EEL 4914 Electrical Engineering Design

(Senior Design)

Final Report

21 April 2008

Project Title: 2.4Ghz Wireless Headphones

Team Name: D&M

Team Members:

Name: Donald Burnette	Name: Mike Franks
Email: donb@phys.ufl.edu	Email: cm.franks@gmail.com
Phone: (954) 520-3658	Phone: (352) 359-2449

Project Abstract:

The project will serve as a wireless audio communication device that can be used with any standard stereo 1/8" audio player such as a computer or iPod, as well as any TV/VCR/DVD player. The analog signals from a TV/VCR/DVD audio out or headphone jack will be filter the signal using an analog filter, and then sample the signal with an A/D. The A/D will send the signal to an Atmega2560, which will communicate to a wireless transmitter, which will send the data out via 2.4Ghz. The receiver will receive the 2.4Ghz signal, send the digital signal to another atmega2560 on the receiving side. The data will be then sent out a D/A, it will be amplified and connected to a headphone jack. The TV audio signal should be audible in real time from anywhere in the room.

Table of Contents

Table of Figures

Project Features/Objectives

The project will serve as a wireless audio communication device that can be used with any standard stereo 1/8" audio player such as a computer or iPod, as well as any TV/VCR/DVD player. The analog signals from a TV/VCR/DVD audio out or headphone jack will be filter the signal using an analog filter, and then sample the signal with an A/D. The A/D will send the signal to an Atmega2560, which will communicate to a wireless transmitter, which will send the data out via 2.4Ghz. The receiver will receive the 2.4Ghz signal, send the digital signal to another atmega2560 on the receiving side. The data will be then sent out a D/A, it will be amplified and connected to a headphone jack. The TV audio signal should be audible in real time from anywhere in the room. The wireless headphones should be able to produce high quality sound from a distance of up to 20m. The audio sampling will be done using 16 bit A/D, and each channel will be broadcast at a rate of 1Mbps. This will allow for a sampling rate up to 62.5kbps under ideal conditions.

Analysis of Competitive Products

This device exists in several forms, but most devices use IR transmission.

Sennheiser Set 820 Wireless Transmitter/Receiver System - **RF Transmitter / Receiver for Home Audio**

Price: $264

Sony® Infra-Red TV Listening Headphones

Price: $59

Concept/Technology

The only technology choices in the project are the wireless transmitter and microprocessor. The ATmega2560 was chosen as the microprocessor for several reasons. First, we are familiar with the AVR architecture and thus it will be easy to program. Second, the ATmega2560 model is twice the size of the ATmega128, allowing us the possibility of dedicating 32 general I/O ports to reading in both 16 bit A/D in parallel. The nRF24L01 was chosen because of its high data rate and relative ease of use. It takes care of all 2.4Ghz transmission/receiving and exposes a simple SPI interface.

Project Architecture

The architecture is straight forward and corresponds directly to the flow chart in Figure 1. The input data is fed into an analog anti-alias filter which cuts out any frequencies above 20KHz. The signal is then sampled by 16 bit A/D's, and the data is then buffered in the microprocessor until it is sent to the RF transmitted. Once received by the receiver, the data is buffered in the microprocessor until it is sent out through the D/A. The signal is smoothed with the same 8th order filter used as the anti-alias filter. Finally, the signal is sent through an amplifier and output on the headphone/speaker jacks.

Flow Charts and Diagrams

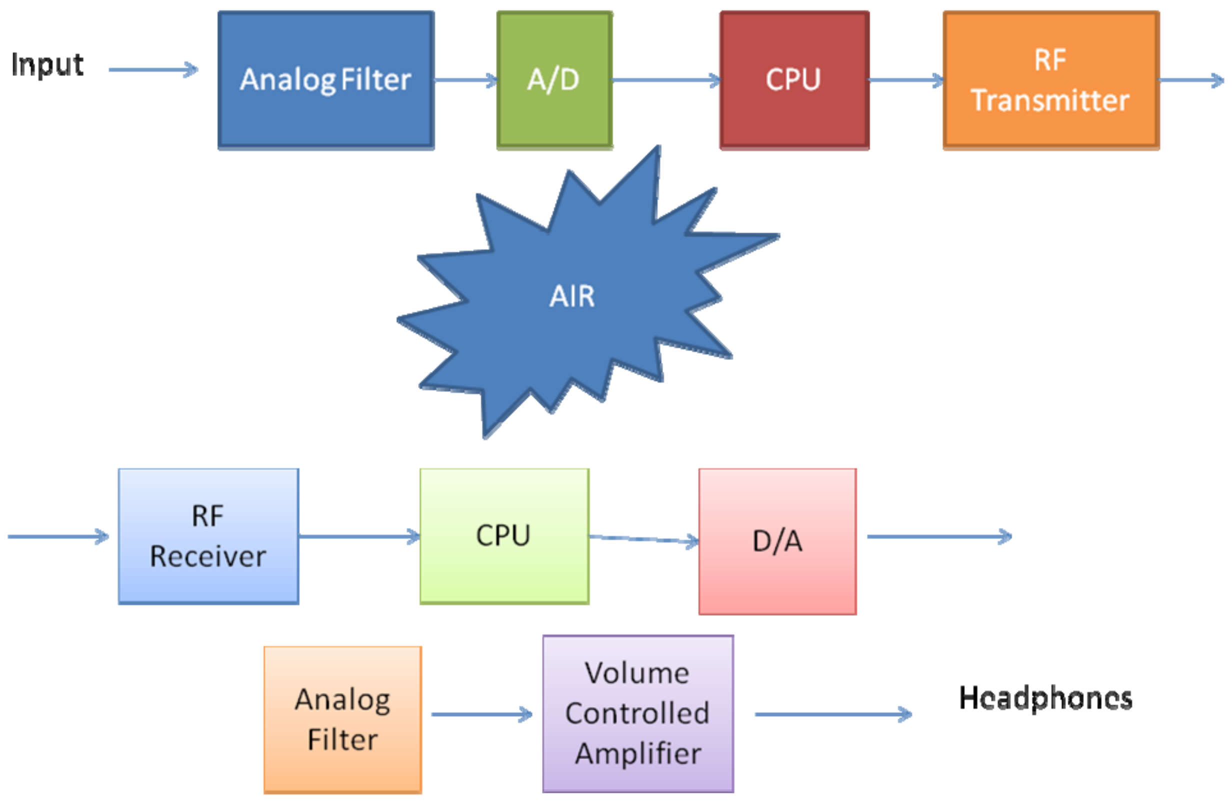


Figure 1 – Block diagram of System

Separation of Work

Don will be responsible for the board design and programming of the CPU while Mike will be responsible for the analog filter design, along with the analog filter debugging and tweaking.

Materials

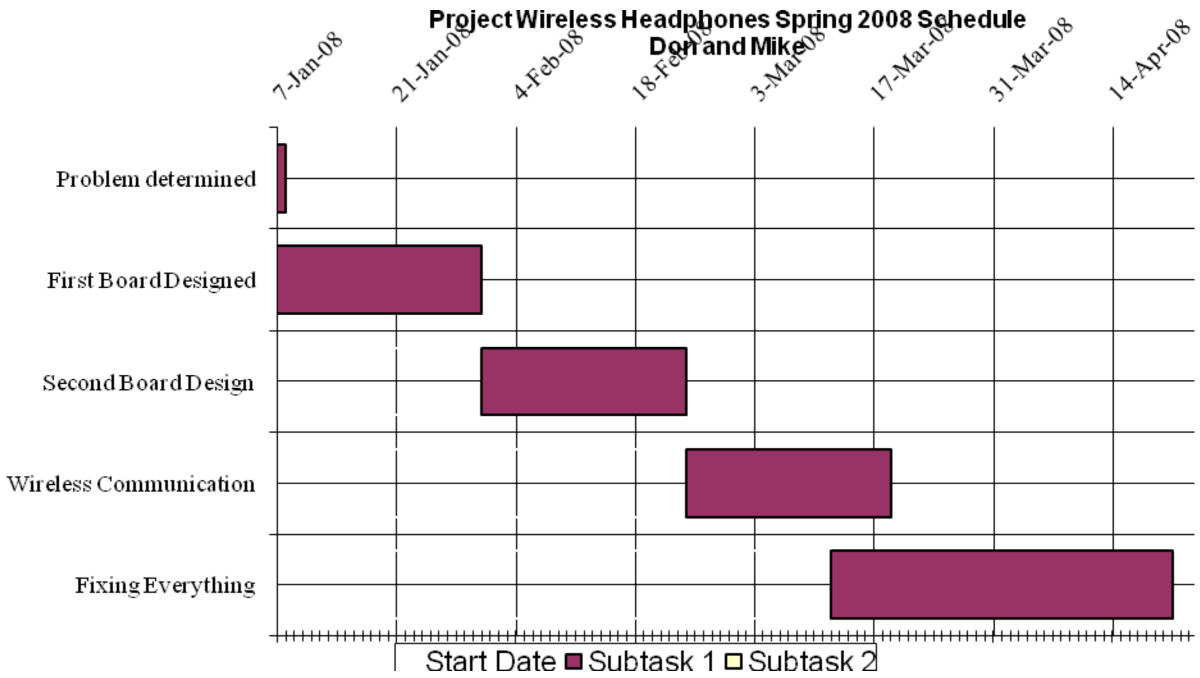
For a prototype transmitter and receiver, the circuits will contain more of less the same parts, and the costs are expected to be as follows:

$30	Wireless Transmitter/Receiver from Nordic (x2)
$20	Atmega2560 MicroProcessor (x2)

$33	Circuit Board Manufacturing (x2)
$5	Op Amp Circuits (TI - TLV2774) (x4)
$8	Two 16 Bit A/D (TI - ADS8505) (x2)
$16	Quad 16 Bit D/A (TI - DAC8544)
$30	Miscellaneous Parts

Total Cost: $248

Gantt Chart:



Code

Transmitter Board:

Main.c

```
/*--------------------------- Include Files -----------------------------*/
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <lcd.h>
#include <nRF24L01.h>
#include <interrupts.h>
#include <AtoD.h>
#include <common_util.h>

/*--------------------------- Main Program ----------------------------*/


int main ()
	{

	//Initailize LCD
	initLCD();
	LCDLine(" Starting...  ", 50, 3, 4); // Slow Display
	LCDClear();

	// Initialize nRF24L01
	init_nRF24L01(TX_MODE);

	//Initalize A To D
	_AtoDInit();

	DDRH = 0b11111100;
	PORTH = 0b00000000;

	//Initialize Interrupts
	_interruptInit();


	//	unsigned int count = 0;

	while (1)
	{
	}
}
```

nRF24L01.c

```
//****************************************************************************
//
//      nRF24L01 Wireless Module Functions
//      Donald Burnette
//      2-22-2008
//
//****************************************************************************

#include <avr/io.h>
#include <util/delay.h>
#include <nRF24L01.h>
#include <lcd.h>
#include <common_util.h>

//****************************************************************************
//
// Register Value definitions
//
// Below are the variables which hold the current state for each register's data in
//  the 24L01.
//
//****************************************************************************

unsigned char CONFIG_REG_TX =     0b01110010;   //     Enable Interupts, Disable CRC, Power
Up, TX Mode
unsigned char CONFIG_REG_RX =     0b00110011;   //     Enable Interupts, Disable CRC, Power
Up, RX Mode
unsigned char EN_AA_REG           =      0b00000000;   //     Disable "ShockBurst" Technology
unsigned char EN_RXADDR_REG =     0b00000001;   //     Enable only Pipe 0
unsigned char SETUP_AW_REG  =     0b00000001;   //     3 Byte Address, (may change later)
unsigned char SETUP_RETR_REG      =      0b00000000;   //     Disable Auto-Retransmit
unsigned char RF_CH_REG           =      119;          //     RF Channel (shown in decimal,
only 0-83 allowed by US law)
unsigned char RF_SETUP_REG  =     0b00001111;   //     2Mbps, Max Output Power
unsigned char OBSERVE_TX_REG      =      0b00000000;   //     Unused
unsigned char CD_REG              =      0b00000000;   //     Detect if a channel is being used

unsigned char RX_ADDR_P0_REG[3]   =      {0x7E, 0x7E, 0x7E}; // RX Pipe 0 Address

unsigned char RX_ADDR_P1_REG      =      0x00;
unsigned char RX_ADDR_P2_REG      =      0x00;
unsigned char RX_ADDR_P3_REG      =      0x00;
unsigned char RX_ADDR_P4_REG      =      0x00;
unsigned char RX_ADDR_P5_REG      =      0x00;

unsigned char TX_ADDR_REG[3]             =      {0x7E, 0x7E, 0x7E}; // TX Pipe 0 Address

unsigned char RX_PW_P0_REG  =     32;                  // RX Pipe 0 Width, Leave at 32


//****************************************************************************
```

```
//
// nRF24L01 Functionality
//
//***********************************************************************************

void init_nRF24L01(char MODE)
{

       //
       // SPI
       //

       DDRB   = 0b01000111;
       PORTB  |= 0b00000001;
       SPCR   = 0b01010000;
       SPSR   |= 0b00000001;

       // EN_AA - Enhanced ShockBurst (Disabled)
       write_instruction(EN_AA, EN_AA_REG, "EN_AA Set...");

       // EN_RXADDR - RX Pipes (Only pipe 0)
       write_instruction(EN_RXADDR, EN_RXADDR_REG, "Only Pipe 0 Set...");

       // SETUP_AW - 3 Byte Address
       write_instruction(SETUP_AW, SETUP_AW_REG, "3 Byte Address...");

       // SETUP_RETR - Disable Auto-Retransmit
       write_instruction(SETUP_RETR, SETUP_RETR_REG, "Retransmit Off...");

       // RF_CH - Set RF Channel (0-83 US Legal).. use 125 :)
       write_instruction(RF_CH, RF_CH_REG, "Set Channel...");

       // RF_SETUP - Set Data Rate 2.0MBps
       write_instruction(RF_SETUP, RF_SETUP_REG, "Data Rate 2Mbps...");

       // RX_ADDR_P0 - Set Pipe 0 Address (7E7E7E)
       write_instruction_data(RX_ADDR_P0, RX_ADDR_P0_REG, "RX Address Set...", 3);

       // TX_ADDR - Set Tx Address (7E7E7E)
       write_instruction_data(TX_ADDR, TX_ADDR_REG, "TX Address Set...", 3);

       // RX_PW_P0 - RX Pipe 0 Width (32 Bytes)
       write_instruction(RX_PW_P0, RX_PW_P0_REG, "Set Width 32...");

       LCDClear();

       // CONFIG - Setup Device and Power Up
       if (MODE == RX_MODE)
       {
              // CONFIG - Set Configuration as Receiver & Power Up
              write_instruction(CONFIG, CONFIG_REG_RX, "Rx: Powering Up...");

              wait(2);
```

```
		clear_CSN();
		spi_master_send(W_REGISTER | STATUS);	// Write to the Register
		spi_master_send(RX_DR_MASK);
		set_CSN();

		clear_CE();
	}
	else if (MODE == TX_MODE)
	{
		// CONFIG - Set Configuration as Transmitter & Power Up
		write_instruction(CONFIG, CONFIG_REG_TX, "Tx: Powering Up...");

		wait(2);

		clear_CE();
	}

}


unsigned char LCDCounter = 0;
void write_instruction(unsigned char ADDR, unsigned char data, char message[])
{

	unsigned char response = 0;

	clear_CSN();
	spi_master_send(W_REGISTER | ADDR);	// Write to the Register
	spi_master_send(data);
	set_CSN();

	clear_CSN();
	spi_master_send(R_REGISTER | ADDR);
	response = spi_master_send_receive(NOP);
	set_CSN();


	// verify
	if (data == response)
	{
		LCDLine("                   ", 0, LCDCounter%4 + 1, 1);
		LCDLine(message, 1, LCDCounter++%4 + 1, 1);
	}
	else
	{
		LCDClear();
		LCDLine("ERROR: ", 0, 1, 2);
		LCDLine(message, 0, 2, 2);
		for (long i = 0; i < 32000000; i++)
		{
			wait(1000);
		};
	}

}

void write_instruction_data(unsigned char ADDR, unsigned char data[], char message[], int
length)
{

	clear_CSN();
	spi_master_send(W_REGISTER | ADDR);	// Write to the Register

	for (int i = 0; i < length; i++)
	{
		spi_master_send(data[i]);
	}
	set_CSN();

	LCDLine("                   ", 0, LCDCounter%4 + 1, 1);
	LCDLine(message, 10, LCDCounter++%4 + 1, 1);

}

//clears the pin that is attached to the 24l01's CSN pin
void clear_CSN()
{
	PORTB &= 0b11111110;
}

//sets the pin that is attached to the 24l01's CSN pin
void set_CSN()
{
	PORTB |= 0b00000001;
}

//clears the pin that is attached to the 24l01's CSN pin
void clear_CE()
{
	PORTB &= 0b10111111;
}

//sets the pin that is attached to the 24l01's CSN pin
void set_CE()
{
	PORTB |= 0b01000000;
}
#define SPI_SEND_TIMEOUT   20

unsigned char spi_slave_send_receive(char data)
{
	int i = 0;
	SPDR = data;			// Transmit Byte
	while(!(SPSR & (1<<SPIF)))  // Wait for transmission complete
	{
		if (i++ > SPI_SEND_TIMEOUT) break;
	}
	return SPDR;				// Clear Flag
}


unsigned char spi_master_send_receive(char data)
{
	SPDR = data;				// Transmit Byte
	while(!(SPSR & (1<<SPIF)));	// Wait for transmission complete
	return SPDR;				// Clear Flag
}

void spi_slave_send(char data)
{
	int i = 0;
	SPDR = data;				// Transmit Byte
	while(!(SPSR & (1<<SPIF)))  // Wait for transmission complete
	{
		if (i++ > SPI_SEND_TIMEOUT) break;
	}
	SPDR;				// Clear Flag
}


void spi_master_send(char data)
{
	SPDR = data;					// Transmit Byte
	while(!(SPSR & (0b10000000)));	// Wait for transmission complete
	SPDR;						// Clear Flag
}
```

LCD.c

```
// ****************
//
//	LCD Functions
//	Donald Burnette
//	2-4-2007
//
// ****************


#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <lcd.h>


//Writes a single character to the LCD Screen
void LCDChar(char character)
{
	char tempChar;

	//First nibble
	tempChar = character & 0xF0;
	tempChar |= 0x0A;
	PORTC  = tempChar;
	_delay_us(1);
	tempChar &= 0b11110111;
	PORTC  = tempChar;
	_delay_us(1);

	//Second nibble
	tempChar = character & 0x0F;
	tempChar *=  16;
	tempChar |= 0x0A;
	PORTC  = tempChar;
	_delay_us(1);
	tempChar &= 0b11110111;
	PORTC  = tempChar;
	_delay_us(1);

}

//Writes a single character to the LCD Screen
void LCDCommand(char character)
{

	char tempChar;

	//First nibble
	tempChar = character & 0xF0;
	tempChar |= 0x08;
	PORTC  = tempChar;
	_delay_us(1);
	tempChar &= 0b11110111;
	PORTC  = tempChar;
	_delay_us(1);

	//Second nibble
	tempChar = character & 0x0F;
	tempChar *=  16;
	tempChar |= 0x08;
	PORTC  = tempChar;
	_delay_us(1);
	tempChar &= 0b11110111;
	PORTC  = tempChar;
	_delay_us(1);

}

//Clears screen and writes message to LCD
void LCDLine(char characters[], int delay, int x, int y)
{
        //Clear Display
        LCDjump(x, y);
        int i=0;
        int j=0;

        while (characters[i] != '\0' && i < 50)
        {
                LCDChar(characters[i]);
                for(j=0; j< delay; j++)
                {
                        _delay_ms(1);
                }
                i++;
                _delay_us(100);
        }

}

//Writes an integer value to the screen
void LCDNumber(unsigned int number, int delay, int x, int y)
{
        char charNumber[7];
/*
        if (number < 0)
        {
                number = -1*number;
                charNumber[0] = '-';
        }
        else*/
        {
                charNumber[0] = ' ';
        }
        if (number > 9999)
        {
                charNumber[1] = 0x30+(number/10000)%10;
                charNumber[2] = 0x30+(number/1000)%10;
                charNumber[3] = 0x30+(number/100)%10;
                charNumber[4] = 0x30+(number/10)%10;
                charNumber[5] = 0x30+(number)%10;
                charNumber[6] = '\0';
        }
        else if (number > 999)
        {
                charNumber[1] = 0x30+(number/1000)%10;
                charNumber[2] = 0x30+(number/100)%10;
                charNumber[3] = 0x30+(number/10)%10;
                charNumber[4] = 0x30+(number)%10;
                charNumber[5] = ' ';
                charNumber[6] = '\0';
        }
        else
        {
                charNumber[1] = 0x30+(number/100)%10;
```

```
            charNumber[2] = 0x30+(number/10)%10;
            charNumber[3] = 0x30+(number)%10;
            charNumber[4] = ' ';
            charNumber[5] = ' ';
            charNumber[6] = '\0';
        }

        LCDLine(charNumber, delay, x, y);

}

// Code initiallizes LCD and display's Welcome
void initLCD()
{
                PORTC = 0;
                DDRC  = 0xFF;

        LCDCommand(0b00110011);          // Set 4 bit mode
        _delay_ms(2);
        LCDCommand(0b00110010);          // Set 4 bit mode
        _delay_ms(2);
        LCDCommand(0b00101000);          // Function Set N=1. F=0
        _delay_ms(2);
        LCDCommand(0b00000001);          // Clear Display
        _delay_ms(2);
        LCDCommand(0b00000110);          // Entry Mode Set ID=1, S = 0
        _delay_ms(2);
        LCDCommand(0b00001100);          // Display On: Cursor=0, Blink=0
        _delay_ms(2);

}

//Clears LCD Display
void LCDClear()
{
        LCDCommand(0b00000001);          // Clear Display
        _delay_ms(5);
}

// Moves the cursor to any position on the LCD screen anywhere from (1,0) to (4,19)
void LCDjump(int x, int y)
{
        char address = 0x80;

        if (-1 > y || y < 20)
        {
            switch (x)
            {
                case 1:
                    address |= (char)(y);
                    LCDCommand(address);
                    break;
                    case 2:
                          address |= (char)(64+y);
                          LCDCommand(address);
                    break;

                    case 3:
                          address |= (char)(20+y);
                          LCDCommand(address);
                    break;

                    case 4:
                          address |= (char)(84+y);
                          LCDCommand(address);
                    break;

                    default:
                    break;
              }
      }
      _delay_us(100);
}
```

Interrupts.c

```
// ****************
//
//      Interrupt Functions
//      Donald Burnette
//      2-19-2008
//
// ****************


unsigned char RDataH[256];
unsigned char RDataL[256];

unsigned char LDataH[256];
unsigned char LDataL[256];

unsigned char DataCounter = 0;
unsigned char SendCounter = 128;

unsigned char       STATUS_REG = 0;

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <DAC8544.h>
#include <nRF24L01.h>
#include <lcd.h>
```

```
//Initialization
void _interruptInit()
{

/********************* 16 Bit Timers *************************/

       //
       // Timer 1   PWM Mode (A to D Conversions 50KHz)
       //

       TCCR1A       = 0b00000011; // COM1A = 00 (Disconnected)
                                              // COM1B = 00 (Disconnected)
                                              // COM1C = 00 (Disconnected)
                                              // WGM1 = xx11 (CTC Mode)
       TCCR1B       = 0b00011001; // ICNC1 = 0, ICES1 = 0, Reserved = 0,
                                              // WGM1 = 11xx, CS12:0 = 001 (divider = 1)

       OCR1A  = 395; // 720
       TIMSK1 |= 0b00000001;

       //
       // Timer 3   PWM Mode (A to D Conversions 50KHz)
       //

       TCCR3A       = 0b00000011; // COM1A = 00 (Disconnected)
                                              // COM1B = 00 (Disconnected)
                                              // COM1C = 00 (Disconnected)
                                              // WGM1 = xx11 (CTC Mode)
       TCCR3B       = 0b00011001; // ICNC1 = 0, ICES1 = 0, Reserved = 0,
                                              // WGM1 = 11xx, CS12:0 = 001 (divider = 1)

       //OCR3A      = OCR1A;            // Conversion Rate * 8
       OCR3A  = (OCR1A + 1) * 16 - 1;    // Conversion Rate * 8
       TIMSK3 |= 0b00000001;

       TCNT1 = 0;
       TCNT3 = 30;


       sei();                           // Globally Enable Interrupts


}


       // PK7 = L_Byte
       // PK6 = L_Convst
       // PK5 = L_RD
       // PK4 = L_CS
       // PK3 = R_Byte
       // PK2 = R_Convst
       // PK1 = R_RD
       // PK0 = R_CS
```

```
ISR(TIMER1_OVF_vect)
{
	PORTH = PINH ^ 0b10000000;

	// Read Right Channel

	PORTK &= 0b11111011;	// Bring R_Convst Low
	PORTK |= 0b00000100;	// Bring R_Convst High

	RDataH[DataCounter] = PIND;
	//RDataL[DataCounter] = PINL;

	// Read Left Channel

	PORTK &= 0b10111111;	// Bring L_Convst Low
	PORTK |= 0b01000000;	// Bring L_Convst High

	LDataH[DataCounter] = PINA;
	//LDataL[DataCounter] = PINJ;

	DataCounter++;


	PORTH = PINH ^ 0b10000000;
}

ISR(TIMER3_OVF_vect)
{
	//
	// Transmit Data
	//
	sei();
	PORTH = PINH ^ 0b00100000;

	PORTB &= 0b11111110;	// Bring Slave Select Low

	SPDR = W_TX_PAYLOAD;			// Transmit Byte
	//_delay_us(.1);				// Wait for transmission complete
	_delay_us(1);				// Wait for transmission complete
	SPDR;					// Clear Flag

	// 1
	SPDR = RDataH[SendCounter];		// Transmit Byte
	_delay_us(.75);				// Wait for transmission complete
	SPDR;					// Clear Flag
	SPDR = LDataH[SendCounter];		// Transmit Byte

	SendCounter++;

	_delay_us(.9);				// Wait for transmission complete
	SPDR;					// Clear Flag
```

```
        SPDR;                                   // Clear Flag
        SPDR = RDataH[SendCounter];             // Transmit Byte
        _delay_us(1.2);                         // Wait for transmission complete
        SPDR;                                   // Clear Flag
        SPDR = LDataH[SendCounter];             // Transmit Byte

        SendCounter++;

        _delay_us(.9);                          // Wait for transmission complete
        SPDR;                                   // Clear Flag


        // 2
        SPDR = RDataH[SendCounter];             // Transmit Byte
        _delay_us(.75);                         // Wait for transmission complete
        SPDR;                                   // Clear Flag
        SPDR = LDataH[SendCounter];             // Transmit Byte

        SendCounter++;

        _delay_us(.9);                          // Wait for transmission complete
        SPDR;                                   // Clear Flag

        SPDR;                                   // Clear Flag
        SPDR = RDataH[SendCounter];             // Transmit Byte
        _delay_us(1.2);                         // Wait for transmission complete
        SPDR;                                   // Clear Flag
        SPDR = LDataH[SendCounter];             // Transmit Byte

        SendCounter++;

        _delay_us(.9);                          // Wait for transmission complete
        SPDR;                                   // Clear Flag

        // 3
        SPDR = RDataH[SendCounter];             // Transmit Byte
        _delay_us(.75);                         // Wait for transmission complete
        SPDR;                                   // Clear Flag
        SPDR = LDataH[SendCounter];             // Transmit Byte

        SendCounter++;

        _delay_us(.9);                          // Wait for transmission complete
        SPDR;                                   // Clear Flag

        SPDR;                                   // Clear Flag
        SPDR = RDataH[SendCounter];             // Transmit Byte
        _delay_us(1.2);                         // Wait for transmission complete
        SPDR;                                   // Clear Flag
        SPDR = LDataH[SendCounter];             // Transmit Byte

        SendCounter++;
```

```
_delay_us(.9);                                // Wait for transmission complete
SPDR;                                         // Clear Flag

// 4
SPDR = RDataH[SendCounter];             // Transmit Byte
_delay_us(.75);                               // Wait for transmission complete
SPDR;                                         // Clear Flag
SPDR = LDataH[SendCounter];             // Transmit Byte

SendCounter++;

_delay_us(.9);                                // Wait for transmission complete
SPDR;                                         // Clear Flag

SPDR;                                         // Clear Flag
SPDR = RDataH[SendCounter];             // Transmit Byte
_delay_us(1.2);                               // Wait for transmission complete
SPDR;                                         // Clear Flag
SPDR = LDataH[SendCounter];             // Transmit Byte

SendCounter++;

_delay_us(.9);                                // Wait for transmission complete
SPDR;                                         // Clear Flag

// 5
SPDR = RDataH[SendCounter];             // Transmit Byte
_delay_us(.75);                               // Wait for transmission complete
SPDR;                                         // Clear Flag
SPDR = LDataH[SendCounter];             // Transmit Byte

SendCounter++;

_delay_us(.9);                                // Wait for transmission complete
SPDR;                                         // Clear Flag

SPDR;                                         // Clear Flag
SPDR = RDataH[SendCounter];             // Transmit Byte
_delay_us(1.2);                               // Wait for transmission complete
SPDR;                                         // Clear Flag
SPDR = LDataH[SendCounter];             // Transmit Byte

SendCounter++;

_delay_us(.9);                                // Wait for transmission complete
SPDR;                                         // Clear Flag

// 6
SPDR = RDataH[SendCounter];             // Transmit Byte
_delay_us(.75);                               // Wait for transmission complete
SPDR;                                         // Clear Flag
SPDR = LDataH[SendCounter];             // Transmit Byte
```

```
        SendCounter++;

        _delay_us(.9);                                  // Wait for transmission complete
        SPDR;                                           // Clear Flag

        SPDR;                                           // Clear Flag
        SPDR = RDataH[SendCounter];             // Transmit Byte
        _delay_us(1.2);                                 // Wait for transmission complete
        SPDR;                                           // Clear Flag
        SPDR = LDataH[SendCounter];             // Transmit Byte

        SendCounter++;

        _delay_us(.9);                                  // Wait for transmission complete
        SPDR;                                           // Clear Flag

        // 7
        SPDR = RDataH[SendCounter];             // Transmit Byte
        _delay_us(.75);                                 // Wait for transmission complete
        SPDR;                                           // Clear Flag
        SPDR = LDataH[SendCounter];             // Transmit Byte

        SendCounter++;

        _delay_us(.9);                                  // Wait for transmission complete
        SPDR;                                           // Clear Flag

        SPDR;                                           // Clear Flag
        SPDR = RDataH[SendCounter];             // Transmit Byte
        _delay_us(1.2);                                 // Wait for transmission complete
        SPDR;                                           // Clear Flag
        SPDR = LDataH[SendCounter];             // Transmit Byte

        SendCounter++;

        _delay_us(.9);                                  // Wait for transmission complete
        SPDR;                                           // Clear Flag

        // 8
        SPDR = RDataH[SendCounter];             // Transmit Byte
        _delay_us(.75);                                 // Wait for transmission complete
        SPDR;                                           // Clear Flag
        SPDR = LDataH[SendCounter];             // Transmit Byte

        SendCounter++;

        _delay_us(.9);                                  // Wait for transmission complete
        SPDR;                                           // Clear Flag

        SPDR;                                           // Clear Flag
        SPDR = RDataH[SendCounter];             // Transmit Byte
        _delay_us(1.2);                                 // Wait for transmission complete
        SPDR;                                           // Clear Flag
```

```
	SPDR = LDataH[SendCounter];		// Transmit Byte

	SendCounter++;

	_delay_us(.9);					// Wait for transmission complete
	SPDR;						// Clear Flag


	PORTB |= 0b00000001;	// Bring Slave Select High

	_delay_us(1);




	PORTH = PINH ^ 0b00000100;

	PORTB |= 0b01000000;
	_delay_us(10);
	PORTB &= 0b10111111;

	PORTH = PINH ^ 0b00000100;


	PORTB &= 0b11111110;		// Bring Slave Select Low

	SPDR = (W_REGISTER | STATUS);	// Transmit Byte
	_delay_us(1);				// Wait for transmission complete
	SPDR;						// Clear Flag

	SPDR = TX_DS_MASK;			// Transmit Byte
	_delay_us(1);				// Wait for transmission complete
	SPDR;						// Clear Flag

	PORTB |= 0b00000001;		// Bring Slave Select High
/*
	PORTB &= 0b11111110;		// Bring Slave Select Low

	SPDR = FLUSH_TX;			// Transmit Byte
	_delay_us(1);				// Wait for transmission complete
	SPDR;						// Clear Flag

	PORTB |= 0b00000001;		// Bring Slave Select High
*/

	PORTH = PINH ^ 0b00100000;

	cli();

}
```

```
void delay(int time)
{
	int j;
	for (int i = 0; i < time; i++)
	{
		j++;
	}
}
```

Receiver Board:

Interrupts.c

```
// ****************
//
//	Interrupt Functions
//	Donald Burnette
//	2-19-2008
//
// ****************


/*-------------------------- Global Variables -------------------------- */

unsigned char	Data_In_RH = 0;
unsigned char	Data_In_RL = 0;
unsigned char	Data_In_LH = 0;
unsigned char	Data_In_LL = 0;

unsigned char RDataH[256];
unsigned char RDataL[256];

unsigned char LDataH[256];
unsigned char LDataL[256];

unsigned char DataCounter = 0;
unsigned char PlayCounter = 128;

char data[96];

int good_packets = 0;
int bad_packets = 0;

/*-------------------------- Include Files -------------------------- */

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
```

```
#include <DAC8544.h>
#include <nRF24L01.h>
#include <lcd.h>


//Initialization
void _interruptInit()
{

/********************** 16 Bit Timers **************************/

	//
	// Timer 1	PWM Mode (A to D Conversions 50KHz)
	//

	TCCR1A		= 0b00000011; // COM1A = 00 (Disconnected)
					     // COM1B = 00 (Disconnected)
					     // COM1C = 00 (Disconnected)
					     // WGM1 = xx11 (CTC Mode)
	TCCR1B		= 0b00011001; // ICNC1 = 0, ICES1 = 0, Reserved = 0,
					     // WGM1 = 11xx, CS12:0 = 001 (divider = 1)

	OCR1A  = 395; //390
	TIMSK1 |= 0b00000001;

	//
	// PCINT External Interrupts
	//

	PCICR  |= 0b00000001;	// Enable Interrupt PCINT[7..0]
	PCMSK0	|= 0b10000000;	// Enable Interrupt PCINT7


	sei();				// Globally Enable Interrupts


}

ISR(TIMER1_OVF_vect)
{

		PORTH = PINH ^ 0b00000001;



		PORTK &= 0b11111001;	// A1 A0 = 0 0
		PORTD = LDataH[PlayCounter];
		PORTL = 0;
		PORTK &= 0b11110111;	// Bring CS Low
		PORTK |= 0b00001000;	// Bring CS High

		PORTK |= 0b00000110;	// A1 A0 = 1 1
		PORTD = RDataH[PlayCounter];
```

```
             PORTL = 0;
             PORTK &= 0b11110111;       // Bring CS Low
             PORTK |= 0b00001000;       // Bring CS High



             PORTK |= 0b00100000;       // Bring LDAC High
             PORTK &= 0b11011111;       // Bring LDAC Low

             PlayCounter++;

             PORTH = PINH ^ 0b00000001;

}

unsigned char FIFO_STATUS_REG      =      0b00000000;   // Info About FIFO Data


ISR(PCINT0_vect)
{
       sei();
       PORTH = PINH ^ 0b10000000;

       if ((PINB & 0b10000000) == 0)
       {


             PCMSK0       &= 0b01111111;      // Disable Interrupt PCINT7

             PORTB &= 0b10111111;

             PORTB &= 0b11111110;       // Bring Slave Select Low

             SPDR = R_RX_PAYLOAD;                    // Transmit Byte
             //_delay_us(.1);                        // Wait for transmission complete
             _delay_us(1);                           // Wait for transmission complete
             SPDR;                                          // Clear Flag

             // 1
             SPDR = 0;                                      // Transmit Byte
             _delay_us(1.2);                                // Wait for transmission complete
             RDataH[DataCounter] = SPDR;             // Clear Flag
             SPDR = 0;                                      // Transmit Byte
             _delay_us(1.2);                                // Wait for transmission complete
             LDataH[DataCounter] = SPDR;             // Clear Flag
             DataCounter++;

             SPDR = 0;                                      // Transmit Byte
             _delay_us(1.2);                                // Wait for transmission complete
             RDataH[DataCounter] = SPDR;             // Clear Flag
             SPDR = 0;                                      // Transmit Byte
             _delay_us(1.2);                                // Wait for transmission complete
             LDataH[DataCounter] = SPDR;             // Clear Flag
```

```
DataCounter++;

// 2
SPDR = 0;                                    // Transmit Byte
_delay_us(1.2);                              // Wait for transmission complete
RDataH[DataCounter] = SPDR;          // Clear Flag
SPDR = 0;                                    // Transmit Byte
_delay_us(1.2);                              // Wait for transmission complete
LDataH[DataCounter] = SPDR;          // Clear Flag
DataCounter++;

SPDR = 0;                                    // Transmit Byte
_delay_us(1.2);                              // Wait for transmission complete
RDataH[DataCounter] = SPDR;          // Clear Flag
SPDR = 0;                                    // Transmit Byte
_delay_us(1.2);                              // Wait for transmission complete
LDataH[DataCounter] = SPDR;          // Clear Flag
DataCounter++;

// 3
SPDR = 0;                                    // Transmit Byte
_delay_us(1.2);                              // Wait for transmission complete
RDataH[DataCounter] = SPDR;          // Clear Flag
SPDR = 0;                                    // Transmit Byte
_delay_us(1.2);                              // Wait for transmission complete
LDataH[DataCounter] = SPDR;          // Clear Flag
DataCounter++;

SPDR = 0;                                    // Transmit Byte
_delay_us(1.2);                              // Wait for transmission complete
RDataH[DataCounter] = SPDR;          // Clear Flag
SPDR = 0;                                    // Transmit Byte
_delay_us(1.2);                              // Wait for transmission complete
LDataH[DataCounter] = SPDR;          // Clear Flag
DataCounter++;

// 4
SPDR = 0;                                    // Transmit Byte
_delay_us(1.2);                              // Wait for transmission complete
RDataH[DataCounter] = SPDR;          // Clear Flag
SPDR = 0;                                    // Transmit Byte
_delay_us(1.2);                              // Wait for transmission complete
LDataH[DataCounter] = SPDR;          // Clear Flag
DataCounter++;

SPDR = 0;                                    // Transmit Byte
_delay_us(1.2);                              // Wait for transmission complete
RDataH[DataCounter] = SPDR;          // Clear Flag
SPDR = 0;                                    // Transmit Byte
_delay_us(1.2);                              // Wait for transmission complete
LDataH[DataCounter] = SPDR;          // Clear Flag
DataCounter++;
```

```
// 5
SPDR = 0;                                   // Transmit Byte
_delay_us(1.2);                             // Wait for transmission complete
RDataH[DataCounter] = SPDR;             // Clear Flag
SPDR = 0;                                   // Transmit Byte
_delay_us(1.2);                             // Wait for transmission complete
LDataH[DataCounter] = SPDR;             // Clear Flag
DataCounter++;

SPDR = 0;                                   // Transmit Byte
_delay_us(1.2);                             // Wait for transmission complete
RDataH[DataCounter] = SPDR;             // Clear Flag
SPDR = 0;                                   // Transmit Byte
_delay_us(1.2);                             // Wait for transmission complete
LDataH[DataCounter] = SPDR;             // Clear Flag
DataCounter++;

// 6
SPDR = 0;                                   // Transmit Byte
_delay_us(1.2);                             // Wait for transmission complete
RDataH[DataCounter] = SPDR;             // Clear Flag
SPDR = 0;                                   // Transmit Byte
_delay_us(1.2);                             // Wait for transmission complete
LDataH[DataCounter] = SPDR;             // Clear Flag
DataCounter++;

SPDR = 0;                                   // Transmit Byte
_delay_us(1.2);                             // Wait for transmission complete
RDataH[DataCounter] = SPDR;             // Clear Flag
SPDR = 0;                                   // Transmit Byte
_delay_us(1.2);                             // Wait for transmission complete
LDataH[DataCounter] = SPDR;             // Clear Flag
DataCounter++;

// 7
SPDR = 0;                                   // Transmit Byte
_delay_us(1.2);                             // Wait for transmission complete
RDataH[DataCounter] = SPDR;             // Clear Flag
SPDR = 0;                                   // Transmit Byte
_delay_us(1.2);                             // Wait for transmission complete
LDataH[DataCounter] = SPDR;             // Clear Flag
DataCounter++;

SPDR = 0;                                   // Transmit Byte
_delay_us(1.2);                             // Wait for transmission complete
RDataH[DataCounter] = SPDR;             // Clear Flag
SPDR = 0;                                   // Transmit Byte
_delay_us(1.2);                             // Wait for transmission complete
LDataH[DataCounter] = SPDR;             // Clear Flag
DataCounter++;

// 8
SPDR = 0;                                   // Transmit Byte
```

```
		_delay_us(1.2);						// Wait for transmission complete
		RDataH[DataCounter] = SPDR;			// Clear Flag
		SPDR = 0;							// Transmit Byte
		_delay_us(1.2);						// Wait for transmission complete
		LDataH[DataCounter] = SPDR;			// Clear Flag
		DataCounter++;

		SPDR = 0;							// Transmit Byte
		_delay_us(1.2);						// Wait for transmission complete
		RDataH[DataCounter] = SPDR;			// Clear Flag
		SPDR = 0;							// Transmit Byte
		_delay_us(1.2);						// Wait for transmission complete
		LDataH[DataCounter] = SPDR;			// Clear Flag
		DataCounter++;



		PORTB |= 0b00000001;			// Bring Slave Select High


		// Command to clear RX flag
		PORTB &= 0b11111110;

		SPDR = (W_REGISTER | STATUS);	// Transmit Byte
		_delay_us(1);					// Wait for transmission complete
		SPDR;							// Clear Flag

		SPDR = RX_DR_MASK;				// Transmit Byte
		_delay_us(1);					// Wait for transmission complete
		SPDR;							// Clear Flag

		PORTB |= 0b00000001;

		PORTB |= 0b01000000;



		PCMSK0		|= 0b10000000;		// Enable Interrupt PCINT7

	}

	PORTH = PINH ^ 0b10000000;

	cli();
}



void Write_good()
{
			LCDNumber(good_packets, 0, 1, 13);
}
void Write_bad()
```

```
{
                    LCDNumber(bad_packets, 0, 2, 13);
}
```

Main.c

```
/*--------------------------- Function Prototypes ---------------------------*/

void manage_buttons(void);


/*---------------------------- Include Files -----------------------------*/
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <lcd.h>
#include <DAC8544.h>
#include <interrupts.h>
#include <nRF24L01.h>
#include <common_util.h>



/*---------------------------- Main Program ----------------------------*/


unsigned char       STATUS_REG1 = 0;
unsigned char volume = 80;


int main ()
       {

       //Initailize LCD
       initLCD();
       LCDLine(" Starting...  ", 50, 3, 4); // Slow Display
       LCDClear();

       // Initialize DAC
       DAC_Initialize();

       // Initialize Input Buttons
       DDRJ = 0b00000010;

       DDRH = 0b11111111;
       PORTH = 0b00000000;

       // Set Initial Volume
       PORTK &= 0b11111011;       // A1 A0 = 0 1
       PORTK |= 0b00000010;
       PORTD = volume;
       PORTL = 0;
       PORTK &= 0b11110111;       // Bring CS Low
```

```
	PORTK |= 0b00001000;		// Bring CS High

	PORTK |= 0b00100000;		// Bring LDAC High
	PORTK &= 0b11011111;		// Bring LDAC Low

	// Initialize nRF24L01
	init_nRF24L01(RX_MODE);

	LCDClear();
	LCDLine("Playing...", 0, 2, 4);

	set_CE();

	clear_CSN();
	spi_master_send(W_REGISTER | STATUS);	// Write to the Register
	spi_master_send(RX_DR_MASK);
	set_CSN();

	clear_CE();

	clear_CSN(); // Bring Slave Select Low

	spi_master_send(R_RX_PAYLOAD);


	for (int i = 0; i < 32; i++)
	{
		spi_master_send(0);
	}

	set_CSN();		// Bring Slave Select High


	// Command to clear RX flag
	clear_CSN();
	spi_master_send(W_REGISTER | STATUS);	// Write to the Register
	spi_master_send(RX_DR_MASK);
	set_CSN();

	set_CE();

	//Initialize Interrupts
	_interruptInit();

	while(1)
	{
/*
			Write_good();
			Write_bad();

		wait(100);
*/
```

```
        manage_buttons();
        //wait(10);
        }
}


#define volMax 5000

int volUpPressed = volMax;
int volDownPressed = volMax;
int volMutePressed = volMax;

int muted = 0;


void manage_buttons()
{

        if ((PINJ & 0b00001000) == 0b00001000)
        {
                if (volDownPressed-- == 0)
                {
                        volDownPressed = volMax;
                        if (volume > 0)
                        {
                                volume--;
                                cli();
                                // Set Initial Volume
                                PORTK &= 0b11111011;        // A1 A0 = 0 1
                                PORTK |= 0b00000010;
                                PORTD = volume;
                                PORTL = 0;
                                PORTK &= 0b11110111;        // Bring CS Low
                                PORTK |= 0b00001000;        // Bring CS High

                                PORTK |= 0b00100000;        // Bring LDAC High
                                PORTK &= 0b11011111;        // Bring LDAC Low
                                sei();
                        }

                }
        }
        else
        {
                volDownPressed = volMax;
        }

        if ((PINJ & 0b00010000) == 0b00010000)
        {

                if (volUpPressed-- == 0)
                {
                        volUpPressed = volMax;
```

```
				if (volume < 132)
				{
					volume++;
					cli();
					// Set Initial Volume
					PORTK &= 0b11111011;	// A1 A0 = 0 1
					PORTK |= 0b00000010;
					PORTD = volume;
					PORTL = 0;
					PORTK &= 0b11110111;	// Bring CS Low
					PORTK |= 0b00001000;	// Bring CS High

					PORTK |= 0b00100000;	// Bring LDAC High
					PORTK &= 0b11011111;	// Bring LDAC Low
					sei();
				}

			}
		}
		else
		{
			volUpPressed = volMax;
		}


		if ((PINJ & 0b01000000) == 0b01000000)
		{
			if (volMutePressed != -1) volMutePressed--;

			if (volMutePressed == 0)
			{
				volMutePressed = -1;
				if (muted == 0)
				{
					muted = 1;
					cli();
					// Set Initial Volume
					PORTK &= 0b11111011;	// A1 A0 = 0 1
					PORTK |= 0b00000010;
					PORTD = 0;
					PORTL = 0;
					PORTK &= 0b11110111;	// Bring CS Low
					PORTK |= 0b00001000;	// Bring CS High

					PORTK |= 0b00100000;	// Bring LDAC High
					PORTK &= 0b11011111;	// Bring LDAC Low
					sei();
				}
				else
				{
					muted = 0;
					cli();
					// Set Initial Volume
```

```
                                PORTK &= 0b11111011;       // A1 A0 = 0 1
                                PORTK |= 0b00000010;
                                PORTD = volume;
                                PORTL = 0;
                                PORTK &= 0b11110111;       // Bring CS Low
                                PORTK |= 0b00001000;       // Bring CS High

                                PORTK |= 0b00100000;       // Bring LDAC High
                                PORTK &= 0b11011111;       // Bring LDAC Low
                                sei();
                        }

                }
        }
        else
        {
                volMutePressed = volMax;
        }
}
```