EE445L – Lab 5: Music Player and Audio Amp

Michael Park and Jack Zhao
03/07/16

1.0 OBJECTIVE
Requirements document
1. Overview
1.1. Objectives: Why are we doing this project? What is the purpose?

The objectives of this project are to design, build and test a music player. Educationally, students are learning how to interface a DAC, how to design a speaker amplifier, how to store digital music in ROM, and how to perform DAC output in the background. Our goal is to play Phantom of the Opera.

1.2. Process: How will the project be developed?

The project will be developed using the TM4C123 board. There will be three switches that the operator will use to control the music player. The system will be built on a solderless breadboard and run on the usual USB power. The system will use three off-board switches. A hardware/software interface will be designed that allows software to control the player. There will be at least three hardware/software modules: switch input, DAC output, and the music player. The process will be to design and test each module independently from the other modules. After each module is tested, the system will be built and tested.

1.3. Roles and Responsibilities: Who will do what? Who are the clients?

EE445L students are the engineers and the TA is the client. Michael will design and implement hardware. He will also write basic software modules to test the hardware. Jack will deisgn and implement the software application.

1.4. Interactions with Existing Systems: How will it fit in?

The system will use the TM4C123 board, a solderless breadboard, and the speaker. It will be powered using the USB cable. You may use a +5V power from the lab bench.

1.5. Terminology: Define terms used in the document.

SSI - Synchronous Serial Interface: A device to transmit data with synchronous serial communication protocol.
Linearity - A measure of the straightness of the static calibration curve.
Frequency response – A standard technique to describe the dynamic behavior of linear systems
Loudness – Determined by amplitude of a wave.
Pitch – Determined by frequency of a wave.
Instrument – A embedded system that collects information, same as data acquisition system.
Tempo – Defined by the speed of music.
Envelope – Defines the amplitude vs time.
Melody - Linear succession of musical notes and tones and is a combination of pitch and rhythm
Harmony - The use of simultaneous tones, notes or chords and is referred to as the ‘vertical’ aspect of music

1.6. Security: How will intellectual property be managed?

The system may include software from StellarisWare and from the book. No software written for this project may be transmitted, viewed, or communicated with any other EE445L student past, present, or future (other than the lab partner of course). It is the responsibility of the team to keep its EE445L lab solutions secure.

2. Function Description

2.1. Functionality: What will the system do precisely?

If the operator presses the play/pause button the music will play or pause. If the operator presses the play/pause button once the music should pause. Hitting the play/pause again causes music to continue. The play/pause button does not restart from the beginning, rather it continues from the position it was paused. If the rewind button is pressed, the music stops and the next play operation will start from the beginning. There is a mode switch that plays a horn sound.

There must be a C data structure to hold the music. There must be a music driver that plays songs. The length of the song should be at least 30 seconds and comprise of at least 8 different frequencies. Although you will be playing only one song, the song data itself will be stored in a separate place and be easy to change. The player runs in the background using interrupts. The foreground (main) initializes the player, then executes **for(;;){}** do nothing loop. If you wish to include LCD output, this output should occur in the foreground. The maximum time to execute one instance of the ISR is xxxx. You will need public functions **Rewind**, **Play** and **Stop**, which perform operations like a cassette tape player. The **Play** function has an input parameter that defines the song to play. A background thread implemented with output compare will fetch data out of your music structure and send them to the DAC.

There must be a C data structure to store the sound waveform or instrument. You are free to design your own format, as long as it uses a formal data structure (i.e., **struct**). The generated music must sound beautiful utilizing the SNR of the DAC. Although you only have to implement one instrument, it should be easy to change instruments.

2.2. Scope: List the phases and what will be delivered in each phase.

Phase 1 is the preparation; phase 2 is the demonstration; and phase 3 is the lab report. Details can be found in the lab manual.

2.3. Prototypes: How will intermediate progress be demonstrated?

A prototype system running on the TM4C123 board and solderless breadboard will be demonstrated. Progress will be judged by the preparation, demonstration and lab report.

2.4. Performance: Define the measures and describe how they will be determined.

The system will be judged by three qualitative measures. First, the software modules must be easy to understand and well-organized. Second, the system must employ an abstract data structures to hold the sound and the music. There should be a clear and obvious translation from sheet music to the data structure. Backward jumps in the ISR are not allowed. Waiting for SSI output to complete is an acceptable backwards jump. Third, all software will be judged according to style guidelines. Software must follow the style described in Section 3.3 of the book. There are three quantitative measures. First, the SNR of the DAC output of a sine wave should be measured. Second, the maximum time to run one instance of the ISR will be recorded. Third, you will measure power supply current to run the system. There is no particular need to optimize any of these quantitative measures in this system.

2.5. Usability: Describe the interfaces. Be quantitative if possible.

There will be three switch inputs. The DAC will be interfaced to a 32-ohm speaker. Between the Speaker and the DAC, there will be an audio amplifier with gain of 1. Switch will use negative logic. DAC will be interfaced with TM4C123 using SSI. DAC will use approximately an 1.5V reference.

2.6. Safety: Explain any safety requirements and how they will be measured.

Will use speaker.

3. Deliverables

3.1. Reports: How will the system be described?
A lab report described below is due by the due date listed in the syllabus. This report includes the final requirements document.

3.2. Audits: How will the clients evaluate progress?
The preparation is due at the beginning of the lab period on the date listed in the syllabus.

3.3. Outcomes: What are the deliverables? How do we know when it is done?
There are three deliverables: preparation, demonstration, and report.

2.0 HARDWARE DESIGN

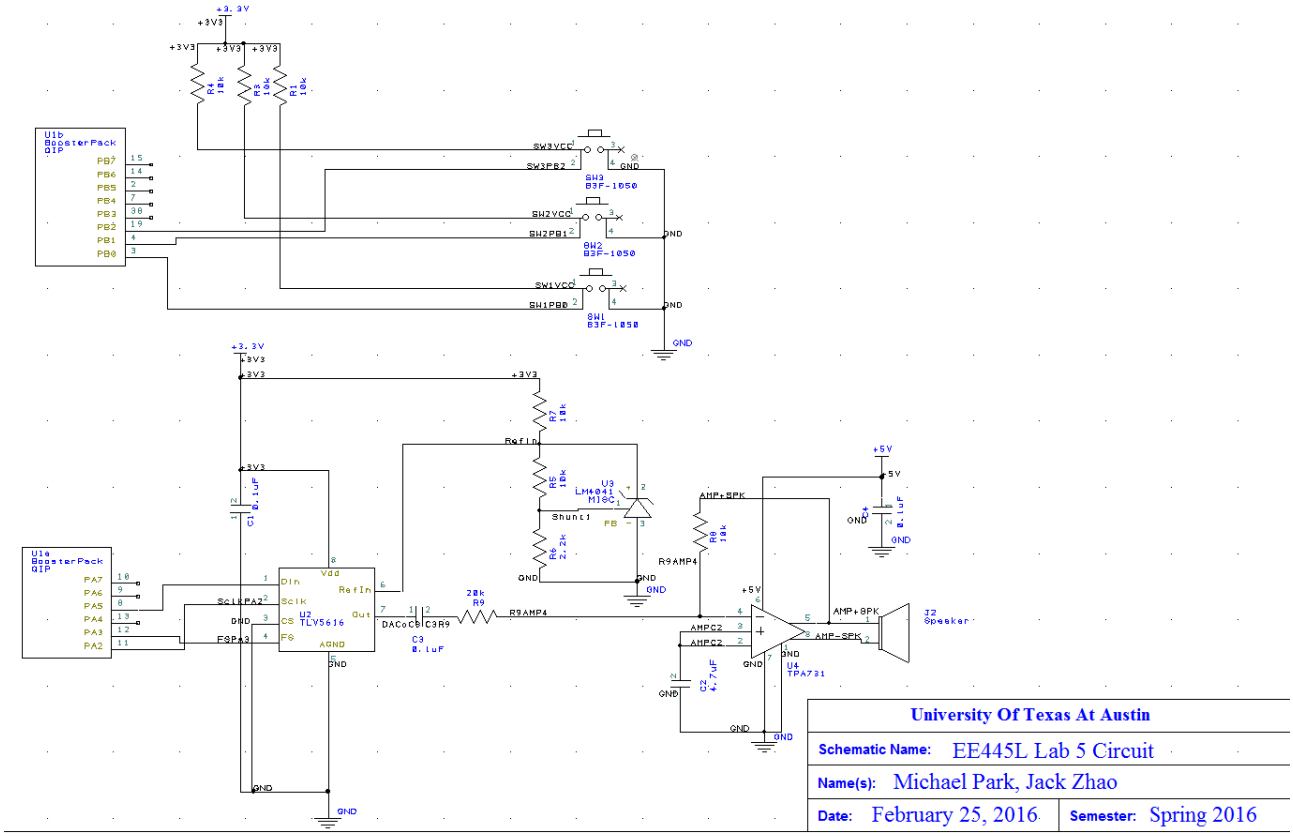

Figure 1: Lab5 hardware schematic

3.0 SOFTWARE DESIGN

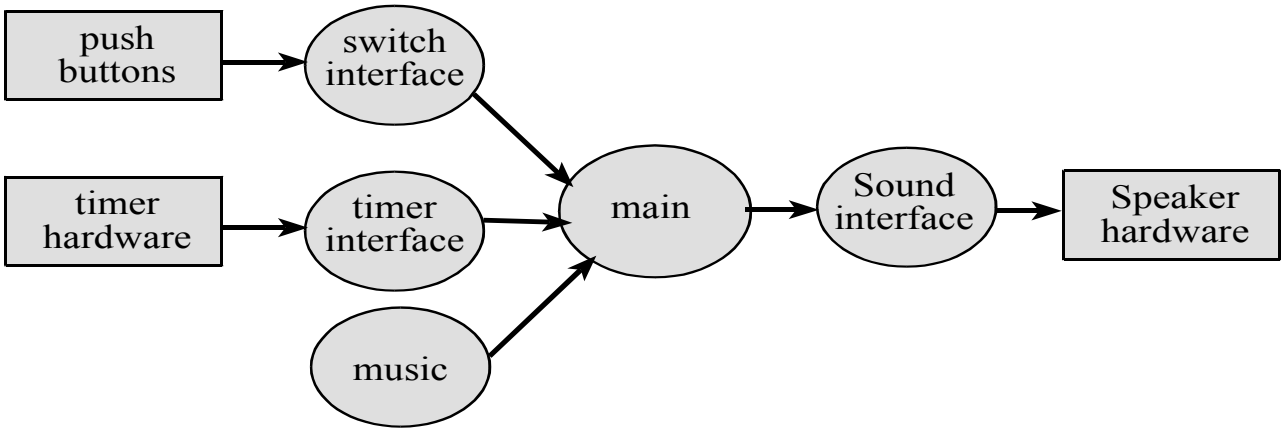

Figure 2: Data flows from the memory and the switches to the speaker

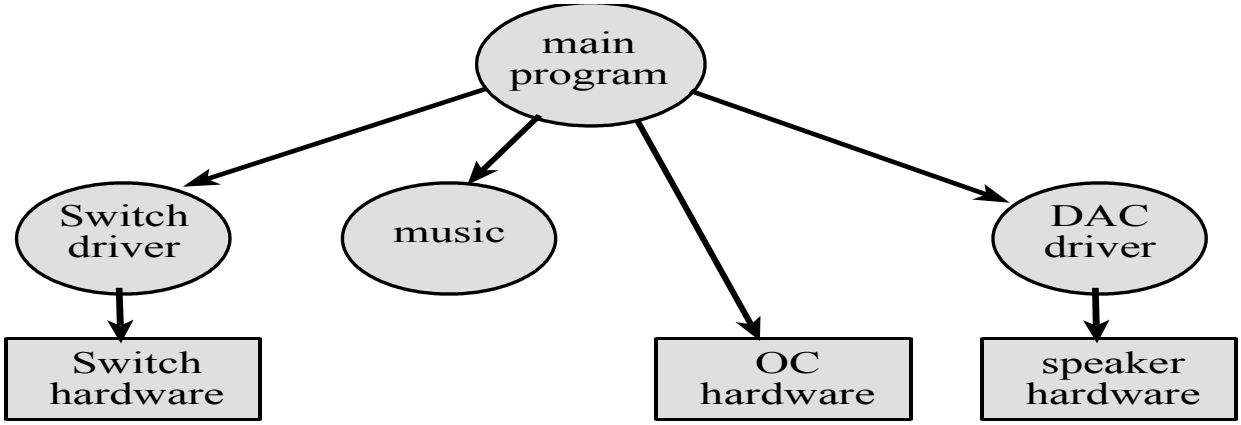

Figure 3: A call graph showing the three modules used by the music player.

4.0 MEASUREMENT DATA

(i) DAC output versus digital input for 8 different digital inputs. Compare the measured data with the expected values. Calculate resolution, range, precision and accuracy of the DAC.

(Measured Output)
intput to DAC - Output Voltage
348 - 0.295 V
261 - 0.221 V
400 - 0.335 V
440 - 0.373 V
522 - 0.443 V
660 - 0.560 V
742 - 0.630 V
880 - 0.747 V

(Expected Output)
intput to DAC - Output Voltage
348 - 0.280 V
261 - 0.210 V
400 - 0.322 V
440 - 0.355 V
522 - 0.421 V
660 - 0.532 V

742 - 0.598 V
880 - 0.709 V

DAC
range: 0 – 3.3V
precision: 12bit (0-4095)
resolution: 3.3/4095 = 0.0008058608
accuracy: (0.295 – 0.28)/0.295 = 0.05 -> 5% error. Thus 95% accurate

(ii) Using an oscilloscope and spectrum analyzer, measure the time-domain and frequency-domain outputs from your system at one frequency, like Figure 8.35 in the textbook. Using the spectrum, calculate SNR (ratio of the sinewave output to the largest noise component).

Figure 4: time-domain

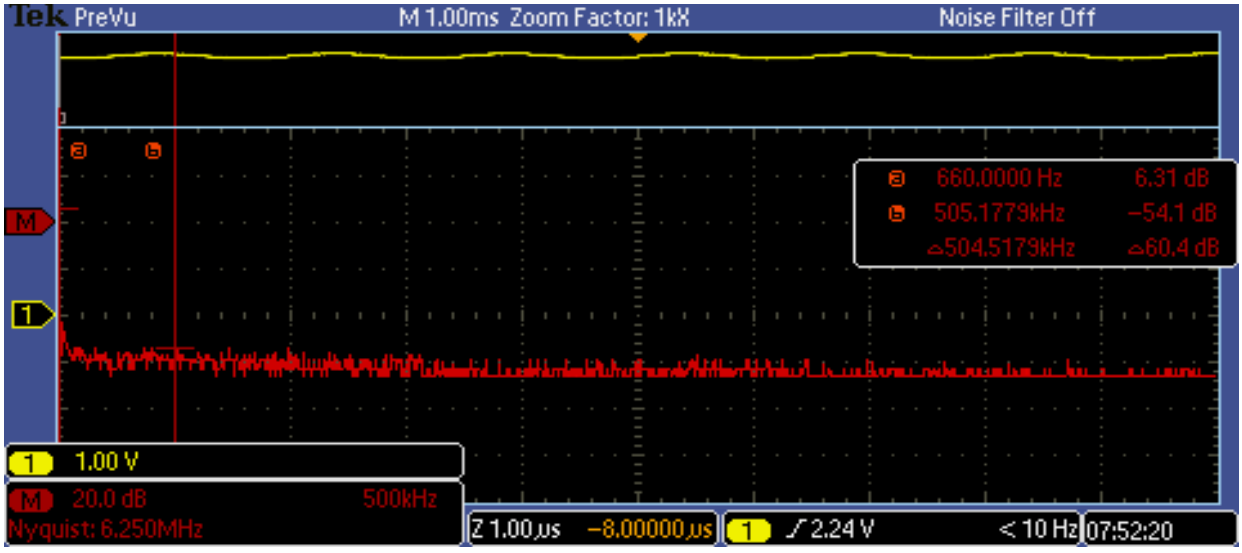

Figure 5: frequency-domain

SNR = 6.31dB - (-54.1dB) = 6.31 + 54.1dB = 60.41 dB

(iii) Using debugging instruments, measure the maximum time required to execute the periodic interrupt service routines. In particular, create a debugging profile to measure the percentage processor time required to play the song.

Interrupts = 95 cycles

Main = 271 cycles

95/271 = 0.35; Percentage of time required to play song = 35%

(iv) Measurements of current required to run the system, with and without the music playing

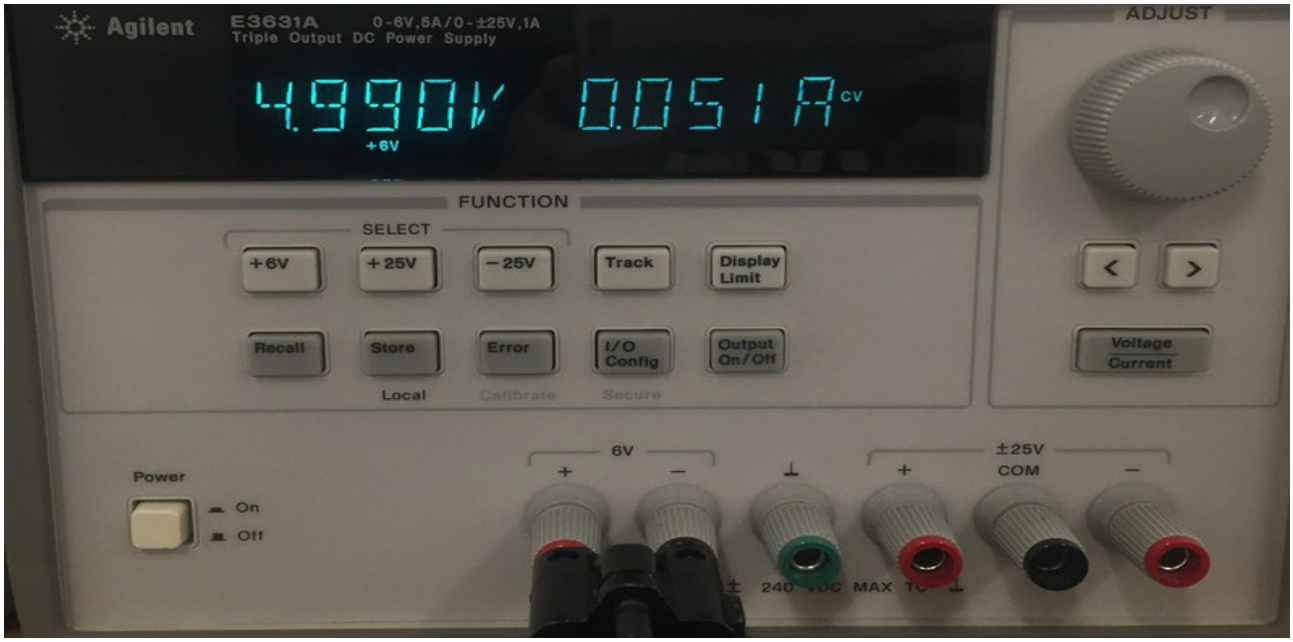

Figure 6: current required without the music playing

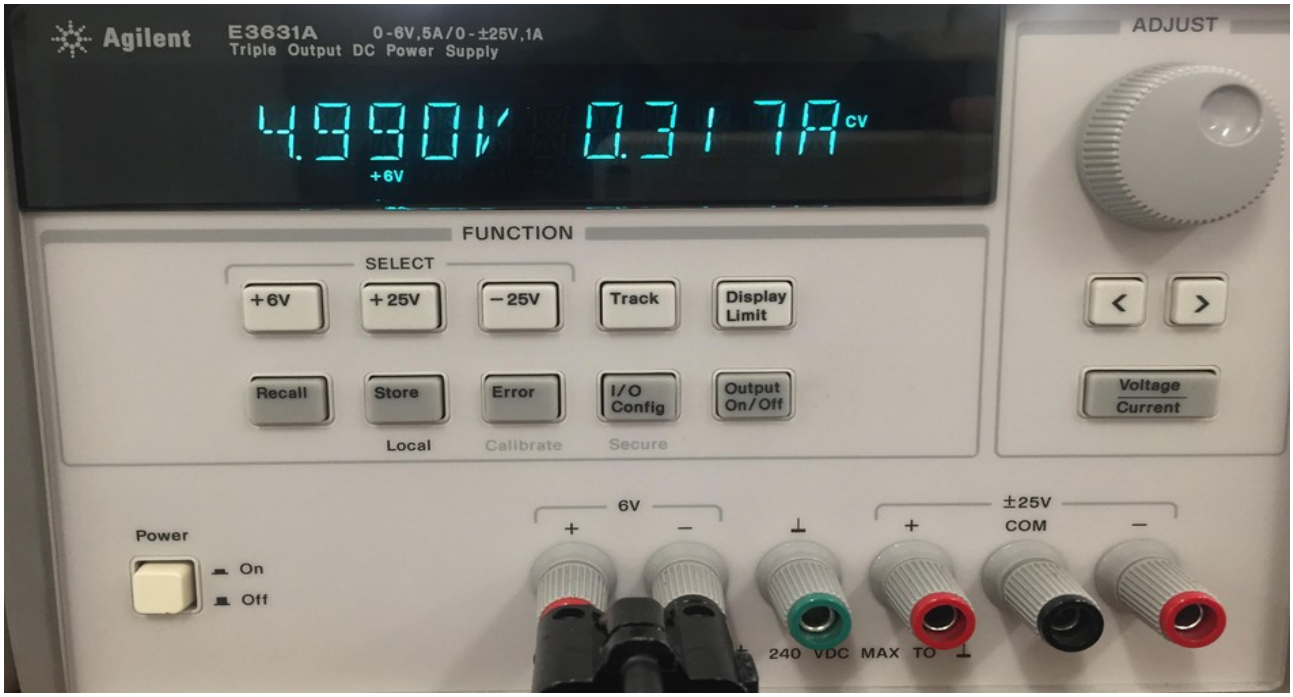

Figure 7: current required with the music playing

5.0 ANALYSIS AND DISCUSSION

1) Briefly describe three errors in a DAC.

(i) DAC gain error- A DAC gain error is a shift in the slope of the Vout versus the digital input static response. This will affect the tuning of our notes.

(ii) DAC offset error- A DAC offset error is a shift in the Vout verses digital input static response. This affects the initial values of Vout and affects the range.

(iii) DAC linearity error- A DAC linearity error is the maximum deviation at any point in the transfer function, of the output voltage level to the ideal voltage level. This means that our outputs will not be shaped perfectly linearly which will affect the shape of our sine wave

2) Calculate the data available and data required intervals in the SSI/DAC interface. Use these calculations to justify your choice of SSI frequency.

Data Required: $(cp - tsu, cp + th) = (25-8, 25+5) = (17, 30)ns$

For this lab which uses the TM4C123 and TLV5616, the maximum baud rate of the TM4C123 (initialized to 50 MHz) gives a setup and hold time (based on the baud rate, 1 system bus period and 2 system bus periods respectively) above the minimum for the DAC chip(8MHz). Therefore, we can use the maximum baud rate without any trouble.

3) How is the frequency range of a spectrum analyzer determined?

The frequency range of the spectrum analyzer is determined through a process of mapping out differences in dB with frequncy. The graph is created through a process of fourier transform which will convert data in the time domain to the frequency domain. In terms of our spectrum analyzer, it uses a FFT which is a fast fourier transform to map out time domain into the frequency domain.

4) Why did we not simply drive the speaker directly from the DAC? I.e., what purpose is the TPA731?

The TPA731 is an audio amplifier. The reason we cannot drive the speaker directly from the DAC is that there is not enough current outputted from the DAC to actually drive the speaker to emit a sound. Once we implement the Audio Amp circuit, the current that will pass into the speaker input is increaesed enough to power the speaker.

6.0 CODE WRITTEN FOR THIS LAB

```
// Main.c
// Michael Park, Jack Zhao
// Date Created: 02/24/2016
// initialize pll, heartbeat, switch, timer
// Lab Number: 16340
// TA: Mahesh Srinivasan
// Last Revised: 02/29/2016

#include <stdint.h>
#include <stdio.h>
#include "DAC.h"
#include "SysTickInts.h"
#include "PLL.h"
#include "Music.h"
#include "Switch.h"
#include "../Shared/tm4c123gh6pm.h"
#include "Timer0A.h"
#include "Timer1.h"
#include "Timer2.h"


#define PF1       (*((volatile uint32_t *)0x40025008))
#define PF2       (*((volatile uint32_t *)0x40025010))
#define PF3       (*((volatile uint32_t *)0x40025020))
#define LEDS      (*((volatile uint32_t *)0x40025038))

extern const uint32_t Song2[128];
extern const uint32_t Song[128];
extern const uint32_t Song3[128];

void DisableInterrupts(void); // Disable interrupts
void EnableInterrupts(void);  // Enable interrupts
long StartCritical (void);    // previous I bit, disable interrupts
void EndCritical(long sr);    // restore I bit to previous value
void WaitForInterrupt(void);  // low power mode
void PORTF_Init(void);

volatile uint16_t stop = 0;
volatile uint16_t changeSound = 0;
extern volatile uint32_t note;


int main(void)
{
        PLL_Init(Bus50MHz);             // bus clock at 50 MHz
        PORTF_Init();
        Switch_Init();
        SysTick_Init(50000000);                                         //Tempo set to 1 bit/sec
        Timer0A_Init(&PlaySong, (Song3[note] / 32));  // initialize timer0A to 440 Hz(A4) * 32(Size of Wave)
        Timer1_Init(&PlaySong2, (Song2[note] / 32));
        Timer2_Init(&addWaves, 500);

        DAC_Init(2048);
        Music_Init();
        //stop=1;
        EnableInterrupts();
        while(1)
        {
                WaitForInterrupt();
        }
}
```

```
//Initialize Port F
void PORTF_Init(void)
{
	SYSCTL_RCGCGPIO_R |= 0x20;	// activate port F
	while((SYSCTL_PRGPIO_R&0x0020) == 0){ };// ready?
	GPIO_PORTF_DIR_R |= 0x0E;	// make PF3-1 output (PF3-1 built-in LEDs)
	GPIO_PORTF_AFSEL_R &= ~0x0E;	// disable alt funct on PF3-1
	GPIO_PORTF_DEN_R |= 0x0E;	// enable digital I/O on PF3-1

					// configure PF3-1 as GPIO
	GPIO_PORTF_PCTL_R = (GPIO_PORTF_PCTL_R&0xFFFFF0FF)+0x00000000;
	GPIO_PORTF_AMSEL_R = 0;		// disable analog functionality on PF
	LEDS = 0;			// turn all LEDs off
}

// Music.c
// Michael Park, Jack Zhao
// Date Created: 02/24/2016
// Includes music data and timer invoked task functions
// Lab Number: 16340
// TA: Mahesh Srinivasan
// Last Revised: 02/29/2016

#include <stdint.h>
#include "DAC.h"
#include "Music.h"


#define PF1	(*((volatile uint32_t *)0x40025008))
#define PF2	(*((volatile uint32_t *)0x40025010))
#define PF3	(*((volatile uint32_t *)0x40025020))
#define LEDS	(*((volatile uint32_t *)0x40025038))

void DisableInterrupts(void); // Disable interrupts
void EnableInterrupts(void);  // Enable interrupts
long StartCritical (void);    // previous I bit, disable interrupts
void EndCritical(long sr);    // restore I bit to previous value
void WaitForInterrupt(void);  // low power mode


const uint16_t Wave[32] =
{1024,1124,1218,1304,1380,1443,1488,1516,1526,1516,1488,1442,1380,1304,1218,1124,1026,928,835,748,672,610
,564,536,526,536,564,610,672,748,835,928};
const uint16_t Horn[32] =
{1065,1084,1122,1277,1680,1750,1277,757,663,663,705,734,770,847,1040,1136,1211,1333,1467,1547,1429,1589,1
372,1088,709,520,450,492,568,685,803,994};


const uint32_t Song3[128] ={0, 0, A4, A4, E5, E5, A4, A4,
E5, E5, E5, D5, D5, D5, D5, D5,
D5, D5, B4, B4, D5, D5, D5, B4,
A4, A4, A4, A4, A4, A4, A4, A4,
A4, A4, E4, E4, E4, E4, A4, A4,
A4, A4, A4, B4, B4, B4, B4, B4,
B4, B4, B4, B4, D5, D5, B4, B4,
A4, A4, A4, A4, A4, A4, A4, A4,

A4, A4, A4, A4, A4, A4, A4, A4,
E5, E5, E5, E5, E5, E5, E5, E5,
E5, D5, D5, A4, B4, A4, G4, E4,
D4, D4, D4, D4, D4, D4, D4, D4,
D4, D4, D4, D4, D4, D4, D4, C4,
C4, C4, C4, C4, C4, C4, C4, C4,
```

```
0, 0, 0, 0, 0, 0, 0, 0, };

const uint32_t Song2[128]={0, 0, E5, E5, A5, A5, E5, E5,
G5, G5, G5, F5, F5, F5, F5, F5,
F5, F5, D5, D5, G5, G5, G5, D5,
E5, E5, E5, E5, E5, E5, E5, E5,
E5, E5, A4, A4, A4, A4, C5, C5,
E5, E5, E5, D5, D5, D5, D5, D5,
D5, D5, D5, D5, G5, G5, D5, D5,
E5, E5, E5, E5, E5, E5, E5, E5,
E5, E5, E5, E5, E5, E5, E5, E5,
A5, A5, A5, A5, A5, A5, A5, A5,
A5, G5, F5, E5, D5, C5, B4, A4,
Gs4, Gs4, Gs4, Gs4, Gs4, Gs4, Gs4, Gs4,
Gs4, Gs4, F4, F4, F4, F4, F4, E4,
E4, E4, E4, E4, E4, E4, E4, E4,
0, 0, 0, 0, 0, 0, 0, 0, };


volatile uint32_t note = 0;
volatile Music current_song;
unsigned char soundIndex = 0; //varies from 0 to 32
unsigned char soundIndex2 = 0; //varies from 0 to 32
extern volatile uint16_t stop;
extern volatile uint16_t changeSound;
volatile uint32_t addedWaves;
volatile uint32_t addedWaves2;
volatile uint32_t refresh =0;

void Music_Init(void){
        current_song.length=128;
        for(int i =0; i <current_song.length; i++){
                current_song.Song[i]=Song3[i];
        }
}


void PlaySong(void)
{

        if(!stop)
        {
                        refresh++;
                PF1 ^= 0x02;
                soundIndex += 1;
                if(soundIndex > 31)
                {
                        soundIndex = 0;
                }
                if(changeSound == 0)
                {
                        //DAC_Out(Wave[soundIndex]);//
                }
                else if(changeSound==1){
                        DAC_Out(Horn[soundIndex]);
                }
                PF1 ^= 0x02;
        }
}

void PlaySong2(void)
{
        //refresh+=1;
```

```
	if(!stop)
	{
		PF1 ^= 0x02;
		soundIndex2 += 1;
		if(soundIndex2 > 31)
		{
			soundIndex2 = 0;
		}
		if(changeSound == 0)
		{
			//DAC_Out(Wave[soundIndex2]);
		}
		else if(changeSound==1){
			DAC_Out(Horn[soundIndex2]);
		}
		PF1 ^= 0x02;
	}
}


void addWaves(void){
long sr=StartCritical();

	addedWaves=Wave[soundIndex]+Wave[soundIndex2];//
	addedWaves=(addedWaves)/2;//
//	if(addedWaves<1024)
//	{
//		addedWaves+=refresh;
//	}
//	else{
//		addedWaves-=refresh;
//	}
	addedWaves2=Horn[soundIndex]+Horn[soundIndex2];
	addedWaves2=addedWaves2/2;


	if(changeSound == 0)
	{
			DAC_Out(addedWaves);
	}
	else if(changeSound==1){
			DAC_Out(addedWaves2);
	}
	EndCritical(sr);
}

// Switch.c
// Runs on LM4F120/TM4C123
// Provide functions that initialize a GPIO as an input pin and
// allow reading of two negative logic switches on PF0 and PF4
// and an external switch on PA5.
// Use bit-banded I/O.
// Daniel and Jonathan Valvano
// September 12, 2013

// Switch.c
// Michael Park, Jack Zhao
// Date Modified: 02/24/2016
// PORTB switch init and handler
// Lab Number: 16340
// TA: Mahesh Srinivasan
// Last Revised: 02/29/2016
```

```
/* This example accompanies the book
   "Embedded Systems: Introduction to ARM Cortex M Microcontrollers",
   ISBN: 978-1469998749, Jonathan Valvano, copyright (c) 2015
   Section 4.2    Program 4.2

  "Embedded Systems: Real Time Interfacing to ARM Cortex M Microcontrollers",
   ISBN: 978-1463590154, Jonathan Valvano, copyright (c) 2015
   Example 2.3, Program 2.9, Figure 2.36

 Copyright 2015 by Jonathan W. Valvano, valvano@mail.utexas.edu
    You may use, edit, run or distribute this file
    as long as the above copyright notice remains
 THIS SOFTWARE IS PROVIDED "AS IS".  NO WARRANTIES, WHETHER EXPRESS, IMPLIED
 OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
 MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE.
 VALVANO SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL,
 OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
 For more information about my classes, my research, and my books, see
 http://users.ece.utexas.edu/~valvano/
 */

// negative logic switches connected to PF0 and PF4 on the Launchpad
// red LED connected to PF1 on the Launchpad
// blue LED connected to PF2 on the Launchpad
// green LED connected to PF3 on the Launchpad
// NOTE: The NMI (non-maskable interrupt) is on PF0.  That means that
// the Alternate Function Select, Pull-Up Resistor, Pull-Down Resistor,
// and Digital Enable are all locked for PF0 until a value of 0x4C4F434B
// is written to the Port F GPIO Lock Register.  After Port F is
// unlocked, bit 0 of the Port F GPIO Commit Register must be set to
// allow access to PF0's control registers.  On the LM4F120, the other
// bits of the Port F GPIO Commit Register are hard-wired to 1, meaning
// that the rest of Port F can always be freely re-configured at any
// time.  Requiring this procedure makes it unlikely to accidentally
// re-configure the JTAG and NMI pins as GPIO, which can lock the
// debugger out of the processor and make it permanently unable to be
// debugged or re-programmed.
#include <stdint.h>
#include "../Shared/tm4c123gh6pm.h"
#include "SysTick.h"

#define GPIO_LOCK_KEY           0x4C4F434B  // Unlocks the GPIO_CR register
#define PF0                     (*((volatile uint32_t *)0x40025004))
#define PF4                     (*((volatile uint32_t *)0x40025040))
#define PA5                     (*((volatile uint32_t *)0x40004080))
#define SWITCHES                (*((volatile uint32_t *)0x40025044))
#define SW1       0x10                  // on the left side of the Launchpad board
#define SW2       0x01                  // on the right side of the Launchpad board
#define SYSCTL_RCGC2_GPIOF      0x00000020  // port F Clock Gating Control

#define PB0 (*((volatile uint32_t *)0x40005004))
#define PB1 (*((volatile uint32_t *)0x40005008))
#define PB2 (*((volatile uint32_t *)0x40005010))
#define PB3 (*((volatile uint32_t *)0x40005020))

extern volatile uint32_t isPlaying;
extern volatile uint32_t isFast;


void DisableInterrupts(void); // Disable interrupts
void EnableInterrupts(void);  // Enable interrupts
long StartCritical (void);    // previous I bit, disable interrupts
void EndCritical(long sr);    // restore I bit to previous value
```

```
void WaitForInterrupt(void);  // low power mode


extern volatile uint16_t stop;
extern volatile uint16_t changeSound;
extern volatile uint32_t note;
extern unsigned char soundIndex;


//------------Switch_Init------------
// Initialize GPIO Port B bit 0-2 for input
// Input: none
// Output: none
void Switch_Init(void){
  SYSCTL_RCGCGPIO_R |= 0x02;        // 1) activate clock for Port B
  //while((SYSCTL_PRGPIO_R&0x02) == 0){};// ready?

            GPIO_PORTB_DIR_R &= ~0x07;        // PB0-2 is an input
  //GPIO_PORTB_AFSEL_R &= ~0x0F;      // regular port function
  GPIO_PORTB_AMSEL_R &= ~0x07;      // disable analog on PB0-2
  //GPIO_PORTB_PCTL_R &= ~0x0000FFFF; // PCTL GPIO on PB1
  GPIO_PORTB_DEN_R |= 0x07;         // PB2-0 enabled as a digital port
            GPIO_PORTB_IS_R &= ~0x07;                                       // PB 0-2 is edge-sensitive
            GPIO_PORTB_IBE_R &= ~0x07;                                      // PB 0-2 is not both edges
            GPIO_PORTB_IEV_R &= ~0x07;                                      // PB 0-2 falling edge event
            GPIO_PORTB_ICR_R = 0x07;                                        // clear flag 0-2
            GPIO_PORTB_IM_R |= 0x07;                                        // arm interrupt on PB 0-2
            //NVIC_PRI0_R = (NVIC_PRI0_R&0xFF00FFFF)|0x00A00000; // (5) priority 5
            NVIC_EN0_R = 0x00000002;                                        //enable interrupt 1(PB) in
NVIC
}
//------------Switch_Input------------
// Read and return the status of GPIO Port A bit 5
// Input: none
// Output: 0x20 if PA5 is high
//         0x00 if PA5 is low
uint32_t Switch_Input(void){
  return PA5; // return 0x20(pressed) or 0(not pressed)
}

#define DELAY10MS 160000
#define DELAY10US 160
//------------Switch_Debounce------------
// Read and return the status of the switch
// Input: none
// Output: 0x02 if PB1 is high
//         0x00 if PB1 is low
// debounces switch
uint32_t Switch_Debounce(void){
uint32_t in,old,time;
  time = 1000; // 10 ms
  old = Switch_Input();
  while(time){
    SysTick_Wait(DELAY10US); // 10us
    in = Switch_Input();
    if(in == old){
      time--; // same value
    }else{
      time = 1000;  // different
      old = in;
    }
  }
  return old;
```

```
}

/*
GIPOPortB_Handler
ISR for Switch interface: PB0-2
Input: None
Output: None
*/
void GPIOPortB_Handler(void)
{
	if (GPIO_PORTB_RIS_R & 0X01) //poll PB0
	{
		GPIO_PORTB_ICR_R = 0x01; //acknowledge flag1 and clear
		if(stop==0){
			stop =1;
		}
		else{
			stop = 0;                                                    //PLAY
		}

	}
	if (GPIO_PORTB_RIS_R & 0X02) //poll PB1
	{
		GPIO_PORTB_ICR_R = 0x02; //acknowledge flag1 and clear
		long sr= StartCritical();
		stop =1;
		note=0;
		soundIndex = 0;
		EndCritical(sr);

	}
	if (GPIO_PORTB_RIS_R & 0X04) //poll PB2
	{
		GPIO_PORTB_ICR_R = 0x04; //acknowledge flag1 and clear
		//SysTick_Init(5000000);                                    //Tempo set to 1 bit/sec
		if(changeSound==0){
			changeSound =1;
		}
		else{
			changeSound = 0;
	//PLAY
		}
	}
}



// MAX549.c
// Runs on LM4F120/TM4C123
// Use SSI0 to send a 16-bit code to the MAX539 and return
// the reply.
// Daniel Valvano
// September 11, 2013

// DAC.c
// Michael Park, Jack Zhao
// Date Created: 02/24/2016
// 16bit DAC init and output
// Lab Number: 16340
// TA: Mahesh Srinivasan
// Last Revised: 02/29/2016

/* This example accompanies the book
```

*/

```
// Max549 pin 1 ground
// Max549 pin 2 OUTA
// Max549 pin 3 CS, SSI0Fss connected to PA3
// Max549 pin 4 DIN, SSI0Tx connected to PA5
// Max549 pin 5 SCLK SSI0Clk connected to PA2
// Max549 pin 6 OUTB
// Max549 pin 7 REF (cheap solution connects this to +3.3V)
// Max549 pin 8 +3.3V
#include <stdint.h>
#include "../Shared/tm4c123gh6pm.h"

#define SSI_CR0_SCR_M           0x0000FF00  // SSI Serial Clock Rate
#define SSI_CR0_SPH             0x00000080  // SSI Serial Clock Phase
#define SSI_CR0_SPO             0x00000040  // SSI Serial Clock Polarity
#define SSI_CR0_FRF_M           0x00000030  // SSI Frame Format Select
#define SSI_CR0_FRF_MOTO        0x00000000  // Freescale SPI Frame Format
#define SSI_CR0_DSS_M           0x0000000F  // SSI Data Size Select
#define SSI_CR0_DSS_16          0x0000000F  // 16-bit data
#define SSI_CR1_MS              0x00000004  // SSI Master/Slave Select
#define SSI_CR1_SSE             0x00000002  // SSI Synchronous Serial Port
#define SSI_SR_RNE              0x00000004  // SSI Receive FIFO Not Empty
#define SSI_SR_TNF              0x00000002  // SSI Transmit FIFO Not Full
                                // Enable
#define SSI_CPSR_CPSDVSR_M      0x000000FF  // SSI Clock Prescale Divisor

long StartCritical (void);    // previous I bit, disable interrupts
void EndCritical(long sr);    // restore I bit to previous value


//********DAC_Init*****************
// Initialize Max539 dual 16-bit DAC
// inputs:  initial voltage output (0 to 4096)
// outputs: none
// assumes: system clock rate less than 50 MHz
void DAC_Init(uint16_t data)
{
          SYSCTL_RCGCSSI_R |= 0x01;       // activate SSI0
  SYSCTL_RCGCGPIO_R |= 0x01;      // activate port A
  while((SYSCTL_PRGPIO_R&0x01) == 0){};// ready?
  GPIO_PORTA_AFSEL_R |= 0x2C;     // enable alt funct on PA2,3,5
  GPIO_PORTA_DEN_R |= 0x2C;       // configure PA2,3,5 as SSI
  GPIO_PORTA_PCTL_R = (GPIO_PORTA_PCTL_R&0xFF0F00FF)+0x00202200;
  GPIO_PORTA_AMSEL_R = 0;         // disable analog functionality on PA
  SSI0_CR1_R = 0x00000000;        // disable SSI, master mode
  SSI0_CPSR_R = 0x02;             // 8 MHz SSIClk
  SSI0_CR0_R &= ~(0x0000FFF0);    // SCR = 0, SPH = 0, SPO = 0 Freescale
  SSI0_CR0_R |= 0x0F;             // DSS = 16-bit data
  SSI0_DR_R = data;               // load 'data' into transmit FIFO
```

```
  SSI0_CR1_R |= 0x00000002;       // enable SSI
}

/*
****** DAC_Out ******
*           Input: uint16_t value corresponding to the next value to be output from the DAC
*           Output: none
*           Sets the next value for the DAC to output
*
*/
void DAC_Out(uint16_t output)
{
            uint32_t measure;
            while((SSI0_SR_R&0x00000002)==0){ };// SSI Transmit FIFO Not Full
  SSI0_DR_R = output;     // data out, no reply
            measure=0;
            measure++;
}
// SysTickInts.c
// Runs on LM4F120/TM4C123
// Use the SysTick timer to request interrupts at a particular period.
// Daniel Valvano
// October 11, 2012

// SysTickInts.c
// Michael Park, Jack Zhao
// Date modified: 02/24/2016
// handles note chaniging in a song
// Lab Number: 16340
// TA: Mahesh Srinivasan
// Last Revised: 02/29/2016

/* This example accompanies the book
   "Embedded Systems: Real Time Interfacing to Arm Cortex M Microcontrollers",
   ISBN: 978-1463590154, Jonathan Valvano, copyright (c) 2015

   Program 5.12, section 5.7

 Copyright 2015 by Jonathan W. Valvano, valvano@mail.utexas.edu
    You may use, edit, run or distribute this file
    as long as the above copyright notice remains
 THIS SOFTWARE IS PROVIDED "AS IS".  NO WARRANTIES, WHETHER EXPRESS, IMPLIED
 OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
 MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE.
 VALVANO SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL,
 OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
 For more information about my classes, my research, and my books, see
 http://users.ece.utexas.edu/~valvano/
 */

#include <stdint.h>
#include "../Shared/tm4c123gh6pm.h"
#include "SysTickInts.h"
#include "Music.h"


#define PF1       (*((volatile uint32_t *)0x40025008))
#define PF2       (*((volatile uint32_t *)0x40025010))
#define PF3       (*((volatile uint32_t *)0x40025020))


extern volatile uint16_t stop;
extern volatile uint16_t changeSound;
```

```
extern volatile uint32_t note;
extern unsigned char soundIndex;
extern Music current_song;
extern uint32_t addedWaves;
extern volatile uint32_t refresh;

extern const uint32_t Song2[128];

void DisableInterrupts(void); // Disable interrupts
void EnableInterrupts(void);  // Enable interrupts
long StartCritical (void);    // previous I bit, disable interrupts
void EndCritical(long sr);    // restore I bit to previous value
void WaitForInterrupt(void);  // low power mode

// **************SysTick_Init**********************
// Initialize SysTick periodic interrupts
// Input: interrupt period
//        Units of period are 12.5ns (assuming 50 MHz clock)
//        Maximum is 2^24-1
//        Minimum is determined by length of ISR
// Output: none
void SysTick_Init(uint32_t period){long sr;
  sr = StartCritical();
  NVIC_ST_CTRL_R = 0;         // disable SysTick during setup
  NVIC_ST_RELOAD_R = period-1;// reload value
  NVIC_ST_CURRENT_R = 0;      // any write to current clears it
//  NVIC_SYS_PRI3_R = (NVIC_SYS_PRI3_R&0x00FFFFFF)|0x40000000; // priority 2
                          // enable SysTick with core clock and interrupts
  NVIC_ST_CTRL_R = 0x07;
  EndCritical(sr);
}

// Time delay using busy wait.
// The delay parameter is in units of the core clock. (units of 20 nsec for 50 MHz clock)
void SysTick_Wait(uint32_t delay){
  volatile uint32_t elapsedTime;
  uint32_t startTime = NVIC_ST_CURRENT_R;
  do{
    elapsedTime = (startTime-NVIC_ST_CURRENT_R)&0x00FFFFFF;
  }
  while(elapsedTime <= delay);
}

// Time delay using busy wait.
// This assumes 50 MHz system clock.
void SysTick_Wait10ms(uint32_t delay){
  uint32_t i;
  for(i=0; i<delay; i++){
    SysTick_Wait(500000);  // wait 10ms (assumes 50 MHz clock)
  }
}

void SysTick_Handler(void)
{

  PF2 ^= 0x04;
  PF2 ^= 0x04;
            if(!stop)
            {
                        refresh=0;
                        note += 1;
                        if(note > 127)
                        {
```

```
                note = 0;
            }
            TIMER0_TAILR_R = (current_song.Song[note] / 32)  - 1;
            TIMER1_TAILR_R = (Song2[note]/32)  - 1;


        }

 PF2 ^= 0x04;
}

// Music.h
// Michael Park, Jack Zhao
// Date Created: 02/24/2016
// Includes notes, struct, and music.c function prototypes
// Lab Number: 16340
// TA: Mahesh Srinivasan
// Last Revised: 02/29/2016

#ifndef _NOTE_H_
#define _NOTE_H_
#include <stdint.h>

#define C4                  (50000000/261)
#define D4                  (50000000/293)
#define E4                  (50000000/330)
#define F4                  (50000000/348)
#define G4                  (50000000/391)
#define A4                  (50000000/440)
#define B4                  (50000000/495)
#define C5                  (50000000/522)
#define D5                  (50000000/587)
#define E5                  (50000000/660)
#define F5                  (50000000/695)
#define G5                  (50000000/782)
#define A5                  (50000000/880)
#define B5                  (50000000/990)
#define C6                  (50000000/1043)
#define Gs4                 (50000000/415)

#define SONG_LENGTH 128

typedef struct {
        uint32_t length;
        uint32_t Song[SONG_LENGTH];
}Music;

//const uint16_t Wave[32];

//unsigned char soundIndex; //varies from 0 to 32

//const uint32_t Song[128];

//static uint16_t stopped;
//static uint16_t altSound;
//int note;

void Music_Init(void);
void PlaySong(void);
void PlaySong2(void);
void addWaves(void);
#endif
```