Space Invaders

Design Documentation

Erik Stromberg and Brian Cluff

4/22/13

Table of Contents

1 Introduction

This document explains the design for Space Invaders. In Space Invaders the objective is to destroy incoming aliens. The player is a small spaceship located at the bottom of the screen. The player must fire missiles at the incoming aliens while avoiding the enemy alien missiles that are fired back. If the player gets hit too many times, the player loses. If the player successfully destroys all the incoming alien ships, then the player is victorious.

2 Scope

This document describes the design for Space Invaders. It contains the requirements, dependencies and theory of operation. Schematics, diagrams, and code are used to demonstrate the architecture described. A list of protocol used to test functionality of the Space Invaders are explained. The mechanical description of the Space Invaders is not included.

3 Design Overview

3.1 Requirements

The given requirements for the Space Invaders are the following:

1. The game shall be tested on the STM32F103ZE Waveshare board and shall use the following components: LCD Display, Left/Right/Fire buttons, DAC, DIP switches and an external speaker.
2. The player shall only be able to have one missile on the screen at a given time.
3. There shall be no more than 4 alien missiles on the screen at a given time.
4. A random alien shall fire the missile.
5. There shall be varying levels of difficulty. This shall be implemented by allowing the player to determine the number of aliens and the speed at which they advance toward the player.
6. Sound shall be implemented in the game. The DAC will output to the external speaker when one of the following events occur:
    - a The player fires a missile.
    - b An enemy alien fires a missile.
    - c The player is hit by an alien missile.
7. The high score shall be saved and displayed when the player wins or loses the game.

3.2 Dependencies

The dependencies of the Space Invaders are:

- 5.0 Volt DC Power Supply (for Microcontroller and external speaker power)
- 3.3 Volt DC Power Supply (can be drawn from Microcontroller or external source)
- 8 MHz Crystal Oscillator

3.3 Theory of operation

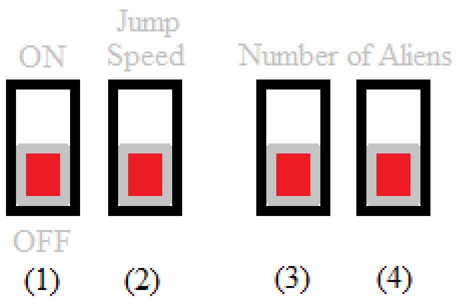
On power or reset, the inputs from the DIP switches are read in the microcontroller to set the number of rows of aliens (between two and five) and as well the speed at which the aliens jump towards a player. During gameplay the aliens shift right across the screen until the furthest right alien collides with the side of the screen. At which point the aliens advance toward the player (referred to as a jump, and the rate at which they move is the jump speed). Then the aliens shift left across the screen until they bump the left side of screen, at which point they jump and then go back to shifting right. While they are shifting they fire missiles at the player in attempt to kill the player.

The player's objective is to kill all the aliens. The player is armed with a cannon, and can fire one missile at a time. If he is able to destroy all the aliens without taking too much damage (the player can take five hits) then the player wins. The player must adroitly maneuver his ship using the move left and move right buttons and shoot missiles with his/her fire button until all the aliens are destroyed.

4 Design Details

4.1 Hardware Design

Space Invaders is designed to run on the ARM Cortex M3 STM32103ZE microcontroller. The output and game is displayed on XPT2046 LCD screen. This game utilizes three buttons, two to move the player left or right, and one for the player to fire a missile. When the player wins or loses, the game may be reset by pressing the fire button again. A 4-position DIP switch is used to determine the player's difficulty. See Figure 1 for a pinout of the DIP switch hardware. The two right switches determine the amount of aliens the player must destroy. The third to right switch sets the jump speed for the aliens. If the jump speed switch is set to a one, then the aliens jump twice as far towards the player then if the jump speed is set to zero. See Figure 2 for the hardware layout and Appendix A for a detailed schematic.



(1) This switch is reserved for future implementations of Space Invaders.

(2) On - The Jump Speed is 4 pixels per jump.
Off -The Jump Speed is 2 pixels per jump.

(3) & (4) - Control the number of rows of aliens between 2 rows (both switches off) to 5 rows (both switches on).

Figure 1: Bits of 4-position DIP switch

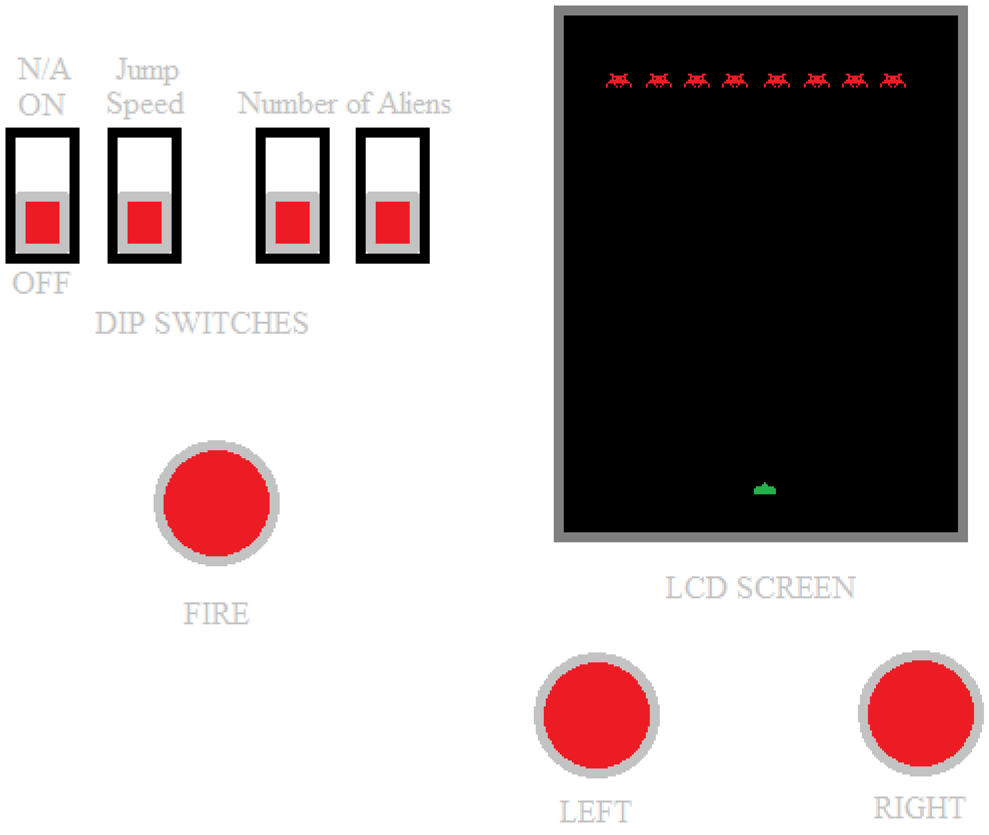


Figure 2: Hardware Design Layout

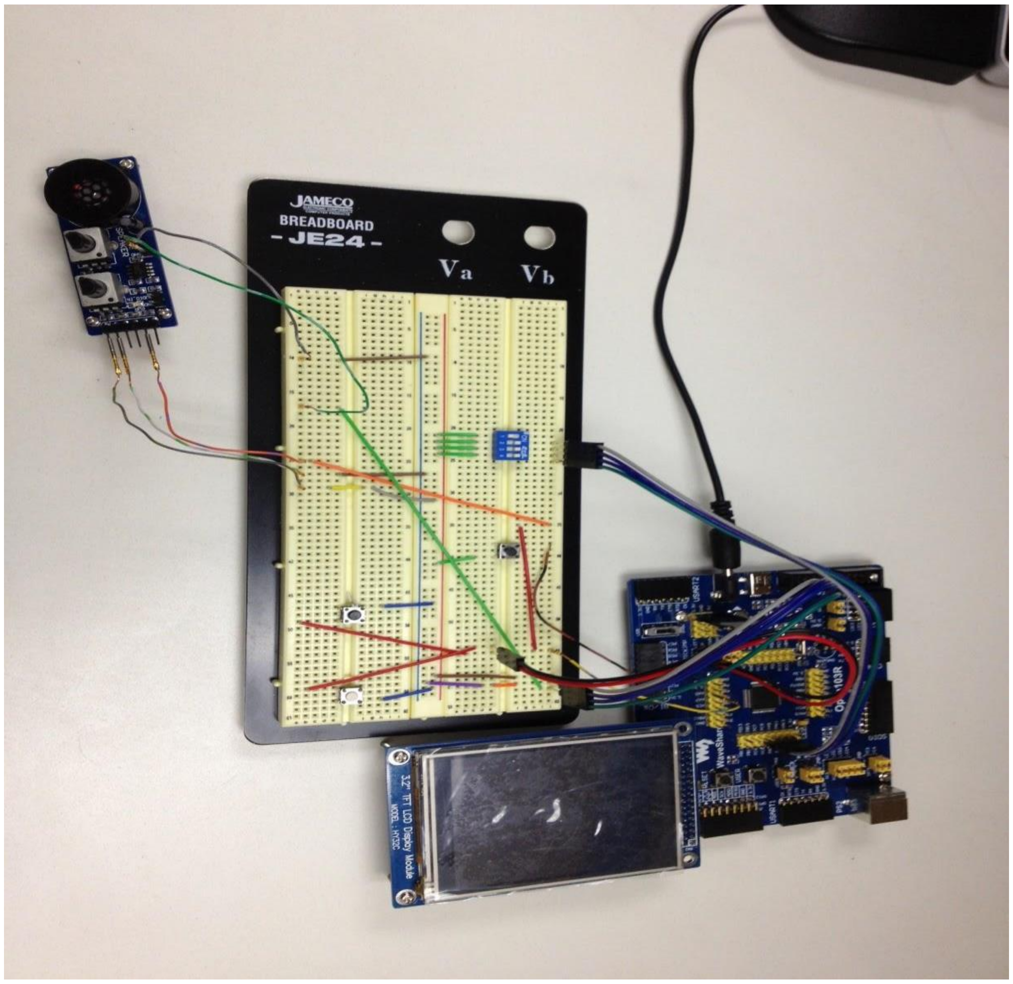


Figure 3: Prototype Development

4.2 Software Design

The software design of Space Invaders is divided into three major components. The first component is the Setup Phase, which is triggered by a Power On or Reset. During this phase variable values are initialized to the proper starting point and sprites are drawn as according to the DIP switch settings. It is during this phase that each object (Alien, Player, Barrier, etc) is created and is given the necessary attributes (location, lives, etc). Once the Setup Phase is complete, the Gameplay Loop is initiated. It is during this loop that interaction between the objects occur. Here the player may move and fire missiles. The Aliens advance toward the player and try to hit him/her with their own missiles. Finally logic to check for collisions is implemented. A check is made to see if the player won or lost. If not the Gameplay Loop repeats itself. When a player has either won or lost the game transitions to the EndGame Phase. During this phase the high score is compared with the player's final score to see which is larger. If the player has the new high score it is saved in flash memory and is displayed to the screen. The game then waits for the missile button to be pressed to reset the game and transition back to the Setup Phase. This flow is outlined in Figure 4 below.

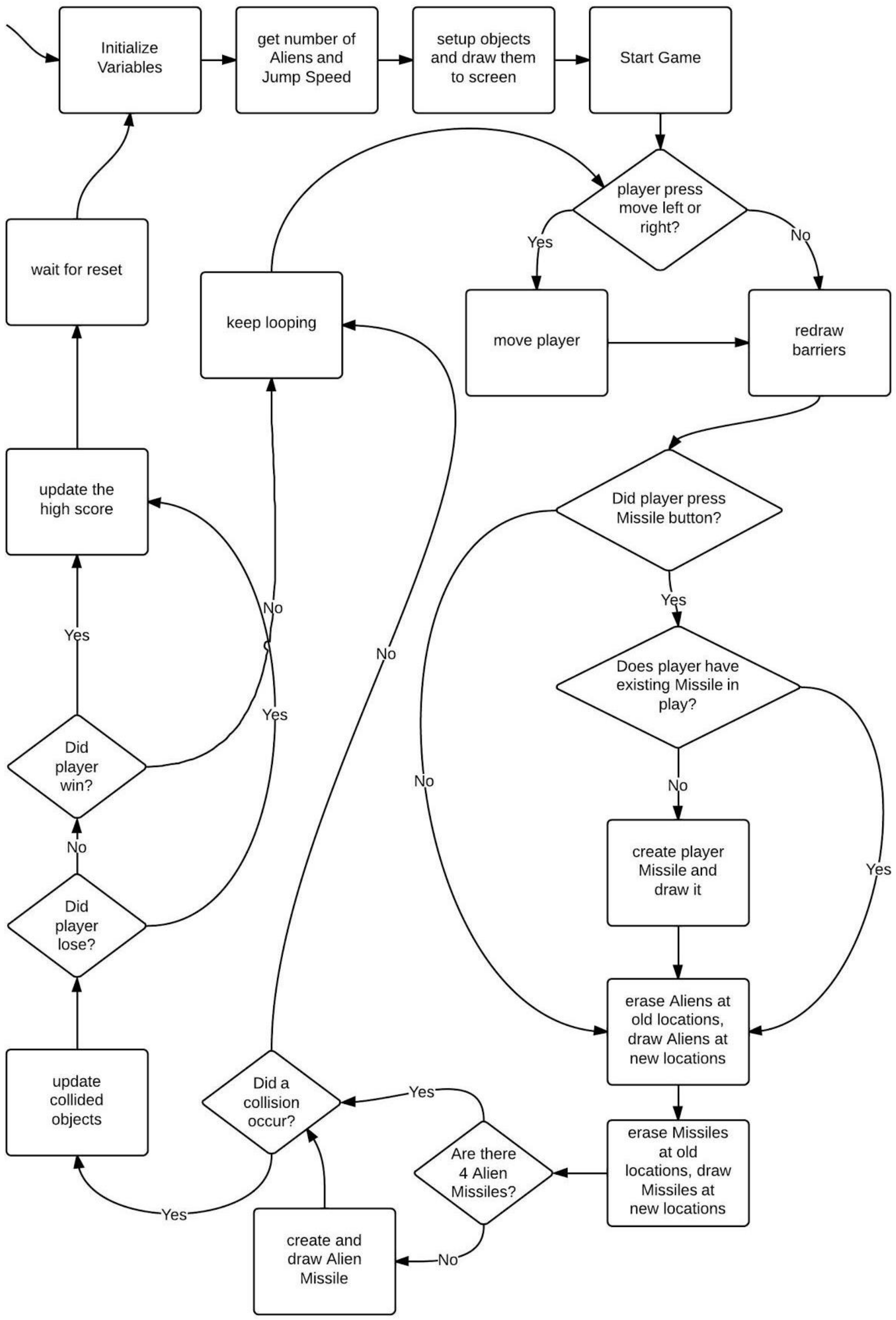

Initialize Variables
get number of Aliens and Jump Speed
setup objects and draw them to screen
Start Game
player press move left or right?
Yes
No
move player
redraw barriers
Did player press Missile button?
Yes
No
Does player have existing Missile in play?
No
Yes
create player Missile and draw it
erase Aliens at old locations, draw Aliens at new locations
erase Missiles at old locations, draw Missiles at new locations
Are there 4 Alien Missiles?
Yes
No
create and draw Alien Missile
Did a collision occur?
Yes
No
update collided objects
Did player lose?
No
Yes
Did player win?
Yes
No
update the high score
wait for reset
keep looping

Figure 4: Flowchart of Gameplay and Software

4.2.1 Power or Reset State

This state is the first state the program enters and the return state once the program is finished. If it is the first time the the microcontroller is being powered on, the register values for the GPIO ports B and C are configured and control registers are placed into the appropriate register addresses. In doing so, this will initialize the appropriate registers and memory locations to be able to have the program accomplish the set tasks as required. The pins being used from GPIO ports B and C are shown in Table 1 below.

GPIO A	0, 1, 2, 3, 4, 5, 6, 7, 8, 10
GPIO B	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
GPIO C	0, 1, 2, 3, 6, 7

Table 1: GPIO Pin Selection

Table 2 below describes which pins are being used for which purpose.

Device / Component	**Port**	**Pin Setup**
Player Move Left Button	GPIO A Port 0	Input
Player Move Right Button	GPIO A Port 1	Input
DAC	GPIO A Port 5	Output
Player DIP 0 (# of Aliens)	GPIO A Port 8	Input
Player DIP 1 (# of Aliens)	GPIO A Port 10	Input
LCD_D0	GPIO B Port 0	Output
LCD_D1	GPIO B Port 1	Output
LCD_D2	GPIO B Port 2	Output
LCD_D3	GPIO B Port 3	Output
LCD_D4	GPIO B Port 4	Output
LCD_D5	GPIO B Port 5	Output
LCD_D6	GPIO B Port 6	Output
LCD_D7	GPIO B Port 7	Output
LCD_D8	GPIO B Port 8	Output
LCD_D9	GPIO B Port 9	Output

LCD_D10	GPIO B Port 10	Output
LCD_D11	GPIO B Port 11	Output
LCD_D12	GPIO B Port 12	Output
LCD_D13	GPIO B Port 13	Output
LCD_D14	GPIO B Port 14	Output
LCD_D15	GPIO B Port 15	Output
LCD_Reset	GPIO C Port 0	Output
LCD_WR	GPIO C Port 1	Output
LCD_RD	GPIO C Port 2	Output
Player Fire Button	GPIO C Port 3	Input
LCD_CS	GPIO C Port 6	Output
LCD_RS	GPIO C Port 7	Output
Player DIP 2 (Jump Speed)	GPIO C Port 8	Input

Table 2: Pin Selection and Function

This game utilizes the 16-bit timer found at Timer 2's address. This timer is initialized to count down from a set value until it reaches zero, then reset the count value and start all over again. In this state, the timer is initialized and also started. The value initialized with this timer is 0xFFFF. This is important because it is used when choosing the random alien from which the missile fires. Since 0xFFFF much larger than 50, we use this timer to generate a random Alien to fire a missile when needed. This count down value is polled in further states in order to generate a random time value. Every time it expires the Gameplay Loop can repeat. If the Gameplay Loop executes once before the timer is finished counting down, it must wait until the timer expires before repeating its execution. This allows the game to never become too fast for the player to respond.

The code snippet shown below describes the setup and where the hardware is mapped to on the Waveshare board.

```
//Setup LCD
LCD_Initialization();

        //initialize variables
        update = false;
        count = 0;
        increment = 0;
        erasecounter = 0;
        numAliens = 0;
```

```
	m1.enable = 0;
	missileEraseCounter = 0;
  alienMissileCounter = 0;
  for (i = 0; i < 4; i++)
  {   missileArray[i].enable = 0;
  }
  for (i = 0; i < 50; i++)
  {   alienArray[i].enable = 0 ;
  }

	// get number of Aliens from difficulty
	// set in run time.  This was just used to debug
	//origAliensNum = 20;

	dacIndex = 0;

	//Configure timer
	RCC->APB1ENR |= 0x20000007;   //timer 2, timer 3, timer 4 enable

	TIM2->ARR = 0xFFFF;                   // 2ms / (1 / 8Mhz) => .002 / 0.000000125 = 16000 =>
0x3E80 - 1 = 3E7F
	TIM2->CNT = 0xFFFF;
	TIM2->CR1 = 0x10;                     // direction down and enable and one-pulse

	//Enable GPIO
	RCC->APB2ENR |= 0x3D;                         // enables AFIO and clocks on D

	// Setup External Interrupts
  // player movement on PA0 and PA1
	// player fire on PC3
	AFIO->EXTICR[0x0] = 0x2000;   // Sets External Interrupts to Pins
  EXTI->RTSR = 0xB;         // set buttons to rising edge

	NVIC->ISER[0] = 0x2C0;    // turn on EXTI interrupts for pins 0, 1, 3
	EXTI->IMR = 0xB;          // enables interrupt requests from pins 0, 1, 3

	GPIOA->CRL &= 0xFFFFFF00;
	GPIOA->CRL |= 0x00000088; // configure pins A0, A1 to input

	GPIOC->CRL &= 0xFFFF0FFF;
	GPIOC->CRL |= 0x00008000;     // configure pin C3 to input

	GPIOA->CRH &= 0xFFF0F0F0;     //
	GPIOA->CRH |= 0x00080808; // configure DIP switches

	GPIOC->CRH &= 0xFFFFFFF0;
	GPIOC->CRH |= 0x00000008; // PA = Number of Aliens, PC = Speed

	//Enable timer interrupt
	NVIC->ISER[0] |= 0x70040000; //enable TIM2, TIM4 interrupts
	TIM2->DIER |= 0x41; //trigger interupt enable & update interrupt enable

	//Setup DAC
	//Dac Out 2 is mapped to PA5
	//Configure timer
	RCC->APB1ENR |= 0x20000007;   //timer 2, timer 3, timer 4 enable
	TIM4->CR1 = 0x10;                     // direction down and enable and one-pulse
```

```
        soundSpeed = 0x1111;
        TIM4->CNT = soundSpeed;
        TIM4->ARR = soundSpeed;

        //Enable timer interrupt
        NVIC->ISER[0] |= 0x70040000; //enable TIM4 interrupts
        TIM4->DIER |= 0x41; //trigger interrupt enable & update interrupt enable

        //clear the screen
        LCD_Clear( Black );
```

This code is referenced in more detail in Appendix C and can be used to setup and initialize the appropriate registers in order for this program to function correctly.

Since this is the return function once the game has finished, there are a few variables that need to be reset. These deal with creating the new objects, resetting the player's lives and reading in the difficulty settings. Once these are reset, the game continues in the Creation State.

4.2.2 Creation State

This state creates all the objects needed to play Space Invaders. To complete this task some variables that were assigned in the Power or Reset State are used, namely the number of Aliens and the speed at which they move.

The Creation State assigns key characteristics that are parts of each object. The code snippet below shows the different objects and their composition.

```
//data structures
struct alien{
        unsigned short x, y;
        unsigned char enable, direction, type, nextmove;
};

struct player{
   unsigned short x, y;
   unsigned char lives, enable;
                unsigned int score;
};

struct missile{
                unsigned short x, y;
                unsigned char enable, direction, posture, type;
};

struct barrierpiece{
      unsigned short x, y;
      unsigned short lives;
      unsigned char enable;
};

struct barrier{
      unsigned short x, y;
      struct barrierpiece pieces[24];
      unsigned char totalEnable;
      //pieces 0, 5, 19-22 must be black
};
```

To create the Aliens, a predefined array is used. A for loop iteratively steps through the array where the direction, enable, type, location (x and y), and animation are all set. Also, a variable numAliens is incremented every time an Alien is created. By holding the number of Aliens alive at a given time it is easy to see if the player won the game by checking if numAliens is zero.

To create the Player, he/she is initialized to the bottom center of the screen and given 5 lives and a current score of 0. The score changes value every time an alien is shot.

To create the Barriers is a less straightforward than the other objects. This is due to the fact that the Barriers are really a grouping of Barrier Pieces that each have their own properties. Essentially the location, enabling, and lives of each piece are set within a set of nested for loops. The outer loop iterates through the Barriers and the inner loop steps through each Barrier Piece that makes up a barrier.

Each barrier piece is given 3 lives. The color of the barrier piece is indicative of the number of lives remaining on that piece.   Once the barrier piece is red if it is hit again it is destroyed.

4.2.3 Update States

As was aforementioned, the Gameplay Loop waits for timer 2 to expire. When it has expired, the update functions are called to update each object in the game. All the objects on the screen except the Barriers have 2 animations that are toggled each time the update occurs. This makes the the motion of the object look more fluid.

Player Update

First the player move is checked. The player move left and move right buttons have interrupts so that when pressed the player can move immediately. However, once the button is pressed it is polled until it is released so that the player can hold the button down to continuously move across the screen.

Whenever the player is either hit or moves the player is redrawn. When hit the player loses 1 life and changes color. When the player moves the color does not change but rather the location of the player is changed and the player is redrawn at the new location of the player.

Barrier Update

Each Barrier Piece is redrawn according to the number of lives it has remaining. If it has 3 lives, it is drawn green. If it has 2 lives, it is drawn yellow. If it has 1 life it is drawn red. If it has no lives remaining it is not drawn.

Alien Update

Afterward the Aliens are updated. The Aliens are updated in rows. This way the Alien rows move together but independently from other rows. The Aliens move horizontally across the screen at a rate of 2 pixels per update until they hit the side wall. When this bounds checking occurs it updates each Alien within the row to move horizontally in the opposite direction after executing a jump (an advance toward the player set as Jump Speed).

Missile Update

First the Player's missile is updated. It moves at a rate of 8 pixels upward per update. If the missile fails to hit an Alien and reaches the top of the screen a check is made to disable and erase the missile.

Next the Aliens' missiles are updated. It does this by stepping through a 4 element array of Alien missiles (missileArray[4] in the code) and moving each one by 8 pixels per update.

If there are less than 4 Alien missiles on the screen then a new missile is created. To find which alien should fire the missile a value is read in from timer 2. Since this is a value between 0 and 65535 it is changed to a number between 0 and the number of aliens by the modulo operation.

Score Update

The score at the top of the screen is updated with the current player score. Every time an Alien is killed, the Player's score is increased depending on which row the Alien resides. In this state, the score is updated with the current score located in p1's struct.

4.2.4 Check Collisions State

This state uses a brute-force method to check and determine if collisions have occurred. The Alien missile array found at missileArray[4] is looped through one at a time. There are two checks that occur. The first is if the X coordinate (up and down direction) is between 260 and 290 pixels, the missile is in range of one of the four barriers potentially. Each barrier is looped through to see if the missile is in range of their respective Y coordinate (left and right direction). If it is determined that the missile is in range of one of the barriers, each individual piece of the barrier must be checked to determine which piece it will come into contact with. Each barrier contains 24 pieces, each 4 pixels x 4 pixels in size. Once the piece is located, the state determines if the piece is enabled. If it is, then one life is subtracted from the pieces life. If it is not, then the next missile is checked.

The second check is if the X coordinate is larger than 290 pixels. If the missile is in this range, it has the chance to hit the Player. The Y coordinates of the Player is retrieved and compared to the missile's Y coordinates. If it is determined that the missile is hitting the Player, a life is subtracted from the Player. Once the life is subtracted, a check is run on the player's life count. If the count is zero, the Player Lose state is triggered and the game is ended. If the count is greater than zero, the next missile is then checked to see whether it falls in one of these two checks.

The final portion of this state is to check if the Player's missile has hit an alien or one of the barriers. The X coordinate of the lowest row of Aliens is determined and used as the lower bound to the check. If the Player's missile is in the X range necessary for a collision with an Alien, this state jumps into checking each individual Alien. Every Alien's X and Y coordinates are checked to see if the missile is going to hit that Alien. If it is within an Alien's range and the Alien is enabled, the Alien is disabled and determined "dead". The Player's score is then updated with the value for that Alien. If the missile is within an Alien's boundary and the Alien is disabled then the state moves on.

4.2.5 Check for Win State

This state checks to see if the Player has won the game. This only occurs if the Player's lives are greater than zero and the number of active Aliens is zero. If the Player has won, the Win state is triggered.

4.2.6 Player Win / Lose State

The Win and Lose states are essentially the same. The only difference lies in how each state is triggered. In these states, the current high score stored in code space is retrieved and compared to the Player's score from that game. If the score is higher than the stored score, then the score is written to code space in place of the one previously there. If it is not higher, then the high score previously stored in code space is kept as the high score. After this is done, the high score is displayed on the blackened screen.

5 Testing

Design testing was implemented in several forms. Code demonstrations were used in demonstrating the requirements as outlined in the project proposal.

5.1 One Player Missile and Four Alien Missiles at a Time

Following the design of the original Arcade Space Invaders, the Player only has the ability to fire one missile at a time. Once this missile fires, the Player must wait until it hits a barrier, hits an Alien, or hits the edge of the screen. A missile is fired when the “Fire” button is pressed, which causes an interrupt in order to activate the missile. The code below outlines how even if the “Fire” button is pressed while a Player missile is active, it will not fire a second.

```c
//missile button
void EXTI3_IRQHandler()
{
        //enable missile sound
        if (m1.enable == 0)
        {       soundSpeed = 0x1111;
                TIM4->CNT = soundSpeed;
                TIM4->ARR = soundSpeed;
                dacIndex = 0;
        }
        EXTI->PR |= 0x4;                        // acknowledge interrupt
        //enable player (only used after a win or lose)
        p1.enable = 1;
        //check if already on screen
        if (m1.enable)
                return;
        m1.enable = 1;                                  // turn on missile flag
        m1.direction = 1;
        m1.posture = 0;
        m1.type = 0;
        m1.x = p1.x - 8;                        // sets coordinates of missile
        m1.y = p1.y - 6;
}
```

On line 15 of the above code, a check is made to see if m1 (the Player’s missile) is active. If it is, then the function returns. If the player’s missile is not active, it plays a sound indicating that a missile has been fired, activate the missile, and initialize the X and Y coordinates based on where the Player happens to be at the time of the interrupt.

The Aliens have the ability to fire up to four missiles simultaneously. This is limited in code as shown below:

```c
void createAlienMissile(void)
{
  int k;
  for (k = 0; k < 4; k++)
  {
    if (missileArray[k].enable == 0)
    {
        randomVal = (TIM2->CNT) % origAliensNum;
        soundSpeed = 0x3333;
        TIM4->CNT = soundSpeed;
        TIM4->ARR = soundSpeed;
        dacIndex = 0;
        while(alienArray[randomVal].enable == 0)
        randomVal = (TIM2->CNT) % origAliensNum;
        missileArray[k].x = alienArray[randomVal].x + 25;
        missileArray[k].y = alienArray[randomVal].y - 8;
        missileArray[k].type = (randomVal % 2) + 1; // creates type
```

```
                missileArray[k].posture = 0; // animation
                missileArray[k].direction =0 ; // down
                missileArray[k].enable = 1;// enabled
                return;
            }
        }
    }
```

Every time the running loop runs, it calls the above function. In this function, it loops through the Alien missile array (missileArray[4]) and determines if any of the missiles have been disabled during the checkCollisions() function. If a missile has been disabled, the above function randomly picks one Alien using Timer 2's count. If that Alien is disabled, it polls the value of Timer 2 again and picks another Alien. Once an Alien has been selected, the missile's coordinates and properties are set and the missile is enabled. Figure 5 shows 4 active Alien missiles and 1 Player missile.

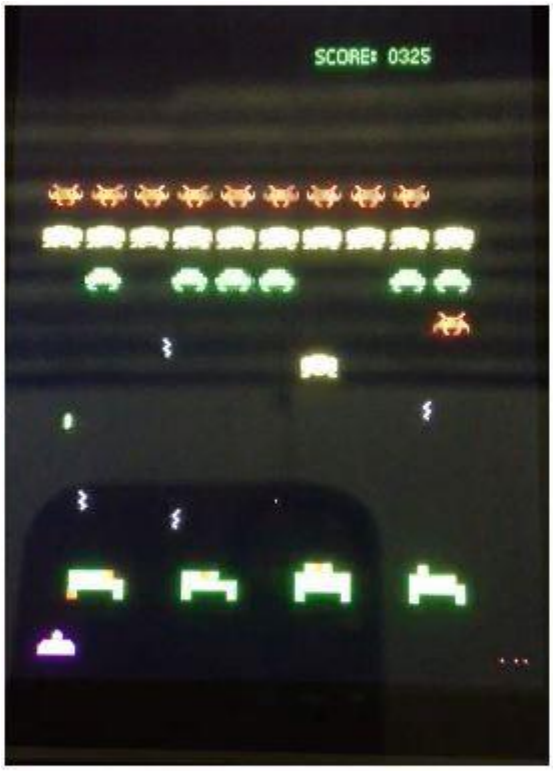


Figure 5: Missiles

5.2 Varying Levels of Difficulty

This is implemented in several different ways, all done using the DIP switches. One way to vary the difficulty is to adjust the number of Aliens that will invade during gameplay. As was aforementioned, the number of aliens can be adjusted between 20 and 50 in increments of 10. This two rightmost DIP switches add numbers of rows equal to the binary representation of the switches (i.e. both switches off implies no added rows, while both on implies 3 added rows to the original 2 rows). Having more Aliens is harder because of the added time it takes to destroy them. If the player fires errant missiles he/she will be unable to kill the Aliens before they are already upon him.

The other way in which the difficulty can be adjusted is through setting the Jump Speed DIP switch on. This makes the Aliens move at a vertical rate of 4 pixels per jump instead of 2. Therefore the Aliens arrive at the domain of the player in half the time then if the Jump Speed were switched off. The code snippets below show where the DIP switches vary the difficulty.

```
//find difficulty
        getval = GPIOA->IDR;
        getval = getval & 0x0500;
        getval >>= 8;

        //converts DIP switches to a difficulty
        //difficulty configures the number of alien rows (between 2 and 5)
        switch (getval)
        {
        case 1: difficulty = 2; break;
        case 4: difficulty = 1; break;
        case 5:     difficulty = 3; break;
        default: difficulty = 0;
        }

        //gets speed
        getval = GPIOC->IDR;
        getval = getval & 0x100;                //get pin 8
        getval >>= 8;

        //jump speed is the number of pixels at which the aliens jump
        //every time they hit a side wall.  Having it set to 1 increases difficulty.
        if (getval == 1)
                jumpSpeed = 1;
        else
                jumpSpeed = 0;

        //sets the amount of aliens
        origAliensNum = 20 + (10 * difficulty);
        if (origAliensNum > 50)
                origAliensNum = 50;
```

5.3 Sounds

Sound is implemented using the Open103R ADC/DAC development module. As was mentioned before, the DAC pin are mapped to Port A pin 5’s pin. In addition to this DAC line, which connects to the AIN line on the module, a 5V, 3.3V, and ground line are necessary to power the external module. Figure 6 shows the DAC module setup that was implemented using the STM32F103ZE.

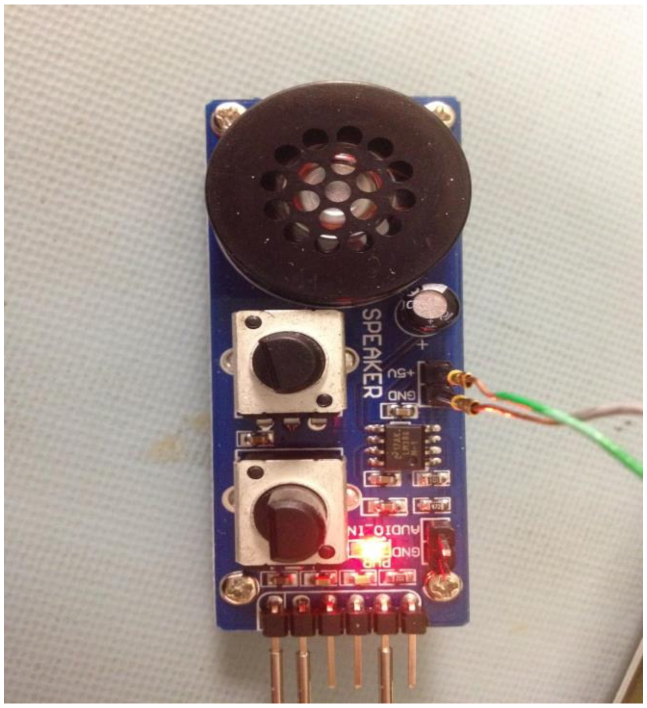


Figure 6: ADC / DAC Module

```
void TIM4_IRQHandler()
{
            TIM4->SR &= 0xFFFE;         // acknowledge interrupt

            DAC->DHR12R2 = sin_40[dacIndex % 40];
            if (dacIndex <120)
            {       dacIndex++;
                    DAC->CR = 0x00010000;       // enable DAC
            }

}
```

As outlined in the above interrupt handler, Timer 4 was used to control the DAC output. Using the 40-part sine array pre-defined, Timer 4's countdown value is used to control the pitch of the output. This module is setup so that it outputs three tones. There is a tone for an Alien missile, one for the Player missile, and another for when either an Alien or Player missile contacts a barrier or their target.

5.4 Saving the Highscore

Saving the highscore into code space is accomplished through unlocking the flash memory section of the microcontroller. This is done by passing key codes that disable the ready-only lock on the code space. After it has been unlocked, a bit is set to erase the memory at a certain location set by the FLASH->AR register. Another bit is set to commence erasing the memory. After this is done a separate programming bit is set so that at the same location the highscore can be written.

Afterwards whenever the score needs to be compared (which occurs when a player wins or loses) the value is simply read in from the memory location at which the highscore is written (in the code we wrote to the location 0x0807F800). Below verifies the described process.

```
void writeHighScore(short score)
{
        int i;
        int key1 = 0x45670123;
        int key2 = 0xCDEF89AB;

        while ((FLASH->SR & 0x1) == 0x1);               //wait for busy flag

        while((FLASH->CR & 0x80) == 0x80)
        {
                FLASH->KEYR = key1;
                FLASH->KEYR = key2;
                //flashKEYR = key1;
                //for (i = 0; i < 255; i++);              //delay
                //flashKEYR = key2;
                //for (i = 0; i < 255; i++);              //delay
        }

        FLASH->CR |= 0x2;                       // set PER bit

        while ((FLASH->SR & 0x1) == 0x1);               //wait for write

        FLASH->AR = 0x0807F800;

        FLASH->CR |= 0x40;

        while ((FLASH->SR & 0x1) == 0x1);               //wait for write

        FLASH->CR &= 0xFFFFFFFD;

        FLASH->CR |= 0x1;                       // set PG bit
        for (i = 0; i < 255; i++);              //delay

        vFlashWrite(0x0807F800, score);

        while ((FLASH->SR & 0x1) == 0x1);               //wait for write

        FLASH->CR &= 0xFFFFFFFE;

        i = scoreLocation;              //verify contents

        i = i;
}
```

6 Conclusion

The design described in this document provides a functional prototype of the Space Invaders as setup on the ARM Cortex M3 STM32103ZE microcontroller. A total of six timing and functionality tests occurred to satisfy all 7 requirements. The tests all passed, showing that this design of the Space Invaders works as expected on the prototype setup.

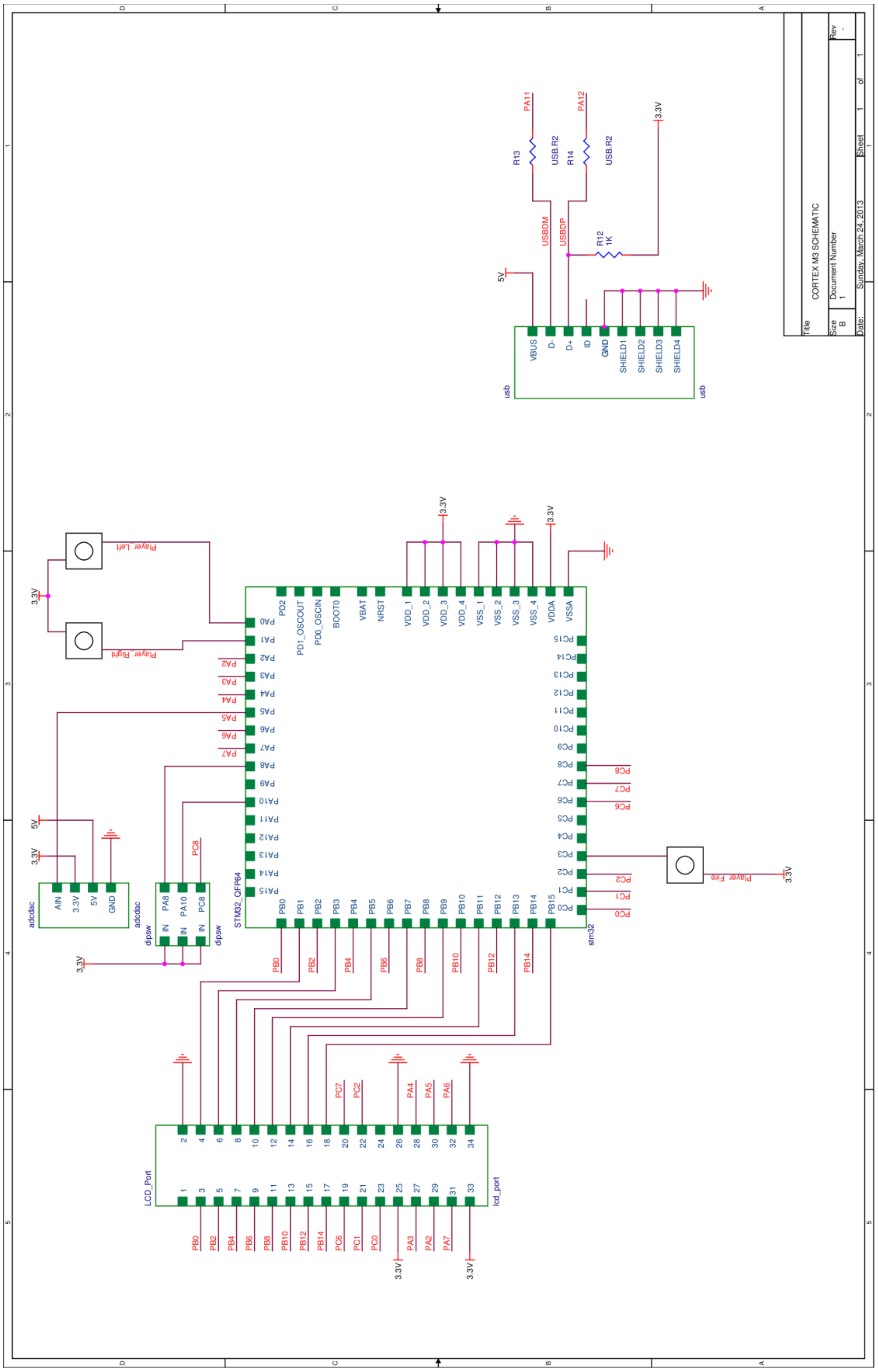
This prototype fulfills the basic requirements to implement the Space Invaders game. There are some potential optimization opportunities that are available to improve the overall gameplay. There are several interrupts that dictate the precedence of the possible events. In addition, there are also some potential things that could be simplified in the basic operation of the game. Regardless of the situation, every state is called during the loop. It isn't until the state is called that conditions are checked, such as Player missile activation or having Aliens disabled during collision detection. It could speed the game up and provide more stability if update functions are called only under certain conditions, moving everything into the main program loop versus branched loops.

There are certain aspects of this game that were simplified greatly by developing a spreadsheet that was used to produce color mapping tables. Rather than produce complicated methods of writing out the Alien models to the screen, the Aliens were converted to color tables and when writing out to the screen, the tables are just ran through from start to finish. This avoids the need to have a complicated pixel-skip algorithm.

One of the biggest difficulties that was faced on this project was understanding how to write out the high score to a page file in flash memory. This process is outlined in Section 5.4 above. After understanding how the flash memory worked, it opens up some additional possibilities, such as storing an array of high scores, player initials, and more. This would make this game more like the original Arcade game.

7 Appendix A

Shown below is the detailed schematic of the Space Invaders setup on the ARM Cortex M3 STM32103ZE microcontroller.



CORTEX M3 SCHEMATIC
Title
Size B
Document Number 1
Rev -
Date: Sunday, March 24, 2013
Sheet 1 of 1
STM32_QFP64
stm32
PD2
PD1_OSCOUT
PD0_OSCIN
BOOT0
VBAT
NRST
VDD_1
VDD_2
VDD_3
VDD_4
VSS_1
VSS_2
VSS_3
VSS_4
VDDA
VSSA
PA0
PA1
PA2
PA3
PA4
PA5
PA6
PA7
PA8
PA9
PA10
PA11
PA12
PA13
PA14
PA15
PB0
PB1
PB2
PB3
PB4
PB5
PB6
PB7
PB8
PB9
PB10
PB11
PB12
PB13
PB14
PB15
PC0
PC1
PC2
PC3
PC4
PC5
PC6
PC7
PC8
PC9
PC10
PC11
PC12
PC13
PC14
PC15
usb
VBUS
D-
D+
ID
GND
SHIELD1
SHIELD2
SHIELD3
SHIELD4
USBDM
USBDP
R12
1K
R13
R14
USB.R2
PA11
PA12
5V
3.3V
Player Left
Player Right
Player Fire
adcdac
AIN
3.3V
5V
GND
dipsw
IN
PA8
PA10
PC8
LCD_Port
lcd_port
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
PB0
PB2
PB4
PB6
PB8
PB10
PB12
PB14
PC6
PC1
PC0
PA3
PA2
PA7
PC7
PC2
PA4
PA5
PA6

8 Appendix B

Below is the code for implementing the Space Invaders game using the STM32F103ZE microcontroller.

```
//main.c

#include "stm32f10x.h"
#include "LCD.h"
#include "stdbool.h"
#include "pixelmaps.c"

#define CS_low_Pin  0x0040  // Pin 6

//global variables
volatile int erasecounter, missileEraseCounter, alienMissileCounter, randomVal, soundSpeed;
volatile bool update;
volatile unsigned char count, increment, numAliens, jumpSpeed;
volatile struct alien alienArray[50];
volatile struct missile missileArray[4];
volatile struct barrier barrierArray[4];
volatile struct player p1;
volatile struct missile m1;
volatile unsigned short highScores;
volatile unsigned int scoreLocation __attribute__((at(0x0807F800)));
volatile unsigned int flashACR __attribute__((at(0x40022000)));
volatile unsigned int flashKEYR __attribute__((at(0x40022004)));
volatile unsigned int flashOPTKEYR __attribute__((at(0x40022008)));
volatile unsigned int flashSR __attribute__((at(0x4002200C)));
volatile unsigned int flashCR __attribute__((at(0x40022010)));
volatile unsigned int flashAR __attribute__((at(0x40022014)));
volatile unsigned int flashOBR __attribute__((at(0x4002201C)));
volatile unsigned int flashWRPR __attribute__((at(0x40022020)));
unsigned short color_map[] = {White, Black, Blue, Red, Magenta, Yellow, Green};
volatile int origAliensNum, dacIndex;
//volatile int key1 = 0x45670123;
//volatile int key2 = 0xCDEF89AB;
volatile int rdprt = 0x00A5;

//table used for sound
unsigned short sin_40[] =
{512,593,671,745,813,875,927,969,999,1018,1024,1018,999,969,927,875,813,745,671,593,512,432,354,280,212,150,98,56,26,7,0,
7,26,56,98,150,212,280,354,432};


//data structures
struct alien{
	unsigned short x, y;
	unsigned char enable, direction, type, nextmove;
};

struct player{
    unsigned short x, y;
    unsigned char lives, enable;
		unsigned int score;
};

struct missile{
		unsigned short x, y;
		unsigned char enable, direction, posture, type;
};

struct barrierpiece{
        unsigned short x, y;
        unsigned short lives;
        unsigned char enable;
};
```

```
struct barrier{
        unsigned short x, y;
        struct barrierpiece pieces[24];
        unsigned char totalEnable;
        //pieces 0, 5, 19-22 must be black
};

//prototypes
void writeHighScore(short);
int retrieveHighScore(void);
void updateAliens(int);
void drawAlien(struct alien, char);
void drawMissiles(struct missile, char);
void updateMissiles(void);
void checkCollisions(void);
void createAliens(void);
void playerWin(void);
void playerLose(void);
void createPlayer(void);
void drawPlayer(char);
void drawScore(void);
void createBarriers(void);
void updateBarriers(void);
void drawBarriers(struct barrier, char);
void drawBarrierPiece(struct barrierpiece, char);
void createAlienMissile(void);
void updateHighScore(void);


void SystemInit(void)
{
    int i;

           LCD_Initialization();

           //initialize variables
           update = false;
           count = 0;
           increment = 0;
           erasecounter = 0;
           numAliens = 0;
           m1.enable = 0;
           missileEraseCounter = 0;
  alienMissileCounter = 0;
  for (i = 0; i < 4; i++)
  {   missileArray[i].enable = 0;
  }
  for (i = 0; i < 50; i++)
  {   alienArray[i].enable = 0 ;
  }

           // get number of Aliens from difficulty
           origAliensNum = 20;
           //writeHighScore(0); //check if writes a 0

           dacIndex = 0;

           //Configure timer
           RCC->APB1ENR |= 0x20000007;               //timer 2, timer 3, timer 4 enable

           TIM2->ARR = 0xFFFF;                                 // 2ms / (1 / 8Mhz) => .002 / 0.000000125 = 16000 => 0x3E80 - 1 =
3E7F
           TIM2->CNT = 0xFFFF;
           TIM2->CR1 = 0x10;                         // direction down and enable and one-pulse

           //Enable GPIO
           RCC->APB2ENR |= 0x3D;                                         // enables AFIO and clocks on D

           // Setup External Interrupts
  // player movement on PA0 and PA1
           // player fire on PC3
           AFIO->EXTICR[0x0] = 0x2000;   // Sets External Interrupts to Pins
  EXTI->RTSR = 0xB;        // set buttons to rising edge

           NVIC->ISER[0] = 0x2C0;    // turn on EXTI interrupts for pins 0, 1, 3
           EXTI->IMR = 0xB;          // enables interrupt requests from pins 0, 1, 3

           GPIOA->CRL &= 0xFFFFFF00;
           GPIOA->CRL |= 0x00000088; // configure pins A0, A1 to input

           GPIOC->CRL &= 0xFFFF0FFF;
           GPIOC->CRL |= 0x00008000;     // configure pin C3 to input

           GPIOA->CRH &= 0xFFF0F0F0;  //
           GPIOA->CRH |= 0x00080808; // configure DIP switches

           GPIOC->CRH &= 0xFFFFFFF0;
           GPIOC->CRH |= 0x00000008; // PA = Number of Aliens, PC = Speed

           //Enable timer interrupt
           NVIC->ISER[0] |= 0x70040000; //enable TIM2, TIM3, TIM4 and ADC interupts
           TIM2->DIER |= 0x41; //trigger interupt enable & update interrupt enable


           //Setup DAC
           //Dac Out 2 is mapped to PA5
           //Configure timer
           RCC->APB1ENR |= 0x20000007;             //timer 2, timer 3, timer 4 enable
           TIM4->CR1 = 0x10;                       // direction down and enable and one-pulse
           soundSpeed = 0x1111;
           TIM4->CNT = soundSpeed;
           TIM4->ARR = soundSpeed;

           //Enable timer interrupt
           NVIC->ISER[0] |= 0x70040000; //enable TIM4 interrupts
           TIM4->DIER |= 0x41; //trigger interupt enable & update interrupt enable

           //clear the screen
           LCD_Clear( Black );



}

void reset(void)
{
           int i;
           volatile short getval, difficulty;

           //initialize variables
           update = false;
           count = 0;
           increment = 0;
           erasecounter = 0;
           numAliens = 0;
           m1.enable = 0;
           missileEraseCounter = 0;
  alienMissileCounter = 0;
  for (i = 0; i < 4; i++)
  {   missileArray[i].enable = 0;
  }
  for (i = 0; i < 50; i++)
  {   alienArray[i].enable = 0 ;
  }

           // get number of Aliens from difficulty
           origAliensNum = 20;
           //writeHighScore(0); //check if writes a 0
```

```
	dacIndex = 0;

	//clear the screen
	LCD_Clear( Black );

	difficulty = 0;

	//find difficulty
	getval = GPIOA->IDR;
	getval = getval & 0x0500;
	getval >>= 8;

	//converts DIP switches to a difficulty
	//difficulty configures the number of alien rows (between 2 and 5)
	switch (getval)
	{
	case 1: difficulty = 2; break;
	case 4: difficulty = 1; break;
	case 5:		difficulty = 3; break;
	default: difficulty = 0;
	}

	//gets speed
	getval = GPIOC->IDR;
	getval = getval & 0x100;				//get pin 8
	getval >>= 8;

	//jump speed is the number of pixels at which the aliens jump
	//every time they hit a side wall.  Having it set to 1 increases difficulty.
	if (getval == 1)
			jumpSpeed = 1;
	else
			jumpSpeed = 0;

	//sets the amount of aliens
	origAliensNum = 20 + (10 * difficulty);
	if (origAliensNum > 50)
			origAliensNum = 50;

	createAliens();
	createPlayer();
	drawPlayer(0);
	createBarriers();

}

//This timer is used so that the screen is only updated when timer 2 expires (setting update flag true)
//This way the game doesn't move impossibly fast when there are only a few objects to draw on the screen.
void TIM2_IRQHandler()
{
	TIM2->SR &= 0xFFFE;			// acknowledge interrupt
	if (increment == 1)
	{
	    update = true;		// set update flag
	    increment = 0;
	}
	else
	    increment++;
}

void vFlashWrite(int iAdd, short sDta)
{
	    __asm
	{   strh sDta,[iAdd]
	}
}

int main()
{
    int i;
    volatile short getval, difficulty;

    difficulty = 0;

    //find difficulty
    getval = GPIOA->IDR;
    getval = getval & 0x0500;
    getval >>= 8;

    //converts DIP switches to a difficulty
    //difficulty configures the number of alien rows (between 2 and 5)
    switch (getval)
    {
    case 1: difficulty = 2; break;
    case 4: difficulty = 1; break;
    case 5:         difficulty = 3; break;
    default: difficulty = 0;
    }

    //gets speed
    getval = GPIOC->IDR;
    getval = getval & 0x100;                    //get pin 8
    getval >>= 8;

    //jump speed is the number of pixels at which the aliens jump
    //every time they hit a side wall.  Having it set to 1 increases difficulty.
    if (getval == 1)
                jumpSpeed = 1;
    else
                jumpSpeed = 0;

    //sets the amount of aliens
    origAliensNum = 20 + (10 * difficulty);
    if (origAliensNum > 50)
                origAliensNum = 50;

    createAliens();
    createPlayer();
    drawPlayer(0);
    createBarriers();


    while(1)
    {
      TIM2->CR1 |= 1;                           //A way to speed it up and slow it down
      if (update)
      {

                update = false;         //reset update flag
if((GPIOA->IDR & 0x0002)==0x0002)                                   //move player
{   drawPlayer(1);
                                  if (p1.y >= 35)
                                                p1.y -=5;
    drawPlayer(0);
}
if((GPIOA->IDR & 0x0001)==0x0001)
{   drawPlayer(1);
                                  if (p1.y <= 225)
                                                p1.y +=5;
    drawPlayer(0);
}
                updateBarriers();
                for (i = 0; i < origAliensNum / 10; i++)
                        updateAliens(i);
                updateMissiles();
createAlienMissile();
                checkCollisions();
                drawScore();
                if (dacIndex < 120)
```

```
                        {                       TIM4->CR1 |= 1; // enable timer for sound
                        }

                        //check if player won
                        if (numAliens == 0)
                                                playerWin();

                }
            }
}


void writeHighScore(short score)
{
        int i;
        int key1 = 0x45670123;
        int key2 = 0xCDEF89AB;

        while ((FLASH->SR & 0x1) == 0x1);                       //wait for busy flag

        while((FLASH->CR & 0x80) == 0x80)
        {
            FLASH->KEYR = key1;
            FLASH->KEYR = key2;
            //flashKEYR = key1;
            //for (i = 0; i < 255; i++);                //delay
            //flashKEYR = key2;
            //for (i = 0; i < 255; i++);                //delay
        }

        FLASH->CR |= 0x2;                               // set PER bit

        while ((FLASH->SR & 0x1) == 0x1);                       //wait for write

        FLASH->AR = 0x0807F800;

        FLASH->CR |= 0x40;

        while ((FLASH->SR & 0x1) == 0x1);                       //wait for write

        FLASH->CR &= 0xFFFFFFFD;

        FLASH->CR |= 0x1;                               // set PG bit
        for (i = 0; i < 255; i++);              //delay

        vFlashWrite(0x0807F800, score);

        //FLASH->AR = 0x0807F800;
        //FLASH->OBR |= (score<<10);
        //scoreLocation = score;

        while ((FLASH->SR & 0x1) == 0x1);                       //wait for write

        FLASH->CR &= 0xFFFFFFFE;

        i = scoreLocation;                      //verify contents

        i = i;
}

int retrieveHighScore(void)
{       short val;
        val = scoreLocation;
        return val;
}

void createAliens(void)
{
        int i, j;
        j = 2;
```

```
		// creates the aliens
		for(i=0; i<origAliensNum; i++)
		{
		    if (i % 10 == 0)
		    {		j++;
				j = j % 3;
		    }
		    alienArray[i].direction = 0;
		    alienArray[i].enable = 1;
		    alienArray[i].type = j;
		    alienArray[i].x = 40+ 20*(i / 10);
		    alienArray[i].y = 40+(i % 10)*20;
		    alienArray[i].nextmove = 0;
		    numAliens += 1;
		}

	}


	void updateAliens(int currRow)
	{
			int j, z;

			for (z=0; z<21; z++) // this max must be 1 greater than erasecounter
			{

				if (erasecounter == 20)
				{

						for (j = 0; j < 10; j++)
		// erase specific row
						{		if (alienArray[j + (currRow*10)].enable == 1)
											drawAlien(alienArray[j + (currRow*10)], 1);
									if (alienArray[j + (currRow*10)].nextmove)
											alienArray[j + (currRow*10)].y += 2;
									else
											alienArray[j + (currRow*10)].y -= 2;
									alienArray[j + (currRow*10)].direction = (alienArray[j +
	(currRow*10)].direction + 1) % 2; // toggles animations
									if (alienArray[j + (currRow*10)].enable)	// if alien is enabled, then
	draw it back to the screen
				drawAlien(alienArray[j + (currRow*10)], 0);
						}

						if (alienArray[9+ (10*currRow)].y >= 238 || alienArray[0 + (10*currRow)].y <= 18)
			//bounds checking
						{		for (j = 0; j < 10; j++)
								{		if (alienArray[j + (currRow*10)].enable == 1)
											drawAlien(alienArray[j + (currRow*10)], 1);
									alienArray[j + (10*currRow)].x += 2 + (2*jumpSpeed);
									alienArray[j + (10*currRow)].nextmove = (alienArray[j +
	(10*currRow)].nextmove + 1) % 2;
			    if (alienArray[j + (currRow*10)].enable)
				drawAlien(alienArray[j + (currRow*10)], 0);
								}
						}
						erasecounter = 0;
				}
				else
						erasecounter++;
			}

	}

	void drawAlien(struct alien ain, char erase)
	{
		short x, y, i, j, k;
		char type, dir;
```

```
        unsigned char colorMap[160];

        x = ain.x;
        y = ain.y;
        type = ain.type;
        dir = ain.direction;

        k = 0;

        //this copies the correct bit map depending on the alien

        if (type == 0 && dir == 0)                              // alien1 down
        {
            for (i=0; i<160; i++)
                        colorMap[i] = a1down[i];
        }
        else if (type == 0 && dir == 1)      // alien1 up
        {
            for (i=0; i<160; i++)
                        colorMap[i] = a1up[i];
        }
        else if (type == 1 && dir == 0) // alien2 down
        {
            for (i=0; i<160; i++)
                        colorMap[i] = a2down[i];
        }
        else if (type == 1 && dir == 1) // alien2 up
        {
            for (i=0; i<160; i++)
                        colorMap[i] = a2up[i];
        }
        else if (type == 2 && dir == 0) // alien3 down
        {
            for (i=0; i<160; i++)
                        colorMap[i] = a3down[i];
        }
        else                                                                                    // alien3 up
        {
            for (i=0; i<160; i++)
                        colorMap[i] = a3up[i];
        }

        //draws aliens
        for (i=x; i < x+10; i++)
        {
            LCD_SetCursor(i,y);
            GPIOC->BRR  = CS_low_Pin;
    LCD_WriteIndex( 0x0022 );
    for(j=y; j < y+16; j++ )
    {
                        if (erase)
                        {           LCD_WriteData( Black );

                        }
                        else
                        {           LCD_WriteData(color_map[colorMap[k]]);

                        }
                        k++;
    }
    GPIOC->BSRR = CS_low_Pin;
        }

}
void createAlienMissile(void)
{
    int k;
    for (k = 0; k < 4; k++)
    {
        if (missileArray[k].enable == 0)
        {
        randomVal = (TIM2->CNT) % origAliensNum;
                                                                soundSpeed = 0x3333;
                                                                TIM4->CNT = soundSpeed;
                                                                TIM4->ARR = soundSpeed;
                                                                dacIndex = 0;
        while(alienArray[randomVal].enable == 0)
            randomVal = (TIM2->CNT) % origAliensNum;
        missileArray[k].x = alienArray[randomVal].x + 25;
        missileArray[k].y = alienArray[randomVal].y - 8;
        missileArray[k].type = (randomVal % 2) + 1; // creates type
        missileArray[k].posture = 0; // animation
        missileArray[k].direction =0 ; // down
        missileArray[k].enable = 1;// enabled
        return;
        }
    }
}
void updateMissiles(void)
{
        int j,k;

            for(j = 0; j < 10; j++) // has to be 1 + missileEraseCounter max
            {
                                    // update player missile
                                    if (m1.enable)
                                    {


                                                if (missileEraseCounter == 9)
                                                {
                                                                    drawMissiles(m1, 1);  // erase current missile
                                                                    m1.x -= 8;                                          // set new
coordinate for missile

                                                                    m1.posture = (m1.posture + 1) % 2;                  // sets missile to
next animation

                                                                    if (m1.x <= 8)                                      // check if he is off
screen
                                                                    {                   m1.enable = 0;          // disable missile
                                                                                        drawMissiles(m1, 1);
                                                                    }
                                                                    else
                                                                                        drawMissiles(m1, 0);            // draw the missile

                                                                    missileEraseCounter = 0;

                                                }
                                                else
                                                                    missileEraseCounter++;
                                    }
            }



            // update alien missiles
            for (j = 0; j < 5; j++)     //has to be 1 + alienMissileCounter max
        {
        if (alienMissileCounter == 4)
        {
            for (k = 0; k < 4; k++)         //loop through all alien missiles
            {
                if (missileArray[k].enable)
                {

                    drawMissiles(missileArray[k], 1);           // erase current missile
                     missileArray[k].x += 8;                                                // set new coordinate for missile

                     missileArray[k].posture = (missileArray[k].posture + 1) % 2;           // sets missile to next animation
                    if (missileArray[k].x >= 310)                                    // check if he is off screen
                        missileArray[k].enable = 0;     // disable missile



                    if (missileArray[k].enable)
                      drawMissiles(missileArray[k], 0);          //draw missile

                    alienMissileCounter = 0;

                }
              }
            }
            else
              alienMissileCounter++;
          }
          updateBarriers();

      }

      void drawMissiles(struct missile m, char erase)
      {
              int x, y, i, j, k;
              unsigned char mColorMap[40];

              // toggles missile animation
              for (i = 0; i < 40; i++)
              {           if (m.posture == 1)
                          {                 if (m.type == 0)
                                                            mColorMap[i] = playerMisUp[i];
                                            else if (m.type == 1)
                                                            mColorMap[i] = alienMis1Up[i];
                                            else
                                                            mColorMap[i] = alienMis2Up[i];
                          }
                          else
                          {                 if (m.type == 0)
                                                            mColorMap[i] = playerMisDown[i];
                                            else if (m.type == 1)
                                                            mColorMap[i] = alienMis1Down[i];
                                            else
                                                            mColorMap[i] = alienMis2Down[i];
                          }
              }

        //find current location
          x = m.x;
          y = m.y;

                                  k = 0;

        //draw missile
          for (i=x; i < x+8; i++)
          {
              LCD_SetCursor(i,y);
              GPIOC->BRR  = CS_low_Pin;
              LCD_WriteIndex( 0x0022 );
              for(j=y; j < y+5; j++ )
              {
                if (erase)
                {         LCD_WriteData( Black );

                                                                                      }
                else
                { LCD_WriteData(color_map[mColorMap[k]]);

                                                                                      }
                                                                                      k++;
              }
```

```
           GPIOC->BSRR = CS_low_Pin;
       }
}

void checkCollisions(void)
{
  int i, j, k;

           // check alien missiles
           for (i = 0; i < 4; i++)
                                                                                                                  // loop through missiles
           {
                        if ((missileArray[i].x >= 260) && (missileArray[i].x <= 290) && (missileArray[i].enable))          // see if missile is in
range of barrier
                        {
                                    for (j = 0; j < 4; j++)
                                                                                                                  // loop through barriers
                                    {
                                                if((missileArray[i].y >= barrierArray[j].y-3) && (missileArray[i].y <=
barrierArray[j].y+22))                                   // check to see if it is within a barrier
                                                {
                                                            for(k = 0; k < 24; k++)
                                                            {
                                                                        if((missileArray[i].y >=
barrierArray[j].pieces[k].y - 2) && (missileArray[i].y <= barrierArray[j].pieces[k].y +4+2) && (barrierArray[j].pieces[k].lives > 0))
                                                                        {
           missileArray[i].enable = 0;                                          // disable missile

           drawMissiles(missileArray[i], 1);   // erase missile

           barrierArray[j].pieces[k].lives -= 1;
                                                                                    goto SkipToNext;
                                                                        }
                                                            }
                                                }
                                    }
                        }

                        else if((missileArray[i].x > 290) && (missileArray[i].enable))
                        {
                                    if((missileArray[i].y >= p1.y-16) && (missileArray[i].y <= p1.y+2))
           // check to see if it is within player bounds
                                    {
                                                            missileArray[i].enable = 0;
                                    // disable missile
                                                            drawMissiles(missileArray[i], 1);
             // erase missile
                                                            p1.lives -= 1;

                                                            soundSpeed = 0x2222;
                                                            TIM4->CNT = soundSpeed;
                                                            TIM4->ARR = soundSpeed;
                                                            dacIndex = 0;

                                                            drawPlayer(0);
                                                            if (p1.lives == 0)
                                                                        playerLose();
                                                            else
                                                                        goto SkipToNext;
                                    }
                        }
                        SkipToNext: continue;
           }

           // check player missile
           if (m1.enable == 1)
           {
              if ((m1.x >= 260) && (m1.x <= 290))                              // within barrier range
              {
                                 for (j = 0; j < 4; j++)                                                                          // loop
through barriers
                                        {
                                                            if((m1.y >= barrierArray[j].y-3) && (m1.y <= barrierArray[j].y+22))
                                                            {
                                                                                for (i = 24; i > 0; i--)
                                                                                {
                                                                                                    if((m1.y >= barrierArray[j].pieces[i-1].y -
2) && (m1.y <= barrierArray[j].pieces[i-1].y +4+2) && (barrierArray[j].pieces[i-1].lives > 0))
                                                                                                    {                   m1.enable = 0;
                    // disable missile
                                                                                                                        drawMissiles(m1,
1);      // erase missile

      barrierArray[j].pieces[i-1].lives -= 1;

                                                                                                                        return;
                                                                                                    }
                                                                                }
                                                            }

                                        }

                    }

                    for(j = origAliensNum; j > 0; j--)
                    {
                                        if ((m1.x >= alienArray[j-1].x) && (m1.x <= alienArray[j-1].x+10) && (alienArray[j-1].enable == 1))
                                        {
                                                            if((m1.y >= alienArray[j-1].y-16) && (m1.y <= alienArray[j-1].y+2))
                                                            {
                                                                                m1.enable = 0;
                                                                                alienArray[j-1].enable = 0;
                                                                                drawMissiles(m1, 1);
            //erase missile
                                                                                drawAlien(alienArray[j-1], 1);   //erase alien

                                                                                soundSpeed = 0x2222;
                                                                                TIM4->CNT = soundSpeed;
                                                                                TIM4->ARR = soundSpeed;
                                                                                dacIndex = 0;

                                                                                //update player score
                                                                                if (((j-1) / 39) > 0)
      // row 5
                                                                                                    p1.score += 10;
                                                                                else if (((j-1) / 29) > 0)                     // row 4
                                                                                                    p1.score += 15;
                                                                                else if (((j-1) / 19) > 0)                     // row 3
                                                                                                    p1.score += 20;
                                                                                else if (((j-1) / 9) > 0)                       // row 2
                                                                                                    p1.score += 25;
                                                                                else
                                                                                // row 1
                                                                                                    p1.score += 30;

                                                                                return;
                                                            }
                                        }
                    }

      }

}


void playerWin(void)
{
      p1.enable = 0;                    //disable player
      highScores = retrieveHighScore();
      if (p1.score > highScores)
      {
```

```
                        highScores = p1.score;                    //set highscores
                        writeHighScore(p1.score);

                }
                updateHighScore();                                                          //draw to the screen
                while(p1.enable != 1);                              //wait for button press
                //reset
                reset();
}

void playerLose(void)
{
                p1.enable = 0;                      //disable player
                highScores = retrieveHighScore();
                if (p1.score > highScores)
                {
                        highScores = p1.score;                    //set highscores
                        writeHighScore(p1.score);

                }
                updateHighScore();                                                          //draw to screen
                while(p1.enable != 1);
                //reset
                reset();
}

void createPlayer(void)
{
                  p1.enable = 1;                    //enable player
                  p1.x = 300;                                                   //set player location
                  p1.y = 128;
                  p1.lives = 5;                               //set amount of lives
                  p1.score = 0;                               //reset score

}

void drawPlayer(char erase)
{
                int x, y, i, j, k;
    //find current location
        x = p1.x;
        y = p1.y;

                                k = 0;

    //draw player
        for (i=x; i < x+10; i++)
        {
            LCD_SetCursor(i,y);
            GPIOC->BRR  = CS_low_Pin;
            LCD_WriteIndex( 0x0022 );
            for(j=y; j < y+16; j++ )
            {
                if (erase || p1.lives == 0)
                {           LCD_WriteData( Black );

                                                                                    }
                else
                { if(player1[k] == 1)

                LCD_WriteData(color_map[player1[k]]);
                                                                                            else

                LCD_WriteData(color_map[player1[k]-(5-p1.lives)]);   //changes colors when loses lives
                                                                                    }
                                                                                    k++;
            }
            GPIOC->BSRR = CS_low_Pin;
        }
}
```

```
void drawScore(void)
{
        int x, y, i, j, k, m;
        int dig[4];
        unsigned char digit_map[28];
  //set score location
    x = 30;
    y = 100;

                        k = 0;

  //draw score words
    for (i=x; i < x+7; i++)
    {
        LCD_SetCursor(i,y);
        GPIOC->BRR  = CS_low_Pin;
        LCD_WriteIndex( 0x0022 );
        for(j=y; j < y+30; j++ )
        {
          LCD_WriteData(color_map[score[k]]);
                                                        k++;
        }
      GPIOC->BSRR = CS_low_Pin;
    }

        //find individual numbers of score

        dig[0] = p1.score % 10;

        dig[1] = (((p1.score % 100) - dig[0]) / 10);

        dig[2] = (((p1.score % 1000) - dig[1] - dig[0]) / 100);

        dig[3] = (((p1.score % 10000) - dig[2] - dig[1] - dig[0]) / 1000);

//draw digits
        x = 30;
        y = 50;
        for(k = 0; k < 4; k++)
  {
                        switch (dig[k])
                        {
                        case 0:    for (m = 0; m < 28; m++)         //load map for number 0
                                                {                       digit_map[m] = zero[m];
                                                }
                                                break;
                        case 1:    for (m = 0; m < 28; m++)         //load map for number 1
                                                {                       digit_map[m] = one[m];
                                                }
                                                break;
                        case 2:    for (m = 0; m < 28; m++)         //load map for number 2
                                                {                       digit_map[m] = two[m];
                                                }
                                                break;
                        case 3:    for (m = 0; m < 28; m++)         //load map for number 3
                                                {                       digit_map[m] = three[m];
                                                }
                                                break;
                        case 4:    for (m = 0; m < 28; m++)         //load map for number 4
                                                {                       digit_map[m] = four[m];
                                                }
                                                break;
                        case 5:    for (m = 0; m < 28; m++)         //load map for number 5
                                                {                       digit_map[m] = five[m];
                                                }
                                                break;
                        case 6:    for (m = 0; m < 28; m++)         //load map for number 6
                                                {                       digit_map[m] = six[m];
                                                }
```

```
                                                                        break;
                                        case 7:    for (m = 0; m < 28; m++)            //load map for number 7
                                                                        {                       digit_map[m] = seven[m];
                                                                        }
                                                                        break;
                                        case 8:    for (m = 0; m < 28; m++)            //load map for number 8
                                                                        {                       digit_map[m] = eight[m];
                                                                        }
                                                                        break;
                                        case 9:    for (m = 0; m < 28; m++)            //load map for number 9
                                                                        {                       digit_map[m] = nine[m];
                                                                        }
                                                                        break;
                                        }

                                        m = 0;
                                        for (i=x; i < x+7; i++)
        {
                LCD_SetCursor(i,y +(k*5));
                GPIOC->BRR  = CS_low_Pin;
                LCD_WriteIndex( 0x0022 );
                for(j=y; j < y+4; j++ )
                {
                  LCD_WriteData(color_map[digit_map[m]]);
                                                                                        m++;
                }
            GPIOC->BSRR = CS_low_Pin;
        }
                }
}

void createBarriers(void)
{
    int i, j, newRow;
    newRow = 0;
    for (i = 0; i < 4; i++)         //each barrier
    {   barrierArray[i].x = 270;
        barrierArray[i].y = 36 + (i*50);
        barrierArray[i].totalEnable = 1;
        for (j = 0; j < 24; j++)    //each barrier piece
        {   newRow = j / 6;
            if( (j!=0) && (j!=5) && (j!=19) && (j!=20) && (j!=21) && (j!=22))
            {   barrierArray[i].pieces[j].enable = 1;
                barrierArray[i].pieces[j].lives = 3;
                barrierArray[i].pieces[j].x = barrierArray[i].x + (newRow * 4);
                barrierArray[i].pieces[j].y = barrierArray[i].y + ((4*j) % 24);
            }
        }

    }
}

void updateBarriers(void)
{
    int i;
    for(i = 0; i < 4; i++)
        drawBarriers(barrierArray[i], 0);
}
void drawBarriers(struct barrier b, char erase)
{
    int i;
    for (i = 0; i < 24; i++)
    {   if (b.pieces[i].enable)
        {   drawBarrierPiece(b.pieces[i], erase);
        }
    }
}

void drawBarrierPiece(struct barrierpiece bp, char erase)
{
    short x, y, i, j;

            x = bp.x;
            y = bp.y;



            //draws barrier piece -- changes color when hit
            for (i=x; i < x+4; i++)
            {
                LCD_SetCursor(i,y);
                GPIOC->BRR  = CS_low_Pin;
    LCD_WriteIndex( 0x0022 );
    for(j=y; j < y+4; j++ )
    {
                                if ((erase == 1) || (bp.lives==0))
                                {               LCD_WriteData( Black );

                                }
                                else if (bp.lives == 3)
                                {               LCD_WriteData( Green );

                                }
      else if (bp.lives == 2)
      { LCD_WriteData( Yellow );

                                }
      else if (bp.lives == 1)
      { LCD_WriteData( Red );

                                }
    }
    GPIOC->BSRR = CS_low_Pin;
            }
}

// move left
void EXTI0_IRQHandler()
{
    int checkDir;
    checkDir = GPIOA->IDR & 0x0002; // find which button is pressed

    if((GPIOA->IDR & 0x0002) != 0x0002)
    {   EXTI->PR |= 0x1;        // acknowledge interrupt
    }
    //EXTI->PR |= 0x1;                                              // acknowledge interrupt

    drawPlayer(1);              // erase current player
    //
            if ((checkDir == 0x0002) && (p1.y >= 35))
        p1.y = p1.y - 5;
    drawPlayer(0);              // draw player

            return;
}

// move right
void EXTI1_IRQHandler()
{
    int checkDir;
    checkDir = GPIOA->IDR & 0x0001; // find which button is pressed

    if((GPIOA->IDR & 0x0001) != 0x0001)
    {   EXTI->PR |= 0x2;        // acknowledge interrupt
    }
    //EXTI->PR |= 0x2;                                              // acknowledge interrupt

    drawPlayer(1);              //erase player
    if ((checkDir == 0x0001)&& (p1.y <= 225))
        p1.y = p1.y + 5;
    drawPlayer(0);              //draw player

            return;
}

//missile button
void EXTI3_IRQHandler()
{
            //enable missile sound
            if (m1.enable == 0)
            {   soundSpeed = 0x1111;
                TIM4->CNT = soundSpeed;
                TIM4->ARR = soundSpeed;
                dacIndex = 0;
            }

            EXTI->PR |= 0x4;                                    // acknowledge interrupt

            //enable player (only used after a win or lose)
            p1.enable = 1;

            //check if already on screen
            if (m1.enable)
                return;

            m1.enable = 1;                                                  // turn on missile flag
            m1.direction = 1;
            m1.posture = 0;
            m1.type = 0;

            m1.x = p1.x - 8;                                    // sets coordinates of missile
            m1.y = p1.y - 6;

}

void TIM4_IRQHandler()
{
                TIM4->SR &= 0xFFFE;            // acknowledge interrupt

                DAC->DHR12R2 = sin_40[dacIndex % 40];
                if (dacIndex <120)
                {               dacIndex++;
                                DAC->CR = 0x00010000;       // enable DAC
                }

}

void updateHighScore(void)
{
                unsigned char digit_map[28];
                int x, y, i, j, k, m, dig[4];

                LCD_Clear( Black );

                //find individual numbers of score

                dig[0] = highScores % 10;

                dig[1] = (((highScores % 100) - dig[0]) / 10);

                dig[2] = (((highScores % 1000) - dig[1] - dig[0]) / 100);

                dig[3] = (((highScores % 10000) - dig[2] - dig[1] - dig[0]) / 1000);

//draw digits
                x = 150;
                y = 125;
                for(k = 0; k < 4; k++)
    {
                                        switch (dig[k])
                                        {
                                        case 0:    for (m = 0; m < 28; m++)                //load map for number 0
                                                                            {               digit_map[m] = zero[m];
                                                                            }
                                                                            break;
                                        case 1:    for (m = 0; m < 28; m++)                //load map for number 1
                                                                            {               digit_map[m] = one[m];
                                                                            }
                                                                            break;
                                        case 2:    for (m = 0; m < 28; m++)                //load map for number 2
                                                                            {               digit_map[m] = two[m];
                                                                            }
                                                                            break;
                                        case 3:    for (m = 0; m < 28; m++)                //load map for number 3
                                                                            {               digit_map[m] = three[m];
                                                                            }
                                                                            break;
                                        case 4:    for (m = 0; m < 28; m++)                //load map for number 4
                                                                            {               digit_map[m] = four[m];
                                                                            }
                                                                            break;
                                        case 5:    for (m = 0; m < 28; m++)                //load map for number 5
                                                                            {               digit_map[m] = five[m];
                                                                            }
                                                                            break;
                                        case 6:    for (m = 0; m < 28; m++)                //load map for number 6
                                                                            {               digit_map[m] = six[m];
                                                                            }
                                                                            break;
                                        case 7:    for (m = 0; m < 28; m++)                //load map for number 7
                                                                            {               digit_map[m] = seven[m];
                                                                            }
                                                                            break;
                                        case 8:    for (m = 0; m < 28; m++)                //load map for number 8
                                                                            {               digit_map[m] = eight[m];
                                                                            }
                                                                            break;
                                        case 9:    for (m = 0; m < 28; m++)                //load map for number 9
                                                                            {               digit_map[m] = nine[m];
                                                                            }
                                                                            break;

                                        }

                                        m = 0;
                                        for (i=x; i < x+7; i++)
    {
        LCD_SetCursor(i,y +(k*5));
        GPIOC->BRR  = CS_low_Pin;
        LCD_WriteIndex( 0x0022 );
        for(j=y; j < y+4; j++ )
        {
          LCD_WriteData(color_map[digit_map[m]]);
                                                                                            m++;
        }
      GPIOC->BSRR = CS_low_Pin;
    }
        }
}




//
//  lcd.c
//

#include "stm32f10x.h"
#include "lcd.h"

#define WR_low_Pin  0x0002  // Pin 1
```

```
#define RD_low_Pin  0x0004  // Pin 2
#define CS_low_Pin  0x0040  // Pin 6
#define DC_Pin      0x0080  // Pin 7



//  configuration of the LCD port pins
void LCD_Config(void)
{
    unsigned int config_temp;

    RCC->APB2ENR |= 0x1D;       // Enable port A, B, and C

    config_temp  = GPIOA->CRL;  // Pin A.3 for Back light
    config_temp &= ~0x0000F000;
    config_temp |=  0x00003000;
            //changed here
            //No remap (NSS/PA4, SCK/PA5, MISO/PA6, MOSI/PA7)
            //PA4 NSS
            //PA5 SCK                   ALT PUSH-PULL 23 - 20
            //PA6 MISO  ALT PUSH-PULL
            //PA7 MOSI  Input Floating / Input pull-up
                //config_temp &= 0x0000FFFF;               // makes last 4 bytes 0
                //config_temp |= 0xAAA80000;                // Setup for SLAVE config_temp |= 0x8A480000;  Setup for SLAVE 0xA8A80000;
                config_temp  = 0xBBB83344;  // this is config for LCD program
    GPIOA->CRL   = config_temp;

    GPIOB->CRL   = 0x33333333;  // Port B for Data[15:0] pins
    GPIOB->CRH   = 0x33333333;

    config_temp  = GPIOC->CRL;  // PC.0(LCD RST), PC.1(WR), PC.2(RD) , PC.6(CS), PC.7(DC)
    config_temp &= ~0xFF000FFF;
    config_temp |=  0x33000333;
    GPIOC->CRL   = config_temp;
                GPIOC->CRH           = 0x33300000;

                GPIOC->BSRR = CS_low_Pin;
}

void LCD_Initialization(void)
{
    unsigned int config_temp;

    LCD_Config();

    config_temp  = AFIO->MAPR;  // enable SW Disable JTAG
    config_temp &= ~0x07000000;
    config_temp |=  0x02000000;
    AFIO->MAPR   = config_temp;

    GPIOC->BRR  = 0x0001;   // LCD reset
    delay_ms(100);
    GPIOC->BSRR = 0x0001;
    GPIOA->BSRR = 0x0008;   // back light

    LCD_WriteReg(0x0000,0x0001);    delay_ms(50);   /* Enable LCD Oscillator */
    LCD_WriteReg(0x0003,0xA8A4);    delay_ms(50);   // Power control(1)
    LCD_WriteReg(0x000C,0x0000);    delay_ms(50);   // Power control(2)
    LCD_WriteReg(0x000D,0x080C);    delay_ms(50);   // Power control(3)
    LCD_WriteReg(0x000E,0x2B00);    delay_ms(50);   // Power control(4)
    LCD_WriteReg(0x001E,0x00B0);    delay_ms(50);   // Power control(5)
    LCD_WriteReg(0x0001,0x2B3F);    delay_ms(50);   // Driver Output Control /* 320*240 0x2B3F */
    LCD_WriteReg(0x0002,0x0600);    delay_ms(50);   // LCD Drive AC Control
    LCD_WriteReg(0x0010,0x0000);    delay_ms(50);   // Sleep Mode off
    LCD_WriteReg(0x0011,0x6070);    delay_ms(50);   // Entry Mode
    LCD_WriteReg(0x0005,0x0000);    delay_ms(50);   // Compare register(1)
    LCD_WriteReg(0x0006,0x0000);    delay_ms(50);   // Compare register(2)
    LCD_WriteReg(0x0016,0xEF1C);    delay_ms(50);   // Horizontal Porch
    LCD_WriteReg(0x0017,0x0003);    delay_ms(50);   // Vertical Porch
    LCD_WriteReg(0x0007,0x0133);    delay_ms(50);   // Display Control
    LCD_WriteReg(0x000B,0x0000);    delay_ms(50);   // Frame Cycle control
    LCD_WriteReg(0x000F,0x0000);    delay_ms(50);   // Gate scan start position
    LCD_WriteReg(0x0041,0x0000);    delay_ms(50);   // Vertical scroll control(1)
    LCD_WriteReg(0x0042,0x0000);    delay_ms(50);   // Vertical scroll control(2)
    LCD_WriteReg(0x0048,0x0000);    delay_ms(50);   // First window start
    LCD_WriteReg(0x0049,0x013F);    delay_ms(50);   // First window end
    LCD_WriteReg(0x004A,0x0000);    delay_ms(50);   // Second window start
    LCD_WriteReg(0x004B,0x0000);    delay_ms(50);   // Second window end
    LCD_WriteReg(0x0044,0xEF00);    delay_ms(50);   // Horizontal RAM address position
    LCD_WriteReg(0x0045,0x0000);    delay_ms(50);   // Vertical RAM address start position
    LCD_WriteReg(0x0046,0x013F);    delay_ms(50);   // Vertical RAM address end position
    LCD_WriteReg(0x0030,0x0707);    delay_ms(50);   // gamma control(1)
    LCD_WriteReg(0x0031,0x0204);    delay_ms(50);   // gamma control(2)
    LCD_WriteReg(0x0032,0x0204);    delay_ms(50);   // gamma control(3)
    LCD_WriteReg(0x0033,0x0502);    delay_ms(50);   // gamma control(4)
    LCD_WriteReg(0x0034,0x0507);    delay_ms(50);   // gamma control(5)
    LCD_WriteReg(0x0035,0x0204);    delay_ms(50);   // gamma control(6)
    LCD_WriteReg(0x0036,0x0204);    delay_ms(50);   // gamma control(7)
    LCD_WriteReg(0x0037,0x0502);    delay_ms(50);   // gamma control(8)
    LCD_WriteReg(0x003A,0x0302);    delay_ms(50);   // gamma control(9)
    LCD_WriteReg(0x003B,0x0302);    delay_ms(50);   // gamma control(10)
    LCD_WriteReg(0x0023,0x0000);    delay_ms(50);   // RAM write data mask(1)
    LCD_WriteReg(0x0024,0x0000);    delay_ms(50);   // RAM write data mask(2)
    LCD_WriteReg(0x0025,0x8000);    delay_ms(50);   // Frame Frequency
    LCD_WriteReg(0x004f,0);                         // Set GDDRAM Y address counter
    LCD_WriteReg(0x004e,0);                         // Set GDDRAM X address counter

    delay_ms(50);
}

// Paints the LCD with Color
void LCD_Clear( unsigned short Color )
{
    unsigned int i;

    LCD_SetCursor(0,0);

    GPIOC->BRR  = CS_low_Pin;

    LCD_WriteIndex( 0x0022 );
    for( i=0; i< MAX_X*MAX_Y; i++ )
        LCD_WriteData( Color );

    GPIOC->BSRR = CS_low_Pin;
}

// Write a command
void LCD_WriteIndex( unsigned short index )
{ //look at Chapter 13 to figure out how long
            //each pin should be high/low... create the ns Delay loop
            GPIOC->BRR = DC_Pin;                                        // Set Command
            GPIOC->BSRR = RD_low_Pin;                   // set Rd pin high (why?)
            //delay_ms(0);
            GPIOB->ODR = index;                                         // write out data
            //delay_ms(0);
            GPIOC->BRR = WR_low_Pin;                    // set WR pin low then high
            GPIOC->BSRR = WR_low_Pin;

}

// Write data
void LCD_WriteData( unsigned short data )
{

            GPIOC->BSRR = DC_Pin;                                       // Set to Data
            GPIOC->BSRR = RD_low_Pin;                   // set Rd pin high (why?)
            //delay_ms(0);
            GPIOB->ODR = data;                                          // write out data
            //delay_ms(0);
            GPIOC->BRR = WR_low_Pin;                    // set WR pin low then high
            GPIOC->BSRR = WR_low_Pin;

}

//
void LCD_WriteReg( unsigned short LCD_Reg, unsigned short LCD_RegValue )
{
    GPIOC->BRR  = CS_low_Pin;

    LCD_WriteIndex( LCD_Reg );
    LCD_WriteData( LCD_RegValue );

    GPIOC->BSRR = CS_low_Pin;
}

// Read data: called twice for display data (first is dummy)
unsigned short LCD_ReadData(void)
{
    unsigned short data;

    GPIOC->BSRR = DC_Pin;
    GPIOC->BSRR = WR_low_Pin;

    GPIOC->BRR  = RD_low_Pin;

    GPIOB->CRL  = 0x44444444;   // floating input
    GPIOB->CRH  = 0x44444444;

    data = GPIOB->IDR;
                //data = GPIOB->IDR;                                        // we added this

    GPIOB->CRL  = 0x33333333;   // output pp
    GPIOB->CRH  = 0x33333333;

    GPIOC->BSRR = RD_low_Pin;

    return data;
}

// Used for reading device code (should be 0x8989)
unsigned short LCD_ReadReg( unsigned short LCD_Reg )
{
    unsigned short data;

    GPIOC->BRR  = CS_low_Pin;

    LCD_WriteIndex( LCD_Reg );
    data = LCD_ReadData();

    GPIOC->BSRR = CS_low_Pin;

    return data;
}

// Set cursor to x y adress
void LCD_SetCursor( unsigned short x, unsigned int y )
{
    #if   ( DISP_ORIENTATION == 90 ) || ( DISP_ORIENTATION == 270 )

        unsigned short swap_temp;

        y = (MAX_Y-1) - y;
        swap_temp = y;
        y = x;
        x = swap_temp;

    #elif ( DISP_ORIENTATION ==  0 ) || ( DISP_ORIENTATION == 180 )

        y = (MAX_Y-1) - y;
    #endif

    LCD_WriteReg( 0x004E, x );
    LCD_WriteReg( 0x004F, y );
}

void delay_ms( unsigned int ms )
{
                unsigned int i; // # of 1 ms iterations
                unsigned int d;         // 1 ms delay
                unsigned int val = 0x63E; //1 ms delay
                for (i = 0; i < ms; i++)
                {                               for (d = 0; d < val; d++);
                }
}
//  END OF FILE
```