The 551 Student's Portable Theater System

David Breece, III
Aaron Krol
Lincoln Westfall

{dcb2, akrol, westfall}@andrew.cmu.edu

18-551 Digital Communications and Signal Processing Design
Spring 2002

Table of Contents

Introduction

In recent years, DVDs have increased in popularity as a movie storage medium for home viewing. As a result, a growing number of people are purchasing home theater systems that take advantage of the increased sound capabilities that DVD offers over the traditional VHS format. The DVD format most commonly uses Dolby Digital encoding, also known as AC-3, to store five discrete channels and one low frequency effects channel (often referred to as 5.1 channels). The five discrete channels are positional and are sent by a Dolby Digital receiver to speakers placed at different points in the home theater (front left, center, front right, surround right, and surround left).

The popularity of DVDs has also become strong in the laptop computer market, where users often use their laptops to watch movies while traveling. In addition, more manufacturers are beginning to produce and market portable DVD players. The increased competition has driven the price down to a point within reach of many consumers. Home theater systems are not what one would consider portable, however, leaving traveling movie buffs using headphones without the positional audio they've become used to hearing at home. Furthermore, even technologies that attempt to bring positional audio to this market often suffer from the fact that headphone audio images tend to appear as if they were inside the listener's head, rather than around it. We aim to provide a better (and cheaper) solution.

Significant research has been done in the area of human sound perception. Through digital signal processing, the 5.1 channels can be "down-mixed" using perceptual techniques to two channels and delivered directly to the ears via headphones in such a way that the listener perceives sounds to be coming from different places in the room. We will utilize research in Head-Related Transfer Functions to provide superior surround sound quality from a pair of headphones. Additional positional cues can be provided by delay and attenuation effects based on the location of a simulated speaker. Such a pair of headphones - that can reproduce the entire spectrum of human hearing (20 Hz – 20 kHz) - can be purchased relatively inexpensively.

Background - Dolby Digital (AC-3)

The perceptual coding algorithm implemented by AC-3 is a data-compression scheme that seeks to eliminate the data we cannot hear while maintaining most or all of the information that we can. AC-3 divides the audio spectrum of each channel into narrow frequency bands that correlate closely to the frequency selectivity of human hearing. That allows coding noise to be very sharply filtered by taking advantage of the psychoacoustic phenomenon known as auditory masking. Coding noise stays close in frequency to the audio signal being coded, so it is effectively masked. AC-3 uses a "shared bit-pool" arrangement to use data as efficiently as possible. Bits are distributed among the various channels according to need. AC-3 allows multichannel surround sound to be encoded at a lower bit rate than required by just one channel on a CD.

Dolby Digital 5.1 Audio Bitstream Format

DVDs use Dolby Laboratories' AC-3 format as a coding standard for multi-channel digital surround sound. AC-3 uses perceptual coding technology for compression through differential encoding of the exponents and a local psychoacoustic model in the decoder. "An AC-3 audio bitstream is made up of a sequence of synchronization frames (see Figure 1). Each frame contains six coded audio blocks (AB), each of which represents 256 new audio samples, and each frame represents a constant time interval of 1536 audio samples. The frame size varies with the sample rate and coded data rate.

Figure 1. AC-3 Frame Format

Each frame starts with SI (sync information) and BSI (bit-stream information). The BSI has fields describes the sample rate, data rate, number of coded channels, and information on the coded audio service. There are two CRC words per frame, one at the beginning and one at the end, for error detection. An optional aux data field is located at the end of the frame for providing control or status information into the AC-3 bitstream for system-wide transmission.

The actual audio information consists of the quantized frequency coefficients in floating point form with an exponent and a mantissa. The mantissas are quantized with a variable number of bits, based on a psychoacoustic masking. More bits are allocated to perceptually important frequency bands. The parameters used for bit allocation are embedded in the bitstream for the decoder to reconstruct the coefficients. The audio block has information for reconstructing the frequency coefficients.

Figure 2. Audio Block Format

Each audio block represents 256 new audio samples per coded channel (see Figure 2). The audio block contains block switching flags, dither flags, dynamic-range compression information, coupling information, exponents, bit allocation information and the mantissas. The information in audio block 0 can be reused by other blocks in the same frame, allowing data sharing within a frame." [i]

Background - HRTFs

HRTFs (Head-Related Transfer Functions) are attempts to model the human perception of spatial sound. Not every aspect of this perception is well understood, but there are three main factors that are generally accepted as influencing our perception of 3D sound. One is the time delay between when a sound is heard in one ear and when it is heard in the other. The second factor is the sound intensity difference between ears.

And the third is the effect of the outer ear's shape as a filter on the incoming sound (due to multi-path reflections and diffraction). Sometimes, the reflective and refractive characteristics of the shoulders and upper body also play a role. HRTFs attempt to combine all these effects into a pair of filters, one for each ear of the listener. The transfer functions are acoustic filters that vary in not only frequency, but also in the azimuth, elevation, and range to the source[ii].

One problem that HRTFs address is that typically, headphones produce a sound image located inside the head. Ideally, one would like the image to be located at a specified spot somewhere outside the user's head. This gives a true "surround" feeling.

Not all HRTFs are the same. Some are based off of various mathematical equations and models, with a variety of approximations to simplify the work. Others are constructed from experimental data using dummy-heads with embedded microphones. Some HRTF sets are customized to a particular listener, while others are not (customized HRTFs tend to do slightly better at localization of sound sources, though non-customized ones do not fare too badly[iii]).

A significant choice in the implementation of any processing system is the choice of which HRTFs to use. One choice would be to use the HRTF measurements provided free of usage restrictions by the MIT Media Lab[iv]. We feel that the ready availability of this data, and the publication of refined[v] and real-time[vi] algorithms based off of this data makes it a good initial choice for our HRTF model.

Comparison Format - "Stereo" Processing

An output format that was not perceptually-encoded was needed for the purpose of comparison with the output format that was perceptually-encoded. The term "stereo" is a bit of a misnomer, however, for two reasons: 1) the perceptually-encoded output format is stereo (in that it is a two-channel format) and 2) the non-perceptually-encoded format is also based upon all 5.1 channels of information.

All 5.1 channels of information were used in creating the comparison output to remain consistent with the perceptually-encoded output. It was desired that any difference (improvement) in sound be caused by the perceptual encoding techniques, and not just by the additional information afforded by the extra channels. Each positional channel contributes a similar amount of information to what is heard by each ear in the "stereo" format as in the "Dolby Digital 5.1" format, only without the perceptual encoding techniques.

Essentially, when an off-axis channel is contributing to an ear (i.e., when the left front speaker is contributing to the right ear), a multiplier of 0.1 is used. This is based on inspection of the HRTFs and estimation of the energy difference received by the off-axis ear vs. the on-axis ear. The on-axis ear (the left ear in the above case) receives the full signal.

Previous Work

In the realm of processing 3-D positional sound, a significant emphasis has been from the perspective of 3-D gaming, and the desire of game developers to produce immersive 3-D environments. The most significant work in this area that we've seen is the Open Audio Library, or OpenAL[vii], which is a "joint effort to create an open, vendor-neutral, cross-platform API for interactive, primarily spatialized audio." Created by Creative Labs and Loki Entertainment Software and targeted at game audio, OpenAL provides an easy-to-use API that allows developers to position audio sources around a listener and get reasonable 3-D effects. Creative Labs has also created an extension to the API that allows it to read in an AC-3 soundtrack as the game's background music. We differentiate ourselves from the OpenAL platform in our targeting of headphone movie listeners and our use of Head-Related Transfer Functions to model the alteration of the sound at each ear caused by the listener's body.

As for previous 551 projects, there is one related project of which we are aware. This is the Group 2 (Matt Juhasz, Vivek Krishna, and Julie Peoples) project from Spring 2001, which dealt with the use of HRTFs in positioning a single audio source to create a virtual audio environment. Our purpose and situation is different, in that we will have multiple positioned sources (the 5.1 Dolby Digital channels) which themselves are used to create a surround effect. This surround effect is then enhanced by the HRTFs.

SRS Technologies also has a technology that attempts to encode 6 channel-surround into 2 channels, and even has a demo one can stream off their website. The technology supposedly will give some benefits even to standard stereo users without their special decoder. In listening to the demo, there were some left-right effects, but front-back detail seemed to be lacking. Some of the HRTF literature discusses modeling this very thing to provide the more realistic environment that we hope to implement. (It's unclear whether SRS is using any HRTFs in this "Circle Surround" system, though they have used them in some earlier stereo applications)

Sony also offers headphones that provide Dolby surround using headphones. The headphones sell for around $499.99, and as a consequence we have been unable to test how well these devices work.

Dolby also has their own specification, appropriately termed "Dolby Headphone," althought little is known about this scheme. Although mentioned on their website, we were unable to find it implemented in any product nor discussion of its processing methods.

Obtaining the Data

Dolby Digital 5.1 Audio Channels

Although we would have liked to obtain a Dolby license and implemented a complete solution for playing DVD movies on the PC's DVD-ROM drive and process the AC3 audio stream in real-time, we decided to base our positionality processing work on the assumption that we have the six Dolby Digital 5.1 audio channels in separate PCM-WAV files. To make this assumption a reality for our work, we ripped the DVD audio channels to the hard drive and converted them to individual workable PCM data files. The PCM data also requires more PCI bandwidth for transfers than would the AC-3 data but allows us to implement a simple and legal mixing algorithm on the six audio channels.

In order to transfer the DVD audio channels to individual PCM files we could work with, several steps and a variety of audio tools were required. We chose to use the Dolby Digital Demo DVD we ordered from Dolby as our source, since it had a number of mini-movies roughly thirty seconds in length that demonstrate the positional capabilities of Dolby Digital 5.1 rather well. We chose the "City Demo" to work with the most because we felt it had the most positional elements of all the demos.

We ripped the "City Demo" VOB file to the hard disk using SmartRipper v2.41. Next we opened the "City Demo" VOB file in VOBrator Beta Release 0.2 and demuxed it into an MP2 video file and an AC3 5.1 channel audio file using VOBrator. Then we opened the AC3 file in Sonic Foundry SoftEncode v1.0 using its "Dolby Digital (decode to PCM) (*.ac3)" file open option. It turns out that at this point SoftEncode saves six files called AC3xx.TMP (where xx is a hex number) in the current Windows TEMP directory that are actually WAV-PCM encoded files of the six separate Dolby Digital 5.1 channels. We just had to find the six latest AC3xx.TMP files (with the highest xx digits; they are numbered following the order L, C, R, LS, RS, LFE channels) and rename their names to the audio channel and the extensions to WAV, and we had the six DD5.1 channels of "City Demo" in six separate PCM-WAV files!

Now, these WAV files all had a sampling rate of 48000 kHz, but our HRTF WAV files from the MIT Kemar project were all recorded at 44.1 kHz. For the project, both were needed to have the same sampling rate in order to do proper processing between the HRTFs and the sound channels. FFTs of the "City Demo" WAV files showed that they really had no audio information above 20 kHz, so we decided to downsample the six channels to 44.1 kHz. We opened the six files in the audio editor GoldWave v4.12, resampled them at 44.1 kHz, and resaved them. GoldWave resamples using simple linear interpolation[viii], which may have introduced a tiny bit of distortion, but nothing significant enough to be audible. For more lossless DVD audio processing in the future we would hope to get 48 kHz HRTF samples a different source, which would avoid the need to downsample the audio channels for processing.

Since the six channel's WAV files were about 3 MB apiece and thus nonideal for all our preliminary work on them, we decided put them into a more workable size. So we trimmed the WAV files to the same exact 200,000 samples of an action-packed moment in the middle of the demo that contained data on all six channels. These files were 400044 Bytes each - 16 bits per sample, plus the standard 44 Byte WAV header - which was a much more practical size to work with until we were ready for final testing and demonstration. Now the focus shifted to the actual processing of the sound channels.

The HRTFs

We obtained a copy of the HRTF database from the MIT Media Laboratory. The database was stored conveniently in the WAV format, as a volunteer had converted it from the original raw data format that MIT had first released. This saved us a significant amount of work, since the raw data format was in big-endian byte order. While the EVM can work with big-endian byte order through a setting in Code Composer, both the WAV file format and the Intel platform use a little-endian byte order. This would have significantly complicated our implementation.

Implementation

Our implementation is comprised of three parts: the PC back end, the GUI, and the EVM processing. In this section, we describe the implementation of each major component.

The PC Back End

Our PC Back End component is itself composed of three major parts:

1. GUI Integration
2. WAV File I/O
3. PC ? EVM Communication

The implementation of each is described below. See also: *pc_comm.c, arrays.h*, *read.h, read.c*.

GUI Integration

Since the graphical user interface is implemented via a Web-based interface, the GUI is located in a separate portion of code from the remainder of the project. The GUI and the GUI Integration code communicate using a shared parameters file that the GUI writes to and the PC reads from. The two parts utilize a locking mechanism to prevent the Integration and GUI programs from attempting to access the file at the same time. This prevents potential parameter-file corruption issues that could arise if both were to access the file simultaneously.

The GUI Integration code then compares the new set of parameters with the previously stored parameters, notes any changes, and stores the new parameters. A status-change word is returned to the PC ? EVM Communication code indicating which parameters have changed and will need to be re-transferred. See *pc_comm.c*.

WAV I/O

Our WAV code was written from scratch after we failed to find a free implementation that would fit our needs. We don't doubt that there is suitable code out there somewhere, but a reasonable search of the Web failed to uncover it. For this reason, we referred to the readily available WAV file specification. The WAV file format is fairly simple, consisting of a standard "RIFF" header, followed by an arbitrary number of metadata or data "chunks." The two most relevant chunks are the format and data chunks, which contain all the necessary formatting information (sampling rate, number of channels, etc) and the data itself. All other chunks are optional under the specification. Our read code ignores all nonessential chunks, and our write code does as well. See *read.h, read.c, arrays.h.*

PC ? EVM Communication

Our overall scheme for PC ? EVM Communication was based upon requests sent by the EVM. Periodically the EVM queries the PC using one of several different commands, asking for information such as parameter changes or new chunks of input data.

There were a couple of different techniques available for transferring data between the PC and the EVM: synchronous or asynchronous transfers. Our implementation used synchronous transfers. This approach had two major benefits. The first was that it enabled us to use the utility functions from lab3, thus simplifying development and reducing debugging time. The second benefit was that we avoided the transfer stability problems of some I/O functions (see the lab 3 handout). This approach also meant that in the event we wanted to switch to asynchronous transfers, we could do so later by rewriting the utility functions rather than the whole PC codebase. The downside to using synchronous transfers is that the processor is not free to perform other operations while the transfer is happening. This could have yielded a potential boost in speed if implemented.

We made heavy use of the utility functions and their associated transfer structure (which used mailboxes 1, 2, and 3 for transferring data, its size, and an associated command, respectively). The variety of data that needed to be transferred between the PC and EVM (12 different HRTF buffers, 6 input data buffers, the output, parameter change indicators, etc.) necessitated over 30 different commands.

The GUI

The goal of the GUI was to give users an option to easily choose the virtual audio environment to be simulated. We included a variety of options to the user so they could create their ideal virtual home theater. Figure 3 below shows a screenshot of the GUI using the default options.

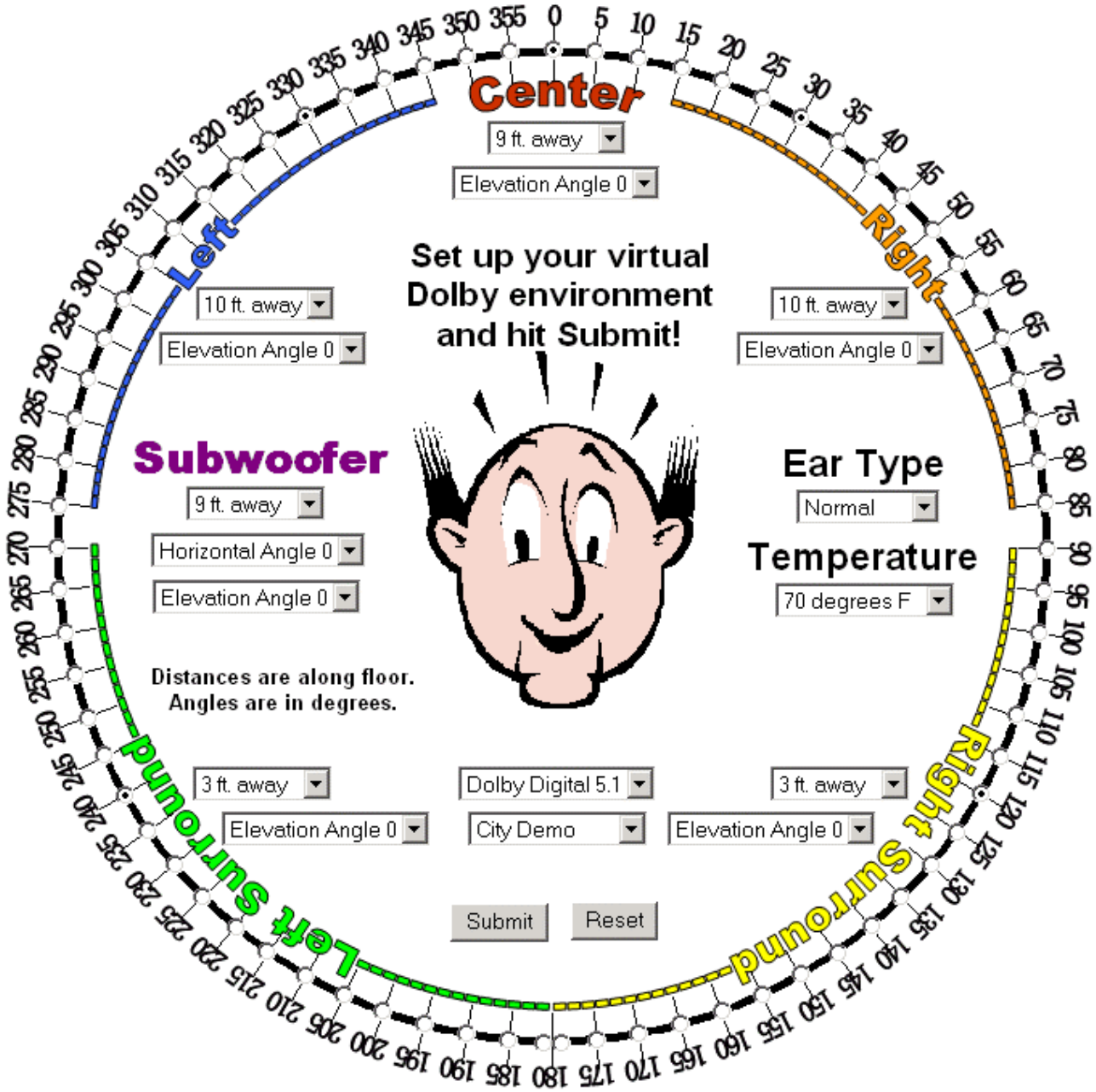


Figure 3. Screenshot of GUI with default options selected.

The GUI was created as an HTML Web page (default.htm) that sends the user's selections as form data to an ASP file (formproc.asp) written in Visual Basic Script (VBScript). The VBScript code in formproc.asp uses the selected environment settings to write all necessary processing parameters to a file. The PC back-end then reads this parameter file. This setup makes future real-time processing possible, even if the user changes the environment settings while audio is being played and processed at the moment. See *default.htm, formproc.asp*. See also example *parameters.txt* and its explanatory key *paramkey.txt* in Appendix C.

For audio environment settings selectable in the GUI, we decided to include the horizontal angle (azimuth), elevation angle, and floor distance of each speaker, as well as temperature of the room being simulated. The azimuth and elevation values determine which HRTFs are used to process the audio channels, and these angles and floor distances, along with the room's temperature, are used to add appropriate delay and attenuation to the individual channels. We included environment options for the Subwoofer, since most of the contribution of the LFE channel is usually output through the subwoofer. The Subwoofer options don't actually change the positionality of the sound as bass typically radiates omni-directionally from the speaker driver. In a typical home theather setup, because of this and interactions with the room, it is very difficult to deteremine the position of a low-frequency sound source. We built in the option for future expansion, however.

Another option is the ear type, normal or large red. The size and density of one's ear auricle (or pinna) affects the ear's frequency response (i.e. the HRTFs). There are two sets of HRTFs based on different ear types, and this option determines which is used. Different people will get a better positional effect by choosing one ear type or the other.

The last two options selectable in the GUI are which Demo to use and whether the output should be a Dolby Digital 5.1 emulation (using our processing) or a simple stereo downmix. We had four different demos available: Canyon, City, Egypt and Train. The simple stereo downmix, if selected, is performed by the EVM without any processing other than channel weighting and direction to the headphones' two speakers. This can be used to generate a comparison for testing the Dolby (HRTF-based) processing.

Units for parameters selectable in the GUI are all US-English (feet, Fahrenheit), but our GUI Integration code converts them to metric units (meters, Celsius) for better integration with the reference data we had available.

In order to get this implementation of the GUI to work, we had to set up Personal Web Server 4.0 (installed through the NT Option Pack) on our lab station. Only with such a web server set up could the VBScript in our ASP files have any permission to write files to the hard drive, which is necessary to integrate the GUI into the PC back-end of our EVM code.

EVM Processing

The data from the 5.1 channels goes through a series of steps after arriving on the EVM before it is finally returned to the computer as 2 stereo channels.

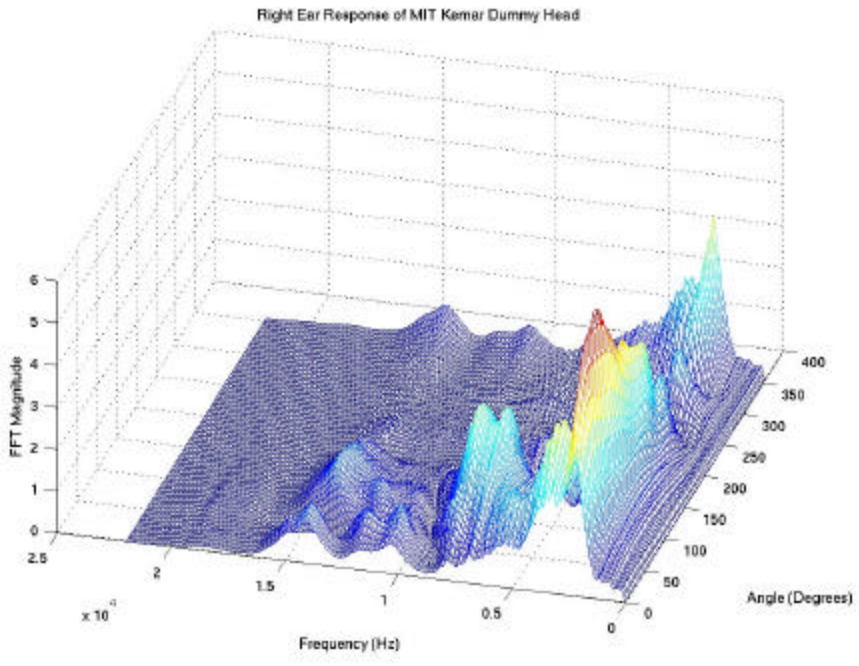
1) The 5.1 channels of data is transferred to the EVM via the PCI bus using a mailbox transfer (as in lab 3). The channels are represented in the time domain. They are stored off-chip in storage buffers until they are needed for processing.
2) The HRTF to be used is transferred from an off-chip storage buffer to another off-chip buffer. Originally, this second buffer was stored on-chip. Due to problems related to the stack, however, it was moved off-chip.

3) The channel data to be used is copied from the off-chip storage buffer to an on-chip buffer. It is then changed to a complex-interleaved format represented by an array of floats, and stored in another buffer. During this process it is multiplied by a windowing function (Hamming Window) to reduce processing artifacts caused by the small block size, and then attenuated based on distance from the virtual sound source to the listener.
4) The channel data is then FFTed to bring it into the frequency domain.
5) The channel data is then delayed through multiplication by a complex exponential representing a linear phase change (delay in time ? phase change in frequency). After this delay, it is complex multiplied by the appropriate HRTF.
6) The result of this multiplication is then added to the appropriate output buffer (for either the left or the right ear).
7) Steps 2 through 7 are completed for each chunk of processing being done (the contribution of one sound source to one ear represents a “chunk”).
8) The resulting 2 channels of perceptually-processed data are then transferred back to the PC for storage in the output data file.

After all of the data has been processed by the EVM, the PC writes the output file. This can then be accessed by a PC-based player program and played back to the user.

HRTFs

Convolution with the HRTFs was done in the frequency domain using complex multiplication. A MATLAB-generated plot of one set of HRTFs is represented below.



Delay

Our processing required delaying various chunks of input data by an arbitrary number of samples. Since the data is represented in the frequency domain for much of the processing, it seemed natural (and easy) to implement the n-sample delay by a complex multiplication by e^{-jwn}. Using Euler's relation, this is can be represented as a complex multiplication by $\cos(-wn) + j \cdot \sin(-wn) = \cos(wn) - j \cdot \sin(wn)$. Computations are done using local (on-chip) variables to minimize the number of off-chip memory references. Due to storage limitations, the delay computation is done in-place.

There are two different types of delay taken into account in perceptually processing the data stream: delay based on distance from the virtual sound source to the listener, and a delay from a sound source to one ear relative to the other caused by the angle that the virtual speaker lies at to the listener.

Range-based Delay

The time that a signal takes to traverse a range is: $time = \frac{range}{velocity}$

The velocity of a sound wave in air is dependent upon the temperature, and is defined as:

$$v_T = 344 + 0.6 \cdot (T - 20) \text{ m/s}$$

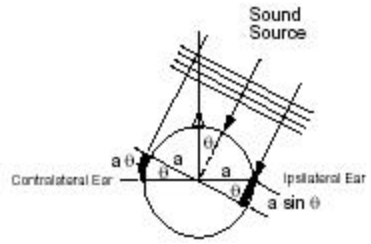
Distance (range) and the sampling rate (44,100 samples/second) are known. Therefore, the number of samples of delay can be determined as:

$$samples = sampling\_rate \cdot time = sampling\_rate \cdot \frac{range}{velocity} = sampling\_rate \cdot \frac{range}{v_T}$$

Woodworth Delay

When a sound source is at an angle to the listener's head, the sound waves arrive at each ear at a different time. This time difference is based not only on the angle of incidence, but also on the diameter of the person's head. This delay is referred to as "Woodworth Delay"[ix], and accounts for a portion of how your mind perceives sound as coming from a certain direction.

WOODWORTH'S FORMULA



The additional distance that the sound wave must travel to the farther ear is: $d ? a\sin ?$. This delay in samples can be determined much in the same way as it was previously to account for the positioning of the sound sources. The advancement of the sound wave for the other ear is determined in much the same way by basically taking the negative of the delay that was calculated for the other ear (assuming that the head in question is symmetrical).

Attenuation

Attenuation[x] was also calculated based on the distance from the user to the virtual sound source. Distance is related to a decrease in relative intensity through the *inverse-square law*. Assuming no reflected sound, this law states that the intensity of a sound wave decreases proportionally with the square of the distance: $\frac{I_2}{I_1} ? \left(\frac{r_1}{r_2}\right)^2$

Furthermore, we know that the intensity of a sound wave is related proportionally to the square of the amplitude of the wave: $\frac{I_2}{I_1} ? \left(\frac{A_2}{A_1}\right)^2$

We can derive from these two equations the relationship: $A_2 ? A_1 ? \left(\frac{r_1}{r_2}\right)^2$ where $A_1 ? 1$, being the reference amplitude of the signal.

Windowing Function

The storage limitations of the on-chip C67 memory limited the size of the data blocks we could process to 1024 samples. The unfortunate consequence of this is an abundance of irritating processing artifacts. In attempt to reduce this noise, we used a Hamming windowing function. The well-known Hamming window is represented by the equation $w[n] ? 0.54 ? 0.46 ?\cos\left(\frac{2 ?? ?n}{M}\right)$ where M is the block size. The input data is then multiplied by this function before processing. Use of the windowing function was

successful in removing many processing artifacts, though it introduced a low-frequency modulation that some of us found almost as irritating as the artifacts it removed.

Optimizations

DMA Utilization

Extensive use of DMA (Direct Memory Access) transfers were made in order to increase the overall processing speed of the EVM program. One convenience of the DMA transfer is that it is asynchronous, meaning that it can be initiated and the processor can then be free to do other processing while the transfer takes place in the background. DMA also is a burst transfer method, meaning that it only suffers the access penalty to off-chip RAM once. DMA transfers were started before other processing routines not yet needing this data were run, and then afterwards, when the data was needed, a check was performed to see if the DMA transfer had finished yet or not. If it had not, the program waits for it to finish before processing continued. The fact that there were two DMA channels available was also taken advantage of. In some cases, it was possible to utilize both DMA channels for background transfers at the same time, thus also increasing the processing speed.

Memory Allocation (Data)

Reliability and speed problems were in encountered in attempting to determine an optimal configuration for the placement of variables used in the processing. Initially, all of the variables most important to processing were located on-chip. These included the buffers for the HRTFs being used for processing, the buffers for the sections of data being processed, and the buffers for the output of the data to the Codec. When it became apparent that buffers large enough to be worthwhile would not fit in this format, the number of buffers was decreased. The HRTFs were stored off-chip, and then brought on-chip into a single buffer as they were being processed. The sections of data were also stored off-chip, and then brought on into a single buffer as they were necessary for processing. The output buffers remained on-chip, in order to not make too large of speed sacrifices. Later, as overwriting the stack became a problem (see later section: Issues Overcome), the HRTF being used was moved off-chip. This resulted in a significant performance hit, but was necessary in order to get the EVM program to function properly. An attempt was never made, however, to store the off-chip variables in SBSRAM (as opposed to SDRAM0). This might have helped performance a slight amount, for although they both suffer the same access penalty (~15 processor cycles), the SBSRAM operates at a slightly higher clock rate (133 MHz) than the SDRAM does (100 MHz).

Memory Allocation (Program)

The C67 DSP Chip only has 64k of on-chip memory dedicated to program code. Initially, the program code was too large to fit on-chip and had to be stored off-chip in SBSRAM. A major speed improvement came when the program code was made to fit on the chip. This way, accesses did not need to be made to off-chip memory in order to

obtain the program code for execution. By moving the code on-chip, we obtained a speed improvement up to 15 times our original processing speed.

PC ? EVM Communication

The PC code was written from the beginning to be efficient. The biggest possibility for optimization would be implementing asynchronous transfers between the PC and the EVM. In the end, we determined that our on-board processing was so computationally intensive that the PC to EVM communication was not a significant bottleneck, so we would not benefit greatly from a switch to asynchronous communication.

The only other possible PC-side optimization would be a removal of the current requirement that the PC allocate a slice of memory equal to the size of the output file. Currently, these files have tended to be several MB. For really long demos (longer than ours) this approach is inefficient. Improving the efficiency would require some more sophisticated WAV-writing code, and very minor modifications to the PC-side transfer-request processing loop. However, this didn't seem to be causing issues for our performance, so it is left as "future work".

Interrupts

A significant speed increase was obtained by removing the Codec outputs from the EVM processing routine. Not only was the speed of the processing increased (by not having the ~17 processor cycle delay each time that a request was made for a sample to be outputted), but also the size of the code was decreased. Removing the interrupt routines and the associated libraries also made it more possible to fit the program code on-chip.

Convolutional-based Attempt

A quick aside was taken in the form of a version of the code that did convolution in the time domain as opposed to in the frequency domain with the FFT. It was thought that this format of processing would make the implementation of effects like delay and attenuation easier to implement. The basic formula for discrete convolution was implemented in this version: $y[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot h[n - k]$. This version was significantly slower, however, taking roughly three times as much time to process one half of the amount of information as the FFT based version. It was abandoned after a minimal amount of experimentation.

Profiling

EVM Program Version	**11**	**11**	**11**	**11**	**11**
Demo	City - Full	City - Full	City - Full	City - Full	City - Full
Processing Type	FFT	FFT	FFT	FFT	FFT
Down-Mix Type	Dolby 5.1	Dolby 5.1	Dolby 5.1	Dolby 5.1	Stereo
Location Given to Linker (.cmd)					
Program Code (.text)	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA
Global Variables (.bss)	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA
Stack (.stack)	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA
Malloc Variables (.sysmem)	SDRAM0	SDRAM0	SDRAM0	SDRAM0	SDRAM0
Location of Declared Variables					
Channel_Buffer	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA
FFT_Buffer	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA
HRTF	SDRAM0	SDRAM0	SDRAM0	SDRAM0	SDRAM0
Temp (left/right) Buffers	SDRAM0	SDRAM0	SDRAM0	SDRAM0	SDRAM0
Temp_Out	SDRAM0	SDRAM0	SDRAM0	SDRAM0	SDRAM0
Processing Attributes					
Delay	Dist. & Wood.	Dist. & Wood.	Dist. & Wood.	Dist. & Wood.	N/A
Attentuation	Distance	Distance	None	None	N/A
Windowing Function	Hamming	None	Hamming	None	N/A
FFT Version	ASM	ASM	ASM	ASM	ASM
Number of HRTFs Used	10	10	10	10	10
Codec Output	None	None	None	None	None
Memory Sizes (bytes)					
ONCHIP_PROG	41632	41536	41120	41024	41120
ONCHIP_DATA	27136	27136	27136	27136	27136
SBSRAM_PROG	0	0	0	0	0
SDRAM0	1048768	1048768	1048768	1048768	1048768
Function Times (cycles)					
process_data( )	13783941	13548456	13631205	13434530	N/A
get_HRTFs( )	9232705	8508147	3590549	8686503	8102605
process_data_stereo( )	N/A	N/A	N/A	N/A	784463
Notes					

EVM Program Version	**10**	**9**	**8**	**8**	**7**
Demo	City	City	City	City	City
Processing Type	FFT	FFT	Convolution	Convolution	FFT
Down-Mix Type	Dolby 5.1	Dolby 5.1	Dolby 5.1	Stereo	Dolby 5.1
Location Given to Linker (.cmd)					
Program Code (.text)	SBSRAM_PROG	SBSRAM_PROG	SBSRAM_PROG	SBSRAM_PROG	SBSRAM_PROG
Global Variables (.bss)	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA
Stack (.stack)	ONCHIP_DATA	ONCHIP_DATA	SBSRAM_DATA	SBSRAM_DATA	SBSRAM_DATA
Malloc Variables (.sysmem)	SDRAM0	SDRAM0	SDRAM0	SDRAM0	SDRAM0
Location of Declared Variables					
Channel_Buffer	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA
FFT_Buffer	ONCHIP_DATA	ONCHIP_DATA	N/A	N/A	ONCHIP_DATA
HRTF	SDRAM0	SDRAM0	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA
Temp (left/right) Buffers	SDRAM0	SDRAM0	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA
Temp_Out	SDRAM0	SDRAM0	ONCHIP_DATA	ONCHIP_DATA	ONCHIP_DATA
Processing Attributes					
Delay	Distance	None	Dist. & Wood.	None	None
Attentuation	None	None	None	None	None
Windowing Function	Hamming	None	N/A	N/A	None
FFT Version	C	C	N/A	N/A	C
Number of HRTFs Used	10	5	5	5	5
Codec Output	N/A	2 Channel	2 Channel	2 Channel	2 Channel
Memory Sizes (bytes)					
ONCHIP_PROG	N/A	704	704	704	704
ONCHIP_DATA	N/A	32993	22917	22917	31985
SBSRAM_PROG	N/A	95136	88800	88128	91776
SDRAM0	N/A	1050392	1050376	1050376	1050392
Function Times (cycles)					
process_data( )	N/A	21034699	129320626	N/A	33837456
get_HRTFs( )	N/A	14209275	119633	1306646	22222327
process_data_stereo( )	N/A	N/A	N/A	281872	N/A
Notes	Intermediate version… not really used.		Only 512 samples processed.	Only 512 samples processed.	

In running the profiler, the first few iterations of any given function were allowed to run and then the maximum value recorded. Running through the entire processing cycle would have taken far too long to complete with the profiler running.

The version number above denotes which iteration of our code is represented. Versions 1 through 6 are not worth mentioning. Version 11 was the version that was demoed.

Issues Overcome

FFT

There were a few different options for the choice of FFT routines. Our initial choice was the TI radix-4 FFT routine cfftr4(), due to its speed as hand-optimized assembly code. We unfortunately ran into problems with this code entering an infinite loop, so we investigated alternative routines. TI also provides radix-2 assembly code, which we tried but could not compile. We also investigated the possibility of using a TI-supplied radix-2 IFFT routine (also in assembly), but later discovered a workaround for the radix-4 version before we began testing it. Our first workaround for the radix-4 code was to use the equivalent C code; this seemed to work well (albeit with sub-optimal performance). We then discovered that the ASM infinite loop was due to the enabling of interrupts. Since we had come to the conclusion that the computationally intense processing demanded by our application could not run in real time on the C67, we realized that we could disable interrupts and write the output to the PC instead of the codec. The disabled interrupts enabled us to use our preferred FFT routine: TI's optimized radix-4 assembly code.

The Linker and Memory Addressing

When trying to compile some of our code, we would receive (one or more) error messages from the linker stating that there was a relocation error, and that .text was being truncated at a given hex address. After some investigation of the Code Composer help system, and an examination of the linker-generated list file (.lst - search the second column for the hex address in the error message), we determined that the error was related to the addressing of our global arrays. The C67 apparently uses a form of relative addressing, and the linker was attempting to calculate the difference between the start of the BSS segment and the start of our array. This difference exceeded the 16 bits that the corresponding instruction could hold, so the linker failed. We guessed that the linker allocated arrays in the same order that they were declared in the source, so we simply declared the offending arrays at the top of our C source code. Our hope was that this would move the arrays closer to the BSS segment. This workaround fixed the problem. One caveat: if there are too many global (on-chip) arrays, no amount of moving their declarations around is likely to help.

Stack Growth

Another vexing issue that presented itself was mysterious program code errors. Our processing function would suddenly jump to the middle of main() for no discernable reason. We surmised that the stack must have been corrupted, producing a bad return address. Moving the stack temporarily off-chip eliminated the error, confirming our suspicion. We found that the problem was due to a declaration of large arrays within the scope of our processing function. The local arrays were being placed on the stack,

causing the stack to grow so large that it extended into the space occupied by some global arrays. When we wrote to these arrays during processing, the stack became corrupted. It seems that Code Composer does not check for such possibilities at compile-time. Our workaround to this problem was to instead declare pointers inside the scope of the function, and use malloc() to assign those pointers. This placed the arrays in the off-chip SDRAM bank rather than on the stack, eliminating the problem. The downside to this necessity is that the local arrays in question now require off-chip accesses.

This workaround could be made more efficient by calling malloc() only once, in main(), rather than in each call of the processing function. However, we did not have time to implement this optimization.

Future Work

In order to transform this project into a commercially-viable product, a number of additional steps would need to be taken.

1) The DSP architecture would need to be integrated with some communications scheme that would read the Dolby Digital bitstream from whatever source was desired to be perceptually-encoded. The input of an optical connection could potentially be tested using the Hoontech ST DB III Digital I/O Bracket (see http://www.hoontech.com/). This would allow the input of digital data to a Sound Blaster sound card, which could then be passed over the PCI bus to the EVM.
2) The Dolby Digital bitstream would need to be decoded directly on the DSP chip. This could potentially be done by an additional chip, but decoding the data on the DSP chip would cut out the cost necessitated by the additional chip. This decoding could be done with optimized assembly code available from TI for the fixed-point C62 processor (which can also be run on the floating point C67).
3) The processing would naturally need to be real-time in order for this product to be commercially-viable. This could probably be accomplished through a series of optimizations in the code written for the EVM. First, all important variables would need to be stored on-chip for processing. Second, asynchronous transfers would need to be used wherever possible. Next, further extensive profiling of the code would need to be performed in order to determine bottlenecks in the processing. Other general optimizations could also be made. Another possible option would be to use the fixed point C62 processor instead of the floating-point C67 processor. All of the necessary code (FFT, AC-3 decoding, etc.) is available from TI for the C62, and none of the processing techniques performed truly require a floating-point processor. In addition, the C62 (150 - 300 MHz) can be obtained in clock speeds faster than the C67 (100 - 225 MHz). The faster clock speeds would give more headroom in making the processing real time.
4) Output of the data would need to be sent to a Codec (D/A) of some sort for output. Originally, this was done by interrupting the processor on the EVM and then requesting a sample for the on-board Codec. This proved to be problematic as it not only incurred a significant speed hit (~17 processor cycles to process

each interrupt request), but it also led to problems with using the TI radix-4 FFT assembly code. This output could be done, however, by transferring the data over the PCI bus to a sound card. The sound card would be programmed to buffer the data and then output it via its own Codec.

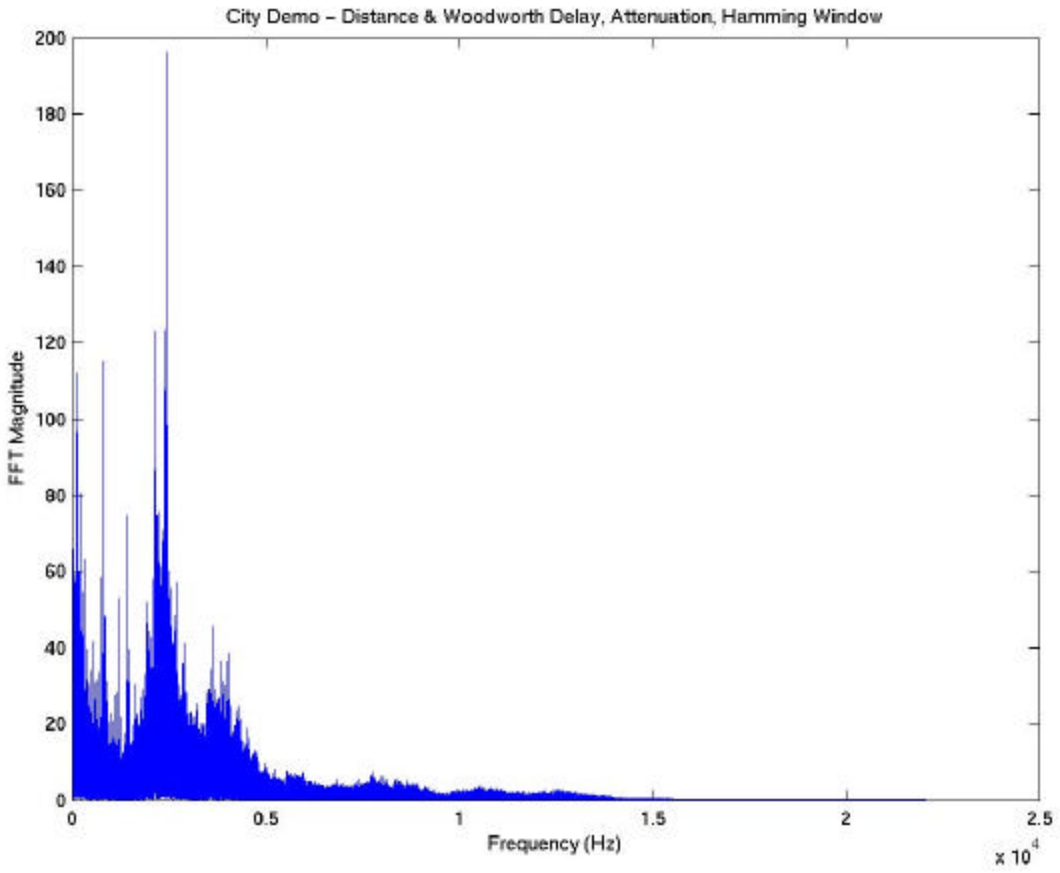
After these testing issues were resolved, the various pieces could then be integrated together to form a product that could then be debugged, tested, and eventually brought to market.

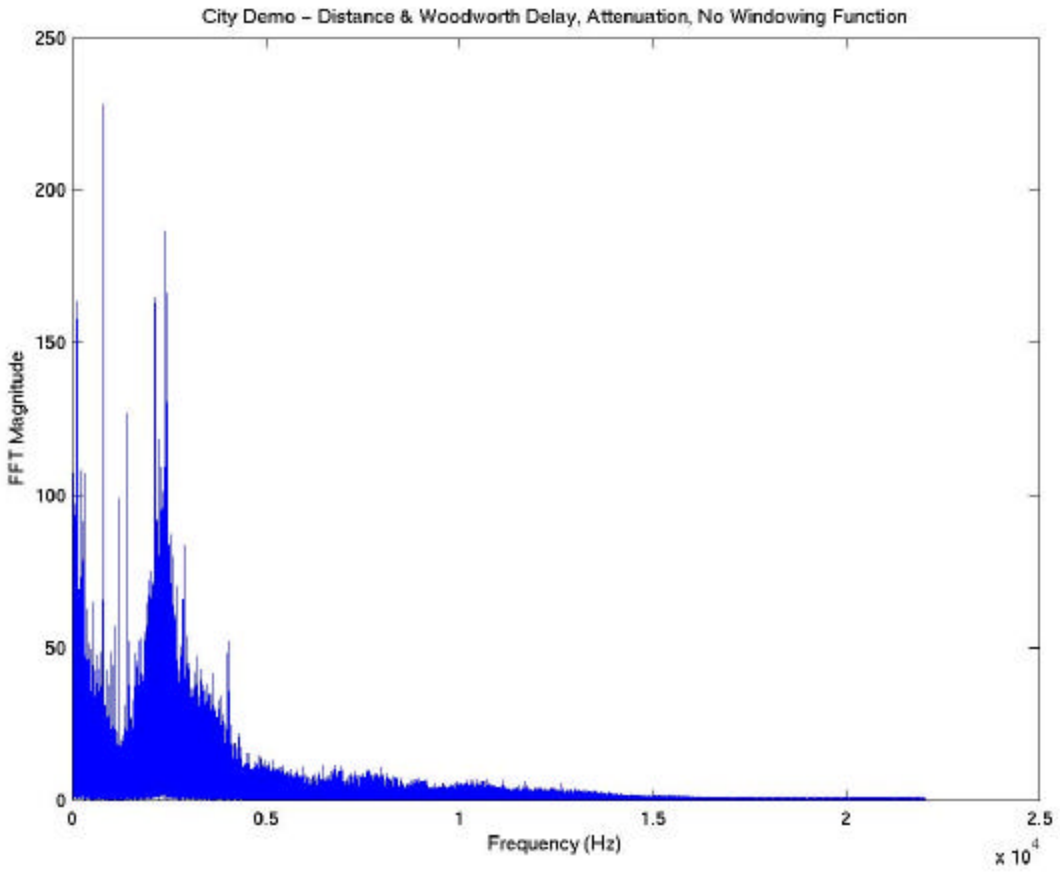
Conclusion

The actual perception of the generated [perceptually-processed] stereo signal varied from user to user. In general, both versions produced (with and without use of the windowing function) had annoying processing artifacts. When the windowing function was used, it caused a periodic variation in the audible volume of the output signal. This was most likely due to the shape of the windowing function and the frequency with which it was multiplied by the input signals (44100 samples/second / 1024 samples processed/piece = 43.07 Hz). A larger processing chunk was not feasible, however, due to limitations in memory allocation. When the windowing function wasn't used, an annoying background static was present throughout the signal. This was most likely due to the sidelobes caused by multiplication by essentially a rectangular windowing function.

The HRTFs also had some undesired effects on the processed data. Because of the way that the HRTFs were originally recorded at the MIT Media Lab, the response of your ear cavity is already factored into the HRTF. When you listen to the resulting sound files, however, your ear is still having these effects on the sound. Therefore, you are essentially getting a "double dose" of ear response in the processed sound. Your ear is not as responsive to very low (<100 Hz) or very high (>10,000 Hz) frequencies as it is to ones centered around the vocal range (~1000 Hz). The HRTF also models the effects of the resonances at approximately 4,000 Hz and 12,000 Hz caused by the ear cavity itself. This "double duty" effect cause some sounds to sound as if they were pitch-shifted, and others to not be as audibly loud as they should have been. In a future revision, the ear response could possibly be removed from the HRTFs using an inverse filter before they are used in processing.

Overall, the processing did improve the spatial impression caused by the audio to the user. One significant drawback was the lack of accompanying video. Many cues to the location of sound are drawn, quite simply, from what your eyes see. Without the accompanying video, some of the perceptual cues in the audio are decentuated. Certain negative effects, like front-back reversal (hearing sounds "in front of you" that are supposed to sound as if they came from "in back of you," and visa versa) and the perception that sounds are "inside your head" as opposed to in front of or in back of you arise without the visual component of the signal. The negative results of processing also made it difficult to pick up these perceptual cues. Still, the results prove promising the possibility of this eventually becoming a commercially-viable product.


City Demo – Distance & Woodworth Delay, Attenuation, Hamming Window
FFT Magnitude
Frequency (Hz)
x 10^4
0
20
40
60
80
100
120
140
160
180
200
0.5
1
1.5
2
2.5



These two FFT plots are appended in case you were curious (I was).

Appendix A R– EVM Code

----------File: EVM_Main.c----------

```
/*************************************************************
 * 18-551 Group 4 Final Project                              *
 * EVM_Main.c                                                *
 * Author: dcb2@andrew.cmu.edu                               *
 *************************************************************/

#include <stdio.h>
#include <stdlib.h>

#include <common.h>
#include <board.h>            /* EVM Library */
#include <mathf.h>            /* Math Library */
#include <dma.h>              /* DMA Library */
#include <pci.h>              /* PCI Library */

#define HRTF_SIZE 512           // # samples of the MIT HRTF
#define TRANSFER_SIZE 1024      // # samples of data to get from PC at once
#define INPUT_BUFFER_SIZE 1024  // # samples to be processed at once on chip
#define COMPLEX_INPUT_BUFFER_SIZE 2*INPUT_BUFFER_SIZE

// # bytes of INPUT_BUFFER_SIZE
#define MEMORY_BUFFER_SIZE INPUT_BUFFER_SIZE*sizeof(short int)

// # bytes of complex-interleaved HRTF
#define HRTF_MEMORY_SIZE INPUT_BUFFER_SIZE*2*sizeof(float)

#define WEIGHT 0.75           /* Weighting factor for recombined signal */
#define HEAD_WIDTH 7.5        /* Width of your head in cm */

#define STORAGE_SIZE 1024     // # samples of off-chip data storage

// # bytes of STORAGE_SIZE
#define MEMORY_STORAGE_SIZE STORAGE_SIZE*sizeof(short int)

/* transfer commands */
#define XFER_LF   0x00  /* transfer the LF channel */
#define XFER_C    0x01
#define XFER_RF   0x02
#define XFER_RS   0x03
#define XFER_LS   0x04
#define XFER_LFE  0x05
#define XFER_HRTF_L_LF  0x06  /* transfer the HRTF at Left ear, LF channel */
#define XFER_HRTF_R_LF  0x07  /* ... Right ear, LF channel */
#define XFER_HRTF_L_C   0x08  /* ... */
#define XFER_HRTF_R_C   0x09
#define XFER_HRTF_L_RF  0x0a
#define XFER_HRTF_R_RF  0x0b
#define XFER_HRTF_L_RS  0x0c
#define XFER_HRTF_R_RS  0x0d
#define XFER_HRTF_L_LS  0x0e
#define XFER_HRTF_R_LS  0x0f
#define XFER_HRTF_L_LFE 0x10
```

```
#define XFER_HRTF_R_LFE 0x11

/* transfer status flag whether params have been updated */
#define XFER_UPDATE_FLAG  0x12

#define XFER_DISTANCES    0x13  /* send new distances */
#define XFER_ANGLES       0x14
#define XFER_TEMPERATURE  0x15
#define XFER_ELEV_ANGLE   0x16

#define XFER_RECV_BEGIN   0x17 /* tell PC about to start sending data */
#define XFER_RECV_DATA    0x18 /* send PC data */
#define XFER_RECV_FINISH  0x19 /* tell PC done sending data blocks, write to
                                  file
                               */

#define XFER_DOLBY_FLAG   0x20

#define XFER_EXIT_PROGRAM 0xFF


/* Communication Variables */
int dev;
unsigned int temp_msg = 0;
unsigned int useDolby = 1;  /* this starts off as 1. The PC assumes the same.
                               If you change the default value of useDolby,
                               be sure to change the corresponding default in
                               the PC code getParams()
                             */

/* Delay Variables */
int LF_Delay = 0;
int C_Delay = 0;
int RF_Delay = 0;
int RS_Delay = 0;
int LS_Delay = 0;
int LFE_Delay = 0;

float V_sound = 344;
unsigned int sampling_rate = 44100;

/* Data Buffers */
short int Channel_Buffer[INPUT_BUFFER_SIZE];          // size = 2k
float FFT_Buffer[COMPLEX_INPUT_BUFFER_SIZE];          // size = 8k

/* FFT Stuff */
float Twiddle_Factors[COMPLEX_INPUT_BUFFER_SIZE];     // size = 8k
short int Reverse_Table[INPUT_BUFFER_SIZE];           // size = 2k

/* Other process_data( ) Variables */
float Hamming_Window[INPUT_BUFFER_SIZE];              // size = 4k
                                          // total size = 24k

/* HRTF Processing Variables */

/* Buffers to store HRTF's -- would be nice on chip, but can't fit */
float *Left_Front_Left_HRTF;        // size = 8k
```

```
float *Left_Front_Right_HRTF;       // size = 8k
float *Center_Left_HRTF;            // size = 8k
float *Center_Right_HRTF;           // size = 8k
float *Right_Front_Left_HRTF;       // size = 8k
float *Right_Front_Right_HRTF;            // size = 8k
float *Right_Surr_Left_HRTF;        // size = 8k
float *Right_Surr_Right_HRTF;       // size = 8k
float *Left_Surr_Left_HRTF;         // size = 8k
float *Left_Surr_Right_HRTF;        // size = 8k

/* Buffers to store unpacked Dolby Digital -- should be off chip */
short int *LF_Chan_Storage;
short int *C_Chan_Storage;
short int *RF_Chan_Storage;
short int *RS_Chan_Storage;
short int *LS_Chan_Storage;
short int *LFE_Chan_Storage;

/* Environment variables */
int temperature = 20;
int elevationAngle = 0;
float *distances = NULL; /* distances to LF, C, RF, RS, LS, LFE. 6 floats */
int *angles = NULL;      /* angles (LF, C, RF, RS, LS, LFE) - ints */

/* Function prototypes */

/* Stolen from Lab 2 */
void mkrevtable(int n);
void fillwtable(int n);
void cfftr4_dif(float* x, float* w, short n); /* Prototype for FFT routine */

/* Stolen from Lab 3 */
int request_transfer(void *buf, int size, int command);
int wait_transfer( );

/* Written for Project */
void process_data( );
int receive_message( );
int receive_data_from_PC( );
void get_HRTF(float *HRTF_Location, unsigned int cmd);

/* Other functions present but not prototyped... */

/********************** FUNCTIONS ****************************/

/* dma_copy_block: Copies numBytes from src to dest using DMA channel
      chan.  chan can be 0 or 1.  this function is ASYNCHRONOUS!  You must
      poll DMA0_TRANSFER_COUNT (or DMA1_TRANSFER_COUNT) to see when the
      transfer is complete */
/* Only valid for numBytes < 4 * 0xFFFF */
int dma_copy_block(void *src, void *dest, int numBytes, int chan)
{
  unsigned int dma_pri_ctrl=0;
  unsigned int dma_tcnt=0;

  /* Give DMA priority over CPU, and increment src and dest
      after each element */
```

```
  dma_pri_ctrl = 0x01000050;

  /* One frame, and we're using 4 byte elements */
  dma_tcnt = 0x00010000 | (numBytes/4);

  /* Write to DMA channel configuration registers */
  dma_init(chan,
           dma_pri_ctrl,
           0,
           (unsigned int) src,
           (unsigned int) dest,
           dma_tcnt);

  DMA_START(chan);
  return(OK);
}


/* This function creates the lookup table for digit reversing.  After
   it is run, Reverse_Table[n] equals the pairwise digit-reversal of n.
   n is the size of the FFT this table will be used for.*/
void mkrevtable(int n) {
  int bits, i, j, r, o;

  bits= (31 - _lmbd(1, n))/2;  /* _lmbd(1,n) finds leftmost 1 bit in n */
  for(i=0; i<n; i++) {
    r=0; o=i;
    _nassert(bits>=3);
    for(j=0; j<bits; j++) {
      r <<= 2;
      r |= o & 0x03;
      o >>= 2;
    }
    Reverse_Table[i] = r;
  }
}


/* Function for filling the table of FFT twiddle factors.  n is the
   size of the FFT to be used */
void fillwtable (int n) {
  int i;

  for (i = 0; i < 2*n; i+=2) {
    Twiddle_Factors[i] = cosf(PI*i/n);
    Twiddle_Factors[i + 1] = sinf(PI*i/n);
  }
}


/* function to delay a signal. Inputs are frequency domain representation
   of the signal (float array and its length, where length is the number
   of complex points), and the number of samples to delay by. Negative
   delays would be an advance in time.
   Assumes that freqResponse is complex interleaved.
 */
void delay(float *freqResponse, int freqLength, int delay)
```

```
{
  int i = 0;
  float A, B;
  float a, b;

  /* basically just multiplying freqResponse by e^(-j*w*delay) */
  for(i = 0; i < 2*freqLength; i+=2)
  {
    A = cosf(PI*delay*i/freqLength);
    B = sinf(PI*delay*i/freqLength);
    a = freqResponse[i];
    b = freqResponse[i+1];
    freqResponse[i] = a*A - b*B;
    freqResponse[i+1] = a*B + b*A;
  }

}


/* use global Input_Buffer, Output_Buffer */
void getHRTF(float *HRTF_Location, unsigned int cmd) {
  int i;
  short int *Input_Buffer;

  Input_Buffer = malloc(HRTF_SIZE*sizeof(short int)); // size = 1k

  /* Receive the HRTF */
  request_transfer((void *) Input_Buffer, HRTF_SIZE*sizeof(short int), cmd);
  wait_transfer();

  /* Format the HRTF as a complex interleaved array of floats. */
  for (i = 0; i < HRTF_SIZE; i++) {
    FFT_Buffer[2 * i] = ((float) Input_Buffer[i]) / 32767.0;
    FFT_Buffer[2*i + 1] = 0.0;
  }

  /* Pad the HRTF with zeros if necessary. */
  if (HRTF_SIZE < INPUT_BUFFER_SIZE)
    for (; i < COMPLEX_INPUT_BUFFER_SIZE; i++)
      FFT_Buffer[i] = 0.0;

  /* FFT the HRTF */
  cfftr4_dif(FFT_Buffer, Twiddle_Factors, (short) INPUT_BUFFER_SIZE);

  /* Move it into the appropriate buffer space. */
  memcpy(HRTF_Location, FFT_Buffer, HRTF_MEMORY_SIZE);
  free(Input_Buffer);
}


int getHRTFs(void)
{
  getHRTF(Left_Front_Left_HRTF, XFER_HRTF_L_LF);
  getHRTF(Left_Front_Right_HRTF, XFER_HRTF_R_LF);
  getHRTF(Center_Left_HRTF, XFER_HRTF_L_C);
  getHRTF(Center_Right_HRTF, XFER_HRTF_R_C);
  getHRTF(Right_Front_Left_HRTF, XFER_HRTF_L_RF);
```

```
  getHRTF(Right_Front_Right_HRTF, XFER_HRTF_R_RF);
  getHRTF(Right_Surr_Left_HRTF, XFER_HRTF_L_RS);
  getHRTF(Right_Surr_Right_HRTF, XFER_HRTF_R_RS);
  getHRTF(Left_Surr_Left_HRTF, XFER_HRTF_L_LS);
  getHRTF(Left_Surr_Right_HRTF, XFER_HRTF_R_LS);

 return 0;
}


/* returns 1 if reached end of data, 0 otherwise */
int receive_data_from_PC()
{
  unsigned int num, val;

  /* Receive the data and place it in the appropriate off-chip storage buffer
*/
  request_transfer((unsigned int *) LF_Chan_Storage,
TRANSFER_SIZE*sizeof(short int), XFER_LF);
  val = wait_transfer();

  request_transfer((unsigned int *) C_Chan_Storage,
TRANSFER_SIZE*sizeof(short int), XFER_C);
  num = wait_transfer();
  if(num < val)
    val = num;

  request_transfer((unsigned int *) RF_Chan_Storage,
TRANSFER_SIZE*sizeof(short int), XFER_RF);
  num = wait_transfer();
  if(num < val)
    val = num;

  request_transfer((unsigned int *) RS_Chan_Storage,
TRANSFER_SIZE*sizeof(short int), XFER_RS);
  num = wait_transfer();
  if(num < val)
    val = num;

  request_transfer((unsigned int *) LS_Chan_Storage,
TRANSFER_SIZE*sizeof(short int), XFER_LS);
  num = wait_transfer();
  if(num < val)
    val = num;

  request_transfer((unsigned int *) LFE_Chan_Storage,
TRANSFER_SIZE*sizeof(short int), XFER_LFE);
  num = wait_transfer();
  if(num < val)
    val = num;

  if(val < TRANSFER_SIZE*sizeof(short int))
      return 1;
  else
    return 0;
}
```

```
int request_transfer(void *buf, int size, int command)
{
    amcc_mailbox_write(2, size);
    amcc_mailbox_write(3, command);
      pci_message_sync_send((unsigned int)buf, FALSE);
      return 0;
}


int wait_transfer()
{
      unsigned int value;

      pci_message_sync_retrieve(&value);
      return value;
}


/* Perceptually processes the response of one channel of data to one ear. */
void process_piece(float *currHRTF, float *HRTF_Buffer, int currChannel,
short int *nextChannel, float *Left, float *Right, int Delay, float
Attenuation) {
  int i;
  int j, k, l, m;
  int rev_i;

  /* Bring the HRTF from off-chip memory into on-chip memory. */
  dma_copy_block(currHRTF, HRTF_Buffer, HRTF_MEMORY_SIZE, 0);

  /* Wait for the data transfer from the previous function call to finish. */
  if(currChannel != 0)
      while(DMA1_XFER_COUNTER != 0) {}

  /* Copy the short int data for the LF Channel to the buffer. */
  for (i = 0; i < INPUT_BUFFER_SIZE; i++) {
    FFT_Buffer[2 * i] = (((float) Channel_Buffer[i]) / 32767.0) *
Hamming_Window[i] /* Attenuation*/;
    FFT_Buffer[2 * i + 1] = 0.0;
  }

  /* Begin the copy of the next piece of data. */
  if(nextChannel != NULL)
    dma_copy_block(nextChannel, Channel_Buffer, MEMORY_BUFFER_SIZE, 1);

  /* FFT the data */
  cfftr4_dif(FFT_Buffer, Twiddle_Factors, INPUT_BUFFER_SIZE);  // might have
to type-cast data to (float *)

  /* Delay the channel data */
  delay(FFT_Buffer, INPUT_BUFFER_SIZE, Delay);

  /* Wait for the DMA transfer of the HRTF to finish. */
  while(DMA0_XFER_COUNTER != 0) {}

  /* Complex multiplication: (a + bj)(c + dj) = (ac - bd) + (ad + bc)j */
  /* FFT --> IFFT: Reverse real & imaginary components */
```

```
  for (i = 0; i < INPUT_BUFFER_SIZE; i++) {
    rev_i = Reverse_Table[i];

    j = 2 * i;
    k = j + 1;
    l = 2 * rev_i;
    m = l + 1;

    if (Left != NULL) {
      Left[k] += FFT_Buffer[l] * HRTF_Buffer[l] - FFT_Buffer[m] *
HRTF_Buffer[m];
      Left[j] += FFT_Buffer[l] * HRTF_Buffer[m] + FFT_Buffer[m] *
HRTF_Buffer[l];
    }
    if (Right != NULL) {
      Right[k] += FFT_Buffer[l] * HRTF_Buffer[l] - FFT_Buffer[m] *
HRTF_Buffer[m];
      Right[j] += FFT_Buffer[l] * HRTF_Buffer[m] + FFT_Buffer[m] *
HRTF_Buffer[l];
    }
  }
}


/* Does straight stereo down-mix (no perceptual processing) from one channel
to both ears. */
void process_stereo_piece(short int *currChannel, int *Left, int *Right, int
leftWeight, int rightWeight) {
  int i;

  dma_copy_block(currChannel, Channel_Buffer, MEMORY_BUFFER_SIZE, 1);

  while(DMA1_XFER_COUNTER != 0) {}

  for (i = 0; i < INPUT_BUFFER_SIZE; i++) {
    Left[i] = Channel_Buffer[i] / leftWeight;
    Right[i] = Channel_Buffer[i] / rightWeight;
  }
}


/* Does straight stereo down-mix of data (no perceptual processing). */
void process_data_stereo( ) {
  int *Left_Ear_Temp;         // size = 4k
  int *Right_Ear_Temp;        // size = 4k
  short int *Temp_Out;        // size = 4k

  int i;

  Left_Ear_Temp = malloc(INPUT_BUFFER_SIZE * sizeof(int));
  Right_Ear_Temp = malloc(INPUT_BUFFER_SIZE * sizeof(int));
  Temp_Out = malloc(COMPLEX_INPUT_BUFFER_SIZE * sizeof(short int));

  process_stereo_piece(LFE_Chan_Storage, Left_Ear_Temp, Right_Ear_Temp, 1,
1);
```

```
  process_stereo_piece(LF_Chan_Storage, Left_Ear_Temp, Right_Ear_Temp, 1,
10);

  process_stereo_piece(C_Chan_Storage, Left_Ear_Temp, Right_Ear_Temp, 1, 1);

  process_stereo_piece(RF_Chan_Storage, Left_Ear_Temp, Right_Ear_Temp, 10,
1);

  process_stereo_piece(RS_Chan_Storage, Left_Ear_Temp, Right_Ear_Temp, 10,
1);

  process_stereo_piece(LS_Chan_Storage, Left_Ear_Temp, Right_Ear_Temp, 1,
10);

  for (i = 0; i < INPUT_BUFFER_SIZE; i++) {
    Temp_Out[2 * i] = (short int) (Left_Ear_Temp[i] / 2);
    Temp_Out[2*i + 1] = (short int) (WEIGHT * Right_Ear_Temp[i] / 2);
  }

  request_transfer(Temp_Out, COMPLEX_INPUT_BUFFER_SIZE * sizeof(short int),
XFER_RECV_DATA);
  wait_transfer( );

  free(Left_Ear_Temp);
  free(Right_Ear_Temp);
  free(Temp_Out);
}


/* Calculates delay to an ear in samples based on Woodworth's formula. */
int woodworthDelay(int angle, int onAxis, int velocity) {
  int samples;

  samples = (HEAD_WIDTH * sinf(PI/180 * ((float) angle)))/velocity *
sampling_rate;

  if (onAxis == 0)
    samples = -samples;

  return samples;
}


/* Returns the relative amplitude (decrease) of a sound at distance 2
   to that of a sound at distance 1. */
float amplitude(float distance1, float distance2) {
  float multiplier;

  multiplier = distance1/distance2;

  return multiplier;
}


/* Does perceptual processing on data. */
void process_data( ) {
  /* This function assumes that the HRTF's have already been FFTed */
```

```
  float *HRTF;                      // size = 8k
  float *Left_Ear_Temp;       // size = 8k
  float *Right_Ear_Temp;      // size = 8k
  short int *Temp_Out;        // size = 4k

  float Attenuation;

  int i;
  int j, k, l, m;
  int rev_i;

  HRTF = malloc(COMPLEX_INPUT_BUFFER_SIZE * sizeof(float));
  Left_Ear_Temp = malloc(COMPLEX_INPUT_BUFFER_SIZE * sizeof(float));
  Right_Ear_Temp = malloc(COMPLEX_INPUT_BUFFER_SIZE * sizeof(float));
  Temp_Out = malloc(COMPLEX_INPUT_BUFFER_SIZE * sizeof(short int));

  /* Low Frequency Effects Channel */

  /* Copy the LFE Channel short int data to the buffer and reset output
buffers */
  dma_copy_block(LFE_Chan_Storage, Channel_Buffer, MEMORY_BUFFER_SIZE, 1);

  /* Calculate attenuation factor */
  Attenuation = amplitude(1.4,distances[5]);

  while(DMA1_XFER_COUNTER != 0) {}

  for (i = 0; i < INPUT_BUFFER_SIZE; i++) {
    j = 2 * i;
    k = j + 1;

    FFT_Buffer[j] = (((float) Channel_Buffer[i]) / 32767.0) * 0.2 *
Hamming_Window[i] /* Attenuation */;
    FFT_Buffer[k] = 0.0;

      /* Resetting doesn't really need to be done...
    Left_Ear_Temp[j] = 0.0;
    Left_Ear_Temp[k] = 0.0;

    Right_Ear_Temp[j] = 0.0;
    Right_Ear_Temp[k] = 0.0;
    */
  }

  dma_copy_block(C_Chan_Storage, Channel_Buffer, MEMORY_BUFFER_SIZE, 1);

  /* FFT the data */
  cfftr4_dif(FFT_Buffer, Twiddle_Factors, INPUT_BUFFER_SIZE);

  /* Delay the channel data */
  delay(FFT_Buffer, INPUT_BUFFER_SIZE, LFE_Delay + 25);

  /* Currently, no processing is being done on the LFE Channel, so it doesn't
need to be
     multiplied by a transfer function. */
  for (i = 0; i < INPUT_BUFFER_SIZE; i++) {
```

```
    rev_i = Reverse_Table[i];

    j = 2 * i;
    k = j + 1;
    l = 2 * rev_i;
    m = l + 1;

    /* Left and right ears are symmetric at this point, so only one needs to
be calculated. */
    Left_Ear_Temp[k] = FFT_Buffer[l];
    Right_Ear_Temp[k] = Left_Ear_Temp[k];

    Left_Ear_Temp[j] = FFT_Buffer[m];
    Right_Ear_Temp[j] = Left_Ear_Temp[j];
  }

  /* Center Channel -- Left Ear */
  process_piece(Center_Left_HRTF, HRTF, 1, NULL, Left_Ear_Temp, NULL,
C_Delay, amplitude(1.4,distances[1]));

  /* Center Channel -- Right Ear */
  process_piece(Center_Right_HRTF, HRTF, 0, LF_Chan_Storage, NULL,
Right_Ear_Temp, C_Delay, amplitude(1.4,distances[1]));

  /* Left Front Channel -- Left Ear */
  process_piece(Left_Front_Left_HRTF, HRTF, 1, NULL, Left_Ear_Temp, NULL,
LF_Delay + woodworthDelay(360 - angles[0],1,V_sound),
amplitude(1.4,distances[0]));

  /* Left Front Channel -- Right Ear */
  process_piece(Left_Front_Right_HRTF, HRTF, 0, RF_Chan_Storage, NULL,
Right_Ear_Temp, LF_Delay + woodworthDelay(360 - angles[0],0,V_sound),
amplitude(1.4,distances[0]));

  /* Right Front Channel -- Left Ear */
  process_piece(Right_Front_Left_HRTF, HRTF, 1, NULL, Left_Ear_Temp, NULL,
RF_Delay + woodworthDelay(angles[2],0,V_sound), amplitude(1.4,distances[2]));

  /* Right Front Channel -- Right Ear */
  process_piece(Right_Front_Right_HRTF, HRTF, 0, RS_Chan_Storage, NULL,
Right_Ear_Temp, RF_Delay + woodworthDelay(angles[2],1,V_sound),
amplitude(1.4,distances[2]));

  /* Right Surround Channel -- Left Ear */
  process_piece(Right_Surr_Left_HRTF, HRTF, 1, NULL, Left_Ear_Temp, NULL,
RS_Delay + woodworthDelay(180 - angles[3],0,V_sound),
amplitude(1.4,distances[3]));

  /* Right Surround Channel -- Right Ear */
  process_piece(Right_Surr_Right_HRTF, HRTF, 0, LS_Chan_Storage, NULL,
Right_Ear_Temp, RS_Delay + woodworthDelay(180 - angles[3],1,V_sound),
amplitude(1.4,distances[3]));

  /* Left Surround Channel -- Left Ear */
  process_piece(Left_Surr_Left_HRTF, HRTF, 1, NULL, Left_Ear_Temp, NULL,
LS_Delay + woodworthDelay(angles[4] - 180,1,V_sound),
amplitude(1.4,distances[4]));
```

```
  /* Left Surround Channel -- Right Ear */
  process_piece(Left_Surr_Right_HRTF, HRTF, 0, NULL, NULL, Right_Ear_Temp,
LS_Delay + woodworthDelay(angles[4] - 180,0,V_sound),
amplitude(1.4,distances[4]));

  /* IFFT the buffers */

  cfftr4_dif(Left_Ear_Temp, Twiddle_Factors, INPUT_BUFFER_SIZE);
  cfftr4_dif(Right_Ear_Temp, Twiddle_Factors, INPUT_BUFFER_SIZE);

  /* Send data to output buffer */
  /* Magnitude values stored in ... ? */
  for (i = 0; i < INPUT_BUFFER_SIZE; i++) {
    rev_i = Reverse_Table[i];

    Temp_Out[2 * i] = (short int) (WEIGHT * Left_Ear_Temp[2*rev_i + 1] *
32767 / INPUT_BUFFER_SIZE);
    Temp_Out[2*i + 1] = (short int) (WEIGHT * Right_Ear_Temp[2*rev_i + 1] *
32767 / INPUT_BUFFER_SIZE);
  }

  request_transfer(Temp_Out, COMPLEX_INPUT_BUFFER_SIZE * sizeof(short int),
XFER_RECV_DATA);
  wait_transfer( );

  free(HRTF);
  free(Left_Ear_Temp);
  free(Right_Ear_Temp);
  free(Temp_Out);
}


/***************************** MAIN *****************************/
int main(void) {

  int i;
  unsigned int exit_program = 0;
  float d_scale;

  evm_init();               /* Standard board initialization */

  pci_driver_init();        /* Call before using any PCI code */
  dev = pci_fifo_open();    /* Open FIFO */

  /* Allocate memory off-chip for distances */
  distances = malloc(6*sizeof(float));
  angles = malloc(6*sizeof(int));

  /* Allocate memory off-chip for input data storage */
  LF_Chan_Storage = (short int *) malloc(MEMORY_STORAGE_SIZE);
  C_Chan_Storage = (short int *) malloc(MEMORY_STORAGE_SIZE);
  RF_Chan_Storage = (short int *) malloc(MEMORY_STORAGE_SIZE);
  RS_Chan_Storage = (short int *) malloc(MEMORY_STORAGE_SIZE);
  LS_Chan_Storage = (short int *) malloc(MEMORY_STORAGE_SIZE);
  LFE_Chan_Storage = (short int *) malloc(MEMORY_STORAGE_SIZE);
```

```
  /* Allocate memory off-chip for HRTF storage */
  Left_Front_Left_HRTF = (float *) malloc(HRTF_MEMORY_SIZE);
  Left_Front_Right_HRTF = (float *) malloc(HRTF_MEMORY_SIZE);
  Center_Left_HRTF = (float *) malloc(HRTF_MEMORY_SIZE);
  Center_Right_HRTF = (float *) malloc(HRTF_MEMORY_SIZE);
  Right_Front_Left_HRTF = (float *) malloc(HRTF_MEMORY_SIZE);
  Right_Front_Right_HRTF = (float *) malloc(HRTF_MEMORY_SIZE);
  Right_Surr_Left_HRTF = (float *) malloc(HRTF_MEMORY_SIZE);
  Right_Surr_Right_HRTF = (float *) malloc(HRTF_MEMORY_SIZE);
  Left_Surr_Left_HRTF = (float *) malloc(HRTF_MEMORY_SIZE);
  Left_Surr_Right_HRTF = (float *) malloc(HRTF_MEMORY_SIZE);

  /* Check that memory was successfully allocated */
  if( LF_Chan_Storage == NULL || C_Chan_Storage == NULL ||
      RF_Chan_Storage == NULL || LFE_Chan_Storage == NULL ||
      LS_Chan_Storage == NULL || RS_Chan_Storage == NULL ||
      Left_Front_Left_HRTF == NULL || Left_Front_Right_HRTF == NULL ||
      Center_Left_HRTF == NULL || Center_Right_HRTF == NULL ||
      Right_Front_Left_HRTF == NULL || Right_Front_Right_HRTF == NULL ||
      Right_Surr_Left_HRTF == NULL || Right_Surr_Right_HRTF == NULL ||
      Left_Surr_Left_HRTF == NULL || Left_Surr_Right_HRTF == NULL ||
      distances == NULL || angles == NULL)
  {
    /* This section commented out so that program code will fit on chip.

    printf("Memory Allocation Failure.  Exiting...\n");
    if(angles != NULL) free(angles);
    if(distances != NULL) free(distances);
    if(LF_Chan_Storage != NULL) free(LF_Chan_Storage);
    if(C_Chan_Storage != NULL) free(C_Chan_Storage);
    if(RF_Chan_Storage != NULL) free(RF_Chan_Storage);
    if(RS_Chan_Storage != NULL) free(RS_Chan_Storage);
    if(LS_Chan_Storage != NULL) free(LS_Chan_Storage);
    if(LFE_Chan_Storage != NULL) free(LFE_Chan_Storage);
    if(Left_Front_Left_HRTF != NULL) free(Left_Front_Left_HRTF);
    if(Left_Front_Right_HRTF != NULL) free(Left_Front_Right_HRTF);
    if(Center_Left_HRTF != NULL) free(Center_Left_HRTF);
    if(Center_Right_HRTF != NULL) free(Center_Right_HRTF);
    if(Right_Front_Left_HRTF != NULL) free(Right_Front_Left_HRTF);
    if(Right_Front_Right_HRTF != NULL) free(Right_Front_Right_HRTF);
    if(Right_Surr_Left_HRTF != NULL) free(Right_Surr_Left_HRTF);
    if(Right_Surr_Right_HRTF != NULL) free(Right_Surr_Right_HRTF);
    if(Left_Surr_Left_HRTF != NULL) free(Left_Surr_Left_HRTF);
    if(Left_Surr_Right_HRTF != NULL) free(Left_Surr_Right_HRTF); */
    exit(1);
  }

  /* Calculate Hamming Window */
  for (i = 0; i < INPUT_BUFFER_SIZE; i++) {
    Hamming_Window[i] = 0.54 + 0.46*cosf(2*PI*(i-
INPUT_BUFFER_SIZE/2)/INPUT_BUFFER_SIZE);
  }

  /* Generate Reverse_Table & Twiddle Factors */
  mkrevtable(INPUT_BUFFER_SIZE);
  fillwtable(INPUT_BUFFER_SIZE);
```

```
  request_transfer(NULL, 0, XFER_RECV_BEGIN);
  wait_transfer();

  /* Basic Program Flow */

  /* Get HRTF's from PC, FFT them, and place them in the appropriate buffers
*/

  /* Get block of data from PC, place in storage off-chip */

  /* Start by getting initial data and HRTF's from PC */

  getHRTFs();
  receive_data_from_PC();

  request_transfer(distances, 6*sizeof(float), XFER_DISTANCES);
  wait_transfer();

  request_transfer(&temperature, sizeof(int), XFER_TEMPERATURE);
  wait_transfer();

  request_transfer(&elevationAngle, sizeof(int), XFER_ELEV_ANGLE);
  wait_transfer();

  d_scale = 1/ cosf(elevationAngle * PI/180);
  V_sound = 344 + 0.6 * (temperature - 20);
  LF_Delay = (int) (d_scale * (distances[0] - 1.4) / V_sound)* sampling_rate;
  C_Delay = (int) (d_scale *(distances[1] - 1.4) / V_sound) * sampling_rate;
  RF_Delay = (int) (d_scale * (distances[2] - 1.4) / V_sound)* sampling_rate;
  RS_Delay = (int) (d_scale * (distances[3] - 1.4) / V_sound)* sampling_rate;
  LS_Delay = (int) (d_scale * (distances[4] - 1.4) / V_sound)* sampling_rate;
  LFE_Delay =(int) (d_scale * (distances[5] - 1.4) / V_sound)* sampling_rate;

  request_transfer(&useDolby, sizeof(int), XFER_DOLBY_FLAG);
  wait_transfer();

  exit_program = 0;
  while(exit_program == 0)
  {
    /* Check for updated parameters - HRTFs, inputs, or distances */
    temp_msg = 0;
    request_transfer(&temp_msg, 4, XFER_UPDATE_FLAG);
    wait_transfer();
    //printf("Update check completed. Update flag = %x\n", temp_msg);

    /* temp_msg = 0x00 -> no updates
     *          = 0x01 -> HRTFs updated (i.e. LSB set)
     *          = 0x02 -> Inputs updated for new demo (i.e. second-LSB set)
     *          = 0x03 -> HRTFs & inputs updated
     *          = 0x04 -> distances updated (i.e. third-LSB set)
     *          = 0x05 -> distances, HRTFs updated
     *          = 0x06 -> distances, inputs updated
     *          = 0x07 -> everything updated
     *          = 0x08 -> temperature changed (fourth LSB set)
     *          = 0x10 -> elevation changed (fifth LSB set)
     *          = 0x20 -> Use Stereo Mix, not Dolby (sixth LSB set)
     */
```

```
    if(temp_msg & 0x01)
    {
      getHRTFs();
    }

    /* if(temp_msg & 0x02) – don't really need to do anything */

    if(temp_msg & 0x04) /* updated distances */
    {
      request_transfer(distances, 6*sizeof(float), XFER_DISTANCES);
      wait_transfer();
    }

    if(temp_msg & 0x08) /* updated temperature */
    {
      request_transfer(&temperature, sizeof(int), XFER_TEMPERATURE);
      wait_transfer();
    }

    if(temp_msg & 0x10) /* updated elevation */
    {
      request_transfer(&elevationAngle, sizeof(int), XFER_ELEV_ANGLE);
      wait_transfer();
    }

    if(useDolby == 0)
    {
      /* do Plain Old Stereo mix here */
      process_data_stereo( );
    }
    else
    {
      /* Perceptually process data here */
      d_scale = 1/ cosf(elevationAngle * PI/180);
      V_sound = 344 + 0.6 * (temperature - 20);
      LF_Delay =(int) (d_scale*(distances[0] - 1.4)/ V_sound)* sampling_rate;
      C_Delay = (int) (d_scale*(distances[1] - 1.4)/ V_sound)* sampling_rate;
      RF_Delay =(int) (d_scale*(distances[2] - 1.4) / V_sound)*sampling_rate;
      RS_Delay =(int) (d_scale*(distances[3] - 1.4)/ V_sound)* sampling_rate;
      LS_Delay =(int) (d_scale*(distances[4] - 1.4)/ V_sound)* sampling_rate;
      LFE_Delay =(int) (d_scale*(distances[5] - 1.4)/ V_sound)*sampling_rate;

      process_data();
    }

    /* ...now get data... */
    exit_program = receive_data_from_PC();
  }

  request_transfer(NULL, 1, XFER_RECV_FINISH);
  wait_transfer();

  request_transfer(NULL, 0, XFER_EXIT_PROGRAM); /* command to exit PC program
*/
  return 0;
}
```

Appendix B – PC Back End Code

---------- File: arrays.h ----------

```
#ifndef _ARRAYS_H_
#define _ARRAYS_H_


/* simple generic array of elements */
struct GenericArray
{
  unsigned int bytesPerElement;
  unsigned int numBytes;    /* total bytes in array */
  unsigned int numElements; /* total number of objects in array */
  void *data;
};
typedef struct GenericArray GenericArray;


/* simple array of shorts */
struct ShortArray
{
  unsigned int length;
  short int *data;
};

typedef struct ShortArray ShortArray;



/* simple array of ints */
struct IntArray
{
  unsigned int length;
  int *data;
};

typedef struct IntArray IntArray;



/* simple array of floats */
struct FloatArray
{
  unsigned int length;
  float *data;
};

typedef struct FloatArray FloatArray;


#endif
```

----------File: pc_comm.c----------

```
/*****************************************************/
/* 18-551 Final Project                              */
/* pc_comm.c: Communication routines for project     */
/*          (PC side)                                */
/* Author: <akrol@andrew.cmu.edu>                    */
/*****************************************************/

#include <stdio.h>
#include <windows.h>
#include "evm6xdll.h"
#include "arrays.h"
#include "read.h"


#undef LOAD_FILE
//#define LOAD_FILE    /* Uncomment to automatically load program */

/* transfer commands */
#define XFER_LF   0x00  /* transfer the LF channel */
#define XFER_C    0x01
#define XFER_RF   0x02
#define XFER_RS   0x03
#define XFER_LS   0x04
#define XFER_LFE  0x05

#define XFER_HRTF_L_LF  0x06  /* transfer the HRTF at Left ear, LF channel */
#define XFER_HRTF_R_LF  0x07
#define XFER_HRTF_L_C   0x08
#define XFER_HRTF_R_C   0x09
#define XFER_HRTF_L_RF  0x0a
#define XFER_HRTF_R_RF  0x0b
#define XFER_HRTF_L_RS  0x0c
#define XFER_HRTF_R_RS  0x0d
#define XFER_HRTF_L_LS  0x0e
#define XFER_HRTF_R_LS  0x0f
#define XFER_HRTF_L_LFE 0x10
#define XFER_HRTF_R_LFE 0x11

#define XFER_UPDATE_FLAG  0x12  /* status flag – have params been updated */
#define XFER_DISTANCES    0x13  /* 6 floats – distances to speakers*/
#define XFER_ANGLES       0x14
#define XFER_TEMPERATURE  0x15
#define XFER_ELEV_ANGLE   0x16

#define XFER_RECV_BEGIN   0x17   /* get ready to receive data from EVM */
#define XFER_RECV_DATA    0x18   /* get data from EVM */
#define XFER_RECV_FINISH  0x19   /* EVM is done sending data, write file */

#define XFER_DOLBY_FLAG   0x20   /* use Dolby, or plain old stereo? */

#define XFER_EXIT_PROGRAM 0xFF

#define PARAM_FILE "D:\\wwwroot\\parameters.txt"
#define UPDATE_FLAG_FILE "D:\\wwwroot\\updated.txt"
#define IN_USE_FLAG_FILE "D:\\wwwroot\\inuse.txt"
```

```
/* Structure for holding transfer information */
struct transfer_s {
      void *buffer;
      unsigned long size;
      unsigned long command;
      /* command will indicate which buffer to transfer:
         see the defined XFER_xx listings above
      */
};

/*******************FUNCTION PROTOTYPES*********************/
#ifdef LOAD_FILE
int load_file(HANDLE hBd, LPVOID hHpi);
#endif
int wait_request(struct transfer_s *ts);
int send_data(struct transfer_s *ts, void *local_buf);
int get_data(struct transfer_s *ts, void *local_buf);
void hpi_write_word(ULONG addr, ULONG data);


/*********************GLOBAL VARIABLES *******************/

/* input files */
char INFILE_LF[80];
char INFILE_C[80];
char INFILE_RF[80];
char INFILE_RS[80];
char INFILE_LS[80];
char INFILE_LFE[80];

/* HRTF input files - these should be variables set by getParams(),
   Naming is HRTF_FILE_<ear>_<channel>
 */
char HRTF_FILE_L_LF[80];
char HRTF_FILE_R_LF[80];
char HRTF_FILE_L_C[80];
char HRTF_FILE_R_C[80];
char HRTF_FILE_L_RF[80];
char HRTF_FILE_R_RF[80];
char HRTF_FILE_L_RS[80];
char HRTF_FILE_R_RS[80];
char HRTF_FILE_L_LS[80];
char HRTF_FILE_R_LS[80];
char HRTF_FILE_L_LFE[80];
char HRTF_FILE_R_LFE[80];


/* input data. See arrays.h for GenericArray structure definition */
/* To use buffered file I/O, change this to CircBuffer elements.
   For now, just use these so we can test that the basic stuff
   works.
 */
GenericArray LF_Storage;
GenericArray C_Storage;
```

```
GenericArray RF_Storage;
GenericArray RS_Storage;
GenericArray LS_Storage;
GenericArray LFE_Storage;


/* HRTF storage - these will be loaded in their entirety in
   the set_HRTF() function
*/
GenericArray HRTF_L_LF_Storage;
GenericArray HRTF_R_LF_Storage;
GenericArray HRTF_L_C_Storage;
GenericArray HRTF_R_C_Storage;
GenericArray HRTF_L_RF_Storage;
GenericArray HRTF_R_RF_Storage;
GenericArray HRTF_L_RS_Storage;
GenericArray HRTF_R_RS_Storage;
GenericArray HRTF_L_LS_Storage;
GenericArray HRTF_R_LS_Storage;
GenericArray HRTF_L_LFE_Storage;
GenericArray HRTF_R_LFE_Storage;

int params_updated = 0; /* value = 0x00 -> no updates
                                = 0x01 -> HRTFs updated
                                = 0x02 -> Inputs updated
                                = 0x03 -> HRTFs & inputs updated
                                = 0x04 -> distances updated
                                = 0x05 -> distances, HRTFs updated
                                = 0x06 -> distances, inputs updated
                                = 0x07 -> everything updated
                                                = 0x08 -> temperature
changed
                                                = 0x10 -> elevation changed
                                     This is set by call to getParams()
                           */


/* input WAV info ... these are opened with open_wav() inside the
   set_input() function, we need to call close_wav() on these after
   we're done with the input file!
 */
WAVFileInfo w_LF;
WAVFileInfo w_C;
WAVFileInfo w_RF;
WAVFileInfo w_RS;
WAVFileInfo w_LS;
WAVFileInfo w_LFE;

float distances[6] = {1.4, 1.4, 1.4, 1.4, 1.4, 1.4};
int angles[6];
int temperature = 0;
int elevationAngle[6] = {0, 0, 0, 0, 0, 0};
int useDolby = 1; /* note that the EVM assumes a default value of 1 for this,
                     so if you change this default value you need to modify
                     the EVM side too
                  */
```

```
WAVFileInfo w_out;


HANDLE hBd    = NULL;
LPVOID hHpi = NULL;
HANDLE h_event;
char s_buffer[80];


/******************FUNCTIONS BEGIN*********************************/


/* Gets parameters from GUI-generated text file.
   Returns >= 1 if changes, 0 if not
 */
int getParams(void)
{
    FILE *fi, *fi2, *fi3;
    float d0,d1,d2,d3,d4,d5;
    int d0_i, d1_i, d2_i, d3_i, d4_i, d5_i;
    int a0, a1, a2, a3, a4, a5;
    int temp, elevAng0, elevAng1, elevAng2, elevAng3, elevAng4, elevAng5;
    unsigned int local_flag = 0;

    char localHRTF_FILE_L_LF[80];
    char localHRTF_FILE_R_LF[80];
    char localHRTF_FILE_L_C[80];
    char localHRTF_FILE_R_C[80];
    char localHRTF_FILE_L_RF[80];
    char localHRTF_FILE_R_RF[80];
    char localHRTF_FILE_L_RS[80];
    char localHRTF_FILE_R_RS[80];
    char localHRTF_FILE_L_LS[80];
    char localHRTF_FILE_R_LS[80];
    char localHRTF_FILE_L_LFE[80];
    char localHRTF_FILE_R_LFE[80];
    char localINFILE_LF[80];
    char localINFILE_C[80];
    char localINFILE_RF[80];
    char localINFILE_RS[80];
    char localINFILE_LS[80];
    char localINFILE_LFE[80];
    char useDolbyString[80];

    /* check if files are in use (if so, IN_USE_FLAG_FILE will exist) */
    fi = fopen(IN_USE_FLAG_FILE, "r");

    if( fi != NULL)
    {
       /* then we need to wait a bit for the GUI to complete IO */
       fclose(fi);
       return 0;  /* the EVM will check again anyway, let's not make it wait
                     for slow IO
                  */
    }
    else
    /* then files aren't in use, so set in_use flag and read */
```

```
{
  fi = fopen(IN_USE_FLAG_FILE, "w");
  fi2 = fopen(UPDATE_FLAG_FILE, "r");
  if(fi2 == NULL) /* then no updates, so remove in_use lock and
                     return 0
                  */
  {
        fclose(fi);
        remove(IN_USE_FLAG_FILE);
        return 0;
  }
  else /* have updates, read them */
  {
        fclose(fi2);
        fi3 = fopen(PARAM_FILE, "r");

        fscanf(fi3, "%s", localINFILE_LF);
        fscanf(fi3, "%s", localINFILE_C);
        fscanf(fi3, "%s", localINFILE_RF);
        fscanf(fi3, "%s", localINFILE_RS);
        fscanf(fi3, "%s", localINFILE_LS);
        fscanf(fi3, "%s", localINFILE_LFE);
        fscanf(fi3, "%s", localHRTF_FILE_L_LF);
        fscanf(fi3, "%s", localHRTF_FILE_R_LF);
        fscanf(fi3, "%s", localHRTF_FILE_L_C);
        fscanf(fi3, "%s", localHRTF_FILE_R_C);
        fscanf(fi3, "%s", localHRTF_FILE_L_RF);
        fscanf(fi3, "%s", localHRTF_FILE_R_RF);
        fscanf(fi3, "%s", localHRTF_FILE_L_RS);
        fscanf(fi3, "%s", localHRTF_FILE_R_RS);
        fscanf(fi3, "%s", localHRTF_FILE_L_LS);
        fscanf(fi3, "%s", localHRTF_FILE_R_LS);
        fscanf(fi3, "%s", localHRTF_FILE_L_LFE);
        fscanf(fi3, "%s", localHRTF_FILE_R_LFE);
        fscanf(fi3, "%i", &d0_i);
        fscanf(fi3, "%i", &d1_i);
        fscanf(fi3, "%i", &d2_i);
        fscanf(fi3, "%i", &d3_i);
        fscanf(fi3, "%i", &d4_i);
        fscanf(fi3, "%i", &d5_i);
        fcanf(fi3, "%i", &a0);
        fscanf(fi3, "%i", &a1);
        fscanf(fi3, "%i", &a2);
        fscanf(fi3, "%i", &a3);
        fscanf(fi3, "%i", &a4);
        fscanf(fi3, "%i", &a5);
        fscanf(fi3, "%i", &elevAng0);
        fscanf(fi3, "%i", &elevAng1);
        fscanf(fi3, "%i", &elevAng2);
        fscanf(fi3, "%i", &elevAng3);
        fscanf(fi3, "%i", &elevAng4);
        fscanf(fi3, "%i", &elevAng5);
        fscanf(fi3, "%i", &temp);
        fscanf(fi3, "%s", useDolbyString);


        fclose(fi3);
```

```
remove(UPDATE_FLAG_FILE);
fclose(fi); // remove in_use lcck
remove(IN_USE_FLAG_FILE);

/* do unit conversions from English to metric */

/* convert temperature from Fahrenheit to Celsius */
temp = (temp - 32) * 5 / 9;

/* convert distances from feet to meters */
d0 = 0.3048 * d0_i;
d1 = 0.3048 * d1_i;
d2 = 0.3048 * d2_i;
d3 = 0.3048 * d3_i;
d4 = 0.3048 * d4_i;
d5 = 0.3048 * d5_i;


/* now check to see what's changed, and set global variables */

/* have HRTFs/angles changed? (angles will necessarily change the
   HRTFs, so we don't have to explicitly compare angle values too
 */
if(strcmp(localHRTF_FILE_L_LF, HRTF_FILE_L_LF) !=0)
      local_flag = local_flag | 0x01;
if(strcmp(localHRTF_FILE_R_LF, HRTF_FILE_R_LF) !=0)
      local_flag = local_flag | 0x01;
if(strcmp(localHRTF_FILE_L_C, HRTF_FILE_L_C) !=0)
      local_flag = local_flag | 0x01;
if(strcmp(localHRTF_FILE_R_C, HRTF_FILE_R_C) !=0)
      local_flag = local_flag | 0x01;
if(strcmp(localHRTF_FILE_L_RF, HRTF_FILE_L_RF) !=0)
      local_flag = local_flag | 0x01;
if(strcmp(localHRTF_FILE_R_RF, HRTF_FILE_R_RF) !=0)
      local_flag = local_flag | 0x01;
if(strcmp(localHRTF_FILE_L_RS, HRTF_FILE_L_RS) !=0)
      local_flag = local_flag | 0x01;
if(strcmp(localHRTF_FILE_R_RS, HRTF_FILE_R_RS) !=0)
      local_flag = local_flag | 0x01;
if(strcmp(localHRTF_FILE_L_LS, HRTF_FILE_L_LS) !=0)
      local_flag = local_flag | 0x01;
if(strcmp(localHRTF_FILE_R_LS, HRTF_FILE_R_LS) !=0)
      local_flag = local_flag | 0x01;
if(strcmp(localHRTF_FILE_L_LFE, HRTF_FILE_L_LFE) !=0)
      local_flag = local_flag | 0x01;
if(strcmp(localHRTF_FILE_R_LFE, HRTF_FILE_R_LFE) !=0)
      local_flag = local_flag | 0x01;

/* have inputs (i.e. which demo we're using) changed? */
if(strcmp(localINFILE_LF, INFILE_LF) !=0)
      local_flag = local_flag | 0x02;
if(strcmp(localINFILE_C, INFILE_C) !=0)
      local_flag = local_flag | 0x02;
if(strcmp(localINFILE_RF, INFILE_RF) !=0)
      local_flag = local_flag | 0x02;
if(strcmp(localINFILE_RS, INFILE_RS) !=0)
      local_flag = local_flag | 0x02;
```

```
if(strcmp(localINFILE_LS, INFILE_LS) !=0)
      local_flag = local_flag | 0x02;
if(strcmp(localINFILE_LFE, INFILE_LFE) !=0)
      local_flag = local_flag | 0x02;

/* have distances changed? */
if(d0 != distances[0])
      local_flag = local_flag | 0x04;
if(d1 != distances[1])
      local_flag = local_flag | 0x04;
if(d2 != distances[2])
      local_flag = local_flag | 0x04;
if(d3 != distances[3])
      local_flag = local_flag | 0x04;
if(d4 != distances[4])
      local_flag = local_flag | 0x04;
if(d5 != distances[5])
      local_flag = local_flag | 0x04;

/* has temperature changed? */
if(temperature != temp)
      local_flag = local_flag | 0x08;

/* has elevation changed? */
if(elevAng0 != elevationAngle[0] ||
   elevAng1 != elevationAngle[1] ||
   elevAng2 != elevationAngle[2] ||
   elevAng3 != elevationAngle[3] ||
   elevAng4 != elevationAngle[4] ||
   elevAng5 != elevationAngle[5] )
{
       local_flag = local_flag | 0x10;
}

/* have we switched between Dolby and Plain Old Stereo? */
if(strcmp("stereo", useDolbyString) == 0)
      useDolby = 0;
else
      useDolby = 1;

/* now copy local versions to global versions (if changes) */
if(local_flag != 0)
{
      memcpy(HRTF_FILE_L_LF, localHRTF_FILE_L_LF, 80);
      memcpy(HRTF_FILE_R_LF, localHRTF_FILE_R_LF, 80);
      memcpy(HRTF_FILE_L_C, localHRTF_FILE_L_C, 80);
      memcpy(HRTF_FILE_R_C, localHRTF_FILE_R_C, 80);
      memcpy(HRTF_FILE_L_RF, localHRTF_FILE_L_RF, 80);
      memcpy(HRTF_FILE_R_RF, localHRTF_FILE_R_RF, 80);
      memcpy(HRTF_FILE_L_RS, localHRTF_FILE_L_RS, 80);
      memcpy(HRTF_FILE_R_RS, localHRTF_FILE_R_RS, 80);
      memcpy(HRTF_FILE_L_LS, localHRTF_FILE_L_LS, 80);
      memcpy(HRTF_FILE_R_LS, localHRTF_FILE_R_LS, 80);
      memcpy(HRTF_FILE_L_LFE, localHRTF_FILE_L_LFE, 80);
      memcpy(HRTF_FILE_R_LFE, localHRTF_FILE_R_LFE, 80);

      memcpy(INFILE_LF, localINFILE_LF, 80);
```

```
                    memcpy(INFILE_C, localINFILE_C, 80);
                    memcpy(INFILE_RF, localINFILE_RF, 80);
                    memcpy(INFILE_RS, localINFILE_RS, 80);
                    memcpy(INFILE_LS, localINFILE_LS, 80);
                    memcpy(INFILE_LFE, localINFILE_LFE, 80);

                    distances[0] = d0;
                    distances[1] = d1;
                    distances[2] = d2;
                    distances[3] = d3;
                    distances[4] = d4;
                    distances[5] = d5;

                    angles[0] = a0;
                    angles[1] = a1;
                    angles[2] = a2;
                    angles[3] = a3;
                    angles[4] = a4;
                    angles[5] = a5;

                    elevationAngle[0] = elevAng0;
                    elevationAngle[1] = elevAng1;
                    elevationAngle[2] = elevAng2;
                    elevationAngle[3] = elevAng3;
                    elevationAngle[4] = elevAng4;
                    elevationAngle[5] = elevAng5;

                    temperature = temp;

                }
                return local_flag;
                }  // end else (have updates)

      } // end else (files aren't in use)
} // end getParams()
```

```
/* init() - set up board connections and windows events, default filenames */
int init(void)
{
  int temp = 0;
  int i;

  /***** SETUP EVM COMM STUFF ****/

  /* Open the board */
#ifdef LOAD_FILE
  hBd = evm6x_open(0, 1);
#else
```

```
  hBd = evm6x_open(0, 0);
#endif

  if ( hBd == INVALID_HANDLE_VALUE )
  {
    fprintf(stderr, "Couldn't open board\n");
    exit(1);
  }
  fprintf(stderr, "Opened connection to board...\n");

  /* Also open connection to HPI */
  hHpi = evm6x_hpi_open(hBd);
  if ( hHpi == NULL ) exit(4);

#ifdef LOAD_FILE
  load_file(hBd, hHpi);
  fprintf(stderr, "Loaded program...\n");
#else
  /* Set DMA AUX priority greater than CPU priority, so we
     don't lock the PCI bus */
  hpi_write_word(0x01840070 /*Addr*/, 0x00000010 /*Data*/);

  /* Reset the mailboxes and FIFO */
  hpi_write_word(0x0170003C, 0x0e000000);
#endif

  /* Setup a windows event so we don't have to poll for incoming messages */
  evm6x_clear_message_event(hBd);
  sprintf( s_buffer, "%s%d",EVM6X_GLOBAL_MESSAGE_EVENT_BASE_NAME, 0);
  h_event = OpenEvent( SYNCHRONIZE, FALSE, s_buffer );



  /******* INITIALIZE DATA ************/

  /* set pointers to be NULL initially */
  LF_Storage.data = NULL;
  C_Storage.data = NULL;
  RF_Storage.data = NULL;
  RS_Storage.data = NULL;
  LS_Storage.data = NULL;
  LFE_Storage.data = NULL;

  HRTF_L_LF_Storage.data = NULL;
  HRTF_R_LF_Storage.data = NULL;
  HRTF_L_C_Storage.data = NULL;
  HRTF_R_C_Storage.data = NULL;
  HRTF_L_RF_Storage.data = NULL;
  HRTF_R_RF_Storage.data = NULL;
  HRTF_L_RS_Storage.data = NULL;
  HRTF_R_RS_Storage.data = NULL;
  HRTF_L_LS_Storage.data = NULL;
  HRTF_R_LS_Storage.data = NULL;
  HRTF_L_LFE_Storage.data = NULL;
  HRTF_R_LFE_Storage.data = NULL;
```

```
  w_LF.isOpen = malloc(sizeof(int));
  w_C.isOpen = malloc(sizeof(int));
  w_RF.isOpen = malloc(sizeof(int));
  w_RS.isOpen = malloc(sizeof(int));
  w_LS.isOpen = malloc(sizeof(int));
  w_LFE.isOpen = malloc(sizeof(int));


  *(w_LF.isOpen) = 0;
  *(w_C.isOpen) = 0;
  *(w_RF.isOpen) = 0;
  *(w_RS.isOpen) = 0;
  *(w_LS.isOpen) = 0;
  *(w_LFE.isOpen) = 0;



  /* use default input file locations */
  strcpy(INFILE_LF, "c:\\WINNT\\Profiles\\551\\Desktop\\group4\\LF.wav");
  strcpy(INFILE_C, "c:\\WINNT\\Profiles\\551\\Desktop\\group4\\C.wav");
  strcpy(INFILE_RF, "c:\\WINNT\\Profiles\\551\\Desktop\\group4\\RF.wav");
  strcpy(INFILE_RS, "c:\\WINNT\\Profiles\\551\\Desktop\\group4\\RS.wav");
  strcpy(INFILE_LS, "c:\\WINNT\\Profiles\\551\\Desktop\\group4\\LS.wav");
  strcpy(INFILE_LFE, "c:\\WINNT\\Profiles\\551\\Desktop\\group4\\LFE.wav");


  strcpy(HRTF_FILE_L_LF,
"c:\\WINNT\\Profiles\\551\\Desktop\\group4\\HRTF_L_LF.wav");
  strcpy(HRTF_FILE_R_LF,
"c:\\WINNT\\Profiles\\551\\Desktop\\group4\\HRTF_R_LF.wav");
  strcpy(HRTF_FILE_L_C,
"c:\\WINNT\\Profiles\\551\\Desktop\\group4\\HRTF_L_C.wav");
  strcpy(HRTF_FILE_R_C,
"c:\\WINNT\\Profiles\\551\\Desktop\\group4\\HRTF_R_C.wav");
  strcpy(HRTF_FILE_L_RF,
"c:\\WINNT\\Profiles\\551\\Desktop\\group4\\HRTF_L_RF.wav");
  strcpy(HRTF_FILE_R_RF,
"c:\\WINNT\\Profiles\\551\\Desktop\\group4\\HRTF_R_RF.wav");
  strcpy(HRTF_FILE_L_RS,
"c:\\WINNT\\Profiles\\551\\Desktop\\group4\\HRTF_L_RS.wav");
  strcpy(HRTF_FILE_R_RS,
"c:\\WINNT\\Profiles\\551\\Desktop\\group4\\HRTF_R_RS.wav");
  strcpy(HRTF_FILE_L_LS,
"c:\\WINNT\\Profiles\\551\\Desktop\\group4\\HRTF_L_LS.wav");
  strcpy(HRTF_FILE_R_LS,
"c:\\WINNT\\Profiles\\551\\Desktop\\group4\\HRTF_R_LS.wav");
  strcpy(HRTF_FILE_L_LFE,
"c:\\WINNT\\Profiles\\551\\Desktop\\group4\\HRTF_L_LFE.wav");
  strcpy(HRTF_FILE_R_LFE,
"c:\\WINNT\\Profiles\\551\\Desktop\\group4\\HRTF_R_LFE.wav");

  for(i = 0; i < 6; i++)
      distances[i] = 1.4;

  angles[0] = 315; angles[1] = 45; angles[2] = 135;
  angles[4] = 225; angles[5] = 180;
```

```
  getParams();

  return 0;
} // end init()




/* set_input() - set what our new input data is based upon a GUI setting.
                 Open input files.
*/
int set_input(void)
{
  /* FIXME -  may do buffering for performance optimization...for now
     just straight file I/O.
  */

  /* if previous files still open, close them */
  if(*(w_LF.isOpen) == 1)
    close_wav(w_LF);
  if(*(w_C.isOpen) == 1)
    close_wav(w_C);
  if(*(w_RF.isOpen) == 1)
    close_wav(w_RF);
  if(*(w_RS.isOpen) == 1)
    close_wav(w_RS);
  if(*(w_LS.isOpen) == 1)
    close_wav(w_LS);
  if(*(w_LFE.isOpen) == 1)
    close_wav(w_LFE);


  /* open new files */
  w_LF = open_wav(INFILE_LF);
  w_C = open_wav(INFILE_C);
  w_RF = open_wav(INFILE_RF);
  w_RS = open_wav(INFILE_RS);
  w_LS = open_wav(INFILE_LS);
  w_LFE = open_wav(INFILE_LFE);


  return 0;

}




/* set_HRTFs() - determine what HRTFs we need, load them, signal EVM */
int set_HRTFs(void)
```

```
{
  WAVFileInfo w_L_LF;
  WAVFileInfo w_R_LF;
  WAVFileInfo w_L_C;
  WAVFileInfo w_R_C;
  WAVFileInfo w_L_RF;
  WAVFileInfo w_R_RF;
  WAVFileInfo w_L_RS;
  WAVFileInfo w_R_RS;
  WAVFileInfo w_L_LS;
  WAVFileInfo w_R_LS;
  WAVFileInfo w_L_LFE;
  WAVFileInfo w_R_LFE;

  /* FIXME -  needs parameters, and some code to figure out which
     files to load based on those parameters/numbers. For now just
     use #defined HRTF files
  */


  /* check for previous HRTF storage elements being allocated,
     and free them before reassigning the variables.
  */
  if(HRTF_L_LF_Storage.data != NULL)
    free(HRTF_L_LF_Storage.data);
  if(HRTF_R_LF_Storage.data != NULL)
    free(HRTF_R_LF_Storage.data);
  if(HRTF_L_C_Storage.data != NULL)
    free(HRTF_L_C_Storage.data);
  if(HRTF_R_C_Storage.data != NULL)
    free(HRTF_R_C_Storage.data);
  if(HRTF_L_RF_Storage.data != NULL)
    free(HRTF_L_RF_Storage.data);
  if(HRTF_R_RF_Storage.data != NULL)
    free(HRTF_R_RF_Storage.data);
  if(HRTF_L_RS_Storage.data != NULL)
    free(HRTF_L_RS_Storage.data);
  if(HRTF_R_RS_Storage.data != NULL)
    free(HRTF_R_RS_Storage.data);
  if(HRTF_L_LS_Storage.data != NULL)
    free(HRTF_L_LS_Storage.data);
  if(HRTF_R_LS_Storage.data != NULL)
    free(HRTF_R_LS_Storage.data);
  if(HRTF_L_LFE_Storage.data != NULL)
    free(HRTF_L_LFE_Storage.data);
  if(HRTF_R_LFE_Storage.data != NULL)
    free(HRTF_R_LFE_Storage.data);



  /* now open the new HRTFs */
  w_L_LF = open_wav(HRTF_FILE_L_LF);
  HRTF_L_LF_Storage = read_wav(w_L_LF, w_L_LF.samplesOfData);
  close_wav(w_L_LF);

  w_R_LF = open_wav(HRTF_FILE_R_LF);
  HRTF_R_LF_Storage = read_wav(w_R_LF, w_R_LF.samplesOfData);
```

```
  close_wav(w_R_LF);

  w_L_C = open_wav(HRTF_FILE_L_C);
  HRTF_L_C_Storage = read_wav(w_L_C, w_L_C.samplesOfData);
  close_wav(w_L_C);

  w_R_C = open_wav(HRTF_FILE_R_C);
  HRTF_R_C_Storage = read_wav(w_R_C, w_R_C.samplesOfData);
  close_wav(w_R_C);

  w_L_RF = open_wav(HRTF_FILE_L_RF);
  HRTF_L_RF_Storage = read_wav(w_L_RF, w_L_RF.samplesOfData);
  close_wav(w_L_RF);

  w_R_RF = open_wav(HRTF_FILE_R_RF);
  HRTF_R_RF_Storage = read_wav(w_R_RF, w_R_RF.samplesOfData);
  close_wav(w_R_RF);

  w_L_RS = open_wav(HRTF_FILE_L_RS);
  HRTF_L_RS_Storage = read_wav(w_L_RS, w_L_RS.samplesOfData);
  close_wav(w_L_RS);

  w_R_RS = open_wav(HRTF_FILE_R_RS);
  HRTF_R_RS_Storage = read_wav(w_R_RS, w_R_RS.samplesOfData);
  close_wav(w_R_RS);

  w_L_LS = open_wav(HRTF_FILE_L_LS);
  HRTF_L_LS_Storage = read_wav(w_L_LS, w_L_LS.samplesOfData);
  close_wav(w_L_LS);

  w_R_LS = open_wav(HRTF_FILE_R_LS);
  HRTF_R_LS_Storage = read_wav(w_R_LS, w_R_LS.samplesOfData);
  close_wav(w_R_LS);

  w_L_LFE = open_wav(HRTF_FILE_L_LFE);
  HRTF_L_LFE_Storage = read_wav(w_L_LFE, w_L_LFE.samplesOfData);
  close_wav(w_L_LFE);

  w_R_LFE = open_wav(HRTF_FILE_R_LFE);
  HRTF_R_LFE_Storage = read_wav(w_R_LFE, w_R_LFE.samplesOfData);
  close_wav(w_R_LFE);

  params_updated = 1;

  return 0;
}



/* main program loop */
int process(void)
{
  int program_exit=0;
  unsigned int result_counter = 0, result_size = 0;
  struct transfer_s ts;
  char * result;  // byte array
```

```
  /* Loop... Waits for messages from the EVM and does a transfer
     depending on the value of the message received */
  while(!program_exit) {
    wait_request(&ts);
     /*fprintf(stderr, "Transfer request: CMD %x, SIZE %i, ADDRESS %x\n",
ts.command,
          ts.size, ts.buffer);*/

    /* Now based on the command that was sent, transfer "size" # of samples
       from a particular buffer. For the input channels, we read from disk
       and then send the data, passing along how many samples we read
       successfully. For the HRTFs, we've already read the HRTFs previously,
       so just send them and their length.
    */

    switch(ts.command) {
    case(XFER_LF): /* Send LF */
      LF_Storage = read_wav(w_LF, ts.size /2); /* read_wav takes # samples,
ts.size is # bytes */
        if(ts.size != LF_Storage.numBytes)  /* reached end, don't have full
ts.size to xfer*/
         ts.size = LF_Storage.numBytes;
      send_data(&ts, LF_Storage.data);
        free(LF_Storage.data);
        printf("."); /* cheesy progress meter :) */
      break;
    case(XFER_C): /* Send C */
      C_Storage = read_wav(w_C, ts.size/2);
        if(ts.size != C_Storage.numBytes)
         ts.size = C_Storage.numBytes;
      send_data(&ts, C_Storage.data);
        free(C_Storage.data);
      break;
    case(XFER_RF): /* Send RF */
      RF_Storage = read_wav(w_RF, ts.size/2);
        if(ts.size != RF_Storage.numBytes)
         ts.size = RF_Storage.numBytes;
      send_data(&ts, RF_Storage.data);
        free(RF_Storage.data);
      break;
    case(XFER_RS): /* Send RS */
      RS_Storage = read_wav(w_RS, ts.size/2);
        if(ts.size != RS_Storage.numBytes)
         ts.size = RS_Storage.numBytes;
      send_data(&ts, RS_Storage.data);
        free(RS_Storage.data);
      break;
    case(XFER_LS): /* Send LS */
      LS_Storage = read_wav(w_LS, ts.size/2);
        if(ts.size != LS_Storage.numBytes)
         ts.size = LS_Storage.numBytes;
      send_data(&ts, LS_Storage.data);
        free(LS_Storage.data);
      break;
    case(XFER_LFE): /* Send LFE */
      LFE_Storage = read_wav(w_LFE, ts.size /2);
        if(ts.size != LFE_Storage.numBytes)
```

```
   ts.size = LFE_Storage.numBytes;
send_data(&ts, LFE_Storage.data);
  free(LFE_Storage.data);
break;
case(XFER_UPDATE_FLAG): /* HRTF update status flag - a 32-bit word */
  params_updated = getParams();
  send_data(&ts, &params_updated);
  if((params_updated & 0x01) != 0)
    set_HRTFs();
  if((params_updated & 0x02) != 0)
    set_input();
  params_updated = 0; /* reset flag, now that EVM has checked it */
  break;
case(XFER_HRTF_L_LF):  /* HRTFs should already by read, so send them */
  ts.size = HRTF_L_LF_Storage.numBytes;
  send_data(&ts, HRTF_L_LF_Storage.data);
  break;
case(XFER_HRTF_R_LF):
  ts.size = HRTF_R_LF_Storage.numBytes;
  send_data(&ts, HRTF_R_LF_Storage.data);
  break;
case(XFER_HRTF_L_C):
  ts.size = HRTF_L_C_Storage.numBytes;
  send_data(&ts, HRTF_L_C_Storage.data);
  break;
case(XFER_HRTF_R_C):
  ts.size = HRTF_R_C_Storage.numBytes;
  send_data(&ts, HRTF_R_C_Storage.data);
  break;
case(XFER_HRTF_L_RF):
  ts.size = HRTF_L_RF_Storage.numBytes;
  send_data(&ts, HRTF_L_RF_Storage.data);
  break;
case(XFER_HRTF_R_RF):
  ts.size = HRTF_R_RF_Storage.numBytes;
  send_data(&ts, HRTF_R_RF_Storage.data);
  break;
case(XFER_HRTF_L_RS):
  ts.size = HRTF_L_RS_Storage.numBytes;
  send_data(&ts, HRTF_L_RS_Storage.data);
  break;
case(XFER_HRTF_R_RS):
  ts.size = HRTF_R_RS_Storage.numBytes;
  send_data(&ts, HRTF_R_RS_Storage.data);
  break;
case(XFER_HRTF_L_LS):
  ts.size = HRTF_L_LS_Storage.numBytes;
  send_data(&ts, HRTF_L_LS_Storage.data);
  break;
case(XFER_HRTF_R_LS):
  ts.size = HRTF_R_LS_Storage.numBytes;
  send_data(&ts, HRTF_R_LS_Storage.data);
  break;
case(XFER_HRTF_L_LFE):
  ts.size = HRTF_L_LFE_Storage.numBytes;
  send_data(&ts, HRTF_L_LFE_Storage.data);
  break;
```

```
  case(XFER_HRTF_R_LFE):
    ts.size = HRTF_R_LFE_Storage.numBytes;
    send_data(&ts, HRTF_R_LFE_Storage.data);
    break;
  case(XFER_DISTANCES): /* send distances to each speaker */
    send_data(&ts, distances);
    break;
  case(XFER_ANGLES):
    send_data(&ts, angles);
    break;
  case(XFER_TEMPERATURE):
    send_data(&ts, &temperature);
    break;
case(XFER_ELEV_ANGLE):
    send_data(&ts, &(elevationAngle[0]));
    break;
  case(XFER_RECV_BEGIN): /* EVM is about to start sending data */
         /* EVM is writing the filtered version of input */
        result = (char *) calloc((w_LF.bytesOfData) * 2,1); // 2 channels
    result_size = (w_LF.bytesOfData) * 2;

        result_counter = 0;
        // now send some nonsense to make the EVM happy
        ts.size = 4;
        send_data(&ts, &result_counter);
        break;
  case(XFER_RECV_DATA): /* EVM is sending a block of data */
        if(result_counter >= result_size)
        {
       printf("result_counter exceeding size of result array\n");
           exit(1);
        }
    get_data(&ts, (result + result_counter));
        result_counter += ts.size;
        break;
  case(XFER_RECV_FINISH): /* write the collection of blocks to disk */
    /* write to file */
  w_out.nChannels = 2; // stereo
    w_out.nBlockAlign = w_out.nChannels * (w_LF.nBitsPerSample / 8) ;
    w_out.nBitsPerSample = w_LF.nBitsPerSample;
    w_out.nSamplesPerSec = w_LF.nSamplesPerSec;
    w_out.wFormatTag = w_LF.wFormatTag;
    w_out.samplesOfData = w_LF.samplesOfData;
  w_out.bytesOfData = result_size;
    w_out.nAvgBytesPerSec = w_out.nChannels * w_LF.nAvgBytesPerSec;
  write_wav(w_out, result, "output.wav");
    if(result != NULL)
      free(result);
    result_counter = 0;
    result_size = 0;
    ts.size = 4;
    /* send some nonsense to keep the EVM happy */
    send_data(&ts, &result_counter);
    break;
  case(XFER_DOLBY_FLAG):
        send_data(&ts, &useDolby);
        break;
```

```
    case(XFER_EXIT_PROGRAM): /* Exit program */
      program_exit = 1;
      break;
    default:
      fprintf(stderr, "Unknown command\n");
      break;
    }
  }

  return 0;
}




int cleanup()
{
  /* free data storage */
  if(LF_Storage.data != NULL)
    free(LF_Storage.data);
  if(C_Storage.data != NULL)
    free(C_Storage.data);
  if(RF_Storage.data != NULL)
    free(RF_Storage.data);
  if(RS_Storage.data != NULL)
    free(RS_Storage.data);
  if(LS_Storage.data != NULL)
    free(LS_Storage.data);
  if(LFE_Storage.data != NULL)
    free(LFE_Storage.data);

  if(HRTF_L_LF_Storage.data != NULL)
    free(HRTF_L_LF_Storage.data);
  if(HRTF_R_LF_Storage.data != NULL)
    free(HRTF_R_LF_Storage.data);
  if(HRTF_L_C_Storage.data != NULL)
    free(HRTF_L_C_Storage.data);
  if(HRTF_R_C_Storage.data != NULL)
    free(HRTF_R_C_Storage.data);
  if(HRTF_L_RF_Storage.data != NULL)
    free(HRTF_L_RF_Storage.data);
  if(HRTF_R_RF_Storage.data != NULL)
    free(HRTF_R_RF_Storage.data);
  if(HRTF_L_RS_Storage.data != NULL)
    free(HRTF_L_RS_Storage.data);
  if(HRTF_R_RS_Storage.data != NULL)
    free(HRTF_R_RS_Storage.data);
  if(HRTF_L_LS_Storage.data != NULL)
    free(HRTF_L_LS_Storage.data);
  if(HRTF_R_LS_Storage.data != NULL)
    free(HRTF_R_LS_Storage.data);
  if(HRTF_L_LFE_Storage.data != NULL)
    free(HRTF_L_LFE_Storage.data);
  if(HRTF_R_LFE_Storage.data != NULL)
    free(HRTF_R_LFE_Storage.data);
```

```
  /* close input files */
  close_wav(w_LF);
  close_wav(w_C);
  close_wav(w_RF);
  close_wav(w_RS);
  close_wav(w_LS);
  close_wav(w_LFE);


  /* Clean up and exit */
  if (!evm6x_hpi_close(hHpi)) exit(9);
  if (!evm6x_close(hBd)) exit(16);

  return(0);
}



/* main() - just here for now. Once we make a DLL, we won't need this
            and will call the other function from the GUI
 */
int main(int argc, char** argv)
{
  init();
  set_HRTFs();
  set_input();
  printf("Beginning processing.\n");
  process();
  printf("\nProcessing finished.\n");
  /*cleanup();    commented out - need to rewrite cleanup a bit to avoid
crash*/
}




/* Waits for windows event signalling incoming message.  Then reads
      address from mailbox 1, size from mailbox 2, and command from
      mailbox 3 */
int wait_request(struct transfer_s *ts)
{
      /* wait for event signaling a message from the DSP */
      WaitForSingleObject( h_event, INFINITE );

      if(!evm6x_retrieve_message(hBd, (unsigned long *)&ts->buffer))
            fprintf(stderr, "Error retrieving 1...\n");
      if(!evm6x_mailbox_read(hBd, 2, (unsigned long *)&ts->size))
            fprintf(stderr, "Error retrieving 2...\n");
      if(!evm6x_mailbox_read(hBd, 3, (unsigned long *)&ts->command))
            fprintf(stderr, "Error retrieving 3...\n");

      return(0);
}

/* Size and address are given in struct ts.  local_buf is the address of
```

```
      the PC buffer to send. Also tells the EVM how much actually transferred
 */
int send_data(struct transfer_s *ts, void *local_buf)
{
      unsigned long int ulLength = ts->size;

      /* Write the data to EVM memory*/
      if (!evm6x_hpi_write(hHpi, (PULONG)local_buf, (PULONG)&ulLength,
(ULONG)ts->buffer)) {
            fprintf(stderr, "HPI write error\n");
            exit(1);
      }
      if (ulLength != ts->size) {
            fprintf(stderr, "HPI only wrote %i bytes\n");
            exit(1);
      }

      /* Use a message to signal the EVM that the transfer is done, and
         how much was transferred (in some cases may not be the same
         as the amount requested
       */
      if (!evm6x_send_message(hBd, (PULONG)&ts->size)) {
            fprintf(stderr, "Send message error!\n");
            exit(1);
      }
      return(0);
}

/* Same format as send_data */
int get_data(struct transfer_s *ts, void *local_buf)
{
      unsigned long int ulLength = ts->size;

      /* Read data from EVM memory */
    if (!evm6x_hpi_read(hHpi, (PULONG)local_buf, (PULONG)&ulLength,
(ULONG)ts->buffer)) {
            fprintf(stderr, "HPI write error\n");
            exit(1);
      }
      if (ulLength != ts->size) {
            fprintf(stderr, "HPI only wrote %i bytes\n");
            exit(1);
      }

      /* Signal EVM that transfer is done */
      if (!evm6x_send_message(hBd, (PULONG)&ts->command)) {
            fprintf(stderr, "Send message error!\n");
            exit(1);
      }
      return(0);
}

/* Write a single 32-bit word to EVM memory */
void hpi_write_word(ULONG addr, ULONG data)
{
      ULONG ulLength = 4;
```

```
      if (!evm6x_hpi_write(hHpi, &data, &ulLength, addr)) {
            fprintf(stderr, "Error writing word via HPI\n");
            exit(1);
      }
      if (ulLength != 4 ) {
            fprintf(stderr, "Error writing word via HPI\n");
            exit(1);
      }
}

#ifdef LOAD_FILE
/* Load the .out file and run the program.  Code Composer must not
      be running */
int load_file(HANDLE hBd, LPVOID hHpi)
{
  if ( !evm6x_reset_board(hBd) ) exit(2);
  if ( !evm6x_reset_dsp(hBd,HPI_BOOT) ) exit(3);
  if ( !evm6x_init_emif(hBd, hHpi) ) exit(5);

  /* Load program */
  if (!evm6x_coff_load(hBd,hHpi,EVM_FILE,0,0,0)) exit(8);

  /* Set HPI priority and reset mailboxes */
  hpi_write_word(0x01840070 /*Addr*/, 0x00000010 /*Data*/ );
  hpi_write_word(0x0170003C, 0x0e000000);

  if (!evm6x_unreset_dsp(hBd)) exit(10);
  if (!evm6x_set_timeout(hBd,1000)) exit(11);
      return(0);
}
#endif
```

---------- File: read.h ----------

```
/*************************************************************************
**
 *  Read.h   - header file for some easy-to-use WAV reading functions
*
 *
*
 *  Usage: First call open_wav(<filename>) to get a WAVFileInfo struct. Then
*
 *         use this struct as the argument (along with number of samples to
*
 *         read) to read_wav(<WAVFileInfo struct>, <numSamples>). When done
*
 *         after some arbitrary number of read_wav() calls, use close_wav()
*
 *         on the struct.
*

*************************************************************************
/
#include "arrays.h"


struct WAVFileInfo
{
  FILE *fi;
  /* isOpen - is the file still open? Declared as a pointer so that
     when passing the structure between functions (call-by-value) the
     value pointed to may be modified by the called function. This is used
     particularly in closing the file, and to ensure the file is closed
     exactly once. The pointed-to integer is 1 if file is open, 0 else
  */
  int* isOpen;
  unsigned short int wFormatTag; /* wave format. Standard format is 1 */
  unsigned short int nChannels; /* number of audio channels in file */
  unsigned int nSamplesPerSec;  /* sampling rate in samples per second */
  unsigned int nAvgBytesPerSec; /* average number of bytes per second */
  unsigned short int nBlockAlign;
  unsigned short int nBitsPerSample; /* # of bits/sample. Typically 16 */
  unsigned int bytesOfData;    /* how many bytes of data present */
  unsigned int samplesOfData;  /* how many samples of data */
};
typedef struct WAVFileInfo WAVFileInfo;


/* function prototypes */
WAVFileInfo open_wav(char *infile);
int close_wav(WAVFileInfo w);
GenericArray read_wav(WAVFileInfo w, unsigned int numSamples);
int write_wav(WAVFileInfo w, void *buf, char *filename);
```

---------- File: read.c ----------

```
/*************************************************************************
 *  Read.c   - some easy-to-use WAV reading functions                    *
 *                                                                       *
 *  Usage: First call open_wav(<filename>) to get a WAVFileInfo          *
 * struct. Then use this struct as the argument (along with number of *
 * samples to read) to read_wav(<WAVFileInfo struct>, <numSamples>).  *
 * When done after some arbitrary number of read_wav() calls, use     *
 * close_wav() on the struct.                                          *
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "arrays.h"
#include "read.h"


/* open a wav file so it can be read. Return structure
   with some basic info about the file.
 */
WAVFileInfo open_wav(char *infile)
{
  FILE *fi;
  unsigned int temp;

  /* header information variables */
  char chunk_name[4];
  unsigned int chunk_size;

  /* output struct */
  WAVFileInfo wav_info;



  /* open file */
  fi = fopen(infile,"rb");
  wav_info.fi = fi;
  if (fi == NULL)
  {
    printf("ERROR: Can not open %s\n",infile);
    exit(1);
  }
  wav_info.isOpen = calloc(1, sizeof(int));
  *(wav_info.isOpen) = 1;

  /******************* READ HEADER INFO ************************/

  /* RIFF header */
  temp = fread( (void *) chunk_name, sizeof(char), 4, fi); /* RIFF id */
  if(temp != 4)
  {
    printf("Error: Failed to read file header (name field)\n");
      printf("fread returned: %i\n", temp);
      if(ferror(fi)) perror("File error ");
      if(feof(fi)) printf("End of file\n");
```

```
  exit(1);
}

if(strncmp(chunk_name, "RIFF", 4) != 0)
{
  printf("Error: Input file is not a valid WAV/RIFF file!\n");
  exit(1);
}

temp = fread( (void *) &chunk_size, 4, 1, fi);
if(temp != 1)
{
    printf("Error: Failed to read RIFF header residual length\n");
    printf("fread returned: %i\n", temp);
    if(ferror(fi)) perror("File error ");
    if(feof(fi)) printf("End of file\n");
    exit(1);
}


temp = fread( (void *) chunk_name, sizeof(char), 4, fi); /* wave_id */
if(temp != 4)
{
  printf("Error: Failed to read file header (wave_id field)\n");
    printf("fread returned: %i\n", temp);
    if(ferror(fi)) perror("File error ");
    if(feof(fi)) printf("End of file\n");
  exit(1);
}
if(strncmp(chunk_name, "WAVE", 4) != 0)
{
  printf("Error: Input file is not a WAVE file.\n");
    printf("Chunk_name:%s\n", chunk_name);
  exit(1);
}


/* RIFF chunks ... arbitrarily many before the "data" chunk */
while(1)
{
  /* get chunk name and size (each is 4 bytes) */
  temp = fread( (void *) chunk_name, sizeof(char), 4, fi);
  if(temp != 4)
  {
    printf("Error: Failed to read RIFF chunk name.\n");
      printf("fread returned: %i\n", temp);
      if(ferror(fi)) perror("File error ");
      if(feof(fi)) printf("End of file\n");
      exit(1);
  }
  temp = fread( &chunk_size, sizeof(unsigned int), 1, fi);
  if(temp != 1)
  {
    printf("Error: Failed to read RIFF chunk size.\n");
    exit(1);
  }
```

```
    /* first check if this is the format (fmt) chunk */
    if(strncmp(chunk_name, "fmt", 3) == 0)
    {
      /* read file format information */
      temp = fread( &(wav_info.wFormatTag), sizeof(unsigned short int), 1,
fi);
      if(temp != 1)
      {
            printf("Error: Failed to read wFormatTag\n");
            exit(1);
      }
      if(wav_info.wFormatTag != 0x01)
      {
            printf("Error: File not in standard WAV/PCM format!\n");
            exit(1);
      }
        temp = fread( &(wav_info.nChannels), sizeof(unsigned short int), 1,
fi);
      if(temp != 1)
      {
            printf("Error: Failed to read nChannels\n");
            exit(1);
      }
      temp = fread( &(wav_info.nSamplesPerSec), sizeof(unsigned int), 1, fi);
      if(temp != 1)
      {
            printf("Error: Failed to read nSamplesPerSec\n");
            exit(1);
      }
      temp = fread( &(wav_info.nAvgBytesPerSec), sizeof(unsigned int), 1,
fi);
      if(temp != 1)
      {
            printf("Error: Failed to read nAvgBytesPerSec\n");
            exit(1);
      }
      temp = fread( &(wav_info.nBlockAlign), sizeof(unsigned short int), 1,
fi);
      if(temp != 1)
      {
            printf("Error: Failed to read nBlockAlign\n");
            exit(1);
      }
      temp = fread( &(wav_info.nBitsPerSample), sizeof(unsigned short int),
1, fi);
      if(temp != 1)
      {
            printf("Error: Failed to read nBitsPerSample\n");
            exit(1);
      }
      /* done reading fmt chunk */
      continue;
    }

    /* next check for data chunk */
    if(strncmp(chunk_name, "data", 4) != 0)
    {
```

```
      /* then we have some other (non-fmt, non-data) RIFF chunk
       here, not "standard" per se but allowed by the specification.
       So just ignore the extension chunk, seek past it, and go back
       to beginning of loop to see if the next chunk is the data
      */
      fseek(fi, chunk_size, SEEK_CUR);
      continue;
    }
    else
    {
      /* we're to the last chunk - the data. Chunk_size therefore is the
       length in bytes of the remaining data in the file. Exit this loop
       and move on to the audio-data-reading code in read_wav().
      */
      wav_info.bytesOfData = chunk_size;
      wav_info.samplesOfData = chunk_size * 8/wav_info.nBitsPerSample;
      break;
    }
  } /* end while(1) */



  return wav_info;
}






int close_wav(WAVFileInfo w)
{
  *(w.isOpen) = 0;
  return fclose(w.fi);
}



/* read_wav() -- read numSamples of the input file.

   Returns GenericArray with length set to the number of samples
   actually read from the file. Check that the length of the returned
   array is the same as the number of samples requested - if not,
   it likely means that we've reached the end of file. (Caller
   should check this).

   For now, the numSamples is the total amount read (so if there
   are 2 channels in the file, there are numSamples/2 read from
   each channel into one output array.  We can change that behavior
   fairly easily later if necessary.
*/
GenericArray read_wav(WAVFileInfo w, unsigned int numSamples)
{
  GenericArray  thearray; /* array of samples read from file */
  thearray.bytesPerElement = w.nBitsPerSample / 8;
  thearray.data = (void *) calloc(numSamples, thearray.bytesPerElement);
```

```
  /*  Read num_samples of input data */
  thearray.numElements = fread(thearray.data, thearray.bytesPerElement,
numSamples, w.fi);
  thearray.numBytes = thearray.numElements * thearray.bytesPerElement;

  return thearray;
}


/* write WAV file to filename. buf should contain data for all channels */
int write_wav(WAVFileInfo w, void *buf, char* filename)
{
  FILE *fi;
  char temp[4];
  unsigned int itemp;

  fi = fopen(filename, "wb");
  if(fi == NULL)
  {
    printf("Couldn't open %s for writing \n", filename);
    exit(1);
  }


  /****** write WAV header *********/

  /* RIFF chunk */
  temp[0] = 'R';  temp[1] = 'I'; temp[2] = 'F'; temp[3] = 'F';
  fwrite(temp, sizeof(char), 4, fi);
  itemp = 36 + w.bytesOfData;
  fwrite(&itemp, sizeof(int), 1, fi);
  temp[0] = 'W'; temp[1] = 'A'; temp[2] = 'V'; temp[3] = 'E';
  fwrite(temp, sizeof(char), 4, fi);

  /* fmt chunk */
  temp[0] = 'f'; temp[1] = 'm'; temp[2] = 't'; temp[3] = ' ';
  fwrite(temp, sizeof(char), 4, fi);
  itemp = 16;
  fwrite(&itemp, sizeof(int), 1, fi);
  fwrite(&(w.wFormatTag), sizeof(unsigned short int), 1, fi);
  fwrite(&(w.nChannels), sizeof(unsigned short int), 1, fi);
  fwrite(&(w.nSamplesPerSec), sizeof(unsigned int), 1, fi);
  fwrite(&(w.nAvgBytesPerSec), sizeof(unsigned int), 1, fi);
  fwrite(&(w.nBlockAlign), sizeof(unsigned short int), 1, fi);
  fwrite(&(w.nBitsPerSample), sizeof(unsigned short int), 1, fi);

  /* data chunk */
  temp[0] = 'd'; temp[1] = 'a'; temp[2] = 't'; temp[3] = 'a';
  fwrite(temp, sizeof(char), 4, fi);
  fwrite(&(w.bytesOfData), sizeof(unsigned int), 1, fi);

  /******* write data *******/
  fwrite(buf, 1, w.bytesOfData, fi);

  fclose(fi);
  return 0;
}
```

Appendix C – GUI Code and Sample Parameter File

------ File: **parameters.txt** (with default parameters) ------

```
D:\\Group4Data\\CityDemo\\L.wav
D:\\Group4Data\\CityDemo\\C.wav
D:\\Group4Data\\CityDemo\\R.wav
D:\\Group4Data\\CityDemo\\RS.wav
D:\\Group4Data\\CityDemo\\LS.wav
D:\\Group4Data\\CityDemo\\LFE.wav
D:\\Group4Data\\HRTFs\\elev0\\L0e330a.wav
D:\\Group4Data\\HRTFs\\elev0\\L0e030a.wav
D:\\Group4Data\\HRTFs\\elev0\\L0e000a.wav
D:\\Group4Data\\HRTFs\\elev0\\L0e000a.wav
D:\\Group4Data\\HRTFs\\elev0\\L0e030a.wav
D:\\Group4Data\\HRTFs\\elev0\\L0e330a.wav
D:\\Group4Data\\HRTFs\\elev0\\L0e120a.wav
D:\\Group4Data\\HRTFs\\elev0\\L0e240a.wav
D:\\Group4Data\\HRTFs\\elev0\\L0e240a.wav
D:\\Group4Data\\HRTFs\\elev0\\L0e120a.wav
D:\\Group4Data\\HRTFs\\elev0\\L0e000a.wav
D:\\Group4Data\\HRTFs\\elev0\\L0e000a.wav
10
9
10
13
13
9
330
0
30
120
240
0
0
0
0
0
0
0
70
Dolby
```

------ File: **paramkey.txt** (explanatory key to parameters.txt) ------

```
1   LeftFront                    | Demo Channel WAV files (with full path)
2   Center                       |
3   RightFront                   |
4   RightSurround                |
5   LeftSurround                 |
6   LowFrequencyEffects          |
7   LeftFront/LeftEar             | HRTF WAV files (with full path)
8   LeftFront/RightEar            |
9   Center/LeftEar                |
10  Center/RightEar               |
11  RightFront/LeftEar            |
12  RightFront/RightEar           |
13  RightSurround/LeftEar         |
14  RightSurround/RightEar        |
15  LeftSurround/LeftEar          |
16  LeftSurround/RightEar         |
17  LowFrequencyEffects/LeftEar   |
18  LowFrequencyEffects/RightEar  |
19  LeftFront                    | Speaker Distances (in feet)
20  Center                       |
21  RightFront                   |
22  RightSurround                |
23  LeftSurround                 |
24  LowFrequencyEffects          |
25  LeftFront                   | Speaker Horizontal Angles (in degrees)
26  Center                      |
27  RightFront                  |
28  RightSurround               |
29  LeftSurround                |
30  LowFrequencyEffects         |
31  LeftFront                    | Speaker Elevation Angles (in degrees)
32  Center                       |
33  RightFront                   |
34  RightSurround                |
35  LeftSurround                 |
36  LowFrequencyEffects          |
37  Temperature (in degrees Fahrenheit)
38  Play Mode (Dolby or stereo)

(Line numbers are of course not included in parameters.txt)
```

------ File: **default.htm** ------

```
<html xmlns:v="urn:schemas-microsoft-com:vml" xmlns:o="urn:schemas-microsoft-
com:office:office" xmlns="http://www.w3.org/TR/REC-html40">

<head>
<meta name="GENERATOR" content="Microsoft FrontPage 5.0">
<meta name="AUTHOR" content="Lincoln Westfall">
<meta name="ProgId" content="FrontPage.Editor.Document">
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>551 Group 4 - Dolby Digital 5.1 Emulation GUI</title>
</head>

<body>

<form method="post" name="SpeakerData" action="formproc.asp">

<IMG alt="" border=0 height=693 hspace=0 src="background.gif" style="HEIGHT:
693px; LEFT: 0px; POSITION: absolute; TOP: 0px; WIDTH: 689px" useMap=""
width=689 > <!-- Buttons --!>
<input type="submit" value="Submit" name="SubmitBut" style="LEFT: 283px;
POSITION: absolute; TOP: 570px" tabindex="0">

<input type="reset" value="Reset" name="ResetBut" style="LEFT: 358px;
POSITION: absolute; TOP: 569px">
<P></P><!-- General Options --!>
<select size="1" name="PlayMode" style="LEFT: 288px; POSITION: absolute; TOP:
485px">
<option value="stereo">Stereo</option>
<option selected value="Dolby">Dolby Digital 5.1</option>
</select>

<select size="1" name="Demo" style="LEFT: 294px; POSITION: absolute; TOP:
513px">
<option value="CanyonDemo">Canyon Demo</option>
<option selected value="CityDemo">City Demo</option>
<option value="EgyptDemo">Egypt Demo</option>
<option value="TrainDemo">Train Demo</option>
</select>

<select size="1" name="EarType" style="LEFT: 498px; POSITION: absolute; TOP:
313px">
<option selected value="L">Normal</option>
<option value="R">Large Red</option>
</select>

<SELECT name=Temperature size=1 style="LEFT: 485px; POSITION: absolute; TOP:
371px">
<OPTION value ="32">32 degrees F</OPTION>
<OPTION value="40">40 degrees F</OPTION>
<OPTION value="50">50 degrees F</OPTION>
<OPTION value="60">60 degrees F</OPTION>
<OPTION selected value="70">70 degrees F</OPTION>
<OPTION value="80">80 degrees F</OPTION>
<OPTION value="90">90 degrees F</OPTION>
<OPTION value="100">100 degrees F</OPTION>
```

```
<OPTION value="110">110 degrees F</OPTION>
<OPTION value="120">120 degrees F</OPTION>
</SELECT>

<!-- Speaker Horizontal Angles (radio buttons) --!>
<input type="radio" value="300" name="LeftAng" style="LEFT: 69px; POSITION:
absolute; TOP: 185px"><input type="radio" value="275" name="LeftAng"
style="LEFT: 29px; POSITION: absolute; TOP: 313px"><input type="radio"
value="345" name="LeftAng" style="LEFT: 256px; POSITION: absolute; TOP:
40px"><input type="radio" value="340" name="LeftAng" style="LEFT: 231px;
POSITION: absolute; TOP: 48px"><input type="radio" value="335" name="LeftAng"
style="LEFT: 207px; POSITION: absolute; TOP: 59px"><input type="radio"
value="330" name="LeftAng" style="LEFT: 182px; POSITION: absolute; TOP: 71px"
checked><input type="radio" value="325" name="LeftAng" style="LEFT: 159px;
POSITION: absolute; TOP: 86px"><input type="radio" value="320" name="LeftAng"
style="LEFT: 139px; POSITION: absolute; TOP: 102px"><input type="radio"
value="315" name="LeftAng" style="LEFT: 119px; POSITION: absolute; TOP:
120px"><input type="radio" value="310" name="LeftAng" style="LEFT: 100px;
POSITION: absolute; TOP: 141px"><input type="radio" value="295"
name="LeftAng" style="LEFT: 57px; POSITION: absolute; TOP: 209px"><input
type="radio" value="290" name="LeftAng" style="LEFT: 47px; POSITION:
absolute; TOP: 234px"><input type="radio" value="285" name="LeftAng"
style="LEFT: 39px; POSITION: absolute; TOP: 259px"><input type="radio"
value="280" name="LeftAng" style="LEFT: 33px; POSITION: absolute; TOP:
285px"><input type="radio" value="305" name="LeftAng" style="LEFT: 83px;
POSITION: absolute; TOP: 162px"><input type="radio" value="350"
name="CenterAng" style="LEFT: 282px; POSITION: absolute; TOP: 34px"><input
type="radio" value="355" name="CenterAng" style="LEFT: 309px; POSITION:
absolute; TOP: 31px"><input type="radio" value="000" name="CenterAng"
style="LEFT: 336px; POSITION: absolute; TOP: 30px" checked><input
type="radio" value="005" name="CenterAng" style="LEFT: 362px; POSITION:
absolute; TOP: 31px"><input type="radio" value="010" name="CenterAng"
style="LEFT: 389px; POSITION: absolute; TOP: 35px"><input type="radio"
value="085" name="RightAng" style="LEFT: 642px; POSITION: absolute; TOP:
311px"><input type="radio" value="015" name="RightAng" style="LEFT: 416px;
POSITION: absolute; TOP: 41px"><input type="radio" value="020"
name="RightAng" style="LEFT: 441px; POSITION: absolute; TOP: 49px"><input
type="radio" value="025" name="RightAng" style="LEFT: 466px; POSITION:
absolute; TOP: 60px"><input type="radio" value="030" name="RightAng"
style="LEFT: 490px; POSITION: absolute; TOP: 72px" checked
class="RightAng"><input type="radio" value="035" name="RightAng" style="LEFT:
513px; POSITION: absolute; TOP: 87px"><input type="radio" value="040"
name="RightAng" style="LEFT: 534px; POSITION: absolute; TOP: 103px"><input
type="radio" value="045" name="RightAng" style="LEFT: 553px; POSITION:
absolute; TOP: 121px"><input type="radio" value="050" name="RightAng"
style="LEFT: 572px; POSITION: absolute; TOP: 142px"><input type="radio"
value="055" name="RightAng" style="LEFT: 587px; POSITION: absolute; TOP:
163px"><input type="radio" value="060" name="RightAng" style="LEFT: 602px;
POSITION: absolute; TOP: 186px"><input type="radio" value="065"
name="RightAng" style="LEFT: 615px; POSITION: absolute; TOP: 209px"><input
type="radio" value="070" name="RightAng" style="LEFT: 625px; POSITION:
absolute; TOP: 235px"><input type="radio" value="075" name="RightAng"
style="LEFT: 633px; POSITION: absolute; TOP: 260px"><input type="radio"
value="080" name="RightAng" style="LEFT: 638px; POSITION: absolute; TOP:
285px"><input type="radio" value="240" name="LeftSurroundAng" style="LEFT:
70px; POSITION: absolute; TOP: 493px" checked><input type="radio" value="180"
name="LeftSurroundAng" style="LEFT: 330px; POSITION: absolute; TOP:
```

```
647px"><input type="radio" value="185" name="LeftSurroundAng" style="LEFT:
308px; POSITION: absolute; TOP: 645px"><input type="radio" value="190"
name="LeftSurroundAng" style="LEFT: 282px; POSITION: absolute; TOP:
641px"><input type="radio" value="195" name="LeftSurroundAng" style="LEFT:
256px; POSITION: absolute; TOP: 635px"><input type="radio" value="200"
name="LeftSurroundAng" style="LEFT: 230px; POSITION: absolute; TOP:
627px"><input type="radio" value="235" name="LeftSurroundAng" style="LEFT:
84px; POSITION: absolute; TOP: 515px"><input type="radio" value="210"
name="LeftSurroundAng" style="LEFT: 182px; POSITION: absolute; TOP:
605px"><input type="radio" value="215" name="LeftSurroundAng" style="LEFT:
158px; POSITION: absolute; TOP: 590px"><input type="radio" value="220"
name="LeftSurroundAng" style="LEFT: 139px; POSITION: absolute; TOP:
575px"><input type="radio" value="225" name="LeftSurroundAng" style="LEFT:
118px; POSITION: absolute; TOP: 556px"><input type="radio" value="230"
name="LeftSurroundAng" style="LEFT: 101px; POSITION: absolute; TOP:
536px"><input type="radio" value="205" name="LeftSurroundAng" style="LEFT:
206px; POSITION: absolute; TOP: 617px"><input type="radio" value="245"
name="LeftSurroundAng" style="LEFT: 57px; POSITION: absolute; TOP:
468px"><input type="radio" value="250" name="LeftSurroundAng" style="LEFT:
47px; POSITION: absolute; TOP: 445px"><input type="radio" value="255"
name="LeftSurroundAng" style="LEFT: 39px; POSITION: absolute; TOP:
418px"><input type="radio" value="260" name="LeftSurroundAng" style="LEFT:
33px; POSITION: absolute; TOP: 392px"><input type="radio" value="265"
name="LeftSurroundAng" style="LEFT: 29px; POSITION: absolute; TOP:
365px"><input type="radio" value="270" name="LeftSurroundAng" style="LEFT:
28px; POSITION: absolute; TOP: 339px"><input type="radio" value="125"
name="RightSurroundAng" style="LEFT: 587px; POSITION: absolute; TOP:
516px"><input type="radio" value="175" name="RightSurroundAng" style="LEFT:
363px; POSITION: absolute; TOP: 645px"><input type="radio" value="180"
name="RightSurroundAng" style="LEFT: 341px; POSITION: absolute; TOP:
647px"><input type="radio" value="090" name="RightSurroundAng" style="LEFT:
643px; POSITION: absolute; TOP: 340px"><input type="radio" value="095"
name="RightSurroundAng" style="LEFT: 641px; POSITION: absolute; TOP:
366px"><input type="radio" value="100" name="RightSurroundAng" style="LEFT:
638px; POSITION: absolute; TOP: 392px"><input type="radio" value="105"
name="RightSurroundAng" style="LEFT: 632px; POSITION: absolute; TOP:
418px"><input type="radio" value="110" name="RightSurroundAng" style="LEFT:
624px; POSITION: absolute; TOP: 443px"><input type="radio" value="115"
name="RightSurroundAng" style="LEFT: 614px; POSITION: absolute; TOP:
469px"><input type="radio" value="120" name="RightSurroundAng" style="LEFT:
602px; POSITION: absolute; TOP: 492px" checked><input type="radio"
value="130" name="RightSurroundAng" style="LEFT: 571px; POSITION: absolute;
TOP: 536px"><input type="radio" value="135" name="RightSurroundAng"
style="LEFT: 552px; POSITION: absolute; TOP: 556px"><input type="radio"
value="140" name="RightSurroundAng" style="LEFT: 534px; POSITION: absolute;
TOP: 574px"><input type="radio" value="145" name="RightSurroundAng"
style="LEFT: 513px; POSITION: absolute; TOP: 591px"><input type="radio"
value="150" name="RightSurroundAng" style="LEFT: 490px; POSITION: absolute;
TOP: 604px"><input type="radio" value="155" name="RightSurroundAng"
style="LEFT: 466px; POSITION: absolute; TOP: 617px"><input type="radio"
value="160" name="RightSurroundAng" style="LEFT: 441px; POSITION: absolute;
TOP: 627px"><input type="radio" value="165" name="RightSurroundAng"
style="LEFT: 415px; POSITION: absolute; TOP: 635px"><input type="radio"
value="170" name="RightSurroundAng" style="LEFT: 389px; POSITION: absolute;
TOP: 641px">
<select size="1" name="LFEAng" style="LEFT: 97px; POSITION: absolute; TOP:
340px">
```

```
<option value="000" selected>Horizontal Angle 0</option>
<option value="000">0 (straight ahead)</option>
<option value="090">90 (to right)</option>
<option value="180">180 (straight back)</option>
<OPTION value="270">270 (to left)</OPTION>
</select>

<!-- Speaker Distances --!>
<select size="1" name="LeftDist" style="LEFT: 125px; POSITION: absolute; TOP:
187px">
<option value="1">1 ft. away</option>
<option value="2">2 ft. away</option>
<option value="3">3 ft. away</option>
<option value="4">4 ft. away</option>
<option value="5">5 ft. away</option>
<option value="6">6 ft. away</option>
<option value="7">7 ft. away</option>
<option value="8">8 ft. away</option>
<option value ="9">9 ft. away</option>
<option value="10" selected>10 ft. away</option>
<option value="11">11 ft. away</option>
<option value="12">12 ft. away</option>
<option value="13">13 ft. away</option>
<option value="14">14 ft. away</option>
<option value="15">15 ft. away</option>
<option value="16">16 ft. away</option>
<option value="17">17 ft. away</option>
<option value="18">18 ft. away</option>
<option value="19">19 ft. away</option>
<option value="20">20 ft. away</option>
</select>

<select size="1" name="CenterDist" style="LEFT: 306px; POSITION: absolute;
TOP: 84px">
<option value="1">1 ft. away</option>
<option value="2">2 ft. away</option>
<option value="3">3 ft. away</option>
<option value="4">4 ft. away</option>
<option value="5">5 ft. away</option>
<option value="6">6 ft. away</option>
<option value="7">7 ft. away</option>
<option value="8">8 ft. away</option>
<option selected value="9">9 ft. away</option>
<option value="10">10 ft. away</option>
<option value="11">11 ft. away</option>
<option value="12">12 ft. away</option>
<option value="13">13 ft. away</option>
<option value="14">14 ft. away</option>
<option value="15">15 ft. away</option>
<option value="16">16 ft. away</option>
<option value="17">17 ft. away</option>
<option value="18">18 ft. away</option>
<option value="19">19 ft. away</option>
<option value="20">20 ft. away</option>
</select>
```

```
<select size="1" name="RightDist" style="LEFT: 482px; POSITION: absolute;
TOP: 187px">
<option value="1">1 ft. away</option>
<option value="2">2 ft. away</option>
<option value="3">3 ft. away</option>
<option value="4">4 ft. away</option>
<option value="5">5 ft. away</option>
<option value="6">6 ft. away</option>
<option value="7">7 ft. away</option>
<option value="8">8 ft. away</option>
<option value ="9">9 ft. away</option>
<option value="10" selected>10 ft. away</option>
<option value="11">11 ft. away</option>
<option value="12">12 ft. away</option>
<option value="13">13 ft. away</option>
<option value="14">14 ft. away</option>
<option value="15">15 ft. away</option>
<option value="16">16 ft. away</option>
<option value="17">17 ft. away</option>
<option value="18">18 ft. away</option>
<option value="19">19 ft. away</option>
<option value="20">20 ft. away</option>
</select>

<select size="1" name="LeftSurroundDist" style="LEFT: 123px; POSITION:
absolute; TOP: 485px">
<option value="1">1 ft. away</option>
<option value="2">2 ft. away</option>
<option value="3" selected>3 ft. away</option>
<option value="4">4 ft. away</option>
<option value="5">5 ft. away</option>
<option value="6">6 ft. away</option>
<option value="7">7 ft. away</option>
<option value="8">8 ft. away</option>
<option value="9">9 ft. away</option>
<option value="10">10 ft. away</option>
<option value="11">11 ft. away</option>
<option value="12">12 ft. away</option>
<option value="13">13 ft. away</option>
<option value="14">14 ft. away</option>
<option value="15">15 ft. away</option>
<option value="16">16 ft. away</option>
<option value="17">17 ft. away</option>
<option value="18">18 ft. away</option>
<option value="19">19 ft. away</option>
<option value="20">20 ft. away</option>
</select>

<select size="1" name="RightSurroundDist" style="LEFT: 482px; POSITION:
absolute; TOP: 485px">
<option value="1">1 ft. away</option>
<option value="2">2 ft. away</option>
<option value="3" selected>3 ft. away</option>
<option value="4">4 ft. away</option>
<option value="5">5 ft. away</option>
<option value="6">6 ft. away</option>
<option value="7">7 ft. away</option>
```

```
<option value="8">8 ft. away</option>
<option value="9">9 ft. away</option>
<option value="10">10 ft. away</option>
<option value="11">11 ft. away</option>
<option value="12">12 ft. away</option>
<option value="13">13 ft. away</option>
<option value="14">14 ft. away</option>
<option value="15">15 ft. away</option>
<option value="16">16 ft. away</option>
<option value="17">17 ft. away</option>
<option value="18">18 ft. away</option>
<option value="19">19 ft. away</option>
<option value="20">20 ft. away</option>
</select>

<select size="1" name="LFEDist" style="LEFT: 119px; POSITION: absolute; TOP:
311px">
<option value="1">1 ft. away</option>
<option value="2">2 ft. away</option>
<option value="3">3 ft. away</option>
<option value="4">4 ft. away</option>
<option value="5">5 ft. away</option>
<option value="6">6 ft. away</option>
<option value="7">7 ft. away</option>
<option value="8">8 ft. away</option>
<option selected value="9">9 ft. away</option>
<option value="10">10 ft. away</option>
<option value="11">11 ft. away</option>
<option value="12">12 ft. away</option>
<option value="13">13 ft. away</option>
<option value="14">14 ft. away</option>
<option value="15">15 ft. away</option>
<option value="16">16 ft. away</option>
<option value="17">17 ft. away</option>
<option value="18">18 ft. away</option>
<option value="19">19 ft. away</option>
<option value="20">20 ft. away</option>

</select> <!-- Speaker Elevation Angles --!>
<SELECT name=LeftElev size=1 style="LEFT: 102px; POSITION: absolute; TOP:
215px">
<OPTION value="0" selected>Elevation Angle 0</OPTION>
<OPTION value=-40>-40 (down)</OPTION>
<OPTION value=-30>-30 (down)</OPTION>
<OPTION value=-20>-20 (down)</OPTION>
<OPTION value=-10>-10 (down)</OPTION>
<OPTION value="0">0 (level)</OPTION>
<OPTION value=10>10 (up)</OPTION>
<OPTION value=20>20 (up)</OPTION>
<OPTION value=30>30 (up)</OPTION>
<OPTION value=40>40 (up)</OPTION>
</SELECT>

<SELECT name=CenterElev size=1 style="LEFT: 284px; POSITION: absolute; TOP:
112px">
<OPTION value="0" selected>Elevation Angle 0</OPTION>
<OPTION value=-40>-40 (down)</OPTION>
```

```
<OPTION value=-30>-30 (down)</OPTION>
<OPTION value=-20>-20 (down)</OPTION>
<OPTION value=-10>-10 (down)</OPTION>
<OPTION value="0">0 (level)</OPTION>
<OPTION value=10>10 (up)</OPTION>
<OPTION value=20>20 (up)</OPTION>
<OPTION value=30>30 (up)</OPTION>
<OPTION value=40>40 (up)</OPTION>
</SELECT>

<SELECT name=RightElev size=1 style="LEFT: 461px; POSITION: absolute; TOP:
215px">
<OPTION value="0" selected>Elevation Angle 0</OPTION>
<OPTION value=-40>-40 (down)</OPTION>
<OPTION value=-30>-30 (down)</OPTION>
<OPTION value=-20>-20 (down)</OPTION>
<OPTION value=-10>-10 (down)</OPTION>
<OPTION value="0">0 (level)</OPTION>
<OPTION value=10>10 (up)</OPTION>
<OPTION value=20>20 (up)</OPTION>
<OPTION value=30>30 (up)</OPTION>
<OPTION value=40>40 (up)</OPTION>
</SELECT>

<SELECT name=LeftSurroundElev size=1 style="LEFT: 141px; POSITION: absolute;
TOP: 513px">
<OPTION value="0" selected>Elevation Angle 0</OPTION>
<OPTION value=-40>-40 (down)</OPTION>
<OPTION value=-30>-30 (down)</OPTION>
<OPTION value=-20>-20 (down)</OPTION>
<OPTION value=-10>-10 (down)</OPTION>
<OPTION value="0">0 (level)</OPTION>
<OPTION value=10>10 (up)</OPTION>
<OPTION value=20>20 (up)</OPTION>
<OPTION value=30>30 (up)</OPTION>
<OPTION value=40>40 (up)</OPTION>
</SELECT>

<SELECT name=RightSurroundElev size=1 style="LEFT: 418px; POSITION: absolute;
TOP: 513px">
<OPTION value="0" selected>Elevation Angle 0</OPTION>
<OPTION value=-40>-40 (down)</OPTION>
<OPTION value=-30>-30 (down)</OPTION>
<OPTION value=-20>-20 (down)</OPTION>
<OPTION value=-10>-10 (down)</OPTION>
<OPTION value="0">0 (level)</OPTION>
<OPTION value=10>10 (up)</OPTION>
<OPTION value=20>20 (up)</OPTION>
<OPTION value=30>30 (up)</OPTION>
<OPTION value=40>40 (up)</OPTION>
</SELECT>

<SELECT name=LFEElev size=1 style="LEFT: 98px; POSITION: absolute; TOP:
369px">
<OPTION value="0" selected>Elevation Angle 0</OPTION>
<OPTION value=-40>-40 (down)</OPTION>
<OPTION value=-30>-30 (down)</OPTION>
```

```
<OPTION value=-20>-20 (down)</OPTION>
<OPTION value=-10>-10 (down)</OPTION>
<OPTION value="0">0 (level)</OPTION>
<OPTION value=10>10 (up)</OPTION>
<OPTION value=20>20 (up)</OPTION>
<OPTION value=30>30 (up)</OPTION>
<OPTION value=40>40 (up)</OPTION>
</SELECT>

</form>

</body>

</html>
```

------ File: **formproc.asp** (called by default.htm) ------

```
<HTML>
<HEAD>
<TITLE>Write Parameters</TITLE>
</HEAD>
<BODY>
Writing your selections to parameters.txt ...
<%
Dim fso
Set fso = Server.CreateObject("Scripting.FileSystemObject")

If fso.FileExists("d:\wwwroot\inuse.txt") then
  Set fso = Nothing
%>
<P>
... UNSUCCESSFUL - parameters.txt is currently being accessed! (inuse.txt
found in wwwroot directory)
<P>
<P>**********************************************************************
***
<P>Please hit Back and Submit again.
<P>

<%
Else
  set InUseFile = fso.CreateTextFile("d:\wwwroot\inuse.txt", true)
    'true here means that file will be overwritten if it already exists.
  InUseFile.WriteLine("Data I/O with parameters.txt is in progress.")
  InUseFile.Close

'Declare variables
  Dim PlayMode, Demo, EarType, Temperature
  Dim LeftAng, CenterAng, RightAng, LeftSurroundAng, RightSurroundAng, LFEAng
  Dim LeftAng2, CenterAng2, RightAng2, LeftSurroundAng2, RightSurroundAng2,
LFEAng2
  Dim LeftDist, CenterDist, RightDist, LeftSurroundDist, RightSurroundDist,
LFEDist
  Dim LeftElev, CenterElev, RightElev, LeftSurroundElev, RightSurroundElev,
LFEElev

'Put form data into variables
  PlayMode = Request.Form("PlayMode")
  Demo = Request.Form("Demo")
  EarType = Request.Form("EarType")
  Temperature = Request.Form("Temperature")

  LeftElev = Request.Form("LeftElev")
  CenterElev = Request.Form("CenterElev")
  RightElev = Request.Form("RightElev")
  LeftSurroundElev = Request.Form("LeftSurroundElev")
  RightSurroundElev = Request.Form("RightSurroundElev")
  LFEElev = Request.Form("LFEElev")

  LeftAng = Request.Form("LeftAng")
  CenterAng = Request.Form("CenterAng")
```

```
  RightAng = Request.Form("RightAng")
  LeftSurroundAng = Request.Form("LeftSurroundAng")
  RightSurroundAng = Request.Form("RightSurroundAng")
  LFEAng = Request.Form("LFEAng")

  LeftAng2 = 360 - LeftAng
  CenterAng2 = 360 - CenterAng
  RightAng2 = 360 - RightAng
  LeftSurroundAng2 = 360 - LeftSurroundAng
  RightSurroundAng2 = 360 - RightSurroundAng
  LFEAng2 = 360 - LFEAng

  LeftDist = Request.Form("LeftDist")
  CenterDist = Request.Form("CenterDist")
  RightDist = Request.Form("RightDist")
  LeftSurroundDist = Request.Form("LeftSurroundDist")
  RightSurroundDist = Request.Form("RightSurroundDist")
  LFEDist = Request.Form("LFEDist")

'Pad 2nd set of angles to have three digits for proper filenames
  If LeftAng2 = 360 then LeftAng2 = "000"
  If LeftAng2 = 5 then LeftAng2 = "005"
  If LeftAng2 = 10 then LeftAng2 = "010"
  If LeftAng2 = 20 then LeftAng2 = "020"
  If LeftAng2 = 30 then LeftAng2 = "030"
  If LeftAng2 = 40 then LeftAng2 = "040"
  If LeftAng2 = 50 then LeftAng2 = "050"
  If LeftAng2 = 60 then LeftAng2 = "060"
  If LeftAng2 = 70 then LeftAng2 = "070"
  If LeftAng2 = 80 then LeftAng2 = "080"
  If LeftAng2 = 90 then LeftAng2 = "090"

  If CenterAng2 = 360 then CenterAng2 = "000"
  If CenterAng2 = 5 then CenterAng2 = "005"
  If CenterAng2 = 10 then CenterAng2 = "010"
  If CenterAng2 = 20 then CenterAng2 = "020"
  If CenterAng2 = 30 then CenterAng2 = "030"
  If CenterAng2 = 40 then CenterAng2 = "040"
  If CenterAng2 = 50 then CenterAng2 = "050"
  If CenterAng2 = 60 then CenterAng2 = "060"
  If CenterAng2 = 70 then CenterAng2 = "070"
  If CenterAng2 = 80 then CenterAng2 = "080"
  If CenterAng2 = 90 then CenterAng2 = "090"

  If RightAng2 = 360 then RightAng2 = "000"
  If RightAng2 = 5 then RightAng2 = "005"
  If RightAng2 = 10 then RightAng2 = "010"
  If RightAng2 = 20 then RightAng2 = "020"
  If RightAng2 = 30 then RightAng2 = "030"
  If RightAng2 = 40 then RightAng2 = "040"
  If RightAng2 = 50 then RightAng2 = "050"
  If RightAng2 = 60 then RightAng2 = "060"
  If RightAng2 = 70 then RightAng2 = "070"
  If RightAng2 = 80 then RightAng2 = "080"
  If RightAng2 = 90 then RightAng2 = "090"

  If LeftSurroundAng2 = 360 then LeftSurroundAng2 = "000"
```

```
  If LeftSurroundAng2 = 5 then LeftSurroundAng2 = "005"
  If LeftSurroundAng2 = 10 then LeftSurroundAng2 = "010"
  If LeftSurroundAng2 = 20 then LeftSurroundAng2 = "020"
  If LeftSurroundAng2 = 30 then LeftSurroundAng2 = "030"
  If LeftSurroundAng2 = 40 then LeftSurroundAng2 = "040"
  If LeftSurroundAng2 = 50 then LeftSurroundAng2 = "050"
  If LeftSurroundAng2 = 60 then LeftSurroundAng2 = "060"
  If LeftSurroundAng2 = 70 then LeftSurroundAng2 = "070"
  If LeftSurroundAng2 = 80 then LeftSurroundAng2 = "080"
  If LeftSurroundAng2 = 90 then LeftSurroundAng2 = "090"

  If RightSurroundAng2 = 360 then RightSurroundAng2 = "000"
  If RightSurroundAng2 = 5 then RightSurroundAng2 = "005"
  If RightSurroundAng2 = 10 then RightSurroundAng2 = "010"
  If RightSurroundAng2 = 20 then RightSurroundAng2 = "020"
  If RightSurroundAng2 = 30 then RightSurroundAng2 = "030"
  If RightSurroundAng2 = 40 then RightSurroundAng2 = "040"
  If RightSurroundAng2 = 50 then RightSurroundAng2 = "050"
  If RightSurroundAng2 = 60 then RightSurroundAng2 = "060"
  If RightSurroundAng2 = 70 then RightSurroundAng2 = "070"
  If RightSurroundAng2 = 80 then RightSurroundAng2 = "080"
  If RightSurroundAng2 = 90 then RightSurroundAng2 = "090"

  If LFEAng2 = 360 then LFEAng2 = "000"
  If LFEAng2 = 5 then LFEAng2 = "005"
  If LFEAng2 = 10 then LFEAng2 = "010"
  If LFEAng2 = 20 then LFEAng2 = "020"
  If LFEAng2 = 30 then LFEAng2 = "030"
  If LFEAng2 = 40 then LFEAng2 = "040"
  If LFEAng2 = 50 then LFEAng2 = "050"
  If LFEAng2 = 60 then LFEAng2 = "060"
  If LFEAng2 = 70 then LFEAng2 = "070"
  If LFEAng2 = 80 then LFEAng2 = "080"
  If LFEAng2 = 90 then LFEAng2 = "090"

'Write data to parameters.txt
  set ParamFile = fso.CreateTextFile("d:\wwwroot\parameters.txt", true)
  ParamFile.writeLine("D:\\Group4Data\\" + Demo + "\\L.wav")
  ParamFile.writeLine("D:\\Group4Data\\" + Demo + "\\C.wav")
  ParamFile.writeLine("D:\\Group4Data\\" + Demo + "\\R.wav")
  ParamFile.writeLine("D:\\Group4Data\\" + Demo + "\\RS.wav")
  ParamFile.writeLine("D:\\Group4Data\\" + Demo + "\\LS.wav")
  ParamFile.writeLine("D:\\Group4Data\\" + Demo + "\\LFE.wav")
If EarType = "L" then
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + LeftElev + "\\" +
EarType + LeftElev + "e" + LeftAng + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + LeftElev + "\\" +
EarType + LeftElev + "e" + Cstr(LeftAng2) + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + CenterElev + "\\" +
EarType + CenterElev + "e" + CenterAng + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + CenterElev + "\\" +
EarType + CenterElev + "e" + Cstr(CenterAng2) + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + RightElev + "\\" +
EarType + RightElev + "e" + RightAng + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + RightElev + "\\" +
EarType + RightElev + "e" + Cstr(RightAng2) + "a.wav")
```

```
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + RightSurroundElev +
"\\" + EarType + RightSurroundElev + "e" + RightSurroundAng + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + RightSurroundElev +
"\\" + EarType + RightSurroundElev + "e" + Cstr(RightSurroundAng2) + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + LeftSurroundElev + "\\"
+ EarType + LeftSurroundElev + "e" + LeftSurroundAng + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + LeftSurroundElev + "\\"
+ EarType + LeftSurroundElev + "e" + Cstr(LeftSurroundAng2) + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + LFEElev + "\\" +
EarType + LFEElev + "e" + LFEAng + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + LFEElev + "\\" +
EarType + LFEElev + "e" + Cstr(LFEAng2) + "a.wav")
End If
If EarType = "R" then
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + LeftElev + "\\" +
EarType + LeftElev + "e" + Cstr(LeftAng2) + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + LeftElev + "\\" +
EarType + LeftElev + "e" + LeftAng + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + CenterElev + "\\" +
EarType + CenterElev + "e" + Cstr(CenterAng2) + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + CenterElev + "\\" +
EarType + CenterElev + "e" + CenterAng + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + RightElev + "\\" +
EarType + RightElev + "e" + Cstr(RightAng2) + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + RightElev + "\\" +
EarType + RightElev + "e" + RightAng + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + RightSurroundElev +
"\\" + EarType + RightSurroundElev + "e" + Cstr(RightSurroundAng2) + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + RightSurroundElev +
"\\" + EarType + RightSurroundElev + "e" + RightSurroundAng + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + LeftSurroundElev + "\\"
+ EarType + LeftSurroundElev + "e" + Cstr(LeftSurroundAng2) + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + LeftSurroundElev + "\\"
+ EarType + LeftSurroundElev + "e" + LeftSurroundAng + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + LFEElev + "\\" +
EarType + LFEElev + "e" + Cstr(LFEAng2) + "a.wav")
  ParamFile.writeLine("D:\\Group4Data\\HRTFs\\elev" + LFEElev + "\\" +
EarType + LFEElev + "e" + LFEAng + "a.wav")
End If
  ParamFile.writeLine(Int(LeftDist))
  ParamFile.writeLine(Int(CenterDist))
  ParamFile.writeLine(Int(RightDist))
  ParamFile.writeLine(Int(RightSurroundDist))
  ParamFile.writeLine(Int(LeftSurroundDist))
  ParamFile.writeLine(Int(LFEDist))

  ParamFile.writeLine(Int(LeftAng))
  ParamFile.writeLine(Int(CenterAng))
  ParamFile.writeLine(Int(RightAng))
  ParamFile.writeLine(Int(RightSurroundAng))
  ParamFile.writeLine(Int(LeftSurroundAng))
  ParamFile.writeLine(Int(LFEAng))

  ParamFile.writeLine(Int(LeftElev))
  ParamFile.writeLine(Int(CenterElev))
  ParamFile.writeLine(Int(RightElev))
  ParamFile.writeLine(Int(RightSurroundElev))
```

```
  ParamFile.writeLine(Int(LeftSurroundElev))
  ParamFile.writeLine(Int(LFEElev))

  ParamFile.writeLine(Int(Temperature))
  ParamFile.WriteLine(PlayMode)
  ParamFile.Close

  set UpdateFile = fso.CreateTextFile("d:\wwwroot\updated.txt", true)
  UpdateFile.WriteLine("parameters.txt has been updated since last use.")
  UpdateFile.Close

  fso.DeleteFile "d:\wwwroot\inuse.txt", false
    'false here means that file will not be deleted if it is read-only.
%>
<P>
... SUCCESS!
<P>
<P>-------------------------------------------------------------------------
--
<P>Hit Back to select and submit another set of parameters.
<%
End If

Set fso = Nothing
Set Elev = Nothing
%>
<P>You can view/download <A HREF="parameters.txt">parameters.txt</A> (set
browser to Refresh every time) or its <A HREF="paramkey.txt">explanatory
key</A>.
</BODY>
</HTML>
```

References

[i] This endnote refers to nothing, but Word won't let me delete it.
[ii] Richard O. Duda. *Modeling Head Related Transfer Functions*, IEEE 1993.
[iii] F. Wightman and D. Kistler. *Localization of Virtual Sound Sources Synthesized From Model HRTFs*.
[iv] Bill Gardner and Keith Martin. *HRTF Measurements of a KEMAR Dummy-Head Microphone*. MIT Media Lab Perceptual Computing - Technical Report #280. 1994.
[v] Ming Zhang, Kah-Chye Tan, M.H. Er. *A Refined Algorithm for 3D Sound Synthesis*. Centre for Signal Processing, Nanyang Technological University. 1998.
[vi] P. Georgiu, C. Kyriakakis. *A Multiple Input Single Output Model For Rendering Virtual Sound Sources In Real Time*. Immersive Audio Laboratory, 2000.
[vii] www.openal.org and developer.creative.com
[viii] Confirmed through personal email communication with Goldwave's author, Chris Craig.
[ix] Richard O. Duda. *An Introduction to Human Spatial Hearing*. Available online at: http://www.umiacs.umd.edu/~ramani/hrtf/duda_umd00_slides.pdf.
[x] Musical Acoustics, 2nd Edition, by Donald E. Hall. Brooks/Cove Publishing Company, Pacific Grove, California, 1990.