Project #3

Michael Schoen, Abdirahman Osman, Illya Starikov

Due Date: Tuesday, December 6th, 2016

For the second project, we were delighted to be able to use a higher level programming language; we[1] decided to apply this new-found excitement to implement a retro game from the 70s: Space Invaders.

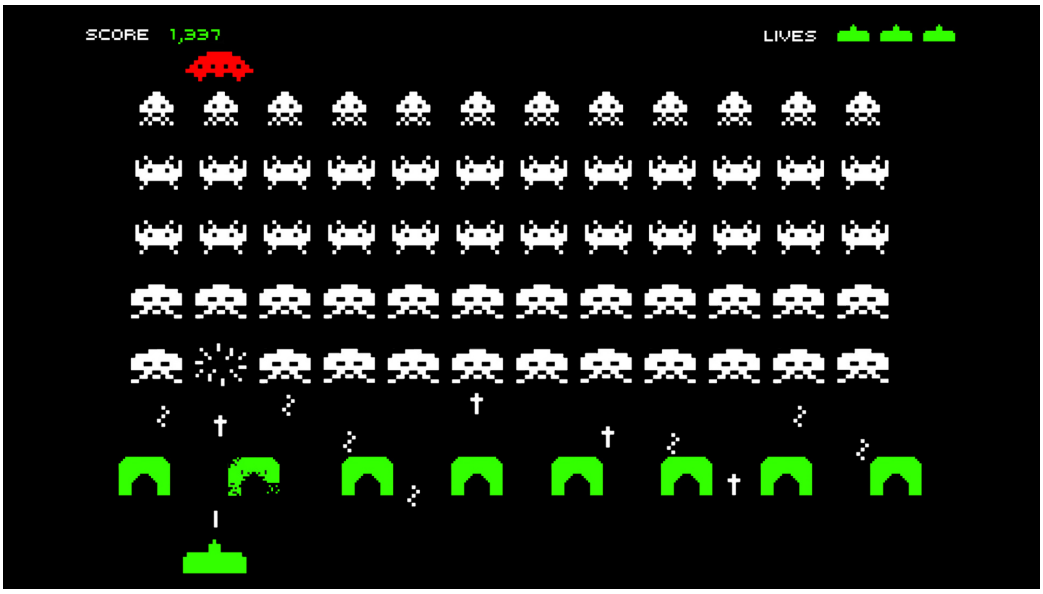

Figure 1 – The original space invaders.

When this idea came crashing and burning down to the ground, we decided to combat this with a additional music, an elegant user interface, and a keyboard (with some other miscellanies).

[1] Illya.

1 Project Description

Below we will go into more detail about each individual parts of our project.

1.1 The Game

The game we initially decided to go with was space invaders. We had intention of doing the game logic on the microcontroller through serial communication; however, we learned early that this was likely not be possible[2]. We describe in *Problems Encountered* how we absolved this. From here, we decided our game would exclusively be in the terminal.

Our game essentially creates a two dimensional array (in three segments — the header to show score and level, the aliens, and the shooter). Then we loop through depending on the input:

- ← Move the shooter left.
- → Move the shooter left.
- q Quits the game
- Space Shoots with the gunner.
- No input meant refresh game.

We achieved the drawing through `curses`[3]. Ultimately, we did not get our game fully done, but a good majority of it. See Figure. 2 for a look at the UI of the game.

1.2 Music

The formula for each note is

$$\frac{1/4 \,\text{Oscillator Frequency}}{\text{Note Frequency}}$$

So for supposing we want to produce the note $C4$,

[2]Our hex file with very basic functionality was 25 kB.

[3]Can be read about here `https://en.wikipedia.org/wiki/Curses_(programming_library)`

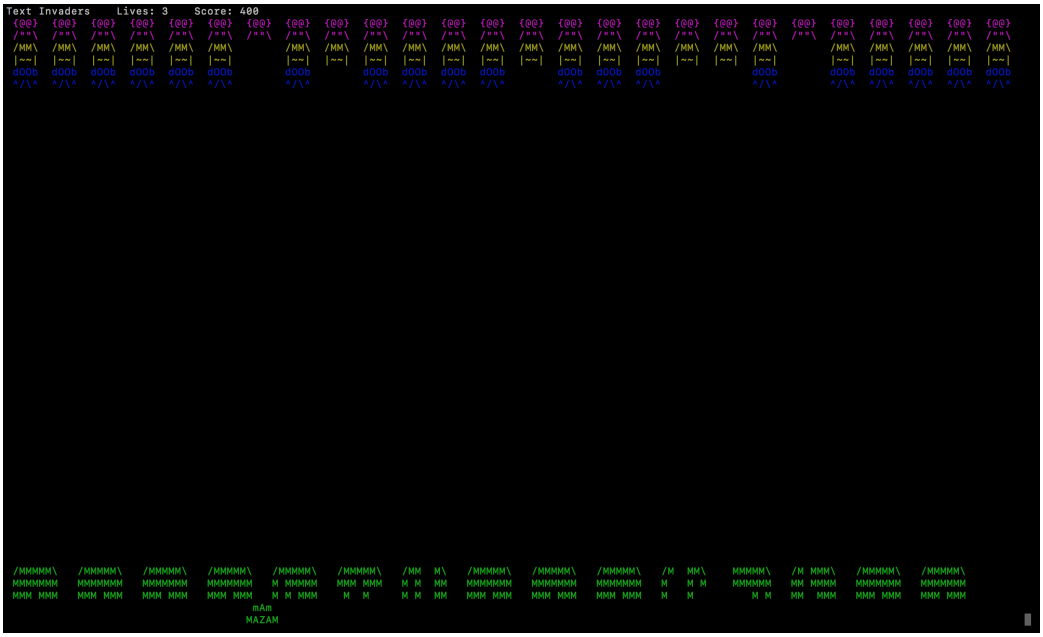

Figure 2 – Text Invaders, our version of space invaders..

$$\frac{1/4\,\text{Oscillator Frequency}}{\text{Note Frequency}} = \frac{1/4 \cdot 7.373\,\text{Hz}}{261.63\,\text{Hz}} = 7.045\,\text{ms} \tag{1}$$

To produce the music, we had a function that took in which note and the type of note (16^{th}, 8^{th}, 4^{th}, half or whole note). The function multiplied the base length of a note by a constant depending on the type of note. A 16^{th} note would be multiplied by 1, an 8^{th} by 2, a quarter note by 4, a half-note by 8, and a whole note by 16. We used timers and interrupts for the period of the note and the length of the note.

We also used sheet music to get the notes. For speeding and slowing down the music, the type of note was just multiplied by 2 or divided by 2 respectively.

1.3 Interactive User Interface

The menu goes through three stages; printing the pen-rose triangle background, printing options, then allowing the selection of items. The following are printed

1. Counter

2. Hurricane's Eye

3. Keyboard

respectively. From there the user has the option to to cycle upwards using P_2 (see Figure. 3), cycle down using P_8, or select using P_5. The cycle starts with 1, highlighting Counter if P_2 is pressed, then Hurricane's eye will be highlighted, and if printed once again Counter will be selected. If the menu user presses P_1 once at the 3rd option (Keyboard) then the menu will cycle back to option 1 and vice versa if P_8 were clicked at the 1st option. Selecting an menu option with P_5 while highlighted will start its respective function. Also each menu option has a corresponding number displayed on a seven segment display.

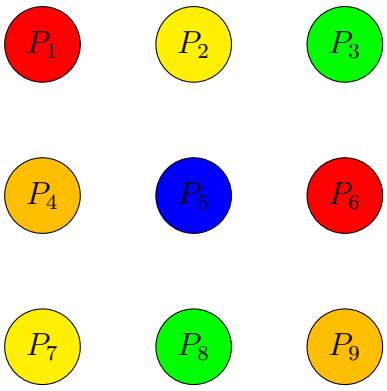

Figure 3 – The Main Menu.

1.4 Seven Segment Display

For the seven segment display, we had to set P_2 to be push/pull so that the display would receive enough power. We then looked up the correct values to output to the pins to make each number show up on the display and used those values in the increment/decremented function as well as the simple function that lets you display any number. We used the data sheet for the seven segment display to see how to connect it on the breadboard. See Figure. 4 for additional information.

PIN NO.			
1	Cathode E	6	Cathode B
2	Cathode D	7	Cathode A
3	Common Anode DIG. 1	8	Common Anode DIG. 2
4	Cathode C	9	Cathode F
5	Cathode DP	10	Cathode G

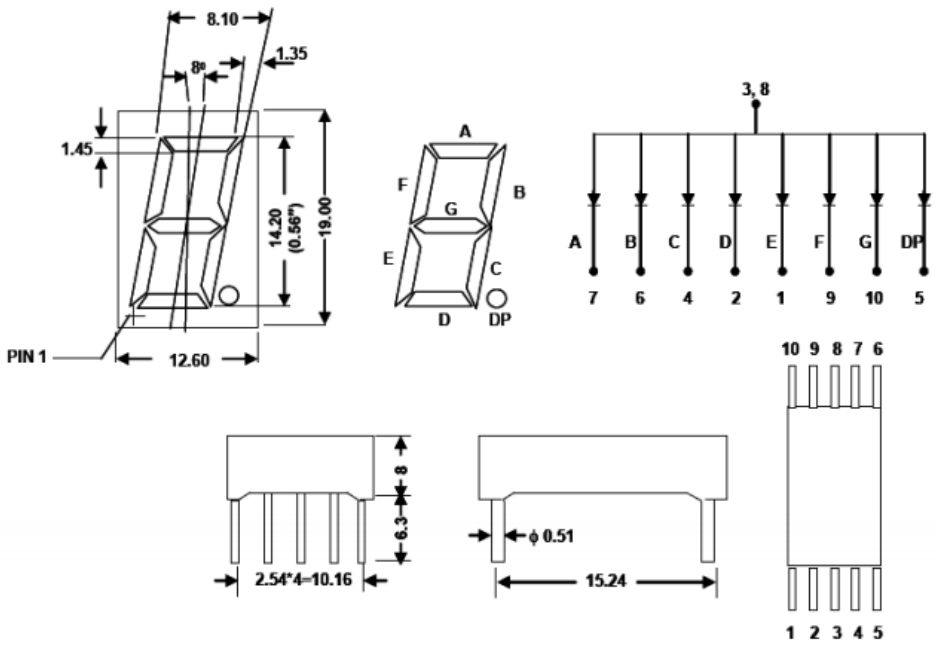

Figure 4 – The schematic for the seven segment display.

1.5 Keyboard

Our keyboard has a C scale set to switches $P_1 \cdots P_8$ with P_9 escaping to the menu. Each switch uses the corresponding light associated with its position save P_9 which is the escape button.

1.6 Curses

In order to print to any part of the screen we needed to implement a structure to send ANSI control sequences and characters to the terminal using `uart_get()` as a base. To do this we implemented an 8051 port of curses that takes in (x, y) coordinates as arguments for printing `chars` and `strings` at any defined (x, y) location. This allows for the implementation of the menu since it requires printing of options in sometimes nonlinear fashion.

We found a whole host of others issues associated with printing characters to a terminal, printing to an arbitrary location, control sequences, fluctuation in crystal timings, and requesting information from a terminal. We accounted for these individually.

2 Problems Encountered

Below we will list some of the "several" problems we ran into.

2.1 Memory Constraints

By far, the biggest problem we encountered was the memory constraint. The 8051 has 8 kB of memory; our code base, with the exclusion of all the `malloc`s[4], it is roughly 25 kB — a size a bit larger than the 8051 allows. Our workaround was to fight fire with fire.

Instead of "dumbing" down our game[5] to get it to fit, we decided to have an external interface; specifically, a port sniffer. It would listen for input from the 8051, and if there is a signal on the serial port, use that as input. If not, default to the keyboard input.

Ultimately, we were unsuccessful with the port sniffer. Originally, we had tried to use a Linux port sniffer so we can just embed it into our program, `slsnif`[6]. Unfortunately this port sniffer does not support "legacy" ports, and the 8051 falls in this category. So we moved onto a Windows port sniffer,

[4]Since `unsigned char` = 1 B, the terminal window will roughly be 20 height × 80 width, we can roughly expect 1.6 kB to be allocated on the heap; a non-insignificant amount compared to 8 kB.

[5]According to back of the hand calculations, a space-optimized version of the game would still be 7 kB. This was likely to be impossible.

[6]Can be found at `https://sourceforge.net/projects/slsnif/`.

`Serial Input For Windows`[7]. This too did not work, because we could only have one interface use the serial port, so we would need a dedicated socket to intercept the `COM1` port's input — something we were not familiar with.

Ultimately, we were simply unsuccessful and had to scrap this.

2.2 Printing Lines and Columns.

We had an issue with the timing crystals of the Simon board where the board heating up would cause the serial port to start printing unrecognizable characters. This lead us to think we were causing a segfault when we was printing string literals. This caused hours of wasted time since nothing we could change would cause the board to start printing what we wanted, but one afternoon after hours of exhaustive research and help at the lead sessions the conclusion was reached that heat changes were causing the board to lose its timing and print out nonsense. The fix for this was to let the board cool off while writing code. After this change garbage outputs dropped significantly.

2.3 Seven Segment Display Buttons

One problem we ran into was some of the buttons use the same pins as we used for the seven segment display. The display used push/pull which makes the buttons not work, so we had to carefully chose buttons that didn't use the same pins.

3 8051 Architecture

For this project, we made sure to utilize the additional functionality we learned in the later parts of the semester — specifically, timers, serialization, and interrupts.

We spent a great deal of time with the serial communication aspect of the board; we took advantage of this the most (with respect to the 8051 architecture). We have already discussed the things that made our board unique; however, we will show an example of how our implimination of `curses`.

[7]Can be found at `http://www.randomnoun.com/wp/2013/02/03/serial-input-for-windows/`.

```
void move(unsigned char y, unsigned char x) {
    unsigned char i; // counter
    zero_cursor(); // moves the code to (0, 0)

    for(i = 0; i < x; i++) {
        cursor_jump(1, 'R'); // move cursor to the x position
    }
    for(i=0; i < y; i++) {
        cursor_jump(1, 'D'); // now to the y
    }
}
```

Now combining that with `addch(char)`,

```
void addch(unsigned char value) {
    uart_isr();
    uart_transmit(value);
    while(TI==0);
}
```

we have a fundamental a functionality to `curses`, printing anywhere!

Next, we implemented all delays with timer interrupts. For example,

```

void delay1ms() {
    TH0=-0x0E;
    TL0=-0x66;
    TR0 = 1; // start
    while(TF0 == 0); // poll to finish
    TR0 = 0; // stop
    TF0 = 0; // clear the finish
}
```

delays for 1 ms, because $(\mathrm{FFFF}_{16} - 0\mathrm{E66}_{16} + 1) \times 1.085\,\mu\mathrm{s} \approx 1\,\mathrm{ms}$. As it turns out, we mostly needed millisecond precision, so we used this as the foundation for the rest of the program.

4 Individual Features

- Michael Schoen — 33% Contribution
  - Song
  - Seven Segment Display
- Abdirahman Osman — 33% Contribution

  - Port Serialization
  - Text User Interface
  - Menu
- Illya Starikov — 33% Contribution
  - Space Invaders Game
  - Keyboard

Happy Holidays
From Michael, Abdirahman, and Illya!