C0
64	17	51392	C8C0
64	106	62784	F540
64	122	64832	FD40
105	17	51433	C8E9
97	62	57185	DF61
105	106	62825	F569
126	1	49406	C0FE
118	62	57206	DF76
126	122	64894	FD7E

The location of each pylon is listed in the table. The pylon consists of 5 orange pixels centered on these coordinates, once at the center and one in each direction North, South, East, and West. The car must drive counter-clockwise through the course.

The program must check if at any time, any pixel of the box is on **any pixel** of a pylon. The program must write the following string to the console and then halt. "CRASH!! GAME OVER !!!" plus 3 line feeds.

When the program begins, the Box location must be set to just left of the center of the start/finish line as shown. It is your choice how to orient the box around this location. The "car" moves counter-clockwise around the course, between the pylons pairs.

You are required to use **at least five (5) subroutines** in your program. You may use more, as many as you like. The choice of subroutines is up to you, but there are several obvious candidates: paint/erase a square, draw the course, check for crash, draw the start/finish line, etc.

Subroutines are used to modularize your code. Separate it into manageable pieces. As you write a subroutine, you should also write some code to test that routine on its own. Once you know that a subroutine works, then you can move on to implement and test some other parts of the program. (When there's a bug, you can be fairly confident that the problem is with the new code, since you've already tested and debugged the earlier subroutines.)

You must define the interface for each subroutine. What does the caller pass in? What does the caller get back? This interface must be documented in your code via comments. Each subroutine must have a **comment header** that describes what the subroutine does and how to use it. Lack of documentation will cause you to lose points.

The user interacts with the program using one-character commands. The commands are typed on the keyboard, but are not printed to the console display. Nothing will be printed to the console during the execution of this program. The program will wait for a keystroke, perform the corresponding command, and repeat. If the keystroke does not correspond to a legal command, it will have no effect.

There are four commands for changing the location of the Box. We use the WASD scheme of navigation, used by various computer games:

Command Character	Action
w	*Move up one block.* Location changes from (x, y) to (x, y-1). If the Box is at the top border of the screen, the command has no effect.
a	*Move left one block.* Location changes from (x, y) to (x-1, y). If the Box is at the left border of the screen, the command has no effect.
s	*Move down one block.* Location changes from (x, y) to (x, y+1). If the Box is at the bottom border of the screen, the command has no effect.
d	*Move right one block.* Location changes from (x, y) to (x+1, y). If the Box is at the right border of the screen, the command has no effect.

There are five commands for changing the Box color:

Command Character	Action
r	Change color to *Red*. Ferrari
g	Change color to *Green*. Mercedes
b	Change color to *Blue*. Red Bull
y	Change color to *Yellow*. Renault
space	Change color to *White*. Williams Racing

Note: When you change the color of the Box, the color of the current location must change, before the next command is processed.

There is one additional command:

Command Character	Action
q	*Quit.* The simulated machine must stop running.

NOTE: You **MUST** use/load **p3os.obj** for this code to run properly!

Details

The PennSim graphics display is bit-mapped, meaning that each pixel has a corresponding memory location. The content of that memory location controls the color of the pixel.

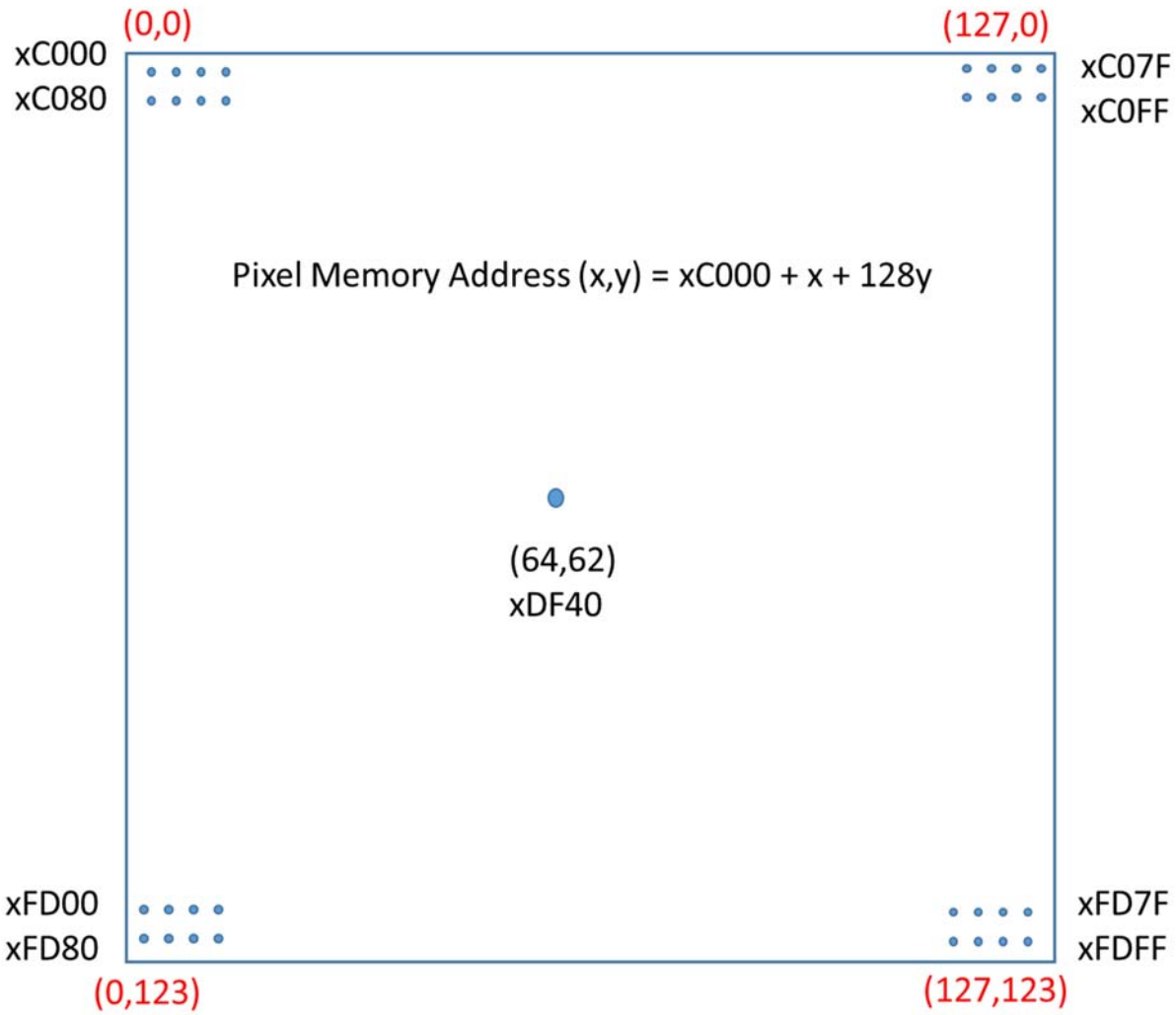
Pixel Addresses

Addresses xC000 through xFDFF are assigned to the graphics display. The low address corresponds to the top left corner (0, 0). Moving one pixel to the right adds one to the address, and it "wraps around" to the next row when it gets to the right edge. Since the display is 128 pixels wide, this means that moving down one pixel is equivalent to adding 128 to the address.

The address of point (x, y) can be calculated as: xC000 + x + 128y.

For this assignment, you will not need to calculate arbitrary pixel addresses, except to figure out where the initial location (64, 62) is. You will be moving left (-8), right (+8), up ((8*-128)) or down ((8*+128)) from the current address.

You will, however, need to recognize when the Box is at an edge of the display, so that you don't go beyond the edge.



Pixel Color

As mentioned above, the value of the pixel address determines the color of the pixel. The 16 bits contain 5 bits for each RGB component of color: bits [14:10] for red, [9:5] for green, and [4:0] for blue. Bit 15 is ignored. The higher value of a component, the more of that color is present. The table below gives the color values (in hex) needed for this program.

Color	Value
Red	x7C00
Green	x03E0
Blue	x001F
Yellow	x7FED
White	x7FFF
Black	x0000

Miscellaneous

For more explanation about the PennSim display, see the PennSim Reference Manual.

The ASCII code for the Return key is x0A (#10). This is listed as linefeed (LF) in the ASCII table.

Hints and Suggestions

- As always, **design before you code**! Draw a flowchart to organize and document your thinking before you start writing the program.
- **Work incrementally!** For example, implement one command at a time. Make sure the program works before moving on to the next command. 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 **monaco.asm**. Submit via **Moodle** to the Program 2 assignment.
- 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:

5 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, Section Number, and

Submission Date. 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.

75 points: The program **handles all function correctly**:

(15 points) Draws the race course on the screen and the initial box.

(25 points) WASD movements, check edges.
(10 points) Change colors
(20 points) Properly identifies a crash (box touches pylon center), prints message.
(5 points) Quit.

NOTE: You **MUST** use/load **p3os.obj** for this code to run properly!

Bonus Points: Available ONLY if all 75 Function Points are completed!

[A] [10 pts] Attach and call a subroutine to be provided by the instructor which draws a car icon instead of a box. Request from instructor when ready.

[B] [10 pts] Add direction to your car icon. If moving right, face right. If moving up, face up, etc.

[C] [20 pts] Use the PennSim Timer function to record the time it takes the driver to complete one lap of the course. You must start the timer when they cross the start line, stop the timer when they cross a second time, and report the time in proper mm:ss.xx format. The accuracy should be ½ second (500 msecs). To do this you will have to check the timer status register in continually and advance your count whenever it "ticks"