Project Abstract

Our project is a wireless headphone that utilizes a transmitter and receiver to wirelessly transmit audio signals to the user's headphones. Our system will have a transmitter unit on the audio source, and a portable, battery powered unit for the receiver where any 3.5 mm headphones may be plugged in. Wireless headphones will be useful for users who would like the freedom of moving around a room while listening to a TV or stationary mp3 player.
# Table of Contents

- Project Features and Objectives .................................................................................. 3
- Cost Objective ............................................................................................................. 4
- Competing Product .................................................................................................... 5
- Concept and Technology Selection ............................................................................ 6
- Flowcharts and Diagrams ............................................................................................ 7
- Parts List ..................................................................................................................... 8
- Separation of Work ..................................................................................................... 9
- Gannt Chart ................................................................................................................ 10
- Appendix A: Schematics ............................................................................................. 11
- Appendix B: PCBs ....................................................................................................... 14
- Appendix C: Project Code ......................................................................................... 16
- Appendix D: Equalizer Simulations .......................................................................... 28
Project Features and Objectives

The main objective for this project is to wirelessly and continuously transmit an audio signal to a pair of headphones.

- Our first objective is to establish a wireless link between our transmitter and receiver, and accurately send and receive an accurate digital data stream at at least 1.41 Mbps (uncompressed CD quality), without interference.
- We will use external audio A/D and D/A converters for the analog-digital conversion.
- Another design goal would be for the receiver to be small as well as low-power, using AA batteries.
- Estimated battery life of 200 hours
- Range of 40 yards LoS

There will be several technical challenges. We would need our receiver unit to have an efficient low-power design for longer battery life and a better user experience. Another challenge would be in getting the microcontroller to process in the audio data outputted from a source and send the data in a compatible form to a transmitter at an acceptable data rate.

The outcome is to have any headphones plugged into the receiver be able to clearly receive audio from the source of the transmitter, i.e. a iPod. The receiver will also include user input to change equalization and volume.
Cost Objective

We’re aiming to have the price of our wireless headphones be under $250. There are a few similar existing products available on the market that are rather expensive, so making the product with a lower price and better value is a goal that we have in mind. Unlike the existing products, which have everything built and packaged into a set of headphones, our product will be a packaged device in which the user can plug in any pair of preexisting headphones they own and start listening. Also, we are aiming to use a display for user input, where the user can change volume and equalizer settings.

Similar products sold by Sony and Sennheiser currently cost between $120 and $300.
Competing Product, Sennheiser RS 170 Headphones, MSRP $279.95
Concept and Technology Selection

Most of the effort went into selecting the proper transceivers that would ensure accurate transmission of audio (stereo) data. In order for to do this, several factors need to be considered. The transceivers will need to be able to continuously transmit an audio stream to be received at a receiver end. The RF transceivers we are considering operate at 2.4GHz. In order to transmit uncompressed CD quality audio, 16 bits resolution at 44.1kHz sampling frequency is needed. This means that a 1.41Mbps rate is needed at a minimum for the specifications of the transceivers (not including overhead). Another factor is to have the transceiver consume as little power as possible, since the receiver will be used in a portable application. Also, to avoid over-complication of the wireless portion of this project, the transceivers are purchased as a wireless module that has the transceiver and antenna on a PCB already, so that RF PCB design will not be necessary. Among our considerations were the Nordic nRF24L01+, nRF24Z1 and various Zigbee transceivers.

Nordic transceivers nRF24Z1 were chosen, which operate at 1.536 Mbps for LPCM audio in 2.4 GHz RF band. It also has a Quality of Service engine (QoS) that ensures audio quality, i.e. retransmitting lost packets.

Other parts needed are ADC’s and DAC’s that can match the 44.1kHz sampling rate needed, 16 bit resolution, as well as having at least 2 channels to take in stereo input. Therefore, the WM8737 was chosen for the ADC, and the TLV320DAC23 was chosen for the DAC. The microcontroller chosen was an MSP430, which can control the transceivers and ADC/DAC. The digital portion functions off of 3.3V.

Analog components will be needed to create an equalizer circuit. We want to be able to adjust the bass and treble independently for an optimized sound for the user. Cutoff frequencies for the 3 bands would be 100 Hz, 1kHz, and 10kHz. The analog portion functions off of 6V.

A stereo headphone amplifier controls the volume. The headphone amplifier would need to be matched impedance-wise to a pair of headphones. There are 3.5mm headphone jacks both on the output of the receiver unit and the input of the transmitter unit.
Flowcharts and Diagrams

Block diagram
Parts List
- 2x Nordic nRF24Z1 module
- 2x TI MSP430
- 1x WM8717 ADC
- 1x TLV320DAC23 DAC
- 2x TLV2450
- 2x LM386

Estimated build cost: ~$90
Separation of Work

Eli
- Interface components together (ADC to transmitter, receiver to microcontroller to DAC)
- Establishing a consistent wireless link between the two tranceivers
- Microcontroller coding (communication with peripherals)

Stephen
- Equalizer circuit stage
- Headphone Amplifier and output stage
- Packaging

Both
- Voltage Regulator
- PCB design and soldering
Gantt Chart
Appendix A: Schematics

Transmitter Schematic

Receiver Schematic
EQ Schematic
Headphone Amplifier Schematic

Voltage Regulator Schematic
Appendix B: PCBs

Transmitter PCB

Receiver PCB
Equalizer PCB

Headphone Amplifier PCB

Voltage Regulator PCB
Appendix C: Project Code

Flowchart

Transmitter:
main.c (for tx)
#include "main.h"

unsigned char State;
char slaveoutbuf[SLAVEBUFSIZE]; // Global data buffer from MCU
to Z1 slave
char slaveinbuf[SLAVEBUFSIZE]; // Global data buffer
from Z1 slave to MCU

void main(void)
{
    volatile unsigned int i;

    init_Timer();
    init_Port();
    init_SPI();

    nRF24Z1_SlaveDisable();

    wm8737_SlaveEnable();
    wm8737_SlaveDisable(); // P1OUT |= BIT6; // Reset slave
    _delay_cycles(d10ms);
wm8737_init();
__delay_cycles(d10ms);

nRF24Z1_SlaveEnable()://P1OUT &= ~BIT5; // Out=0. Now with SPI signals initialized,
nRF24Z1_SlaveDisable()://P1OUT |= BIT5; // Out=1. reset slave

__delay_cycles(d200ms); // Wait for slave to initialize, 200ms

nRF24Z1_InitRFCHRegisters(); //Set up address, AFH, and Frequency Hopping Table
Init_ATXRegs(); //TX Initialization
Init_ARXRegs(); //RX Initialization
nRF24Z1_Write(RXMOD, RXRFEN); // Enable RF on RX
nRF24Z1_Write(TXMOD, MCLK12288 | INTFI2S | TXRFEN); //Enable RF on TX
nRF24Z1_ForceRelink();

while(1)
{
    if(State == POWER_ON_RESET)
    {
        //nRF24Z1_InitSlaveInterface();
        // Delay 200ms
        __delay_cycles(d200ms);

        State = INIT_ATX;
    }
    else if(State == INIT_ATX)
    {
        //Configure to establish link with RX
        Init_ATXRegs();
        Init_ARXRegs();
        nRF24Z1_InitRFCHRegisters();
        nRF24Z1_ForceRelink();

        State = ESTABLISH_LINK;
    }
    else if(State == ESTABLISH_LINK)
    {
        //LED_STATUS = BLINK_LED1;
        //LED_STATUS = LED_OFF;

        if(nRF24Z1_HasLink())
        {
            // Send control data to the ARX through control channel

            nRF24Z1_ToggleTXCSTATE();

            while(nRF24Z1_Read(TXCSTATE) == 0x01)
            {
                Blink_LED1();
            }

            nRF24Z1_ToggleRXCSTATE();

            while(nRF24Z1_Read(RXCSTATE) == 0x01)
            {
        }
Blink_LED1();
}

nRF24Z1_ToggleLNKCSTATE();

while(nRF24Z1_Read(LNKSTATE) == 0x01)
{
    Blink_LED1();
}

State = LINK_ACTIVE;
}
else
{
    State = ESTABLISH_LINK;
}
else if(State == LINK_ACTIVE)
{
    LED_STATUS = LED2;
    //StatusLED2_State = LED_BLINK;
    if(!nRF24Z1_HasLink())
    {
        State = ESTABLISH_LINK;
    }
}
    nRF24Z1_HasLink();
}

Main.c(for rx)
#include "main.h"

void main(void)
{
    volatile unsigned int i;
    //unsigned test;

    init_Timer(); //Stop WDT, 8MHz
    init_Port(); // Set up port pins
    init_SPI(); //Initialize SPI, run at 1MHz

    nRF24Z1_SlaveDisable(); //make sure Z1 turned off so SPI bus used only by TLV

    tlv320_SlaveEnable();
    tlv320_SlaveDisable(); //P1OUT |= BIT6; //Reset slave

    __delay_cycles(d10ms);
tlv320_init(); //Initialize TLV
__delay_cycles(d10ms);

nRF24Z1_SlaveEnable(); // Out=0. Now with SPI signals initialized.
nRF24Z1_SlaveDisable(); // Out=1. reset slave

__delay_cycles(d200ms); // Wait for slave to initialize, 200ms

nRF24Z1_InitRFCHRegisters(); // Set up address, AFH (adaptive frequency hopping), and Frequency Hopping Table
Init_ATXRegs(); // TX Initialization
Init_ARXRegs(); // RX Initialization
nRF24Z1_Write(RXMOD, RXRFEN); // Enable RF on RX
nRF24Z1_Write(TXMOD, MCLK12288 | INTFI2S | TXRFEN); // Enable RF on TX

nRF24Z1_ForceRelink(); // force a re-link

void nRF24Z1_ForceRelink(void)
{
  // Force re-link
  unsigned char reg = nRF24Z1_Read(LNKMOD);
nRF24Z1_Write(LNKMOD, reg | FORCE_CONF);
}

void nRF24Z1_SetAddress(unsigned char a0, unsigned char a1, unsigned char a2, unsigned char a3, unsigned char a4)
{
  nRF24Z1_Write(ADDR0, a0);
nRF24Z1_Write(ADDR1, a1);
nRF24Z1_Write(ADDR2, a2);
nRF24Z1_Write(ADDR3, a3);
nRF24Z1_Write(ADDR4, a4);
}

void Init_ATXRegs(void)
{
  // Enable RF, MCLK = 12.288MHz, I2S, enable DD input
  // nRF24Z1_Write(TXMOD, MCLK12288 | INTFI2S | TXRFEN | DDEN);
  // Normal I2S master, 1 clock before data with 16bits
}
nRF24Z1_Write(I2SCNF_IN, I2SMASTER);

// Set audio rate to 48k (scale of 1)
nRF24Z1_Write(TXSTA, TXRATE_USER | TXRATE_48 | SCALE_ONE);

// Digital audio 16-bit PCM
nRF24Z1_Write(TXFMT, PCM16);

// Full TX power
nRF24Z1_Write(TXPWR, N0DBM);

// High transmit latency
nRF24Z1_Write(TXLAT, LAT_HIGH);

// No reset
nRF24Z1_Write(TXRESO, NORESO);

// No sleep timer
nRF24Z1_Write(TXLTI, 0x00);

// Development purposes, set to master since no ADC
//nRF24Z1_Write(I2SCNF_IN, I2SMASTER);
}

void Init_ARXRegs(void)
{
    // Normal I2S master
    nRF24Z1_Write(I2SCNF_OUT, 0x00);

    // Full power
    nRF24Z1_Write(RXPWR, NODBM);

    // No master interface on ARX
    nRF24Z1_Write(RXDCMD, ARX_NO_INTF);

    // No reset
    nRF24Z1_Write(RXRESO, NORESO);

    // No sleep timer
    nRF24Z1_Write(RXLTI, 0x00);

    // ARX with Serial slave interface
    nRF24Z1_Write(RXSTA, (0x00 <<6));

    // Enable RF on ARX
    // nRF24Z1_Write(RXMOD, RXRFEN);
}

void nRF24Z1_InitRFCHRegisters(void)
{
    // Set address
    nRF24Z1_SetAddress(0x85, 0x34, 0xA2, 0x38, 0x92);

    // Setup AFH
nRF24Z1_Write(NBCH, 16);
nRF24Z1_Write(NACH, 26);
nRF24Z1_Write(NLCH, 26);
nRF24Z1_Write(LNKETH, 255);
nRF24Z1_Write(LNKWTH, 255);

// Setup Frequency Hopping table:
nRF24Z1_Write(CH0, 0x06);
nRF24Z1_Write(CH1, 0x1C);
nRF24Z1_Write(CH2, 0x34);
nRF24Z1_Write(CH3, 0x4C);
nRF24Z1_Write(CH4, 0x18);
nRF24Z1_Write(CH5, 0x30);
nRF24Z1_Write(CH6, 0x48);
nRF24Z1_Write(CH7, 0x14);
nRF24Z1_Write(CH8, 0x2C);
nRF24Z1_Write(CH9, 0x44);
nRF24Z1_Write(CH10, 0x10);
nRF24Z1_Write(CH11, 0x28);
nRF24Z1_Write(CH12, 0x40);
nRF24Z1_Write(CH13, 0x0C);
nRF24Z1_Write(CH14, 0x24);
nRF24Z1_Write(CH15, 0x3C);
nRF24Z1_Write(CH16, 0x08);
nRF24Z1_Write(CH17, 0x20);
nRF24Z1_Write(CH18, 0x38);
nRF24Z1_Write(CH19, 0x04);
nRF24Z1_Write(CH20, 0x1E);
nRF24Z1_Write(CH21, 0x36);
nRF24Z1_Write(CH22, 0x4E);
nRF24Z1_Write(CH23, 0x1A);
nRF24Z1_Write(CH24, 0x32);
nRF24Z1_Write(CH25, 0x4A);
nRF24Z1_Write(CH26, 0x16);
nRF24Z1_Write(CH27, 0x2E);
nRF24Z1_Write(CH28, 0x46);
nRF24Z1_Write(CH29, 0x12);
nRF24Z1_Write(CH30, 0x2A);
nRF24Z1_Write(CH31, 0x42);
nRF24Z1_Write(CH32, 0x0E);
nRF24Z1_Write(CH33, 0x26);
nRF24Z1_Write(CH34, 0x3E);
nRF24Z1_Write(CH35, 0x0A);
nRF24Z1_Write(CH36, 0x22);
nRF24Z1_Write(CH37, 0x3A);
}

void nRF24Z1_Write(unsigned char addr, unsigned char data)
{
    nRF24Z1_SlaveEnable();
    _delay_cycles(d500); // 500us delay
    send_byte(addr&0x7F);
    _delay_cycles(d500);
send_byte(data);
_delay_cycles(d500);

nRF24Z1_SlaveDisable();
_delay_cycles(d500);
}

unsigned char nRF24Z1_Read(unsigned char addr)
{
    unsigned char value;
    nRF24Z1_SlaveEnable();
    _delay_cycles(d500); // 500us delay
    send_byte(addr|0x80);
    _delay_cycles(d500); // 500us delay
    value = send_byte(0x00);
    _delay_cycles(d500); // 500us delay
    nRF24Z1_SlaveDisable();
    _delay_cycles(d500); // 500us delay
    return value;
}

unsigned char nRF24Z1_HasLink(void)
{
    char cnt = 4;
    while(cnt-- != 0)
    {
        // Check for link multiple times to prevent false-positive
        // Nordic uses this method in their examples
        if((unsigned char)(nRF24Z1_Read(LNKSTA) & 0x01) != 0x01)
        {
            return 0;
        }
        _delay_cycles(d500); // 500us delay
    }
    return 1;
}

**TLV320.c**
#include "TLV320.h"

void tlv320_Write(uint8_t addr, uint16_t data)
{
    tlv320_SlaveEnable();
    send_byte(((addr&0x7F)<<1) | ((data & 0x100) ? 0x01:0x00));
    send_byte((uint8_t)data);
    _delay_cycles(d10);
    tlv320_SlaveDisable();
}
void tlv320_init(void) {
    tlv320_Write(TLV320_RESET, TLV320_RESET_TRIGGER);

    /*
    tlv320_Write(TLV320_LEFT_INPUT,
        TLV320_LRS_UPDATE_DISABLE |
        TLV320_L_IN_MUTE);
    tlv320_Write(TLV320_RIGHT_INPUT,
        TLV320_RLS_UPDATE_DISABLE |
        TLV320_R_IN_MUTE); //leave at default values LRS disabled, LIM muted
    */

    //Power down
    tlv320_Write(TLV320_POWER_DOWN,
        TLV320_POWER_ON |
        TLV320_CLOCK_OFF | //ON |
        TLV320_OSC_OFF | //ON |
        TLV320_OUTPUTS_ON |
        TLV320_DAC_ON |
        TLV320_LINE_INPUT_OFF);

    //Set sampling rate
    tlv320_Write(TLV320_ANA_AUDIO_PATH,
        TLV320_DAC_SEL_ON |
        TLV320_BYPASS_DISABLED);

    tlv320_Write(TLV320_DIG_AUDIO_PATH,
        TLV320_DAC_MUTE_DISABLED | //TLV320_DEEMP_DISABLED);
        TLV320_DEEMP_48KHZ);

    tlv320_Write(TLV320_SAMPLE_RATE,
        TLV320_CLKOUT_DIV1 |
        TLV320_CLKin_DIV1 |
        //TLV320_SAMPLING_RATE_44KHZ |
        TLV320_SAMPLING_RATE_48KHZ |
        TLV320_BOSR_256FS |
        TLV320_CLK_MODE_NORMAL); //with MCLK = 12.288MHz, 48KHZ, 0000b,
        BOSR = 0, CLKin=0 FILTER TYPE 1
    //I2S master
    tlv320_Write(TLV320_AUDIO_FORMAT,
        TLV320_MS_SLAVE |
        TLV320_LR_SWAP_DISABLED |
        TLV320_LR_PHASE_LRCIN_LOW |
        TLV320_INPUT_WORD_32BIT |
        TLV320_DATA_FORMAT_I2S);

    tlv320_Write(TLV320_DIG_INTF,
        TLV320_INTERFACE_ACTIVE);

    //Headphone
    tlv320_Write(TLV320_LEFT_HP_VOL,
        TLV320_LRS_HP_UPDATE_DISABLE |
        TLV320_L_ZERO_CROSS_OFF |
        0x60); //TLV320_L_HP_VOL_0dB);

    tlv320_Write(TLV320_RIGHT_HP_VOL,
        TLV320_RLS_HP_UPDATE_ENABLE |
        TLV320_R_ZERO_CROSS_OFF |
        0x60); //TLV320_R_HP_VOL_0dB);
}

24
void init_Timer()
    {
        WDTCTL = WDTPW + WDTHOLD;                   // Stop watchdog timer
        if (CALBC1_16MHZ == 0xFF || CALDCO_16MHZ == 0xFF)
        {
            while(1);                               // If calibration constants erased
            // do not load, trap CPU!!
        }
        BCSCTL1 = CALBC1_8MHZ;                     // Set DCO to 8MHz, use MCLK
        DCOCTL  = CALDCO_8MHZ;
    }

void init_SPI()
    {
        UCA0CTL1 |= UCSWRTST;
        UCA0CTL0 |= UCCKPH + UCMSB + UCMST + UCSYNC; // 3-pin, 8-bit SPI master, CPOL = 0, CPHA = 0
        UCA0BR0 |= 0x08;                          // /8 . Clock prescaler setting of the Baud rate generator.
        UCA0BR1 = 0;                              // The 16-bit value of (UCAxBR0 + UCAxBR1 × 256) forms the
        UCA0MCTL = 0;                             // No modulation
        UCA0CTL1 &= ~UCSWRTST;                     // **Initialize USCI state machine**
        IE2 |= UCA0RXIE;                          // Enable USCI0 RX interrupt
    }

void init_Port()
    {
        P1OUT = 0x00;                             // P1 setup for LED & reset output
        P1DIR |= BIT0 + BIT5 + BIT6;              // LED1 + RST(SSN) + LED2
        P1SEL = BIT1 + BIT2 + BIT4;              //Select pins for USI SPI
        P1SEL2 = BIT1 + BIT2 + BIT4;
    }

//Send and receive a single byte through the SPI interface
unsigned char send_byte(unsigned char txWord)
    {
        unsigned char in;
        UCA0TXBUF = txWord;                      //send byte to transmit buffer
        //_delay_cycles(10);
        while (!(IFG2 & UCA0TXIFG)); // USCI_A0 TX buffer ready? Loop until SPI transmit done
        //_delay_cycles(10);
        in = UCA0RXBUF;                          //byte received in buffer stored in variable
        _delay_cycles(d500);     //500us
        return in;                               //return byte received
    }

// Run the shift registers through MCU master SPI port to Z1 slave SPI port
// DOC: Interrupts must be off during this function!
void mcu_spicycle(char startadr, char endadr) {

}
extern char slaveinbuf[]; // Global data buffer from Z1
nen = 0;
if (endadr - startadr < SLAVEBUFSIZE) {
    // Assume that SPI interface can do
    spi_start(); // Turn on active low
    send_byte(startadr); // Send the start
    while (startadr++ <= endadr)
        slaveinbuf[n] = send_byte(slaveoutbuf[n++]); // Shift register action
    spi_end(); // Turn off active low
}

unsigned char SPI_TransmitByte(unsigned char byte)
{
    unsigned char in;
    UCA0TXBUF = byte;
    while (!(IFG2 & UCA0TXIFG)); // USCI_A0 TX buffer ready?
in = UCA0RXBUF;
return in;
}

WM8737.c
#include "wm8737.h"

void wm8737_Write(uint8_t addr, uint16_t data)
{
    wm8737_SlaveEnable();
send_byte(((addr&0x7F)<<1) | ((data & 0x100) ? 0x01:0x00) );
send_byte((uint8_t)data);
_delay_cycles(d10);
wm8737_SlaveDisable();
}

void wm8737_init(void)
{
    wm8737_Write(WM8737_RESET,0);
    wm8737_Write(WM8737_AUDIO_FORMAT,WM8737_AUDIO_FORMAT_FORMAT_I2S |
WM8737_AUDIO_FORMAT_WL_32_BITS |
WM8737_AUDIO_FORMAT_LRP_I2S_POL_NOT_INVERTED |
WM8737_AUDIO_FORMAT_MS_SLAVE |
WM8737_AUDIO_FORMAT_SDODIS_ENABLE);
wm8737_Write(WM8737_LEFT_PGA, //0xFF |
WM8737_LEFT_PGA_LINVOL_0_DB |
WM8737_LEFT_PGA_LVU_LATCH);
wm8737_Write(WM8737_RIGHT_PGA,
    //0xFF |
    WM8737_RIGHT_PGA_RINVOL_0_DB |
    WM8737_RIGHT_PGA_RVU_LATCH);

wm8737_Write(WM8737_AUDIO_PATH_L,
    WM8737_AUDIO_PATH_L_LINSEL_LINPUT1 |
    WM8737_AUDIO_PATH_L_LMICBOOST_13_DB |
    WM8737_AUDIO_PATH_L_LMBE_DISABLE |
    WM8737_AUDIO_PATH_L_LMZC_IMMEDIATELY |
    WM8737_AUDIO_PATH_L_LPZC_IMMEDIATELY |
    WM8737_AUDIO_PATH_L_LZCTO_256_DIV_FS);

wm8737_Write(WM8737_AUDIO_PATH_R,
    WM8737_AUDIO_PATH_R_RINSEL_RINPUT1 |
    WM8737_AUDIO_PATH_R_RMICBOOST_13_DB |
    WM8737_AUDIO_PATH_R_RMBE_DISABLE |
    WM8737_AUDIO_PATH_R_RMZC_IMMEDIATELY |
    WM8737_AUDIO_PATH_R_RPZC_IMMEDIATELY |
    WM8737_AUDIO_PATH_R_RZCTO_256_DIV_FS);

wm8737_Write(WM8737_POWER_MGMT,
    WM8737_POWER_MGMT_VMID_ON |
    WM8737_POWER_MGMT_VREF_ON |
    WM8737_POWER_MGMT_AI_ON |
    WM8737_POWER_MGMT_PGL_ON |
    WM8737_POWER_MGMT_PGR_ON |
    WM8737_POWER_MGMT_ADL_ON |
    WM8737_POWER_MGMT_ADR_ON |
    WM8737_POWER_MGMT_MICBIAS_OFF);

wm8737_Write(WM8737_ALC1,
    WM8737_ALC1_ALCSEL_STEREO |
    WM8737_ALC1_MAXGAIN_POS_30_DB |
    WM8737_ALC1_NEG_18_DB);

wm8737_Write(WM8737_ALC2,
    WM8737_ALC2_HLD_0_MS |
    WM8737_ALC2_ALCZCE_DISABLE);

wm8737_Write(WM8737_ALC3,
    WM8737_ALC3_ATK_8_MS |
    WM8737_ALC3_DCY_33_MS);

/*wm8737_Write(WM8737_NOISE_GATE,
    WM8737_NOISE_GATE_NGAT_ENABLE |
    WM8737_NOISE_GATE_NGTH_NEG_60_DB);*/
Appendix D: LT SPICE Simulations

Equalizer Circuit

Equalizer - High Pass Filter at Full Gain
Equalizer - High Pass Filter at Full Attenuation

Equalizer - Band Pass Filter at Full Gain
Equalizer - Band Pass Filter at Full Attenuation

Equalizer - Low Pass Filter at Full Gain
Equalizer - Low Pass Filter at Full Attenuation

Equalizer - All Filters in middle setting
Headphone Amplifier Circuit and Simulation