SPACE INVADERS: SI LABS 8051 MICROCONTOLLER

Design Documentation

Team Members: Dan Newton
Aaron Spilker

Document Date: 4/27/2016

(This page is intentionally blank)

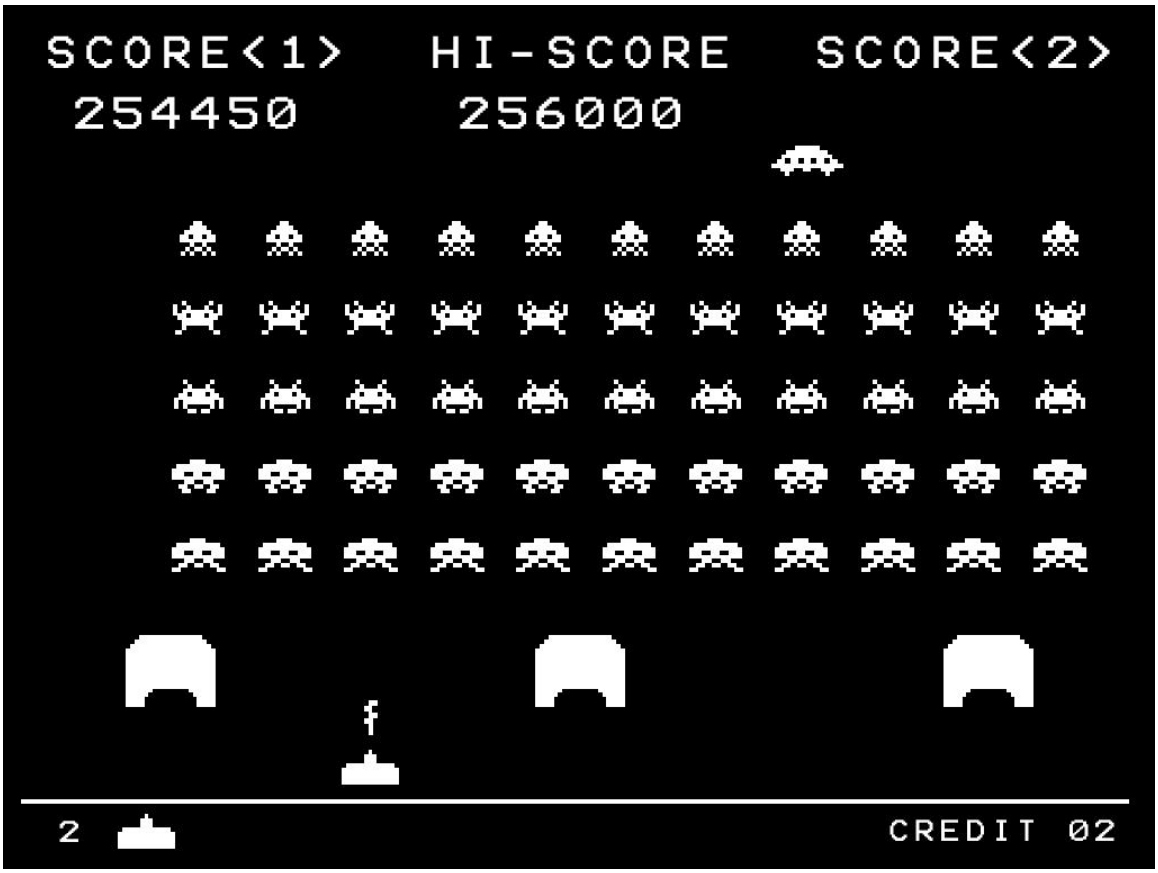

Figure 1: Original Space Invaders

Explanation of Original Game play:

"Space Invaders is a two-dimensional fixed shooter game in which the player controls a laser cannon by moving it horizontally across the bottom of the screen and firing at descending aliens. The aim is to defeat five rows of eleven aliens—some versions feature different numbers—that move horizontally back and forth across the screen as they advance towards the bottom of the screen. The player defeats an alien, and earns points, by shooting it with the laser cannon. As more aliens are defeated, the aliens' movement and the game's music both speed up. Defeating the aliens brings another wave that is more difficult, a loop which can continue without end."

Table of Contents

List of Figures

List of Tables

Revision History

Date	Author	Revision
4/13/2016	**Aaron Spilker**	**Initial**
4/27/2016	**Aaron Spilker**	**-**

1.0-Introduction

The purpose of this project is to design and develop a prototype of the game *Space Invaders* using an 8051 microcontroller, and a peripheral board. This game will be implemented using buttons, switches, a potentiometer, a speaker, and a LCD screen, all which are located on the aforementioned peripheral board.

This project includes both hardware assembly tasks, and software design. The peripheral hardware components were supplied. These components were than assembled by this team. The software was not supplied the c-code was designed, and downloaded by this team to the 8051 using SI Labs IDE software tools.

2.0-Scope

This document discusses the software design of this project, which includes all header and c-code files used to implement *Space Invaders*. This document does not discuss the CCA layout of the 8051 microcontroller or the peripheral board, instead discusses the 8051 ports used, and their pin-to-pin connections between the 8051, and each peripheral.

This document does discuss game play set-up, game play rules, and how to navigate the game successfully from start to finish.

3.0-Design Overview

3.1 –Requirements

1. The system shall run on an external 9v DC supply.
2. The system shall use a 64x128 pixel LCD.
3. The system shall have a reset button, a start button and a fire button.
4. The system shall have a potentiometer located near the bottom of the LCD to control the position of the laser cannon.
5. The laser cannon shall be seven pixels high. The width shall be to 7, 9, 11 or 13 pixels depending on the DIP switch.
6. Invaders shall be 6-8 pixels high. Those on the front row shall be 10 pixels wide with 3 pixels between them. Those on the back row may be narrower.
7. The system shall keep a 4-digit score that tallies the number of aliens destroyed by the current player.
8. When the start button is pressed, the score shall be set to zero and the number of extra lives (laser cannons) shall be set to 3.
9. The display shall always show the score and the number of lives (laser cannons) remaining.
10. At the beginning of each wave, the display shall show (a) the laser cannon (at the bottom of the screen), and (b) 2 rows of 8 alien space invaders (arrayed on the left side of the screen as far from the laser cannon as possible).
11. Once the battle begins, the formation of invaders shall move to the right, firing lasers at random. When the invaders reach the side of the display, they shall move 8 pixels closer to the laser cannon and reverse direction.
12. The invaders shall move left or right one pixel at a time. The movement speed of the invaders shall be roughly inversely proportional to the number of invaders on the screen.

13. The system must be able to display up to 8 simultaneous laser bursts (i.e. projectiles) from the invaders and up to 4 simultaneous laser bursts from the player's laser cannon. All laser bursts are one pixel in size and move at a uniform speed. To improve visibility, trailing pixels (up to 7) shall be displayed.

14. If a laser burst hits an invader, the invader shall be destroyed. If all invaders are destroyed, two new rows of aliens shall be arrayed 8 pixels closer than before (but no more than 16 pixels total), and the battle continues with the movement speed of the aliens increased.

15. If a laser burst hits the laser cannon or if an invader reaches the bottom row, the laser cannon shall be destroyed. If there are lives remaining, the invaders regroup as they were at the beginning of the wave, and the battle restarts with a new laser cannon. If no lives remain, the game is over.

16. A sound shall be generated each time the laser cannon fires, each time an invader is destroyed or when the laser cannon is destroyed. The sound for each of these three events shall be different and shall not exceed 250 milliseconds.

17. If, when the game is over, the words GAME OVER shall appear until the start button is pressed.

3.2 -Dependencies

The 8051 and peripheral board are powered using a Switch-Mode AC/DC Power Supply. The power supply inputs are 100-240 Volts, 50-60Hz, and 0.6 Ampere. The power supply outputs are 9 Volts, and 1.7 Ampere.

3.3 -Theory of Operation

When the system is reset or power is applied the LCD displays the start menu, where gameplay will not begin until the start button is pressed. Using the dip switches the user has the capability of selecting the width pixel length of the cannon. The larger the width of the cannon, the more susceptible the cannon is to getting destroyed. There are four different cannon sizes that can be set using Switch 8 through Switch 5.

Cannon Size (WxL)	SW8	SW7	SW6	SW5
7x7	1	0	0	0
9x7	0	1	0	0
11x7	0	0	1	0
13x7	0	0	0	1

Table 1-Cannon Modes

The gameplay will begin when the start button is pressed, this is the button to the left of the LCD screen. If the cannon mode is to be changed, a reset or power up is required *(Note if the reset button is pressed, the LCD will display the score and the lives, and will wait for the start button to be pressed).* During gameplay the right 50kΩ potentiometer is used to move the cannon across the screen. The full range of the potentiometer is used, and if it is fully turned clockwise the cannon will be positioned at the most right position of the screen, and if counter clockwise the

cannon is positioned at the most left of the screen. The cannon can move from side to side, while firing bullets toward the invaders. The cannon can be firing up to four different bullets on the screen at a time. To fire a bullet the button to the right of the LCD screen is pressed.

There are two rows of invaders that move to the right, which shoot up to eight beams on the display at a time. Once the invaders reach the most right edge of the screen, they move forward eight pixels, and begin moving left. The movement speed of the invaders is proportional to the number of invaders left, so the more invaders destroyed, the faster they advance. The player must move the cannon to avoid beams coming from invaders, while trying to destroy the invaders using the bullets from the cannon. Each time a bullet is fired from the cannon, a sound is played, also a different sound is played if a bullet makes contact with an invader.

Gameplay will end when all of the cannon lives are lost. A life can be lost in two ways, first the cannon is destroyed by an invader beam, second a row of invaders reaches the bottom of the LCD screen. The player will receive a point on the score when an invader is destroyed. When both rows of invaders are destroyed, another wave of invaders are spawned on the screen, and this will happen forever until the cannon is destroyed.

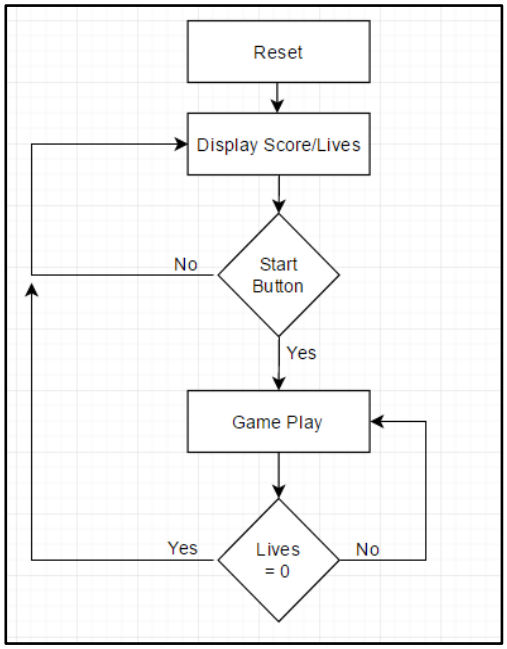

Figure 2-Gameplay Flowchart

4.0-Design Details

4.1 Hardware Design

The Game Space Invaders is implemented using a Silicon Labs development kit C8051F020 microcontroller, and a daughter board containing all peripherals needed which include: a speaker, a bar 8-switch dip switch, two push buttons, two 50kΩ potentiometers, and a LCD screen. This design uses the following: the C8051F020 microcontroller, a 9V power supply, one push button (fire the cannon), one 50kΩ potentiometer (cannon movement), five dip switches (cannon size selection), a LCD screen, and a speaker. This layout can be seen in Figures 3, 4, 5 shown below.

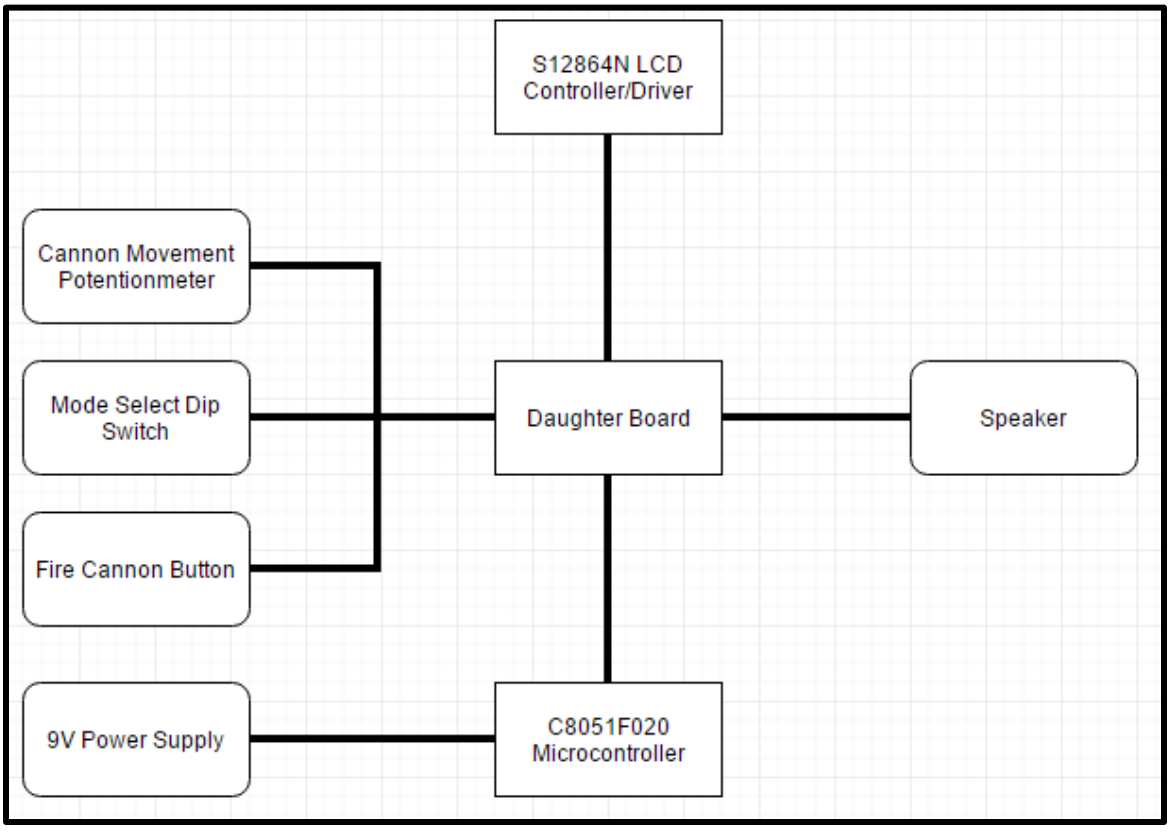

Figure 3-Hardware Block Diagram

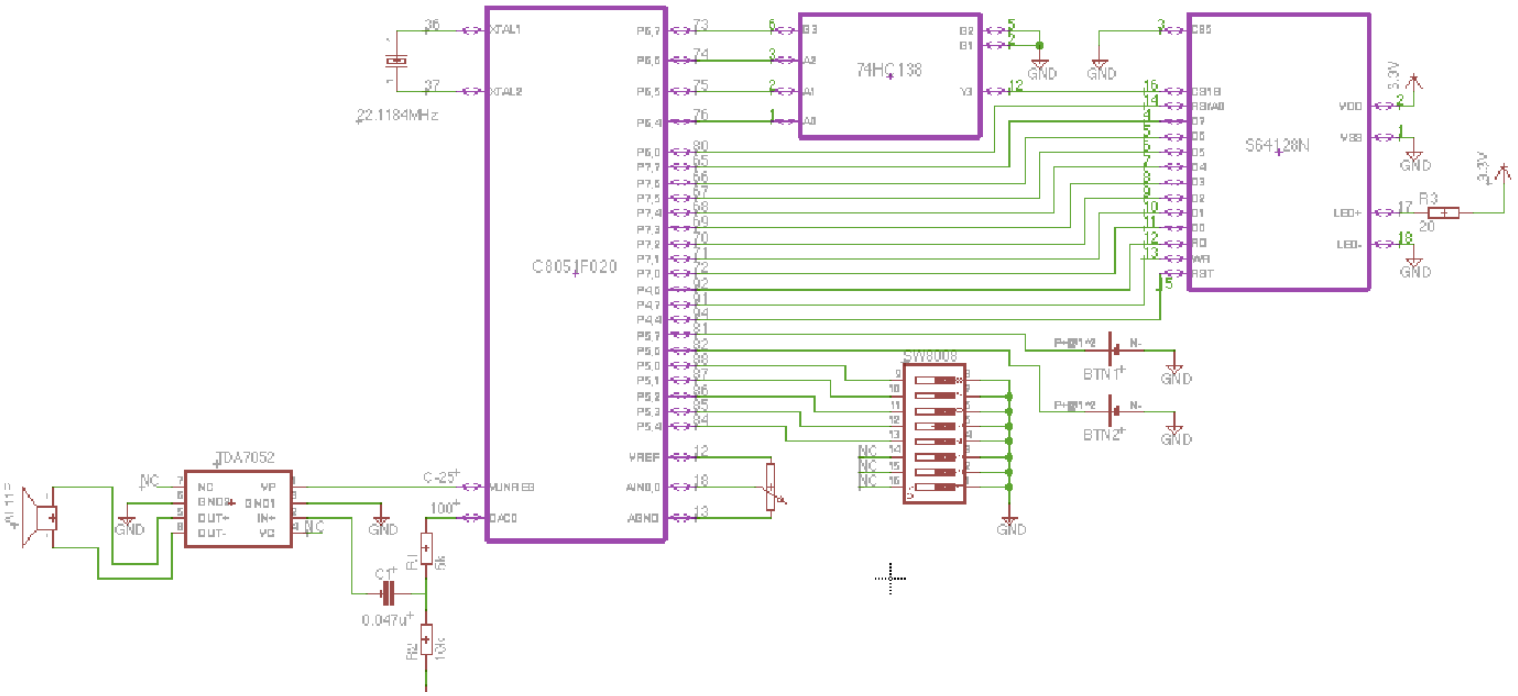

Figure 4- Top Level Schematic

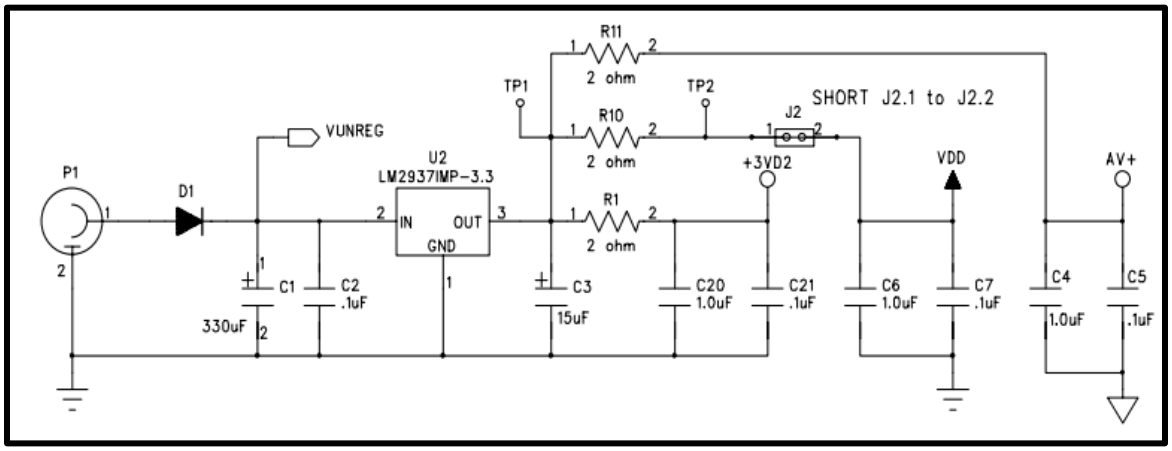

Figure 5- 9V Power Supply

4.2 Software Design

4.2.1- Timer Interrupts Setup

The following table defines Timer Interrupt functions:

Timer 0 ISR	Timer 1 ISR	Timer 2 ISR
Value set to 10ms	Value set to 1ms	Value set to 100us
Checks Buttons Draws strings, ships, cannon, shots, when game is started	Calculates bullet, and beam positions. Invader Movement. Cannon Movement.	Reads ADC0 - Potentiometer Sets DAC0- Speaker

Table 2-Timer Interrupts

4.2.2- Drawing the Cannon, and its movement

The design on the cannon was implemented by indexing its position from a common page. The cannons home position is on page 7. Using a case statement it is determined which cannon design should be used. The pixel design was laid out using excel. (See Figure 6).

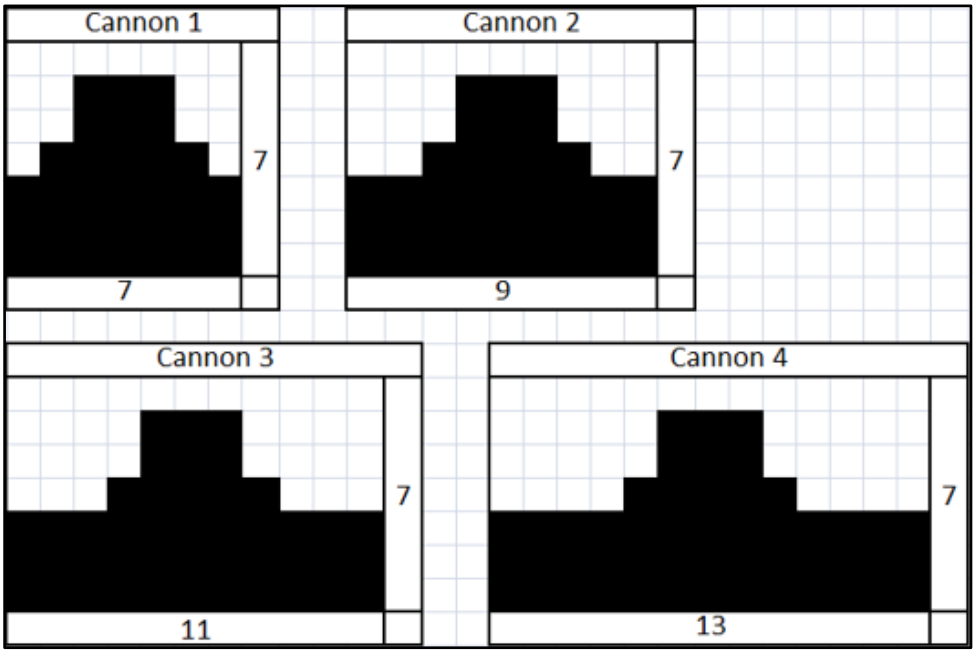

Figure 6-Cannon Pixel Design

```
void draw_cannon(void)
{
    unsigned char i = 0;
    unsigned char cannon_num = switch_current & 0x03; //last two switches

    switch (cannon_num)
    {
        case 3:
            cannon_size = 13;
            break;
        case 2:
            cannon_size = 11;
            break;
        case 1:
            cannon_size = 9;
            break;
        case 0:
        default:
            cannon_size = 7;
            break;
    }
    // draw cannon
    for (i = 0; i < cannon_size; i++)
        screen[896+cannon_position_avg+i] = cannon_gfx[cannon_num][i]; // 896 = page 7
}
```

Figure 7-Cannon Drawing

The code seen in Figure 7, shows how the positon boundary also changes depending on the size of the cannon. This will prevent the cannon from running off of the screen.

The movement of the cannon is determined by the “cannon_position_avg” value, and this value is gathered from the ADC0 or the potentiometer. The value is gathered from ADC0 and it is then sampled and scaled to be in the range from 0 – 127 which is the horizontal boundary for the LCD screen. The ADC0 is sampled 32 times, this ensures there is no glitching or lag occurring in the cannons movement. The size of the cannon is also taken into account, for each different cannon size, a different movement boundary must be set.

```
// if we have enough values then calculate the current average
if (cannon_position_index == 0)
{
    position_sum = 0;
    for (i = 0; i < 32; i++)
        position_sum += cannon_position[i]; // sum the adc samples
    cannon_position_avg = position_sum/32; // calculate the average
    cannon_position_avg = cannon_position_avg/32; // scale to 0-127
    if (cannon_position_avg > (127 - cannon_size)) // make sure we can fit on screen
        cannon_position_avg = 127 - cannon_size;
}
```

Figure 8-ADC0 Reading

4.2.3- Drawing Invaders and their movement

The invader pixel design is as showing in Figure 9.

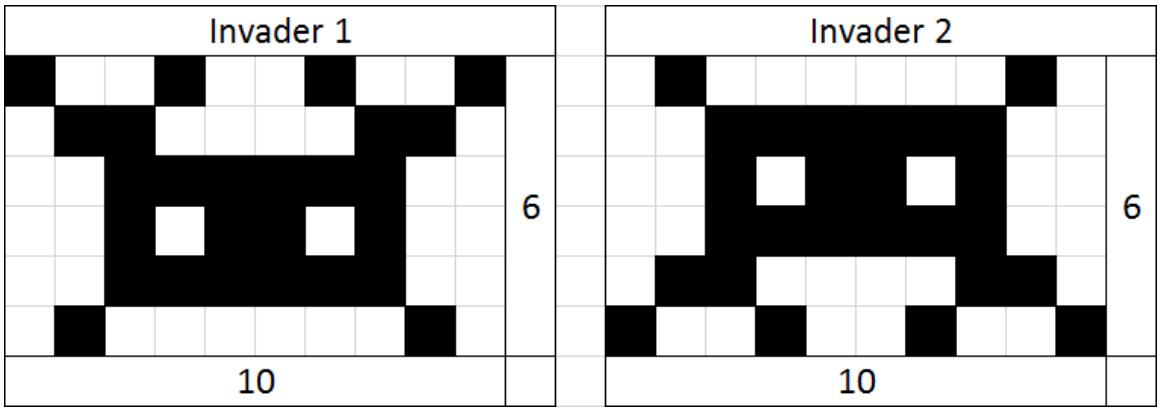

Figure 9-Invader Pixel Design

Each invader was drawn and given an X, and Y coordinate from the LCD. The ships were than given boundary conditions, which prevented the invaders from being drawn off of the screen. Initially two rows of invaders are drawn. The width of each invader is 10 pixels, and this does not change like the cannon could. Initializing the cannons can be seen in Figure 10.

```
for (i = 0; i < 2; i++)
{
    // page*128 + x puts us at reference coords in screen[], i gives ship row num
    cursor = (((y/8)+i)*128) + x;
    for (j = 0; j < 8; j++)
    {
        // don't try to draw outside ship boundaries or draw nonexistent ships
        if ((cursor < 128) || (cursor > 895) || !(ship_alive[j][i]))
        {
            cursor += 13; // 13 = width of ship + spacing
            continue;
        }
        // draw the current frame of the ship
        for (k = 0; k < 10; k++) // 10 = width of ship
        {
            screen[cursor] = ship_gfx[ship_frame_num][k];
            cursor++;
        }
        // add in spacing between ships
        for (k = 0; k < 3; k++) // 3 = ship spacing
        {
            screen[cursor] = 0x00;
            cursor++;
        }
    }
}
```

Figure 10-Drawing Invaders

The real difficult task was if a column of invaders were destroyed than the rest of the invaders need to continue to the edge of the screen, and not stop where the previous invaders had been. This was done by setting limits to know many columns of invaders have been destroyed. We than move through the index of X and Y coordinates of the ships until the living column is found, once the living column reaches the right or left hand limits, the ships than reverse direction using a "ship_direction" variable. This makes managing direction and edge limits much easier to handle. This can be seen in Figure 11.

```
// set right limit based on how many right columns are gone
for (i = 7; i >= 0; i--)
{
    if (!(ship_alive[i][0]) && !(ship_alive[i][1]))
    {
        ship_x_coord_limits[1] += 13; // make sure we move all the way to the right edge
    }
    else
    {
        break; // we found a rightmost ship so stop checking
    }
}

// set left limit based on how many left columns are gone
for (i = 0; i < 8; i++)
{
    if (!(ship_alive[i][0]) && !(ship_alive[i][1]))
    {
        ship_x_coord_limits[0] -= 13; // make sure we move all the way to the left edge
    }
    else
    {
        break; // we found a leftmost ship so stop checking
    }
}
```

```
// move ships in the current direction one pixel
// also check if we're at the edge, if so, drop down and reverse direction
if (ship_direction == 0) // moving right
{
    ship_ref_coords[0] += 1;
    if (ship_ref_coords[0] >= ship_x_coord_limits[1]) // check right limit
    {
        ship_ref_coords[0] = ship_x_coord_limits[1]; // reset x-coord to limit
        ship_ref_coords[1] += 8; // drop down one row
        ship_direction = 1; // reverse direction
    }
}
else // moving left
{
    ship_ref_coords[0] -= 1;
    if (ship_ref_coords[0] <= ship_x_coord_limits[0]) // check left limit
    {
        ship_ref_coords[0] = ship_x_coord_limits[0]; // reset x-coord to limit
        ship_ref_coords[1] += 8; // drop down one row
        ship_direction = 0; // reverse direction
    }
}
```

Figure 11-Invader Movement Horizontal

There is also a lower boundary that had to be accounted for. This was done in a similar manner, the living ships on the most bottom row were identified, and then a limit was imposed so the invaders couldn't go below the bottom of the screen. In this section of code, we also simply added the extra function that causes the

```
// see if we have any ships in the bottom row for lower limit calculation
for (i = 0; i < 8; i++)
    if (ship_alive[i][1])
        ship_in_bottom_row = 1;
// make sure we don't drop below bottom of screen
if (ship_in_bottom_row)
{
    if (ship_ref_coords[1] > 48) // ship(s) in bottom row
        ship_ref_coords[1] = 48;
}
else // no ships in bottom row
{
    if (ship_ref_coords[1] > 56)
        ship_ref_coords[1] = 56;
}

// switch ship animation frames each time we move
ship_frame_num++;
ship_frame_num &= 0x01; // we use this as an index so restrict values
```

Figure 12-Invader Movement Vertical

invaders to change between invader 1, and invader 2 (see Figure 9) each time they moved. (See Figure 12).

4.2.4- Making bullets/beams move and collision

In order to see bullets/beams trailing pixels are enabled, and each shot is given X, and Y values. The shots are moved one pixel at a time, and are given limits so cannon bullets don't pass the top limit, and invader beams don't pass the bottom limit. A random number is generated, and passed through to select a ship to fire from, it is also determined if the selected ship to fire is dead or alive, because only living ships can fire beams. (See Figures 13 & 14 below)

```
// move the cannon shots one pixel
for (i = 0; i < 4; i++)
{
    if (cannon_shot_coords[i][0] != -1) // only update active shots
    {
        cannon_shot_coords[i][1] -= 1;  // move up one pixel
        if (cannon_shot_coords[i][1] < 8)
        {
            cannon_shot_coords[i][0] = -1; // clear shot if we went past top boundary
            cannon_shot_coords[i][1] = -1;
        }
    }
}

// move the ship shots one pixel
for (i = 0; i < 8; i++)
{
    if (ship_shot_coords[i][0] != -1) // only update active shots
    {
        ship_shot_coords[i][1] += 1;  // move down one pixel
        if (ship_shot_coords[i][1] > 63) // see if shot went offscreen
        {
            ship_shot_coords[i][0] = -1; // clear shot if we went past top boundary
            ship_shot_coords[i][1] = -1;
        }
    }
}
```

```
// decide when and where to place random ship shot
rand_num = rand() % ship_update_time; // generate a random number based on current speed (difficulty)
if (rand_num == 0)
{
    rand_num = (rand() % 16); // pick a ship to fire from
    k = 0;
    i = (rand_num % 8); // 0-7 (column)
    j = (rand_num / 8); // 0-1 (row)
    if(ship_alive[i][j])
    {
        // place shot coords in first free slot
        for (k = 0; k < 8; k++)
        {
            if (ship_shot_coords[k][0] == -1) // ignore unused shot slots
            {
                // set x/y coords of ship shot
                ship_shot_coords[k][0] = i*13 + ship_ref_coords[0] + 5; // center shot on ship (ship width /2 = 5)
                ship_shot_coords[k][1] = j*8 + ship_ref_coords[1] + 6; // top/bottom row
                break; // stored shot, so stop checking for free slots
            }
        }
    }
}
```

Figure 13-Bullet/Beam Movement

```
// check if cannon shot(s) hit ship(s)
for (i = 0; i < 4; i++)
{
    if (cannon_shot_coords[i][0] == -1) // if -1 then unused shot slot
        continue;
    x = cannon_shot_coords[i][0];
    y = cannon_shot_coords[i][1];
    // check if y-coord of shot is in top row of ships
    if ((y >= ship_ref_coords[1]) && (y <= ship_ref_coords[1] + 7))
    {
        // check x-coord against alive ships
        for (j = 0; j < 8; j++)
        {
            if ((ship_alive[j][0]) && (x >= (ship_ref_coords[0] + j*13)) && (x <= (ship_ref_coords[0] + j*13 + 9)))
            {
                // we hit a ship, remove it and the shot that hit it
                ship_alive[j][0] = 0;
                cannon_shot_coords[i][0] = -1;
                cannon_shot_coords[i][1] = -1;
                score++;
                if (ship_update_time > 1) // see if we can speed up
                    ship_update_time--; // if so, speed up remaining ships
                // play ship hit sound
                sound_to_play = 1;
            }
        }
    }
```

```
    // check if y-coord of shot is in bottom row of ships
    else if ((y >= ship_ref_coords[1] + 8) && (y <= ship_ref_coords[1] + 15))
    {
        // check x-coord against alive ships
        for (j = 0; j < 8; j++)
        {
            if ((ship_alive[j][1]) && (x >= (ship_ref_coords[0] + j*13)) && (x <= (ship_ref_coords[0] + j*13 + 9)))
            {
                // we hit a ship, remove it and the shot that hit it
                ship_alive[j][1] = 0;
                cannon_shot_coords[i][0] = -1;
                cannon_shot_coords[i][1] = -1;
                score++;
                if (ship_update_time > 1) // see if we can speed up
                    ship_update_time--; // if so, speed up remaining ships
                // play ship hit sound
                sound_to_play = 1;
            }
        }
    }
}
```

Figure 14- Collision Detection Invader

To check in a cannon bullet has hit a ship the X, and Y values of the cannons bullet are compared against the X, and Y values of a living ship. If contact is made, the invader is than removed, and the bullet is removed, and invader movement speed is reassessed. We separately check the X and Y values for the top, and the bottom rows. It makes keeping track of what has been hit easier this way. (See Figure 15 below)

Next we must assess if an invader beam has made contact with the cannon. The X, and Y coordinates of the beam are compared against the X, and Y coordinates of the cannon. This value is dependent on the "cannon_position_avg" and the position information including the size of the cannon. When the cannon is struck, the cannon is removed, the beam is removed, and a life is lost, thus the wave is reset.

```
// check if ship shot(s) hit cannon
for (i = 0; i < 8; i++)
{
    if (ship_shot_coords[i][0] == -1) // if -1 then unused shot slot
        continue;
    x = ship_shot_coords[i][0];
    y = ship_shot_coords[i][1];
    // check if coords of shot is within current cannon area (page 7 and cannon position/size)
    if ((y >= 56) && (x >= cannon_position_avg) && (x <= (cannon_position_avg + cannon_size)))
    {
        // we hit the cannon
        // play cannon hit sound
        sound_to_play = 3;
        // reset wave
        lives--;
        wave_num--;
        initialize_wave();
    }
}
```

Figure 15-Collision Detection Cannon

4.2.5-Sound Conversion using DAC0

Sound was implemented through "Audacity", an audio recording program with sampling capabilities. We designed two different sound waves, one for the cannon firing, another to signify a hit has been made, either for the invaders, or the cannon. (See Figure 16 below) These are 2048 samples long, and have a 10 kHz sample rate, these arrays can be found in the Appendices in Figure X sound. DAC0 is set using these sampled files. The cannon firing sound is Aaron Spilker whistling, and the cannon/invader death sound is Aaron Spilker making a gurgling sound.

```
void set_dac_value(void)
{
    unsigned int sound_data = 0;

    // sounds are only 2048 long, if we went that far we finished playing it
    if (sound_index > 2047)
    {
        sound_index = 0; // reset sound data index
        sound_in_progress = 0; // clear sound in progress
        // make sure DAC output is 0 (silence) if we're done with playback
        DAC0L = 0;
        DAC0H = 0; // write is latched on DAC0H write so write last
    }
    if (sound_in_progress)
    {
        switch (sound_in_progress) // determine which sound we're playing
        {
            case 1:
                sound_data = ship_death_snd[sound_index];
                break;
            case 2:
                sound_data = cannon_fire_snd[sound_index];
                break;
            case 3:
                sound_data = cannon_death_snd[sound_index];
                break;
            default: // unknown sound requested
                sound_data = 0x000;
                break;
        }
        // put the sound data into the DAC registers
        DAC0L = sound_data;
        DAC0H = sound_data >> 8; // write is latched on DAC0H write so write last
        sound_index++; // next loop will play next part of sound
    }
    else if (sound_to_play) // no sound in progress, see if there's one queued
    {
        sound_in_progress = sound_to_play; // start playback of queued sound
        sound_to_play = 0; // clear queued sound
    }
}
```

Figure 16-Sound Conversion

5.0–Testing

5.1 Testing the Power Supply

The C8051F020 microcontroller was powered using a 9V power supply. This requirement is detrimental to the operation of the circuitry. Using a multimeter we probed the supply's output and recorded 9V output power. We than used an amp-meter and recorded a current of 1.7A. This test satisfies requirement number 1.

5.2 Testing the Potentiometer

The potentiometer is used to controller cannon movement in space invaders. We first tested that the cannon would move as follows: when the potentiometer was turned completely counter-clockwise the cannon would be positioned at the most left position on the LCD display, and when turned completely clockwise the cannon would be positioned at the most right position on the LCD display.

The results during game play were as follows: When the potentiometer was turned completely counter-clockwise the cannon was positioned at the most left position on the LCD display, and its values were at the highest point of its 0-50kΩ range. When the potentiometer was turned completely clockwise, the cannon was positioned at the most right position on the LCD display, and its values were at the lowest point of its 0-50kΩ range. Moving between the highest and lowest values caused the cannon to move from the most left position on the screen to the most right position on the screen. This satisfies requirement number 4.

5.3 Testing the Buttons

According to requirement number 3 in section 3.1, the game shall include 3 buttons: a reset button, a fire button, and a start button.

5.3.1 Reset Button

The reset button is to cause the microcontroller to reset which causes the game to restart pending the start button is pressed. First the power was applied, and the start button was pressed, beginning the game. The reset button was than pressed, and the screen was cleared showing only the start menu, and game play did not begin until the start button was pressed. Second the power was applied, but the start button was not pressed, instead the reset button was pressed and the screen was cleared showing only the startup menu, and gameplay again did not start until the start button was pressed. This satisfies requirement number 3, and number 9.

5.3.2 Fire Button

The fire button is to cause 1 bullet to leave the cannon each time it is pressed. The system was powered up, and the start button was pressed, so gameplay began. The fire button was than pressed five times in a row, the LCD displayed four bullets on the screen. The cannon was then moved from left to right pressing the fire button, and bullets successfully left the cannon each time. This satisfies requirement number 3, and part of requirement number 13.

5.3.3 Start Button

Upon reset or power up the display is at the start menu. If the start button is pressed than the gameplay will begin. If the start button is pressed during gameplay, nothing happens. This satisfies requirement number 3, and number 8.

5.4 Testing the LCD Screen

To test the 64x128 LCD first power up / reset the system. The start menu will be displayed, press the start button. The four digit score, starting number of lives (3), the cannon, and the two rows of invaders are displayed moving to the right. When the invaders reach the right side they move forward 8 pixels, and start moving to the left. Move the cannon to one side and fire five times, you will see that only four cannon bullets are displayed at one time. To test that the game is capable of displaying 8 invader beams, and 4 cannon bullets you will need to survive past the 3rd wave of invaders. When a new wave is cleared, the following wave of invaders are appeared closer to the cannon. If an invader is hit by a bullet the invader is destroyed, also if the cannon is hit by an invader beam a life is lost, and this is displayed in the top right corner. If all lives are lost, than the game over screen is displayed. This satisfies requirements 2, 6, 7, 10, 11, 12, 13, 14, 15, 17.

5.5 Testing the Dip Switches

To set the game mode/ cannon size the dip switches must be set before the start button is pressed and the game begins. (See Table 1) To test this start gameplay, and change the dip switch values, the cannon size was not changed. Next adjust the dip switches while at the start menu, when gameplay is started the cannon size is changed. Reset the game and adjust again before pressing start button, the cannon size is correctly changed. This satisfies requirement number 5.

5.6 Testing the Speaker

While at the start menu, and the game over menu there will be an audible sound notifying the user to make a decision. During gameplay a sound is made every time a bullet is fired from the cannon. There is also a sound made when the cannon is hit from an invader beam, signifying a life lost, or when an invader is hit from a bullet, destroying the invader. This satisfies requirement number 16.

6.0-Conclusion

In conclusion this design is too large a task to simply start writing code. We learned early on and spent numerous hours simply discussing the layout of our design, and what challenges we might approach. This design performs, and functions fully to each requirement, and also has many features beyond the design. (I.E invaders changing shape when they move). One feature this design could use, is a sound controller using the second potentiometer. This design was so successful because of the hours of learning about our desired design, and the routes we would take to get there. Early on in the design process we discussed the timers, and which functions each timer would achieve. We also discussed the timing values needed to achieve those tasks. We are very pleased with the results of this design.

7.0–Appendices

Sound.h

```
extern code unsigned int ship_death_snd[];

extern code unsigned int cannon_fire_snd[];

extern code unsigned int cannon_death_snd[];
```