ECE 2534 Lab 3 –Fall 2015
"Snake Game"

YOU MUST COMPLETE THIS ASSIGNMENT INDIVIDUALLY.

Read the entire specification before you begin working the lab.

Honor Code Requirements

For laboratory assignments, students are not allowed to share or discuss any element or detail of a design or solution with any other student. Each student must treat all such details as proprietary. In particular, you may not share or discuss any specific algorithms or any source code with any other student; obtaining source code from *any* source other than your instructor, your course notes, or the PIC32 peripheral libraries is a violation of the Virginia Tech Honor Code, and will be prosecuted as such.

Objectives

- Use the microcontroller to perform analog-to-digital conversion, and use this digital output to drive the behavior of a software task.
- Use the interrupt controller to handle and prioritize all events from the PIC32 peripherals (e.g., the timers and analog-to-digital conversion).
- Use the functionality of the OLED display to use custom characters and/or perform pixel-level graphics.
- Use the state-machine programming model to implement software tasks and maintain system state.

Description

In Lab 3 you will be making a version of classic Snake video game. See the Wikipedia entry for the history of the game: https://en.wikipedia.org/wiki/Snake_%28video_game%29, and/or you can play an on-line version of the game at http://playsnake.org/. Our simple version of the game is to have snake that can "move" around the OLED screen in one of four directions controlled by a joystick. The playing portion of the OLED screen is broken up into 8 by 18 (or more) "playing field positions." Each playing field position can contain one of the following: (1) an "empty" cell; (2) an "apple"; (3) the snake's "head"; or (4) a segment of the snake's "body." The snake itself consists of a "head" that is always followed by some number of "body" segments. Once moving in a particular direction, the snake continues in that direction followed by its body segments until the user uses the joystick to change the snake's head's direction. There is always an "apple" available on the screen at some random position not occupied by the snake. By maneuvering the snake to run into the apple, the snake "eats" the apple, and becomes one body segment longer. The user can select different difficult levels (the speed at which the snake moves) and set a goal for a number of points to get to win the game. The user loses if the snake runs into itself or runs into the edge of the playing field.

To control the snake, you will use the joystick. The two-axis joystick uses two potentiometers (one per axis) that map the joystick position to a pair of analog outputs. You will interface the joystick to the PIC32 analog-to-digital converter (ADC) and write code to interpret this analog information as actions that determine the snake's direction of movement. You will use a timer as a trigger for determining when the snake should move, which you will display by updating the OLED display.

The lab specification will outline system behavior without specifying **all** aspects of the implementation. The specification will mandate certain features, but you will be responsible for choosing other parameters and the "look and feel" of the game based on your personal aesthetics. Successfully completing this lab will require good knowledge of the timers and the analog-to-digital converter. You will also have the opportunity to integrate a number of lower-level software routines into a large (well, large for us!) software project.

Note on Creativity

All the pictures and screen shots in this document represent a particular implementation. You are not limited to this implementation—be creative! The only limitation is to accomplish the tasks required by the specification. Your creativity in performing these tasks is welcome; especially creative implementations will receive extra credit.

LAB SPECIFICATION

0. The two-axis joystick should be connected to your chipKIT PIC32 board via the six-wire “Medusa” cable. The cable should be connected to the **top row of the JA Pmod port**; the connection to the joystick should follow the specification in the table below. These connections should be the same as the ones you used for your homework on the two-axis joystick.

Cable (Pmod Pin)	Joystick
Red wire (Pin 6)	L/R+
Black wire (Pin 5)	GND
Yellow wire (Pin 4)	U/D+
Blue wire (Pin 3)	Not used
Green wire (Pin 2)	U/D
Brown wire (Pin 1)	L/R

1. Upon first running the code, a welcome message should appear on the OLED, including the name of the game, your name, and other information.

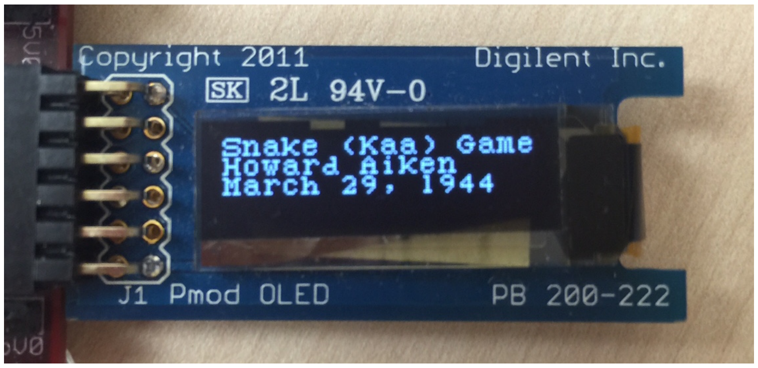

Figure 1: An example of initial information on the OLED display.

2. A second screen appears after five seconds that allows the user to select a point goal in order to win the game. The goal should depend on the number of snake segments. As show in the figure below, pressing button 1 increases the goal whereas pressing button 2 decreases the goal. Obviously the goal cannot be less than 0.

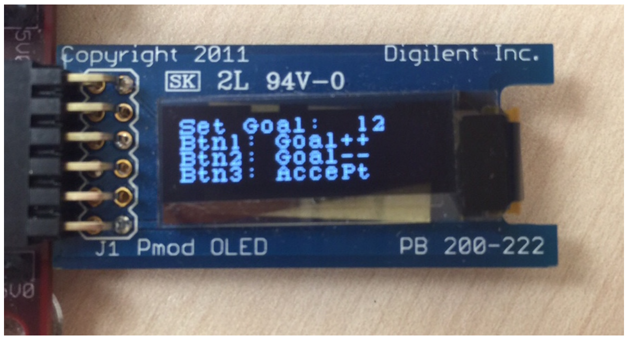

Figure 2: The third screen where the user can select a goal (number of snake segments) for their game.

Pressing button 3, once the goal has been set, starts the game.

3. The OLED display consists of a 128x32 monochromatic pixel screen. So far we have used the screen to display characters, where each character uses an 8x8 pixel array to display a character. Thus, for displaying characters, the screen consists of 4 character rows and 16 character columns. However, for this lab, you will partition the display into a "playing field" on the left and a section on the right for displaying the score and other game information. How exactly you partition up the screen is up to you. However, you need to implement a "playing field" that contains 4 "playing field position" rows by at least 9 "playing field position" columns (how many columns you use depends on how much space you take up for displaying the score). A playing field position (one of the 8x8 pixel character arrays) can hold only one of the following: (1) nothing; (2) the snake's head; (3) a snake body segment; or (4) an apple (note that in the on-line game the snake's head looks the same as a body segment). Thus, you will need to treat each OLED character as a playing field position. To represent the special characters we need for the snake's head, body, and the apple you need to implement user-defined characters (glyphs). A starting point for programming these user-define characters is the glyph example program from class.

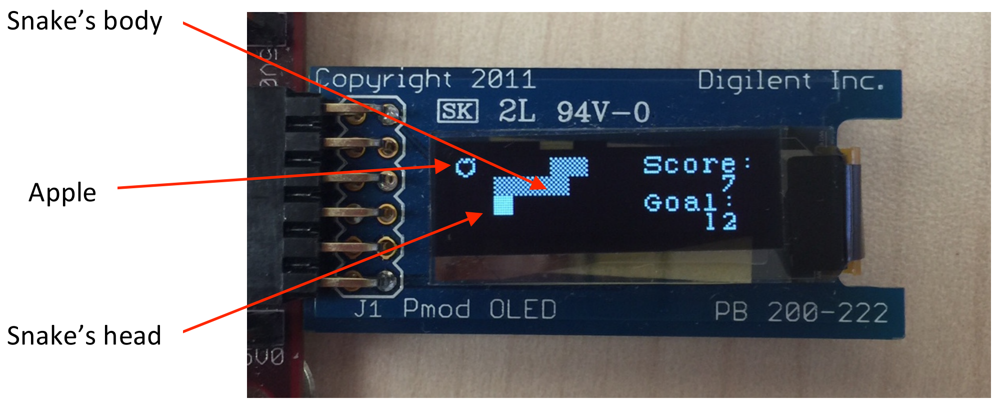

Figure 3: An example showing the game running. The snake is moving based on input from the joystick (explained in section 5). The left portion of the screen is the playing field; the right portion of the screen shows the score.

4. The two-axis joystick controls the movement of the snake. The snake's head moves in the direction given by the last directional input from the joystick (e.g., left, right, up, or down). If the joystick is centered (no directional input), the snake continues to move in the direction last specified. Of course, as everyone knows, the snake's body segments exactly follow the head. Thus, as the head moves forward one position, each body segments moves to where the body segment just in front of it was last time interval.

5. The snake grows one body segment longer every time the snake's head "eats" an apple. Eating an apple consists of the snake's head moving to the playing position occupied by the apple. Once eaten, a new apple should appear at a random location on the playing field not occupied by the snake.

6. As with all games, the game can end in one of two ways, winning or losing. To win, the user reaches their point goal. However, if at any point before winning the snake runs into the edge of the playing field, or runs into itself, the game is lost. How you implement the winning and losing screen is up to you. However, the user can always return to the welcome screen (the first screen) by pressing button 1. At this point they can begin a new game. How you count points is up to you; however, the points should be proportional to the number of apples eaten (and, consequentially, the number of snake body segments).

7. Your code for accomplishing the ADC must be interrupt-driven. The ADC will be used to find the state of the joystick (e.g., left, right, up, down, or centered). Use the example code from class, and your homework code, as a starting point for how to do this. You should use timer3 to start the ADC conversion on the ADC (that is SSRC<2:0>, the conversion trigger source select bits in the ADC control register 1, AD1CON1, must be set to 010). You must also have your interrupts configured to be in multi-vector mode (i.e., not in single-vector mode as with the original Lab1 example code).

8. The exact choice of the speed of the snake is up to you. However, the snake's speed should be constant and not depend on the time to update the OLED display or anything else.

Programming Notes

All the high-level logic in your program should be configured as a finite-state machine—not if-else statements. As we went over in class, to develop an FSM first write down the finite state diagram that corresponds to your logic, then: (1) implement the possible states using the "enum" declaration; (2) define a "system state" variable of this type to keep track of the current state; and (3) implement your finite state machine as "switch/case" statement where the cases correspond to the possible states, and the transitions are those specified in your diagram. *The structure of your code will be a major portion of the grade.* Here are some additional notes about the program.

1. Define short functions that keep your code readable. It is the rare function that has to be more than 20 lines. There should be only one finite-state machine in a function, or other such complex logic. Use static state variables in functions (not global) variables to maintain state between function calls. It should be obvious what the main (and any of your functions) does from your variable names and function names. Anything that has to happen at a particular rate (e.g., the movement of the snake) must use information from the timer, and not depend on delays or on when it runs in the while loop.

2. Obviously, you will have to keep track of where the snake is on the playing field and how long it is. The position of any snake segment has an x and a y coordinate (which row and which column it is on in the playing field), so it would make a lot of sense to define a datatype (via typedef) to store a position. For example, consider the following code segment.

```
typedef struct  __position {
        int x;
        int y;
} position;

position snake_position[MAX_SNAKE_LENGTH];
unsigned int snake_length = 1;
snake_position[0].x = 0;
snake_position[0].y = 0; // snake starts in the upper left-hand corner of the playing field
```

One could use this to define an array to keep track of where the snake's segments are on the playing field. You would have to define which way the snake was "pointed" in the array. That is, the head could either be at snake_position[0] or snake_position[snake_length-1]. This would make it easy to implement the snake's movement. For example, if the snake was moving to the right, and the head was at location 0 in the array, updating the head's position in the array would be the following.

```
snake_position[0].x++;
```

Of course, you would have to update all the snake's segments as well. You could probably figure out a nice way to make this update not involve a bunch of nested if-else statements. Also, note that this representation also makes it easy to check if one has lost the game by going off of the playing field.

3. Drawing the snake on the playing field seems like it would be complicated, but it doesn't have to be. You could draw the state of the playing field directly from the data structure in (2), or you could also keep a representation of the entire playing field—how you draw the playing field on the OLED display is up to you. Suppose you want a representation of the playing field, and suppose that your playing field is 4 rows by 10 columns. You could explicitly represent this playing field as an 4x10 integer array in your code, say be defining the two-dimensional array "int playing_field_array[4][10]". The array value at any location row i, column j, corresponds to the state of that playing field position; thus, we can represent these states a value, say: (0) is empty, (1) is a snake segment, or 2) an apple. (NB: You really should use an "enum" construction here instead of 0, 1, and 2!) You can define each of these 3 values stored in this array as the indices for the respective characters. For example, if we look at the upper left-hand corner of the playing field, perhaps playing_field_array[0][0] = 1 (a snake segment). We can predefine our special characters so that OledDrawGlyph(1) draws the user-define character representing the snake's body segment. There are other ways than this to draw the state of the playing field directly from the data structure in (2) without having to represent the entire field—again, this is up to you.

4. We certainly want our code to be efficient. As we know from class, the thing that takes the longest is updating the OLED. And, in particular, writing the user-defined characters takes much longer than writing a predefined character. Thus, we want to minimize the number of these updates to the screen. Well, thinking about it, note that since the snake's body follows its head, the part of the image that changes (when the apple isn't eaten) is that the head moves forward one position and the last segment of the tail "disappears." Thus, you only have to update the user-defined characters corresponding to changes for the head and tail. On the other hand, if the apple is eaten, you don't have to change the tail (as the snake grows one segment longer). However, when the apple is eaten, you will have to redraw the apple at a new location. Your code should not completely redraw the screen each time the snake moves; instead only update the portions of the screen that need to be updated.

ADDED VALUE FEATURES

You are encouraged to add enhancements to the basic program described above. You may receive up to **ten** points extra credit by introducing totally cool features to the game. You cannot, however, change any of the game requirements. You may use the pushbuttons to control the features that you add.

You should describe any added value features through code documentation. You should document any added features in Word document that you submit with your lab. You have some latitude in designing your game as it comes to game play options, graphics, and other aesthetic choices. Make sure that you conform to the specification's minimum requirements, and use your creativity to expand the player's game play options.

SUBMITTING YOUR LAB MATERIALS TO SCHOLAR

Name your project <last name>_Lab3. Create a ZIP archive of the entire <last name>_Lab3.X directory. A grader should be able to unzip the archive, open the project, and run it as is.

Also submit the validation sheet. You should describe any added value features through code documentation and in the validation sheet, so that the GTA can validate their implementation. You will have a place to describe these features on the validation sheet.

Refer to the Coding Style Guidelines as in Lab2 (that have been posted on Scholar). Be sure to comment your code appropriately. All functions should have block comment headers with at least the following fields: description, pre-conditions, inputs, outputs and post-conditions. As needed, insert in-line comments to document code that would not be clear to someone else (*e.g.* the GTA) who has to review it.