**EE 425**

**Experiment 5**

**Game Development for PIC18F4520:**

**‘PIC Shooter’**

Summer 2016

Section 1XB

Group 4

Members: Steven Eisinger, Kim Bhagwan

Table of Contents

[Objective 3](#_Toc456840807)

[Overview 3](#_Toc456840808)

[The PIC Shooter Program 3](#_Toc456840809)

[Part 1: Sprite Definitions, Mainline, and the GameInit subroutine 3](#_Toc456840810)

[Part 2: The DrawShip subroutine 6](#_Toc456840811)

[Part 3: The DrawEnemy subroutine 7](#_Toc456840812)

[Part 4: The GameOver subroutine 9](#_Toc456840813)

[Conclusion 10](#_Toc456840814)

[Works Cited 10](#_Toc456840815)

Objective

Our goal is developing a small game for the PIC18F4520 microcontroller using the LCD, rotary pulse generator (RPG), and push button provided on the QwikFlash board. In this game, the player controls a spaceship and must shoot incoming aliens to survive. The game will be of our design, coded from scratch, and will be a simpler version of games like *Space Invaders* or *Galaga*.

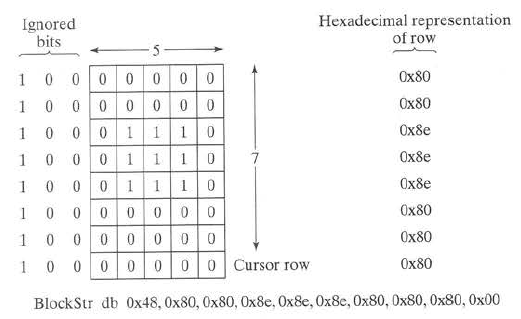
Overview

To approach a task as complicated as a game, we wrote our code with modularity in mind. All of the characters displayed on the screen (we will call them ‘sprites’) are drawn in their own subroutines, each of which will be described in this report. Programming a game for the PIC using assembly is very different from using higher level languages because we don’t have access to proper IF/ELSE statements, which are used heavily in game design. Chapter 3 covers methods of conditional branching in detail. Using pointers and indirect addressing is described in chapter 2, which we had to use in our sprite drawing subroutines. We want to use custom non-ASCII characters as our game sprites; the Hitachi LCD allows us to define eight of them (Chapter 7.7). RPG usage is described in chapter 8 of the textbook and required some modification to be used with our program. Other concepts were used, such as controlling the LCD refresh rate and generating pseudo-random numbers, which are not described in the textbook. There are some subroutines used in this program, such as LoopTime and ByteDisplay, which were described in previous reports and will not be covered here. All code is written in a single file called *GamePIC.asm*.

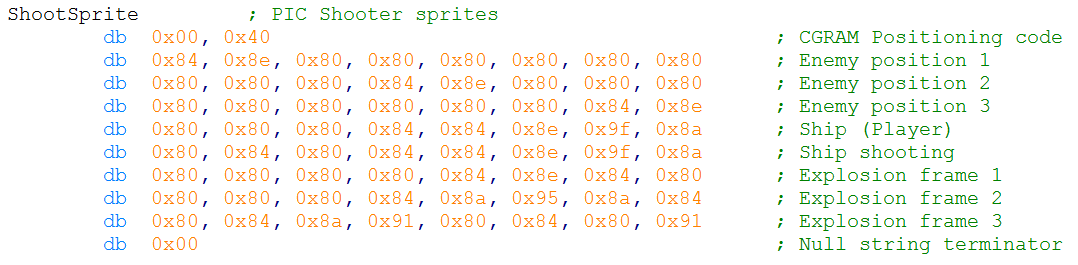
The PIC Shooter Program

Part 1: Sprite Definitions, Mainline, and the GameInit subroutine

Before programming the actual game, we must design the sprites. We have 8 available character slots the LCD gives us (accessed with codes 0x08 to 0x0f) which we can use to store custom characters. Each character can take up an 8x5 block of pixels on the LCD screen, and we turn on individual pixels in a row using the lowest 5 bits of a byte (the highest 3 bits are ignored); a set bit turns that pixel on and a clear bit turns that pixel off. *Figure 1* from chapter 7.7 demonstrates how bytes are used to define each row of pixels. A CGRAM (character-generating RAM) positioning code must be sent to the display first so we can save our characters to the LCD and access them later. The eight positions are 0x40, 0x48, 0x50, 0x58, and so on until 0x78. These positions are different from the access codes, and will only be used once before the main loop begins. We only have to give the display the position of our first character, and it will keep writing sequential characters until it sees a null character (0x00). We save the characters in exactly the same way as writing a constant string to the LCD by using POINT and DisplayC. *Figure 2* shows the code we used to define our sprites and *Figure 3* shows what they actually look like.



*Fig. 1. Defining a custom character for the Hitachi LCD. 1 is on, 0 is off.*

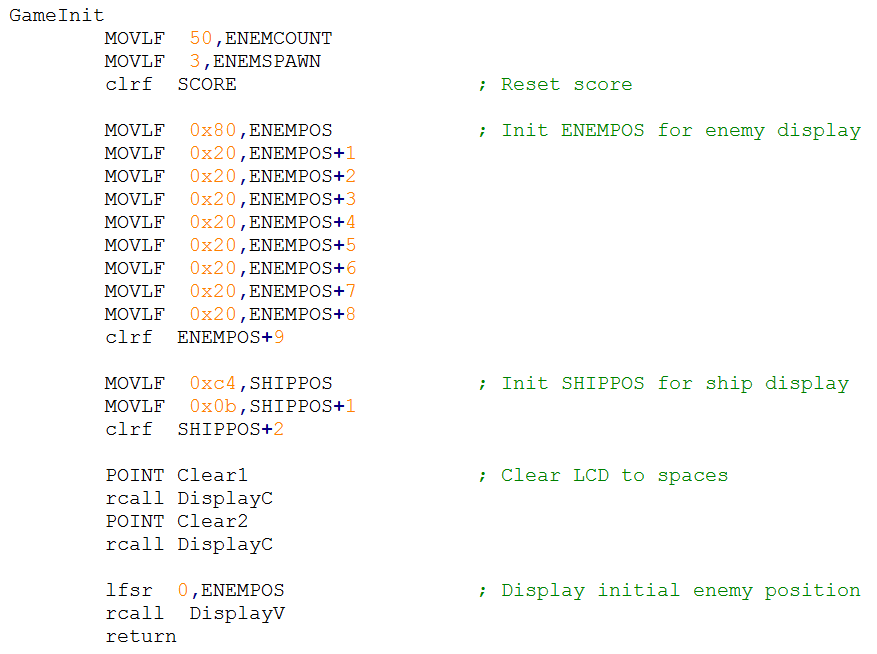


*Fig. 2. PIC Shooter sprite definitions.*



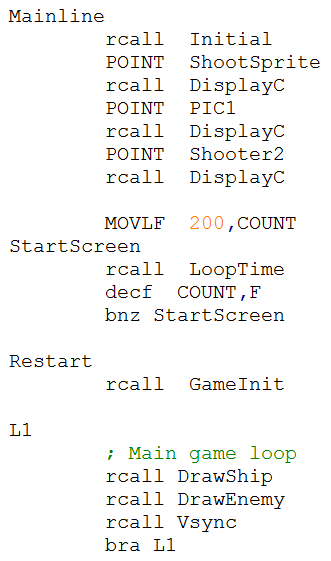
*Fig. 3. PIC shooter sprites (in the order defined from left to right).*

Unlike our previous programs, we have two initialization subroutines. Initial, seen in previous labs, initializes the port and LCD and is only called once when the PIC is turned on. We copied this from *Experiment 1* then made sure pins D0 through D3 were set as input, as we will be using them to control the game. ‘GameInit’ initializes variables used by the game, such as the ship and enemy display strings, and prepares the LCD to be used by the game. GameInit is called every time the game restarts, which will be after a game over occurs (the GameOver subroutine will be discussed later). This guarantees the game begins with a fresh slate and a score of zero. The variables in GameInit will be discussed together with the subroutines which use them.



*Fig. 4. The GameInit subroutine with comments.*

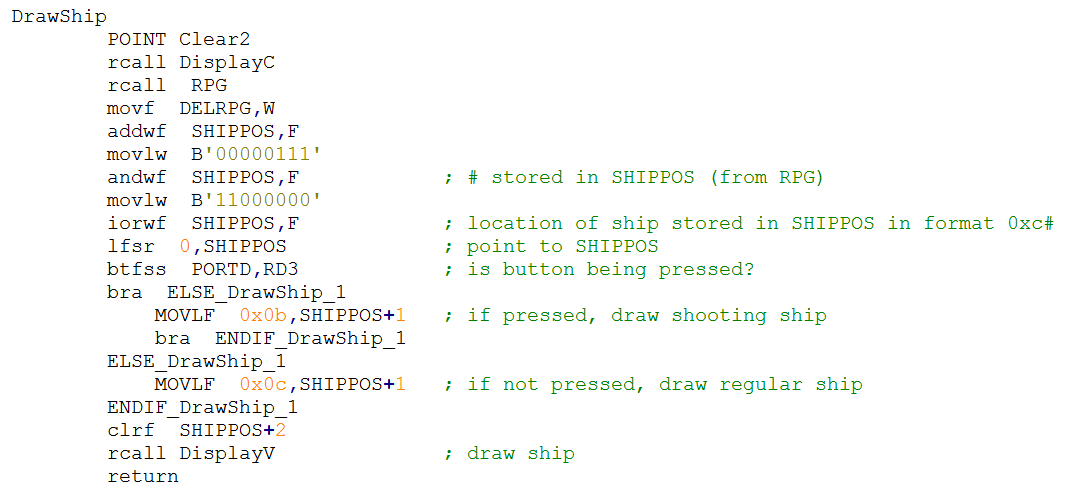
Mainline for the game is extremely simple because of how modular the code is. The PIC is initialized, the sprites are loaded to the LCD, “PIC Shooter” is displayed on the LCD for 2 seconds, the game variables are initialized, then the main game loop starts. The main game loop is simply a call to DrawShip, which handles ship movement and shooting, DrawEnemy, which handles enemy generation and interaction, and Vsync. Vsync simply calls LoopTime twice (which is defined to wait for 10ms), resulting in a 20ms pause before the loop occurs again. We do this to slow down how many times we draw to the LCD because drawing too quickly will result in stuttering characters. If we refresh the display too slowly, characters on the display will appear to have a delayed response to the input. 20ms for Vsync was chosen subjectively through trial and error. After game over occurs, only GameInit will be called before entering the main game loop again.



*Fig. 5. Mainline for the GamePic program.*

Part 2: The DrawShip subroutine

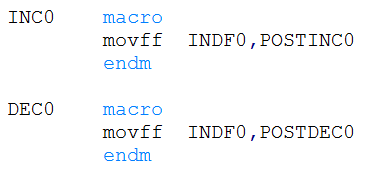
DrawShip handles drawing the ship sprite to the bottom row of the LCD. The position of the ship is determined by the rotary pulse generator and whether the ship is shooting or not is determined by the button being pressed. DrawShip will always clear the entire bottom row of the LCD before drawing the ship so we overwrite the ship’s previous location. Drawing the ship to the proper location is done in a similar way to ByteDisplay; an array of 3 bytes called SHIPPOS is defined, we place the cursor-position code at the beginning, the ship sprite code in the middle (0c0b for standard, 0x0c for shooting), and the null terminator at the end, then we point to the cursor-position register using ‘lfsr 0,SHIPPOS’ and call DisplayV. Whether or not the button is pressed, checked by testing bit RD3 in PORTD, determines if the ship is shooting or not. To determine the ship’s position, the ‘RPG’ subroutine is called. RPG determines if the rotary pulse generator was rotated one position clockwise, counterclockwise, or unmoved, then stores the corresponding value (1, -1, or 0, respectively) in DELRPG. DELRPG is then added to SHIPPOS (which stores the cursor-position) and is manipulated to point to the bottom row again just in case the ships position went below 0xc0 or above 0xc7. This allows the ship to appear on the opposite end of the LCD if traveling past the edge. The RPG subroutine is copied exactly as seen in Chapter 8 Figure 8-3 of the textbook, however since we did not have access to the preprocessor (SASM.EXE) mentioned in the book, the IF and ELSE statements had to be modified to be compatible with the standard MPASM compiler by replacing them with ‘btfsc’, ‘btfss’, and ‘bra’ statements. We used the RPG instead of the potentiometer because it doesn’t require A/D conversion and gives us steady discrete increments.



*Fig. 6. The DrawShip subroutine with comments.*

Part 3: The DrawEnemy subroutine

The job of the DrawEnemy subroutine is to spawn new enemies when it determines it needs to, animate enemies to move downward if they already exist, destroy enemies when the player shoots them, keep track of the score, and call the GameOver subroutine when an enemy would reach the bottom row of the LCD. We determined we want an enemy to spawn every 3 seconds at a random location (unless that location is already populated by an enemy) and to animate the enemies down one position every second. This would result in at most one enemy displayed on the screen at a time, however this is easily modified. The variable that determines how often we animate the enemies is ENEMCOUNT while the variable that determines when to spawn an enemy is ENEMSPAWN. ENEMCOUNT stores the number of iterations of Vsync before animating the enemies and ENEMCOUNT stores the number of times ENEMCOUNT must reach zero before spawning an enemy. Since the enemy positions are determined by sequential sprites, we can easily animate an enemy by incrementing its character code and redrawing it. ENEMPOS is the 10-byte array which stores the cursor position for the top row (0x80) in its first position, null terminator in its last position, and all enemy characters in between. Pointers are used extensively to locate enemy positions, so we wrote to macros to easily increment and decrement a pointer address – INC0 and DEC0 – as shown in *Figure 7*. To dereference a pointer, INDFx, POSTINCx, POSTDECx, PREINCx, and PREDECx can be used in place of the variable name, where x is the number of the FSR we want to dereference. POST- means the pointer will be modified after dereferencing and PRE- means the pointer will be modified before dereferencing. INDFx is used to dereference the pointer and leave it unmodified. To generate a random number to determine where the next enemy will spawn, we simply use the lowest 3 bits of TMR0L as a pseudo-random number.



*Fig. 7. The INC0 and DEC0 macros.*

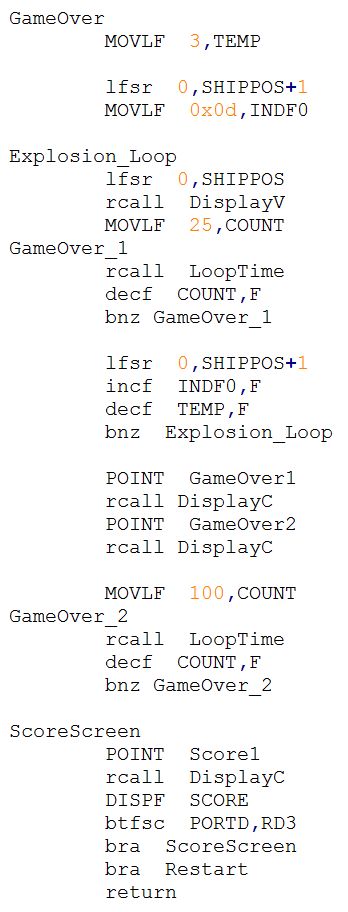
Before calling DrawEnemy, ENEMYPOS is initialized to point to the top row of the LCD and store all spaces until the end of the string, then drawn to the screen. We use spaces to determine whether or not an enemy is there because it is easy to compare the ASCII code for space (0x20) to the code currently stored in an index of ENEMYPOS. We can subtract 0x20 from the value in that position; if it’s zero we know there isn’t an enemy there, otherwise there is. Whenever an enemy is destroyed, we replace it with a space character. The flow of DrawEnemy routine can be seen in *Figure 8*. Due to its complexity, the flowchart simplifies some aspects of the code.



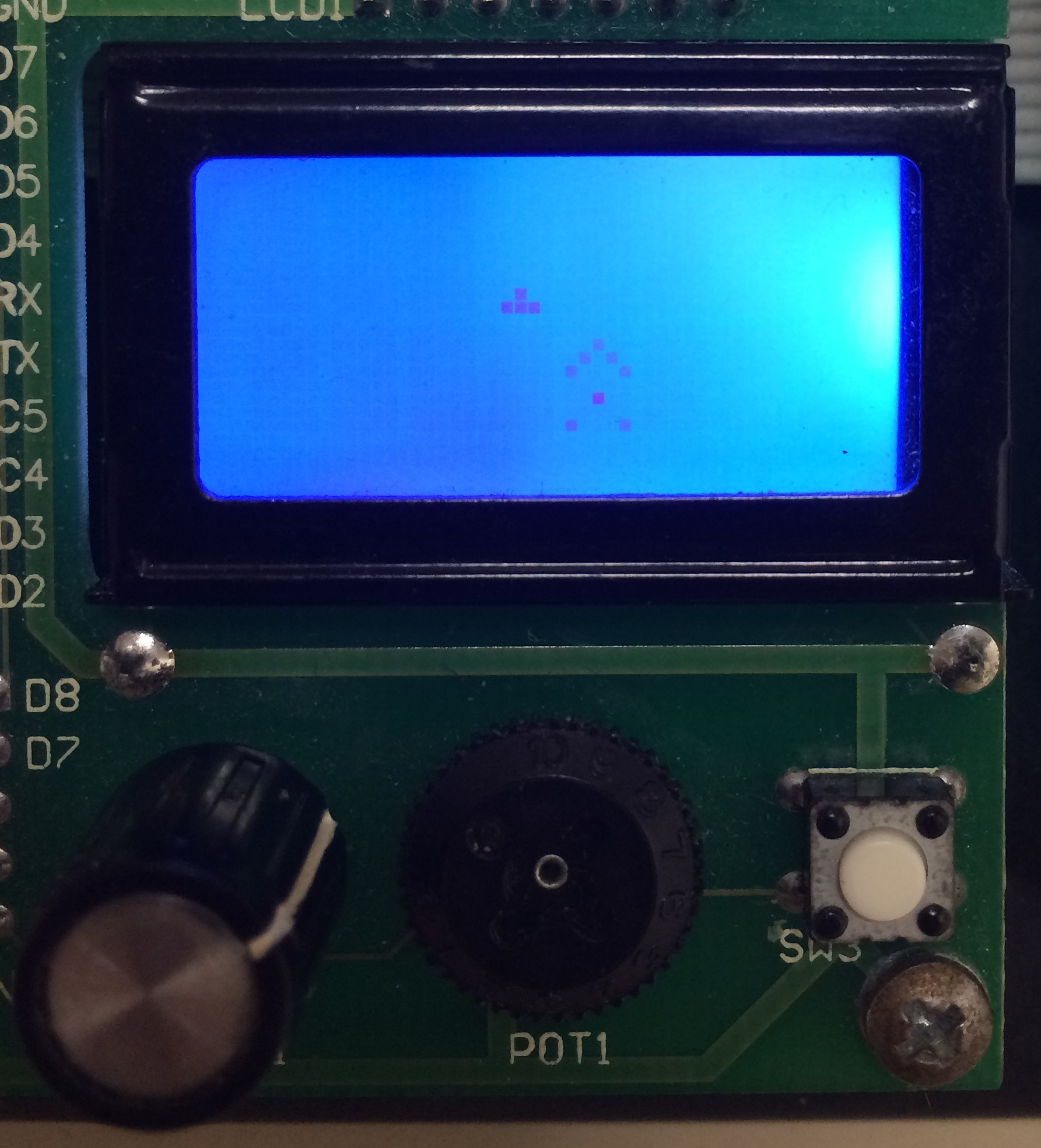
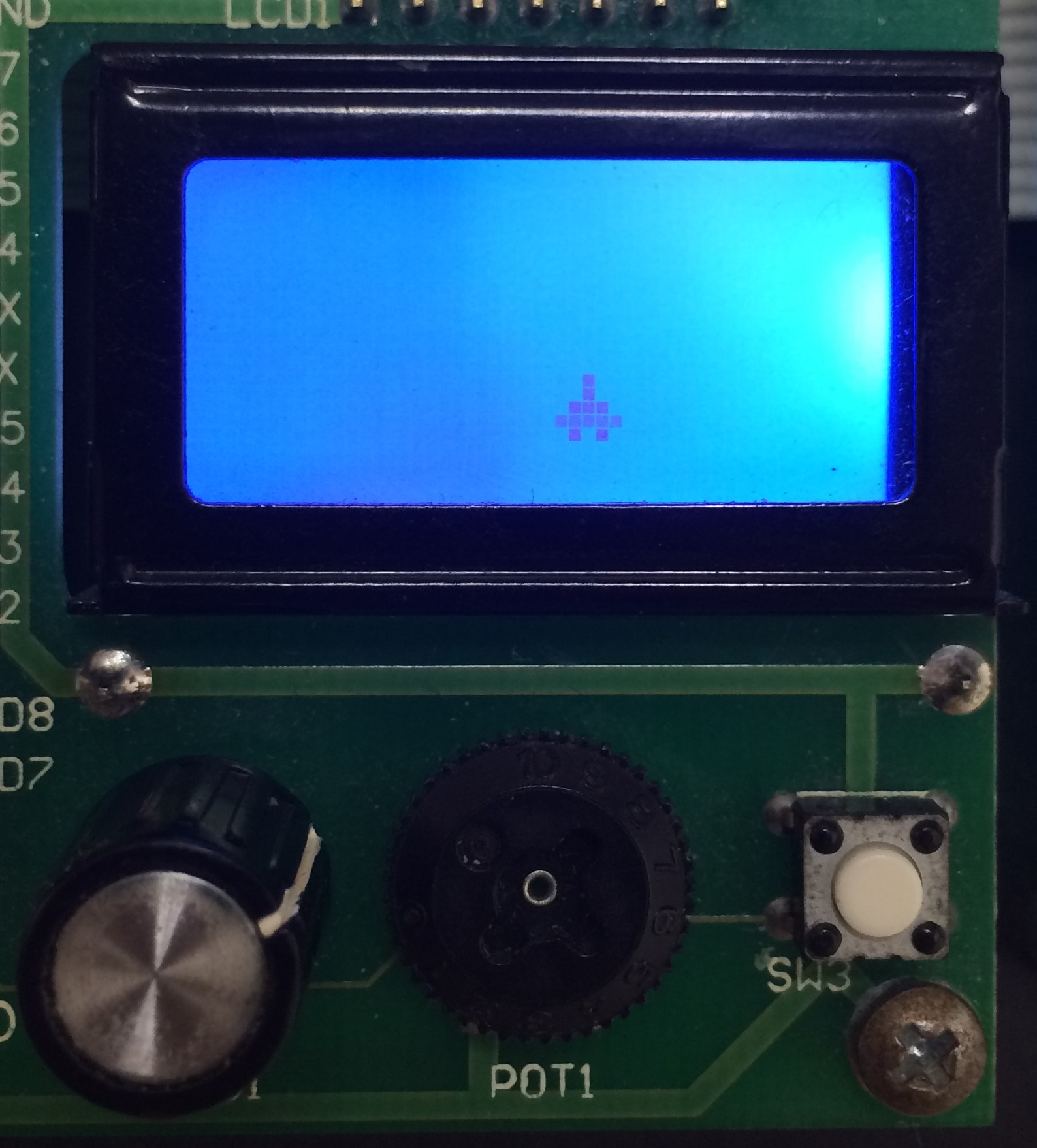
*Fig. 8. Flow of the DrawEnemy subroutine.*

Part 4: The GameOver subroutine

When an enemy would reach the bottom row of the LCD (meaning an enemy is being displayed with sprite 0x0a and 1 second passes without the player shooting it) the game is over. The GameOver subroutine will be called, which starts an explosion animation at the ship’s location, displays “GAME OVER” on the screen, then displays the score as a binary number. If the button is pressed after the score is displayed, GameInit will be called and the game restarts. The explosion animation works by replacing the ship with the first explosion sprite (0x0d), displaying it with DisplayV then incrementing the character code and repeating two more times, with quarter second pauses in between. This makes the explosion animation ¾ seconds long. “GAME OVER” is displayed using POINT, DisplayC, and predefined strings and remains on the screen for 2 seconds. After this time, the score is displayed as a binary number on the bottom row using the DISPLAY (or DISPF in our code) macro from *Experiment 2* and “SCORE:” is displayed on the top row. The score screen remain on the LCD indefinitely until the button is pressed, which will restart the game. Images of the completed game in action can be seen in *Figure 10*.



*Fig. 9. The GameOver subroutine.*



*Fig. 10. Gameplay images of PIC Shooter.*

Conclusion

The game was a challenge in creativity and organization. We were able to use the RPG and define custom characters, which we hadn’t used before, and create a fun game. In its current form, the game has no difficulty levels. In future implementations, enemy spawn rate and enemy speed can increase after you reach certain score thresholds. The game also has a way to “cheat” which is by scrolling the RPG back and forth really fast and holding the button down. This could be prevented by only allowing the player to shoot again after the button is released. Displaying the score as a decimal number would also be an improvement. Using a speaker or the D/A serial output module, we could also implement sounds effects for the game. With the knowledge we obtained making this game, we can design many different types of games. Using a larger LCD or a VGA display would open up many different possibilities for graphics, animation and movement.

Works Cited

Peatman, John B. *Embedded Design with the PIC18F452 Microcontroller*. Upper Saddle River, N.J.:

Prentice Hall, 2003. Print.