ECE 3710

FINAL PROJECT

LASER TAG MODULE

Nathan Tipton

Tarrin Rasmussen

Josh Smith

Microcontroller Hardware and Software

Electrical and Computer Engineering Department

Utah State University

Contents

List of Figures

INTRODUCTION

This document describes the design for a laser tag module. Each player would have their own module. The object is to be the last player alive. Each player starts out with 1-10 lives. A handicap may be set for a player to have less than 10 lives. Background music is played for the players enjoyment. The module is responsive and will alert the player to changes in status such as STUNNED or DEAD with sounds and LED displays.

SCOPE

This document discusses, in detail, the electrical and software design for a laser tag module. It includes requirements, dependencies, and theory of operation. Schematics and code are used to provide a more detailed design explanation. Testing procedures for requirements are included. Complete design code is located in the Appendix. Procedure for generating audio files, for playback, is not discussed in this document. Laser gun design is not discussed in this document.

DESIGN OVERVIEW

Requirements

The following are the given requirements for the laser tag module.

1. The player will start with 1-10 lives. This will be chosen at game start using a DIP switch.
2. A player's lives will be displayed on a LED bar graph.
3. A player's Status: ALIVE, STUNNED, or DEAD will be indicated with three colored LEDs.
4. Upon a player being hit, the player will be stunned and lose a life. A short stun period will begin. During this time the player may not be hit again.
5. Upon losing all lives, a player will be declared dead.
6. A player may be hit by using a laser gun to light up a photoresistor on the player's module.
7. An audible alert will play when a player is hit.
8. Background music will be played using a SD card.

Dependencies

The following are dependencies for the laser tag module.

1. A 5 volt power supply
2. micro SD card containing wav files.
3. MCP4725 12-Bit DAC breakout board
4. micro SD card breakout board
5. speaker
6. light gun

Theory of Operation

Upon power on or reset, the module is calibrated for the current lighting conditions. A sample is taken 100 times to determine the lower bound needed to determine if a player has been hit. The player's amount of starting lives is determined from the current status of the DIP switch. This is displayed on the LED bar graph. A photoresistor, connected in a voltage divider configuration and then connected to the ADC provided by the Tiva C microcontroller, is used as the target/input for the laser tag module. This ADC value is sampled/polled and if it ever drops below the lower bound found during initialization, the player is "stunned" and enters a short period during which he/she cannot be hit again. The LED bar graph updates accordingly each time until finally showing no lives and a red led lights up signifying the end of the game for that player.

Every time that the player is hit. A signal is sent to another microcontroller connected to a DAC and audio device. This microcontroller alerts the player by making a hit noise.

During gameplay, a microcontroller connected to a microSD card is pulling values from a wav file stored on the SD card and playing them through a DAC/audio device system on a 30 second loop. This is for the players enjoyment and should be part of the arena/gameplay area thereby shared by all players.

Design Alternatives

An alternative design would be to implement the module as a part of the laser gun, making them a more compact, single unit. This would allow the module to control when a player can shoot and the player's firing rate. This approach was not taken due to the complications of building a laser gun within the given time constraints. A manufactured laser gun was used instead.

DESIGN DETAILS

Hardware Design

LED Displays and Visual Alerts

LEDs and switches are used for player input and feedback. The LED bar graph is used to display the current lives of the player. At reset, the player's beginning lives are displayed on the graph. Each time the player is hit, he loses a life. When the bar graph is empty, meaning no remaining lives, the player is dead.

Beginning lives are set with the dip switch. By entering a binary handicap of zero, the player begins with all ten lives. If the player enters one for the handicap, he starts with nine, etc. The minimum lives a player can start with is zero. The handicap must be entered before reset.

When the player is hit, there is a stun period of 4 seconds. When stunned, the player cannot be hit. This mode can be seen by other players by the yellow LED. When the player is not stunned, the green LED is lit. When stunned, the yellow LED is lit. When a player dies, the red LED is lit, and there is no more interaction with the user until reset.

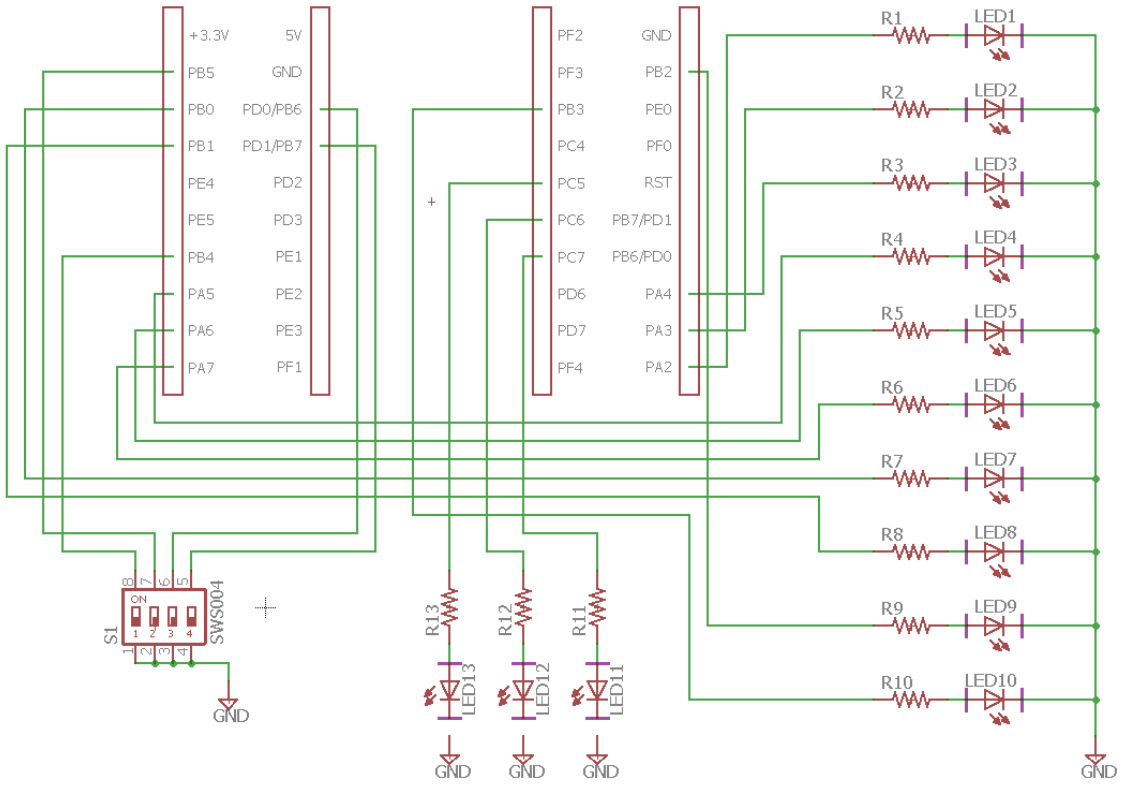

Figure 1: LED Display and DIP Switch Schematic

Photoresistor and Game Input

The analog-to-digital-converter(ADC) of the microcontroller is used to convert the analog signal from the photoresistor. Initially, the photoresistor was not used with a voltage divider. This caused inaccurate readings of the photoresistor voltage change. Without a voltage divider there is no reference point except for the VCC and Ground. A voltage divider provides a well defined reference point to measure the voltage changes.

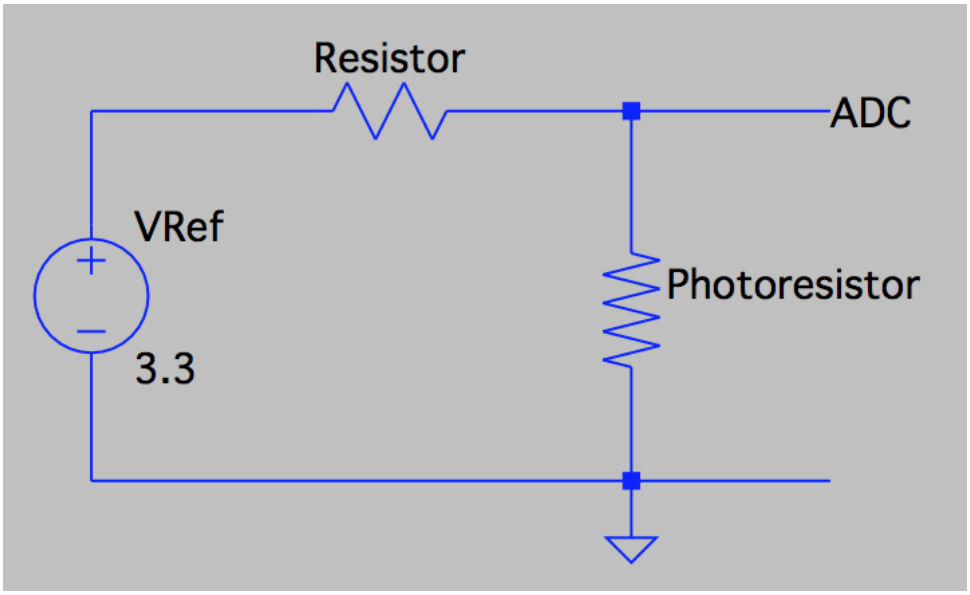

Figure 2: Photoresistor voltage divider circuit diagram

Audio Effects and SD Card Interface

The microcontroller is connected to the SD card as shown in the figure below. The Tiva C SSI module used is SSI3 (PD0 - PD3) and the CS signal is controlled by GPIO pin PA2.

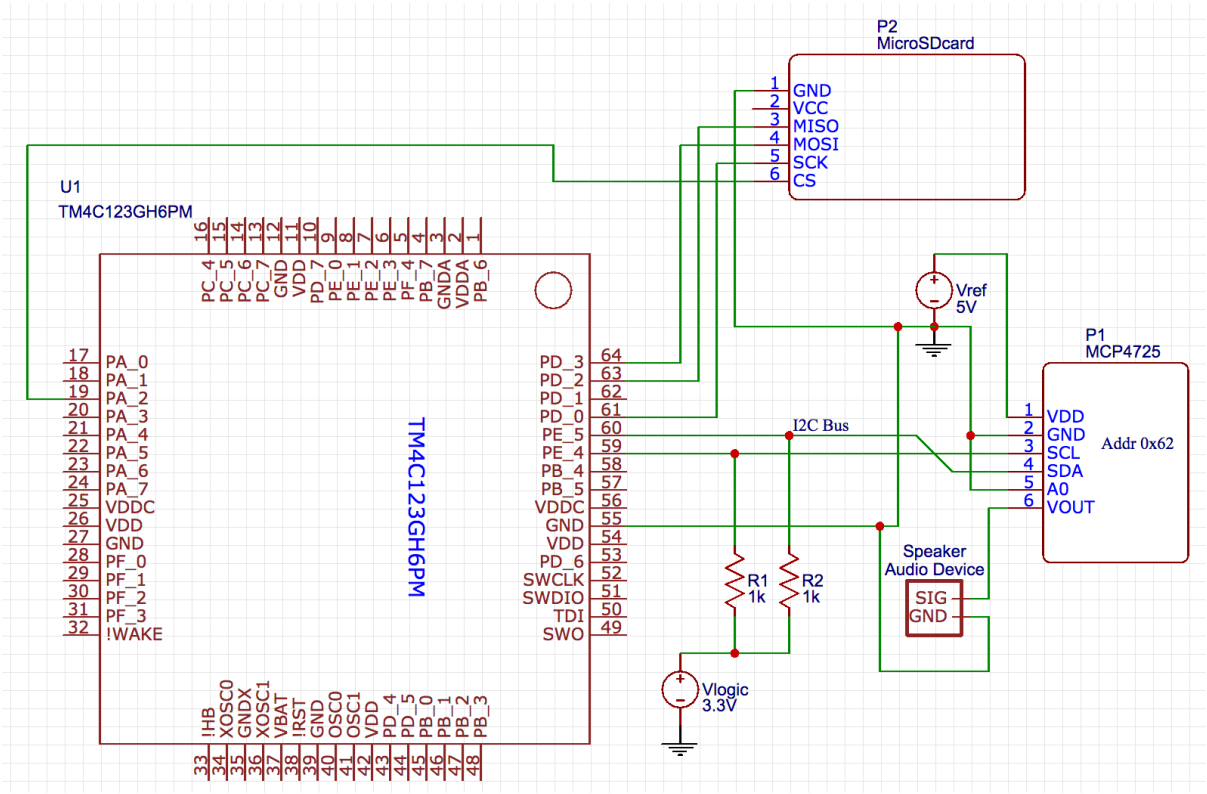

Figure 3: SD Card, DAC, Audio Device and Microcontroller hardware design

Software Design

LED Displays and Visual Alerts

Driving the LEDs and DIP Switch was done with GPIO ports. Internal pull-up resistors were used to bias the switches. To time the four second stun delay, the system clock was divided by four, and SysTick timer was loaded with its maximum load value.
The LED Bar graph was programmed by creating an array with values 0, 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 . The players handicap determines where to start in the array. Each time the player is hit, they are decremented through the array until zero. Once the lives array is at zero, the player is dead and no more interaction can happen until reset.

Photoresistor and Game Input

The laser tag module can be used in a variety of light conditions. To make this possible, a calibration mode is enabled upon power on or reset. The ADC samples the photo resistor to obtain a value for the light conditions in the playing area. It checks this sample against the default lower bound which is set at the max value for the photoresistor. If the sample is less than the current lower bound then the lower bound is set to the sample value. This process is repeated for 100 samples. After which the calibration mode is disabled. This gives the microcontroller the lower bound for the playing area ambient light. This lower bound is reduced further to allow for error thereby avoiding erroneous hits. During game play the ADC will sample the photoresistor when upon a timer expire. An interrupt is triggered for the ADC. The sample value is pulled from the ADC SS3 FIFO. The sample is checked against the lower bound. If the sample is lower the hit function is called. Otherwise, the sample is ignored until the next interrupt.

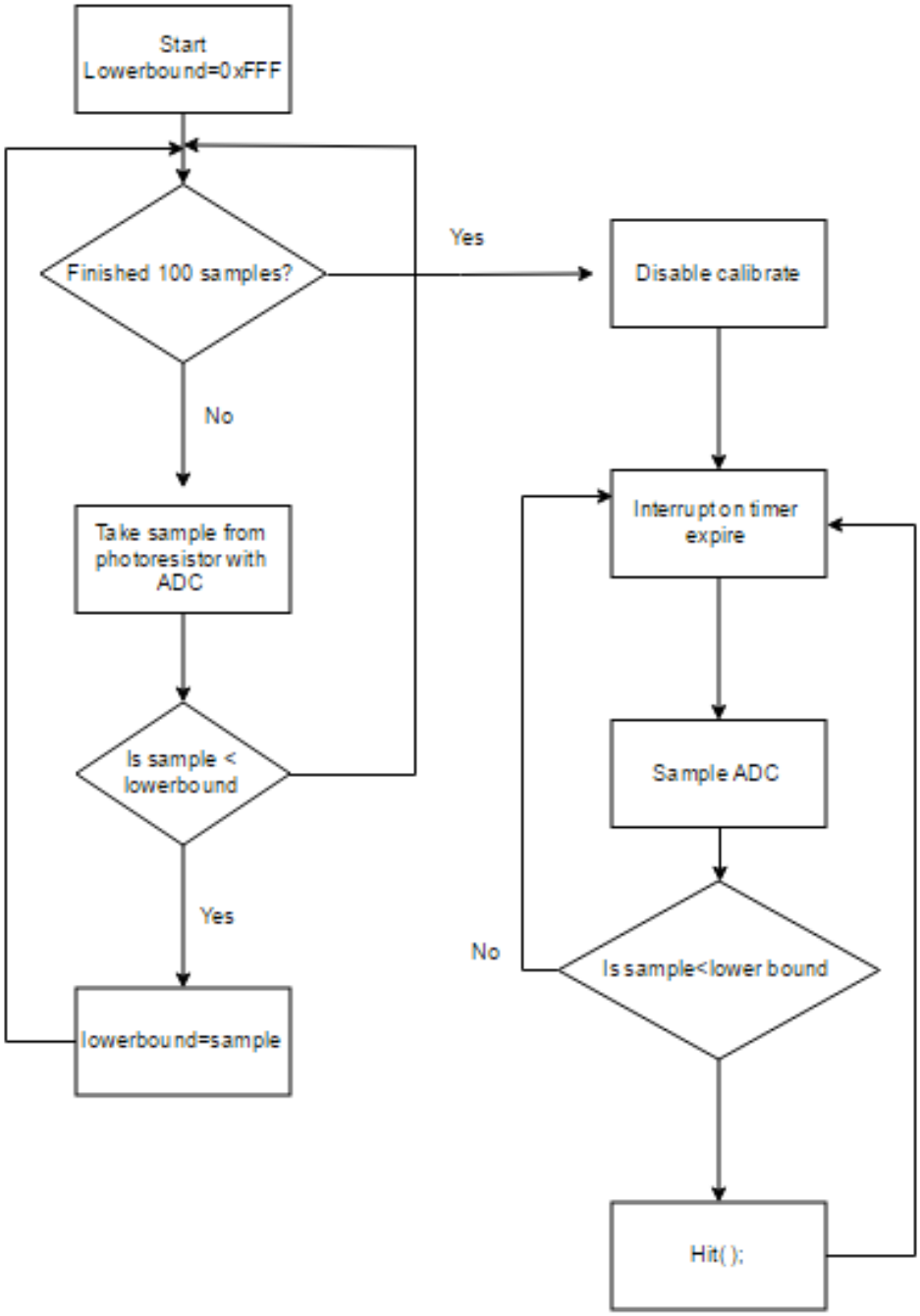

Figure 4: Flowchart for determining a hit using photoresistor

Audio Effects and SD Card Interface

Communicating with an SD card using the SPI protocol follows the format shown in the figure below.

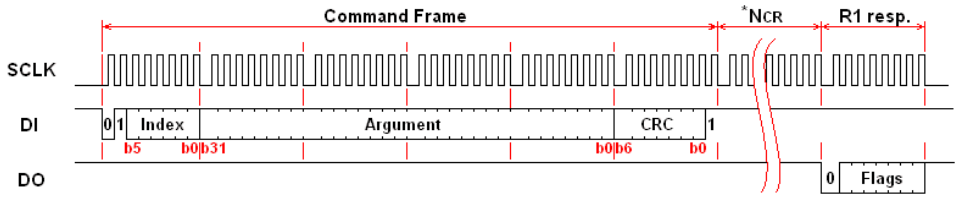

Figure 5: SD card SPI mode command frame

The index is the command index such that CMD0 would send first 0x40 | 0x0 and CMD1 would send first 0x40 | 0x1 and so on. A chart of the supported commands is found in the figure below.

Command Index	Argument	Response	Data	Abbreviation	Description
CMD0	None(0)	R1	No	GO_IDLE_STATE	Software reset.
CMD1	None(0)	R1	No	SEND_OP_COND	Initiate initialization process.
ACMD41(*1)	*2	R1	No	APP_SEND_OP_COND	For only SDC. Initiate initialization process.
CMD8	*3	R7	No	SEND_IF_COND	For only SDC V2. Check voltage range.
CMD9	None(0)	R1	Yes	SEND_CSD	Read CSD register.
CMD10	None(0)	R1	Yes	SEND_CID	Read CID register.
CMD12	None(0)	R1b	No	STOP_TRANSMISSION	Stop to read data.
CMD16	Block length[31:0]	R1	No	SET_BLOCKLEN	Change R/W block size.
CMD17	Address[31:0]	R1	Yes	READ_SINGLE_BLOCK	Read a block.
CMD18	Address[31:0]	R1	Yes	READ_MULTIPLE_BLOCK	Read multiple blocks.
CMD23	Number of blocks[15:0]	R1	No	SET_BLOCK_COUNT	For only MMC. Define number of blocks to transfer with next multi-block read/write command.
ACMD23(*1)	Number of blocks[22:0]	R1	No	SET_WR_BLOCK_ERASE_COUNT	For only SDC. Define number of blocks to pre-erase with next multi-block write command.
CMD24	Address[31:0]	R1	Yes	WRITE_BLOCK	Write a block.
CMD25	Address[31:0]	R1	Yes	WRITE_MULTIPLE_BLOCK	Write multiple blocks.
CMD55(*1)	None(0)	R1	No	APP_CMD	Leading command of ACMD<n> command.
CMD58	None(0)	R3	No	READ_OCR	Read OCR.
*1:ACMD<n> means a command sequense of CMD55-CMD<n>.					
*2: Rsv(0)[31], HCS[30], Rsv(0)[29:0]					
*3: Rsv(0)[31:12], Supply Voltage(1)[11:8], Check Pattern(0xAA)[7:0]					

Figure 6: SD card SPI mode commands

The master then sends the arguments of the command in the next 6 bytes and then a final byte that varies based on the type of sd card and command ending in a '1' to signify the end

of the frame. The master then provides a clock to the slave by sending the byte 0xFF from 0 - 8 times waiting for the response. The response comes in the form shown in Figure 7.

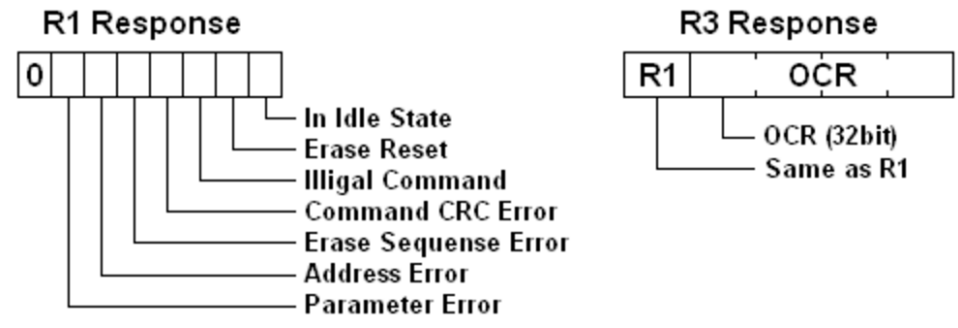

Figure 7: SD Response Format

Initializing an SD card in SPI mode is a complicated process and is outlined in the flowchart below. The steps that were taken for the SD card used in this project are outlined in the function initSD found in the code appendix and defined in the file "functions.c".

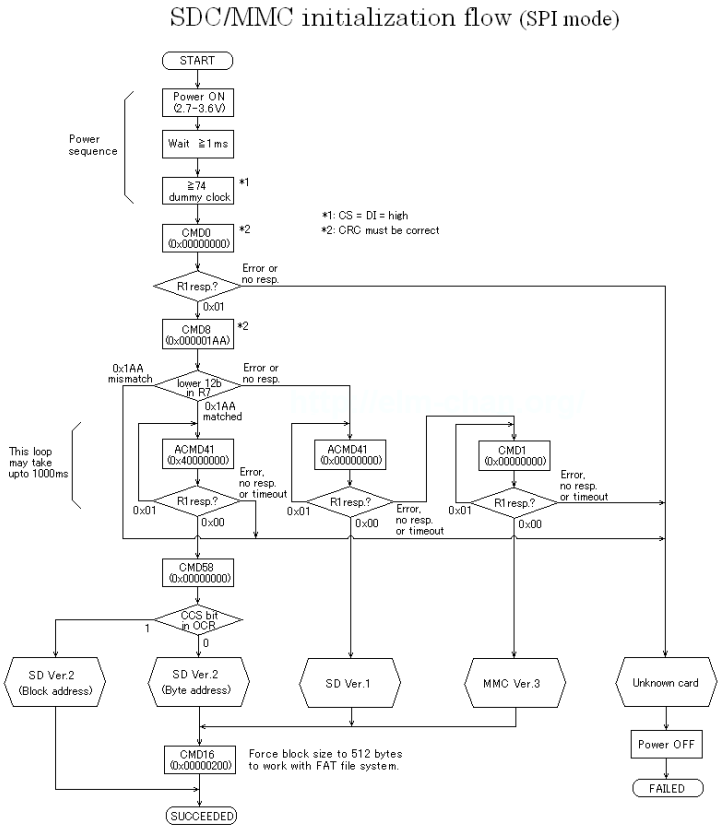

Figure 8: SPI mode initialization [1]

Retrieving information from the SD card is relatively simple. CMD17 is sent to the SD card over the SPI interface. The microcontroller then provides the clock as it would for sending any other command and waits for the response 0x0. This means that there were no errors and the card is ready to transmit. This process in shown in the figures below.

Figure 9: Reading block from SD card

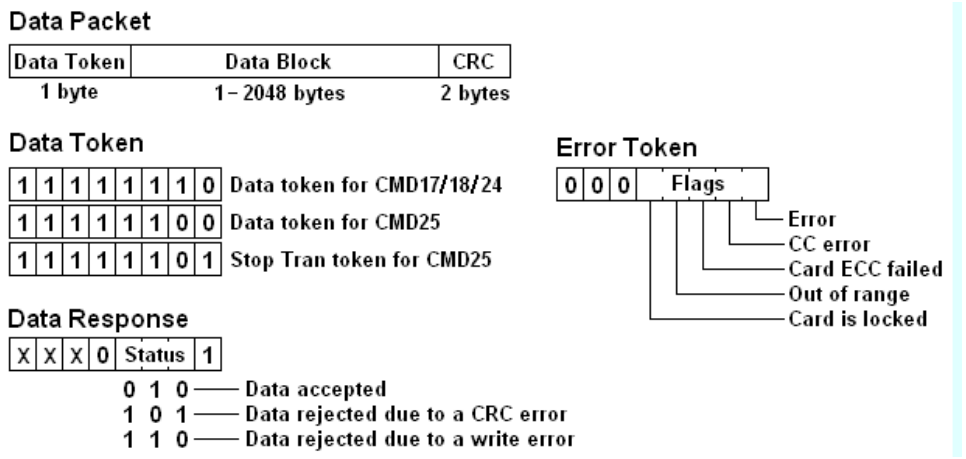

Figure 10: SPI mode Data Packet

The microcontroller continues to provide a clock by sending 0xFF until it receives the data or error token. If it receives the data token, the microcontroller then retrieves a block (512 bytes) from the card.

This process is used in the laser tag module to get successive blocks of a wav file and send them to the DAC over a generic I2C interface. The song used was the theme song of the James Bond (007) movies. The wav file used had 8 bit encoding and was very convenient because the values retrieved could be send directly/unchanged to the DAC and thereby to the audio device to play the song.

TESTING

1. Handicap Test
   (a) Set handicap with DIP switches and reset the microcontroller. Repeat for all possible inputs on the DIP switch.
   (b) Verify each handicap starts with the correct lives by looking at the LED bar graph. Also verify that the minimum starting lives is one.
   (c) With some debugging, we found that each handicap displayed the right starting lives, and the minimum starting lives was one regardless of higher handicaps.
   (d) The Handicap test verifies requirement 1.
2. Lives Test
   (a) Start with handicap and reset microcontroller. Use laser to decrements lives until dead. Repeat for all handicaps.
   (b) Verify the following each time the laser hits the sensor.
      i. Lives are decremented on LED bar graph.
      ii. If the player is not dead, the green LED turns to yellow for four seconds before turning back green. If the player is dead, the LED turns to read and stays indefinitely.
      iii. While in dead or stunned states, lives are not further decremented when sensor is hit by laser.
      iv. An audible alert is heard each time the sensor is hit when not stunned or dead.
      v. The background music is still playing. It should not stop at any time while powered.
   (c) With lots of debugging, we found that every requirement was filled for this test.
   (d) The Lives Test verifies requirements 2, 3, 4, 5, 6, 7, and 8.
3. SPI Communication Test
   (a) Figure 11 shows commands being sent to the SD card over SPI with the SD card sending back responses.

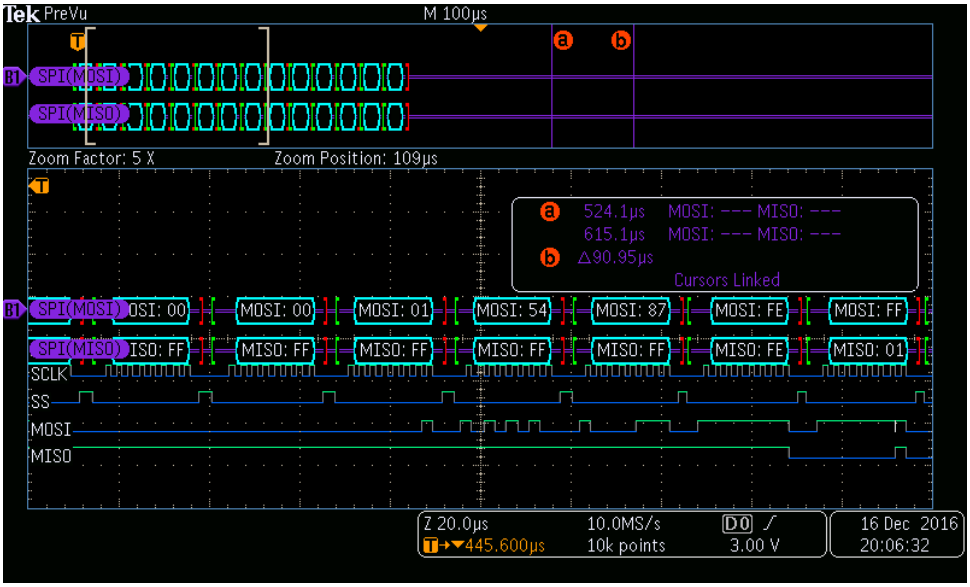

Figure 11: Sending Commands to SD card

(b) Once we were successfully sending commands to the SD card we tested the read block command using our wave file. Using a hex editor we were able to see the location of the wav file on the SD card and the contents of the wav file as seen in figure 12.

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00404000   52 49 46 46 F0 7A 09 00 57 41 56 45 66 6D 74 20   RIFFðz..WAVEfmt      Sector 8224
00404010   10 00 00 00 01 00 01 00 22 56 00 00 22 56 00 00   ........"V.."V..
00404020   01 00 08 00 64 61 74 61 CC 7A 09 00 7C 7E 7E 7E   ....dataÌz..|~~~
00404030   80 83 88 89 88 87 87 84 81 7E 7E 80 81 82 83 83   €ƒˆ‰ˆ‡„.~~€.,ƒƒ
00404040   81 80 80 81 82 83 82 82 82 82 82 83 83 86 89 8C   .€€.‚ƒ‚‚‚‚‚ƒƒ†‰Œ
00404050   8B 8B 89 88 87 89 8D 90 92 92 91 8C 86 83 82 80   ‹‹‰ˆ‡‰..’’‘Œ†ƒ‚€
00404060   7F 7D 7A 77 75 72 6F 6F 71 75 78 79 79 7A 7B 7E   .}zwurooquxyyz{~
00404070   83 8A 8F 92 92 92 92 8E 8A 88 89 8A 89 88 88 86   ƒŠ.’’’’ŽŠˆ‰Š‰ˆˆ†
00404080   83 83 83 83 83 81 7F 7E 7D 7C 7C 7D 7E 81 83 84   ƒƒƒƒƒ..~}||}~.ƒ„
00404090   84 85 87 88 89 8D 92 94 98 99 98 93 91 8E 8D 8C   „…‡ˆ‰.’”˜™˜“‘Ž.Œ
004040A0   8B 89 86 82 7E 79 74 73 74 76 77 77 77 77 77 78   ‹‰†‚~ytstvwwwwwx
004040B0   7B 7F 80 7E 7B 78 73 6D 69 69 6B 6B 6A 6C 6D 6C   {.€~{xsmiikkjlml
004040C0   6D 6F 72 74 76 77 79 7A 7E 80 83 85 89 8C 8C 8B   mortvwyz~€ƒ…‰ŒŒ‹
004040D0   8B 8B 8B 8B 8D 90 92 93 95 93 8D 89 87 86 85 84   ‹‹‹‹..’“•“.‰‡†…„
004040E0   83 82 80 7F 7D 79 79 7D 80 83 83 83 83 86 8B   ƒ‚€.}yy}€ƒƒƒƒƒ†‹
004040F0   90 93 95 93 92 8F 8A 85 83 85 85 83 83 83 80 7E   .“•“’.Š…ƒ……ƒƒƒ€~
00404100   7E 7F 7E 7E 7B 79 77 77 78 78 77 78 79 79 78 78   ~.~~{ywwxxwxyyxx
00404110   79 7B 7B 7D 80 84 88 8B 8D 8A 88 86 85 85 84 85   y{{}€„ˆ‹.Šˆ†……„…
00404120   84 83 80 7D 78 75 76 7A 7E 7E 80 80 80 82 85 88   „ƒ€}xuvz~~€€€‚…ˆ
00404130   8B 8B 89 86 83 80 7A 77 79 7B 7A 79 79 79 76 75   ‹‹‰†ƒ€zwy{zyyyvu
00404140   76 77 78 78 78 76 76 78 79 7C 7E 81 83 83 83 83   vwxxxvvxy|~.ƒƒƒƒ
00404150   83 83 82 82 84 87 88 8B 8A 85 81 80 7E 7E 7C 7B   ƒƒ‚‚„‡ˆ‹Š….€~~|{
00404160   79 75 72 6D 69 69 6D 72 74 77 78 78 79 7D 80 86   yurmiimrtwxxy}€†
00404170   8B 8D 8D 8A 88 84 80 7E 7E 7D 79 79 79 78 77 78   ‹..Šˆ„€~~}yyyxwx
00404180   79 79 79 77 74 72 73 74 77 79 7B 7E 7E 7D 7E 80   yyywtrstwy{~~}~€
00404190   81 81 81 83 84 86 88 8B 89 86 83 83 81 81 80 80   ...ƒ„†ˆ‹‰†ƒƒ..€€
004041A0   7D 7B 78 74 72 73 76 79 7A 7A 7A 7A 7D 81 86 8A   }{xtrsvyzzzz}.†Š
004041B0   8D 8D 8C 8A 87 83 80 80 81 80 80 80 80 80 81 83   ..ŒŠ‡ƒ€€.€€€€€.ƒ
004041C0   84 84 83 83 81 81 84 87 89 8B 8D 8F 8F 8C 8C 8D   ...ƒƒ..„‡‰‹.....
```

Figure 12: Wav file contents and SD card location

Figures 13 and 14 show the results of the read block command. Comparing these

with the contents of the wav file shows that we are correctly reading data from the SD card.

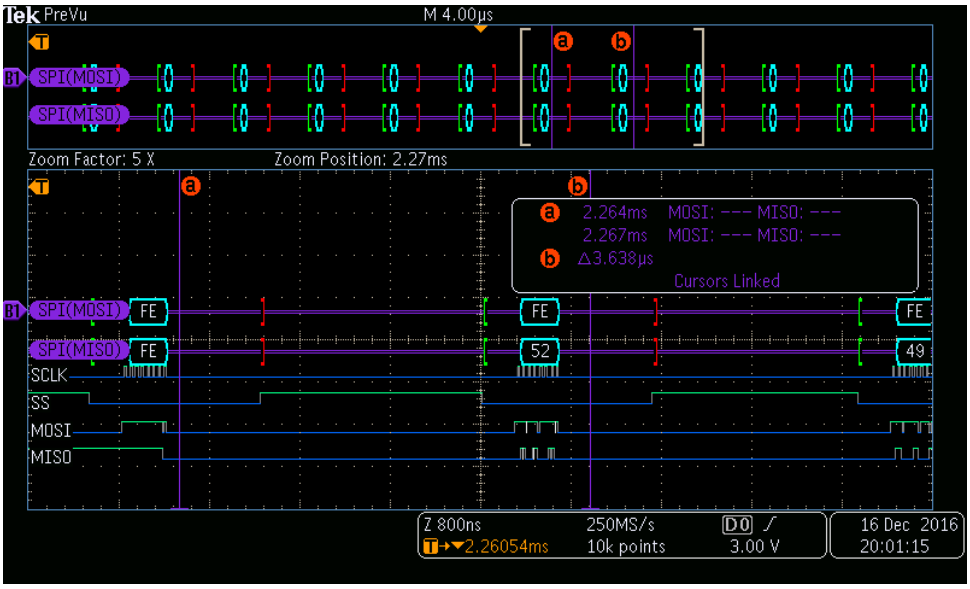

Figure 13: Response from SD card to Read block command

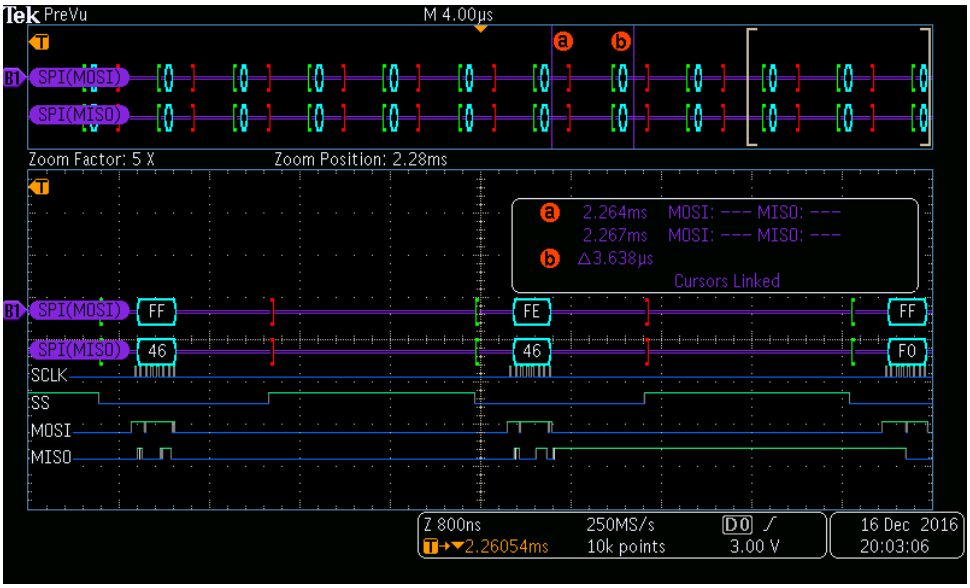

Figure 14: Reading block from SD card

(c) The data from the SD is placed in a buffer before being sent to the DAC. Figure 15 shows the data that was inserted into the buffer. This data matches the data seen on the logic analyzer and in the wav file.

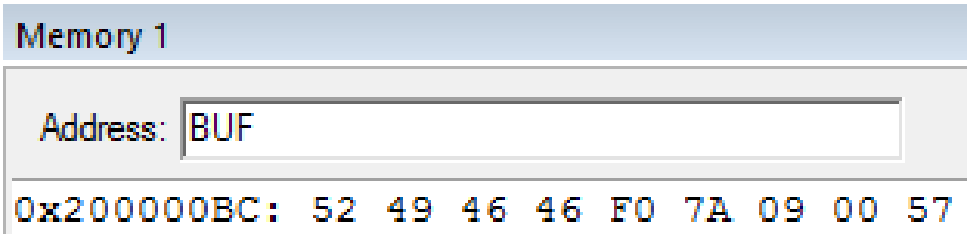

Figure 15: Debug view of buffer containing data pulled from SD card

4. I2C Communication Test

   (a) Figure 16 shows the logic analyzer verification of communication between the microcontroller and the DAC using I2C.

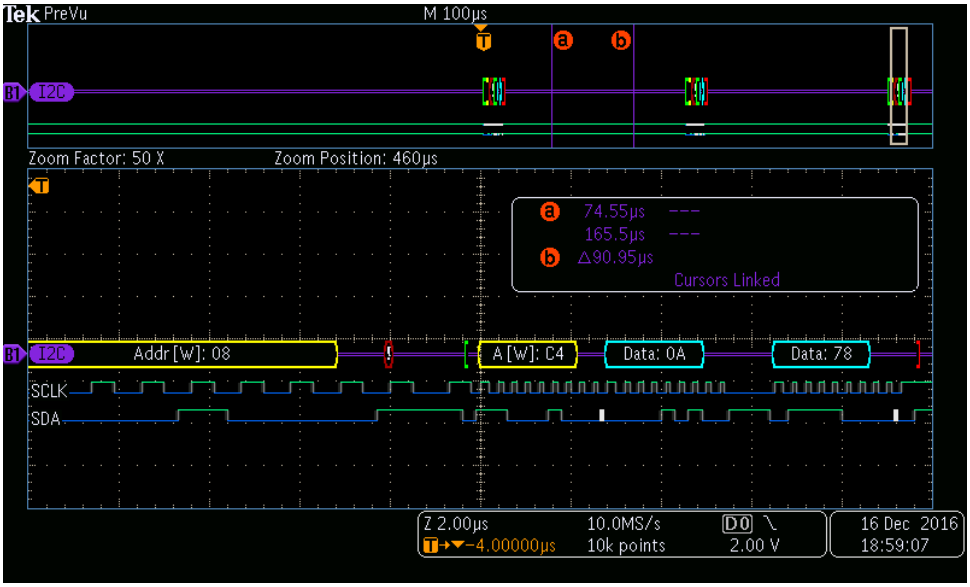

Figure 16: Screenshot of I2C Communication on Logic Analyzer

CONCLUSION

Initially, the same microcontroller was used to play the hit sounds and the background music. The interrupts from the hit were interfering with the SPI communication between the SD card and the microcontroller. When a hit occurred the music was able to restart after the hit was processed. After hours of debugging we were still unable to solve this and decided to have the hit sounds controlled by an additional microcontroller. Ideally, the whole design would be run with one microcontroller. Design revisions to simplify interrupts to remove possible interference with SPI communications would reduce the number of microcontrollers used. A possible improvement would be to allow custom hit sounds and additional background music. These additional files would be on the SD card. The software would need to be able to determine the sector offset for each file and the file size in order to have correct playback. Multiple photoresistors for placing on various locations of the user's body would be a possible future addition.

REFERENCES

[1] http://elm-chan.org/docs/mmc/mmc_e.html

APPENDIX

Code for Module

main.c

```
#include "TM4C123.h"
#include "bargraph.h"

unsigned short adcc;
unsigned short l_bound=0x0FFF;
unsigned char en_calibrate;
unsigned int calibrate_cycle;


void calibrate();

void ADC0SS3_Handler() {

        adcc = ADC0->SSFIFO3;
        ADC0->ISC = (1 << 3);
        if(en_calibrate){
calibrate();
}

        if(adcc<l_bound&&(!en_calibrate)){
        hit();
        }
}


void calibrate(){
```

```
        if(adcc<l_bound){
                l_bound=adcc;
        }

        calibrate_cycle--;


        if(calibrate_cycle==0){
                en_calibrate=0;
                l_bound=(l_bound-0x1A);
        }


}


int main(){


        //
        // Clock init
        //

        //RCC 16 MHz PLL
        SYSCTL->RCC = 0x7CE1540;
        SYSCTL->RCC2 = (0xC10 << 20);
        bgInit();

        //Enable clock to GPIO modules: Port E, B, F (338)
        SYSCTL->RCGCGPIO |= 0x10;

        //Enable ADC clock for ADC module 0 (350)
        SYSCTL->RCGCADC |= 1;

        //Enable Timer0 Clock
```

```
        SYSCTL->RCGCTIMER |= 1;

        //Enable I2C clock (346)
        SYSCTL->RCGCI2C |= 0x1;


        //
        //ADC initialization (814)
        //Register Map (815)
        //

        //Enable alternate function PE3 (668, 1337)
        GPIOE->AFSEL |= (1 << 3);

        //Analog Enable (680)
        GPIOE->DEN &= ~(1 << 3);

        //GPIO Analog mode select (684)
        GPIOE->AMSEL |= (1 << 3);


        //
        // Sequencer3 init
        //



////Send to ss3fifo//
////Disable Sequencer
        ADC0->ACTSS &= ~(1 << 3);

        //Set Trigger Event
        ADC0->EMUX = (0x5 << (3 * 4));

        //Set Input Channel
```

```
        ADC0->SSMUX0 = 0;

        //Configure Sample

        ADC0->SSCTL3 = 0x6;

        //Set Mask
        ADC0->IM = (1 << 3);

        //Enable Sequencer
        ADC0->ACTSS |= (1 << 3);

        //Enable Interrupt
        NVIC->ISER[0] |= (1 << 17);


////-------------------------//


//        //Comparator
////      //Disable Sequencer
//        ADC0->ACTSS &= ~(1 << 3);
//        ADC0->DCCTL0=0x10;
//        ADC0->SSOP3=1;
//        //comp
//        ADC0->DCCMP0 = (0x970 << 16);
//        ADC0->DCCMP0 |=(0x970);
//        ADC0->SSDC3=0;
//        //Set Trigger Event
//        ADC0->EMUX = (0x5 << (3 * 4));
//
//        //Set Input Channel
//        ADC0->SSMUX0 = 0;
//
//        //Configure Sample
```

```
//        //0 or 2?
//        ADC0->SSCTL3 = 0x6;
//
//        //Set Mask
//        ADC0->IM = (1 << 19);
//
//        //Enable Sequencer
//        ADC0->ACTSS |= (1 << 3);
//
//        //Enable Interrupt
//        NVIC->ISER[0] |= (1 << 17);


        //
        //Timer 0 init
        //

        //Disable Timer
        TIMER0->CTL &= ~1;

        //Set Config Register
        TIMER0->CFG = 0;

        //Set Periodic
        TIMER0->TAMR |= 0x2;

        //Set Initial Value
        TIMER0->TAILR = 32000;

        //Set ADC Trigger
        TIMER0->CTL |= (1 << 5);

        //Enable Timer
        TIMER0->CTL |= 1;
```

```
calibrate_cycle=100;
en_calibrate=1;
while(en_calibrate){
        GPIOC->DATA |= 0xE0;


}
GPIOC->DATA &= ~0xE0;
GPIOC->DATA |= 0x20;


        while(1) {


                }


        }
```

bargraph.c

```
#include "bargraph.h"
#include "TM4C123.h"

unsigned short lives [11] = { 0, 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };
unsigned char l;
unsigned char stunned = 0;

void dead();
void stun();
```

```
void bgInit() {

        unsigned char handicap;

        //
        // GPIO Init
        //

        //Enable clock
        SYSCTL->RCGCGPIO |= 7;

        //Set direction
        GPIOA->DIR |= 0xFC;
        GPIOB->DIR = 0x0F;
        GPIOC->DIR |= 0xE0;

        //Disable Alternate
        GPIOA->AFSEL &= ~0xFC;
        GPIOB->AFSEL = 0;
        GPIOC->AFSEL &= ~0xE0;

        //Set Drive Strength
        GPIOA->DR8R |= 0xFC;
        GPIOB->DR8R |= 0x0F;
        GPIOC->DR8R |= 0xE0;

        //Set Pullup Resistors
        GPIOB->PUR |= 0xF0;

        //Digitally Enable Pins
        GPIOA->DEN |= 0xFC;
        GPIOB->DEN |= 0xFF;
        GPIOC->DEN |= 0xE0;

        //
```

```
        // Systick Init
        //

        SysTick->LOAD = 0x00FFFFFF;
        SysTick->CTRL = 2;

        //
        // Set Starting Lives
        //

        handicap = GPIOB->DATA >> 4;
        if (handicap > 9) handicap = 9;

        l = 10 - handicap;

        GPIOA->DATA |= lives[l] << 2;
        GPIOB->DATA |= lives[l] >> 6;

        GPIOC->DATA |= 0x20;

}

void hit() {
        if (!stunned) {
                l--;

                GPIOA->DATA &= ~0xFC;
                GPIOB->DATA &= ~0xF;

                GPIOA->DATA |= lives[l] << 2;
                GPIOB->DATA |= lives[l] >> 6;

                if (l == 0) dead();
                else stun();
        }
```

```
}

void stun() {
        GPIOC->DATA &= ~0xE0;
        GPIOC->DATA |= 0x80;
        stunned = 1;
        SysTick->VAL = 1; //Reset COUNT
        SysTick->CTRL |= 1; //Start SysTick
}

void dead() {
        GPIOC->DATA &= ~0xE0;
        GPIOC->DATA |= 0x40;
        while(1);
}

void SysTick_Handler() {
        GPIOC->DATA &= ~0xE0;
        GPIOC->DATA |= 0x20;
        stunned = 0;
        SysTick->CTRL &= ~1; //Stop SysTick
}
```

bargraph.h

```
#ifndef _BARGRAPH_
#define _BARGRAPH_

//
// Bar graph header for laser tag
//

void bgInit(void); //Initialize lights
void hit(void); //Call to decrement lives
```

```
#endif
```

Code for Audio Component

main.c

```
#include "functions.h"

const unsigned int CHEWY_NOISE = 0x9440;
const unsigned int CHEWY_BLOCK_NUM = 230;
const unsigned int DARTH_VADER_NOISE = 0x9672;
const unsigned int DARTH_VADER_BLOCK_NUM = 0x3D;
const unsigned int BOND_THEME = 0x2020;
const unsigned int BOND_BLOCK_NUM = 0x4BD;
const unsigned int TOO_SEXY = 0x9736;
const unsigned int TOO_SEXY_BLOCK_NUM = 0x153;

int main(void)
{
        initPLL();
        initInterrupts();
        initI2C();
        initSSI3();
        initSD();
        while(1)
        {
                playWavFileAndSFX(BOND_THEME, BOND_BLOCK_NUM, 100);
                //playWavFile(TOO_SEXY, TOO_SEXY_BLOCK_NUM, 100);
                //playHitNoise();
        }
}
```

functions.h

```
#define FUNCTIONS_INCLUDED
#ifndef FUNCTIONS_INCLUDED

void playWavFile(unsigned int block_addr, unsigned int block_num, const unsigned in
void initPLL();
void initI2C();
void send2BytesI2C(unsigned short s, unsigned char addr);
void sinWave(const unsigned int delay, const unsigned int cycles);
void initPBInterrupt();
void playHitNoise();
void playDeadNoise();
void GPIOB_Handler(void);
unsigned char sendByteSSI(unsigned char val, unsigned char cs);
void setCS(unsigned char bool_val);
void initSSI3();
unsigned char maskNumber(unsigned long long num, unsigned char b);
unsigned char sendCmdToSD(unsigned char cmd, unsigned long arg, unsigned char crc);
unsigned char getBlockFromSD(unsigned int addr, unsigned char* buf);
void initSD();
void GPIOA_Handler(void);
void GPIOC_Handler(void);
void GPIOE_Handler(void);

#endif
```

functions.c

```
#include "functions.h"
#include "TM4C123.h"

const unsigned int NUM_VALUES = 40;
unsigned short DACValues[] = {2047,2367,2680,2976,3250,3494,3703,3871,3994,4069,
```

```
                                                        4094,4069,3994,3871,3703,
                                                        2047,1727,1414,1118, 844,
25,
                                                        0,   25,  100,
223, 391, 600, 844,1118,1414,1727};
volatile unsigned int* I2CModule2 = (unsigned int*)0x40022000;
volatile unsigned char* SSI3t = (unsigned char *)0x4000B000;
volatile unsigned char* portA = (unsigned char*)0x40004000; //base address for port
volatile unsigned char* portB = (unsigned char*)0x40005000; //base address for port
volatile unsigned char* portC = (unsigned char*)0x40006000; //base address for port
volatile unsigned char* portD = (unsigned char*)0x40007000; //base address for port
volatile unsigned char* portE = (unsigned char*)0x40024000; //base address for port
volatile unsigned char* portF = (unsigned char*)0x40025000; //base address for port
volatile unsigned int* SYSCTRL = (unsigned int*)0x400FE000;
unsigned int *INTERRUPT = (unsigned int*)0xE000E000;
unsigned char interrupt_status;
unsigned char play_hit = 0x0;
unsigned char play_dead = 0x0;
const unsigned int MAX_ITER_WAIT_FOR_SD = 1200;

void send2BytesI2C(unsigned short s, unsigned char addr)
{
        s &= 0xffff;
        addr <<= 1;
        while((I2CModule2[0x4/4] & 0x1) == 0x1); // wait while busy
        I2CModule2[0x0] = 0x8; // master code byte
        I2CModule2[0x4/4] = 0x13;
        while((I2C2->MCS & 1) == 1) {};
        I2CModule2[0x0] = addr; // address of slave DAC and transmit
        I2CModule2[0x8/4] = ((s&0xff00)>>8);
        I2CModule2[0x4/4] = 0x3;
        while((I2C2->MCS & 1) == 1) {};
        if((I2CModule2[0x4/4] & 0x2) == 0x2) // error
        {
                return;
```

```
        }
        I2CModule2[0x8/4] = (s & 0xff);
        I2CModule2[0x4/4] = 0x5;
        while((I2C2->MCS & 1) == 1) {};
        if((I2CModule2[0x4/4] & 0x2) == 0x2) // error
        {
                return;
        }
}

void setCS(unsigned char bool_val)
{
        if(bool_val)
        {
                portA[0x3fc] |= 0x4;
        }
        else
        {
                portA[0x3fc] &= ~0x4;
        }
}

unsigned char sendByteSSI(unsigned char val, unsigned char cs)
{
        while((SSI3t[0xc] & 0x2) == 0x0){}
        setCS(cs);
        SSI3t[0x8] = (short)val;
        while(SSI3->SR & 0x10) {}
        setCS(0x1);
        if((SSI3t[0xc] & 0x4) == 0) return 0x0;
        return ((unsigned short*)SSI3)[0x8/2];
}

unsigned char maskNumber(unsigned long long num, unsigned char b)
{
```

```
        if(b > 7) return 0x0;
        return (num >> (b<<3)) & 0xff;
}

unsigned char sendCmdToSD(unsigned char cmd, unsigned long arg, unsigned char crc)
{
        unsigned long long cmd_final = 0x0;
        unsigned int i = 0;
        unsigned char receive = 0xff;
        unsigned char retVal = 0xff;
        cmd |= 0x40;
        cmd_final |= cmd;
        cmd_final <<= 32;
        cmd_final |= arg;
        cmd_final <<= 8;
        crc = ((crc & 0x7f) << 1) | 0x1;
        cmd_final |= crc;
        sendByteSSI(maskNumber(cmd_final,5),0);
        sendByteSSI(maskNumber(cmd_final,4),0);
        sendByteSSI(maskNumber(cmd_final,3),0);
        sendByteSSI(maskNumber(cmd_final,2),0);
        sendByteSSI(maskNumber(cmd_final,1),0);
        sendByteSSI(maskNumber(cmd_final,0),0);
        for(i = 0; i < 8; ++i)
        {
                receive = sendByteSSI(0xff,0);
                if(receive != 0xff)
                {
                        retVal = receive;
                }
        }
        return retVal;
}

unsigned char getBlockFromSD(unsigned int addr, unsigned char* buf)
```

```
{
        unsigned int i = 0;
        unsigned int maxIter = 0;
        unsigned char resp = 0x0;
        resp = sendCmdToSD(17,addr,0x0);
        if(resp == 0x0)
        {
                while(sendByteSSI(0xff,0) != 0xFE && maxIter < MAX_ITER_WAIT_FOR_SD
                {
                        ++maxIter;
                }
                if(maxIter < MAX_ITER_WAIT_FOR_SD)
                {
                        for(i = 0; i < 512; ++i)
                        {
                                buf[i] = sendByteSSI(0xff,0);
                        }
                        return 0x1;
                }
        }
        return 0x0;
}
void sinWave(const unsigned int delay, const unsigned int cycles)
{
        unsigned int i = 0;
        unsigned int d = 0;
        unsigned int c = 0;
        for(c = 0; c < cycles; ++c)
        {
                for (i = 0; i < NUM_VALUES; ++i)
                {
                        send2BytesI2C(DACValues[i],0x62);
                        for(d = 0; d < delay; ++d);
                }
        }
```

```
}


void playWavFileAndSFX(unsigned int block_addr, unsigned int block_num, const unsig
{
        unsigned int i = 0;
        unsigned int j = 0;
        unsigned int k = 0;
        unsigned int failures = 0;
        unsigned char BUF[512];
        unsigned char resp = 0x0;
        for(i = 0; i < block_num; ++i)
        {
                resp = getBlockFromSD(block_addr + i, BUF);
                if(resp)
                {
                        for(j = 0; j < 512; ++j)
                        {
                                send2BytesI2C(BUF[j], 0x62);
                                for(k = 0; k < delay; ++k){}
                        }
                }
                else
                {
                        --i;
                        ++failures;
                }
                if(failures > 5)
                        break;
        }
}

void initPLL()
{
        SYSCTRL[0x060/4] |= 0x00000800; // set the bypass bit
```

```
        SYSCTRL[0x060/4] &= 0xFFBFFFFF; // clear the usesysdiv bit

        SYSCTRL[0x060/4] &= 0xFFFFF83F; // clears XTAL bits
        SYSCTRL[0x060/4] |= 0x00000540; // sets XTAL = 15 -> 16Mhz
        SYSCTRL[0x060/4] &= 0xFFFFFFCF; // sets OSCSRC to 00 => main Oscillator

        SYSCTRL[0x060/4] &= 0xFFFFDFFF; // clears PWRDN bit

        SYSCTRL[0x060/4] &= 0xF87FFFFF; // clears sysdiv
        SYSCTRL[0x060/4] |= 0x02000000; // sets sysdiv to 4 for a divisor of 5 -> 4

        SYSCTRL[0x060/4] |= 0x00400000; // sets usesysdiv bit

        while((SYSCTRL[0x050/4] & 0x40) == 0); //poll pllris bit

        SYSCTRL[0x060/4] &= 0xFFFFF7FF; // clears BYPASS bit
}

void initI2C()
{
        SYSCTRL[0x620/4] |= 0x4; // enable clock to i2c module 0
        SYSCTRL[0x608/4] |= 0x10; // enable clock to gpio port e
        portE[0x420] |= 0x30; // enable alt function pe4 and pe5
        portE[0x50c] &= ~0x30;
        portE[0x50c] |= 0x20; // pe5 as open drain
        portE[0x510] |= 0x10; // pe4 as pull-up
        portE[0x51c] |= 0x30; // pe4 and pe5 digital enable
        ((unsigned int*)portE)[0x52c/4] &= ~0xff0000; // clear pmx digital function
        ((unsigned int*)portE)[0x52c/4] |= 0x330000; // 3 is i2c code for pins 2 an
        I2CModule2[0x20/4] = 0x00000010; // configure as master
        I2CModule2[0xc/4] = 0x00000001; // high speed mode and 1 for 3.3 MHz clck f
}

void initInterrupts()
{
```

```
        SYSCTRL[0x608/4] |= 0x4; // enable clock on Port C
        __nop();
        __nop();
        __nop();
        portC[0x400] &= ~0x30; // gpiodir pb0 and pb1 are input
        portC[0x420] &= ~0x30; // set alt function, register AFSEL pd0 - pd3
        portC[0x51c] |= 0x30; // enable digital enable, register GPIODEN

        portC[0x410] = 0x0; // disable interrupts
        portC[0x408] &= ~0x30; // give control to interrupt event reg
        portC[0x404] &= ~0x30; // edge sensitive
        portC[0x40C] |= 0x30; // rising edge triggers interrupt
        portC[0x414] = 0x0; // clear GPIORIS register for safety
        portC[0x410] = 0x30; // enable interrupt on pc4 and pc5

// enable port interrupt
        INTERRUPT[0x100/4] |= 0x4;
        __nop();
        __nop();
}

void GPIOC_Handler(void)
{
        sinWave(2000,100);

                while (GPIOC->MIS != 0) {
                        GPIOC->ICR = 0xFF;
                }
}

void initSSI3()
{
        unsigned int i = 0;
        SYSCTRL[0x61C/4] |= 0x8; // enable SSIO Module 3, register RCGCSSI
        SYSCTRL[0x608/4] |= 0x9; // enable clock on Port D and A
```

```
        __nop();
        __nop();
        portA[0x400] |= 0x4; // gpiodir pa2 is output
        portA[0x420] &= ~0x4; // set alt function, register AFSEL pd0 - pd3
        portA[0x51c] |= 0x4; // enable digital enable, register GPIODEN
        setCS(0x1);
        portD[0x420] = 0xf; // set alt function, register AFSEL pd0 - pd3
        ((unsigned int*)portD)[0x52c/4] &= ~0xffff; // pmux function
        ((unsigned int*)portD)[0x52c/4] |= 0x1111; // pmux function
        portD[0x51c] |= 0xf; // enable digital enable, register GPIODEN
        SSI3t[0x4] &= ~0x1f; // clear SSE bit in SSICR1 register, disable SSIO
        // This also sets SSI as master
        SSI3t[0xFC8] &= ~0xf; // System clock in register SSICC
        SSI3t[0x010] = 0x64; // 400kHz
        ((unsigned short *)SSI3)[0x0] = 0x7; // 8 bit frame
        SSI3t[0x004] |= 0x2; // enable SSI by setting SSe bit in SSICR1 register
        for(i = 0; i < 10; ++i)
        {
                sendByteSSI(0xFF, 0x1);
        }
}
void initSD()
{
        sendCmdToSD(0x0,0x0,0x4a);
        sendCmdToSD(0x8,0x1aa,0x43);
        sendCmdToSD(0x55,0x0,0x0);
        while(1)
        {
                if(sendCmdToSD(0x41,0x40000000,0x0) == 0x0)
                {
                        break;
                }
        }
        SSI3t[0x004] &= ~0x2; // disable SSI by clearing SSe bit in SSICR1
        SSI3t[0x010] = 0x2; // 20MHz
```

```
        SSI3t[0x004] |= 0x2; // enable SSI by setting SSe bit in SSICR1
        __nop();
        __nop();
}
```