ECE-3551 Final Project

MAZE

By Brandon Schmitt
A Dynamic Maze Game

- “A maze is a tour puzzle in the form of a complex branching passage through which the solver must find a route.”

Source: http://en.wikipedia.org/wiki/Maze_generation
Project Goals

• Create a maze generator to run on Analog Device’s Blackfin BF533 DSP
• Utilize push buttons on the EZ-Kite Lite development board to move a cursor within the maze
• Allow customization of maze size
• Capable of generating multiple different mazes
• Display user interface on a television screen
Project Components

• Video
  – ADV7170 Video Encoding
  – NTSC Signal
  – Development Process

• User Input
  – SW4 through SW7
  – Left, Up, Down, Right
Project Components

• **Maze Generation Algorithm**
  
  – Depth First Search

1. Select any cell in the array as the current cell. Set the count of cells visited to one.
2. If one or more neighbor cells exist that have not been visited,
   A. Pick one at random as the next cell.
   B. Remove the walls between the cells.
   C. Make the next cell the current cell.
   D. Increment count of cells visited.
3. else, if none exist
   A. make the previous cell the current cell (backtrack).
4. Repeat step 2 until the count of cells visited equals total number of cells.

Algorithm Text Source: http://home.att.net/~srschmitt/script_maze_generator.html
Challenges and Solutions

- Video

- DMA / Memory Access

- Random Number Generation
  - Linear Congruential Generator
  - $V_{j+1} = (A \times V_j + B) \mod M$, where $A$, $B$, $M$ constants
  - $A = 1664525$, $B = 1013904223$, $M = 232$
MAZE Conclusion

• MAZE successfully implemented
• Project goals met

• Further Improvements
  – User interface for setting maze selection parameters
  – Congratulations screen upon successful completion of a maze
  – Auto solve animation triggered by user input
References

• Maze Generation Algorithm
  – http://home.att.net/~srschmitt/script_maze_generator.html
  – http://en.wikipedia.org/wiki/Maze_generation

• Random Number Generation
  – http://www.random.org/integers/
Bass Effects

Andrew Lash
ECE 3551
DSP on BlackFin BF533
Purpose

- Create a bass guitar multi-effects processor
  - Delay
  - Chorus
  - Flanger
  - Vibrato

- Single unit to replace multiple effects processors
Solution - Operation

- Board has 4 Pushbuttons
  - Left: Switches between effects
  - Middle-Left: Current delay time
  - Middle-Right: Varies on effect
    - Delay: None
    - Chorus, Flanger, and Vibrato: Frequency
  - Right: Amount of decay
Solution – Operation

- Board has six LEDs
  - Two on left: Show current effect mode
    - 00 – Delay
    - 10 – Chorus
    - 01 – Flanger
    - 11 – Vibrato
  - Four on right: Show the current value for the selected effect characteristic
Solution – Usability

- Use the first button to switch between modes
- Other 3 buttons: Press once to set that characteristic as active
  - LEDs will update
- Press more to increment value by 1 as desired
  - LEDs will update
Solutions – Delay

\[ x[n] = x[n] + \alpha x[n - D] \]

\[ y[n] = x[n] \]
Solutions - Chorus

\[ y[n] = x[n] + \alpha x[n - d(n)] \]

- **Variable sinusoidal Delay** \( d(n) \)
- **Modulating Center of Delay-Line**
- **Delay-Line Gain** \( \alpha \)
- **LFO**
- **Sinusoidal Low Frequency Oscillator**
Solutions - Flanger

- Flanger effect is similar to Chorus
  - The amount of overall delay is lower
  - The frequency rate is much higher

\[ d(n) = \frac{D}{2} \left[ 1 - \cos(2\pi n f_{\text{cycle}}) \right] \]
Solutions - Vibrato

\[ y[n] = \alpha x[n - d(n)] \]

\( x[n] \xrightarrow{Z^{-N}} \)  

Modulating Center of Delay-Line  

\[ \alpha x[n-d(n)] \]  

Delay-Line Gain  

N=Variable Delay d(n)  

Professional Samples  
My Samples
Difficulties

- Flanger and Chorus are very similar
  - Hard to differentiate between effects
- Designing with only 4 buttons and 6 LEDs is very limiting
  - Had to be creative when designing user interface
  - Limits how much functionality that can be implemented in the display
Results

- Delay effect works extremely well!
- Chorus sounds as expected
- Flanger works ok but not as well I as I would like it to
- Vibrato works really well
References

- http://my.fit.edu/~vkepuska/ece3551/
- http://www.musiciansfriend.com
:: FINAL PROJECT ::
Bass Guitar Multi-Effects Pedal in DSP

Submitted :: Dec 10, 2006

Andrew Lash
**Problem Statement:** To create a DSP capable of handling multiple Bass Guitar Effects and to achieve as close to professional sounds as possible.

**Literature Review of Various Solutions:**

**Delay:** With delay, I had the choice of using a single tap or multi-tap, with or without feedback, and how much control to allow the user. I decided to use a single tap delay as I have found this to provide a closer approximation to the effect I was looking for. I also used feedback so that even if the bassist stops playing, it will still delay and allows for a really nice digital delay sound.

**Chorus:** My main choice for chorus was which kind of low frequency oscillator to use. I tried to use a random low frequency oscillator however I was unable to achieve a clean sounding effect. I then switched to using a sinusoidal oscillator which resulted in a cleaner sound but a slightly more predictable effect.

**Flanger:** Flanger is almost identical to chorus. The major difference is that it is not delayed as much.

**Vibrato:** This effect was also very similar to flanger and chorus, without the input being added in to the output.

**Detailed explanation of selected solution and implementation:**

**Delay:**

\[ y[n] = x[n] + \alpha x[n-D] \]

The chart above shows how the delay operates. It is a single tap delay and therefore only has one delay added back in. The delay routine does however have feedbacks so a single note from the bass guitar could be repeated indefinitely if there is no decay on the delayed sample. For the code of the delay effect, see the function, Delay_Effect in Process_data.c below.
Chorus and Flanger:

\[ y[n] = x[n] + \alpha [n - d(n)] \]

The chart and equations above show how the chorus and flanger effects work. Both of these effects work using the same basic principles and concepts; however the amount of delay and the rate for each effect is different. The flanger has a smaller delay than a delay in order to maximize the number of pits in the comb filter results. This creates the “swooshing” effect that most people know the flanger for. Chorus on the other hand creates a varying delay in order to give the false appearance that there is more than just a single instrument.

Vibrato:

\[ y[n] = \alpha [n - d(n)] \]

Above is the picture of the flowchart for the vibrato effect. The major difference between this effect and the chorus and flanger is that it lacks the input sample being added into the output. This creates a pitch shifting mechanism that can create an effect like the guitarist is using a whammy bar effect.
Results:
This project led way to the creation of four (4) great effects. The delay effect can be set to very close specifications as a professionally made model and is very impressive. The Chorus also has very impressive sound effect; however it occasionally has clicks that can interfere with the intent of the musician. Flanger was quite disappointing to me. It sounds way too much like chorus and did not give me the “swooshing effect” that I desired. Finally, the vibrato was quite flexible and allowed me to come close to matching a professional vibrato. All four of these effects are great effects to have and having them all in a single place makes a musician’s life that much easier!

Concluding Remarks:
I really enjoyed this project. I learned a lot more about DSP than I had before and it also gave me a great opportunity to touch up on my C/C++ skills. I got to combine something I love doing (playing my bass) with school work to create a project that was fun, challenging, interesting, educational, and exciting. I could not have asked for a better final project.

References:
http://mv.fit.edu/~vkepuska/ece3551/
http://www.musiciansfriend.com

See following pages for code
Simon Says

ECE 3551 Microcomputer Systems I
Elizabeth Nelson
Problem Statement

• To design and implement an interactive game of repetition.
• Computer must generate an array of random values to be associated with one of 4 LEDs and 4 corresponding speakers
• Values are outputted in steps: 1, 1-2, 1-2-3 ...
• User must reproduce pattern
• Different algorithms indicate “wins” and “losses”
Challenges

- **Timing:**
  - Double values
  - User cutoff
  - Off cycles

- **Masking and Unmasking Interrupts**

- **Stepping the output**

- **Input accumulation**

- **Sound shifts**

- **Tone Generation**
• Time limits – second timer?
• User double values
• Rand and Srand – seeding
  – Time function
• Code layering
• The skip and end booleans - allows for an “off cycle” and prevents user cutoff

```c
if (skip == false)
 {...
     ... skip = true
 }
Else
{
    if(end == true)
    {
        skip = false;
        end = false;
    }
    else
    {
        if(n == 0)
        {
            skip = true;
        }
        else
        {
            skip = false;
        }
    }
}
```
Masking the Timer0 interrupt
*pSIC_IMASK = 0x00080200; - masks timer0 interrupt
*pSIC_IMASK = 0x00090000; - masks sport0 interrupt

The n and point indices

```c
if (skip == false) {
    if (compValue[n] == 1) {
        LEDValue = 0x01;
        *pSIC_IMASK = 0x00090200;
    } else if (compValue[n] == 2) {
        LEDValue = 0x02;
        *pSIC_IMASK = 0x00090200;
    } else if (compValue[n] == 3) {
        LEDValue = 0x04;
        *pSIC_IMASK = 0x00090200;
    } else if (compValue[n] == 4) {
        LEDValue = 0x08;
        *pSIC_IMASK = 0x00090200;
    }
    *pFlashA_PortB_Data = LEDValue;
    n++;
    if (n > point) {
        n = 0;
        point ++;
    } if (point > 9) {
        point = 10;
    }
    skip = true;
}
```

if (n == 0) {
    skip = true;
} else {
    skip = false;
}
• **The userPoint variable**

```c
if(pattern == 0x0100)
{
    // confirm interrupt handling
    *pFIO_FLAG_C = 0x0100;

    userValue[userPoint] = 1;
    Guess = 0x01;
    *pFlashA_PortB_Data = Guess;
}
...
userPoint++;
if(userPoint == point)
{
    userPoint = 0;
    ...
}
```
• Sound outputs - delayed indices

if(n == 0)
{
    if(compValue[point-1] == 1)
    {
        iChannel0LeftIn = iRxBuffer1[INTERNAL_ADC_L0];
        Process_Data();
        iTxBuffer1[INTERNAL_DAC_L0] = iChannel0LeftOut;
    }...
}
else
{
    if(compValue[n-1] == 1)
    {
        iChannel0LeftIn = iRxBuffer1[INTERNAL_ADC_L0];
        Process_Data();
        iTxBuffer1[INTERNAL_DAC_L0] = iChannel0LeftOut;
    }...
}
• Tone Generation
  - Sine waves - $2n*\pi/12$ - sample every 5?
  - Sine array
  - Array of 1, -1, 1 ...
  - Sport0 interrupt generation
• Time limits
• User double values
• A second timer
  – Would be unmasked when the user pushes the first button
  – Would create a time limit
  – Use to create user “off cycles”
**Srand and Rand**

```c
srand(time(0));
for(m=0; m<10; m++)
{
    temp = (rand()%32 + 1)/8 + 1;
    if(temp == 5)
    {
        compValue[m] = 4;
    }
    else
    {
        compValue[m] = temp;
    }
}
srand(compValue[0]);
for(m=0; m<10; m++)
{
    temp = (rand()%32 + 1)/8 + 1;
    if(temp == 5)
    {
        compValue[m] = 4;
    }
    else
    {
        compValue[m] = temp;
    }
}
```
Conclusions

• Code Layering Technique
  – Final v1, v2 and v3
• Additional layers
  – Time limit
  – User doubles
  – Tone generation
• Files
• Win and Loss algorithms
Resources

• Audio Talkthrough
• BF533 Flags
  – Analog Devices Example Code
• Class notes
• Blackfin Resource Manual
  – Dr. Veton Kepuska
    http://my.fit.edu/~vkepuska/ece3551
Simon Says
Elizabeth Nelson
Microcomputer Systems I
ECE 3551
Dr. Veton Kepuska
Problem Statement
The purpose of this project is to design and implement an interactive game of repetition. The computer will associate sounds with lights to indicate a random pattern. The user must then reproduce it by pressing the corresponding buttons to recreate the light and sound pattern.

Challenges
- The timing of both the computer indications and the user inputs must be regulated. Timer0 is used as a trigger for the computer outputs, however, when the same value is used twice, it is difficult to tell. Thus, after each value is indicated, the light and sound need to be terminated for at least one cycle of the timer.
- In order to prevent the timer from interrupting the user and the sound from continuously playing, interrupts need to be masked and unmasked during program execution.
- In order to produce random numbers, the rand function in the c standard library is used. The function needs to be seeded using the srand function with a different seed value for each completion of the game.
- The computer must output an increasing number of values this means that two pointers must be used to not only regulate how many values are outputted, but also which ones.
- Following that challenge, the user values must also be accumulated in increasing numbers, but each set must be compared starting from the first value.
- Sound must be shifted from speaker to speaker depending on the outputted value.
- Ideally, individual tones could be generated to correspond to each light and outputted. This was attempted, but all methods for generating or calculating a signal failed to produce any actual sound.
Solution

The basic solution implementation can be visualized as in this flow chart:

- **Initialize timer and flag interrupts. Define variables Initialize variables.**
- **Generate Random numbers. Modulus factor of 6 And store in the comp values Array.**
- **Light up first LED n =0**
- **User inputs Their value By pressing a button**
  - **User value = comp value**
  - **Value is correct. N is incremented.**
  - **Value is incorrect. Return to fist step.**
- **N is checked To see if it is within bounds**
- **If N is beyond the Largest value. Win = true.**
- **All LED’s flash or move in a direction. Perhaps some noise is Made?**
- **For k=0; k<n**
  - **User inputs value**
  - **User value is checked vs kth value**
  - **User value = compvalue[k]**
- **Single long flash of LED’s**
- **Wait for user to press button**
Code and Code Description

//setup.h
//Author: Elizabeth Nelson
//Date: 12-09-06
//Description: Header file for final project v4

ifndef _setup_DEFINED
define __setup_DEFINED

#include <sys\exception.h>
#include <cdefBF533.h>

// addresses for Port A in Flash A
#define pFlashA_PortA_Dir (volatile unsigned char *)0x20270006
#define pFlashA_PortA_Data (volatile unsigned char *)0x20270004

// addresses for Port B in Flash A
#define pFlashA_PortB_Dir (volatile unsigned char *)0x20270007
#define pFlashA_PortB_Data (volatile unsigned char *)0x20270005

// names for codec registers, used for sCodec1836TxRegs[]
#define DAC_CONTROL_1 0x0000
#define DAC_CONTROL_2 0x1000
#define DAC_VOLUME_0 0x2000
#define DAC_VOLUME_1 0x3000
#define DAC_VOLUME_2 0x4000
#define DAC_VOLUME_3 0x5000
#define DAC_VOLUME_4 0x6000
#define DAC_VOLUME_5 0x7000
#define ADC_0_PEAK_LEVEL 0x8000
#define ADC_1_PEAK_LEVEL 0x9000
#define ADC_2_PEAK_LEVEL 0xA000
#define ADC_3_PEAK_LEVEL 0xB000
#define ADC_CONTROL_1 0xC000
#define ADC_CONTROL_2 0xD000
#define ADC_CONTROL_3 0xE000

define INTERNAL_ADC_L0 0
#define INTERNAL_ADC_R0 2
#define INTERNAL_DAC_L0 0
#define INTERNAL_DAC_R0 2
#define INTERNAL_ADC_L1 1
#define INTERNAL_ADC_R1 3
#define INTERNAL_DAC_L1 1
#define INTERNAL_DAC_R1 3

// size of array sCodec1836TxRegs
#define CODEC_1836_REGS_LENGTH 11

// SPI transfer mode
#define TIMOD_DMA_TX 0x0003

// SPORT0 word length
#define SLEN_24 0x0017
// DMA flow mode
#define FLOW_1 0x1000

//Globals
extern int iChannel0LeftIn;
extern int iChannel0RightIn;
extern int iChannel0LeftOut;
extern int iChannel0RightOut;
extern int iChannel1LeftIn;
extern int iChannel1RightIn;
extern int iChannel1LeftOut;
extern int iChannel1RightOut;
extern volatile short sCodec1836TxRegs[];
extern volatile int iRxBuffer1[];
extern volatile int iTxBuffer1[];
extern int point;
extern short pattern;
extern unsigned char LEDValue;
extern int compValue[10];
extern int userValue[10];
extern int n;
extern unsigned char LEDValue;
extern bool skip;
extern bool loss;
extern int userPoint;
extern bool end;
extern bool wins;
extern bool comp;

//prototypes
// in file Initializations.c
void Init_EBIU(void);
void Init_Flash(void);
void Init1836(void);
void Init_Sport0(void);
void Init_DMA(void);
void Init_Flags(void);
void Init_Timers(void);
// in file Process_data.c
void Process_Data(void);

// in file ISRs.c
EX_INTERRUPT_HANDLER(Sport0_RX_ISR);
EX_INTERRUPT_HANDLER(FlagA_ISR);
EX_INTERRUPT_HANDLER(Timer0_ISR);
void lose(void);
void win(void);

#endif

The header file, setup.h, defines the addresses for FlashA, ports A and B. Port A contains the reset pins for AD1836 and portB contains the pins for LEDs 4-7. The next section
defines several global variables. The iChannel integers are the audio copy variables. They store the audio input as it is copied from input to output. The variable point, points to where in the computer value array the process is. The variable userPoint does the same for the userValue array. The short pattern, indicates which of the PF buttons has been pressed. The unsigned char LEDValue is declared in setup.h but is not actually initialized in main.c but in ISRs.c by the Timer0 interrupt handler. For some reason, initializing it in main.c causes it to be ineffective for transferring data to *pFlashA_PortB_Data. There are several Booleans which determine what state the program it is in. There are four basic states. When skip is false, the computer is outputting values. When skip is true, the computer is either between values or finished outputting values. When wins is true, the user has successfully repeated all ten computer defined values and has “won”. When loss is true, the user has entered a non-valid input.

//main.c
//Author: Elizabeth Nelson
//Date: 12-9-06
//Description: main function of final v4

#include "ccblkfn.h"
#include "sysreg.h"
#include <stdlib.h>
#include <time.h>
#include "setup.h"

//Variables
short pattern = 0x0000;  //indicates which button has been pressed
int compValue[10];   //array of random numbers from 1-4
int userValue[10] = {0}; //array containing user guesses 1-4
int n =0;     //index variable for displaying
computer values
int point = 0;    //shared point - maximum value the
coputer has displayed
int userPoint = 0;   //index of next location in
userValue array
int iChannel0LeftIn;  //left input data channel 0
int iChannel1LeftIn;  //left input data channel 1
int iChannel0RightIn;  //right input data channel 0
int iChannel1RightIn;  //right input data channel 1
int iChannel0LeftOut;  //left output data channel 0
int iChannel1LeftOut;  //left output data channel 1
int iChannel0RightOut;  //right output data channel 0
int iChannel1RightOut;  //right output data channel 1
volatile int iTxBuffer1[4]; //Sport0 DMA transmit buffer
volatile int iRxBuffer1[4]; //Sport0 DMA receive buffer
bool skip = false;   //allows for every other timer
cycle to be an "off" cycle
bool loss = false;   //indicates if the user has lost
bool end = false;    //indicates the end of the user input -
allows for one timer cycle of "off" time
bool wins = false;   //indicates if the user has won
bool comp = true;     //indicates whether the computer or user
is inputting data

// array for registers to configure the ad1836
// names are defined in "Talkthrough.h"
volatile short sCodec1836TxRegs[CODEC_1836_REGS_LENGTH] =
{
    // DAC CONTROL 1 | 0x000,
    // DAC CONTROL 2 | 0x000,
    DAC_VOLUME_0   | 0x3ff,
    DAC_VOLUME_1   | 0x3ff,
    DAC_VOLUME_2   | 0x3ff,
    DAC_VOLUME_3   | 0x3ff,
    DAC_VOLUME_4   | 0x000,
    DAC_VOLUME_5   | 0x000,
    ADC_CONTROL_1  | 0x000,
    ADC_CONTROL_2  | 0x000,
    ADC_CONTROL_3  | 0x000
};

// main function
// after initializing all the interrupts and calculating the compValue array main just waits in an infinite loop
void main(void)
{
    sysreg_write(reg_SYSCFG, 0x32);  // Initialize System Configuration
    Init_EBIU();
    Init_Flash();
    Init1836();
    Init_Sport0();
    Init_DMA();
    Init_Interrupts();
    Init_Flags();
    Init_Timers();
    Enable_DMA_Sport0();

    int m = 0;
    int temp;

    srand(time(0));

    for(m=0; m<10; m++)
    {
        temp = (rand()%32 + 1)/8 +1;
        if(temp ==5)
        {
            compValue[m] = 4;
        }
        else
        {
            compValue[m] = temp;
        }
    }

    while(1);
}
In main.c most variables are initialized to 0. The exception would be the Boolean comp which is initialized to true which indicates that the computer is outputting audio data first. The system is initialized and then the initialization functions in Initializations.c are called. Lastly, the compValue array is initialized using an iterative loop and the srand and rand functions.

//initializations.c
//Author: Elizabeth Nelson
//Date: 12-09-06
//Description: contains initialization functions to initialize all interrupts etc.
#include "setup.h"

//function Init_EBIU
//services requests and allows for interface with flash
void Init_EBIU(void)
{
    *pEBIU_AMBCTL0 = 0x7bb07bb0; //sets read, write, setup, hold and transition times and enables ARDY for banks 0 and 1
    *pEBIU_AMBCTL1 = 0x7bb07bb0; //same for banks 2 and 3
    *pEBIU_AMGCTL = 0x000f; //enables CLKOUT, and banks 0, 1, 2 and 3
}

//function Init_Flash
//initializes pin direction of portA and portB in flash A to output
void Init_Flash(void)
{
    *pFlashA_PortA_Dir = 0x1; //AD1836 reset
    *pFlashA_PortB_Dir = 0x3f; //LEDs
}

//function Init1836
//sets up the SPI port to configure the AD1836 and content of the array sCodec1836TxRegs is sent to the codec
void Init1836(void)
{
    int i;
    int j;
    static unsigned char ucActive_LED = 0x01;

    // write to Port A to reset AD1836
    *pFlashA_PortA_Data = 0x00;

    // write to Port A to enable AD1836
    *pFlashA_PortA_Data = ucActive_LED;

    // wait to recover from reset
    for (i=0; i<0xf0000; i++) asm("nop;");

    // Enable PF4
    *pSPI_FLG = FLS4;
    // Set baud rate SCK = HCLK/(2*SPIBAUD) SCK = 2MHz
    *pSPI_BAUD = 16;
    // configure spi port
    // SPI DMA write, 16-bit data, MSB first, SPI Master
*pSPI_CTL = TIMOD_DMA_TX | SIZE | MSTR;

// Set up DMA5 to transmit
// Map DMA5 to SPI
*pDMA5_PERIPHERAL_MAP = 0x5000;

// Configure DMA5
// 16-bit transfers
*pDMA5_CONFIG = WDSIZE_16;
// Start address of data buffer
*pDMA5_START_ADDR = sCodec1836TxRegs;
// DMA inner loop count
*pDMA5_X_COUNT = CODEC_1836_REGS_LENGTH;
// Inner loop address Increment
*pDMA5_X_MODIFY = 2;

// enable DMAs
*pDMA5_CONFIG = (*pDMA5_CONFIG | DMAEN);
// enable spi
*pSPI_CTL = (*pSPI_CTL | SPE);
// wait until dma transfers for spi are finished
for (j=0; j<0xaaff0; j++) asm("nop;");

// disable spi
*pSPI_CTL = 0x0000;
}

//function Init_Sport0
//Configures Sport0 for I2S mode, to transmit/receive data to/from the
//AD1836 and for external clocks and frame syncs
void Init_Sport0(void)
{
    // Sport0 receive configuration
    // External CLK, External Frame sync, MSB first, Active Low
    // 24-bit data, Stereo frame sync enable
    *pSPORT0_RCR1 = RFSR | LRFS | RCKFE;
    *pSPORT0_RCR2 = SLEN_24 | RXSE | RSFSE;

    // Sport0 transmit configuration
    // External CLK, External Frame sync, MSB first, Active Low
    // 24-bit data, Secondary side enable, Stereo frame sync enable
    *pSPORT0_TCR1 = TFSR | LTFS | TCKFE;
    *pSPORT0_TCR2 = SLEN_24 | TXSE | TSFSE;
}

//function Init_DMA
//Initialize DMA1 in autobuffer mode to receive and DMA2 in autobuffer
//mode to transmit
void Init_DMA(void)
{
    // Set up DMA1 to receive
    // Map DMA1 to Sport0 RX
    *pDMA1_PERIPHERAL_MAP = 0x1000;

    // Configure DMA1
    // 32-bit transfers, Interrupt on completion, Autobuffer mode
    *pDMA1_CONFIG = WNR | WDSIZE_32 | DI_EN | FLOW_1;
    // Start address of data buffer
*pDMA1_START_ADDR = iRxBuffer1;
// DMA inner loop count
*pDMA1_X_COUNT = 4;
// Inner loop address increment
*pDMA1_X_MODIFY = 4;

// Set up DMA2 to transmit
// Map DMA2 to Sport0 TX
*pDMA2_PERIPHERAL_MAP = 0x2000;

// Configure DMA2
// 32-bit transfers, Autobuffer mode
*pDMA2_CONFIG = WDSIZE_32 | FLOW_1;
// Start address of data buffer
*pDMA2_START_ADDR = iTxBuffer1;
// DMA inner loop count
*pDMA2_X_COUNT = 4;
// Inner loop address increment
*pDMA2_X_MODIFY = 4;
}
//function Enable_DMA_Sport0
//Enables DMA1, DMA2, Sport0 TX and Sport0 RX
void Enable_DMA_Sport0(void)
{
    // enable DMAs
    *pDMA2_CONFIG = (*pDMA2_CONFIG | DMAEN);
    *pDMA1_CONFIG = (*pDMA1_CONFIG | DMAEN);

    // enable Sport0 TX and RX
    *pSPORT0_TCR1 = (*pSPORT0_TCR1 | TSPEN);
    *pSPORT0_RCR1 = (*pSPORT0_RCR1 | RSPEN);
}
//function Init_Flags
//Initialize flags for PF8-11 as inputs, edge sensitive, and enabled
void Init_Flags(void)
{
    *pFIO_INEN = 0x0F00;
    *pFIO_DIR = 0x0000;
    *pFIO_EDGE = 0x0F00;
    *pFIO_MASKA_D = 0x0F00;
}
//function Init_Timers
//Timer0 as PWM mode, sets period and duty cycle
void Init_Timers(void)
{
    *pTIMER0_CONFIG = 0x0019;
    *pTIMER0_PERIOD = 0x02000000;
    *pTIMER0_WIDTH = 0x01000000;
    *pTIMER_ENABLE = 0x0001;
}
void Init_Interruption(void)
{
    // Set Sport0 RX (DMA1) interrupt priority to 2 = IVG9
    *pSIC_IAR0 = 0xffffffff;
    *pSIC_IAR1 = 0xffffffff2f;
*pSIC_IAR2 = 0xffff5ff4;  //FlagA -> ID5 = IVG12, Timer0 -> ID4 = IVG11

// assign ISRs to interrupt vectors
// Sport0 RX ISR -> IVG 9
register_handler(ik_ivg9, Sport0_RX_ISR);
register_handler(ik_ivg12, FlagA_ISR);
register_handler(ik_ivg11, Timer0_ISR);

// enable Timer0 interrupt and flagA interrupt
*pSIC_IMASK = 0x00090000;

The first function, Init_EBUI initializes the external bus interface unit. It sets the read, write, hold and transition times for all banks. It also enables ARDY. Finally, it enables CLKOUT and banks 0,1,2 and 3. The Asynchronous Memory Global Control Register is always the last to be set. Init_Flash initializes the directions on portA and portB. The function Init1836 sets up the SPI port to configure the AD1836 and send the content of the array sCodec1836TxRegs to the codec. The Init_Sport0 function configures Sport0 for I2S mode, to transmit and receive data to and from the AD1836 and for external clocks and frame syncs. The Init_DMA function initializes DMA1 in autobuffer mode to receive and DMA2 in autobuffer mode to transmit. Init_Flags enables PF8-11, sets all of the PF’s as inputs, makes them all edge sensitive, and then enables them as write-1-to-set. The last function enables all of the interrupts. First, Sport0 RX interrupt priority is set to 2 which is equivalent to IVG9. FlagA is set to ID5 or IVG12 and Timer0 is set to ID4 or IVG11. The interrupt handlers for each interrupt are then registered. Finally, only the flagA and Timer0 interrupts are unmasked. The Sport0 interrupt remains masked for the first part of the program.

//ISR.c
//Author: Elizabeth Nelson
//Date: 12-09-06
//Description: Interrupt Service Handlers for Sport0, Timer0, and FlagA

#include "setup.h"
#include <stdlib.h>
#include <time.h>

EX_INTERRUPT_HANDLER(Sport0_RX_ISR)
{
    // confirm interrupt handling
    *pDMA1_IRQ_STATUS = 0x0001;
    if(comp == true)
    {
        if(n == 0)
        {
            if(compValue[point-1] == 1)
            {
                iChannel0LeftIn = iRxBuffer1[INTERNAL_ADC_L0];

                Process_Data();

                iTxBuffer1[INTERNAL_DAC_L0] = iChannel0LeftOut;
            }
        }
    }
}
} else if(compValue[point-1] == 2)
{
    iChannel0RightIn = iRxBuffer1[INTERNAL_ADC_R0];
    Process_Data();
    iTxBuffer1[INTERNAL_DAC_R0] = iTxBuffer1[INTERNAL_DAC_R0] = iChannel0RightOut;
} else if(compValue[point-1] == 3)
{
    iChannel1LeftIn = iRxBuffer1[INTERNAL_ADC_L1];
    Process_Data();
    iTxBuffer1[INTERNAL_DAC_L1] = iTxBuffer1[INTERNAL_DAC_L1] = iChannel1LeftOut;
} else if(compValue[point-1] == 4)
{
    iChannel1RightIn = iRxBuffer1[INTERNAL_ADC_R1];
    Process_Data();
    iTxBuffer1[INTERNAL_DAC_R1] = iTxBuffer1[INTERNAL_DAC_R1] = iChannel1RightOut;
}
} else if(compValue[n-1] == 1)
{
    iChannel0LeftIn = iRxBuffer1[INTERNAL_ADC_L0];
    Process_Data();
    iTxBuffer1[INTERNAL_DAC_L0] = iTxBuffer1[INTERNAL_DAC_L0] = iChannel0LeftOut;
} else if(compValue[n-1] == 2)
{
    iChannel0RightIn = iRxBuffer1[INTERNAL_ADC_R0];
    Process_Data();
    iTxBuffer1[INTERNAL_DAC_R0] = iTxBuffer1[INTERNAL_DAC_R0] = iChannel0RightOut;
} else if(compValue[n-1] == 3)
{
    iChannel1LeftIn = iRxBuffer1[INTERNAL_ADC_L1];
    Process_Data();
    iTxBuffer1[INTERNAL_DAC_L1] = iTxBuffer1[INTERNAL_DAC_L1] = iChannel1LeftOut;
} else if(compValue[n-1] == 4)
{

iChannel1RightIn = iRxBuffer1[INTERNAL_ADC_R1];
Process_Data();
iTxBuffer1[INTERNAL_DAC_R1] =
iChannel1RightOut;
}
}
else {
    if(userPoint == 0)
    {
        if(userValue[point-1] == 1)
        {
            iChannel0LeftIn = iRxBuffer1[INTERNAL_ADC_L0];
            Process_Data();
iTxBuffer1[INTERNAL_DAC_L0] = iChannel0LeftOut;
        }
    }
else if(userValue[point-1] == 2)
    {
        iChannel0RightIn = iRxBuffer1[INTERNAL_ADC_R0];
        Process_Data();
iTxBuffer1[INTERNAL_DAC_R0] =
iChannel0RightOut;
    }
else if(userValue[point-1] == 3)
    {
        iChannel1LeftIn = iRxBuffer1[INTERNAL_ADC_L1];
        Process_Data();
iTxBuffer1[INTERNAL_DAC_L1] = iChannel1LeftOut;
    }
else if(userValue[point-1] == 4)
    {
        iChannel1RightIn = iRxBuffer1[INTERNAL_ADC_R1];
        Process_Data();
iTxBuffer1[INTERNAL_DAC_R1] =
iChannel1RightOut;
    }
else
    {
        if(userValue[userPoint-1] == 1)
        {
            iChannel0LeftIn = iRxBuffer1[INTERNAL_ADC_L0];
            Process_Data();
iTxBuffer1[INTERNAL_DAC_L0] = iChannel0LeftOut;
        }
else if(userValue[userPoint-1] == 2) {
    iChannel0RightIn = iRxBuffer1[INTERNAL_ADC_R0];
    Process_Data();
    iTxBuffer1[INTERNAL_DAC_R0] = iTxBuffer1[INTERNAL_DAC_R0] = iChannel0RightOut;
} else if(userValue[userPoint-1] == 3) {
    iChannel1LeftIn = iRxBuffer1[INTERNAL_ADC_L1];
    Process_Data();
    iTxBuffer1[INTERNAL_DAC_L1] = iTxBuffer1[INTERNAL_DAC_L1] = iChannel1LeftOut;
} else if(userValue[userPoint-1] == 4) {
    iChannel1RightIn = iRxBuffer1[INTERNAL_ADC_R1];
    Process_Data();
    iTxBuffer1[INTERNAL_DAC_R1] = iTxBuffer1[INTERNAL_DAC_R1] = iChannel1RightOut;
}

if(wins == true) {
    iChannel0RightIn = iRxBuffer1[INTERNAL_ADC_R0];
    iChannel0LeftIn = iRxBuffer1[INTERNAL_ADC_L0];
    Process_Data();
    iTxBuffer1[INTERNAL_DAC_L0] = iTxBuffer1[INTERNAL_DAC_L0] = iChannel0LeftOut;
    iTxBuffer1[INTERNAL_DAC_R0] = iTxBuffer1[INTERNAL_DAC_R0] = iChannel0RightOut;
} else if(loss == true) {
    iChannel1RightIn = iRxBuffer1[INTERNAL_ADC_R0];
    iChannel1LeftIn = iRxBuffer1[INTERNAL_ADC_L0];
    Process_Data();
    iTxBuffer1[INTERNAL_DAC_L0] = iTxBuffer1[INTERNAL_DAC_L0] = iChannel1LeftOut;
    iTxBuffer1[INTERNAL_DAC_R0] = iTxBuffer1[INTERNAL_DAC_R0] = iChannel1RightOut;
}

EX_INTERRUPT_HANDLER(FlagA_ISR)
{
    comp = false;
    if(loss == false && wins == false)
unsigned char Guess;

pattern = *pFIO_FLAG_C;
*pSIC_IMASK = 0x00080200;

if(pattern == 0x0100)
{
    // confirm interrupt handling
    *pFIO_FLAG_C = 0x0100;
    userValue[userPoint] = 1;
    Guess = 0x01;
    *pFlashA_PortB_Data = Guess;
}
else if(pattern == 0x0200)
{
    *pFIO_FLAG_C = 0x0200;
    userValue[userPoint] = 2;
    Guess = 0x02;
    *pFlashA_PortB_Data = Guess;
}
else if(pattern == 0x0400)
{
    *pFIO_FLAG_C = 0x0400;
    userValue[userPoint] = 3;
    Guess = 0x04;
    *pFlashA_PortB_Data = Guess;
}
else if(pattern == 0x0800)
{
    *pFIO_FLAG_C = 0x0800;
    userValue[userPoint] = 4;
    Guess = 0x08;
    *pFlashA_PortB_Data = Guess;
}

userPoint++;

if(userPoint == point)
{
    userPoint = 0;
    int i;
    for(i=0; i< point; i++)
    {
        if(userValue[i] == compValue[i])
        {
            skip = true;
            end = true;
            *pTIMER_STATUS = 0x0001;
*pSIC_IMASK = 0x00090200;

else
{
   lose();
}
if(point == 10)
{
   win();
}
}
}
else
{
   pattern = *pFIO_FLAG_C;
   if(pattern == 0x0100)
   {
      *pFIO_FLAG_C = 0x0100;
      wins = false;
      loss = false;
   }
   *pSIC_IMASK = 0x00090000;
}
}
EX_INTERRUPT_HANDLER(Timer0_ISR)
{
   // confirm interrupt handling
   *pTIMER_STATUS = 0x0001;
   comp = true;
   unsigned char LEDValue = 0x00;
   if (skip == false)
   {
      if(compValue[n] == 1)
      {
         LEDValue = 0x01;
         *pSIC_IMASK = 0x00090200;
      }
      else if(compValue[n] == 2)
      {
         LEDValue = 0x02;
         *pSIC_IMASK = 0x00090200;
      }
      else if(compValue[n] == 3)
      {
         LEDValue = 0x03;
         *pSIC_IMASK = 0x00090200;
      }
   }
LEDValue = 0x04;
*pSIC_IMASK = 0x00090200;
}
else if(compValue[n] == 4)
{
    LEDValue = 0x08;
    *pSIC_IMASK = 0x00090200;
}

*pFlashA_PortB_Data = LEDValue;
n++;
if(n > point)
{
    n = 0;
    point ++;
} if(point > 9)
{
    point = 10;
}
skip = true;
}
else
{
    LEDValue = 0x00;
    *pFlashA_PortB_Data = LEDValue;
    *pSIC_IMASK = 0x00090000;
    if(end == true)
    {
        skip = false;
        end = false;
    }
else
{
    if(n == 0)
    {
        skip = true;
    }
else
{
    skip = false;
}
}
void lose()
{
    unsigned char LED;
    LED = 0x3F;

    *pSIC_IMASK = 0x00080200;
    *pFlashA_PortB_Data = LED;

    loss = true;

    point = 0;
    userPoint = 0;
    n=0;
    skip = false;
    end = false;

    int m = 0;
    int temp;

    srand(compValue[0]);
    for(m=0; m<10; m++)
    {
        temp = (rand()%32 + 1)/8 +1;
        if(temp ==5)
        {
            compValue[m] = 4;
        }
        else
        {
            compValue[m] = temp;
        }
    }
}

void win()
{
    *pSIC_IMASK = 0x00080200;

    unsigned char ucActive_LED = 0x2A;

    *pFlashA_PortB_Data = ucActive_LED;

    wins = true;

    point = 0;
    userPoint = 0;
    n=0;
    skip = false;
    end = false;

    int m = 0;
    int temp;
srand(compValue[0]);

for(m=0; m<10; m++)
{
    temp = (rand()%32 + 1)/8 +1;
    if(temp ==5)
    {
        compValue[m] = 4;
    }
    else
    {
        compValue[m] = temp;
    }
}

The interrupts will be explained in the order that they should occur. The Timer0 interrupt occurs first. The variable LEDValue is initialized first. Since the variable n is initialized in main.c to 0, the value of compValue is tested first. When the value fits one of the tests, the appropriate LED is lit and the Sport0 interrupt is enabled. The n variable is then incremented. Since point has been initialized to 0, n is now greater than point so n is set to 0 and point is incremented. This means that the next set of values to be outputted will include the 0th and 1st values of the compValue array. The variable skip is then set to true. When skip is true, the LED’s are all turned off and the Sport0 interrupt is disabled. If n=0, and the computer has outputted all of the values for this set, skip remains true until the user inputs their guess. If n is not 0 and more values need to be outputted to form the sequence to be repeated, skip is set to false.

The Sport0 interrupt depends on the value of comp. When the Timer interrupt is enabled, the computer is outputting to audio and comp is true. When the user is inputting their values, comp is false. When comp is true, the audio output depends on the value of n. Since n is incremented at the end of the Timer0 interrupt, or, in the case of reaching the “end” or rather the beginning of the array, set to 0, n-1 will actually be the current value. In the case of n=0, n has just been set to 0 because it was incremented above point. Thus, point is the indicating variable now. Again, since point will have just been incremented, point-1 is the index of the current value. If comp is false, the user is inputting values. Since userPoint is incremented at the end of the routine, the value that indicates the current position in the array is userPoint-1. In the case of userPoint=0, the value we must depend on is point, which indicates the last index to be used. Since point has been incremented, we are interested in point-1. For each appropriate value, the correct data is read in, process data is called, and the correct data copied to the corresponding output buffer. In the case of a win or loss, one or the other input is outputted, so instead of one variable being copied to and from, two are used.

The FlagA interrupt immediately sets comp to false. When the game is running, the timer0 interrupt is masked, and then a value is read in to fill the userValue array. For each value inputted, the proper LED is lit. The userPoint variable is then incremented. If user point should equal point (which is one larger than n, the last value of compValue to be indicated), after being incremented, the values are then checked against the computer
values. If the user has entered the correct value(s), Skip is set to true, end is set to true and the timer is unmasked. This forces the time through one skip cycle before allowing it to continue outputting its values. This prevents the user’s input from being “cutoff” and not displayed. If the user inputs an incorrect value, the user loses. If the user manages to make it to 10 values without making a mistake, the user wins. In the even that a user has already won or lost, pressing PF8 resets the system.

The last two functions, win and lose are called when the user wins or loses, respectively. If the user loses, the timer interrupt is masked, all the LED’s are lit and channel0 is outputted. All the variables are set to their default values, srand is reseeded and a new set of values are generated for compValue. If the user wins, the timer is masked, the values reset, and every other LED is lit. Channel1 is outputted. Again, srand is reseeded and new values are calculated for compValue.

//Process_data.c
//Author: Elizabeth Nelson
//Date: 12-09-06
//Description: Processes Audio data
#include "setup.h"

void Process_Data(void)
{
    if(wins == true)
    {
        iChannel0LeftOut = iChannel0LeftIn;
        iChannel0RightOut = iChannel0RightIn;
    }
    else if(loss == true)
    {
        iChannel1LeftOut = iChannel1LeftIn;
        iChannel1RightOut = iChannel1RightIn;
    }
    else
    {
        iChannel0LeftOut = iChannel0LeftIn;
        iChannel0RightOut = iChannel0RightIn;
        iChannel1LeftOut = iChannel1LeftIn;
        iChannel1RightOut = iChannel1RightIn;
    }
}

If the user has won or lost, the data is copied from the correct channel without interference. During normal execution, all values are copied from the input storage to output storage. It is the Sport0 ISR which determines which is actually heard.

Conclusion
I wanted to design a project which was not a mere copy and paste job of previous labs or given audio example solutions. I wanted to come up with something that incorporated all elements of the labs, the LEDs, the buttons and the audio input and
output. I chose a dynamic challenge which required thought and careful implementation. I began with a full implementation, but it failed. So I decided rather, to layer my code. I wrote each section independently, testing and debugging along the way, and then combined them. It does fall short of an actual Simon game. First, there is no time limit on the user input. The user has as much time as they need. I contemplated using Timer1 which would be unmasked once a button was pressed. However, with added difficulty in testing and the required time it would take, it was decided that to time the user would require more effort than it was worth. Given more time, that would be the next “layer of code”.

Second, instead of using different speakers for audio output, different tones should be generated and outputted. I did make an attempt to implement that, but, since the Sport0 interrupt depends on an input of sound, and none of the calculated cosine or sine arrays I attempted worked, I abandoned that idea for alternating speakers. Lastly, while the computer turns off the LEDs and sound after each value, the user algorithm does not. This means that, for repeated values, there is no change in the output. Though the user knows when they should, and do press a button twice, out of a desire for continuity, that might also be addressed. Again, this might necessitate the use of a second timer to time the “off period” between. The wonderful thing about this project is that it allows for many different layers of complexity.

There are no bugs in this final code. There is the odd case of the rand and srand functions. Even though I originally seeded srand with the time function (time(0) returns the seconds since Jan 1 1970), the computer was generating the same set of “random” values each time. For some reason, either time was returning the same number or srand was seeding rand the same regardless. The bug was fixed by using the first value of compValue as the argument of srand. Although this means that each time the program is reset, the values will be different, it means that every time it is halted and then run, the same sets of numbers will be generated.

The following appendices are my previous versions of my code. The first is code I wrote at home, without the EZ KIT to test on, and which was never functional. The other two implementations (Final v2, and Final v3) are layers derived from the Blink and Talkthrough examples. They were eventually combined for the final solution, Final v4 which is what I discussed above.

**References/Bibliography**

Analog Devices – Audio Talkthrough Code  
- BF533 Flags code  
  [www.analog.com](http://www.analog.com)

Dr. Veton Këpuska – Class notes and resources  
[http://my.fit.edu/~vkepuska/ece3551](http://my.fit.edu/~vkepuska/ece3551)
Appendix A

Final v1

Setup.h

#ifndef __SETUP_DEFINED
#define __SETUP_DEFINED

#include <sys\exception.h>
#include <cdefBF533.h>

// addresses for Port B in Flash A
#define pFlashA_PortB_Dir (volatile unsigned char *)0x20270007
#define pFlashA_PortB_Data (volatile unsigned char *)0x20270005

// variables
extern int userValue[10];
extern int compValue[10];
extern int point;
extern int userPoint;
extern bool duplicate;
extern int count;

// in file Initialization.c
void Init_Flags(void);
void Init_Timers(void);
void Init_EBIU(void);
void Init_Flash(void);
void Init_Interrupts(void);

// in file ISRs.c
EX_INTERRUPT_HANDLER(Timer0_ISR);
EX_INTERRUPT_HANDLER(FlagA_ISR);
unsigned char compConvert(int);
void resetComp(void);

#endif //__SETUP_DEFINED

Main.c

#include "ccblkfn.h"
#include "sysreg.h"
#include "Setup.h"
#include <stdlib.h>

// variables
int userValue[10] = {0};
int compValue[10];
int point = 0;
int userPoint = 0;
bool duplicate = false;
int count = 0;

//main function
void main(void)
{

sysreg_write(reg_SYSCFG, 0x32); //Initialize System Configuration Register

Init_Flags();
Init_Timers();
Init_EBIU();
Init_Flash();
Init Interrupty();

srand(100);

int n = 0;
int temp;

for(n=0; n<10; n++)
{
    temp = (rand()%32 +1)/8;
    compValue[n] = temp;
}

static unsigned char LEDValue = 0x00;
*pFlashA_PortB_Data = LEDValue;

while(1);

Initialization.c
#include "Setup.h"

//This function configures PF8, PF9, PF10 and PF11 as input for/
//edge sensitive interrupt generation.
//
void Init_Flags(void)
{
    *pFIO_INEN  = 0x0F00; //enables the input buffer for PF8-
    *pFIO_DIR  = 0x0000; //sets all programmable flags to
    *pFIO_EDGE  = 0x0F00; //sets flags to edge sensitive
    *pFIO_MASKA_D = 0x0F00; //sets PF8-11 to write-1-to-set
    type registers
}

//This function initialises and enables the asynchronous memory banks
//for the External Bus Interface Unit (EBIU), so that access to Flash A/
//is possible. //It enables all banks and CLKOUT for memory region accesses, sets
write access time - 7 cycles,
//read access time - 11, 2 cycle hold time, 3 cycle setup time,
transition time of 4 cycles,
//ARDY is ignored for accesses and transition is completed if ARDY is
sampled low for banks 0 and 1//
void Init_EBIU(void)
{
  *pEBIU_AMBCTL0 = 0x7bb07bb0;  // sets Asynch Mem bank ctrl 0
  register
  *pEBIU_AMBCTL1 = 0x7bb07bb0;  // sets Asynch Mem bank ctrl 1
  register
  *pEBIU_AMGCTL = 0x000f;  // sets Asynch Mem global
  control register
}

// This function sets the pin direction of Port B in Flash A to output.
// The LEDs on the ADSP-BF533 EZ-KIT are connected to Port B.

void Init_Flash(void)
{
  *pFlashA_PortB_Dir = 0x3f;
}

// This function initialises the interrupts for (Timer0) and FlagA

void Init_Interrupts(void)
{
  // assign core IDs to interrupts
  *pSIC_IAR0 = 0xffffffff;
  *pSIC_IAR1 = 0xffffffff;
  *pSIC_IAR2 = 0xffff5ff4;  // Timer0 - ID4; FlagA -> ID5

  // assign ISRs to interrupt vectors
  register_handler(ik_ivg11, Timer0_ISR);  // Timer0 ISR -> IVG 11
  register_handler(ik_ivg12, FlagA_ISR);  // FlagA ISR -> IVG 12

  // enable Timer0 and FlagA interrupt
  *pSIC_IMASK = 0x00009000;  // enables interrupts from peripherals
}

// This function initialises Timer0 for PWM mode. It is used as
// a time limit for the user of 3 seconds
void Init_Timers(void)
{
  *pTIMER0_CONFIG  = 0x0019;
  *pTIMER0_PERIOD  = 0x02000000;
  *pTIMER0_WIDTH  = 0x01000000;
  *pTIMER_ENABLE  = 0x0001;
}

ISRsc

#include "Setup.h"
#include <stdlib.h>
// Timer Interrupt Subroutine

EX_INTERRUPT_HANDLER(Timer0_ISR)
{
    static unsigned char LEDValue;

    // confirm interrupt handling
    *pTIMER_STATUS = 0x0001;

    if(point == 0)
    {
        LEDValue = compConvert(compValue[point]);
        *pFlashA_PortB_Data = LEDValue;
        count ++;
    }
    else if(point < 10)
    {
        int m = 0;
        for(m=0; m<point; m++)
        {
            LEDValue = compConvert(compValue[m]);
            *pFlashA_PortB_Data = LEDValue;
        }
        count ++;
    }
    else
    {
        int l;
        for(l = 0; l<5; l++)
        {
            LEDValue = 0x37;
            *pFlashA_PortB_Data = LEDValue;
            LEDValue = 0x00;
            *pFlashA_PortB_Data = LEDValue;
        }
        resetComp();
        count = 0;
    }
}

// Flag Interrupt Subroutine

EX_INTERRUPT_HANDLER(FlagA_ISR)
{
    static unsigned char LEDValue;

    signed short pattern = *pFIO_FLAG_C;

    if(pattern == 0x0100)
    {
        // confirm interrupt handling
        *pFIO_FLAG_C = 0x0100;
        userValue[userPoint] = 1;
    }
}
LEDValue = 0x01;
}
else if(pattern == 0x0200)
{
   *pFIO_FLAG_C = 0x0200;
   userValue[userPoint] = 2;
   LEDValue = 0x02;
}
else if(pattern == 0x0400)
{
   *pFIO_FLAG_C = 0x0400;
   userValue[userPoint] = 3;
   LEDValue = 0x04;
}
else if (pattern == 0x0800)
{
   *pFIO_FLAG_C = 0x0800;
   userValue[userPoint] = 4;
   LEDValue = 0x08;
}
*pFlashA_PortB_Data = LEDValue;

if(userValue[userPoint] == compValue[userPoint]&& userPoint >0)
{
   count = 0;
   userPoint --;
}
else if(userValue[userPoint] == compValue[userPoint] && userPoint ==0)
{
   point++;  
   userPoint = point;
   count = 0;
}
else
{
   LEDValue = 0x3F;
   *pFlashA_PortB_Data = LEDValue;
   resetComp();
}
}

unsigned char compConvert(int c)
{
   if(c == 1)
   {
      return 0x01;
   }
   else if(c == 2)
   {

```c
return 0x02;
}  
else if(c == 3)  
{  
    return 0x04;
}
else if (c ==4)  
{  
    return 0x08;
}
}

void resetComp()  
{  
    int seed = 10 * compValue[0];  
    srand(seed);  
    int n;  
    int temp;  
    for(n=0; n<10; n++)  
    {  
        temp = (rand()%32 +1)/8;  
        compValue[n] = temp;
    }
    point = 0;  
    userPoint = point;  
    static unsigned char LEDValue = 0x00;  
    *pFlashA_PortB_Data = LEDValue;
}

Appendix B

Final v2
BF533 Flags.h
#ifndef __BF533_FLAGS_DEFINED
#define __BF533_FLAGS_DEFINED

//---------------------------------------------------------------------
------//
// Header files
//
//---------------------------------------------------------------------
------//
#include <sys\exception.h>
#include <cdefBF533.h>

//---------------------------------------------------------------------
------//
// Symbolic constants
//
```
// addresses for Port B in Flash A
#define pFlashA_PortB_Dir (volatile unsigned char *)0x20270007
#define pFlashA_PortB_Data (volatile unsigned char *)0x20270005

// Global variables
extern short sLight_Move_Direction;
extern short pattern;
extern int compValue[10];
extern int userValue[10];
extern int n;
extern unsigned char LEDValue;
extern int point;
extern bool skip;
extern bool loss;
extern int userPoint;
extern bool end;
extern bool wins;

// Prototypes
void Init_Flags(void);
void Init_Timers(void);
void Init_EBIU(void);
void Init_Flash(void);
void Init_Interrupts(void);

EX_INTERRUPT_HANDLER(Timer0_ISR);
EX_INTERRUPT_HANDLER(FlagA_ISR);
void lose(void);
void win(void);

#endif //__BF533_FLAGS_DEFINED

Main.c
/*******************************************************************************
******
**
** Name: FIO pins, Interrupts, and Timer in C
**
**
******************************************************************************
#include "BF533 Flags.h"
#include "ccblkfn.h"
#include "sysreg.h"
#include <stdlib.h>
#include <time.h>

// Variables

short sLight_Move_Direction = 0;
short pattern = 0x0000;
int compValue[10];
int userValue[10] = {0};
int n = 0;
int point = 0;
bool skip = false;
int userPoint = 0;
bool loss = false;
bool end = false;
bool wins = false;

void main(void)
{
}
sysreg_write(reg_SYSCFG, 0x32); //Initialize System Configuration Register
    Init_Flags();
    Init_Timers();
    Init_EBIU();
    Init_Flash();
    Init_Interrupts();

    int m = 0;
    int temp;

    srand(time(0));

    for(m=0; m<10; m++)
    {
        temp = (rand()%32 + 1)/8 +1;
        if(temp ==5)
        {
            compValue[m] = 4;
        }
        else
        {
            compValue[m] = temp;
        }
    }

    while(1);
}

Initialization.c
#include "BF533 Flags.h"

//------------------------------------------------------------
// Function: Init_Flags
// // // Parameters: None
// // // Return: None
// // // Description: This function configures PF8 as input for edge
// sensitive interrupt generation.
// // The switch connected to PF8 (SW7) can be used to change the
// direction of the moving light.
void Init_Flags(void)
{
    *pFIO_INEN  = 0x0F00;
    *pFIO_DIR  = 0x0000;
    *pFIO_EDGE  = 0x0F00;
    *pFIO_MASKA_D = 0x0F00;
}

void Init_Timers(void)
{
    *pTIMER0_CONFIG  = 0x0019;
    *pTIMER0_PERIOD  = 0x02000000;
    *pTIMER0_WIDTH  = 0x01000000;
    *pTIMER_ENABLE  = 0x0001;
}

void Init_EBIU(void)
{
    *pTIMER0_CONFIG  = 0x0019;
    *pTIMER0_PERIOD  = 0x02000000;
    *pTIMER0_WIDTH  = 0x01000000;
    *pTIMER_ENABLE  = 0x0001;
}
that access to Flash A is possible.

---

```c
void Init_EBIU(void)
{
    *pEBIU_AMBCTL0 = 0x7bb07bb0;
    *pEBIU_AMBCTL1 = 0x7bb07bb0;
    *pEBIU_AMGCTL = 0x000f;
}
---
```

---

// Function: Init_Flash
//
// Parameters: None
//
// Return: None
//
// Description: This function sets the pin direction of Port B in Flash A to output.
// The LEDs on the ADSP-BF533 EZ-KIT are connected to Port B.

---

```c
void Init_Flash(void)
{
    *pFlashA_PortB_Dir = 0x3f;
}
---
```

---

// Function: Init Interrupts
//
// Parameters: None
//
// Return: None
//
// Description: This function initialises the interrupts for Timer0 and FlagA (PF8).
void Init_Interrupts(void)
{
    // assign core IDs to interrupts
    *pSIC_IAR0 = 0xffffffff;
    *pSIC_IAR1 = 0xffffffff;
    *pSIC_IAR2 = 0xffff5ff4;     // Timer0 -
> ID4; FlagA -> ID5

    // assign ISRs to interrupt vectors
    register_handler(ik_ivg11, Timer0_ISR);  // Timer0 ISR ->
    IVG 11
    register_handler(ik_ivg12, FlagA_ISR);  // FlagA ISR ->
    IVG 12

    // enable Timer0 and FlagA interrupt
    *pSIC_IMASK = 0x00090000;
}

ISR.c
#include "BF533 Flags.h"
#include <stdlib.h>
#include <time.h>

// Function: Timer0_ISR
// Parameters: None
// Return: None
// Description: This ISR is executed every time Timer0 expires. The old LED pattern is shifted by one; the
direction depends on the state of sLight_Move_Direction, which is changed in FlagA_ISR.

EX_INTERRUPT_HANDLER(Timer0_ISR)
{
    unsigned char LEDValue = 0x00;
    // confirm interrupt handling
    *pTIMER_STATUS = 0x0001;
if (skip == false) {
    if (compValue[n] == 1) {
        LEDValue = 0x01;
    } else if (compValue[n] == 2) {
        LEDValue = 0x02;
    } else if (compValue[n] == 3) {
        LEDValue = 0x04;
    } else if (compValue[n] == 4) {
        LEDValue = 0x08;
    }

    *pFlashA_PortB_Data = LEDValue;
    n++;
    if (n > point) {
        n = 0;
        point ++;
    }
    if (point > 9) {
        point = 10;
    }
    skip = true;
} else {
    LEDValue = 0x00;
    *pFlashA_PortB_Data = LEDValue;
    if (end == true) {
        skip = false;
        end = false;
    } else {
        if (n == 0) {
            skip = true;
        } else {
            skip = false;
        }
    }
}
Function: FlagA_ISR

Parameters: None

Return: None

Description: This ISR is called every time the button connected to PF8 is pressed.

The state of flag sLight_Move_Direction is changed, so the shift-direction for the LED pattern in Timer0_ISR changes.

EX_INTERRUPT_HANDLER(FlagA_ISR)
{
    if(loss == false && wins == false)
    {
        unsigned char Guess;

        pattern = *pFIO_FLAG_C;
        *pSIC_IMASK = 0x00080000;

        if(pattern == 0x0100)
        {
            // confirm interrupt handling
            *pFIO_FLAG_C = 0x0100;

            userValue[userPoint] = 1;
            Guess = 0x01;
            *pFlashA_PortB_Data = Guess;
        }
        else if(pattern == 0x0200)
        {
            *pFIO_FLAG_C = 0x0200;

            userValue[userPoint] = 2;
            Guess = 0x02;
        }
    }
}
else if (pattern == 0x0400) {
    *pFIO_FLAG_C = 0x0400;
    userValue[userPoint] = 3;
    Guess = 0x04;
    *pFlashA_PortB_Data = Guess;
}
else if (pattern == 0x0800) {
    *pFIO_FLAG_C = 0x0800;
    userValue[userPoint] = 4;
    Guess = 0x08;
    *pFlashA_PortB_Data = Guess;
}

userPoint++;

if (userPoint == point) {
    userPoint = 0;
    int i;
    for (i = 0; i < point; i++) {
        if (userValue[i] == compValue[i]) {
            skip = true;
            end = true;

            *pTIMER_STATUS = 0x0001;
            *pSIC_IMASK = 0x00090000;
        } else {
            lose();
        }
        if (point == 10) {
            win();
        }
    }
} else {
    pattern = *pFIO_FLAG_C;
    if (pattern == 0x0100) {
        *pFIO_FLAG_C = 0x0100;
    }
}
wins = false;
loss = false;

*pSIC_IMASK = 0x00090000;

void lose()
{
    unsigned char LED;
    LED = 0x3F;

    *pSIC_IMASK = 0x00080000;
    *pFlashA_PortB_Data = LED;
    loss = true;
    point = 0;
    userPoint = 0;
    n=0;
    skip = false;
    end = false;

    int m = 0;
    int temp;

    srand(time(0));

    for(m=0; m<10; m++)
    {
        temp = (rand()%32 + 1)/8 +1;
        if(temp ==5)
        {
            compValue[m] = 4;
        }
        else
        {
            compValue[m] = temp;
        }
    }
}

void win()
{
    *pSIC_IMASK = 0x00080000;
    unsigned char ucActive_LED = 0x2A;
    *pFlashA_PortB_Data = ucActive_LED;
wins = true;
point = 0;
userPoint = 0;
n=0;
skip = false;
end = false;

int m = 0;
int temp;

srand(time(0));

for(m=0; m<10; m++)
{
    temp = (rand()%32 + 1)/8 +1;
    if(temp ==5)
    {
        compValue[m] = 4;
    }
    else
    {
        compValue[m] = temp;
    }
}

Appendix C

Final v3
Talkthrough.h

 ifndef __Talkthrough_DEFINED
    define __Talkthrough_DEFINED

 //=================================================================================================
 /// Header files
 //=================================================================================================
#ifndef EXCEPTION_H
#include <sys\exception.h>
#include <cdefBF533.h>

 //=================================================================================================
 /// Symbolic constants
 //=================================================================================================
#define pFlashA_PortA_Dir (volatile unsigned char *)0x20270006
#define pFlashA_PortA_Data  (volatile unsigned char *)0x20270004
// addresses for Port B in Flash A
#define pFlashA_PortB_Dir (volatile unsigned char *)0x20270007
#define pFlashA_PortB_Data (volatile unsigned char *)0x20270005

// names for codec registers, used for sCodec1836TxRegs[]
#define DAC_CONTROL_1 0x0000
#define DAC_CONTROL_2 0x1000
#define DAC_VOLUME_0 0x2000
#define DAC_VOLUME_1 0x3000
#define DAC_VOLUME_2 0x4000
#define DAC_VOLUME_3 0x5000
#define DAC_VOLUME_4 0x6000
#define DAC_VOLUME_5 0x7000
#define ADC_0_PEAK_LEVEL 0x8000
#define ADC_1_PEAK_LEVEL 0x9000
#define ADC_2_PEAK_LEVEL 0xA000
#define ADC_3_PEAK_LEVEL 0xB000
#define ADC_CONTROL_1 0xC000
#define ADC_CONTROL_2 0xD000
#define ADC_CONTROL_3 0xE000

// names for slots in ad1836 audio frame
#define INTERNAL_ADC_L0 0
#define INTERNAL_ADC_R0 2
#define INTERNAL_DAC_L0 0
#define INTERNAL_DAC_R0 2
#define INTERNAL_ADC_L1 1
#define INTERNAL_ADC_R1 3
#define INTERNAL_DAC_L1 1
#define INTERNAL_DAC_R1 3

// size of array sCodec1836TxRegs
#define CODEC_1836_REGS_LENGTH 11

// SPI transfer mode
#define TIMOD_DMA_TX 0x0003

// SPORT0 word length
#define SLEN_24 0x0017

// DMA flow mode
#define FLOW_1 0x1000

// Global variables

extern int iChannel0LeftIn;
extern int iChannel0RightIn;
extern int iChannel0LeftOut;
extern int iChannel1RightOut;
extern int iChannel1LeftIn;
extern int iChannel1RightIn;
extern int iChannel1LeftOut;
extern int iChannel1RightOut;
extern volatile short sCodec1836TxRegs[];
extern volatile int iRxBuffer1[];
extern volatile int iTxBuffer1[];
extern int arraySig[100];
extern int point;
extern short pattern;
extern unsigned char LEDValue;
extern int count;
extern int count1;
extern int count2;
extern int count3;

//---------------------------------------------------------------------
-----//
// Prototypes

typedef short int SHORT_TYPE;

void Init_EBIU(void);
void Init_Flash(void);
void Init1836(void);
void Init_Sport0(void);
void Init_DMA(void);
void Init_Interrupts(void);
void Enable_DMA_Sport(void);
void Enable_DMA_Sport0(void);
void Init_Flags(void);

// in file Process_data.c
void Process_Data(void);

// in file ISRs.c
EX_INTERRUPT_HANDLER(Sport0_RX_ISR);
EX_INTERRUPT_HANDLER(FlagA_ISR);

// in file ISRs.c
EX_INTERRUPT_HANDLER(Sport0_RX_ISR);
EX_INTERRUPT_HANDLER(FlagA_ISR);

#endif //__Talkthrough_DEFINED

Main.c

// Name: Talkthrough for the ADSP-BF533 EZ-KIT Lite

// (C) Copyright 2003 - Analog Devices, Inc. All rights reserved.
// Project Name: BF533 C Talkthrough I2S
//
// Date Modified: 04/03/03 HD Rev 1.0
//
// Software: VisualDSP++3.1
//
// Hardware: ADSP-BF533 EZ-KIT Board
//
// Connections: Connect RSCLK0 to TSCLK0 together (Turn SW9 pin 6 on) //
// Connect RFS0 to TFS0 together (Turn SW9 pin 5 ON) //
// (such as a radio) to the Audio //
// Connect an input source //
// input jack and an output source (such as //
// headphones) to //
// the Audio output jack //
//
// Purpose: This program sets up the SPI port on the ADSP-BF533 to //
// configure the AD1836 codec. The SPI port //
// is disabled //
// the codec are //
// after initialization. The data to/from //
// transferred over SPORT0 in I2S mode //
//
//---------------------------------------------------------------
#include "Talkthrough.h"
#include "sysreg.h"
#include "ccblkfn.h"
#include <math.h>

// Variables

// Description: The variables iChannelxLeftIn and iChannelxRightIn contain //
the data coming from the codec AD1836. The playback data are written into the variables iChannelxLeftOut and iChannelxRightOut respectively, which are then sent back to the codec in the SPORT0 ISR. The values in the array iCodec1836TxRegs can be modified to according to the AD1885 data sheet.

//------------------------------------------------------------------------------
// left input data from ad1836
int iChannel0LeftIn, iChannel1LeftIn;
// right input data from ad1836
int iChannel0RightIn, iChannel1RightIn;
// left output data for ad1836
int iChannel0LeftOut, iChannel1LeftOut;
// right output data for ad1836
int iChannel0RightOut, iChannel1RightOut;
// array for registers to configure the ad1836
// names are defined in "Talkthrough.h"
volatile short sCodec1836TxRegs[CODEC_1836_REGS_LENGTH] = {
    DAC_CONTROL_1  | 0x000,
    DAC_CONTROL_2  | 0x000,
    DAC_VOLUME_0   | 0x3ff,
    DAC_VOLUME_1   | 0x3ff,
    DAC_VOLUME_2   | 0x3ff,
    DAC_VOLUME_3   | 0x3ff,
    DAC_VOLUME_4   | 0x000,
    DAC_VOLUME_5   | 0x000,
    ADC_CONTROL_1  | 0x000,
    ADC_CONTROL_2  | 0x000,
    ADC_CONTROL_3  | 0x000
};
// SPORT0 DMA transmit buffer
volatile int iTxBuffer1[4];
// SPORT0 DMA receive buffer
volatile int iRxBuffer1[4];
int point = 0;
int arraySig[100];
short pattern;
int count = 0;
int count1 = 0;
int count2 = 0;
int count3 = 0;

//------------------------------------------------------------------------------
// Function: main
//
// Description: After calling a few initialization routines, main() just
// waits in a loop forever. The code to process
// the incoming data can be placed in the function
// Process_Data() in the file "Process_Data.c".

void main(void)
{
    sysreg_write(reg_SYSCFG, 0x32); //Initialize System Configuration Register
    Init_EBIU();
    Init_Flash();
    Init1836();
    Init_Sport0();
    Init_DMA();
    Init_Interrupts();
    Init_Flags();
    Enable_DMA_Sport0();

    int n;
    for(n=0; n<100; n+=2)
    {
        arraySig[n] = 1;
    }

    int m;
    for(m=1; m<100; m+=2)
    {
        arraySig[m] = -1;
    }

    while(1);
}

Initialize.c
#include "Talkthrough.h"

// Function: Init_EBIU
//
// Description: This function initializes and enables asynchronous
// memory banks in External Bus Interface Unit so that
// Flash A can be //
void Init_EBIU(void)
{
    *pEBIU_AMBCTL0 = 0x7bb07bb0;
    *pEBIU_AMBCTL1 = 0x7bb07bb0;
    *pEBIU_AMGCTL = 0x000f;
}

void Init_Flash(void)
{
    *pFlashA_PortA_Dir = 0x1;
    *pFlashA_PortB_Dir = 0x3f;
}

void Init1836(void)
{  
    int i;
    int j;
    static unsigned char ucActive_LED = 0x01;
    // write to Port A to reset AD1836
    *pFlashA_PortA_Data = 0x00;
    // write to Port A to enable AD1836
*pFlashA_PortA_Data = ucActive_LED;

// wait to recover from reset
for (i=0; i<0xf0000; i++) asm("nop;");

// Enable PF4
*pSPI_FLG = FLS4;
// Set baud rate SCK = HCLK/(2*SPIBAUD) SCK = 2MHz
*pSPI_BAUD = 16;
// configure spi port
*pSPI_CTL = TIMOD_DMA_TX | SIZE | MSTR;

// Set up DMA5 to transmit
// Map DMA5 to SPI
*pDMA5_PERIPHERAL_MAP = 0x5000;

// Configure DMA5
// 16-bit transfers
*pDMA5_CONFIG = WDSIZE_16;
// Start address of data buffer
*pDMA5_START_ADDR = sCodec1836TxRegs;
// DMA inner loop count
*pDMA5_X_COUNT = CODEC_1836_REGS_LENGTH;
// Inner loop address increment
*pDMA5_X_MODIFY = 2;

// enable DMAs
*pDMA5_CONFIG = (*pDMA5_CONFIG | DMAEN);
// enable spi
*pSPI_CTL = (*pSPI_CTL | SPE);

// wait until dma transfers for spi are finished
for (j=0; j<0xaff0; j++) asm("nop;");

// disable spi
*pSPI_CTL = 0x0000;

}//

// Function: Init_Sport0
    //
    // Description: Configure Sport0 for I2S mode, to transmit/receive
data //
    //    to/from the AD1836. Configure Sport for
    //    external clocks and //
    //    frame syncs.
    //
    //------------------------------------------------------------------------

void Init_Sport0(void)
{
    // Sport0 receive configuration
// External CLK, External Frame sync, MSB first, Active Low
// 24-bit data, Stereo frame sync enable
*pSPORT0_RCR1 = RFSR | LRFS | RCKFE;
pSPORT0_RCR2 = SLEN_24 | RXSE | RSFSE;

// Sport0 transmit configuration
// External CLK, External Frame sync, MSB first, Active Low
// 24-bit data, Secondary side enable, Stereo frame sync enable
*pSPORT0_TCR1 = TFSR | LTFS | TCKFE;
pSPORT0_TCR2 = SLEN_24 | TXSE | TSFSE;

//---------------------------------------------------------------------
-----//
// Function: Init_DMA
//
// Description: Initialize DMA1 in autobuffer mode to receive and
// DMA2 in  
//   autobuffer mode to transmit
//
//---------------------------------------------------------------------
-----//
void Init_DMA(void)
{
    // Set up DMA1 to receive
    // Map DMA1 to Sport0 RX
    *pDMA1_PERIPHERAL_MAP = 0x1000;

    // Configure DMA1
    // 32-bit transfers, Interrupt on completion, Autobuffer mode
    *pDMA1_CONFIG = WNR | WDSIZE_32 | DI_EN | FLOW_1;
    // Start address of data buffer
    *pDMA1_START_ADDR = iRxBuffer1;
    // DMA inner loop count
    *pDMA1_X_COUNT = 4;
    // Inner loop address increment
    *pDMA1_X_MODIFY = 4;

    // Set up DMA2 to transmit
    // Map DMA2 to Sport0 TX
    *pDMA2_PERIPHERAL_MAP = 0x2000;

    // Configure DMA2
    // 32-bit transfers, Autobuffer mode
    *pDMA2_CONFIG = WDSIZE_32 | FLOW_1;
    // Start address of data buffer
    *pDMA2_START_ADDR = iTxBuffer1;
    // DMA inner loop count
    *pDMA2_X_COUNT = 4;
    // Inner loop address increment
    *pDMA2_X_MODIFY = 4;
}
void Enable_DMA_Sport0(void)
{
    // enable DMAs
    *pDMA2_CONFIG = (*pDMA2_CONFIG | DMAEN);
    *pDMA1_CONFIG = (*pDMA1_CONFIG | DMAEN);

    // enable Sport0 TX and RX
    *pSPORT0_TCR1  = (*pSPORT0_TCR1 | TSPEN);
    *pSPORT0_RCR1  = (*pSPORT0_RCR1 | RSPEN);
}

void Init_Flags(void)
{
    *pFIO_INEN  = 0x0F00;
    *pFIO_DIR  = 0x0000;
    *pFIO_EDGE  = 0x0F00;
    *pFIO_MASKA_D = 0x0F00;
}

void Init_Interrupts(void)
{
    // Set Sport0 RX (DMA1) interrupt priority to 2 = IVG9
    *pSIC_IAR0 = 0xffffffff;
    *pSIC_IAR1 = 0xffffffff2f;
    *pSIC_IAR2 = 0xffffffff5fff; //FlagA -> ID5 = IVG12

    // assign ISRs to interrupt vectors
    // Sport0 RX ISR -> IVG 9
    register_handler(ik_ivg9, Sport0_RX_ISR);
    register_handler(ik_ivg12, FlagA_ISR);

    // enable Sport0 RX interrupt and flagA interrupt
    *pSIC_IMASK = 0x00080000;
}
ISR.c
#include "Talkthrough.h"

//---------------------------------------------------------------------
-----//
// Function: Sport0_RX_ISR
//
// Description: This ISR is executed after a complete frame of input data
// has been received. The new samples are stored in
// iChannel0LeftIn, iChannel1RightIn, iChannel0LeftIn and iChannel1RightIn respectively. Then the
// function Process_Data() is called in which user code can be executed.
// After that the processed values are copied from the variables iChannel0LeftOut, iChannel0RightOut,
// iChannel1LeftOut and iChannel1RightOut into the dma transmit buffer.
//
// EX_INTERRUPT_HANDLER(Sport0_RX_ISR)
{
    // confirm interrupt handling
    *pDMA1_IRQ_STATUS = 0x0001;

    // copy input data from dma input buffer into variables
    if(pattern == 0x0100)
    {
        iChannel0LeftIn = iRxBuffer1[INTERNAL_ADC_L0];
        Process_Data();
        iTxBuffer1[INTERNAL_DAC_L0] = iChannel0LeftOut;
    }
    else if(pattern == 0x0200)
    {
        iChannel0RightIn = iRxBuffer1[INTERNAL_ADC_R0];
        Process_Data();
        iTxBuffer1[INTERNAL_DAC_R0] = iChannel0RightOut;
    }
    else if(pattern == 0x0400)
    {
        iChannel1LeftIn = iRxBuffer1[INTERNAL_ADC_L1];
        Process_Data();
        iTxBuffer1[INTERNAL_DAC_L1] = iChannel1LeftOut;
    }
}
Process_Data();
    iTxBuffer1[INTERNAL_DAC_L1] = iChannel1LeftOut;
} else if(pattern == 0x0800) {
    iChannel1RightIn = iRxBuffer1[INTERNAL_ADC_R1];
    Process_Data();
    iTxBuffer1[INTERNAL_DAC_R1] = iChannel1RightOut;
} //iChannel0LeftIn = arraySig[point];
// copy processed data from variables into dma output buffer
point++;

if(point >99) {
    point = 0;
}

EX_INTERRUPT_HANDLER(FlagA_ISR) {
    unsigned char LEDValue;
    pattern = *pFIO_FLAG_C;
    if(pattern == 0x0100) {
        *pFIO_FLAG_C = 0x0100;
        if(count == 0) {
            LEDValue = 0x01;
            *pFlashA_PortB_Data = LEDValue;
            *pSIC_IMASK = 0x00080200;
            count++;
        }
        else {
            LEDValue = 0x00;
            *pFlashA_PortB_Data = LEDValue;
            *pSIC_IMASK = 0x00080000;
            count = 0;
        }
    }
    else if(pattern == 0x0200) {
        *pFIO_FLAG_C = 0x0200;
        if(count1 == 0) {
            LEDValue = 0x02;
*pFlashA_PortB_Data = LEDValue;
*pSIC_IMASK = 0x00080200;
count1++;
} else {
    LEDValue = 0x00;
    *pFlashA_PortB_Data = LEDValue;
    *pSIC_IMASK = 0x00080000;
    count1 = 0;
}
}
else if(pattern == 0x0400) {
    *pFIO_FLAG_C = 0x0400;
    if(count2 == 0) {
        LEDValue = 0x04;
        *pFlashA_PortB_Data = LEDValue;
        *pSIC_IMASK = 0x00080200;
        count2++;
    } else {
        LEDValue = 0x00;
        *pFlashA_PortB_Data = LEDValue;
        *pSIC_IMASK = 0x00080000;
        count2 = 0;
    }
} else if(pattern == 0x0800) {
    *pFIO_FLAG_C = 0x0800;
    if(count3 == 0) {
        LEDValue = 0x08;
        *pFlashA_PortB_Data = LEDValue;
        *pSIC_IMASK = 0x00080200;
        count3++;
    } else {
        LEDValue = 0x00;
        *pFlashA_PortB_Data = LEDValue;
        *pSIC_IMASK = 0x00080000;
    }
}
Process_data.c
#include "Talkthrough.h"

void Process_Data(void)
{
    iChannel0LeftOut = iChannel0LeftIn;
    iChannel0RightOut = iChannel0RightIn;
    iChannel1LeftOut = iChannel1LeftIn;
    iChannel1RightOut = iChannel1RightIn;
}
Audio Generation

Valerie Bastien
December 10, 2006

Microcomputer Systems I
Dr. Kepuska
Overview

- Purpose
- Audio Generation
- Code
  - Talkthrough.h
  - Main.c
  - Initialize.c
  - ISR.c
  - Process_data.c
- Demo
Purpose

- Default state: plays “Silent Night”
- Press PF8 once: plays “Away in a Manger”
- Press PF9: speeds up song being played
- Press PF10: slows down song being played
Audio Generation

$x[n] = A \sin \left[ 2\pi \left( \frac{f_0}{f_s} \right) n \right] = A \sin \left[ \pi \left( \frac{f_0}{f_s/2} \right) n \right] = A \sin[\pi F_0 n] = A \sin[\omega_0 n]$

$F_0 = \left( \frac{f_0}{f_s/2} \right), \quad -1 \leq F_0 \leq 1$

C, D, E, F, G, A, B, C

Frequency = $440 \times 2^{n/12}$ Hz
extern char eight; //initialize counter for our switch
extern int sn; //initialize counter for silent night
extern int am; //initialize counter for away in a manger
extern int speedup; //initialize counter for speeding up
extern int slowdown; //initialize counter for slowing down

EX_INTERRUPT_HANDLER(Sport0_RX_ISR);
EX_INTERRUPT_HANDLER(FlagA_ISR);
EX_INTERRUPT_HANDLER(Timer0_ISR);

...
... char eight=0x00; //set counter for PF8
int sn=0;
int am=0;
int speedup=0;
int slowdown=0;
...
void Init_Interrupts(void)
{
    // Set System Interrupt Assignment Registers
    // 2 for Sport0 RX, 5 for Flag A, and 4 for Timer0
    *pSIC_IAR0 = 0xffffffff;
    *pSIC_IAR1 = 0xffffff2f;
    *pSIC_IAR2 = 0xffff5ff4;
    // assign ISRs to interrupt vectors
    register_handler(ik_ivg9, Sport0_RX_ISR);
    register_handler(ik_ivg12, FlagA_ISR);
    register_handler(ik_ivg11, Timer0_ISR);
    // enable interrupts Timer0, Sport0 RX, and Flag A
    *pSIC_IMASK = 0x00090200;
}

...
... void Init_Flags(void)
{
    *pFIO_INEN = 0x0700;  //need PF8, PF9, and PF10
    *pFIO_DIR = 0x0000;
    *pFIO_EDGE = 0x0700;
    *pFIO_MASKA_D = 0x0700;
}
...

void Init_Timers(void)
{
    *pTIMER0_CONFIG = 0x0019;
    *pTIMER0_PERIOD = 0x03000000;
    *pTIMER0_WIDTH = 0x00200000;
    *pTIMER_ENABLE = 0x0001;
}
...
ISR.c

```c
...
EX_INTERRUPT_HANDLER(FlagA_ISR)
...
if(*pFIO_FLAG_C == 0x0100) // switch songs
{
    if(eight == 0) // switch to "Away in a Manger"
    {
        *pFlashA_PortB_Data = 0x01; // turn on LED4
        eight = eight + 1;
        am = 0; // starts song at the beginning
        *pTIMER0_PERIOD = 0x03000000;
    }
    else if (eight == 1) // switch to "Silent Night"
    {
        // turn on LED5
        *pFlashA_PortB_Data = 0x00; // turn off LED4
        eight = 0;
        sn = 0; // starts song at the beginning
        *pTIMER0_PERIOD = 0x02000000;
    }
}
...
```
ISR.c (cont’d)

else if(*pFIO_FLAG_C == 0x0200)//speed up
{
    if(*pTIMER0_PERIOD>0x00f00000)//bottom limit
    {
        speedup=speedup+0x00100000;
    }
}

else if(*pFIO_FLAG_C == 0x0400)//slow down
{
    if(*pTIMER0_PERIOD<0xff000000)//upper limit
    {
        slowdown=slowdown+0x00100000;
    }
}
ISR.c (cont’d)

...  
EX_INTERRUPT_HANDLER(Timer0_ISR)
 ...
 else if(sn==3|sn==15|sn==18|sn==24|sn==27|sn==33|sn==41)
 { 
    sn++;  
    *pTIMER0_PERIOD=0x03000000+slowdown-speedup;
 }
 ...
 else if(am==0|am==11|am==22|am==33)
 { 
    am++;  
    *pTIMER0_PERIOD=0x03000000+slowdown-speedup;
 }
 ...
 ...
... else if(sn==4)//Night (D4) {
    *pFlashA_PortB_Data = 0x10;
    r %= 163;

    y=5*x*sinf(0.038441 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
...


Demo
Questions?
Final Project

Sound Generation

Microcomputer Systems I

Dr. Kepuska

Valerie Bastien

12/10/06
Purpose

The purpose of this project was to generate two songs on the Blackfin. The buttons gave the user the option of which song and the speed of the song.

Equipment

Blackfin EZ-Kit Lite
Visual DSP++

Procedure

Part 1 Create the Project
1. Before powering on the board, I made sure that switch SW9 pin1, pin2, and pin3, pin4, pin5, and pin6 were turned on. All pins in SW9 should be on. See Figure 1.

![Figure 1](image)

2. I copied the project from Lab 9 to “u:\ece3551\labs\final.”

Part 2 Modify the Project
1. In this project, I removed all the filtering in Process_data.c. I used Process_data.c to generate the songs.
2. I added the Timer0 interrupt. Now I have 3 interrupts (FlagA, Sport0, and Timer0).
3. To change the songs and the speed of the songs I programmed the buttons to do the following:
   - PF8 controls which song is played:
     - In the beginning, LED4 is off and Silent Night plays.
     - When PF8 is pressed LED4 turns on and Away in a Manger plays.
     - When PF8 is pressed a second time, LED4 turns off, Silent Night plays, and the pattern starts over.
   - PF9 increases the speed of the song played:
- Pressing PF9 increases the speed until a preset limit of the Timer period is reached.
- PF10 decreases the speed of the song played:
  - Pressing PF10 decreases the speed until a preset limit of the Timer period is reached.

4. We modified the code in Talkthrough.h, main.c, Initialize.c, ISRs.c and Process_data.c.
5. We connected the equipment for the experiment as seen in Figure 2.
Discussion

In this project I hard-coded two songs to be generated by the Blackfin 533 Processor. While the song plays, the user can follow along by watching the LEDs. The first LED represents which song is playing (off – Silent Night, on – Away in a Manger). The other 5 LEDs follow along the playing of the song. Button PF8 changes the songs, button PF9 speeds up the song, and button PF10 slows down the song.

To prepare for the sound generation, I first created a table (Table 1 in Appendix B) that listed some musical notes, their frequencies, the radial frequency (for the digital sinusoid used in programming), and the maximum for the counter n. The n max is when the argument inside the digital sinusoid is 2pi. For n’s greater than the n max, you start from the beginning.

I used the musical notes from the sheet music (Appendix B Figures 8 and 9) to help determine timing. I would find a value for TIMER0_PERIOD that represented the timing of the first note, and then changed the timing for other notes accordingly. For example, in “Away in a Manger” the first note is a quarter note. I set the TIMER0_PERIOD for that note to 02000000. For an eighth note (half a quarter note), I set the TIMER0_PERIOD to 01000000 (half of that for the quarter note). For “Silent Night,” however I did the timing by ear. I didn’t like the timing that came about from looking at the sheet music.

The tricky thing with the timing is you had to reset the TIMER0_PERIOD one note ahead of time. The changed period doesn’t take effect until the next time the interrupt flag is enabled. So, when the song is changed, or when a song starts over, there is a slight delay so the TIMER0_PERIOD can be set to the correct value for the first note of the song about to be played.

I added the math.h header file to Process_data.c so it could use the sine function for the sound generation. I used the float version of the sine function because it worked.

I referenced the ADSP-BF533 Blackfin® Processor Hardware Reference to make sure I enabled all the Interrupts correctly (see Figures 1 through 5 from Appendix A). I was using 3 interrupts: SPORT0 RX (for the audio), FLAGA (programmable flag interrupt for the buttons), and TIMER0 (for the timing). I used the Hardware Reference to know what register handler was needed for each interrupt and what bits to set for SIC_IMASK, SIC_IAR0, SIC_IAR1, and SIC_IAR2.

The file Process_data.c is the file that provides the data for the output. Each note for the two songs is hard-coded here. They are labeled by note and word to keep track. In the file ISR.c the interrupts are processed. For the TIMER0 interrupt, the counter for the note of a particular song is either incremented or set to zero. Also, the TIMER0_PERIOD is changed to prepare for the next note. For the FLAGA interrupt, the song can be changed or the speed. The SPORT0_RX interrupt takes the data from Process_data.c and sends it to the audio output.
Recommendations

Some improvements to this project could be sending a chord to the output. Also, it would be cool to change the tone generated so it has a more natural quality (adding vibrato, etc.). Another button that could be added would be to change the octave of the song being played.
Appendix A
Interrupt Information

Figure 1: System and Core Event Mapping

<table>
<thead>
<tr>
<th>Event Source</th>
<th>Core Event Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLL Wakeup Interrupt</td>
<td>IVG7</td>
</tr>
<tr>
<td>DMA Error (generic)</td>
<td></td>
</tr>
<tr>
<td>PPI Error Interrupt</td>
<td></td>
</tr>
<tr>
<td>SPORT0 Error Interrupt</td>
<td></td>
</tr>
<tr>
<td>SPORT1 Error Interrupt</td>
<td></td>
</tr>
<tr>
<td>SPI Error Interrupt</td>
<td></td>
</tr>
<tr>
<td>UART Error Interrupt</td>
<td></td>
</tr>
<tr>
<td>Real-Time Clock Interrupts</td>
<td>IVG8</td>
</tr>
<tr>
<td>DMA0 Interrupt (PPI)</td>
<td></td>
</tr>
<tr>
<td>DMA1 Interrupt (SPORT0 RX)</td>
<td>IVG9</td>
</tr>
<tr>
<td>DMA2 Interrupt (SPORT0 TX)</td>
<td></td>
</tr>
<tr>
<td>DMA3 Interrupt (SPORT1 RX)</td>
<td></td>
</tr>
<tr>
<td>DMA4 Interrupt (SPORT1 TX)</td>
<td></td>
</tr>
<tr>
<td>DMA5 Interrupt (SPI)</td>
<td>IVG10</td>
</tr>
<tr>
<td>DMA6 Interrupt (UART RX)</td>
<td></td>
</tr>
<tr>
<td>DMA7 Interrupt (UART TX)</td>
<td></td>
</tr>
<tr>
<td>Timer0, Timer1, Timer2 Interrupts</td>
<td>IVG11</td>
</tr>
<tr>
<td>Programmable Flags Interrupt A/B</td>
<td>IVG12</td>
</tr>
<tr>
<td>DMA8/9 Interrupt (Memory DMA Stream 0)</td>
<td>IVG13</td>
</tr>
<tr>
<td>DMA10/11 Interrupt (Memory DMA Stream 1)</td>
<td></td>
</tr>
<tr>
<td>Software Watchdog Timer</td>
<td></td>
</tr>
<tr>
<td>Software Interrupt 1</td>
<td>IVG14</td>
</tr>
<tr>
<td>Software Interrupt 2 (lowest priority)</td>
<td>IVG15</td>
</tr>
</tbody>
</table>
Figure 4: SIC_IAR1

Figure 5: SIC_IAR2
Figure 6: TIMER_ENABLE

Timer Enable Register (TIMER_ENABLE)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0xFFFF0 0640

Reset = 0x0000

<table>
<thead>
<tr>
<th>TIMEN0 (Timer0 Enable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - Enable timer</td>
</tr>
<tr>
<td>Read as 1 when enabled</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TIMEN1 (Timer1 Enable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - Enable timer</td>
</tr>
<tr>
<td>Read as 1 when enabled</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TIMEN2 (Timer2 Enable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - Enable timer</td>
</tr>
<tr>
<td>Read as 1 when enabled</td>
</tr>
</tbody>
</table>

Figure 7: TIMERx_CONFIG

Timer Configuration Registers (TIMERx_CONFIG)

<table>
<thead>
<tr>
<th>Timer0:</th>
<th>0xFFFF0 0600</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timer1:</th>
<th>0xFFFF0 0610</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timer2:</th>
<th>0xFFFF0 0620</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ERR_TYP[1:0] (Error Type) - RO</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 - No error</td>
</tr>
<tr>
<td>01 - Counter overflow error</td>
</tr>
<tr>
<td>10 - Period register programming error</td>
</tr>
<tr>
<td>11 - Pulse width register programming error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EMU_RUN (Emulation Behavior Select)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - Timer counter stops during emulation</td>
</tr>
<tr>
<td>1 - Timer counter runs during emulation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TOGGLE_HI (PULSE_HI Toggle Mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - The effective state of PULSE_HI is the programmed state</td>
</tr>
<tr>
<td>1 - The effective state of PULSE_HI alternates each period</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CLK_SEL (Timer Clock Select)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - Use system clock SCLK for counter</td>
</tr>
<tr>
<td>1 - Use PWM_CLK to clock counter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TMODE[1:0] (Timer Mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 - Reset stateUnused</td>
</tr>
<tr>
<td>01 - PWM OUT mode</td>
</tr>
<tr>
<td>10 - WDTH_CAP mode</td>
</tr>
<tr>
<td>11 - EXT_CLK mode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PULSE_HI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - Negative action pulse</td>
</tr>
<tr>
<td>1 - Positive action pulse</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PERIOD_CNT (Period Count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - Count to end of width</td>
</tr>
<tr>
<td>1 - Count to end of period</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IRQ_ENA (Interrupt Request Enable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - Interrupt request disable</td>
</tr>
<tr>
<td>1 - Interrupt request enable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TIN_SEL (Timer Input Select)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - Sample TMRx pin or PFI pin</td>
</tr>
<tr>
<td>1 - Sample UART RX pin or PPI_CLK pin</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUT_DIS (Output Pad Disable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - Enable pad in PWM_OUT mode</td>
</tr>
<tr>
<td>1 - Disable pad in PWM_OUT mode</td>
</tr>
</tbody>
</table>
Appendix B
Musical Information

I used the data from Table 1 to determine the frequencies of the required notes. Note the sampling frequency $f_s$ is 48000kHz.

Digital Sinusoid Signal:

$$x[n] = A\sin\left[2\pi\left(\frac{f_0}{f_s}\right)n\right] = A\sin\left[\pi\left(\frac{f_0}{f_s/2}\right)n\right] = A\sin(\pi F_0 n) = A\sin(\omega_0 n)$$

<table>
<thead>
<tr>
<th>Musical Note</th>
<th>Half steps relative to A4</th>
<th>Frequency $f_0$ (Hz)</th>
<th>$\omega_0 = \pi \left(\frac{f_0}{f_s/2}\right)$</th>
<th>Highest n for $2\pi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3</td>
<td>-12</td>
<td>220</td>
<td>0.028798</td>
<td>218</td>
</tr>
<tr>
<td>A#3</td>
<td>-11</td>
<td>233.082</td>
<td>0.03051</td>
<td>206</td>
</tr>
<tr>
<td>B3</td>
<td>-10</td>
<td>246.942</td>
<td>0.032325</td>
<td>194</td>
</tr>
<tr>
<td>C4</td>
<td>-9</td>
<td>261.626</td>
<td>0.034247</td>
<td>183</td>
</tr>
<tr>
<td>C#4</td>
<td>-8</td>
<td>277.183</td>
<td>0.036283</td>
<td>173</td>
</tr>
<tr>
<td>D4</td>
<td>-7</td>
<td>293.665</td>
<td>0.038441</td>
<td>163</td>
</tr>
<tr>
<td>D#4</td>
<td>-6</td>
<td>311.127</td>
<td>0.040726</td>
<td>154</td>
</tr>
<tr>
<td>E4</td>
<td>-5</td>
<td>329.628</td>
<td>0.043148</td>
<td>146</td>
</tr>
<tr>
<td>F4</td>
<td>-4</td>
<td>349.228</td>
<td>0.045714</td>
<td>137</td>
</tr>
<tr>
<td>F#4</td>
<td>-3</td>
<td>369.994</td>
<td>0.048432</td>
<td>130</td>
</tr>
<tr>
<td>G4</td>
<td>-2</td>
<td>391.995</td>
<td>0.051312</td>
<td>122</td>
</tr>
<tr>
<td>G#4</td>
<td>-1</td>
<td>415.305</td>
<td>0.054363</td>
<td>116</td>
</tr>
<tr>
<td>A4</td>
<td>0</td>
<td>440</td>
<td>0.057596</td>
<td>109</td>
</tr>
<tr>
<td>A#4</td>
<td>1</td>
<td>466.164</td>
<td>0.061021</td>
<td>103</td>
</tr>
<tr>
<td>B4</td>
<td>2</td>
<td>493.883</td>
<td>0.064649</td>
<td>97</td>
</tr>
<tr>
<td>C5</td>
<td>3</td>
<td>523.251</td>
<td>0.068493</td>
<td>92</td>
</tr>
<tr>
<td>C#5</td>
<td>4</td>
<td>554.365</td>
<td>0.072566</td>
<td>87</td>
</tr>
<tr>
<td>D5</td>
<td>5</td>
<td>587.33</td>
<td>0.076881</td>
<td>82</td>
</tr>
<tr>
<td>D#5</td>
<td>6</td>
<td>622.254</td>
<td>0.081453</td>
<td>77</td>
</tr>
<tr>
<td>E5</td>
<td>7</td>
<td>659.255</td>
<td>0.086296</td>
<td>73</td>
</tr>
</tbody>
</table>
Figure 8: “Away in a Manger”
Appendix C
Code

Talkthrough.h

#ifndef __Talkthrough_DEFINED
#define __Talkthrough_DEFINED

// Header files

#include <sys/exception.h>
#include <cdefBF533.h>

// Symbolic constants

// addresses for Port B in Flash A
#define pFlashA_PortA_Dir (volatile unsigned char *)0x20270006
#define pFlashA_PortA_Data (volatile unsigned char *)0x20270004
#define pFlashA_PortB_Dir   (volatile unsigned char *)0x20270007
#define pFlashA_PortB_Data  (volatile unsigned char *)0x20270005

// names for codec registers, used for sCodec1836TxRegs[]
#define DAC_CONTROL_1  0x0000
#define DAC_CONTROL_2  0x1000
#define DAC_VOLUME_0  0x2000
#define DAC_VOLUME_1  0x3000
#define DAC_VOLUME_2  0x4000
#define DAC_VOLUME_3  0x5000
#define DAC_VOLUME_4  0x6000
#define DAC_VOLUME_5  0x7000
#define ADC_0_PEAK_LEVEL 0x8000
#define ADC_1_PEAK_LEVEL 0x9000
#define ADC_2_PEAK_LEVEL 0xA000
#define ADC_3_PEAK_LEVEL 0xB000
#define ADC_CONTROL_1  0xC000
#define ADC_CONTROL_2  0xD000
#define ADC_CONTROL_3  0xE000

// names for slots in ad1836 audio frame
#define INTERNAL_ADC_L0   0
#define INTERNAL_ADC_R0   2
#define INTERNAL_DAC_L0   0
#define INTERNAL_DAC_R0   2
#define INTERNAL_ADC_L1   1
#define INTERNAL_ADC_R1   3
#define INTERNAL_DAC_L1   1
#define INTERNAL_DAC_R1   3
// size of array sCodec1836TxRegs
#define CODEC_1836_REGS_LENGTH 11

// SPI transfer mode
#define TIMOD_DMA_TX 0x0003

// SPORT0 word length
#define SLEN_24 0x0017

// DMA flow mode
#define FLOW_1 0x1000

// Global variables
extern int iChannel0LeftIn;
extern int iChannel0RightIn;
extern int iChannel0LeftOut;
extern int iChannel0RightOut;
extern int iChannel1LeftIn;
extern int iChannel1RightIn;
extern int iChannel1LeftOut;
extern int iChannel1RightOut;
extern volatile short sCodec1836TxRegs[];
extern volatile int iRxBuffer1[];
extern volatile int iTxBuffer1[];
eight; //initialize counter for our switch
extern int sn; //initialize counter for silent night
extern int am; //initialize counter for away in a manger
extern int speedup; //initialize counter for speeding up
extern int slowdown; //initialize counter for slowing down

// Prototypes
void Init_EBIU(void);
void Init_Flash(void);
void Init1836(void);
void Init_Sport0(void);
void Init_DMA(void);
void Init_Interrupts(void);
void Init_Flags(void);
void Init_Timers(void);

void Process_Data(void);

// in file ISRs.c
EX_INTERRUPT_HANDLER(Sport0_RX_ISR);
EX_INTERRUPT_HANDLER(FlagA_ISR);
EX_INTERRUPT_HANDLER(Timer0_ISR);

#endif //__Talkthrough_DEFINED

Main.c

//Name: Talkthrough for the ADSP-BF533 EZ-KIT Lite
//%(C) Copyright 2003 - Analog Devices, Inc. All rights reserved.
// Project Name: BF533 C Talkthrough I2S
// Date Modified: 04/03/03 HD Rev 1.0
// Software: VisualDSP++3.1
// Hardware: ADSP-BF533 EZ-KIT Board
// Connections: Connect RSCLK0 to TSCLK0 together (Turn SW9 pin 6 on)
// Connect RFS0 to TFS0 together (Turn SW9 pin 5 ON)
// Connect an input source (such as a radio) to the Audio
// input jack and an output source (such as headphones) to
// the Audio output jack
// Purpose: This program sets up the SPI port on the ADSP-BF533 to
// configure the AD1836 codec. The SPI port is disabled
// after initialization. The data to/from the codec are
// transferred over SPORT0 in I2S mode

#include "Talkthrough.h"
#include "sysreg.h"
#include "ccblkfn.h"

Variables

// Description: The variables iChannelxLeftIn and iChannelxRightIn contain
// the data coming from the codec AD1836. The (processed)
// playback data are written into the variables
// iChannelxLeftOut and iChannelxRightOut respectively, which
// are then sent back to the codec in the SPORT0 ISR.
// The values in the array iCodec1836TxRegs can be modified to
// set up the codec in different configurations according to
// the AD1885 data sheet.

int iChannel0LeftIn, iChannel1LeftIn;
int iChannel0RightIn, iChannel1RightIn;
int iChannel0LeftOut, iChannel1LeftOut;
int iChannel0RightOut, iChannel1RightOut;
int iChannel0RightOut, iChannel1RightOut;
// array for registers to configure the ad1836
// names are defined in "Talkthrough.h"
volatile short sCodec1836TxRegs[CODEC_1836_REGS_LENGTH] = {
    DAC_CONTROL_1 | 0x000,
    DAC_CONTROL_2 | 0x000,
    DAC_VOLUME_0  | 0x3ff,
    DAC_VOLUME_1  | 0x3ff,
    DAC_VOLUME_2  | 0x3ff,
    DAC_VOLUME_3  | 0x3ff,
    DAC_VOLUME_4  | 0x000,
    DAC_VOLUME_5  | 0x000,
    ADC_CONTROL_1 | 0x000,
    ADC_CONTROL_2 | 0x000,
    ADC_CONTROL_3 | 0x000
};
// SPORT0 DMA transmit buffer
volatile int iTxBuffer1[4];
// SPORT0 DMA receive buffer
volatile int iRxBuffer1[4];

char eight=0x00;//set counter for PF8
int sn=0;
int am=0;
int speedup=0;
int slowdown=0;
//---------------------------------------------------------------//
// Function: main
// Description: After calling a few initialization routines, main() just
//   waits in a loop forever. The code to process the incoming
//   data can be placed in the function Process_Data() in the
//   file "Process_Data.c".
//---------------------------------------------------------------//
void main(void)
{
    sysreg_write(reg_SYSCFG, 0x32);      //Initialize System Configuration
    Init_EBIU();
    Init_Flash();
    Init1836();
    Init_Sport0();
    Init_DMA();
    Init_Interrupts();
    Enable_DMA_Sport0();
    Init_Flags();
    Init_Timers();
}
while(1);

Initialize.c

#include "Talkthrough.h"

// Function:  Init_EBIU
// Description: This function initializes and enables asynchronous memory banks in External Bus Interface Unit so that Flash A can be accessed.

void Init_EBIU(void)
{
    *pEBIU_AMBCTL0 = 0x7bb07bb0;
    *pEBIU_AMBCTL1 = 0x7bb07bb0;
    *pEBIU_AMGCTL = 0x000f;
}

// Function:  Init_Flash
// Description: This function initializes pin direction of Port A in Flash A to output. The AD1836_RESET on the ADSP-BF533 EZ-KIT board is connected to Port A.

void Init_Flash(void)
{
    *pFlashA_PortA_Dir = 0x1;
    *pFlashA_PortB_Dir = 0x3f;
    
    *pFlashA_PortB_Data = 0x00;//all LEDs are off
}

// Function:  Init1836()
// Description: This function sets up the SPI port to configure the AD1836. The content of the array sCodec1836TxRegs is sent to the codec.

void Init1836(void)
{
    int i;
    int j;
    static unsigned char ucActive_LED = 0x01;

    // write to Port A to reset AD1836
    *pFlashA_PortA_Data = 0x00;

    // write to Port A to enable AD1836
    *pFlashA_PortA_Data = ucActive_LED;
// wait to recover from reset
for (i=0; i<0xf0000; i++) asm("nop;");

// Enable PF4
*pSPI_FLG = FLS4;
// Set baud rate SCK = HCLK/(2*SPIBAUD) SCK = 2MHz
*pSPI_BAUD = 16;
// configure spi port
// SPI DMA write, 16-bit data, MSB first, SPI Master
*pSPI_CTL = TIMOD_DMA_TX | SIZE | MSTR;

// Set up DMA5 to transmit
// Map DMA5 to SPI
*pDMA5_PERIPHERAL_MAP = 0x5000;

// Configure DMA5
// 16-bit transfers
*pDMA5_CONFIG = WDSIZE_16;
// Start address of data buffer
*pDMA5_START_ADDR = sCodec1836TxRegs;
// DMA inner loop count
*pDMA5_X_COUNT = CODEC_1836_REGS_LENGTH;
// Inner loop address increment
*pDMA5_X_MODIFY = 2;

// enable DMAs
*pDMA5_CONFIG = (*pDMA5_CONFIG | DMAEN);
// enable spi
*pSPI_CTL = (*pSPI_CTL | SPE);

// wait until dma transfers for spi are finished
for (j=0; j<0xaff0; j++) asm("nop;");

// disable spi
*pSPI_CTL = 0x0000;
}

// Function: Init_Sport0
// Description: Configure Sport0 for I2S mode, to transmit/receive data
to/from the AD1836. Configure Sport for external clocks and frame syncs.
void Init_Sport0(void)
{
    // Sport0 receive configuration
    // External CLK, External Frame sync, MSB first, Active Low
    // 24-bit data, Stereo frame sync enable
    *pSPORT0_RCR1 = RFSR | LRFS | RCKFE;
*pSPORT0_RCR2 = SLEN_24 | RXSE | RSFSE;

// Sport0 transmit configuration
// External CLK, External Frame sync, MSB first, Active Low
// 24-bit data, Secondary side enable, Stereo frame sync enable
*pSPORT0_TCR1 = TFSR | LTFS | TCKFE;
*pSPORT0_TCR2 = SLEN_24 | TXSE | TSFSE;
}

//--------------------------------------------------------------------------//
// Function: Init_DMA
// Description: Initialize DMA1 in autobuffer mode to receive and DMA2 in
// autobuffer mode to transmit
//--------------------------------------------------------------------------//
void Init_DMA(void)
{
    // Set up DMA1 to receive
    // Map DMA1 to Sport0 RX
    *pDMA1_PERIPHERAL_MAP = 0x1000;
    // Configure DMA1
    // 32-bit transfers, Interrupt on completion, Autobuffer mode
    *pDMA1_CONFIG = WNR | WDSIZE_32 | DI_EN | FLOW_1;
    // Start address of data buffer
    *pDMA1_START_ADDR = iRxBuffer1;
    // DMA inner loop count
    *pDMA1_X_COUNT = 4;
    // Inner loop address increment
    *pDMA1_X_MODIFY = 4;

    // Set up DMA2 to transmit
    // Map DMA2 to Sport0 TX
    *pDMA2_PERIPHERAL_MAP = 0x2000;
    // Configure DMA2
    // 32-bit transfers, Autobuffer mode
    *pDMA2_CONFIG = WDSIZE_32 | FLOW_1;
    // Start address of data buffer
    *pDMA2_START_ADDR = iTxBuffer1;
    // DMA inner loop count
    *pDMA2_X_COUNT = 4;
    // Inner loop address increment
    *pDMA2_X_MODIFY = 4;
}

//--------------------------------------------------------------------------//
// Function: Enable_DMA_Sport
// Description: Enable DMA1, DMA2, Sport0 TX and Sport0 RX
//--------------------------------------------------------------------------//
void Enable_DMA_Sport0(void)
{  
    // enable DMAs
    *pDMA2_CONFIG = (*pDMA2_CONFIG | DMAEN);
    *pDMA1_CONFIG = (*pDMA1_CONFIG | DMAEN);

    // enable Sport0 TX and RX
    *pSPORT0_TCR1  = (*pSPORT0_TCR1 | TSPEN);
    *pSPORT0_RCR1  = (*pSPORT0_RCR1 | RSPEN);
}

// Function:    Init_Interrupts
// Description: Initialize Interrupts for Sport0 RX, TIMER0, and FLAGA
//---------------------------------------------------------------

void Init_Interrupts(void)
{
    // Set System Interrupt Assignment Registers  
    // 2 for Sport0 RX, 5 for Flag A, and 4 for Timer0
    *pSIC_IAR0 = 0xffffffff;
    *pSIC_IAR1 = 0xffffff2f;
    *pSIC_IAR2 = 0xffff5ff4;

    // assign ISRs to interrupt vectors
    register_handler(ik_i vg9, Sport0_RX_ISR);
    register_handler(ik_i vg12, FlagA_ISR);
    register_handler(ik_i vg11, Timer0_ISR);
    // enable interrupts Timer0, Sport0 RX, and Flag A
    *pSIC_IMASK = 0x00090200;
}

// Function: Init_Flags
// Description: Initialize switch PF8.
//---------------------------------------------------------------

void Init_Flags(void)
{
    *pFIO_INEN                = 0x0700;  //need PF8, PF9, and PF10
    *pFIO_DIR                = 0x0000;
    *pFIO_EDGE                = 0x0700;
    *pFIO_MASKA_D             = 0x0700;
}

// Function:   Init_Timers
// Parameters:  None
// Return:  None
// Description: This function initializes Timer0 for
// PWM mode. It is used as reference for
// the "shift-clock."
//---------------------------------------------------------------

void Init_Timers(void)
{
}
ISR.c

#include "Talkthrough.h"

// Function: Sport0_RX_ISR
// Description: This ISR is executed after a complete frame of input data has been received. The new samples are stored in iChannel0LeftIn, iChannel0RightIn, iChannel1LeftIn and iChannel1RightIn respectively. Then the function Process_Data() is called in which user code can be executed. After that the processed values are copied from the variables iChannel0LeftOut, iChannel0RightOut, iChannel1LeftOut and iChannel1RightOut into the dma transmit buffer.

EX_INTERRUPT_HANDLER(Sport0_RX_ISR)
{
    // confirm interrupt handling
    *pDMA1_IRQ_STATUS = 0x0001;

    // copy input data from dma input buffer into variables
    iChannel0LeftIn = iRxBuffer1[INTERNAL_ADC_L0];
    iChannel0RightIn = iRxBuffer1[INTERNAL_ADC_R0];
    iChannel1LeftIn = iRxBuffer1[INTERNAL_ADC_L1];
    iChannel1RightIn = iRxBuffer1[INTERNAL_ADC_R1];

    // call function that contains user code
    Process_Data();

    // copy processed data from variables into dma output buffer
    iTxBuffer1[INTERNAL_DAC_L0] = iChannel0LeftOut;
    iTxBuffer1[INTERNAL_DAC_R0] = iChannel0RightOut;
    iTxBuffer1[INTERNAL_DAC_L1] = iChannel1LeftOut;
    iTxBuffer1[INTERNAL_DAC_R1] = iChannel1RightOut;
}

EX_INTERRUPT_HANDLER(FlagA_ISR)
short a=0x0700;

//check interrupt register
//checking eight
if(*pFIO_FLAG_C == 0x0100)//switch songs
{
    if(eight==0)//switch to "Away in a Manger"
    {
        *pFlashA_PortB_Data = 0x01; //turn on LED4
        eight=eight+1;
        am=0; //starts song at the beginning
        *pTIMER0_PERIOD=0x03000000;
    }
    else if (eight==1)//switch to "Silent Night"
    {
        //turn on LED5
        *pFlashA_PortB_Data = 0x00; //turn off LED4
        eight = 0;
        sn=0; //starts song at the beginning
        *pTIMER0_PERIOD=0x02000000;
    }
}
else if(*pFIO_FLAG_C == 0x0200)//speed up
{
    if(*pTIMER0_PERIOD>0x00f00000)//bottom limit
    {
        speedup=speedup+0x00100000;
    }
}
else if(*pFIO_FLAG_C == 0x0400)//slow down
{
    if(*pTIMER0_PERIOD<0xff000000)//upper limit
    {
        slowdown=slowdown+0x00100000;
    }
}
*pFIO_FLAG_C=a; //clear

EX_INTERRUPT_HANDLER(Timer0_ISR)
{
    //confirm interrupt handling
    *pTIMER_STATUS = 0x0001;
    if(eight==0)//timing for "Silent Night"
    {

if(sn==1|sn==5|sn==8|sn==11|sn==14|sn==17|sn==20|sn==23|sn==26|sn==
=29|sn==32|sn==35|sn==38|sn==39|sn==40|sn==43)
{
    sn++;  
    *pTIMER0_PERIOD=0x02000000 +slowdown -speedup;
}
else if(sn==3|sn==15|sn==18|sn==24|sn==27|sn==33|sn==41)
{
    sn++;  
    *pTIMER0_PERIOD=0x03000000+slowdown -speedup;
}
else if(sn==4|sn==16|sn==19|sn==25|sn==28|sn==34|sn==42)
{
    sn++;  
    *pTIMER0_PERIOD=0x01000000+slowdown -speedup;
}
else if(sn==2|sn==6|sn==12|sn==21|sn==30|sn==36|sn==37)
{
    sn++;  
    *pTIMER0_PERIOD=0x04000000+slowdown -speedup;
}
else if(sn==7|sn==9|sn==10|sn==13|sn==22|sn==31)
{
    sn++;  
    *pTIMER0_PERIOD=0x03700000+slowdown -speedup;
}
else if(sn==44)
{
    sn++;  
    *pTIMER0_PERIOD=0x05000000+slowdown -speedup;
}
else if(sn==45)
{
    sn++;  
    *pTIMER0_PERIOD=0x03000000+slowdown -speedup;
}
else if(sn==46)
{
   sn=0;
    *pTIMER0_PERIOD=0x03000000+slowdown -speedup;
}
else if(sn==0)
{
    sn++;
*pTIMER0_PERIOD=0x01000000+slowdown -speedup;
}
}
}
if(eight==1)//timing for "Away in a Manger"
{
    if(am==2|am==3|am==4|am==5|am==6|am==7|am==8|am==10|am==13|am==14|am==15|am==16|am==17|am==18|am==19|am==21|am==24|am==25|am==26|am==27|am==28|am==29|am==30|am==32|am==35|am==36|am==37|am==38|am==39|am==40|am==41)
        am++;
        *pTIMER0_PERIOD=0x02000000+slowdown -speedup;
    }
    else if(am==0|am==11|am==22|am==33)
        am++;
        *pTIMER0_PERIOD=0x03000000+slowdown -speedup;
    }
    else if(am==1|am==12|am==23|am==34)
        am++;
        *pTIMER0_PERIOD=0x01000000+slowdown -speedup;
    }
    else if(am==9|am==20|am==31|am==42)
        am++;
        *pTIMER0_PERIOD=0x04000000+slowdown -speedup;
    }
    else if(am==43)
        am++;
        *pTIMER0_PERIOD=0x02000000+slowdown -speedup;
    }
    else if(am==44)
        am=0;
        *pTIMER0_PERIOD=0x03000000+slowdown -speedup;
    }
}
#include "Talkthrough.h"

#include "math.h"

// Function: Process_Data()
// Description: This function is called from inside the SPORT0 ISR every
// time a complete audio frame has been received. The new
// input samples can be found in the variables iChannel0LeftIn,
// iChannel0RightIn, iChannel1LeftIn and iChannel1RightIn
// respectively. The processed data should be stored in
// iChannel0LeftOut, iChannel0RightOut, iChannel1LeftOut,
// iChannel1RightOut, iChannel2LeftOut and iChannel2RightOut
// respectively.
Tabla Start
Tabla End
int r=0;
float x = (1<<20);
float y; //initialize output
int y0; //integer form of output

void Process_Data(void)
{
    if(eight==0)//"Silent Night"
    {
        if(sn==1)//Si (F4)
        {
            *pFlashA_PortB_Data = 0x02;
            r %= 137;

            y=7*x*sinf(0.045714 *r);
            y0=(int)(y);
            iChannel0RightOut=y0;
        }
        else if(sn==2)//(G4)
        {
            *pFlashA_PortB_Data = 0x04;
            r %= 122;

            y=5*x*sinf(0.051312*r);
            y0=(int)(y);
            iChannel0RightOut=y0;
        }
        else if(sn==3)//..lent (F4)
        {
            *pFlashA_PortB_Data = 0x08;
            r %=137;

            ...
y = 5 * x * sinf(0.045714 * r);
y0 = (int)(y);
iChannel0RightOut = y0;
}
else if(sn == 4) // Night (D4)
{
    *pFlashA_PortB_Data = 0x10;
    r %= 163;

    y = 5 * x * sinf(0.038441 * r);
    y0 = (int)(y);
    iChannel0RightOut = y0;
}
else if(sn == 5) // Ho.. (F4)
{
    *pFlashA_PortB_Data = 0x20;
    r %= 137;

    y = 7 * x * sinf(0.045714 * r);
    y0 = (int)(y);
    iChannel0RightOut = y0;
}
else if(sn == 6) // (G4)
{
    *pFlashA_PortB_Data = 0x02;
    r %= 122;

    y = 7 * x * sinf(0.051312 * r);
    y0 = (int)(y);
    iChannel0RightOut = y0;
}
else if(sn == 7) // ..ly (F4)
{
    *pFlashA_PortB_Data = 0x04;
    r %= 137;

    y = 7 * x * sinf(0.045714 * r);
    y0 = (int)(y);
    iChannel0RightOut = y0;
}
else if(sn == 8) // Night (D4)
{
    *pFlashA_PortB_Data = 0x08;
    r %= 163;

    y = 7 * x * sinf(0.038441 * r);
    y0 = (int)(y);
    iChannel0RightOut = y0;
}
else if(sn==9) // All (C5)
{
    *pFlashA_PortB_Data = 0x10;
    r %= 92;

    y = 7*x*sinf(0.068493 *r);
    y0 = (int)(y);
    iChannel0RightOut = y0;
}
else if(sn==10) // is (C5)
{
    *pFlashA_PortB_Data = 0x20;
    r %= 92;

    y = 7*x*sinf(0.068493 *r);
    y0 = (int)(y);
    iChannel0RightOut = y0;
}
else if(sn==11) // calm (A4)
{
    *pFlashA_PortB_Data = 0x02;
    r %= 109;

    y = 7*x*sinf(0.057596 *r);
    y0 = (int)(y);
    iChannel0RightOut = y0;
}
else if(sn==12) // All (B4 FLAT)
{
    *pFlashA_PortB_Data = 0x04;
    r %= 103;

    y = 7*x*sinf(0.061021 *r);
    y0 = (int)(y);
    iChannel0RightOut = y0;
}
else if(sn==13) // is (B4 FLAT)
{
    *pFlashA_PortB_Data = 0x08;
    r %= 103;

    y = 7*x*sinf(0.061021 *r);
    y0 = (int)(y);
    iChannel0RightOut = y0;
}
else if(sn==14) // bright (F4)
{
    *pFlashA_PortB_Data = 0x10;
r %= 137;

y=7*x*sinf(0.045714 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==15)//Round (G4)
{
    *pFlashA_PortB_Data = 0x20;
    r %= 122;

    y=7*x*sinf(0.051312 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(sn==16)//yon (G4)
{
    *pFlashA_PortB_Data = 0x02;
    r %= 122;

    y=7*x*sinf(0.051312 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(sn==17)//vir.. (B4 FLAT)
{
    *pFlashA_PortB_Data = 0x04;
    r %= 103;

    y=7*x*sinf(0.061021 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(sn==18)//(A4)
{
    *pFlashA_PortB_Data = 0x08;
    r %= 109;

    y=7*x*sinf(0.057596 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(sn==19)//...gen (G4)
{
    *pFlashA_PortB_Data = 0x10;
    r %= 122;

    y=7*x*sinf(0.051312 *r);
    y0=(int)(y);
iChannel0RightOut=y0;

} else if(sn==20)//mo.. (F4)
{
    *pFlashA_PortB_Data = 0x20;
    r %= 137;

    y=7*x*sinf(0.045714 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(sn==21)//..ther (G4)
{
    *pFlashA_PortB_Data = 0x02;
    r %= 122;

    y=7*x*sinf(0.051312 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(sn==22)//and (F4)
{
    *pFlashA_PortB_Data = 0x04;
    r %= 137;

    y=7*x*sinf(0.045714 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(sn==23)//child (D4)
{
    *pFlashA_PortB_Data = 0x08;
    r %= 163;

    y=7*x*sinf(0.038441 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(sn==24)//Ho.. (G4)
{
    *pFlashA_PortB_Data = 0x10;
    r %= 122;

    y=7*x*sinf(0.051312 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(sn==25)//..ly (G4)
{

}
*pFlashA_PortB_Data = 0x20;
r %= 122;

y=7*x*sinf(0.051312 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==26)//in.. (B4 FLAT)
{
    *pFlashA_PortB_Data = 0x02;
r %= 103;

    y=7*x*sinf(0.061021 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==27)//...fant (A4)
{
    *pFlashA_PortB_Data = 0x04;
r %= 109;

    y=7*x*sinf(0.057596 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==28)//so (G4)
{
    *pFlashA_PortB_Data = 0x08;
r %= 122;

    y=7*x*sinf(0.051312 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==29)//ten.. (F4)
{
    *pFlashA_PortB_Data = 0x10;
r %= 137;

    y=7*x*sinf(0.045714 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==30)//...der (G4)
{
    *pFlashA_PortB_Data = 0x20;
r %= 122;

    y=7*x*sinf(0.051312 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==31)//and (F4)
{
    *pFlashA_PortB_Data = 0x02;
    r %= 137;

    y=7*x*sinf(0.045714 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==32)//mild (D4)
{
    *pFlashA_PortB_Data = 0x04;
    r %= 163;

    y=7*x*sinf(0.038441 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==33)//sleep (C5)
{
    *pFlashA_PortB_Data = 0x08;
    r %= 92;

    y=7*x*sinf(0.068493 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==34)//in (C5)
{
    *pFlashA_PortB_Data = 0x10;
    r %= 92;

    y=7*x*sinf(0.068493 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==35)//heav... (E5 FLAT)
{
    *pFlashA_PortB_Data = 0x20;
    r %= 77;

    y=7*x*sinf(0.081453 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==36)//..en..(C5)
{  
  *pFlashA_PortB_Data = 0x02;
  r %= 92;

  y=7*x*sinf(0.068493 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==37)//...ly (A4)
{
  *pFlashA_PortB_Data = 0x04;
  r %= 109;

  y=7*x*sinf(0.057596 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==38)//pea... (B4 FLAT)
{
  *pFlashA_PortB_Data = 0x08;
  r %= 103;

  y=7*x*sinf(0.061021 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==39)//...ce (D5)
{
  *pFlashA_PortB_Data = 0x10;
  r %= 82;

  y=7*x*sinf(0.076881 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==40)//sle.. (B4 FLAT)
{
  *pFlashA_PortB_Data = 0x20;
  r %= 103;

  y=7*x*sinf(0.061021 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==41)//..ep (F4)
{
  *pFlashA_PortB_Data = 0x02;
  r %= 137;
}

y=7*x*sinf(0.045714 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==42)//in (D4)
{
    *pFlashA_PortB_Data = 0x04;
    r %= 163;

    y=7*x*sinf(0.038441 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==43)//heav... (F4)
{
    *pFlashA_PortB_Data = 0x08;
    r %= 137;

    y=7*x*sinf(0.045714 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==44)//..en.. (E4 FLAT)
{
    *pFlashA_PortB_Data = 0x10;
    r %= 154;

    y=7*x*sinf(0.040726 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==45)//..ly (C4)
{
    *pFlashA_PortB_Data = 0x20;
    r %= 183;

    y=7*x*sinf(0.034247 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(sn==46)//peace (B3 FLAT)
{
    *pFlashA_PortB_Data = 0x02;
    r %= 206;

    y=7*x*sinf(0.03051 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(eight==1)//"Away in a Manger"
{
    if(am==1)//A.. (C5)
    {
        *pFlashA_PortB_Data = 0x02|0x01;
        r %= 92;
        
        y=7*x*sinf(0.068493 *r);
        y0=(int)(y);
        iChannel0RightOut=y0;
    }
}
else if(am==2)//..way (C5)
{
    *pFlashA_PortB_Data = 0x04|0x01;
    r %= 92;
    
    y=7*x*sinf(0.068493 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==3)//in (B4 FLAT)
{
    *pFlashA_PortB_Data = 0x08|0x01;
    r %= 103;
    
    y=7*x*sinf(0.061021 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==4)//a (A4)
{
    *pFlashA_PortB_Data = 0x10|0x01;
    r %= 109;
    
    y=7*x*sinf(0.057596 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==5)//man.. (A4)
{
    *pFlashA_PortB_Data = 0x20|0x01;
    r %= 109;
    
    y=7*x*sinf(0.057596 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==6)//..ger (G4)
{ *
  pFlashA_PortB_Data = 0x02|0x01;
  r %= 122;

  y=7*x*sinf(0.051312 *r);
  y0=(int)(y);
  iChannel0RightOut=y0;
}
else if(am==7)//no (F4)
{
  *
  pFlashA_PortB_Data = 0x04|0x01;
  r %= 137;

  y=7*x*sinf(0.045714 *r);
  y0=(int)(y);
  iChannel0RightOut=y0;
}
else if(am==8)//crib (F4)
{
  *
  pFlashA_PortB_Data = 0x08|0x01;
  r %= 137;

  y=7*x*sinf(0.045714 *r);
  y0=(int)(y);
  iChannel0RightOut=y0;
}
else if(am==9)//for (E4)
{
  *
  pFlashA_PortB_Data = 0x10|0x01;
  r %= 146;

  y=7*x*sinf(0.043148 *r);
  y0=(int)(y);
  iChannel0RightOut=y0;
}
else if(am==10)//a (D4)
{
  *
  pFlashA_PortB_Data = 0x20|0x01;
  r %= 163;

  y=7*x*sinf(0.038441 *r);
  y0=(int)(y);
  iChannel0RightOut=y0;
}
else if(am==11)//bed (C4)
{
  *
  pFlashA_PortB_Data = 0x02|0x01;
  r %= 183;
}
y=7*x*sinf(0.034247 *r);
y0=(int)(y);
iChannel0RightOut=y0;
}
else if(am==12)//the (C4)
{
    *pFlashA_PortB_Data = 0x04|0x01;
    r %= 183;
    y=7*x*sinf(0.034247 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==13)//lit.. (C4)
{
    *pFlashA_PortB_Data = 0x08|0x01;
    r %= 183;
    y=7*x*sinf(0.034247 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==14)//..tle (D4)
{
    *pFlashA_PortB_Data = 0x10|0x01;
    r %= 163;
    y=7*x*sinf(0.038441 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==15)//Lord (C4)
{
    *pFlashA_PortB_Data = 0x20|0x01;
    r %= 183;
    y=7*x*sinf(0.034247 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==16)//Je.. (C4)
{
    *pFlashA_PortB_Data = 0x02|0x01;
    r %= 183;
    y=7*x*sinf(0.034247 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==17)//sus (G4)
{
    *pFlashA_PortB_Data = 0x04|0x01;
    r %= 122;

    y=7*x*sinf(0.051312 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==18)//laid (E4)
{
    *pFlashA_PortB_Data = 0x08|0x01;
    r %= 146;

    y=7*x*sinf(0.043148 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==19)//down (D4)
{
    *pFlashA_PortB_Data = 0x10|0x01;
    r %= 163;

    y=7*x*sinf(0.038441 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==20)//his (C4)
{
    *pFlashA_PortB_Data = 0x20|0x01;
    r %= 183;

    y=7*x*sinf(0.034247 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==21)//sweet (F4)
{
    *pFlashA_PortB_Data = 0x02|0x01;
    r %= 137;

    y=7*x*sinf(0.045714 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==22)//head (A4)
{
    *pFlashA_PortB_Data = 0x04|0x01;
    r %= 109;
y=7*x*sinf(0.057596 *r);
y0=(int)(y);
iChannel0RightOut=y0;

} else if(am==23)//the (C5)
{
    *pFlashA_PortB_Data = 0x08|0x01;
    r %= 92;

    y=7*x*sinf(0.068493 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==24)//stars (C5)
{
    *pFlashA_PortB_Data = 0x08|0x01;
    r %= 92;

    y=7*x*sinf(0.068493 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==25)//in (B4 FLAT)
{
    *pFlashA_PortB_Data = 0x20|0x01;
    r %= 103;

    y=7*x*sinf(0.061021 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==26)//the (A4)
{
    *pFlashA_PortB_Data = 0x02|0x01;
    r %= 109;

    y=7*x*sinf(0.057596 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==27)//sky (A4)
{
    *pFlashA_PortB_Data = 0x04|0x01;
    r %= 109;

    y=7*x*sinf(0.057596 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
else if(am==28)//(G4)
{
    *pFlashA_PortB_Data = 0x08|0x01;
    r %= 122;

    y=7*x*sinf(0.051312 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==29)//looked (F4)
{
    *pFlashA_PortB_Data = 0x10|0x01;
    r %= 137;

    y=7*x*sinf(0.045714 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==30)//down (F4)
{
    *pFlashA_PortB_Data = 0x20|0x01;
    r %= 137;

    y=7*x*sinf(0.045714 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==31)//where (E4)
{
    *pFlashA_PortB_Data = 0x02|0x01;
    r %= 146;

    y=7*x*sinf(0.043148 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==32)//he (D4)
{
    *pFlashA_PortB_Data = 0x04|0x01;
    r %= 163;

    y=7*x*sinf(0.038441 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==33)//lay (C4)
{
    *pFlashA_PortB_Data = 0x08|0x01;
    r %= 149;

    y=7*x*sinf(0.043312 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
r %= 183;

y = 7 * x * sinf(0.034247 * r);
y0 = (int)(y);
iChannel0RightOut = y0;
}
else if (am == 34) // the (C4)
{
    *pFlashA_PortB_Data = 0x10|0x01;
    r %= 183;

    y = 7 * x * sinf(0.034247 * r);
y0 = (int)(y);
iChannel0RightOut = y0;
}
else if (am == 35) // lit.. (B4 FLAT)
{
    *pFlashA_PortB_Data = 0x20|0x01;
    r %= 103;

    y = 7 * x * sinf(0.061021 * r);
y0 = (int)(y);
iChannel0RightOut = y0;
}
else if (am == 36) // tle.. (A4)
{
    *pFlashA_PortB_Data = 0x02|0x01;
    r %= 109;

    y = 7 * x * sinf(0.057596 * r);
y0 = (int)(y);
iChannel0RightOut = y0;
}
else if (am == 37) // Lord (G4)
{
    *pFlashA_PortB_Data = 0x04|0x01;
    r %= 122;

    y = 7 * x * sinf(0.051312 * r);
y0 = (int)(y);
iChannel0RightOut = y0;
}
else if (am == 38) // Je.. (A4)
{
    *pFlashA_PortB_Data = 0x08|0x01;
    r %= 109;

    y = 7 * x * sinf(0.057596 * r);
y0 = (int)(y);
iChannel0RightOut=y0;
}
else if(am==39)//..sus (G4)
{
    *pFlashA_PortB_Data = 0x10|0x01;
    r %= 122;
    
y=7*x*sinf(0.051312 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==40)//as.. (F4)
{
    *pFlashA_PortB_Data = 0x20|0x01;
    r %= 137;
    
y=7*x*sinf(0.045714 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==41)//..leep (G4)
{
    *pFlashA_PortB_Data = 0x02|0x01;
    r %= 122;
    
y=7*x*sinf(0.051312 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==42)//on (D4)
{
    *pFlashA_PortB_Data = 0x04|0x01;
    r %= 163;
    
y=7*x*sinf(0.038441 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==43)//the (E4)
{
    *pFlashA_PortB_Data = 0x08|0x01;
    r %= 146;
    
y=7*x*sinf(0.043148 *r);
    y0=(int)(y);
    iChannel0RightOut=y0;
}
else if(am==44)//hay (F4)
{
*pFlashA_PortB_Data = 0x10|0x01;
r %= 137;

y=7*x*sinf(0.045714 *r);
y0=(int)(y);
iChannel0RightOut=y0;

}
EzDJ

An easy DJ Mixing Solution

By Todd Alexander
General Overview

- The EzDJ program allows common DJ functions to be accomplished using the EzKit Lite
- This provides for a cheap alternative to very expensive DJ hardware
Features

- Input level monitors
- Sound effects available for mixing
- Two versions of cross-fading with beat matching and fade status
Problems

- As indicated in the first slide, Disk Jockeys usually work with two input sources, allowing them to switch between songs.
- Normally while a current input source is being played, a DJ would queue another song on the other input channel.
- Even if a song is queued, it is very difficult to tell when to start playing it so that the beats are in sync.
- Why should the beats be in sync? Whenever playing two input sources at the same time with beats that are not synchronized, the result is considered noise by most listeners. [sample]
- In order to solve this problem, beat matching is used.
Beat Matching Algorithms

- There are a few approaches to matching beats. Mostly differing in the way the beats are detected.
- One of the methods I looked at was identifying the beats by their IDs according to a paper at Microsoft research.
- In short they determined the peak then stored an ID of it.
EzDJ’s Algorithm

- First I thought that: 1. A DJ would never try to mix two songs with a largely varied beat count. In fact it is desirable to have songs of similar beat counts; hence, assume the beat counts are similar.

- Next the input peak levels can be setup to fall within a certain range. The input will be amplified to determine the output thus not adversely affecting the output.

- Having control of the input we can then detect the beats, match the beats then perform the cross-fade after successfully matching the beat. Further explanation in subsequent code.

- Bearing in mind that the music is of good quality and does not contain noise at the peaks the average value at the peak was ignored to improve the algorithm’s performance.
Code Details

- Presented in source code documents
Graph of Type 1 Fade
Graph of Type 2 Fade
Improvements

- Monitor feature
- Tempo changing to match beat
- Dynamic level adjustment
Credits

• Cover image – www.ftw-design.com
• Code adapted from Analog Devices EzKit Lite examples
• Dr. Kepuska’s Lectures and Slides
• *Beat This, A Beat Synchronization Project*  
  http://www.owlnet.rice.edu/~elec301/Projects01/beat_sync/beatalgo.html
• ECE3551 is the best 😊!
Microcomputer Systems

Final Project

“Speaker and Sound Modulation”
The aim of the project was to create a system that allows a user to have useful controls over any audio input signal.

A wide variety of options for modifying the audio output while keeping complexities to a minimum.
Basic Layout

System Features

- 1 Stereo Input Channel

- 2 Stereo Output Channels
  - 4 Speakers
  - Left / Right
  - Front / Back
System Options

- Original Input/Output
- Modified Input
  1. Volume Control
  2. Balance Control
  3. Fader Control
  4. Digital Delay
  5. Multitap Delay
  6. Circling Sound

The button PF8 switches between the Original I/O and Modified Output.

The modified output modes are then switched using the button PF9.
Feedback systems

In a system such as this where there are multiple modes of operation it can be confusing at times to know where the system is at any given point.

Feedback created using the available LEDs in the system.

Only 6 LEDs are available so options were limited in differentiating modes.
This mode allows the user to switch back at anytime to the original input signal that was passed into the board.

Comparison at anytime during execution.

For Feedback the LEDS 4 and 5 are Turned OFF

Toggle Mode On / Off
Modified Inputs
Volume Control

Control over Output’s Gain or “Volume”

7 Steps are Available From Zero to Maximum Gain

Done by multiplying a gain value to Input Signal
Feedback Volume

Volume Mode – First Mode

For Feedback the LEDS 4 and 5 reflect a “1” in binary

Volume Level

LEDS 6 – 9 signifies relative Volume Gain

Increasing volume, turns on LEDS 6 through 9
Decreasing volume turns off LEDS 9 through 6
Balance Control

Balance adjusts Left and Right Gains

This is done by reducing the gain on one side while increasing the gain to the other.

Select Balance Mode

Increase Left Gain

Increase Right Gain
Feedback Balance

- The number “2” Signifies – Balance mode

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

- Gain Full Left - LED6 on
- Gain Half Way left - LED7 ON
- Gain Full Right - LED9 on
- Gain Half Way right - LED8 On
Fader Control

Controls the front and back speaker Gains

Gain of the front speakers can be increased while decreasing the gain of the back speakers and vice versa.

Select Fader Mode

Increase Front Gain

Increase Rear Gain
Feedback Fader

The number “3” Signifies – Fader mode

LEDs

1 1
4 5 6 7 8 9

Gain Full Front LED6 on
Gain Half Way Front LED7 ON
Gain Full Rear LED9 on
Gain Half Way Rear LED8 ON
In order to fully replicate a modern audio system, these three factors had to be linked in some way.

<table>
<thead>
<tr>
<th>Gain Type</th>
<th>Left Gain</th>
<th>Right Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front Gain</td>
<td>L0</td>
<td>R0</td>
</tr>
<tr>
<td>Rear Gain</td>
<td>L1</td>
<td>R1</td>
</tr>
</tbody>
</table>

By multiplying their individual gains and calculating an equivalent value, the gains for each speaker could be easily transferred.
Digital Delay

Adds a delayed version of the input to the present input.

By storing values and then replaying them after a delay time such as 45 ms, the effect is produced.

The system gives the user the option of increasing the delay time between 45 ms to 180 ms.
Feedback Digital Delay

At this point, to distinguish more modes would ultimately decrease the amount of LEDs available. This was solved by placing the binary “count” on the right LEDs.

The number “1” Signifies – Digital Delay mode

45ms Delay
LED6 on

90ms Delay
LED7 ON

180ms Delay
LED9 on

135ms Delay
LED8 ON
Multitap Delay

- Iteration of the single delay.
- By incorporating multiple delays at different times, and adding them all together the effect can be produced.
- Stored the required amount of samples then delayed versions of the input is played back at specific delay times.

Select Multitap Delay Mode
Feedback Multitap Delay

- Multitap delay on - Number “2”

The number “2” Signifies – Multitap Delay Mode
Circular Effect

- The output is sent from speaker to speaker as time goes by.
- It gives a simulation of a moving sound if the user is placed in the center of all the speakers.

Select Circling Sound Mode
Feedback Circling Sound

Circling Sound Effect On – Number “3”

The number “3” Signifies – Circling Sound Mode

<table>
<thead>
<tr>
<th>LEDS</th>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Closing Comments

This system is quite flexible because of the options that the linking of the Fade, Balance and Volume give.

Any modified input signal with one stereo output could be passed into this function and the necessary outputs would be calculated.
Ronald Ramdhan
ECE 3551
Microcomputer Systems
Final Project

“Speaker and Sound Modulation”
Problem Statement

To create an audio system that can service the needs of a typical user while further demonstrating the wide array of effects the Blackfin board could produce.

Most audio systems today, in both home and car audio, allow users to modify the input audio signal before it is processed to the output especially where multiple speaker outputs are in use. The basic and now expected options are volume control, balance control and fader control. The premise of this project was to recreate these options while using the Blackfin chip as the center of operations.

The additional effects that were added are well known and widely used. The first is a digital delay, which simply plays a sound with a delayed component of itself. The second option is a Multitap Delay. This type of delay is an iteration of the digital delay because it incorporates multiple delays of the same sound playing in harmony with the present output. These delay functions can create very interesting sound effects that are not only different but pleasing to the human ear.

The Multitap delay even produces the effect of multiple speakers when used. This may be because the same sound is heard over and over but at different gains due to the delays, so it seems as if the sound is coming from many speakers instead of one. Hence the usefulness of this effect is clear, virtual speaker sources.

The last effect is a circling sound effect which basically pushes the sound from one speaker to the other after a specified time has expired. If the time is short enough, the illusion of moving sound can be created very easily.

An important feature that is required by almost any system is feedback. A user needs to know exactly where a system is at any given time. The Blackfin board gives 6 LEDs which is one way of providing feedback. A problem was that such a small number of LEDs significantly reduces the amount of information that can be provided to the user at any given time.
Overall System Functions

The entire audio system can be summarized into 7 basic features:-

1. Original(Unmodified) Input Signal
2. Volume Control
3. Balance (Left and Right Speaker) Control
4. Fader (Front and Back Speaker) Control
5. Digital Delay
6. Multitap Delay
7. Circling Effect

Each of these features will now be explained in parts in order to display their functions and capabilities.
**System Mode Selection**

The 7 different modes had to be selected in some way. This was done using the push buttons. The first push Button, PF8 controlled the original input source being ON / OFF. Once the user has chosen the “off mode” variations can be done to the circuit.

The second button, PF9 controlled the modes that were to be used.

1. The first mode was the Volume control button. The other two switches, PF10 and PF11 would then control either decreasing or increasing the volume respectively.
2. The second mode would turn on the Balance Control. As before, PF10 and PF11 would control either front or back audio output.
3. The third mode would turn on the Fader Control. Again the other two buttons, PF10 and PF11, would control the signal location.
4. The fourth mode would create a digital delay effect on the audio input. PF10 and PF11 controls the delay time’s magnitude by increasing or decreasing it.
5. This mode would create the Multitap delay effect
6. This mode would create the Circling effect.
Original Signal Output

It must be noted that the basis of this program is the Analog Devices Audio Talkthrough I2S code. All of the declarations to the audio codec were already done so the modifications required were minor before elaborate coding was done for system functions such as volume control. All modifications are documented in this report.

In the Audio Talkthrough code, there is a function called Process_Data. It replicates the input to the output, but requires two input channels. To clarify what is going on the variables must be discussed. The “Channel In” variables such as ‘iChannel0LeftIn’ are integer variables that hold the input audio signal that are temporarily stored in the receiver buffer of the codec. The variable is called iRxBuffer[], and by copying its values, the signal can be saved. The “Channel Out” variables such as ‘iChannel0LeftOut’ are also integer variables that hold the output data that is copied to the iTxBuffer[] which transmits the signal out to the codec.

The code was modified to use only one channel, that is one stereo input was taken into the system and it was copied to two separate stereo output channels. The code shows how this is done. The channels are distinguished by the number in them…such as “iChannel0RightOut” means the channel 0’s right output and so forth.

```c
void Process_Data(void)
{
    iChannel0LeftOut = iChannel0LeftIn;  //Channel Zero
    iChannel0RightOut = iChannel0RightIn;
    iChannel1LeftOut = iChannel0LeftIn;  //Channel One
    iChannel1RightOut = iChannel0RightIn;
}
```
**Volume Control**

Volume control means to adjust the gain of the output signals. In larger systems today this is usually employed using an amplifier to add gain, however in this case the highest gain or volume would be the highest input volume. Hence any changes to the volume would mean reducing the gain to lower that the input’s highest volume.

At first it can be quickly noticed that multiplying the input by a factor of zero would dissolve the signal and turn it off. The other threshold was that by multiplying the signal by an integer value of 35000, the output would have a gain that is similar to the input. Using this conclusion, the volume control would multiply the signal by integer increments between 0 and 35000 in order to either increase or decrease the volume. In this system, the user was given 7 steps of volume to control. This meant that each volume increment would cause a addition factor of 5000 into the multiplication factor. The following pseudo code explains the process.

```plaintext
if(Volume Increase)
    Volume factor = Volume factor + 5000; //For Volume Factor < 35000
if(Volume Decrease)
    Volume factor = Volume factor - 5000; //For Volume Factor > 0
Output Signal = Input Signal * Volume Factor;
```

This integer multiplication works because all of the values are kept as integers when the multiplication is done. The final code employs these factors but is tied into other system features. The reason for this is that inorder to maximize the amount of control the user has over the speakers, the volume, fader and balance controls are linked together. Hence, speaker play location and volume is fully manipulable during usage. The basic sense is that that the gains from the volume, balance and fader are linked together to produce one total gain which is passed onto the speakers. This will be elaborated as the report continues.

The volume control is done by pressing either of two buttons on the EZ kit board. The first one would increase the volume factor as shown in the psuedo code, while the second would decrease the volume factor also shown in the pseudo code.
Feedback because an issue in a system that presents as many modes of operation such as this one. Hence in order to differentiate between modes a code system was developed. In the case of the Volume Mode, LEDs 4 and 5, displayed a signal of “01,” that is LED4 was off while LED5 was on. The other 4 LEDs from 6 to 9, then displayed the relative volume level. The actual code is relatively straightforward to understand. Is shows that when the volume factor is between a certain range LEDs would turn on to give visual representation. An added feature is that as the volume is increased, more LEDs are turned on to signify a greater volume, just as in modern stereo systems.

```c
if(volumeVal == 2)
    LEDtemp = 0x04;
if(volumeVal <= 4 && volumeVal > 2)
    LEDtemp = 0x04 | 0x08;
if(volumeVal <= 6 && volumeVal >= 4)
    LEDtemp = 0x04 | 0x08 | 0x10;
if(volumeVal <= 7 && volumeVal >= 6)
    LEDtemp = 0x04 | 0x08 | 0x10 | 0x20;

LED = 0x02 | LEDtemp;//turn LED4 ON
```

LEDtemp is a temporary value that was created to hold the value of the LEDs that require to be turned on every time the volume value is changed.
Balance Control

Balance control gives a user the option of moving the input signal to the left or to the right incrementally. In this program, the idea of the volume control was employed such that a gain was used to control the “volume” on each speaker. As the left button was depressed, the gain on the left speaker would increase until it was at its maximum then the gain on the right speaker would be decreased until it became zero. This meant that at the full left position, only the left speaker would play. The same analogy was used for the right speaker. Since 2 channels were implemented, the left channel output was copied to the two left outputs, and the right output was copied to both right outputs. The channels were pictured as follows.

```
L                  R
5        4       3           2            1             0            1          2           3          4          5
```

From the diagram it can be noted that there is a 5 step gain factor between zero output and output. So when L =5 and R =5, the output is at full range on both sides. To create the balance effect, some output has to be deducted as the user interacts with the program. If a right effect is desired every time the button is pressed, the gain on the left side would reduce from 5 to 0 in 5 steps. Vice versa would occur if the left effect is created. The tricky part was if the position of either R or L is not at its maximum value…5. This was simply accounted for by adding the gain factor until it reaches 5 or its maximum attainable value and then decreasing the gain on the other channel. An initial attempt decreased the gain of one speaker side while increasing the other. This was problematic because once the balance is used, the signal can never be re-centered with both speakers at full gain. Hence the above method was used, where a channel side would experience full gain before the other would be decremented. The following pseudo code demonstrates the basic operation of the balance circuit.

```
if(Left Required)
    if(Left Gain == MAX)
        Decrement Right Gain;
    else
        Increment Left Gain;
    Output = (Left Input * Left Gain) + (Right Input * Right Gain);
if(Right Required)
    if(Right Gain == MAX)
        Decrement Left Gain;
    else
        Increment Right Gain;
    Output = (Left Input * Left Gain) + (Right Input * Right Gain);
```
For user feedback in this case, the balance mode was represented by a value of “10” on LEDs 4 and 5. The other 4 LEDs were then used to display the relative position of the speakers when the balance mode is turned on. As the balance is shifted from the to the left, the LED 7 turns on when it is half way to signify this fact. When the balance is totally to the left the LED 6 turns on and LED 7 turns off. The same type of effect occurs when the system shifts to the right except that LED 8 turns on half way and LED 9 turns on when it is fully shifted to the right. The following code shows how this was implemented in the program.

```c
if(leftControl == 0 && rightControl == 0)
    LEDtemp = 0x00;
if(rightControl == 2 )
    LEDtemp = 0x08;
if(leftControl == 5 && rightControl == 0)
    LEDtemp = 0x04;
if(leftControl == 2)
    LEDtemp = 0x10;
if(rightControl == 5 && leftControl == 0)
    LEDtemp = 0x20;
```

The code basically turns on the light whenever the gain of the left is at max or midway (2). It does the same for the right channel.
Fader Control

This control gives the user the power to move the output from the front outputs to the rear outputs incrementally. It is basically a replica of the balance function with the exception that instead of adding gain to the one side of the outputs on both channels, the gain is applied to the entire channel. The pseudo code is as follows.

```plaintext
if(Front Required)
    if(Front Gain == MAX)
        Decrement Back Gain;
    else
        Increment Front Gain;
Output = (Front Input * Front Gain) + (Back Input * Back Gain);
if(Back Required)
    if(Back Gain == MAX)
        Decrement Front Gain;
    else
        Increment Back Gain;
Output = (Front Input * Front Gain) + (Back Input * Back Gain);
```

The feedback notation was different in this case. For the fader the LEDs 4 and 5 were both on signifying a “11.” The other 4 LEDs were again used to represent the relative position of the channel output with the higher gain. If the gain approached the front, the LEDs 7 and 6 would signify this. However, if the gain approached the back, LEDs 8 and 9 would turn on respectively. The following code snippet shows the actual code used.

```plaintext
if(upControl == 0 && downControl == 0)
    LEDtemp = 0x00;
if(downControl == 2)
    LEDtemp = 0x08;
if(upControl == 5 && downControl == 0)
    LEDtemp = 0x04;
if(upControl == 2)
    LEDtemp = 0x10;
if(downControl == 5 && upControl == 0)
    LEDtemp = 0x20;
FadVolBal(iChannel0LeftIn, iChannel0RightIn);
LED =  0x02 | 0x01 | LEDtemp;
```
Total Volume, Balance and Fader Control

As stated earlier, it was desired to give the user total control over the speakers through this system. For example, the user can push the entire output to the left using the balance control and then push the output to the front using the fader control. This would cause the front-left speaker to only be playing while the rest are in the off mode. The volume of this particular speaker can also be controlled using the volume control.

In order to fully demonstrate the understanding of this task, the speakers were placed in a matrix like table.

<table>
<thead>
<tr>
<th></th>
<th>Lg</th>
<th>Rg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ug</td>
<td>L0</td>
<td>R0</td>
</tr>
<tr>
<td>Dg</td>
<td>L1</td>
<td>R1</td>
</tr>
</tbody>
</table>

Where Lg = Left gain  
Rg = Right Gain  
Ug = Front gain  
Dg = Back Gain

The symbols L0, R1 represent the speaker position in relation to their channel. For example, the Channel 0-Left Speaker is L0. The main use of this matrix is that L0 would have a gain due to Ug and Lg. However Ug was created by the Fade function while Lg was created by the Balance function. The problem was linking them. Since the highest gain factor was sought at ‘5’ the highest gain ratio could be 5. To reiterate this point, it is useful to recall the initial technique of the gain used where the maximum gain was accepted as 35000. Therefore by multiplying 5 by 7000, the maximum gain can be achieved as required in this case.

A method was required to get an equivalent way or representing these two gains. This was done by multiplying Ug by Lg which gave a number. To ensure that it fell within the 0-5 range, it was divided by 5. This brought the gain back down to the range used and gave the required ratio. It was now possible to use the board to isolate one speaker source by just using the push buttons.

To implement the volume into the controls, the same logic was employed. The gain was given 7 intervals to allow the user 7 levels of control. As stated before the max volume would be the input signal’s maximum amplitude in order to avoid any distortion in the signal. The overall function then simply multiplied the gain of the volume, the fader gain and the balance gain for each speaker. The result was divided by “35” in order to keep the gain range between 0 and 5. For example if all the gains were at their maximum values, then the result of multiplication would be $7 \times 5 \times 5 = 175$. Now $175 / 35$ gives 5, which was used as the max gain ratio in this project.
The following code shows the function used to incorporate these three functions.

```c
void FadVolBal( int leftInput, int rightInput)
{
    short temp1 = leftInput >> 8;
    short temp2 = rightInput >> 8;

    int tempFadeL0 = ( leftControl * upControl * volumeVal) / 35;
    int tempL0 = 7000 * tempFadeL0; //to get 5 jumps to 35000

    int tempFadeL1 = ( leftControl * downControl * volumeVal ) / 35;
    int tempL1 = 7000 * tempFadeL1; //to get 5 jumps to 35000

    int tempFadeR0 = ( rightControl * upControl * volumeVal) / 35;
    int tempR0 = 7000 * tempFadeR0; //to get 5 jumps to 35000

    int tempFadeR1 = ( rightControl * downControl * volumeVal ) / 35;
    int tempR1 = 7000 * tempFadeR1; //to get 5 jumps to 35000

    tempL0 = temp1 * tempL0;
    tempL1 = temp1 * tempL1;

    tempR0 = temp2 * tempR0;
    tempR1 = temp2 * tempR1;

    iChannel0LeftOut = tempL0 >> 7;
    iChannel1LeftOut = tempL1 >> 7;
    iChannel0RightOut = tempR0 >> 7;
    iChannel1RightOut = tempR1 >> 7;
}
```

It can be noticed that the function accepts two integer inputs. The reason for this is that there are additional effects to be added to the output. In order to pass the effects to whichever speaker/s the user requires, the modified output from each function must be passed through this function before it can go to the output so that all the fade/balance/volume settings are kept.
Digital delay

In this section a digital delay was created using a simple delay cycle. The user was to be given the option of increasing or decreasing the delay as necessary. The delay ranged from 45ms to 60ms. Using the fact that 10,000 samples was roughly 1 second of sound playback the relative values were calculated as follows:-

1 sec = 10000 samples
1 ms = 10 samples
45 ms = 450 samples
60 ms = 600 samples

The digital delay accepts an input, delays it by a specified time and then adds it to the current input value and is sent out as an output. The following flow chart demonstrates.

The input signal is represented by \( x[n] \), the output is shown as \( y[n] \) and the delay factor of 1 is represented by \( Z^{-1} \). The method used was to store the required amount of samples and when completed, they were added to the input signal. A gain factor of one was used. In the case where 45ms delay was created, an array was created that held that 450 samples. Once it was filled to 450 values, its pointer would loop back to the start and first push out the sample value, then overwritten it with a new value. The same logic was taken for the other delay lengths. In order to change the delay time, the user would push either SW6 or SW7 to decrease or increase the delay time respectively. This change would basically cause the upper limit of the array to change to program set default values.

At this point, feedback is required but it seems that options are limited because the method of counting to “4” in binary has been exhausted. The next option to display three more modes while leaving 4 other LEDs free for further user feedback, such as delay length, was to use the last two LEDs 8 and 9. In this case they were lit in the form “01.” The other four LEDs from 4 to 7 are then used to display the delay length size. The code is shown after the function used for the delay.
The following shows the actual function used to create this effect.

```c
void DigitalDelay(void)
{
    if(start)
    {
        iChannel0LeftOut = iChannel0LeftIn + samplesL[index];
        iChannel0RightOut = iChannel0RightIn + samplesR[index];
    }
    else
    {
        iChannel0LeftOut = iChannel0LeftIn;
        iChannel0RightOut = iChannel0RightIn;
    }
    samplesL[index] = iChannel0LeftIn;
    samplesR[index] = iChannel0RightIn;
    index++;
    if(index == delayLength)
    {
        index = 0;
        start = 1;
    }
}
```

As seen, the digital delay effect can only start when there are enough values in the samples array. The variable “delayLength” sets the length of the delay and hence the amount of values held in the delay array.

The following shows the code for the LEDs or feedback.

```c
if(delayLength <= 587)
    LEDtemp = 0x01;
if(delayLength <= 724 && delayLength > 587)
    LEDtemp = 0x02;
if(delayLength <= 861 && delayLength > 724)
    LEDtemp = 0x04;
if(delayLength <= 1000 && delayLength > 861)
    LEDtemp = 0x08;
LED = 0x20 | LEDtemp;
```
Multi tap delay

The Multitap delay is basically an extension of the digital delay method. The only change is that multiple delays are added to the input at the same time. This problem was resolved in steps in order to simplify it. The following chart shows how the Multitap delay operation works.

As can be seen, the output is the addition of various levels of delays plus the original input signal. In order to have the data, samples had to be stored, the catch was that each delay had to be given a time length. The values were set as one delay equals 45ms or 450 samples. Hence in order to replicate the above diagram, a total of $450 \times 5 = 2250$ samples had to be stored. This was similarly done using an array. The delays were situated at different points in the array which was a key part in getting the proper delay effect. The line below shows where each delay would stand at the instant the 2250 samples are stored.
As expected the 1st delay is closest to the input signal because it is 450 samples behind it, while the 5th delay is 2250 behind. The output was simply the input plus these delayed samples. The delayed samples could then be extracted by adding 450, 900, 1350 or 1800 to the array pointer. The only problem was then when the pointer moved forward 450 samples the 1st delay would overflow the limits of the array, and the rest would also as time continues forward. The way to get around this was to picture exactly where the overflow would occur and where it should land. The overflow show basically wrap around the array and add its remainder to the start of the array. For example once the value of the first delay exceeds 2250, the following computation is done. ANS = 1800 – (2250- POINTER). This brings the pointer back to the front of the array and hence correlates the proper value. This was done for each delay and hence the multi tap delay was created.

At first, the gain on all the delayed outputs were kept as unity, however by letting the current input be the master, the effect was drastically clearer. The delayed signals were given gains that made the longest delay the smallest amplitude with the shortest delay had the second largest gain after the input signal.

For feedback the LED’s 8 and 9 were lit as “10.” This again just means that LED 8 was on and LED was off.
The following code shows how it was actually implemented in c language.

```c
void MultiDelay(void)
{
    short temp = iChannel0LeftIn >> 8;

    if(start5)   //2250 samples or 5 delays have been recorded
    {
        int fifthdelay = multisam[delay5];
        fifthdelay = fifthdelay * 35000;

        int temp4 = 450 + delay5;
        if(temp4 > 2250)
            temp4 = 450 - (2250 - delay5);
        int fourthdelay = multisam[temp4];
        fourthdelay = fourthdelay * 28000;

        int temp3 = 900 + delay5;
        if(temp3 > 2250)
            temp3 = 900 - (2250 - delay5);
        int thirddelay = multisam[temp3];
        thirddelay = thirddelay * 21000;

        int temp2 = 1350 + delay5;
        if(temp2 > 2250)
            temp2 = 1350 - (2250 - delay5);
        int seconddelay = multisam[temp2];
        seconddelay = seconddelay * 14000;

        int temp1 = 1800 + delay5;
        if(temp1 > 2250)
            temp1 = 1800 - (2250 - delay5);
        int firstdelay = multisam[temp1];
        firstdelay = firstdelay * 7000;

        iChannel0LeftOut = (temp + fifthdelay + fourthdelay +
                          thirddelay + seconddelay + firstdelay)  >> 7;
        iChannel0RightOut = iChannel0LeftOut ;

    }
    multisam[delay5] = temp;
    delay5++;

    if(delay5 == 2250)
    {
        delay5 = 0;
        start5 = 1;
    }
}
```

Again it can be noted that the output cannot start until the array is filled with delay values just as in the case of the digital delay.
Circular Effect

This was a simple effect that just rotates the output around to each speaker. The logic is straightforward in that it is simply playing an output for a specific time and then when the time expires, the output is shifted to another speaker and this process is looped every time all the speakers are played.

The time was roughly 3 seconds of play back and this was done by creating a counter that incremented 20000 times while playing the output on one channel. Once this expires, the output is pushed to another speaker and the process is repeated.

For feedback, the LEDs 8 and 9 would turn on displaying a “11” signal.

The code implementation is shown below.

```c
void Surround_Snd(void)
{
    if(iSpkLocation == 0)
    {
        if(check<20000)
        {
            iChannel0LeftOut = iChannel0LeftIn;
            check++;
        }
    }
    if(iSpkLocation == 1)
    {
        if(check<40000 && check >= 20000)
        {
            iChannel0RightOut = iChannel0RightIn;
            check++;
        }
    }
    if(iSpkLocation == 2)
    {
        if(check<60000 && check>=40000)
        {
            iChannel1RightOut = iChannel0RightIn;
            check++;
        }
    }
    if(iSpkLocation == 3)
    {
        if(check<80000 && check >= 60000)
        {
            iChannel1LeftOut = iChannel0LeftIn;
            check++;
        }
    }
}
```
Results

The final result was a fully functional system with no flaws when its system purpose is considered.

The volume, balance and fader controls work flawlessly in sync to create the required gain and speaker output that is desired by the user. The feedback systems are easy to understand represents the approximate state of the system where available.

The delay effects create some interesting variations in the sound that can be easily identified. Also because the system allows for an easy switch between original and modified outputs, comparison between the delayed effects and the original is literally a button click away.

The Multitap delay effect as stated above, gives a virtual speaker effect that is quite interesting. It makes the one speaker sound as if there are like 6 others. This illusion is only experienced if the user sits in the center of the speakers.

Closing Remarks

In closing, the problem was solved just as desired. The one issue was the feedback process. The user could have been given more feedback if there were more LEDs available so that more accurate system status signals could be given. The end result is a program that can accept any amount of effects and easily incorporate volume, balance and fader control into them. This means that the potential of expanding on this program base is unlimited. All a programmer needs to do is pass the modified output signal into the FadVolBal( ) function and then pass its output to the transmit buffer.

References / Bibliography

Analog Devices – Audio Talkthrough Code
www.analog.com

Dr. Veton Këpuska – Various Class Notes on the BF 533 EZ Kit Lite
Basic Equalizer

ECE 3553-03
Multifarious Systems 1
Xerxes Beharry
8th December, 2006
Problem Statement

• To create a system that was able to manipulate an audio signal so that the output is more pleasing to the ear. This means adding bass, midrange or heights to an incoming signal.

• The most basic of audio systems today allow an user to manipulate the audio signals, whether it be from one’s car stereo system or laptop. I attempted to create such a system, a basic equalizer paying special attention to the low and high frequency ranges because they are most easily recognizable.
My Attempt

• To mimic such a system capable of adjusting a few frequency bands

• To put into practice what was learnt from the ECE 3551 labs and lectures.
System Modes of Operation

• PF8 is used to toggle between increment and decrement
• LED 4 on represents increment and off represents decrement
• PF9 is used to adjust the low frequency band
• PF10 is used to represent the high frequency band
• If the max or min gain is reached for the low frequency band, it is indicated by LED5
• If the max or min gain is reached for the high frequency band, it is indicated by LED6
System Modes of Operation

- LED 7 is an intensity indicator for the low frequency band
- LED 8 is an intensity indicator for the high frequency band
- PF11 toggles between the original signal being heard on 4 speakers, and the original signal heard on only 2 specific speakers and the manipulated signal on the other 2 speakers. This was done for comparison purposes.
Implementation

• Any audio signal can be broken up into a band of frequencies.

• Lower frequencies have deeper sounding tones as compared to high frequencies.

• Higher frequencies have higher sounding tones.
Frequencies I concentrated on:

- **0-170 Hz**  Used to adjust the bass
- **600-3kHz**  Used to adjust the lower end of the midrange
- **3k-12kHz**  Used to adjust the higher end of the midrange
- **12k-16kHz** Used to adjust the height
What I Used

\[ x[n] \]

- 2\textsuperscript{nd} Order IIR Band 1 Filter
- 2\textsuperscript{nd} Order IIR Band 2 Filter
- 2\textsuperscript{nd} Order IIR Band i Filter
- 2\textsuperscript{nd} Order IIR Band N Filter

\[ g_2, g_i, g_N \]

\[ g_{\text{Master}} \]
What I Used

\[
H(z) = \frac{B_0 + B_1 z^{-1} + B_2 z^{-2}}{1 - A_1 z^{-1} - A_2 z^{-2}}
\]

\[
d[m] = x(m) - A_2 d[m - 2] - A_1 d[m - 1]
\]

\[
y[m] = B_2 d[m - 2] + B_1 d[m - 1] + B_0 d[m]
\]
Difficulties

• Using different types of filters with different data types.

• Obtaining proper filter coefficients from fdatool in MATLAB

• Implementing filters that overlap each other for accurate sound quality
Difficulties

- How to obtain a reasonable user adjustable gain for the different bands with the different types of filters used
Results

- The results of my second attempt at creating the equalizer were much more desirable than the first.

- A couple of mistakes that I made were attempting to manipulate too wide of a frequency band which actually defeats the purpose of the equalizer.

- Not filtering the entire frequency range I believe is another reason for the results of my first attempt.
Improvements

• Through trial an error it was realized that more bands that are available the more accurate the response that can be achieved from the equalizer. Ideally about 10 or 12 different bands would be sufficient to perform efficient equalizer functions on an input signal.

• Having preset modes such as Rock, Hall, Soft Rock etc which may save the user time in trying to figure out the best combination of settings.
Conclusion

- This project was considered a success as the result was as desired. It was very challenging but rewarding and although I have not created a full 12 band equalizer I believe that I now possess that knowledge necessary to do so given more time. I have learnt a lot about filters and their potential if implemented properly.

- I have also learnt to use the very powerful MATLAB fdatool. It has broadened my horizons to see the many different ways to implement a filter that would produce similar results. I have also learnt that if you type in “why” in MATLAB it can relieve some stress as seen below:
MATLAB

>> why
Pete wanted it that way.
>> why
I told me to.
>> why
A young kid told me to.
>> why
Bill insisted on it.
>> why
For the love of Jack.
>> why
The computer did it.
>> why
Some not very bald very rich system manager wanted it.
>> why
You told the rich system manager.
>> why
Some not excessively rich young and smart hamster obeyed some rich and not very bald system manager.
>> why
Cleve suggested it.
Special Thanks

• To:
  – Dr. V Kepuska
    • for taking the time to look at my first attempt.
  – Guogang Hua
    • for helping me out in the labs during the semester.
References

• http://my.fit.edu/~vkepuska/ece3551/
• http://www.altera.com/support/examples/dsp/dxm-dsp.html
• Dr. Kepuska’s class slides.