Adding DSP Enhancement to the Bose® Noise Canceling Headphones

by

Nathan Fox Hanagami, S.B.

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005 [June 2005]

Author . . . [illegible] . . .
Department of Electrical Engineering and Computer Science
May 18, 2005

Certified by . . . [illegible] . . .
Joel E. Schindall
Professor, Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . [illegible] . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Adding DSP Enhancement to the Bose® Noise Canceling Headphones

by

Nathan Fox Hanagami, S.B.

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2005, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

A Discrete Time Signal Processor for use on the audio path in the Bose QuietComfort® Acoustic Noise Cancelling® [5] Headphones was designed and implemented. The purpose was to demonstrate that digital processing was feasible for this type of portable consumer product, and to design and add new equalization and adaptive abilities not currently available in the all analog circuitry. The algorithm was first prototyped in Simulink® and Real-Time Workshop®[1]; it was then adapted to embedded code for an ADI Blackfin® ADSP-BF533 DSP.

Thesis Supervisor: Joel E. Schindall
Title: Professor, Department of Electrical Engineering and Computer Science

[1]Both these software products are made by MathWorks. See http://www.mathworks.com for more information.

Acknowledgments

I would like to acknowledge everyone at Bose Corporation who contributed both in terms of resources and in terms of insight. Dan Gauger who served as my direct supervisor in the design of the algorithm deserves special acknowledgment. Also, I would like to acknowledge Adam Cook and Giuseppe Olivadot, the applications engineers at Analog Devices who helped familiarize me with the Blackfin processor.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations for DSP

Digital Signal Processing (DSP) can provide much functionality that is difficult to implement in an all-analog system. Furthermore, it provides a medium of design that is highly modifiable and customizable. Adding a DSP component to an all analog system allows the designer to rapidly prototype, test and implement many new and powerful algorithms. Also, once the DSP infrastructure has been installed further revisions can be implemented at a pace much quicker than in all-analog.

The goal of this project was to evaluate the usefulness of DSP in noise canceling headphone applications particularly for use in the Quiet Comfort® 2 (QC2) Acoustic Noise Cancelling® headphones. This evaluation was to serve as a proof of concept to Bose Corporation for future work with DSP in their Noise Reduction Technology Group (NRTG). Supporting a MIT Masters Thesis allowed Bose to explore the potential for DSP in this group without having to invest in permanently employing an engineer.

1.2 Background of the QC2 operation

The QC2 currently exists as an all-analog device. It has a power supply that regulates an AAA battery to 2.75 V DC (this is a factor we will have to consider when

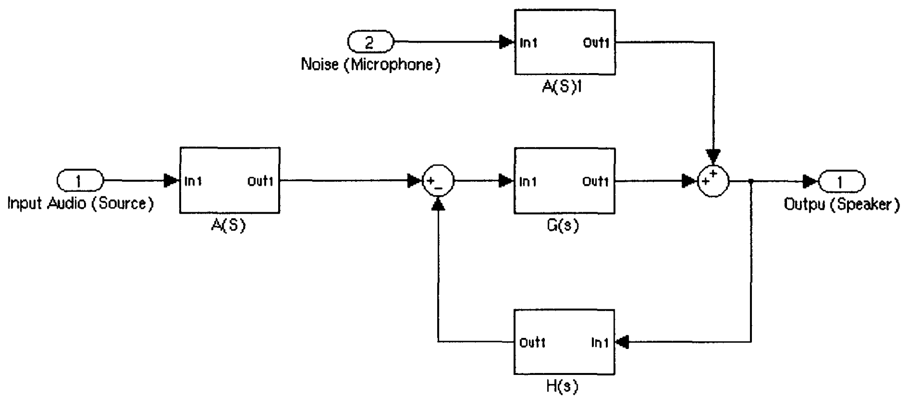

Figure 1-1: Analog Block Diagram

choosing our digital components). The audio signal is split into high frequency and low frequency components that are injected into the signal path at different points. There is a forward path that only effects the audio equalization, and there is feedback path that effects both the audio equalization as well as the active noise reduction. Both the forward path and the feedback path are composed of high order active and passive filters (8th order in the feedback compensation).

Figure 1-1 depicts a block diagram of the current implementation. For simplicity's sake we are ignoring the frequency splitting of the input audio and visualizing the audio as being injected through one block. Note that the feedback is achieved through the coupling between the speaker and microphone in the ear cup. The transfer function between these components is included in the H(s) block.

The active noise reduction peaks at about 20 dB of attenuation. The steady state power dissipation of the system is about 8-12 mA. If there is a sudden change in noise levels the power supply can supply up to around 100 mA.

Our goal is to replace, improve, and expand the functionality of the existing analog circuitry. We want to remove any analog circuitry that can be replaced by the digital components we are already installing. We want to improve upon the analog EQ by creating filter topologies that are difficult to generate in analog. Finally, we want to add functionality to the headphones by using clever algorithms that will enable

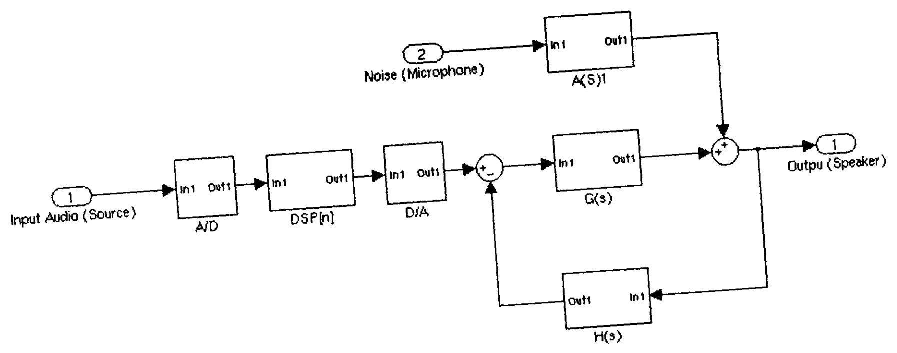

Figure 1-2: Digitally Modified Block Diagram

the headphones to adapt in ways that are difficult to do with limited board space in analog.

1.3 Initial Hardware Modifications

First, we must determine which analog components do not affect the feedback path. if the component does not affect the feedback path, then it can be said not to affect the noise cancelation. Our digital circuitry is going to exist outside of the feedback loop. Figure 1-2 shows a generalized block diagram of our digitally modified system.

We determined what analog components could be removed first by simulating the changes in Spice. We tested to make sure that the feedback path would not be changed by more than half a dB at any point in its contour. These tests resulted in the modified schematic seen in Figure 1-3.

Once the circuit was modified we then ran frequency sweep tests on the resultant circuit to make sure that the simulated results were accurate. We also simulated and tested the resultant analog EQ curve. The functionality of the components removed from the analog forward path must be replaced in the digital domain. So, we must know what the new analog EQ is so that it may be appropriately adjusted with the digital EQ. The new analog EQ curve is shown in Figure 1-4.

Finally, we had to add circuitry in order to read the signal produced by the ear cup microphone. Knowing the signal seen at the microphone allows us to create

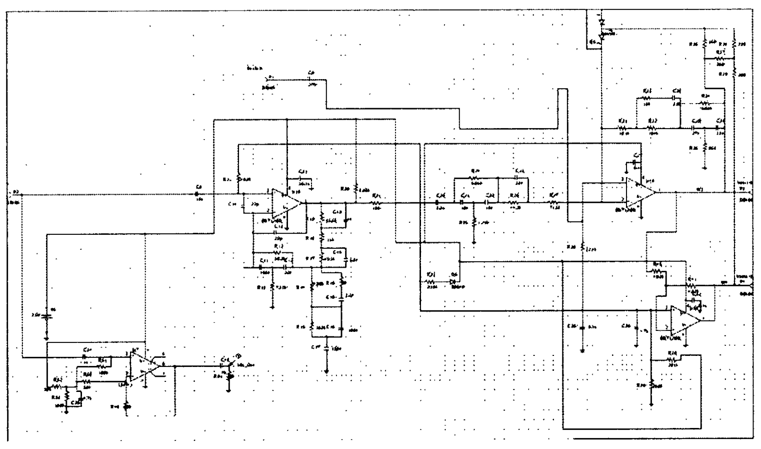

Figure 1-3: Modified QC2 Schematic

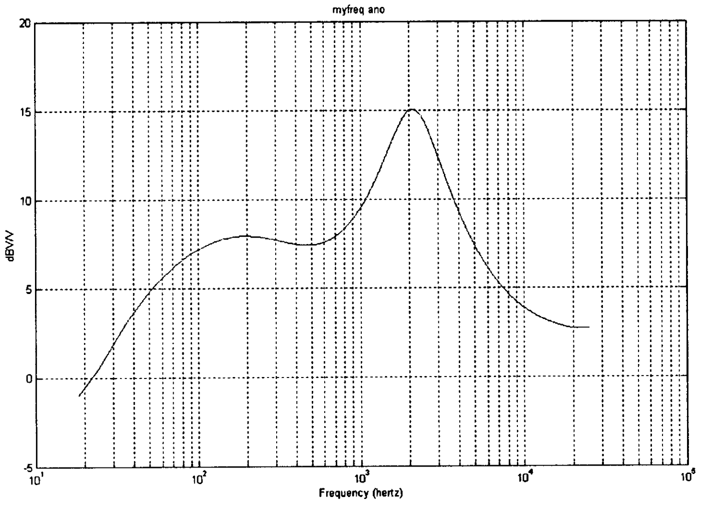

Figure 1-4: Modified Analog EQ Curve

active compression algorithms that respond to the background noise level as well as the audio signal level. The QC2 headphone has the benefit of already having a microphone installed to close the Active Noise Cancelation feedback loop. In order to read the voltage from the microphone we must buffer the signal as not to effect the impedance seen at the microphone' output node. We do this with a simple non-inverting opamp buffer with a gain of 2 (see Figure 1-3). The gain is for prototyping purposes because the microphone has a lower sensitivity than that of the speaker. If the input impedance of the actual DSP chip is high enough the opamp buffer may not be necessary in the final board level design, but it is important for our evaluation system.

1.4 Algorithm Development

In order to begin building a DSP prototype the designer must first classify the algorithm. That is where we continued our project. We first interfaced our modified headphones with a Simulink® and Real-Time Workshop®[1] floating-point system in order to test initial design ideas (See Chapter 2.1). At this point in the design phase we classified the ultimate functionality of our system. We decided that the system was to firstly provide the ideal static equalization; secondly provide a dynamically adaptive equalization; and lastly provide noise adaptive compression (See Chapter 3). Our implementation of noise adaptive compression was particularly interesting. Prior systems operated by running computationally intensive adaptive filtering to isolate background noise from signal source based on a microphone signal (a microphone located in the same environment as the system speaker will have an element of the signal plus the noise) in order to obtain a signal-to-noise ratio. This must be done because the transfer function from the speaker to microphone, in many cases, varies over time and an adaptive filter is necessary to implement a reverse convolution. The noise reduction circuitry, already present in the QC2, leads to a time stable and flat transfer function over the 80 to 800 Hz bandwidth. We took advantage of this as-

[1]Simulink and Real-Time Workshop are manufactured by MathWorks

pect to make our compression algorithm as a function of Signal-to-Signal-Plus-Noise (SNSR) thus avoiding much complicated filtering.

1.5 Algorithm Implementation

The next stage in the project was to implement our Simulink design in a functional DSP system. At this stage the actual software that will run on a DSP chip was coded. Our final choice for our DSP was the Analog Devices ADSP-BF533 (See Section 2.2.1). This meant that we had to convert our code into a power-sensitive, fixed-point world. Fixed-point introduced many design concerns that were not apparent while designing in floating-point. Particularly we had to worry about our systems dynamic range and round off errors (See Chapter 4). Dynamic range became a problem if large gains were applied to the signal and digital clipping resulted. Round off errors were a problem because standard fixed-point notation only rounds to the nearest integer, and Analog Devices' fractional library only allows for numbers of magnitude less than one. Rounding became a problem in situations such as conversions from a dB boost value to a linear gain. In linear gains we needed values to range from 0 to about 15. Integer values would have been fine at high gains, but at lower gains integer transitions would be abrupt and unsatisfactory. Fortunately, the ADSP-BF533 is capable of emulating floating-point computation with a fixed-point processor, and many of these problems were avoided.

1.6 Further Work

Further work would call for code optimization and installation into the QC2 Printed Circuit Board (See Sections 2.3, 2.4, and 5.1). The major optimization step would have been implementing as little floating-point emulation as possible, and then making design tradeoff choices based on the design complexity and lack of fidelity in fixed-point algorithms versus the computational intensiveness of floating-point emulation. We would then have to design a final board-level layout and install our system in the headphones. At this point we would have a working prototype in its entirety, that would hopefully offer exciting advantages over the current state of the art in analog headphone technology.

Chapter 2

System Design and Development

In this project we designed a Digital Signal Processing (DSP) algorithm for applications on the audio path of the Bose QuietComfort® 2 Noise Cancelling® headphones. The goal was to build a fixed-point system that sampled input audio signals at 48 kHz and processed envelope levels at 2.4 kHz. The system takes as inputs the left audio signal, the right audio signal, and the microphone signal from the left ear cup, and outputs stereo audio signals to the left and right earcup speakers. The project was designed first on a Simulink rapid prototyping system. Next, the algorithm was ported to an embedded, fixed-point environment. Further work would call for system optimization and installation into the board level circuit of the QC2 headphones. This chapter will cover the steps associated with building a prototype of a DSP system. Our specific design is representative of many problems that are encountered in the prototyping process.

2.1 Algorithm Development

The development of our algorithm was done using Simulink® and Real-Time Workshop® (RTW)[1]. Simulink is a Matlab® design tool that allows the user to specify signal processing systems in block diagrams. Simulink is ideal for building complicated

[1]Both of these products are made by MathWorks as is Matlab. See http://www.mathworks.com for more information.

systems spanning several layers of abstraction by allowing the user to add subsystems of blocks within blocks. Real-Time Workshop is a design tool which compiles Simulink systems into executable files. These executables may either run statically, or be actively updated through controls the designer may build into the Simulink system. Our Real-Time Workshop system was made to compile the code to a linux executable. RTW would then transfer this code remotely to a separate linux computer. The linux computer was connected to a 96 kHz, 24-bit RME brand firewire sound card. The sound card functioned as an Analog-to-Digital/Digital-to-Analog converter (CODEC), the linux computer functioned as the DSP, and the Simulink system functioned as the DSP controller. All together these components created a DSP system that provided extremely fast prototyping capabilities, with high computational power; however, it was not close to board level in terms of size, power consumption, or limitations on its capabilities.

The Simulink system was designed without regard for fixed-point limitations or power consumption. The rational was to design the best possible system to prove that DSP could provide useful functionality, without worrying about actual implementation. We intended this stage of the project to be a proof of concept. Since the QC2 headphone had never had a DSP system associated with it, we felt that a proof of concept would be useful. One of our "lessons learned" is that more attention could have been paid during the design phase to how the final system was to be implemented. Simulink has a fixed-point block set and limiting the design to fixed-point in the Algorithm Development stage could have saved much time in implementing this algorithm on the final processor (see Section 2.2). At the time, though, we did not know with certainty that the final processor would be fixed-point. In the end this stage served to classify what it was that we wanted the system to do; it did not necessarily provide insight as to how that functionality was to be obtained in a fixed-point world.

It was during the Algorithm Development stage that we decided exactly how the system was to function. In the initial planning stages we had brainstormed many potential uses for a DSP system. The "dream idea" was to have the DSP aid in active noise reduction (ANR). Current technology, however, precludes digital ANR because the lag and phase delay associated with current analog-to-digital converters would result in an unstable system. Another notable idea was active loop gain adjustment. This functionality would allow the system to always remain on the verge of instability, where the loop gain (and hence the noise reduction) is maximized but the system would still barely remain stable. There was also an idea for a compression algorithm where a "night mode" could be turned on to compress the dynamic range of the music in such a manner to be just audible above the background noise. This function would allow the listener to use music to replace noise as the background environment. In the end "night mode" was discarded for a subtler form of compression.

During the Algorithm Development stage, we decided to limit the system implementation to three components. First, the system was to replace any analog circuitry that only acted on the forward audio path. This would eliminate much of the audio equalization circuitry. The rational behind this functionality was that by installing the DSP circuitry we are taking up valuable board space and power, and any analog components that could be emulated in digital should be replaced.

The second function of the DSP was to provide an equalization that adapted to the content of the input audio (Dynamic EQ). This came out of the experimental observations that the response of human hearing actually varies depending on the input level. Studies show (see the curves generated by Fletcher and Munson)[2] that the human ear gets worse at detecting low and high frequency sounds as the signal level decreases. The fall off for high frequencies is much slighter than that for low frequencies and was deemed inconsequential. The low frequency loss in response, though, can be quite noticeable. There are several pieces of music that exemplify how noticeable this effect can be when listened to at contrasting loud and quiet volumes.[3] This effect was fixed by implementing a system that first detected the input level, then added appropriate gain to the low frequency content of the music.

[2]See http://www.webervst.com/fm.htm[6] for an explanation of the Fletcher-Munson curves

[3]See Saint-Saens Symphony #3 where the organ disappears at low levels. We also listened to Holly Cole's "See Clearly Now", Freeze Thaw's "Basia", and Nine Inch Nails' "Piggy" for other examples of low frequency attenuation.

The result of the system was a signal that where the equalization sounds appropriate at both high and low volumes.

The third and final component of the floating-point system is referred to as Noise Adaptive Upwards Compression (NAUC). Studies in psychoacoustics literature[4] have shown that signals more than 10 to 15 dB below background noise cannot be heard. This can be a problem in high noise areas especially if the piece has widely varying dynamic range. If the dynamic range varies slowly, then in order to keep the loud passages bearable and the quiet passages audible the listener may find himself constantly adjusting the volume. If the dynamic range varies rapidly, either the quiet passages will disappear or the loud passages will appear obtrusive.[5] In an attempt to remedy this problem, a research engineer at Bose[6] suggested that the audio be aggressively compressed upwards when below the background noise while leaving it largely uncompressed when above the noise. This was the inspiration for the NAUC system. To do this we analyze both the input signal level (after Dynamic EQ correction) and the signal level already seen from the microphone in the noise cancelation feedback loop. We then calculate a wideband gain based on these levels. The goal of the NAUC system was to make sure that the low level inputs are compressed upwards in order to always be audible over the background noise. The user can then set the level based on the loudest parts of the piece, and the system will guarantee that the quiet passages will be at least audible. Ideally the quiet sections would be compressed the minimum amount to preserve as much of the dynamic range as possible. In prior art, noise adaptive compressor systems implemented complicated adaptive filtering to isolate the level or spectrum of the background noise sensed by a microphone from the reproduced audio input signal also present at that microphone. We took advantage of the noise reduction feedback loop inherent in the QC2 headphones to avoid this

[4]See Stevens and Guirao (1967), "Loudness Functions and Inhibition"[9]

[5]See Respighi's "Pines of the Appian Way" for a piece that begins very quietly and ends very loudly. Set the original volume so that the opening passages are audible in the presence of background noise (e.g., in an airliner or car); by the end of the piece the signal levels will be almost unbearable. See Flim and the BBs' "Tricycle" for a piece with rapidly varying dynamic range. Set the volume such that the beginning passages are at a comfortable level; there will be percussive hits that seem painfully loud.

[6]Chris Ickler

complicated filtering, which will be discussed in section 3.6.

2.2 Algorithm Implementation

In the algorithm implementation stage the designer must convert the system from an existing rapid prototyping environment, such as Simulink, to an environment that can be implemented in a real product. In our case this meant porting the code from a floating-point, computationally intensive, Simulink environment to embedded DSP code. Our embedded DSP system was coded in C++ for the Analog Devices Blackfin® ADSP-BF533 EZ-kit Lite™Evaluation Board[7] as described later in this section. There are many challenges associated with porting an algorithm from a system like Simulink to embedded DSP code. Furthermore, code compiled onto an embedded DSP chip, especially a low power fixed-point chip like the Blackfin, may respond differently than it would compiled onto a CPU, or the DSP may not even be able to handle certain code fragments at all.

2.2.1 DSP Selection

We looked at several criteria for selecting the DSP. The most important were computation capabilities, power consumption, fidelity and dynamic range (bit-count), ease of portability from Simulink to embedded code, amount of on chip memory, size of the chip, and any added features that make the floating-point to fixed-point conversion easier.

The two DSP providers we considered were Analog Devices (ADI) and Texas Instruments (TI). We chose these companies primarily because they were organizations with which the Bose Corporation had prior experience. We believed that if Bose employees were familiar with the part we worked with, then they would be able to offer valuable advice. Each company offers multiple lines of DSP parts. ADI recommended their low power Blackfin line of DSP, while TI recommended either their low power

[7]see http://www.analog.com/en/epProd/0,,ADSP-BF533,00.html for information on this product.

Table 2.1: DSP Spec Comparisons

Processor	Blackfin	C55xx	C64xx	C67xx
CPU Type (bit Fixed)	16*	16	16/32	32/64**
Maximum Clock Speed (MHz)***	600	300	1000	300
Power Consumption (mw/MHz)	0.3	0.65	1.60	4.15
FFT Efficiency (cycles/16 bit FFT)	2324	4786	3001	3696****

* with available Floating-Point Instruction Set

** Floating Processor

*** ADI specifications were listed at mW for a fixed clock speed, and TI specifications were listed as mA/MHz. The numbers were calculated assuming all processors operated with linear power consumption increases with respect to clock speed and at a 1.45 mV supply.

**** cycles for an unlabled FFT

C5000 series or their higher performing C6000 series. The C6000 series offered many performance benefits such as higher maximum clock speeds and a floating-point processor but the power consumption was too high to make it viable in our application. For a comparison over the various processors see Table 2.1[8]. With computational capability and power consumption being our greatest concern it became obvious that either the ADI Blackfin or the TI C55xx boards would most closely meet our needs.

Many of the TI boards have support from MathWorks for automatically converting Simulink systems to embedded code. This ease of use added some value to the TI boards that may not be obvious through simple benchmark statistics. Upon further investigation, though, we found that there is also third party support that offers similar functionality for the ADI DSP evaluation boards. This software is called DSP*developer*™ and is manufactured by SDL.[9]

Another statistic not listed in Table 2.1 was the onboard memory. The C55xx processors offer up to 320 kb of RAM.[10] The Blackfin offered up to 148 kB of on chip memory.[11] While the TI board offered more maximum storage space, these

[8]All data regarding ADI parts was taken from "BF533 Vs TIC55x.ppt" [7] produced by ADI. All data regarding TI parts was taken from "boseenrg121404.pdf' [3] produced by TI. Both number sets were verified on www.analog.com and www.ti.com.

[9]See: http://www.sdltd.com/_dspdeveloper/dspdeveloper.htm

[10]See: http://dspvillage.ti.com/

[11]Taken from the Blackfin datasheet found at http://www.analog.com

Table 2.2: DSP Efficiency Comparison

Processor	Blackfin	C55xx
Delayed LMS Filter	$1.5h + 4.5$	$2h + 5$
Radix 2 Butterfly Kernal	3	5
256pt Complex FFT	2324	4786
Block FIR Filter	$(x/2)(2 + h)$	$(x/2)(4 + h)$
Complex FIR Filter	$2h + 2$	$2h + 4$
Max Search	$0.5x$	$0.5x$
Max Index Search	$0.5x$	$0.5x$
Biquad IIR	$2.5bq + 3.5$	$3bq + 2$

h = # of taps
x = # of samples

two statistics were comparable and both were more than were anticipated for our design. Having narrowed our choice down to either the Blackfin or the C55xx we could then look at more in depth efficiency comparisons to come to a final decision (See Table 2.2)[12].

After a final analysis of the performance of the two DSP kits the ADI Blackfin appeared to most closely match our needs. This was due to a combination of power consumption, maximum clock speed, and the Blackfin's ability to emulate floating-point computation. We selected the ADSP-BF533 EZ-kit Lite Evaluation board, because it offered the best combination of performance, stability, and support. Also, the engineers at Bose were, in general, more familiar with the ADI parts than the TI parts offering more support possibilities internal to Bose Corporation.

2.2.2 Data Converter Selection

Along with selecting the DSP we also had to choose data converters. Analog-to-Digital Converters (ADCs) are necessary to digitally read both the input music signal and the input microphone signal. Also, Digital-to-Analog Converters (DACs) are necessary to then feed the digital output to the headphone speaker. Combined two-channel Analog-to-Digital/Digital-to-Analog Converters (CODECs) are available

[12]All numbers taken from
http://www.analog.com/processors/processors/blackfin/benchmarks/comparison.html[4]

from many companies. We decided that one stereo CODEC for the audio signal conversion along with one lower speed ADC for the microphone would provide the best solution.[13] First, we talked to representatives from AKM who offered the AK4555, a low power 20-bit CODEC that ran at 16 mW. Also, they recommended their AK5355 16-bit ADC that ran at 15 mW. We then talked to representatives from Cirrus Logic as well as TI. Both companies were in the process of completing low power CODECs, but they did not have support or evaluation boards available as of yet. This made the decision to go with AKM relatively straight forward, because we decided the potential performance benefits of the competitors parts did not outweigh the advantages of having a well supported and tested device.

2.2.3 Embedded Design

There can be many difficulties encountered in porting a floating-point algorithm from a rapid prototyping system such as Simulink to embedded code on a functional, fixed-point DSP. The designer now has to worry about what functions are native to the embedded language (C++ or Assembly in our case); the designer must worry about problems associated with fixed-point calculations; the designer must worry about how many computations the system is using on any given sample; also the designer must worry about how much memory is used in order to store the algorithm.

Rewriting the algorithm in C++ can be more challenging than it may seem initially. There are several built in functions in Simulink that may not be native in the C++ environment. One particular difference encountered in our design was that our Simulink model was originally designed as a streaming implementation and many of the ADI libraries called for data to be given in blocks. A streaming implementation processes the input in a sample-by-sample methodology while a block processing implementation collects some predetermined number of samples and runs the algorithm on the block of data. This is analogous to the "Windowing" associated with many signal processing algorithms. Block processing algorithms usually ignore data asso-

[13]The microphone only operates on the envelope domain of the signal for level processing. For information on this see Chapters 3 and 4.

ciated with previous blocks leading to artifacts around the beginnings and ends of blocks when running IIR and FIR filters. The advantage of using a block processing implementation is that it is usually less computationally intensive. Let us look at a generic filter as an example. In the streaming methodology the DSP must load the coefficient values out from memory every sample cycle. In a block implementation the system must only load these values once per block. As a result there is less set up computation. In Simulink there are native functions for buffering and unbuffering samples into blocks and the samples into blocks. In C++ something like this must be coded by the designer. In the end we experienced difficulty getting the ADI block implementations of IIR filters to work properly in our system, so custom sample-by-sample IIR filters had to be coded. In Simulink there are standard blocks that will run IIR filters on either a block of data or a single sample. In our embedded code custom biquads were hard coded. Each biquad operated via[8]:

$$\begin{aligned} x[n-2] &= x[n-1]; \\ x[n-1] &= x[n]; \\ x[n] &= input; \\ y[n-2] &= y[n-1]; \\ y[n-1] &= y[n]; \\ y[n] = b_0 * x[n] + b_1 * x[n-1] + b_2 * x[n-2] - a_1 * y[n-1] &- a_2 * y[n-2]; \end{aligned}$$

The biquad example is also perfect for illustrating some of the difficulties in implementing code in fixed-point. In our algorithm the first biquad of the digital equalization filter[14] called for coefficients of:

$$a_0 = 1$$
$$a_1 = -2$$
$$a_2 = 1$$
$$b_1 = 0.93461$$
$$b_2 = 1.30045$$

Standard fixed-point convention only allows the programer to operate on integers. Rounding these coefficients to the nearest integer would cause a vastly different filter. The ADI libraries, however, have conventions for specifying fixed-point numbers that all range from -1 to 1. A binary value of all 1's (listed from MSB to LSB) is constructed as:

$$-2^0 + 2^{-1} + 2^{-2} + 2^{-3} ... + 2^{-29} + 2^{-30} + 2^{-31}$$

This allows us to account for values less than 1 in our designs; however, it does not provide the programmer with a method for accounting for numbers greater than 1 with a fractional component. To design fixed-point filters we must redesign our biquads to only use coefficients less than 1. To do this we first divide each coefficient by the lowest multiple of 2 greater than the largest coefficient value (multiples of 2 are used so the the divide may be accomplished using simple binary shifts) in our case this value was 2. This reduces each coefficient to a value less than 1. Our output equation is now:

$$\frac{y[n]}{2} = \frac{b_0}{2} * x[n] + \frac{b_1}{*} x[n-1] + \frac{b_2}{2} * x[n-2] - \frac{a_1}{2} * y[n-1] - \frac{a_2}{2} * y[n-2]$$

This would be a rather straight forward equation except for the $\frac{y[n]}{2}$ term. We cannot simply divide y[n] by two because we need to get access to what y[n] actually is in

[14]We refer to the ideal equalization curve for the QC2 as the Dewey after the engineer who suggested it. In this manner, we refer to our digital equalization filter as the Digital Dewey.

order to generate the delayed output terms. In order to accomplish this we use:

$$y[n]_{frac} = \frac{b_0}{2} * x[n] + \frac{b_1}{*} x[n-1] + \frac{b_2}{2} * x[n-2] - \frac{a_1}{2} * y[n-1] - \frac{a_2}{2} * y[n-2]$$
$$2 * y[n]_{frac} = y[n]$$

and then feed this y[n] back into the delay line. The multiply by two may also be accomplished with a simple binary shift. In the end we used floating-point emulation in order to accurately generate the Dewy filters (see Section 2.2.1 for information on the Blackfin's floating-point emulation). Even though emulating floating-point is relatively inefficient in terms of computation the fidelity it added was deemed appropriate for an initial step in completing an embedded prototype.

Fixed-point processors also limit the instruction sets that the C compiler is able to use. Many operations are only available as floating functions. The two examples of operations like this that affected our design were the log and power functions. In order to go from linear to dB space one must run a log_{10} on the linear input according to the dB20 equation:

$$Output_{dB} = 20 * log_{10}(Input_{Linear})$$

and to go from dB space back to linear one must compute a power function according to the undB20 equation:

$$Output_{Linear} = 10^{\frac{Input_{dB}}{20}}$$

These functions, though, will always return either floats or doubles (extended precisions floats). Some implementation of these functions typically much unavoidable if the programmer wants to use algorithms acting in the dB domain unless a logarithmic tapered ADC is used. We can use floating-point emulation in order to generate these functions. The log10f and powf functions are floating-point functions that can accomplish the desired computation. The powf operation, however, seemed to overwork our processor and resulted in a "fuzzy" sounding output. In fact, if the operation was anywhere in the code, even if it was not affecting the output, the output would still be fuzzy. The probable explanation for this effect is that the powf function, being generated through floating-point emulation, was too computationally intensive for our fixed-point processor to compute in the time it took to run one sample. As a result samples were lost and a lower fidelity output was heard. A similar effect was heard if the printf operation was used (displays data values on the terminal) in which case almost every output sample was lost and only fragmented blips were heard. The log10f function, though, emulated without resulting any any lost samples or "fuzziness".

One possible solution would be to use a Taylor series in order to approximate these functions. Another solution would be to remap any dB based operations into linear space. The later was the first solution that we attempted. For the Dynamic EQ this resulted in a function that had an $Input^C$ term in it (where C is some constant). In order to obtain the accuracy and output range needed we used a 5 term Taylor series that was expanded around multiple regions. We would check the input and run a different 5 term Taylor Series depending on its value. Another solution was to use the log10f function in order to get into dB space, and then use a Taylor series to return to Linear space. This Taylor series takes the form of:

$$\begin{aligned} Output_{Linear} &= 10^a + ln(a) * 10^a * (x-a) + \frac{ln(a)^2 * 10^a}{2!} * (x-a)^2 \\ &+ \frac{ln(a)^3 * 10^a}{3!} * (x-a)^3 + \frac{ln(a)^4 * 10^a}{4!} * (x-a)^4 \end{aligned}$$

Where a is the point about which we are expanding. Converting back from dB space to a linear gain via a Taylor expansion of the undB20 equation resulted in a fairly accurate output with only having to Taylor expand about a singule point. Also, it is computationally simple enough to result in no dropped samples. We pre-computed many of the factors so that they were not actually have to be calculated by the processor. Instead they are stored as coefficients to the (x-a) terms.

We are still left with the problem of fractional numbers that we encountered in our filter design problem. In our application many of our coefficients are non-integers greater than 1, and even if the coefficient was purely fractional the (x-a) term may be a non-integer greater than 1. In order to use fractional arithmetic both inputs must have amplitudes less than 1. It would be possible to design a more complicated system that used only integer and fractional multiplication and addition, but floating emulation was deemed sufficient for our purposes. This resulted in a clean, accurate output.

This operation is an example of a case where a floating-point processor may one day be lower power than a fixed-point processor. Fixed-point processors will always be lower power than floating-point processors in terms of mW/MHz, but accurately computing this Taylor expansion would use fewer clock cycles with a floating-point processor than it would with a fixed-point processor even without using floating emulation.

A final concern to keep in mind at this point in the design stage was the amount of memory being used. If the internal memory were to be exhausted external memory would have to be installed. This would require more board space to be allocated to DSP components in the final design, and external memory accesses are power intensive. The VisualDSP++® Lite software (the software that comes with the EZ-Kit Lite evaluation board for compiling C++ or Assembly code onto the DSP) limits the programmer to 22 kB of storage space. This amount of memory was quickly exhausted. Once a full VisualDSP++ license was obtained (free for 90 days with the ADI Test Drive program)[15] and full access to the 148 kB of on chip memory was available, there were not any problems with memory limitations.

2.3 Optimization

Once the basic design is operational, the designer must take into consideration all the ways the system can be optimized. The first step is to quantify how much power is being used. The designer can do this by setting the clock speed of the DSP processor. The lower the clock speed the less power is being used by the chip (See Table 2.1 for

[15]See http://www.analog.com

statistics on power as a function of clock speed). If the chip cannot be clocked at the desired clock speed (this will most likely be apparent by a lossy signal where samples are dropped), then the algorithm must be optimized to reduce the amount of power consumed.

The most straight forward way to optimize the system would be to reduce or remove instances of floating-point emulation. If we were to rebuild our filters in the fractional method mentioned earlier, and redesign all floating-point operations as combinations of integer and fractional operations, then there would be a good savings in power costs.

The second way power consumption can be reduced is to switch from a streaming to a block processing implementation. As mentioned earlier block processing reduces setup costs in initializing computation. However, artifacts can be created and we may not get as true a response as we would with a streaming implementation. The designer must take these types of considerations into account. The longer the block being processed the truer the result will be and the more setup computations will be reduced, but at the same time the the amount of memory used in delay storage will increase and the amount of computation that must be done when the block executes increases. If the block becomes too long the system may not have time to fully compute it in a sample-cycle and the data computation may have to become asynchronous with sampling. By asynchronous computation we mean that the we would collect a new block and output an old block synchronously with the sample-clock, while the current block is processed on it's own. Also, the designer can utilize tricks with overlapping neighboring blocks in order to eliminate edge artifacts, but those techniques result in there being less savings in setup costs for a given block length. Finally, if the block becomes too long delays that are unacceptable may occur, because the input to output delay must be at least one block length (in this manner closed loop active noise reduction would not be possible using block computation).

2.4 System Installation

Once the algorithm has been verified and optimized we can build the final system. If we had multiple engineers working on the project the hardware design would have been conducted in parallel with the software work. In our case we had a single engineer working, so the hardware design was not completed. This stage consists of PCB layout design, power supply design, and interfacing the digital components. For relevance to this project we are assuming that we are starting with an already existing PCB board that must be redesigned to fit our new needs.

The first thing we did was remove all of the unnecessary parts in the analog portion of the circuit that were only contributing to the forward path equalization. There were a handful of capacitors and resistors that did not effect the loop gain but affected the forward path. First, the effects of removing these parts were simulated and then they were verified in lab tests to make sure that the feedback bath (and hence the noise cancelation) would not be altered. The effects of these components were taken into account when we designed our Digital Dewey filter to provide the correct audio equalization. Removing these parts freed up some amount of space for the new digital components.

The AKM AK5355 ADC operates on power supplies ranging from 2.1 to 3.6 V. The power supply on our board is 2.75 V so we do not need any modifications to obtain new voltages for this part.[16] There is good documentation for how to configure this part in its data sheet[17]. We see that adding this part will call for the addition of 8 capacitors as well as the added board space required for the part itself. The Controller and Mode Setting blocks would be control pins from the DSP chip.

The AKM AK4555 CODEC operates on power supplies ranging from 1.6 to 3.6 V. So, once again we would be able to operate this part without any power supply modifications. Like the ADC this part's configuration is documented in its application

[16]For a truly power optimized system we would want to include circuitry to allow the digital components to run at their lowest allowed supply voltages. The lower the supply voltages the lower their power consumption will be

[17]See: http://www.asahi-kasei.co.jp/akm/en/product/ak5355/ak5355.html

notes.[18] Installation of this part requires 7 additional capacitors and 2 additional load resistors. The components already installed in the system may be adequate for the load resistors, and it is suspected that these parts are not necessary. Again, the Controller and Mode Control blocks would be control pins from the DSP chip.

The BF533 is a significantly larger chip and more care has to go into installing it. It comes in a mini BFG package and the traces to and from the chip will require that much care goes into the layout design. Our current PCB board has four layers, and it may be worth considering adding more trace layers to make for easier design of the wiring. This chip has control pins that interface with both the CODEC and the ADC. It also has I/O pins that must be able to read the digital data from the CODEC and ADC and then feed the digital outputs back to the stereo CODEC. Bypass capacitors may be necessary on the supply rails as well to make sure that digital noise is not injected back into the analog signal (these capacitors will be similar to those called for by the converters). The DSP chip calls for two power supplies; the first is the external supply voltage can range from 2.5 to 3.3 V the second is the core voltage which can range from 0.8 to 1.2 V. Fortunately, the Blackfin has an on-board voltage regulator so the core voltage can be generated from the external voltage source combined with a few discrete components (as documented by the application notes). So, again we will be able to run this chip off of our currently existing supply voltage.

2.5 Summary

In summary we find that there are several concerns that we must be aware of in the process of developing a prototype. It is useful if we can be aware of our final design space while we are trying to design the functionality in the algorithm. If we are not aware of our final design space during the preliminary design, then there are several design adaptations that we can expect to make. Design efficiency, fixed-point limitations, sampling methodology tradeoffs are aspects that we must be aware of.

Ideally this project would have been conducted with at least two engineers so

[18]See: http://www.asahi-kasei.co.jp/akm/en/product/ak4555/ak4555.html

that the hardware and software designs may be conducted in parallel. The work flow in the case of multiple engineers would be: algorithm design and initial hardware modifications; embedded code and layout design; software optimization and system installation. In our project, however, we only had a single engineer. The algorithm design and initial hardware modifications were still done in parallel, but embedded coding and layout design both require large amounts of attention and the layout would have been designed after the embedded code was made.

Chapter 3

Simulink Algorithm

In this chapter we detail the algorithm as it exists in its Simulink, floating-point form. This chapter also documents the theory and rationale behind the design of the various operations as applied to the overall DSP design. Ideally a fixed-point system would be identical to the algorithm, as described below, in terms of functionality even though it would be different in implementation. As stated in Section 2.1 the Algorithm Development stage in DSP prototyping is to demonstrate functionality, as this chapter details for our system.

3.1 Input Adjustment

The inputs to both the right and left channels are scaled by a gain factor of 5.8. This serves to offset a constant adjustment associated with the analog-to-digital conversion. This scaling results in a 1 Vrms input yielding a value of 1 Vrms at the input level detector. In a computationally efficient version this gain would be omitted and the constant difference would be taken into consideration in the level processing stage.

After the initial scaling the inputs are unbuffered. The original signal is sampled in frames of 256 samples, a buffer of 50 ms, and 10 ms Period/Fragment. For our floating-point, computationally intensive, implementation we want to process the data via a streaming methodology, so we first unbuffer the signal to obtain 1 sample per 48 kHz system cycle.

3.2 Dynamic EQ Level Detection

The level detection for the Dynamic EQ is run on a combination of both the left and right channels. Due to the fact that in music the left and right channels tend to be in phase we were able to obtain the signal of interest by simply averaging both channels. A more accurate result would be obtained by running individual RMS detections on both the left and right channel, but this is computationally inefficient and summing and scaling the two audio signals provides an accurate representation. The purpose of looking at a combination of both channels rather than a single channel is to guarantee that the system will react to stereo effects that are present in only one channel. To save on more computation the divide by two adjust could be combined with later level adjustment scaling.

We then run our level detection in the envelope domain of the audio signal. Since, the envelope domain operates at a much lower frequency than the audio spectrum we are able to first down-sample to 2400 Hz. Our decimation lowpass filter consists of an 11th order Chebyshev type II filter. The passband frequency is 800 Hz and the stop band is 1200 Hz with 80 dB of attenuation. We used a Chebyshev type II filter because it has greater attenuation per order than a Butterworth filter, in exchange it has ripple in the stopband. However, for audio purposes we are only concerned with the passband, and ripple in the stopband is acceptable. We did not want to use a Chebyshev type I or an Elliptical filter because they both have passband ripple. For a less computationally intense filter we could adjust the parameters so that the the stopband frequency would be above 1200 Hz, and lower the attenuation to only 60 dB.

After anti-aliasing the audio signal is down-sampled to 2400 Hz. The down-sample is made up of a simple divide by 20 decimation, followed by a zero order hold. The zero order hold is necessary because the system clock still operates at 48 kHz.

The RMS detection operates under a feedback degeneration topology. Firs,t the input is squared. The current output is delayed by 1 sample to save the previous output. The previous output is then multiplied by a scaling factor and summed with the current output. This detector serves as an approximation of a weighted integration. We adjust the scaling factors to give our system a fast attack and a slow decay. This allows us to quickly detect large increases in level but not jump up and down during quick, quiet segments. The decay constant is decided by comparing the current output with the previous output. If the current output is greater than the previous output we use the fast attack constant, otherwise we use the slow decay constant. The result of the comparison is delayed by one sample for stability purposes. The fast attack constant is 0.9995, and the slow decay constant is 0.99999.

These coefficients result in rise and fall times that are similar in linear space. Our gains, however, are applied linearly in dB space. Our RMS output ranges from 0 to 1 Vrms^2 (because we never actually take the root). This then needs to be converted into dB units, which we obtain via the equation:

$$Output_{dB} = 10 * log(Input_{Linear})$$

We use the factor of 10 rather than 20 as an alternative to taking the square root in the RMS. This can range from negative infinity to 0. We then add a factor of 110 in order to obtain units of dBSPL (dB Sound Pressure Level). For practical considerations we will say a "quiet" input corresponds to a dBSPL level of 20 and a "loud" input corresponds to a dBSPL value of 110. For our practical rise and fall times (in dB space) we will consider the times it takes to go from a steady state quiet level to 90% of the difference (20 to 101 dBSPL) and a steady state loud level to 10% of the difference (110 to 29 dBSPL). For the time being, imagine the linear output of the RMS detector increases and decreases linearly with respect to time. First, imagine that the linear output of the RMS detector increases from 0 to 1 Vrms^2 in one second (this is a contrived situation). The dBSPL level will reach 20 dBSPL at $1 * 10^{-9}$ seconds and 101 dBSPL at 0.126 seconds, yielding a dB rise time of 0.126 seconds. Now imagine that the output of the RMS detector fall from 1 to 0 Vrms^2 in 1 second. The dBSPL level will reach 110 dBSPL at 0 seconds and 29 dBSPL at $7.9 * 10^{-9}$ seconds. We see that if we view the output of the dB conversion as a linear function the steady state to 90% rise time is an order of magnitude less than the steady state to 10% fall time. Because our gains are applied as linear functions of dB we can treat our rise and fall times in this way. In terms of dB space we should expect to see a difference in rise and fall times of about one order of magnitude not accounting for linear differences in the attack and decay of the RMS filter.

Measured results show that the RMS rise and fall times are on the order of a couple seconds in linear space. The rise time in dB space was measured to be a little under a second, while the fall time in dB space was measured to be on the order of 10 seconds. This is consistent with the previous paragraph also taking into account the minor differences in the attack and decay coefficients. The output of the RMS detector is scaled by 0.0005 in order to preserve the 1 Vrms in leading to 1 Vrms^2 out of the detector. To save computation this scaling along with all the other previously mentioned scaling factors would be merged into the dB to dBSPL conversion.

The output of the RMS detector is then converted to dB units. This is done with a dB10 conversion (dB10 instead of dB20 is used because we have already squared the input, thus we have combined the root part of the RMS detection and the dB conversion into one stage as mentioned above). After the initial dB conversion we add a factor of 110 to convert dB units to dBSPL (dB sound pressure units) units. This is because 1 Vrms applied to the speaker yields a 110 dBSPL signal in the ear cup of the headphones. This is the stage where all the previous constant gain stages could be merged to optimize for efficiency.

3.3 Dynamic EQ

3.3.1 Theory of Dynamic EQ

It is commonly known that the response of a human ear is level dependant. The Fletcher-Munson curves (See Figure 3-1)[1] showed that furthermore the human ear's sensitivity decays faster at low frequencies than at mid-band frequencies as the signal

[1]This Figure taken from http://www.webervst.com/fm.htm[6]

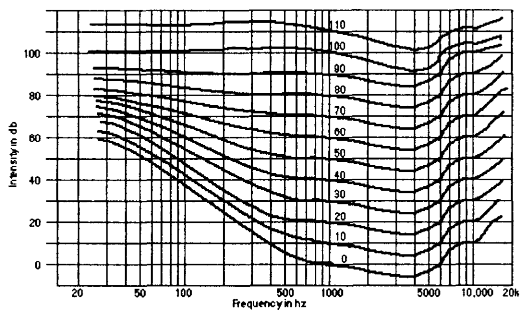

Figure 3-1: Fletcher-Munson Curves

level is lowered. It also showed that human hearing responds in a similar manner to high frequencies. This high frequency response is much less severe than the low frequency response so compensating for this has been deemed unnecessary. Older hi-fi systems addressed this with a switchable loudness control that added a fixed bass and possibly treble boost that approximated the inverse of the Fletcher-Munson curves at some predetermined low level relative to a high level; more sophisticated systems adjusted this boost in response to the volume control setting. Our system uses digital level detection and gain adjustment to provide a more natural and effective bass boost than that which was provided by the loudness control.

The Dynamic EQ is fundamentally a block that takes a signal level as input and then computes a gain value. This gain is mapped onto a two-segment piece-wise linear function in order to approximate an asymptotic function. If the the level drops bellow a certain threshold the gain increases from 0 with a certain slope. Once the input then drops below a second threshold the gain slope increases to the second piece-wise component slope. If the signal falls too low, though, the gain will remain its maximum value as to not apply overly large gain values when the input is turned off.

The two-segment piece-wise linear function is generated by feeding the RMS dB-SPL level into two parallel Dynamic Range Processors (DRP). The DRPs function to compute a linearly increasing boost, in dB units, from an input. Each DRP is configured with a max gain (dB), a slope (dB/dB), and a breakpoint (dBSPL) that is the maximum input that will yield a gain. The actual parameters are Bose trade secrets and have been omitted from the report. One of the DRPs is configured to start at our actual breakpoint, and hit its maximum value when we want the second piece of the function to begin. The second DRP begins to apply gains when the first unit reaches its maximum gain output value. The output of the two DRPs are then summed together. This means that the max gain of the second DRP is set equal to our desired max gain minus the max gain of the first DRP.

The output from the DRP's is then converted to linear units. This is done simply by the equation:

$$Output = 10^{\frac{Input}{20}}$$

The result is then reduced by one. One way of justifying the reduction of one is that a boost of 0dB maps to one through the undB, but we want a gain of zero. An alternative way of looking at this is that this gain is going to be applied to the low frequency components of our signal and then summed back to the wideband. The wideband, however, contains all of the low frequency information as well, so we must subtract one to have a gain of 1 in the overall system. In short terms the undB gives us a total gain, but what we really want is a boost above the original signal.

3.3.2 DRP Operation

First, the input is subtracted from the breakpoint value. The result of this is then sent through a *max* function with 0. This basically means that if input is greater than the breakpoint, where we do not wish to apply gain, then a 0 value is passed, if however the input is less than the breakpoint then the difference is passed. This difference is then multiplied by the slope to give us the amount of boost to apply.

$$Boost = (Breakpoint - Input) * slope$$

The result of the multiplication is then sent through a *min* function with the maximum gain value. Thus if the gain to be applied is greater than our maximum desired value, then we will simply pass the maximum value to the output.

3.4 Up sampling

The gain value then must be up sampled back to 48 kHz in order to be applied it to the audio signal. First, the gain value is zero-padded by a factor of 20. Then a *holdvalue* function must be implemented in order to maintain the gain value during the zero padded sections. Since we are only concerned with gain levels, we can simply hold the DC value and then lowpass filter to smooth the transitions, rather than running a rigorous interpolation function. The hold value function works by looking for edges of the input. If an edge (corresponding to a sample) is seen then the input is passed to the output. If an input is not seen then the output is fed back and sent again. This results in the output only changing when gain samples are sent, and then holding value during the zero padded sections, similar to the built in zero order hold. To further smooth the result the output could be passed through a lowpass smoothing filter (anti-zipper filter) to smooth the transitions, but since the gain only acts on the bassband, sudden changes are not significantly noticeable

3.5 Dynamic EQ Level Processor

The output of the Dynamic EQ is calculated by multiplying the gain value by a filtered version of the input. This filtered signal is then summed back in with the wideband signal to yield the original audio plus a boosted component. The filter used is a single biquad section; the filter parameters were determined by extrapolating from listening tests done during the design of the Bose Wave® Music System. Both the filter parameters and the gain calculation parameters were extracted from these studies. This level processing is run individually on both the left and right channels, but with the same gain value for each channel.

The signal that was filtered to generate the bass boost only comes from the left channel. This is because most music at the low frequencies processed by the Dynamic EQ are monaural. A more accurate level processor would operate individually on each channel, but this would add too much complexity to offset the added value.

3.6 Noise Adaptive Upwards Compression (NAUC) [Patent Pending]

3.6.1 NAUC Theory of operation

Studies show that audio with a signal-to-noise ratio less than -10 to -15 dB is fundamentally not present to the listener; this effect was discussed in Chapter 2.1 when we first introduced the concept of the NAUC. Fundamentally, the NAUC guarantees that an input signal will always be above this signal-to-noise threshold. To do this the system looks at the input signal, compares it to the signal level seen at the microphone which is already in place for the closed loop active noise cancellation, and calculates an amount of gain to apply to the signal. On the surface this operation seems very similar to that of the Dynamic EQ; however, operating on the wide band as well as making a decision based both on signal-level and microphone-level adds complexity.

The NAUC could be used in a few different ways. One major use for the NAUC could be for music with very wide dynamic ranges over a long period of time. Respighi's "Pines of the Appian Way" for example starts very quietly, but by the end is overbearingly loud if one leaves the volume level set only at the level that is needed to make the beginning audible. The NAUC would allow the user to initially set the volume level higher, and then the compression will back off as the signal level increases.

This results in the user riding the volume less. A second use is for music with sudden large dynamic transitions. In "Tricycle" by the jazz group Flim & the BBs there are relatively soft sections intermixed with very loud and sudden percussive hits. These percussive hits can be uncomfortable for many listeners. The NAUC would allow the user to set the volume to the level they want these loud sections to be heard at, and then will bring the soft sections up to a comfortable listening level. In both these uses the goal of an ideal NAUC would be to preserve the perceived dynamic range of the music by only compressing those sections that are outside of the listeners own range of hearing by either allowing the listener to lower the painfully loud sections or by automatically raising the inaudible sections.

3.6.2 Input Signal Level Detection

The NAUC is calibrated to adapt to the total signal level seen after the Dynamic EQ has been included. Thus, the level processor from the Dynamic EQ sends a copy of the bass signal, before it has been added into the wideband, lowpass filters and down-samples the copy to the envelope domain, and adds this back to the decimated wideband signal. In the current implementation the NAUC only listens to the microphone in the left ear. It would seem unnatural if each ear adapted individually and only one side ever compressed individually. Dual ear microphone sensing would also negate almost all stereo effects. Instead we have decided to only listen to one microphone and compare it to the signal in that ear. To save complexity we could have looked at the L+R/2 signal level we already generated for the Dynamic EQ. The left microphone was chosen because the DSP itself will be put in the left ear cup since there is more available space due to the power management circuitry being in the right ear cup. We want the microphone analyzed at to be physically close to the digital processing so that less analog signal corruption would be possible.

The microphone signal gives us a representation of the Signal (modified by the transfer function from the speaker to the microphone) plus the Noise component. First, the microphone signal is band-limited to 800 Hz and down-sampled to the same envelope-domain frequency as the audio signal. The result of this is then high-passed to above 80 Hz. This results in an 80 to 800 Hz bandwidth for the audio and microphone signal being input to the NAUC level detectors. In systems such as the Bose AudioPilot® car equalization system the speaker-to-microphone transfer function can change as people move and the environment changes. As a result complex adaptive filtering is needed. In our implementation the transfer function is relatively constant (somewhat user ear and head-shape dependent) to begin with, but is then desensitized to these variations by the closed loop system that provides active noise cancellation thus allowing us to avoid complicated adaptive filtering. When the AudioPilot system has determined its transfer function via adaptive filtering it is able to isolate noise from the microphone signal (which contains both noise and audio from the speaker), given that audio signal, by implementing a reverse convolution. We use our closed loop stable transfer function, input audio level, and the microphone level to determine a Signal to Signal Plus Noise Ratio (SNSR) rather than a simple SNR. This saves a great deal of computation by not having to explicitly isolate the noise signal.

The high pass filter is a 12th order Chebyshev type II filter. It operates with a sample frequency of 2,400 Hz, has a stopband frequency of 80 Hz, a passband frequency of 100 Hz, and a stopband attenuation of 60 dB.

The result of the band limiting filters is then RMS detected and converted to dB units in the same way the Dynamic EQ level was detected. This results in a Signal envelope level. This dB level is then converted to dBSPL by adding a factor of 110 as it was for the Dynamic EQ.

3.6.3 Microphone Signal Level Detection

The microphone signal is unbuffered in the same way that the input audio was unbuffered from 256 sample blocks to a sample-by-sample streaming input. This microphone input is then multiplied by a factor of 20. This scaling is analogous to the multiply by 5.8 to the input audio. Now, however, instead of wanting a voltage to eventual dBSPL scaling we are thinking in the opposite direction. We want 110 dBSPL on the input to map to 1 Vrms seen by the detector. This factor could be moved to the eventual dBSPL conversion in the same way the input audio factors could be condensed.

The microphone signal is preprocessed in the same way as the input signal is preprocessed for the NAUC. It is lowpass filtered, down-sampled, highpass filtered, converted to RMS, and converted to dBSPL in the same manner as the audio input signal. This filtering results in a Signal plus Noise envelope level.

3.6.4 NAUC Gain Calculation

Like the Dynamic EQ, the NAUC maps a gain value onto a two-segment piece-wise linear function. The NAUC has three parameters specified externally to the Simulink block and three internally specified parameters. The first is the break point gain specified as $G_{BreakPoint}$ (dB); this is the gain of the system at the break between the first and second piece-wise linear segments. When the microphone level is low, we assume that compression should be slightly less agressive. To accomplish this we lower the break point gain, thus decreasing the upper slope (more on this below). The second parameter is the 0-dB-boost (no gain) intersection specified as BP_{0dBInt} (dB); this specifies what the Signal to Noise Plus Signal Ratio is where the mapped gain slope intersects with unity gain. These first two parameters specify the slope of the first piece of the function. The third parameter is the slope of the lower piece in the function specified as $Slope_{Low}$. The internally specified parameters are the maximum gain, the SNSR level where the break point occurs specified as BP_C, and an expansion region for low levels, but these are all internally specified and are not as simply modified as the other three parameters.

Let us consider the general operation of the NAUC without worrying about the expansion region or the automatically adjusting breakpoint gain. First, the microphone level is subtracted from the input signal level. This results in a Signal to Noise Plus Signal Ratio (SNSR in dB units). The system then looks at this ratio. If it is greater than 3 dB, then the system applies the low compression gain, if it is less than 3 dB then the high compression gain is applied; remember that the SNSR now goes from 0 to 1 linearly or negative infinity to 0 on a dB scale (See Figure 3-2).

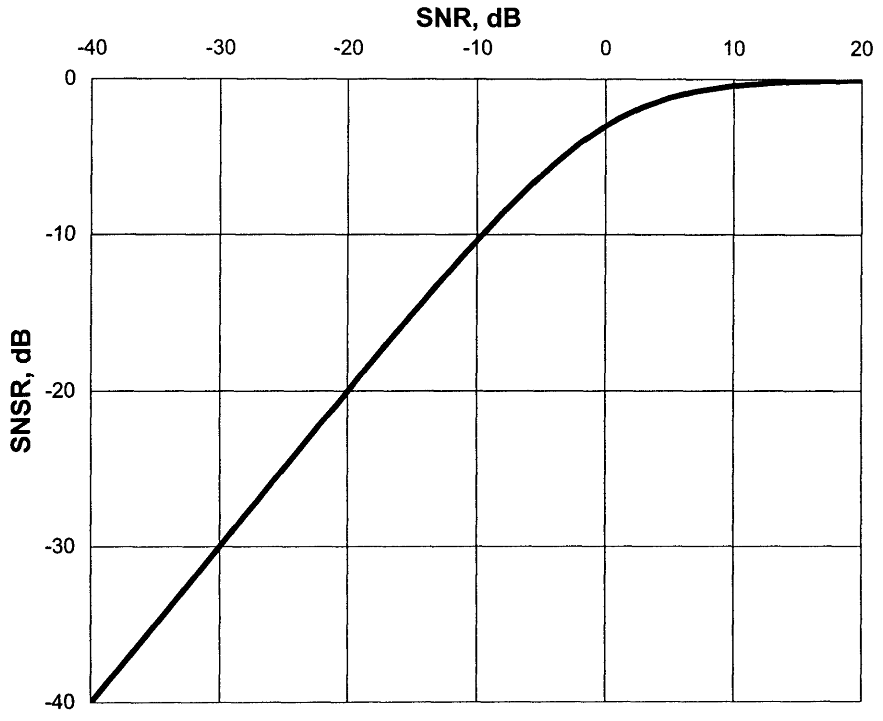

Figure 3-2: Plot of how SNSR relates to SNR

First, we will consider the high compression gain. This is the region when the SNSR is below the breakpoint in the two-segment piece-wise linear function. First, the distance below the breakpoint is calculated by subtracting the SNSR from 3. This gives us Δx. We then multiply this result by the slope giving us the Δy; this result is the amount of gain we need to apply.

The low compression region is slightly more complicated because it is defined with a breakpoint gain and a 0 gain intersect rather than simply a slope. First, we subtract the breakpoint SNSR (-3) from our input SNSR now to find how far above the breakpoint level our signal is. We now have to calculate the slope. We subtract the breakpoint SNSR from the 0 dB intersect to give us the Δx component. The breakpoint gain (3 dB) gives us the Δy component. We then divide the Δy by the Δx to give us the slope. This is multiplied by the distance above the breakpoint we are. In the final implementation the slope would simply be defined as a constant, but defining the slope this way is useful in prototyping where it is constantly adjusted. The result of this multiple is subtracted from the breakpoint gain to give us a gain that falls from 3 to 0 as the input increases.

We do not want the gain to increase arbitrarily even if we do not enter the expansion region. The output is then sent through a *min* function with 25. This yields a maximum gain of 25 dB. The result of this is then sent through a *max* function with 0 to set the minimum gain to 0 dB. For a graphical interpretation of the NAUC gain characteristics see Figures ?? and ??.

$$G_{dB} = \begin{cases} 0 & SNSR < BP \\ G_{BreakPoint} * (1 - \frac{SNSR - BP_C}{BP_{0dBInt} - BP_C}) & BP_C < SNSR < BP_{0dBInt} \\ G_{BreakPoint} + (BP_C - SNSR) * Slope_{Low} & SNSR < BP_C \end{cases}$$

Description of the Gain Calculation not including the Expansion Region or Maximum Gain Limiting

The break point gain is actively adjusted so that it provides us with more accurate characteristics. The theory is that when the mic level is low we have a better representation of our audio signal. A lower microphone level means that we have both a lower audio source as well as lower background noise. In low background noise conditions we can be slightly less aggressive in our compression especially when on the low compression slope. Lowering BP_C as the microphone level falls allows us to achieve this functionality. To do this we first run a minimum function on the breakpoint gain and the mic signal minus 30 dB (this was a constant experimentally arrived at). This means that when the mic drops to low levels (under 33 dBSPL) that the break point gain will also start to fall. The result of this is then sent through a maximum function with zero to make sure we never have a negative gain. This is possible because the low compression slope is not set explicitly but rather by the 0 gain intersect and the break point gain. In our implementation this adjustment only results in minor tweaking of the BP_C value because it can only range from 0 to 3 dB.

$$BP_{C Actual} = \begin{cases} 0 & Input_{Microphone} < 30dBSPL \\ Input_{Microphone} - 30 & 30dBSPL < Input_{Microphone} < 33dBSPL \\ 3 & Input_{Microphone} > 33dBSPL \end{cases}$$

Active Breakpoint Gain Adjustment

The NAUC then determines if the signal is low enough that it should enter the expansion region. At low levels the gain decreases back to zero rather than increasing to the max gain. This decision is noise independent, yet the noise level effects the slope. The purpose of the expansion region is to prevent compression swells during off periods or between tracks. If the gain goes to its maximum during off sections the system only amplifies the noise floor of the device it is plugged into. If the device has a poor noise floor, then this amplification can be quite obtrusive. The low signal expansion region helps to negate this effect. To make this decision the system checks to see if the input is greater than 30 dBSPL (a value experimentally arrived at). If the input signal is greater than 30 dBSPL then it is simply passed to the NAUC. If the input is less than 30 dBSPL then the expanded result is passed instead. If expansion

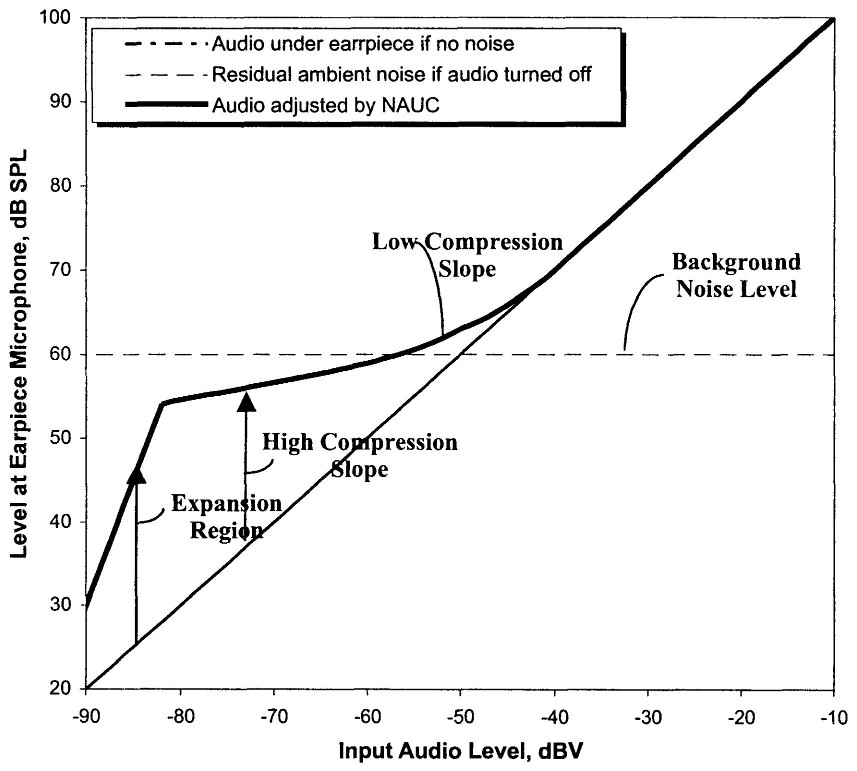

Figure 3-3: Plot of the NAUC Compression Curve

is deemed necessary then the input is subtracted from 30 giving us how far into the expansion region the signal is. This is then multiplied by the microphone level, and divided by 15. This is then passed as the signal. As a result the SNSR will increase as the signal level decreases. The reason the microphone level was used rather than a constant is that this promises that the SNSR will increase regardless of arbitrarily loud background noises. The factor of 15 is to offset the multiple of 30 encountered in the equation:

$$SignalLevel = \begin{cases} Input_{Audio} & Input_{Audio} > 30dBSPL \\ Input_{Audio} + \frac{(30-Input_{Audio})*Input_{Microphone}}{15} & Input_{Audio} < 30dBSPL \end{cases}$$

Expansion Region Equation

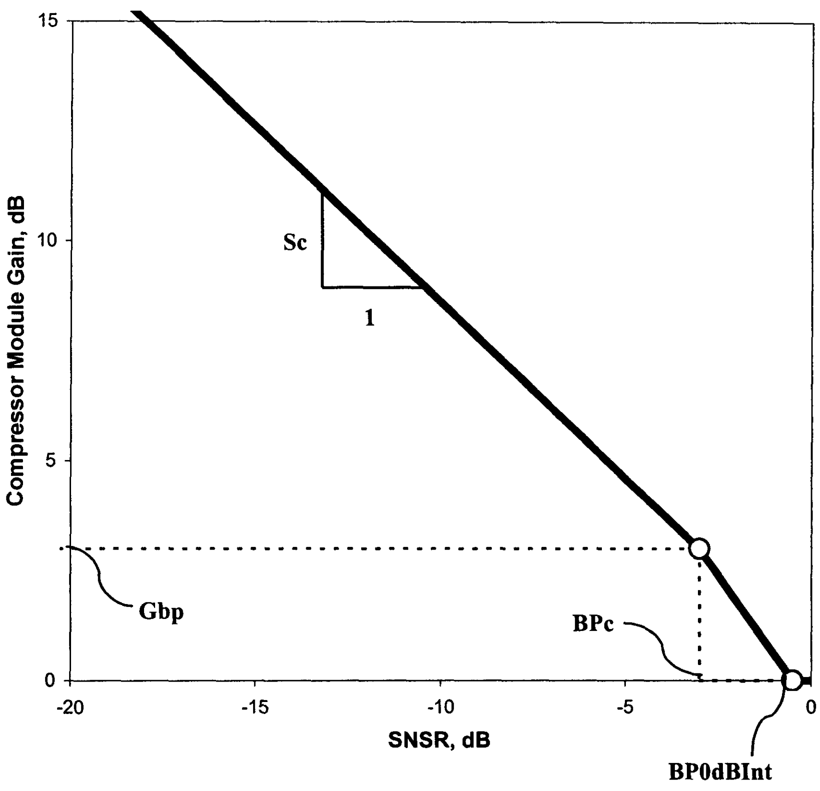

Figure 3-4: Plot of the NAUC Gain Curve

3.6.5 NAUC gain slewing

One of the more annoying features of a poorly designed compression scheme can be a sudden gain leap that result from impulsive noise. For example if the user coughs or taps their ear cup, then a large amount of noise is present for a short time. We have already configured our microphone RMS detector to increase rapidly so that it can respond quickly to rapid audio level increases (ensuring an accurate SNSR value in constant noise). We still want our RMS detector to increase rapidly so that it can quickly detect audio changes in constant noise levels, but we do not want it to quickly jump up in response to impulsive noise and stay high. The ideal impulse rejection scheme would be a median filter. This type of filter will preserve sharp edges but will also reject impulsive noise. Unfortunately a median filter is non-causal and thus is not practical for processing real-time noise, because the signal must be delayed for an amount of time equal to the longest impulse duration that needs to be rejected.

As a fix to the impulsive noise problem we simply rate limit the adaptation of our gain. We now let the gain fall off rapidly but only increase at a fixed rate. As a result, the microphone dBSPL will quickly jump up in impulsive noise, the intended gain will also jump up, but the output will only raise slowly. If the noise is impulsive the microphone level will fall off, resulting in the intended gain decreasing, as the slewed gain slowly increases. When the intended gain intersects the slewed gain the slewing will end and the intended gain will again become the actual gain. There is, however, a trade off in the adaptability of the NAUC and the amount of slewing used. If there is a sudden yet constant change in background noise our gain will only increase at a rate dictated by our rate limiter. Compression pumping is also often noticeable, but it is no longer uncomfortable. Our rate limiter is set to a rising rate of 20 dB per second. The output of the slew function is then once again confined to the minimum and maximum gain for redundancy and to protect against the rate limiters startup conditions. This output is then delayed by one sample in order to synchronize the compression calculation

3.6.6 NAUC Level Processing

First the gain must be up sampled back to the audio domain. For the most part this works the same as the up sampling conducted in the Dynamic EQ stage of the system. The output is zero padded up from a 2,400 Hz signal to 48 kHz. Then the sample and hold scheme that was used in the Dynamic EQ is applied. Now, however, we must use an anti-zipper filter to smooth the transitions. The NAUC gain operates on the wideband spectrum of the music, so any type of a sudden gain transition is noticeable. The filter is called an anti-zipper filter because sudden gain transitions have a sound similar to that of a zipper. Applying a lowpass filter serves to modify the sudden jumps so they become more gradual. The sample and hold in combination with a causal lowpass filter provide a good approximation to an ideal lowpass filter. Our anti-zipper filter has the same characteristics of the lowpass filter used in the initial down sampling.

The level processing itself is much simpler than that used in the Dynamic EQ. In fact it is simply a matter of multiplying the wideband signal, after the Dynamic EQ gain has been applied, by the gain calculated by the NAUC. This scaling is conducted individually onto both the left and right channel audio sources with the same gain value.

3.7 Digital Dewey

The digital Dewey filter, named after a critical listener who qualified the equalization curve of the QuietComfort 2 headphones, is simply an 8th order IIR filter. To characterize the filter we first had to characterize the response of the headphones after we removed all the analog components that only affected the audio path response. The analog response was characterized through Orcad simulation and verified with direct measurement. We then determined the filter that would have to be placed in series with this modified analog response to give us the desired total response. For the floating-point, Simulink model a simple direct form filter was implemented. In embedded code the filter should be implemented as second order segments through either cascade parallel form [2] to minimize fixed-point round off errors.

3.8 Final Output

After the Digital Dewey filter has been applied the final output is rebuffered and scaled. The buffering is necessary because the DAC block operates on a 256-sample data block in the same way the ADC gave us a block of 256 samples originally. The result of this buffering is scaled by 0.2022 so that a signal of 1 Vrms will lead to 1 Vrms on the output. In a computationally efficient system this scaling would be combined with the level detection scaling

[2]This is discussed in Oppenheim and Schaffer. [8]

Chapter 4

Embedded Code

The initial code was written to operate on the Blackfin EZ-Kit Lite Evaluation board. Additional changes will be needed to interface the system with our desired AKM CODEC and ADC chips. Currently the EZ-Kit uses the ADI1836 audio stereo 48 kHz CODEC[1] Our system was written to work with this CODEC. In order to configure the system to work with the external converter parts the programmer must specify control and I/O pins and program the initialization files to account for these. Our interface accepts the stereo audio signal in first of the two audio inputs (J5)[2]; the microphone signal is sent to the Left Channel of the second analog input; and the speaker output is taken from the first analog output (J4).

The code was expanded from the ADI "Talkthrough" template, which functions to run the CODEC without any processing conducted on the data. This code initializes the CODEC to operate in 48 kHz dual channel TDM mode. This is an ideal starting point for our system since we need 3 input channels (Left, Right, and Microphone) and we desire a 48 kHz sample frequency. The Talkthrough code executes the code in a file called Process_data.c once per every 48 kHz sample. Originally this simply assigns the sampled input to the output. From here we are able to insert code to process the audio samples. (See Appendix B for all of the code used in the project)

[1]See: "ADSP-BF533 EZ-Kit Lite Evaluation System Manual"

[2]See P. 3-16 of "ADSP-BF533 EZ-Kit Lite Evaluation System Manual"[2]

4.1 Conversion to the Envelope Domain

Much of the signal processing is done in the envelope domain of the signal. Chapter 3 discussed the algorithm in detail, and from it we see that much of the computation is done at a rate one twentieth of the sample rate. Likewise, in the embedded algorithm, much of the computation is also done in the envelope domain.

4.1.1 Anti-Aliasing

The first step in down sampling the operating frequency is to anti-alias the input signal. For the embedded anti-aliasing filter we are looser on the tolerances than we were in the Simulink version. All filters in the embedded code are made out of second order section biquads. Each IIR Biquad is made as:

$$y[n] = b_0 * x[n] + b_1 * x[n-1] + b_2 * x[n-2] - a_1 * y[n-1] - a_2 * y[n-2]$$

The looser the parameters on a lowpass filter, the fewer biquads it takes in order to complete the filter, and thus the fewer computations the DSP must execute in a sample. In the embedded version the filter was designed with Matlab's Filter Design Tool to have a F_c of 800 Hz, a F_{stop} of 1300 Hz, and a stopband attenuation of 60 dB. The filter was specified as a Chebyshev type II; this filter type is flat in the passband with ripple in the stopband. Because we are attempting to eliminate the audio signal in the stopband, we do not care about ripple in this region. Also, Chebyshev II filters give us greater attenuation per order than a Butterworth filter would (flat in both the pass and stop bands). The result is a eighth order lowpass filter rather than the 11th order filter that was used in the Simulink Algorithm.

The first anti-aliasing filter that operates on the audio signal takes as input the the average of the left and right channel signals. We accomplish this by shifting each signal to the right by one binary value (the equivalent of divide by 2) and then summing the signals. We divide before we sum to avoid digital overflow even though this results in two binary shifts rather than one.

4.1.2 Down Sample

In order to achieve the down-sampling we simply look at the signal of interest once every 20 samples. This yields a down-sampling to 2400 Hz from 48 kHz. We must be careful to run the signal of interest through a lowpass filter to avoid aliasing of high frequency signal components. In order to look at every twentieth sample we write:

```
$if(samp++ \quad \%20 = 19)$
    {
            $samp = 0;$
            $\ldots$
    }
```

This causes the variable *samp* to increment once per sample with modulo 20. If *samp* is equal to 19, then *samp* will be reset to 0 and the rest of our logic is run. This logic includes setting some storage variable equal to our input. The result of setting the storage variable only when the sample modulo condition is met results in the down-sampling. We can then run all of our envelope domain logic within the sample modulo *if* statement.

4.1.3 Level Detection

For dynamic range reasons we now run the level processing on the absolute value of the input signal rather than the square. Music can be listened to over a wide dynamic range; this could range from 40 or 50 dBSPL for quiet music to 100 or 110 dBSPL for loud music. Clearly a wide dynamic range on our detector is also needed. A range of 70 dB is not an unreasonable amount of dynamic range to expect out of a commercial ADC. In the Simulink Algorithm, however, we run the linear to dB conversion on the square of the input rather than the input itself. This means that instead of the standard dB equation we would run:

$$Output_{dB} = 10 * log(Input)$$

So as a result a dynamic range of 70 dB in the dB domain requires us to have a dynamic range of 140 dB in the linear domain before the dB conversion is run. Also, by running the square function we are throwing away half of the dynamic range already due to the way a fixed-point multiply functions.

The easiest solution to the problem was to rewrite the RMS detection function to operate on the absolute value of the input rather than the square. This accomplishes the same functionality with the difference that we run the standard dB equation rather than the equation previously stated. This allows us to compute a level detection that has over the required dynamic range in the log space using the same amount of dynamic range in the linear space.

The level detector functions using a feedback decay methodology. It looks at the input and outputs the input plus a feedback factor. If the input is greater than the previous input, then the output is simply set to the input's value, resulting in an instantaneous attack time. If the input is less than the previous output then a fraction of the difference between the previous output and the input output is subtracted from the previous output resulting in a slow decay time. The result of the system should be an operation that gives a DC output voltage that is proportional to the peaks of on input AC signal in much the same way a RMS level detector would work.

The level detector should operate on a range from 0 to 1. This means that the log operation run on the RMS detector will yield a range from negative infinity to 0. As a result of the noise floor of the ADC and the slow decay time associated with the level detection we will never have a 0 input. To achieve the linear to dB conversion we simply use the floating-point operation logf. This causes the DSP to use its floating-point emulation capabilities. Since 1 Vrms corresponds to 110 dBSPL inside the ear cup, we must add a factor of 110 to the result of the logf operation to get a sound pressure level value.

4.2 Dynamic EQ

Like in the Simulink Model the purpose of the Dynamic EQ is to apply a gain to the bassband of the audio signal in order to compensate for low level hearing attenuation. The specifics of the Dynamic EQ are documented in Chapter 3. In order achieve this components functionality we monitor the input audio levels; calculate a Dynamic EQ gain based on the input level; apply this gain to a filter designed to isolate out the frequencies of interest; and finally sum the result back in with the wideband audio signal in order to create an output where the equalization varies with input audio level.

4.2.1 Dynamic EQ Envelope Domain Computation

Like the Simulink algorithm the embedded code works by implementing two Dynamic Range Processor (DRP) units. Each DRP looks at the input dBSPL and calculates the its distance below the DRP break point. If the input level is below the breakpoint, then an intermediate variable is assigned to the distance multiplied by a slope value. If the input level is above the breakpoint, then the intermediate variable is assigned to 0. The minimum of the result and the Maximum Gain value is then assigned to the final gain value of the DRP. The gains of the two DRPs are then summed together to obtain the resultant Dynamic EQ gain value.

After we have computed the gain in terms of dB units, it must be converted to a linear factor in order to be applied to the audio signal. As stated in Chapter 2.2.3 the powf function uses too many MIPS (Millions of Instructions Per Second) for our processor to handle. As a result we drop samples and hear a fuzzy output. To remedy this a fifth-order Taylor Series expansion of the function $Gain_{Linear} = 10^{\frac{Gain_{dB}}{20}}$ is implemented to convert the dB gain to a linear term. The output of this conversion is a emulated float value; it is necessary to use a float in this case because fractional gain components are important for smaller gain values. Following the conversion we must subtract 1 to get the final gain value. One is subtracted to account for the low frequency components present in the wideband content of the signal. To visualize this imagine that we calculate a 0 dB gain; this will result in a gain of 1 after the undB operation is run; if we apply a gain of 1 to the Dynamic EQ biquad and sum the result in we will actually double the bass components; so to account for this we subtract 1 from our gain term.

4.2.2 Dynamic EQ Sample Domain Computation

Before we apply the Dynamic EQ gain to the input signal we must first run it through an anti-zippering filter. As discussed in Sections 3.4 and 3.6.6 an anti-zippering filter is a lowpass filter whose purpose is to smooth the transitions between concurrent gain values. The anti-zipper filter we implemented was designed with Matlab's Filter Design Tool to be a first order Butterworth filter with a cutoff frequency of 10 Hz. We used a Butterworth filter because we are more concerned with just creating a smoothing effect between gain transitions rather than actually eliminating higher frequency components. Smoothing transitions in discrete-time analogous building an integrator in continuous-time analog where frequencies who fall on the downward slope of the filter response will be integrated. Because the gain terms are relatively small (relative to the maximum fixed-point values) we can multiply them by 100 before we apply the anti-zippering filter. The anti-zippering filter returns an integer value, and we want to preserve part of the fractional gain component. We then divide the result by 100 to get the correct value that is stored as a float.

This gain value is then applied to the Dynamic EQ biquad filter. The biquad filter is second order and designed with Matlab's Filter Design Tool. As we discussed in Chapter 3.5 this paper has omitted the parameters due to IP reasons. The topology of the filter code is the same as the second order sections used in the lowpass filters. The biquad is run just on the left channel input. This saves the computation of running two individual Dynamic EQs on each channel. Also, most bass components tend to be monaural effects so the added benefit of having two independent Dynamic EQ filters would not outweigh the added costs of computation.

4.3 Noise Adaptive Upwards Compression (NAUC)

The goal of the embedded NAUC algorithm is to apply upwards compression to the audio signal in order to guarantee that the audio signal will always be audible over the background noise. This can be used to either bring quite passages up into the audible region or to allow the user to set a maximum listening level so that sudden loud passages will not be obtrusive while the quite passages will still be audible. The details of this algorithm are discussed in Chapter 3.6.

4.3.1 NAUC Envelope Domain Computation

To begin the NAUC computation we first have to run a level detection on both the audio signal and the signal seen from the microphone. For the audio signal we run the level detection on the original signal summed with the boost component resulting from the Dynamic EQ in order to more accurately reflect the signal that the listener hears. To do this we run an anti-aliasing filter on the low frequency component of the Dynamic EQ (the result of the biquad filtered signal multiplied by the Dynamic EQ gain); down sample this signal to the envelope domain; and sum the result with the already down sampled audio signal we have previously processed. This method of computation is slightly different than the way we computed the NAUC signal level in the Simulink algorithm where we ran the level processing only on the left channel audio component. By running the NAUC on the stereo audio level we computed in the Dynamic EQ, though, we are able to avoid an extra anti-aliasing filter. We then run the level detection in the same manner as we did for the Dynamic EQ. We also have to detect the input level as seen from the microphone. We anti-alias filter the microphone signal the same way we did the audio; the microphone is seen only from the left ear cup so we do not have to worry about stereo considerations. We then down sample and level process the anti-aliased signal. We convert both the audio signal level and the microphone signal level from linear space to dB space using the logf function as we did with the Dynamic EQ; when we convert the microphone level to dB space, though, we must add a different constant in order to obtain dBSPL units because the microphone has a different sensitivity than the speaker. In order to have sensible level processing both the audio signal and the microphone level are converted to dBSPL units; dBSPL units allow the programmer to view values in dB space in an absolute sense while standard dB yield values that are merely relative. In the Simulink design we also highpass filtered the audio and microphone signals to be above 80 Hz. We did this to limit the sample content to the spectrum where the active noise cancelation feedback path creates a flat response as seen from the speaker to microphone transfer function. In the current implementation the highpass filter was not used to save on computation. This could be included if low frequency oscillations in the NAUC performance prove to be a problem.

Once we have processed the audio and microphone levels we can run the logic to compute the NAUC gain term. For detailed description of the theory behind the NAUC see Chapter 3.6.1. The embedded NAUC operation only takes two inputs; the input signal level and the microphone level. The other parameters: the break point gain, the low slope, and the high slope(originally the 0-dB-boost and the break point location) as well as the expansion region parameters have been hard coded into the system. The embedded NAUC has three steps in its operation: a low-level expansion region, a mid-level upwards compression region (standard operation), and a high-level downwards compression region. The low-level expansion is a safety precaution region that prevents us from amplifying regions deemed to be sourceless (this could include an unplugged audio source or segments between tracks). If the audio sample is less than 30 dBSPL, then it is deemed to be sourceless in which case the gain is decreased rather than increased. The expansion region logic is calibrated such that if the microphone signal level is held constant and the audio source is decreased, then once the audio signal drops below 30 dBSPL the gain will decrease starting at what it was at 30 dBSPL (in other words if the NAUC has the gain set at 20 dB when the input is slightly above 30 dBSPL the gain will be 20 when the input is slightly below 30 dBSPL).

If the input signal is above 30 dBSPL, then the NAUC operates in its normal compression region. First, the breakpoint gain is adjusted if the microphone level is low. This allows us to actively adjust the NAUC operation based on background noise conditions. Then, if the signal to microphone ratio (Signal to Signal plus Noise or SNSR) is less than -3 (the SNSR point we chose for our break in slope of our two piece wise linear function) we apply the low level gain based on how far away from this point we are. If the SNSR is greater than -3 we apply our high level gain based on how far above this point we are. We then set our maximum and minimum gain values at 25 dB and 0 dB respectively.

The third region of operation is the high level downwards compression region. This region is not present in the floating-point operation, because it is a safety precaution designed to reduce the amount of fixed-point digital clipping we hear. If the NAUC causes high gains to occur in high signal conditions (if there is a high level audio source present in high level background noise situations), then we may overflow the number of bits we are able to compute. If this occurs our signal will start to distort. In order to prevent this we check to see if the level input to the NAUC plus the gain is greater than a certain dBSPL threshold. If it is then we linearly, yet rapidly, decrease the amount of gain applied to the output. Again, we restrict the result of this to values between 0 dB and 25 dB.

Finally, we scale the gain value from a dB value to a linear multiple using the same Taylor Series expansion as we did for the Dynamic EQ system. Both systems yield gain values that have about the same range. As a result the same Taylor Series expansion is fairly accurate for each operation.

4.3.2 NAUC Sample Domain Computation

Once the NAUC gain term has been converted to a linear multiple we are able to process the audio signal. The audio domain processing for the NAUC is fairly straight forward. First we must anti-zipper the gain values using the same methodology as we did in the Dynamic EQ operation. Now, however, we want to preserve all of the stereo effects because we are operating on the wide band. So, we multiply the anti-zippered NAUC gain term individually to both the left and right stereo channels. This gives us our compressed audio signal.

4.4 Digital Dewey and Final Output

The Digital Dewey Filter for the embedded system was constructed using the same parameters as the Simulink Dewey Filter. We originally classified the Digital Dewey as Second Order Section IIR coefficients. Like the Simulink Dewey the embedded version is eighth order made up of four second order sections. So, in order to build the embedded Dewey we used the same coefficients, and it was constructed using the same topology as we did with all the other embedded filters. Because the Digital Dewey operates directly on the audio signal we used floating-point emulation for the coefficient multiplication; this was done with the goal of preserving as much fidelity as possible. Matlab tests show that the first second order section needs six significant figures of precision after the decimal point for stability while the last three only need five. Also, to avoid digital clipping we must be sure to apply the gain term (less than 1 and preserves the 0 dB maximum gain of the filter) in the first section of the filter.

The Digital Dewey is simply applied individually to both the left and right stereo channels after all the other computation has been accounted for. Here is a very top level view of the operations:

$$\begin{aligned} DynEQ &= Gain_{DynEQ} * BPF(audio); \\ Output &= Dewey(Gain_{NAUC} * audio + DynEQ); \end{aligned}$$

The result of running the Dewey filters on both the left and right channel is then sent to the output. The system process then sends this output to the CODEC once per cycle. The output is finally converted back to an analog signal where it is sent to the J4 audio output on the BF533 evaluation board.

Chapter 5

Further Work and Conclusions

5.1 Further Work

5.1.1 Polishing the Embedded Code

There are several aspects of the embedded system that need to function better than they do in the current implementation. The aspects of the embedded system that were most troublesome were those components that interfaced with the actual audio signal. In general all the purely mathematical operations functioned well in the end. I define purely mathematical operations as those functions that do not necessarily act directly on the signal but instead compute gain factors based on qualities of the signal. In our system the purely mathematical functions were the gain calculations for the Dynamic EQ and the NAUC. The problem areas were operations such as filters and level detection.

If gain components are placed in the wrong biquad either digital clipping or noise floor amplification can occur. Floating-point emulation fixed many of these problems as long as the gain was embedded into the correct second order segment. It is important to make sure that each segment has a maximum gain of 0 dB or else digital clipping could easily emerge. Noise floor amplification occurs when we have an attenuation followed by a multiplication. The earlier in the filter the gain term (in our case the gain is less than 1) occur, the more noise floor amplification occurs.

Basically, the sooner we attenuate the signal the more robustness we have against digital clipping, but the more possible noise floor amplification there is. If the gain is embedded into the filter coefficients correctly, then the noise floor amplification will be kept to a minimum.

Ideally all of the floating-point filter coefficients would be converted to fixed-point fractional multiplication. If we scale the filter coefficients to magnitudes less than 1 and undo the scaling on the output term before it is sent to the delay line, then we can compute the filter outputs using only fixed-point, fractional arithmetic. We should also limit our scaling to dividing by multiples of two so that they may be accomplished with binary shifts. If we implemented a fixed-point filter where we scaled the coefficients by a factor of two the logic would look like this:

$$\begin{aligned} y[n]_{frac} = \frac{b_0}{2} * x[n] + \frac{b_1}{*} x[n-1] + \frac{b_2}{2} * x[n-2] - \frac{a_1}{2} * y[n-1] - \frac{a_2}{2} * y[n-2] \\ 2 * y[n]_{frac} = y[n] \end{aligned}$$

We could then use the fractional multiply and add functions for fixed-point computation. When we attempted to implement this code a problem arose in extracting the correct bits. Fixed-point multiplies that result in a 32-bit output are only able to multiply two 16-bit inputs[1] The Blackfin libraries have extract functions designed to extract either the high or low order bits in a fractional number. These functions, however, produced unexpected outputs much of the time. We believe that appropriately shifting a 32-bit value and manually setting the result to a 16-bit variable would alleviate much of this problem. The only case where fractional, fixed-point arithmetic would be impossible would be cases where more precision is required than 16-bits can provide. For example, the first second order section of the Dewey Filter requires six significant figures of precision after the decimal point for stability; if we only gave it 5 decimal values of precision low frequency oscillations occur. This instability was observed in implementation and was verified by the Matlab Filter Design Tool.

[1]The logic can be easily shown that every bit in a digital multiplication results in two bits on the output. So, for every bit you multiply you need two bits of storage in the output. We have 32-bits in our ALU so only 16-bit multiplication is possible.

The level detectors still need to be accurately calibrated. Three problems remain: the first is the problem of preserving dynamic range after the anti-aliasing filter has been applied; the second is accurately setting the linear to dB conversion so that true dBSPL levels are represented; the third is appropriately setting the rise and fall times of the operation. By applying a digital filter much of the dynamic range is going to be lost. The ADC on the BF-533 evaluation kit provides a 24 bit digital input, but due to restrictions on our ALU we can only multiply 16 bit numbers. Thus as soon as we apply a digital multiplication we reduce our dynamic range to 16 bits. We must take care to preserve as much of this dynamic range through the course of the filtering process as is possible. If we carefully designed each stage in the filter rather than relying on Matlbab's filter generation, then more dynamic range preservation could be possible.

Assuming we have accurately down-sampled the input signal, we must then accurately convert our detected level to a dBSPL level. The problem we encounter is that different signals translate to different dBSPL sensitivities. A clear example of this is that 100 mVrms at the speaker creates a different sound pressure level than the sound pressure level needed to create 100 mVrms at the microphone. As a result different dBSPL conversion factors are required for both the input signal level detector and the microphone signal level detector. We, also have to clearly characterize the benefits of level detecting on the original signal plus the Dynamic EQ base component versus just reusing the already existing level in the NAUC computation. In the Simulink model we did a dedicated level detection acting solely on the left channel taking into account the boosted base component. Simply reusing the level computed for the Dynamic EQ though may yield results that are accurate enough for our purposes considering that over time the left and right channels tend to be similar levels and noise in the left ear cup is usually similar to noise in the right ear cup. Finally, we need to adequately characterize the appropriate rise and fall time characteristics of our level detection. This is characterization is somewhat objective. In the end these features must be determined by critical evaluation by skilled listeners.

5.1.2 Algorithm Optimization

As stated in Chapter 2.3 further steps are needed to optimize the embedded code for performance and power. Implementing an all fixed-point version of the embedded code would be the first major step in optimizing the current algorithm. This means eliminating floating-point computation from the IIR filters (as stated above) and eliminating floating-point computation from the envelop domain computation. The later is the more complicated of the two. Removing floating-point from the dB conversion is going to either require clever conversions between integer and fractional notation or dividing numbers up into separate fractional and integer parts that would have to be combined in some way as to yield results equivalent with operating on non-integers greater than 1.

Once all floating-point computation is removed from the system we then have to decide which operations are not truly necessary in order to obtain our desired performance. This may mean using the same level detector for both the Dynamic EQ and the NAUC (as proposed in section 5.1.1). There could also be other places where computation could be reused in order to save clock cycles. Also, the implementation of the system could be converted from a sample by sample topology to a block processing topology. In our implementation we run all the computation on each sample individually. In a block processing implementation we would save a larger vector of samples (say a block of 100 samples) and once a full set has been acquired we would run the computation on the entire block at once. This would saves the set up time associated with starting a process where for example coefficient values must be read from cache and stored to registers. In a block process this setup happens only once per block rather than once per sample.

Finally, once we have optimized the algorithm we can find the minimum clock cycle to set the processor to. This involves determining how many processor clock cycles are used in a sample period and setting the processor clock appropriately. This can be done by programming the chip's PLL with a multiplication factor for CLKIN to set the Voltage Controlled Oscillator (VCO) clock. Then, the user can program values to divide the VCO clock signal by to generate the core clock (CCLK) and the system clock (SCLK)[2]. It is also possible to dynamically change the clock speed based on computation needs. This requires more in depth optimization coding, but it could allow us to further optimize our system. For example, if block processing was used we could speed up the core clock when a complete block has been collected so that it can be quickly processed in one sample period. Then, while the block data is being output and the next block is being collected we could drastically slow down the core clock, thus lowering the average power consumption.

5.1.3 System Implementation

The final step in developing a DSP prototype is the hardware design and implementation (See Chapter 2.4). Ideally this stage would be conducted in parallel with the software design by a second engineer. The first step in the hardware design is determining what pins of the DSP will be the control pins as well as signal I/O pins. Once this has been done (and eventually coded into the software) the layout for the connections between the CODEC and microphone ADC can be designed. At this point the evaluation kits should be connected and the software designer should implement the software such that the system is designed to communicate with the AKM evaluation kits rather than the ADI1836 CODEC and the J4 and J5 I/O ports.

Once this intermediary evaluation setup has been completed the actual layout design can commence. This mainly consists of determining where the new ICs and their supporting analog components[3] can fit onto the already existing analog layout. Fortunately we have already cleared a small amount of space by eliminating some existing analog components that were digitally replaced as stated in Chapter 1.3. This frees up some space on the board, but additional changes may be needed. In order to place the needed digital components into the system a new layout must be added that allows for the digital IC pin connections. Old parts may need to be rearanged in order to more efficiently use the available physical space. If there is still

[2]See: ADSP-BF533 Blackfin® Processor Hardware Reference Page 8-1[1]
[3]See the AKM4555, AKM5355, and BF-533 application notes.

not enough space or wiring is too difficult the board may have to be converted to a more complicated layer system. The current board has four wiring layers, but a complicated digital chip like the Blackfin may be easier to wire with more metal layers in the PCB layout. Power supply restrictions and implementation was discussed in Chapter 2.4.

5.2 Conclusion

We have found that DSP can add many interesting and useful functions to the audio signal processing in headphones. This functionality is particularly interesting in a set of noise canceling headphones that already have a microphone installed in the interior of the ear cup. In a system with such a microphone installed we are able to do noise level measurements and estimations to implement noise adaptive compression algorithms. There are other DSP algorithms that we were not able to explore such as "night mode" compression, active loop gain adjustment, and active noise reduction. These algorithms were not explored for varying reasons. Night mode was not explored because we built the NAUC instead. Active loop gain adjustment was not explored because we deemed this less important than other algorithms. Finally, digital active noise reduction was not explored because current technology prevented its implementation in our system.

This project would ideally be completed in three major phases of development. The first stage is the algorithm development. Algorithm development is the initial prototyping phase, conducted in Simulink in our case. This stage could be done either in Simulink or the simulation mode of a program like Visual DSP++. In retrospect it would aid in the development process to restrict this stage to adequately represent the final system. In our case we did not know the nature of the final DSP when the algorithm was developed, so limiting our resources in this stage could be counterproductive. Knowing what we know now, if a new algorithm were to be developed for headphone application at the very least this stage should be limited to a fixed-point operation set.

The second stage consists of the algorithm implementation and the hardware design. For the maximum efficiency these two stage aspects should be completed in parallel by two engineers. At this point the algorithm will be ported to the embedded design, and the hardware layout will be designed. The hardware designer should choose the I/O pins for the DSP and the software designer should program the system appropriately. This stage will take significantly less time if the algorithm development stage is conducted taking into account possible fixed-point limitations of the final system.

The final stage in the design process is algorithm optimization and final production. At this point the algorithm should be optimized for power consumption and the hardware designer should work on assembling actual prototypes. After all of this has been completed a final prototype will be complete.

One of the main discoveries made during this project's research process were just how much time can be saved by classifying the design space while still in the algorithm development stage. If we had limited the Simulink system to the fixed-point block set, then many of the fixed-point problems may have been uncovered earlier. The second discovery we made was just what the design space for headphone applications currently are. Based on commercially available DSP products, fixed-point processors are most appropriate for portable applications. This was discovered by designing our algorithm, classifying our power requirements, looking at the computational demands of our algorithm, and then researching available processors. We did not know the design space of our algorithm before the process began. So, in the end one of the most important discoveries we made was classifying just what the computational demands we require are, and then classifying the type of processor the algorithm must exist on. This will allow future headphone DSP designers to know their design restrictions when they start the process, a luxury not afforded to us.

Appendix A

Simulink Model

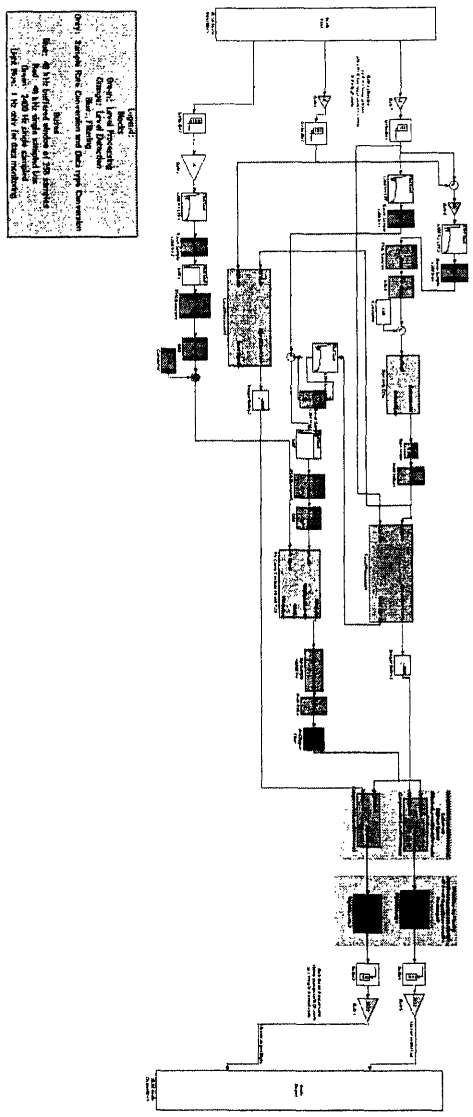
Legend:
Blocks
Green: Level Processing
Orange: Level Detection
Blue: Filtering
Grey: Sample Rate Conversion and Data type Conversion
Buses
Blue: 48 kHz buffered window of 256 samples
Red: 48 kHz single sampled bus
Green: 2400 Hz single sampled
Light Blue: 1 Hz only for data monitoring

Figure A-1: Top Level Simulink Model

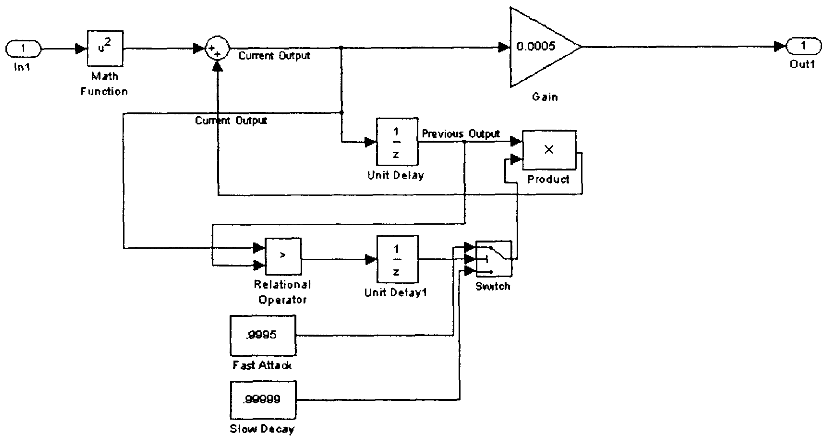

Figure A-2: RMS Detector

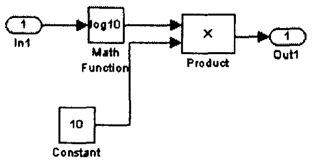

Figure A-3: dB10 Function

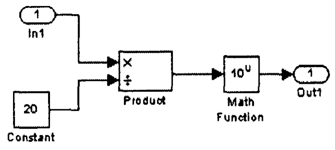

Figure A-4: undB20 Function

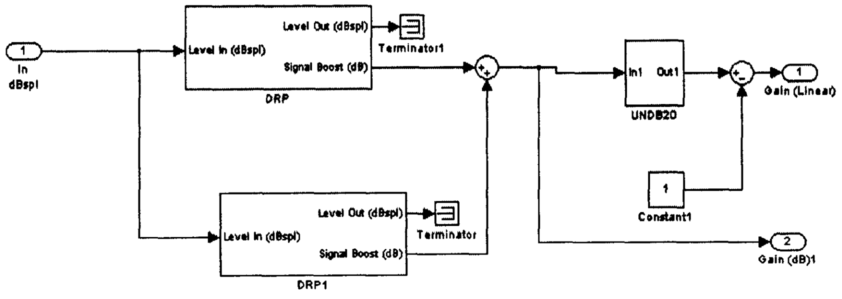

Figure A-5: Dynamic EQ

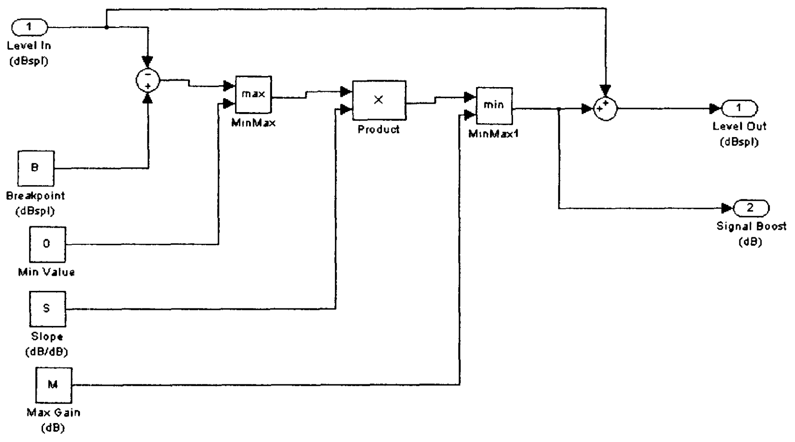

Figure A-6: Dynamic Range Processor (DRP)

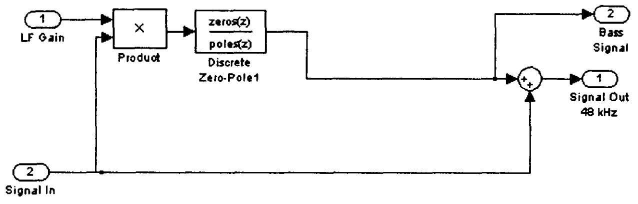

Figure A-7: Dynamic EQ Level Processor

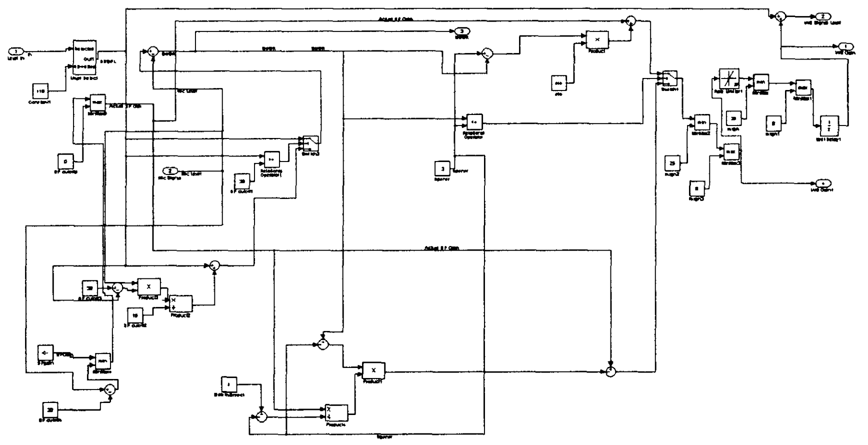

Figure A-8: NAUC

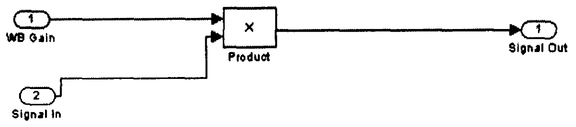

Figure A-9: NAUC Level Processor

Appendix B

Embedded Code

Figure B-1: Main Function

```
#include "Talkthrough.h"
#include <fract.h>
#include "math.h"
#include <stdio.h>
//--------------------------------------------------------------------------//
// Function:    Process_Data()                                              //
//                                                                          //
// Description: This function is called from inside the SPORT0 ISR every    //
//              time a complete audio frame has been received. The new      //
//              input samples can be found in the variables iChannel0LeftIn, //
//              iChannel0RightIn, iChannel1LeftIn and iChannel1RightIn      //
//              respectively. The processed data should be stored in        //
//              iChannel0LeftOut, iChannel0RightOut, iChannel1LeftOut,      //
//              iChannel1RightOut, iChannel2LeftOut and iChannel2RightOut   //
//              respectively.                                               //
//--------------------------------------------------------------------------//


fract32 env;
fract32 micenv;
fract32 sigenv;
float prev;
float micprev;
float sigprev;
float envsqr;
float micenvsqr;
float sigenvsqr;
float current;
float miccurrent;
float sigcurrent;
int samp = 0;

int left;
float dyngain = 0;

int miclow = 0;
int michigh = 0;
int basslow = 0;
int siglow;
int sighigh;
int micgain = 0;
float NAUCgain=1;
float appliedgain = 1;
int naucgaintemp = 1;

//Pre-computed parameters for Taylor Series
float a = 0.115129;
float b = 3.16228;

void Process_Data(void)
{



        fract32 LR, lowfreq;
        fract32 inpL = iChannel0LeftIn;
        fract32 inpR = iChannel0RightIn;


        fract32 halfleft = inpL >> 1;
```

```
fract32 halfright = inpR >> 1;
LR = L_add(halfleft, halfright);
lowfreq = LPFfloat(LR);


 //connect mic to the Left 1 input
 //mic prefiltering
miclow = LPFmic(iChannel1LeftIn);


//RMS detection scheme


if (samp++%20==19)//(samp >= 20)
{
        samp=0;
        env = abs(lowfreq);
        envsqr = fr32_to_float(env);


        if (envsqr >= prev)
        {

                current = envsqr;
        }
        else
        {
                float diff = envsqr - prev;
                current = prev+0.0005*diff;
        }
        prev = current;

        float rmsout = current;
        //rmsout should range from 0 to 1
        float rmsdB = 20*log10f(rmsout);
        int inputdB = rmsdB+110;

        //first drp
        //Dynamic EQ Characteristics Have Been
        //removed for IP reasons
        int dist1 = BP1-inputdB;
        if(dist1>=0)
                {
                        dist1 = Slope1*dist1;
                }
        else
                {
                        dist1 = 0;
                }
        int dyngain1 = min(MaxGain1, dist1);

        int dist2 = BP2-inputdB;
        if(dist2>=0)
                {
                        dist2 = Slope2*dist2;
                }
        else
                {
                        dist2 = 0;
                }
        int dyngain2 = min(MaxGain2, dist2);
        int dyngaindB = dyngain1+dyngain2;

        \\Taylor Series Expansion of the UNDB Function
        float partdyneq1 = a*(dyngaindB - 10);
        float partdyneq2 = partdyneq1*partdyneq1;
        float partdyneq3 = partdyneq2*partdyneq1;
        float partdyneq4 = partdyneq3*partdyneq1;

        dyngain = b+b*partdyneq1+partdyneq2*1.58114+partdyneq3*0.527046+partdyneq4*0.131762;
        dyngain = dyngain-1;


//Mic computation
//michigh = HPFmic(miclow);  //High Pass Filtering was not used in embedded version

        micenv = abs(miclow);
        micenvsqr = fr32_to_float(micenv);

        //run same RMS detection on the mic input but recalibrate to mic sensitivity
        if (micenvsqr >= micprev)
        {
                float micdiff = micenvsqr-micprev;
                miccurrent = micprev+0.5*micenvsqr;
                                }
        else
        {
                float micdiff = micenvsqr - micprev;
                miccurrent = micprev+0.0005*micdiff;

        }

        micprev = miccurrent;

        float rmsmicout = 20*logf(miccurrent);
        int micdB = rmsmicout+140;


        //PreNAUC signal Level Detection (this my be eliminated
        //in favor of not counting Dynamic EQ component in NAUC
        siglow = basslow + lowfreq;  //changed from using just left to stereo
        sighigh = HPFsig(siglow);  //We Could eliminate HPF for computation purposes

        sigenv = abs(siglow);
        sigenvsqr = fr32_to_float(sigenv);

        if (sigenvsqr >= sigprev)
        {

                sigcurrent = sigenvsqr;
        }
        else
        {
                float sigdiff = sigenvsqr - sigprev;
                sigcurrent = sigprev+0.0005*sigdiff;
        }

        sigprev = sigcurrent;

        float rmssigout = 20*logf(sigcurrent);
        int sigdB = rmssigout+110;

        //Could Use either sigdB or inputdB and eliminate
        //Extra filtering
        micgain = NAUC(inputdB/*sigdB*/, micdB);

        //Taylor Expansion of undB20
        float part1 = a*(micgain - 10);
        float part2 = part1*part1;
        float part3 = part2*part1;
        float part4 = part3*part1;

        NAUCgain = b+b*part1+part2*1.58114+part3*0.527046+part4*0.131762;

    //End of dowlsampled computation
    }



    //Anti-Zippering
    int dyngaintemp = dyngain*100;
    dyngaintemp = LPFzip(dyngaintemp);
    dyngain = dyngaintemp*0.01;

    left = inpL >> 8;
    //Dynamic EQ Filtering  Details of Filter are omitted for IP reasons
    left = DynEQBiquad(left);

    left = dyngain*left;
    basslow = LPFbass(left);


    naucgaintemp = NAUCgain*100;
    naucgaintemp = LPFzipmic(naucgaintemp);
    appliedgain = 0.01*naucgaintemp;

    int NAUCsigL = appliedgain*iChannel0LeftIn;
    int NAUCsigR = appliedgain*iChannel0RightIn;

    int sigL = left+NAUCsigL;
    int sigR = left+NAUCsigR;

    iChannel0LeftOut = Dewey(sigL);
    iChannel0RightOut = DeweyR(sigR);
    iChannel1LeftOut = iChannel1LeftIn;
    iChannel1RightOut = iChannel1RightIn;

}
```

Figure B-2: Anti-Aliasing Filter

```
float lpffltx1del2, lpffltx1del1, lpffltx1inp, lpfflty1del2, lpfflty1del1, lpfflty1cur;
float lpfflty2del2, lpfflty2del1, lpfflty2cur;
float lpfflty3del2, lpfflty3del1, lpfflty3cur;
float lpfflty4del2, lpfflty4del1, lpfflty4cur;


int LPFfloat(input)
        {



        lpffltx1del2 = lpffltx1del1;
        lpffltx1del1 = lpffltx1inp;
        lpffltx1inp = input*0.02828;//L_mult(extract_h(input), Gaindew);
        lpfflty1del2 = lpfflty1del1;
        lpfflty1del1 = lpfflty1cur;
        lpfflty1cur = 0.64358*lpffltx1inp-1.26884*lpffltx1del1+0.64358*lpffltx1del2+1.95535*lpfflty1del1-0.96781*lpfflty1del2;


        lpfflty2del2 = lpfflty2del1;
        lpfflty2del1 = lpfflty2cur;
        lpfflty2cur = 0.57136*lpfflty1cur-1.12014*lpfflty1del1+0.57136*lpfflty1del2+1.88525*lpfflty2del1-0.89898*lpfflty2del2;

        lpfflty3del2 = lpfflty3del1;
        lpfflty3del1 = lpfflty3cur;
        lpfflty3cur = 0.27905*lpfflty2cur-0.53371*lpfflty2del1+0.27905*lpfflty2del2+1.80239*lpfflty3del1-0.81885*lpfflty3del2;

        lpfflty4del2 = lpfflty4del1;
        lpfflty4del1 = lpfflty4cur;
        lpfflty4cur = 0.30942*lpfflty3cur-0.42898*lpfflty3del1+0.30942*lpfflty3del2+1.73079*lpfflty4del1-0.75004*lpfflty4del2;
        int lpffltout = lpfflty4cur;

        return (lpffltout);
        }
```

Figure B-3: HPF Filter

```
#include <fract.h>

fract32 hpfsigx1, hpfsigx1del1, hpfsigx1del2;
fract32 hpfsigy1, hpfsigy1del1, hpfsigy1del2;

fract32 hpfsiga10 = 0x0e7d;
fract32 hpfsiga11 = 0xe360;
fract32 hpfsiga12 = 0x0e7d;
fract32 hpfsigb11 = 0x725a;
fract32 hpfsigb12 = 0xc80f;

fract32 hpfsigy2, hpfsigy2del1, hpfsigy2del2;

fract32 hpfsiga20 = 0x29e3;
fract32 hpfsiga21 = 0xacc7;
fract32 hpfsiga22 = 0x29e3;
fract32 hpfsigb21 = 0x33d3;
fract32 hpfsigb22 = 0xe9f3;

fract32 hpfsigy3, hpfsigy3del1, hpfsigy3del2;

fract32 hpfsiga30 = 0x2060;
fract32 hpfsiga31 = 0xbf4d;
fract32 hpfsiga32 = 0x2060;
fract32 hpfsigb31 = 0x18a5;
fract32 hpfsigb32 = 0xf672;

int HPFsig(input)
{

        //first section requires a *2 inloop on the output
        hpfsigx1del2 = hpfsigx1del1;
        hpfsigx1del1 = hpfsigx1;
        hpfsigx1 = input;
        hpfsigy1del2 = hpfsigy1del2;
        hpfsigy1del1 = hpfsigy1;
        hpfsigy1 = L_mult(extract_h(hpfsigx1), hpfsiga10);
        hpfsigy1 = L_add(hpfsigy1, L_mult(extract_h(hpfsigx1del1), hpfsiga11));
        hpfsigy1 = L_add(hpfsigy1, L_mult(extract_h(hpfsigx1del2), hpfsiga12));
        hpfsigy1 = L_add(hpfsigy1, L_mult(extract_h(hpfsigy1del1), hpfsigb11));
        hpfsigy1 = L_add(hpfsigy1, L_mult(extract_h(hpfsigy1del2), hpfsigb12));
        hpfsigy1 = hpfsigy1 << 1;

        //second section requires a *4 inloop on the output
        //hpfsigx2 = hpfsigy1...
        hpfsigy2del2 = hpfsigy2del1;
        hpfsigy2del1 = hpfsigy2;
        hpfsigy2 = L_mult(extract_h(hpfsigy1), hpfsiga20);
        hpfsigy2 = L_add(hpfsigy2, L_mult(extract_h(hpfsigy1del1), hpfsiga21));
        hpfsigy2 = L_add(hpfsigy2, L_mult(extract_h(hpfsigy1del2), hpfsiga22));
        hpfsigy2 = L_add(hpfsigy2, L_mult(extract_h(hpfsigy2del1), hpfsigb21));
        hpfsigy2 = L_add(hpfsigy2, L_mult(extract_h(hpfsigy2del2), hpfsigb22));
        hpfsigy2 = hpfsigy2 << 2;

        //third section requires a *8 inloop on the output
        //hpfsigx3 = hpfsig y2 ...
        hpfsigy3del2 = hpfsigy3del1;
        hpfsigy3del1 = hpfsigy3del2;
        hpfsigy3 = L_mult(extract_h(hpfsigy2), hpfsiga30);
        hpfsigy3 = L_add(hpfsigy3, L_mult(extract_h(hpfsigy2del1), hpfsiga31));
        hpfsigy3 = L_add(hpfsigy3, L_mult(extract_h(hpfsigy2del2), hpfsiga32));
        hpfsigy3 = L_add(hpfsigy3, L_mult(extract_h(hpfsigy3del1), hpfsigb31));
        hpfsigy3 = L_add(hpfsigy3, L_mult(extract_h(hpfsigy3del2), hpfsigb32));
        hpfsigy3 = hpfsigy3 <<3;
        int hpfsigout = hpfsigy3;
        return(hpfsigout);

}
```

Figure B-4: Anti-Zipper Filter

```
float lpfzipmicfltx1del2, lpfzipmicfltx1del1, lpfzipmicfltx1inp;
float lpfzipmicflty1del2, lpfzipmicflty1del1, lpfzipmicflty1cur;

//1st order LPF with Fc=10Hz
int LPFzipmic(input)
        {

        //lpfzipmicfltx1del2 = lpfzipmicfltx1del1;
        lpfzipmicfltx1del1 = lpfzipmicfltx1inp;
        lpfzipmicfltx1inp = input*0.051133;//L_mult(extract_h(input), Gaindew);
        //lpfzipmicflty1del2 = lpfzipmicflty1del1;
        lpfzipmicflty1del1 = lpfzipmicflty1cur;
        lpfzipmicflty1cur = 0.012792*lpfzipmicfltx1inp+0.012792*lpfzipmicfltx1del1+0.99869*lpfzipmicflty1del1;

        int lpfzipmicfltout = lpfzipmicflty1cur;

        return(lpfzipmicfltout);

        }
```

Figure B-5: NAUC

```
#include <math.h>
#include <stdio.h>

int NAUC(levelin, miclevel)
//Note SLH = 1/(0dBintersect-bpsnsr(-3)) = 0.4 in our case

{
        int node1 = 0;
        int gain = 0;
        if(levelin>=30)
                {
                        //Procede as normal
                        node1 = levelin;
                }
        else
                {
                        //Enter Expansion Region
                        node1 = (30-levelin)*miclevel;
                        node1 = node1*0.0667;//L_mult(extract_h(node1), 0x0888);
                        node1 = node1+levelin;
                }

        int actualbpgain = max(min((miclevel-30), 3), 0);
        int SNSR = node1-miclevel;

        if(SNSR<=-3)
                {
                        gain = (-3-SNSR);
                        //printf("gain1:  %d\n", gain);
                        gain = 0.875*gain;
                        //gain = L_mult(gain1, slo);
                        //printf("gain2:  %d\n", gain);
                        gain = gain+actualbpgain;
                        //printf("gain3:  %d\n", gain);
                }
        else
                {
                        gain = (SNSR+3);
                        gain = gain*0.4;
                        //gain = L_mult(extract_h(gain), slh);
                        gain = gain*actualbpgain;
                        gain = actualbpgain-gain;
                }
        int wbgain = max(min(gain, 25), 0);

        //Digital clipping protection
        int distcheck = levelin+wbgain;
        if(distcheck>=60)
                {

                        wbgain = (64-distcheck)*wbgain;
                        wbgain = wbgain >> 2;

                }
        //redundancy for safety after clip protection
        wbgain = max(min(gain, 25), 0);
        return(wbgain);
}
```

Figure B-6: Dewey Filter

```
float x1del2, x1del1, x1inp, y1del2, y1del1, y1cur;
float y2del1, y2del2, y2cur;
float y3del1, y3del2, y3cur;
float y4del1, y4del2, y4cur;

int Dewey(input)
{
        //Note Coefficients of the 4th SOS have been obscured for IP reasons
        x1del2 = x1del1;
        x1del1 = x1inp;
        x1inp = input >> 2;
        y1del2 = y1del1;
        y1del1 = y1cur;
        y1cur = x1inp-2*x1del1+x1del2+1.997485*y1del1-0.997494*y1del2;


        y2del2 = y2del1;
        y2del1 = y2cur;
        y2cur = y1cur-1.39615*y1del1+0.93461*y1del2+1.30045*y2del1-0.80200*y2del2;

        y3del2 = y3del1;
        y3del1 = y3cur;
        y3cur = y2cur-1.87101*y2del1+0.887356*y2del2+0.99962*y3del1-0.13778*y3del2;

        y4del2 = y4del1;
        y4del1 = y4cur;
        y4cur = b0*y3cur+b1*y3del1+b2*y3del2+a1*y4del1+a2*y4del2;
        int dewoutput = y4cur;

return (dewoutput);

}
```

Bibliography

[1] Analog Devices, Inc., One Technology Way, Norwood, Mass. 02062-9106. *ADSP-BF533 Blackfin$^{®}$ Processor Hardware Reference*, 3.0 edition, September- 2004. This is the full Hardware Reference Manual.

[2] Analog Devices, Inc., One Technology Way, Norwood, Mass. 02062-9106. *ADSP-BF533 EZ-Kit Lite^{TM}Evaluation System Manual*, 1.3 edition, August- 2004. This is the Evaluation Kit Instruction Manual.

[3] Joe George. Bose nrg, dsp options. This was an outline of various TI DSPs, December, 2004.

[4] http://www.analog.com. Analog devices website.

[5] http://www.bose.com. Quiet comfort$^{®}$ 2 acoustic noise cancelling$^{®}$ headphones website.

[6] http://www.webervst.com/fm.htm. Equal loudness curves.

[7] Giuseppe Olivadoti. Bf533 vs tic55x.ppt. This was a comparison the two DSPs, December, 2004.

[8] Alan V. Oppenheim & Ronald W. Schafer. *Discrete-Time Signal Processing*, section 6.3, pages 354–363. Prentice Hall Signal Processing. Prentice-Hall Inc, Upper Saddle River, New Jersey 07458, second edition, 1999. This section is on IIR Filter design.

[9] M. Guirao. & S.S. Stevens. Loudness functions and inhibition. *Perception and Psychophysics*, 2:459–465, 1967.