

**EQUALIZER DESIGN FOR SHAPING THE
FREQUENCY CHARACTERISTICS OF
DIGITAL VOICE SIGNALS IN IP TELEPHONY**

By:

Manpreet Kaur Gakhal

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF APPLIED SCIENCE
IN THE SCHOOL OF
ENGINEERING SCIENCE

© Manpreet Kaur Gakhal 2003
SIMON FRASER UNIVERSITY
February 3, 2004

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

Approval

Name: Manpreet Kaur Gakhal
Degree: Bachelor of Applied Science
Title of Thesis: Equalizer Design for Shaping the Frequency Characteristics of Digital Voice Signals in IP Telephony

Dr. Mehrdad Saif
Director
School of Engineering Science, SFU

Examining Committee:

Chair and Academic Supervisor:

Dr. Ljiljana Trajkovic
Associate Professor
School of Engineering Science, SFU

Technical Supervisor:

Andy Fung
Manager, IP Phone Software Development
Broadcom Corporation

Committee Member:

Taro Umezawa
Software Design Engineer
Broadcom Corporation

Date Approved: _____

Abstract

This thesis project describes the development of an equalizer filter used to shape the frequency characteristics of digital voice signals transmitted using an IP phone. The equalizer filter is a software solution allowing IP phone users to meet industry standards defining the frequency response requirements for telephone audio devices.

The equalizer filter was designed as a minimum phase FIR filter. It was implemented as a programmable software module. We developed a design method that, given the desired frequency response and filter order, generates filter coefficients to approximate the desired frequency response. The generic nature of the design process allows the equalizer filter to be adapted for various user requirements. By designing the software for maximum efficiency in terms of processor and memory requirements, we were able to meet the software constraints in addition to meeting the functional requirements of the equalizer filter.

Acknowledgments

I would like to thank my academic supervisor, Dr. Ljiljana Trajkovic, for her guidance and feedback on my work.

I would also like to thank Taro Umezawa, Andy Fung, and the members of the Broadcom PhonexChange software team for their support and technical advice.

Table of Contents

Approval.....	ii
Abstract	iii
Acknowledgments.....	iv
List of Figures	vii
List of Tables.....	ix
1 Introduction.....	1
2 Voice over IP Background.....	2
3 IP Phone System Overview	4
3.1 IP Phone System Architecture.....	4
3.2 PhonexChange Software Architecture	5
4 General Filter Design and Implementation Considerations	8
4.1 Types of Digital Filters	8
4.2 Digital Filter Implementation Considerations	10
4.2.1 Finite Word-length Effects	10
4.2.2 Fixed Point Arithmetic	12
5 Equalizer Filter Specifications and Design.....	14
5.1 Filter Specifications	14
5.2 Design Method	15
5.3 Coefficient Generation	17
5.4 Filter Design Example.....	20
6 Software Implementation	23
6.1 Software Requirements.....	23
6.2 Equalizer Filter Module.....	24
6.2.1 Application Programming Interface (API)	24
6.2.2 C Language Implementation	27

6.2.3	ZSP Assembly Language Implementation	29
6.2.4	Optimizing the Filter Implementation.....	31
6.3	Equalizer PxD Function	33
6.3.1	Application Program Interface.....	33
6.3.2	PxD Function Implementation	35
7	Software Testing and Verification	38
7.1	Test Platform	38
7.2	Unit Testing.....	39
7.2.1	Test Procedure.....	39
7.2.2	Example Unit Test Results.....	39
7.2.3	Example Software Resource Usage.....	46
7.3	System Testing	48
7.3.1	Test Procedure.....	48
7.3.2	Example System Test Results	49
8	Conclusion.....	51
9	References	52

List of Figures

Figure 1 – A Basic VoIP System.	2
Figure 2 – System Architecture of the BCM1101 IP Phone Chip.	5
Figure 3 – The DSP Software Architecture.....	7
Figure 4 – Binary Representation of Q15 Fixed Point Data.	13
Figure 5 – Summary of filter coefficient generation.....	19
Figure 6 – Equalizer filter ideal frequency reponse.	21
Figure 7 – Equalizer filter coefficients.	22
Figure 8 – Comparison of equalizer frequency response for various filter orders.	22
Figure 9 – FIR Direct Form Structure.....	24
Figure 10 – Memory location of the input buffer pointer parameter.	26
Figure 11 – Algorithm for computing the FIR filter samples.....	28
Figure 12 – Performing Dual-MAC Instructions.....	30
Figure 13 – Algorithm to check for accumulator overflow.	31
Figure 14 – Equalizer PxD function implementation.....	37
Figure 15 – Equalizer filter frequency response for the example test cases.....	40
Figure 16 – Filter output for test case with no saturation.....	41
Figure 17 – Filter output for test case with saturation.....	41
Figure 18 – Actual gain for test case with no saturation.....	42
Figure 19 - Actual gain for test case with saturation.....	43
Figure 20 – Error between the actual and expected gains for test case with no saturation.....	44
Figure 21 - Error between the actual and expected gains for test case with filter saturation.	44

Figure 22 – Handsfree receive frequency response mask	46
Figure 23 – System level test setup.....	48
Figure 24 – System response to frequency sweep with equalizer filter disabled.....	50
Figure 25 – System response to frequency sweep with equalizer filter enabled.....	50

List of Tables

Table 1 – Equalizer Filter API Parameters.	25
Table 2 – Equalizer PxD Function API Commands.....	34
Table 3 – Coordinates of handsfree receive response limit curves as specified in TIA/EIA-810-A.....	45
Table 4 – Comparison of processor requirements for different filter order and frame size values.....	47
Table 5 – Example results for optimized branch prediction.	47

1 Introduction

The development of IP telephony products is guided by various telecommunications standards. The developers of such products would like their designs to meet the specifications defined in these standards. One such standard relating to Voice over IP (VoIP) is the Telecommunications Industry Association's TIA/EIA 810-A, *Transmission Requirements for Narrowband Voice over IP and Voice over PCM Digital Wireline Telephones* [1]. This standard establishes handset, headset, and handsfree telephone audio performance requirements, including the frequency response requirements for telephone audio devices.

The primary objective of this thesis project is to develop an equalization filter to shape the frequency characteristics of digital voice signals passing through an IP phone system. The equalization filter enables IP phone users to compensate for the frequency response of the speaker and microphone transducers in order to meet industry standards. Implemented as a generic Finite Impulse Response (FIR) filter software module, the equalization filter parameters may be easily adjusted to achieve the desired frequency response.

The thesis is organized as follows: we begin with a brief introduction to Voice over IP concepts in Section 2, followed by Section 3 which provides an overview of the IP phone software for which we are developing the equalization filter. In Section 4 we discuss general filter design and implementation issues. Sections 5 and 6 provide a detailed discussion of the filter design and implementation. Section 7 concludes with a discussion of the testing and verification process.

2 Voice over IP Background

IP telephony provides the ability to transmit digitized voice over a packet network. Voice over IP (VoIP) is the enabling technology in the convergence of data and telephone networks [3]. Figure 1 illustrates a VoIP network.

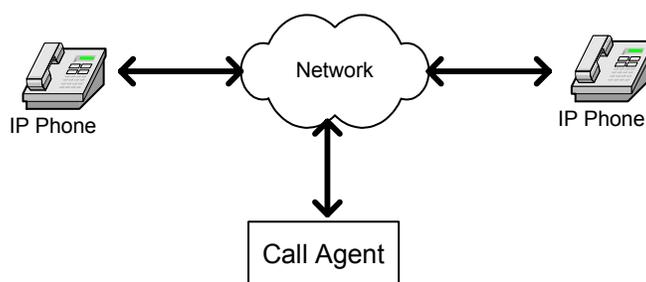


Figure 1 – A Basic VoIP System.

The Public Switched Telephone Network (PSTN) is a circuit switched network that handles traditional voice communications. A circuit switched network requires the bandwidth associated with a voice call to be committed for the entire duration of the connection. Packet switched networks, on the other hand, transmit data in packets. Hence, the resources are not reserved for the entire duration of the call. However problems such as packet latency, jitter, and loss arise when data is transmitted over a network as a sequence of packets.

VoIP relies on data compression as the key to sending voice over a packet switched network. The analog voice signals are converted to a digital format and compressed. The data is then packed into IP packets along with the necessary header information. The Real-time Transport Protocol (RTP) is used to provide end-to-end delivery services such as payload type identification and packet sequencing.

The IP network is a shared network with often unpredictable congestion levels. Hence, packets are not necessarily received at the same rate that they were transmitted. Issues such as packet latency, packet jitter, and packet loss are handled on the receiving end of the voice call. The receiving codec uses various intelligent algorithms to handle latency, jitter, and loss in order to maintain voice quality.

3 IP Phone System Overview

The equalization filter provides users of IP phone software with a means to shape the frequency characteristics of the voice signals. It was developed for use within the IP phone software developed by Broadcom. In this section, we provide an overview of the hardware and software architecture of the IP phone solution employing the equalization filter.

3.1 IP Phone System Architecture

The Broadcom BCM1101 chip provides a single chip IP phone solution. The VoIP software for the BCM1101 is called PhonexChange.

The PhonexChange software is used in two processors: a control processor and a data processor. The control processor is a MIPS Central Processing Unit (CPU). The data processor is an LSI Logic ZSP Digital Signal Processor (DSP). Figure 2 shows the overall system architecture for the two processors within the BCM1101 chip.

The handset, headset, and handsfree devices are external to the IP phone chip and provide the audio interface to the IP phone user. The coder-decoder (codec) converts the voice signals between analog and digital formats. Signals passing to and from the IP network are processed by software elements running on both the MIPS CPU and the ZSP DSP processors.

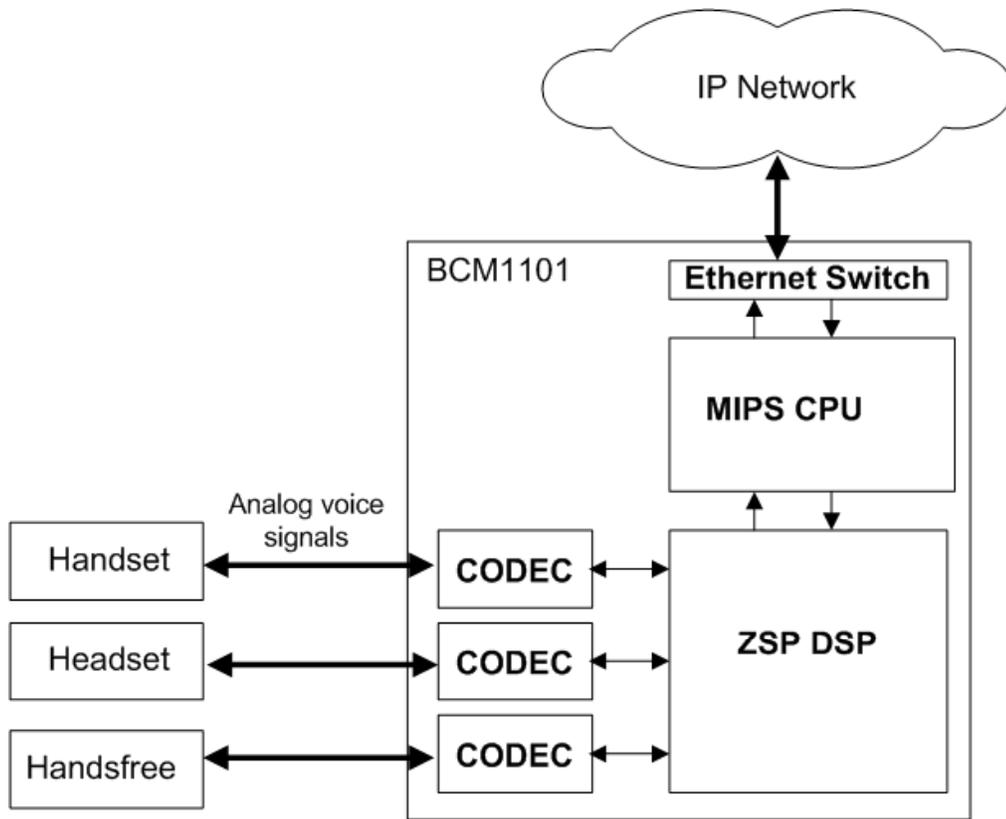


Figure 2 – System Architecture of the BCM1101 IP Phone Chip.

3.2 PhonexChange Software Architecture

The microcontroller software residing in the MIPS CPU is called the Host or MIPS code. The Host software contains a Call Client module that sends and receives commands from other software modules. The Host software also supports H.248/MEGACO, MGCP, H.323, and SIP protocol stacks.

HausWare refers to part of the PhonexChange software. The HausWare software implements VoIP signal processing and supports speech coders such as G.711. The HausWare software runs on both the MIPS CPU and the ZSP DSP.

This thesis deals with the digital signal processing elements within the system.

A summary of the DSP software architecture is shown in Figure 3. The DSP software supports multiple voice data paths or media channels. There is one Virtual HausWare Device Driver (VHD) instance for each media channel. Within a VHD, the voice data is modified by a number of different software services. VHD services, such as packet assembly, are split between the Host and the DSP software. Figure 3 illustrates several services that are performed within a VHD.

The Physical Device Driver (PxD) is a software object for interfacing with external physical devices such as a telephone handset or a speakerphone device. Each PxD can interface with one external physical device. Functions such as echo cancellation (ECAN) and filtering are performed on the signal within the PxDs. The switchboard provides a means of routing media between the virtual and physical device drivers.

As illustrated in Figure 3, voice signals are processed by various functions they pass through a PxD and by various services they pass through a VHD. The term ingress, or tx, refers to the direction of signals passing towards the IP network. The term egress, or rx, refers to the direction of signals received from the IP network.

The equalization filter was developed to operate as a software function within a PxD, as shown in Figure 3. It may be used to shape the frequency characteristics of the ingress and egress voice signals independently. IP phone users are thus provided with a method to compensate for the frequency characteristics of the audio transducers in order to meet industry standards.

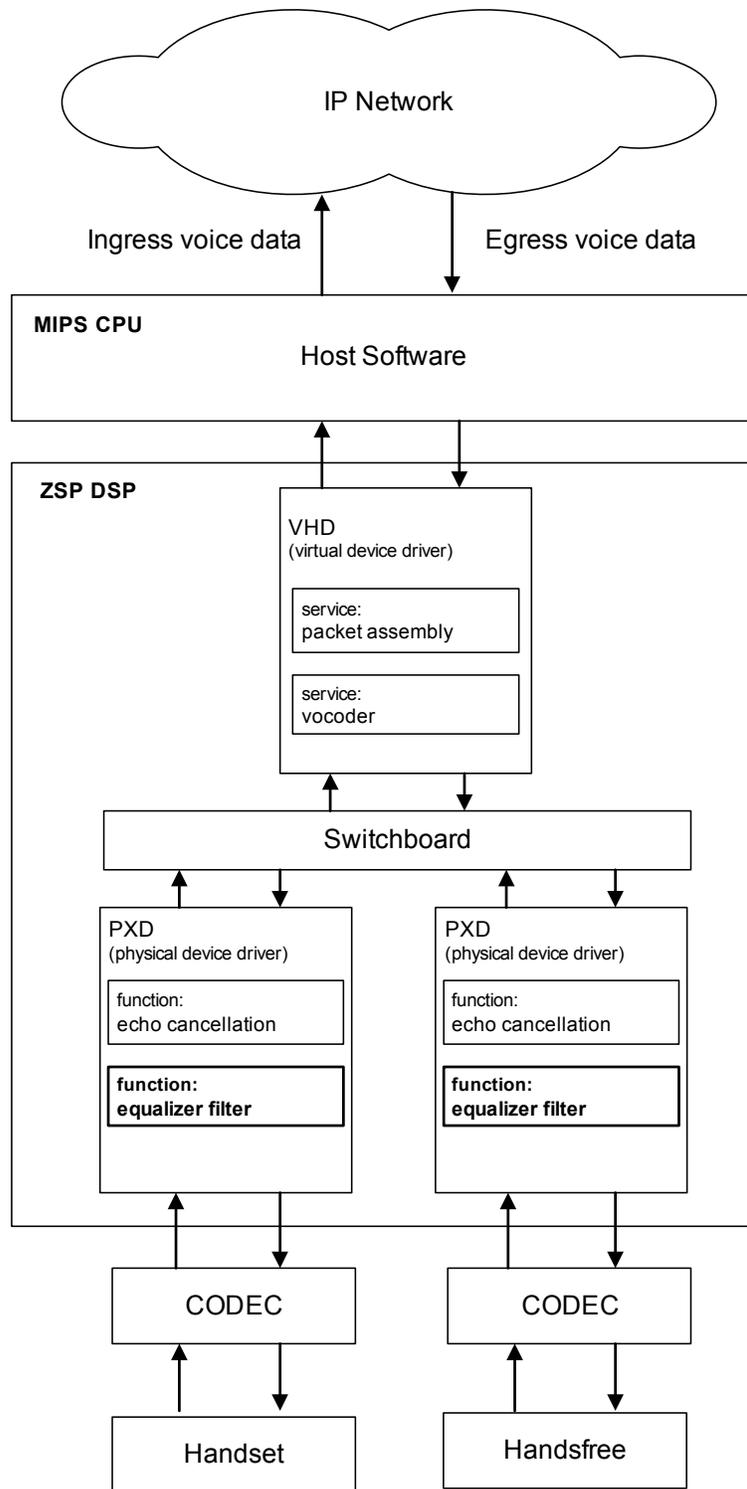


Figure 3 – The DSP Software Architecture.

4 General Filter Design and Implementation Considerations

Filters are widely used in communication systems to selectively modify the characteristics of a signal. Digital filters are mathematical algorithms that operate on digital signals and are implemented in either hardware or software. Digital filters have several advantages over analog filters that make them preferable for use in a variety of signal processing applications, including IP telephony. For example, digital filters tend to have a more predictable, precise, and repeatable behavior than the analog filters [3]. They can be easily programmed and redesigned if implemented using a programmable processor. In this section, we discuss the basic types of digital filters, as well as some issues that need to be considered when implementing digital filters on a DSP.

4.1 Types of Digital Filters

There are two general classes of digital filters: infinite impulse response (IIR) and finite impulse response (FIR) filters. IIR filters are recursive in nature and include the past and present output of the filter in the calculation of coefficients.

Equation (1) represents the relationship between the input $x(n)$, the impulse response $h(n)$, and the output $y(n)$, of an IIR filter. The second part of (1) represents this relationship in the form used for implementation. The symbols a_k and b_k represent the IIR filter coefficients.

$$\begin{aligned}
y(n) &= \sum_{k=0}^{\infty} h(k)x(n-k) = \sum_{k=0}^N a_k x(n-k) - \sum_{k=1}^M b_k y(n-k). \\
y(n) &= \sum_{k=0}^N a_k x(n-k) - \sum_{k=1}^M b_k y(n-k)
\end{aligned} \tag{1}$$

The feedback enables IIR filters allows these types of filters to achieve a better amplitude response than FIR filters for the same computational complexity and memory. IIR filters are ideally used when a sharp frequency response with high throughput is required. Fewer coefficients are required with IIR filters than for FIR filters. FIR filters tend to be more computationally intensive.

FIR filters are generally not recursive; the output depends only on the present and past input samples. Equation (2) represents the FIR filter relationship between the filter input samples $x(n)$, the filter impulse response $h(n)$, and the filter output samples $y(n)$, for a filter of order N .

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k). \tag{2}$$

One of the primary advantages of FIR filters over IIR filters is that FIR filters may have a linear phase response, while IIR filters generally have nonlinear phase responses. The advantage of a linear phase response is that the signal will not suffer from phase distortion introduced by the filter. IIR filters may be unstable, while FIR filters are inherently guaranteed to be stable. Furthermore, finite word effects, such as coefficient quantization errors and round-off noise, are less severe in FIR filters than in IIR filters.

4.2 Digital Filter Implementation Considerations

In practice, digital FIR filter algorithms are usually implemented in software on DSPs. In real-time signal processing, efficient data flow is a prime consideration, and DSPs have specialized architectures tailored to allow high data throughput. For example, DSPs often have complex Harvard architecture memory systems [7]. The Harvard architecture uses separate memory spaces for program and data memory allowing simultaneous access to a program instruction and a data word.

Some DSP instruction sets are specialized to perform common computations within one execution cycle. For example, many DSP algorithms require inner loop calculations of the form $a += b[n]*c[n]$, where a represents an accumulated value, and $b(n)$ and $c(n)$ represent data samples. DSP instruction sets often contain multiply and accumulate (MAC) instructions that perform entire multiply and accumulate operation within one processor cycle. Also, DSPs typically have built in mechanisms to efficiently handle finite word effects such as overflow.

The equalizer filter in this thesis project was implemented on a LSI Logic ZSP DSP. This DSP has a number of architectural features ideal for the implementation of digital signal processing algorithms. However, numbers can only be represented with a finite number of bits on a DSP. Problems associated with finite word lengths are discussed in the following sections.

4.2.1 Finite Word-length Effects

Digital computation has a finite precision, resulting in the approximation of numerical values. The implementation of numerical

algorithms on DSPs suffers from problems associated with these numerical approximations. The use of a finite number of bits to represent numerical data results in quantization, round-off, and overflow errors.

Coefficient quantization errors occur because coefficient values cannot be precisely represented with a limited number of bits. The following equation represents the relationship between the quantized coefficients $h_q(n)$, and the unquantized coefficients $h(n)$:

$$h_q(n) = h(n) + e(n). \quad (3)$$

The function $e(n)$ represents the error between the quantized and unquantized coefficients [3]. The effect of this quantization error is a deviation in the actual filter frequency response from the desired frequency response.

In order to represent numerical values with a finite number of bits the numerical value must be rounded or truncated. This process results in round-off errors. For example, the product of two 16 bit signed numbers is a 32 bit value. If the filter output samples must be represented as 16 bit signed numbers, the 32 bit product will either be rounded or truncated to 16 bits, which will inherently introduce error. In multiply and accumulate loops, round-off errors can be minimized by using additional bits to represent products exactly and then rounding the results after obtaining the final sum. This method incurs less error than rounding each product individually.

If the sum of two numbers exceeds the largest number that may be represented with the given number of bits, overflow occurs. If no

corrective action is taken to handle overflow, significant error is introduced. Overflow is often handled by saturating the output at the maximum allowed positive or negative number. However, the implementation of such overflow handling consumes extra processor cycles. Filters are usually designed to minimize the likelihood of overflow by scaling the coefficient or data values. For example, FIR filter coefficients may be scaled according to the following equation to guarantee that the filter will never saturate:

$$h_{new}(m) = \frac{h(m)}{\sum_{k=0}^{N-1} |h(k)|} . \quad (4)$$

However, scaling the filter coefficients introduces larger coefficient quantization noise [3]. For this reason the filter coefficients are usually not scaled according to equation (4), that guarantees the filter will not saturate under the worst case situation. Instead, the filter coefficients are scaled by a smaller factor depending on the filter application, the expected input signal range, and the filter requirements.

4.2.2 Fixed Point Arithmetic

The LSI Logic ZSP DSP is a 16 bit fixed-point DSP, hence numerical data is represented by 16 bits in fixed-point notation. Fixed-point arithmetic is defined by Q formats, where the number following the Q represents the quantity of fractional bits [4,7]. For example, Q15 is a commonly used fixed-point format. It is illustrated in Figure 4.

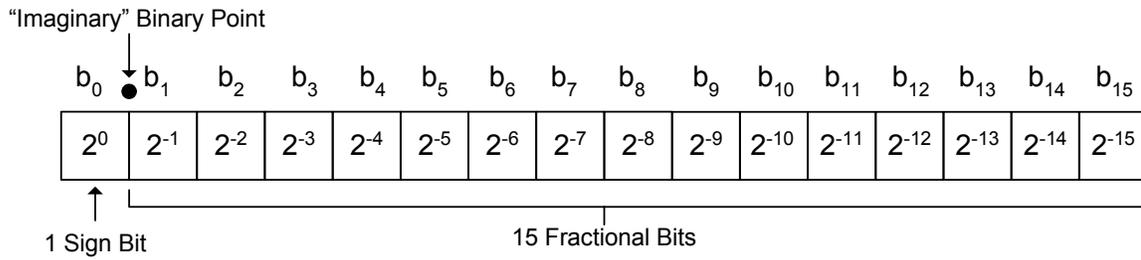


Figure 4 – Binary Representation of Q15 Fixed Point Data.

A Q15 format number is represented with 1 sign bit, 15 fractional bits, and no integer bits. The decimal range of Q15 numbers is between -1 and 0.9999 (a hexadecimal range of 0x8000h to 0x7FFFh). Equation (5) represents the compact form expression for Q15 format numbers, where h is the fractional number and b is the binary value 0 or 1:

$$h = \sum_{n=0}^N b_n 2^{-n} . \tag{5}$$

Addition and subtraction of fixed-point numbers is simple, while multiplication is slightly more complex. The product of two 16-bit Q15 numbers is a 32-bit Q30 number with two sign bits and 30 fractional bits. The second sign bit is redundant so the Q30 product may be shifted by one bit. The result is a Q31 number with 1 sign bit and 31 fractional bits, and the upper 15 bits of the fractional part are taken as the final result.

5 Equalizer Filter Specifications and Design

The purpose of the equalizer filter is to shape the frequency characteristics of the voice signals transmitted through the IP phone system. Transducers in telephone audio devices apply an undesirable frequency response to the voice signals, and the equalizer filter is designed to compensate for this response. In this section, we discuss the filter specifications and design process.

5.1 Filter Specifications

The user of the IP phone software specifies the frequency magnitude response of the equalizer filter. The user needs to measure the transducer frequency response of the telephone device. Based on the measured frequency response, the user needs to determine what frequency response the equalizer filter should have to ensure that the overall frequency response of the voice signals meets the user requirements.

We initially designed the equalizer filter to meet the specific frequency magnitude response provided by the IP phone user for whom the development was being done. However, we also ensured that both the design process and the filter implementation were generic enough so that the equalizer filter could be used again to meet different frequency requirements with minimal additional effort.

When processing real-time voice signals, we would like to minimize the filter delay and phase distortion. A minimum phase filter minimizes the filter group delay and a linear phase filter prevents phase distortion [5]. Our initial filter specifications did not require a linear phase

response. Hence, we designed a minimum phase filter in order to minimize the overall filter delay. In addition, minimum phase filters do not have the coefficient symmetry constraints that linear phase filters do, allowing for greater design flexibility.

The filter design process was influenced by several software constraints. The primary software constraints included the memory requirements and processor consumption of the equalizer filter. These considerations are especially important in large software systems where resources can be limited and should be used with maximum efficiency. For example, a higher order filter will achieve a closer approximation to the desired frequency magnitude response, but will require more memory space to store the coefficients and more processor cycles to compute a single output sample, than a lower order filter. There is a tradeoff between achieving a frequency response closely matching the original filter specifications and using minimal processor and memory resources. We took this tradeoff into consideration during the design process.

5.2 Design Method

Given the filter specifications, we chose to implement the equalizer filter as a minimum phase FIR filter. As discussed in Section 4, although FIR filters are more computationally intensive than IIR filters, they are guaranteed to be stable and are particularly simple to implement on a DSP.

We designed the equalizer filter to be a minimum phase filter. This implies that the filter magnitude response and phase response uniquely specify each other to within a scale factor. Minimum phase systems are

causal and stable and must have all their poles and zeros within the unit circle on the complex plane [5].

We used the window method to calculate the FIR coefficients. This method employs the Fourier transform relationship between the impulse response $h_d(n)$, and the frequency response of the filter $H_d(\omega)$ [3].

$$h_d(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} H_d(\omega) e^{j\omega n} d\omega. \quad (6)$$

Given the desired filter frequency response, the Fourier transform can be used to compute the time domain impulse response $h_d(n)$. However, this response is an ideal response and includes more coefficients than we can retain for the filter implementation. Therefore, we truncate the impulse response to the desired filter order by multiplying it with a window function. An example of such a window function is the rectangular window, which retains all samples within the width of the window and discards all other samples.

Multiplication of two signals in the time domain corresponds to convolution of the signals in the frequency domain. The multiplication of the filter impulse response with a window function distorts the filter frequency response because of this convolution. Equation (7) represents this relationship between the ideal filter response $H_d(\omega)$, the window function $W(\omega)$, and the actual filter response $H(\omega)$:

$$h(n) = h_d(n)w(n) \Rightarrow H(\omega) = H_d(\omega) * W(\omega). \quad (7)$$

The frequency distortion resulting from windowing depends on the window function. For example, the more coefficients we retain, or equivalently, the higher the filter order, the closer the filter frequency

response is to the ideal response. Therefore, ideally the width of the window function $w(n)$ is as large as possible so that most of the samples of the impulse response are retained, and $H(n)$ approaches $H_d(n)$.

However, the filter order should be kept small in order to minimize the number of computations required to produce each filtered output sample. Retaining more coefficients also requires more memory for their storage. In selecting the filter order, we need to consider the practical implementation constraints in addition to our desire to approximate the ideal frequency response as closely as possible. For the equalizer filter we implemented, the filter order is a programmable parameter to allow for flexibility when designing the equalizer filter according to the user specific requirements.

5.3 Coefficient Generation

From the user specified frequency response, we calculated FIR filter coefficients using Matlab. This Matlab function, *genFIRCoeff*, can easily be converted to an executable using the Matlab compiler with the *mcc* Matlab command. The *genFIRCoeff* function has the following Matlab definition:

```
function [q15ScaledCoeff] =  
    genFIRCoeff( N, inputFile, scaleFactor )
```

The input parameters are the filter order, the name of the file containing the desired frequency response, and a scale factor to be applied to the filter coefficients. The input file should specify the desired frequency magnitude response in dB. The gain values must be specified at evenly spaced frequencies from 0 Hz to the $F_s/2$ Hz, where F_s is the

sampling frequency. The *scaleFactor* parameter allows the user to scale the coefficients by a desired amount. The scaling may be useful to prevent filter saturation. The function returns the N scaled filter coefficients in Q15 format.

We chose to use Q15 format in our generic design because it allows for the greatest resolution using 16 bits. Numerical values can easily be scaled to accommodate the +1 to -1 range restriction of Q15 numbers. However, if a different Q format was determined to be more appropriate for a particular application, then the filter implementation could be easily modified to use a different numerical representation. This may be desirable if filter saturation was a significant problem and one wanted a larger numerical range.

Figure 5 summarizes the *genFIRCoeff* function. The desired frequency mask values are first read in from the specified input file. These frequency values specify the desired filter response over the frequency range of 0 Hz to $F_s/2$ Hz, where F_s is the data sampling frequency. The inverse Fourier transform is computed over a range of 2π . Hence the desired frequency response should be specified over the range of 0 Hz to F_s Hz. Since the discrete time Fourier transform is periodic with period 2π , we reflect, or 'fold', the frequency mask to obtain the required range.

Prior to finding the inverse Fourier transform, we interpolate the frequency mask by an arbitrary factor of 64. This provides a smooth approximation of the specified frequency mask. We also convert the values from the dB scale to the linear scale. The inverse Fourier transform is computed using the Matlab *ifft* function. The resulting time

domain response is truncated to the desired filter order using the Hamming window function [5].

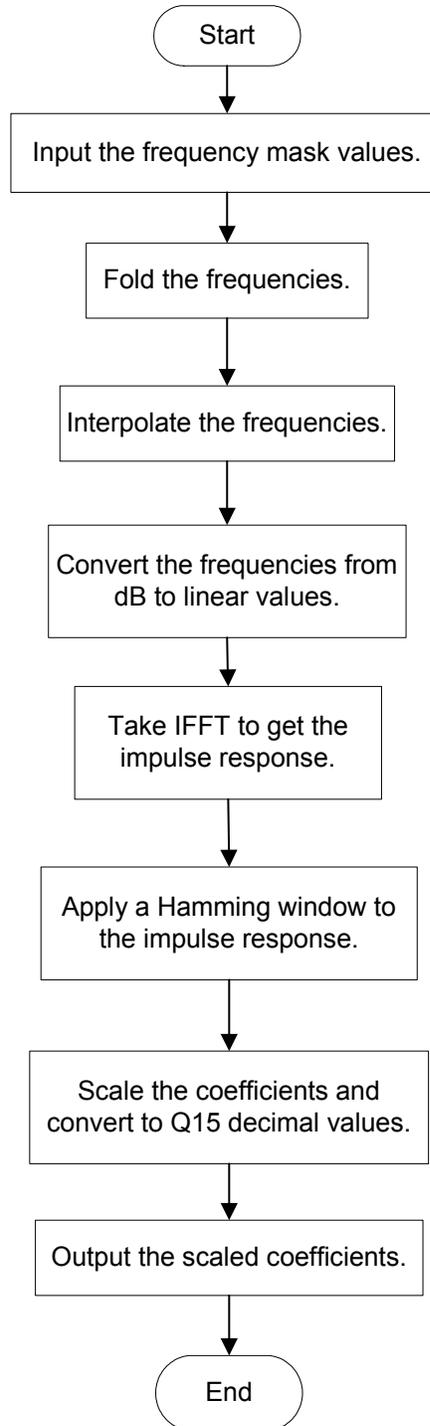


Figure 5 – Summary of filter coefficient generation.

Before scaling the filter coefficients, the Matlab function checks that the filter is a minimum phase filter. This condition will be true if the filter zeros are all within the unit circle. If the filter is not minimum phase, we exit the function exits returning a value of -1.

The filter coefficients are scaled using the user specified scale factor. As discussed in Section 4.2.2, Q15 format numbers have a range of +1 to -1. If any of the scaled coefficients are beyond this range the coefficients need to be scaled again so that the maximum absolute coefficient value is equal to 1. The scaled coefficients are then multiplied by the maximum 16 bit value (32767) and these coefficients are returned to the user.

5.4 Filter Design Example

This thesis project was initially developed to meet the specific frequency requirements of a particular IP phone customer. The customer had defined the Rx and Tx frequency response that the equalizer filter must have in order to meet industry standards. One such specified frequency response is shown in Figure 6.

The frequency mask shown in Figure 6 is applied to the voice signals in the 0 to 4 kHz frequency band. The voice data is sampled at the Nyquist frequency of 8 kHz.

As discussed in Section 5.1, the filter design process is primarily driven by the user requirements. For example, if the requirements indicate that the actual filter response must match the user specified ideal frequency response closely the filter may need to be of a relatively high order. However, if there is more flexibility in approximating the

frequency response the filter order will not need to be as high. The user specifications for this example are shown in Figure 6.

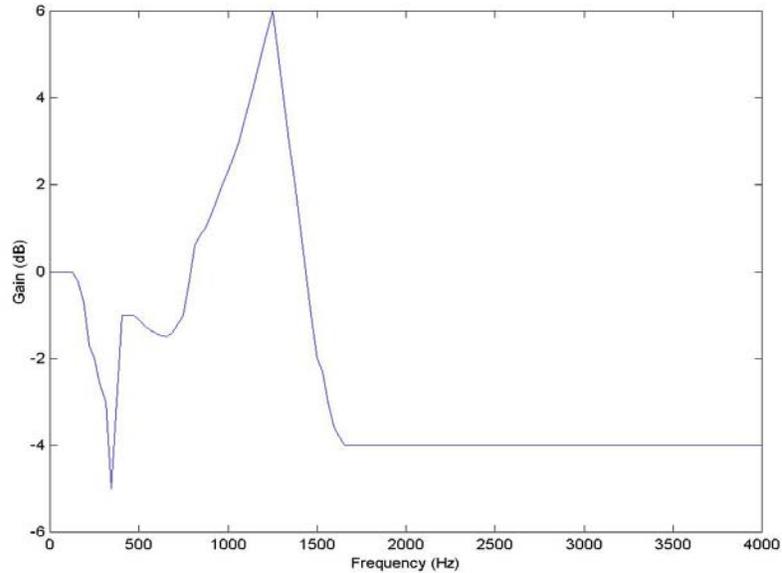


Figure 6 – Equalizer filter ideal frequency response.

We satisfied the user requirements shown in Figure 6 by designing the equalizer filter as a 40 tap FIR filter. A 40 tap filter also allowed us to meet our software constraints on memory and processor consumption. Using the Matlab function described in Section 5.3, we generated the coefficients for the 40 tap FIR equalization filter. The filter coefficients are shown in Figure 7.

Figure 8 shows the frequency response of our 40 tap filter in comparison to the ideal frequency response. Also shown in this figure are the frequency response for an equivalent 20 tap and 80 tap filter. We can see from this figure that a higher filter order allows us to approximate the ideal frequency response more closely, in particular in the 0 to 1 kHz frequency range. Based on user-defined error tolerance,

the approximation generated by the 40 tap filter was determined to be sufficient in this example.

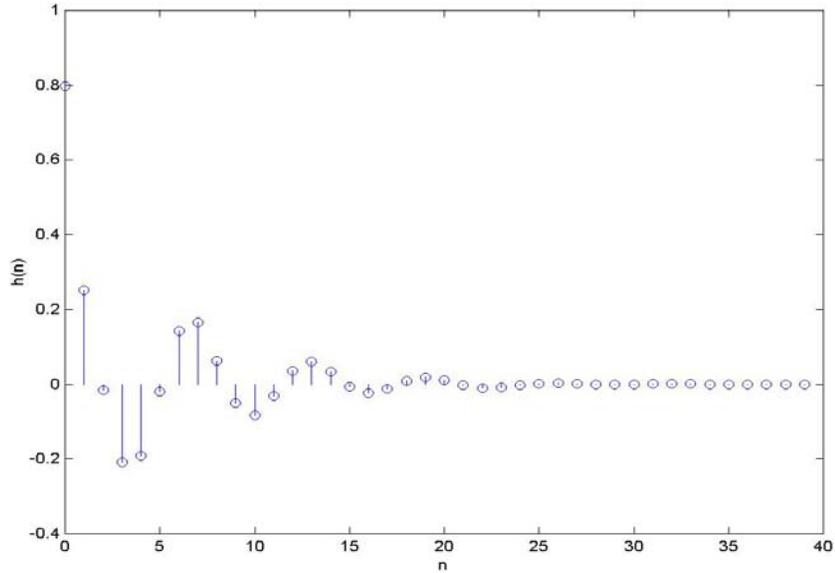


Figure 7 – Equalizer filter coefficients.

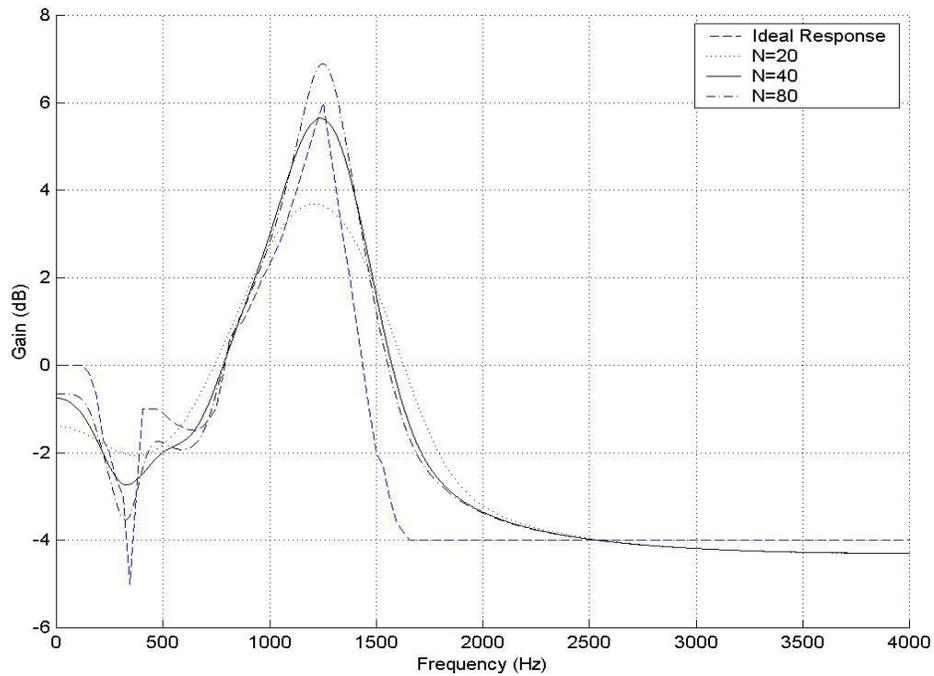


Figure 8 – Comparison of equalizer frequency response for various filter orders.

6 Software Implementation

The equalizer filter was implemented as a software module with programmable coefficients. A separate software module, the equalizer PxD module, acts as a software wrapper for the equalizer filter module. The equalizer PxD module provides an interface for the Host software to control and configure the equalizer filter. In this section, we discuss the software requirements and implementation of the equalizer filter module and the equalizer PxD module.

6.1 Software Requirements

The equalizer filter module had to be generic in order to provide maximum user flexibility and to allow for future code reuse. The generic application programming interface (API) for the equalizer filter module reflects this requirement. The equalizer PxD module operates as a software module within HausWare, and was therefore required to conform to the existing HausWare software architecture.

In general, DSP software needs to be designed so that resources (such as memory space and processor usage) are utilized in the most efficient manner possible. The efficient use of memory space by individual software modules becomes especially important in large software systems, such as the PhonexChange IP phone system. The PhonexChange software is a large, multi-threaded, real-time system. The processor requirements of individual software modules also need to be minimized. It was therefore required that the equalizer filter be optimized for efficient memory and processor usage. In section 6.2.4, we discuss the optimization of the equalizer filter module in further detail.

6.2 Equalizer Filter Module

6.2.1 Application Programming Interface (API)

The equalizer filter is a FIR filter with a programmable filter order and programmable coefficients. An FIR filter implements the relationship given by equation (8), where $x(n)$ represents the input samples, $h(n)$ represents the filter coefficients, $y(n)$ represents the output samples, and N represents the filter order:

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k). \quad (8)$$

The FIR filter was implemented using the direct form structure shown in Figure 9 for an N -tap FIR filter [4]. The filter coefficients are multiplied by the input samples and the N products are accumulated to form one output sample. The filter implementation requires that the $N-1$ previous input samples be retained in memory for computing the next output sample.

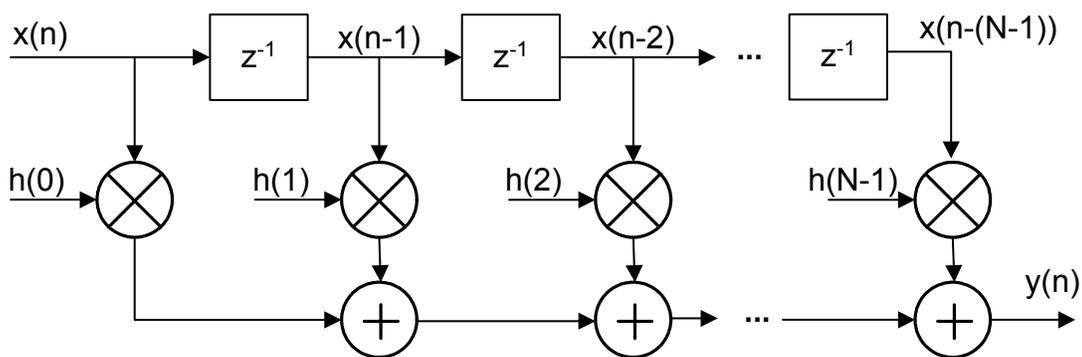


Figure 9 – FIR Direct Form Structure.

The equalizer filter module filters operates on blocks of multiple samples at a time. The number of samples in a block is configurable at

compile time. The block size is dependant on the frame rate at which the software is configured to run at. For example, for 8 kHz sampled data, if the software is configured to process frames every 5 milliseconds, the equalizer filter will process and output 40 samples every 5 milliseconds. This method of processing voice samples at a specific frame rate is more efficient than processing one sample at a time.

The API for the equalizer filter module is as follows:

```
void fir( SINT16 *x,
         SINT16 *y,
         SINT16 *h,
         UINT16 n,
         UINT16 k );
```

Name	Direction	Data Type	Description
*x	Input	Pointer to 16 bit signed data	Pointer to the start of the input sample buffer of size k. The input samples should be preceded by n-1 input history samples.
*y	Output	Pointer to 16 bit signed data	Pointer to the output data buffer of size k.
*h	Input	Pointer to 16 bit signed data	Pointer to the filter coefficient buffer of size n.
n	Input	16 bit unsigned value	Filter order. Should be an even integer.
k	Input	16 bit unsigned value	Data block size.

Table 1 – Equalizer Filter API Parameters.

The input parameters for the equalizer module are given in Table 1. Pointers to the data buffers are passed to the equalizer filter module. The input data pointer x should point to the start of the k new input samples to be filtered, and should be preceded by $n-1$ input history samples, as shown in Figure 10. Having the new input samples contiguous with the history samples reduces the number of processor

cycles required by the equalizer filter for computing one output sample. The equalizer filter module can simply use a pointer to circle through the input samples while performing the multiply and accumulate calculations. Alternatively, the history samples could be passed to the equalizer filter module as a pointer to a separate buffer in memory. However, the filter module would have to allocate a buffer large enough for the history and the new data samples, and then copy the data to this new buffer. Hence, the run-time memory usage and processor consumption of the equalizer filter module has been reduced with the proposed API design.

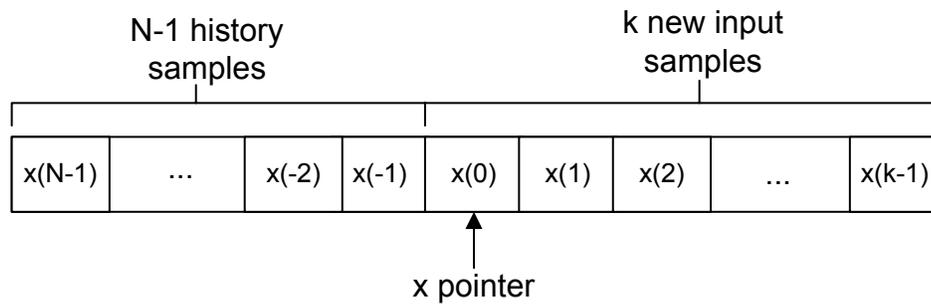


Figure 10 – Memory location of the input buffer pointer parameter.

The output samples are stored in a buffer that has been allocated and passed to the equalizer filter function. The coefficients, the input samples, and the output samples are interpreted as Q15 format numbers. In section 4.2.2, we discussed the Q15 format for representing numerical data.

The implementation of the FIR filter requires that the filter order be even. The ZSP DSP architecture provides a “dual-MAC” instruction that simultaneously computes the results of two multiply and accumulate (MAC) instructions. The use of the dual-MAC instruction significantly reduces the number of processor cycles used by the equalizer filter. The

dual-MAC instruction uses two filter coefficients at a time. Hence, the total number of coefficients must be even. We decided to restrict the filter order to an even number instead of implementing the overhead required to handle the case when the filter order is odd.

6.2.2 C Language Implementation

The equalizer filter module was implemented in the C programming language. The C language model is useful for simulating the behaviour of the filter and porting the equalizer filter to other software platforms. It was necessary for the C language implementation to produce the exact same numeric results as the assembly language implementation for the same input data.

The implementation of the FIR filter module is shown in Figure 11. The C language model simulates the ZSP dual-MAC instruction used in the assembly language implementation by computing two MAC instructions at once. The use of the dual-MAC instruction reduces the number of iterations of the inner loop from N to $N/2$.

The C language model employs the same number of bits to represent the numerical data as the assembly language implementation. The variable used to accumulate the numeric results is defined as a 40 bit signed integer type. However, on some platforms the 40 bit signed integer type is not supported and 64 bits are used instead. This type conversion does not change the filter behaviour because a 40 bit value is assumed to be large enough to hold the accumulated values. The final accumulated value is checked for overflow. If overflow has occurred the value is truncated to the maximum 16 bit positive value (32767) or

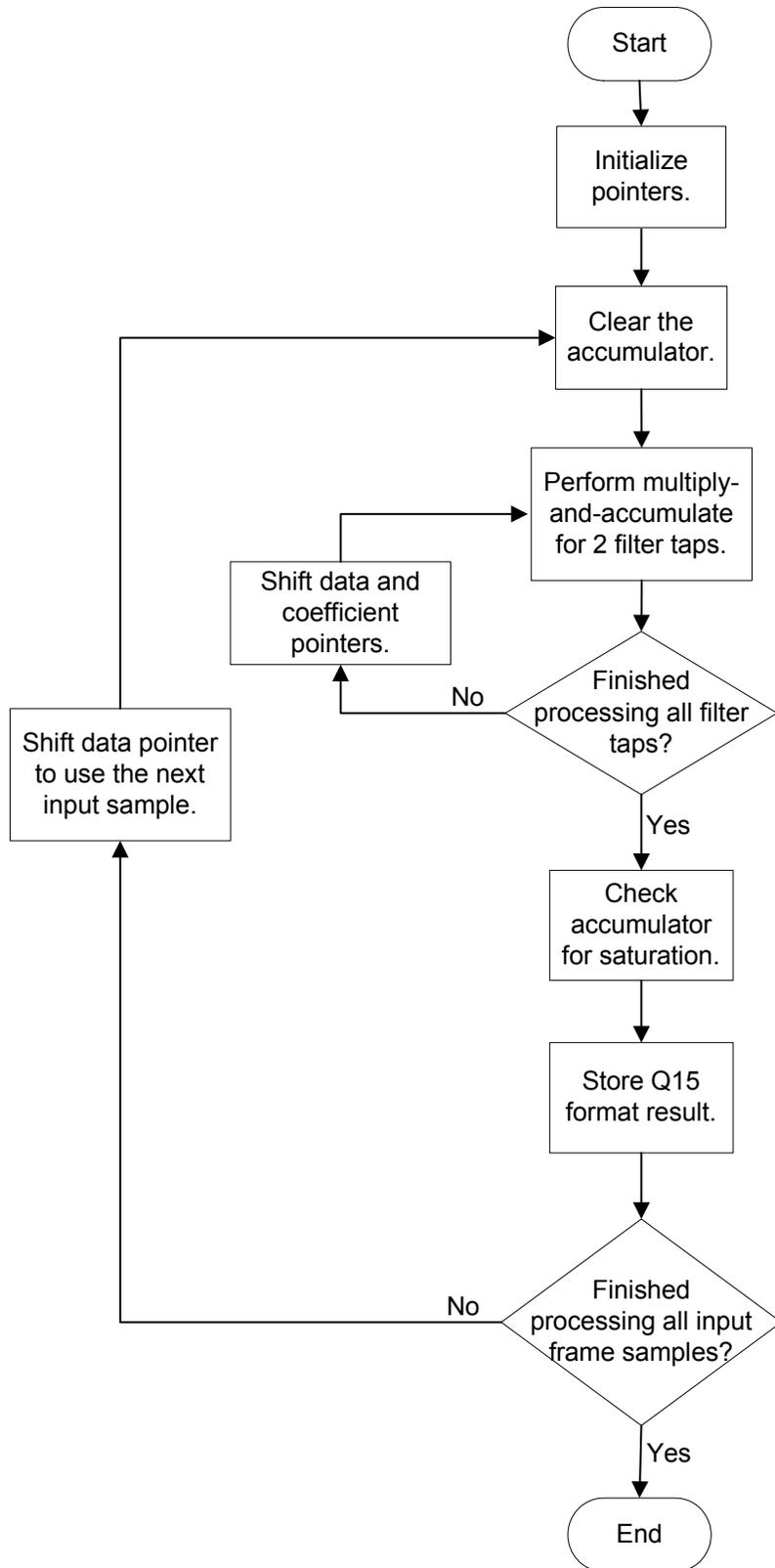


Figure 11 – Algorithm for computing the FIR filter samples.

the maximum 16 bit negative value (-32768) before it is saved in the output buffer.

6.2.3 ZSP Assembly Language Implementation

We also implemented the equalizer filter module using the ZSP assembly language. Although assembly language is more difficult to use because it requires a low level understanding of the hardware, it allows the programmer to optimize code for maximum efficiency. With high level software languages, the programmer must rely on the compiler to translate the C language implementation to the assembly language code executed by the processor. It is therefore advantageous to implement the equalizer filter module directly in the ZSP assembly language.

The LSI Logic ZSP DSP has several hardware architecture features that are specially designed for efficient implementation of common digital signal processing algorithms [4]. For example, there are two multiply and accumulate (MAC) units designed to independently multiply two 16 bit values and form a 40 bit accumulated value within a single processor cycle. The software dual-MAC instruction performs two multiply and accumulates within one processor cycle, which reduces the number of iterations required to compute a single output sample by two. An example of the dual-MAC instruction is shown in Figure 12.

Filter algorithms require sequential access to data buffers. The filter implementation is most efficient if the data is stored in circular buffers. The ZSP DSP has special registers used to implement circular buffers. The use of circular buffers improves the efficiency of the code because the programmer does not have to deal with the overhead pointer arithmetic for circling through the data buffers.

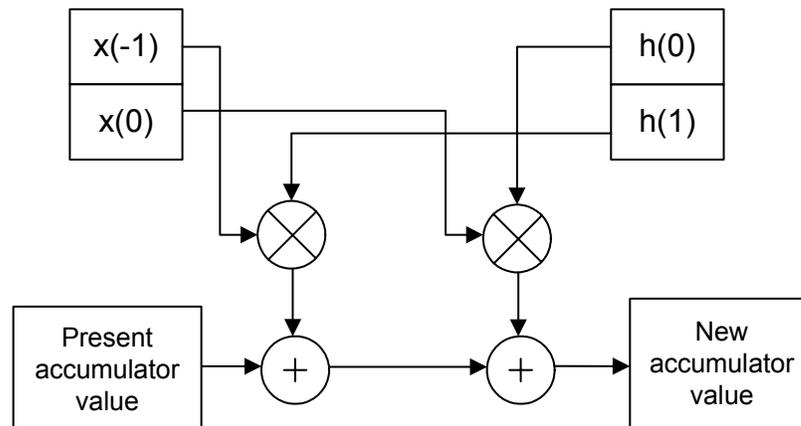


Figure 12 – Performing Dual-MAC Instructions.

The assembly language implementation of the equalizer filter follows the same algorithm as the C language implementation outlined in Figure 11. However, in the assembly language implementation additional steps such as saving and restoring the register contents and setting up the circular buffer pointers are formed at the beginning of the function.

The ZSP DSP has a Q15 mode that can be used to handle Q15 format data more efficiently. When the Q15 mode is enabled, the product from a MAC instruction is automatically shifted left by one bit to remove the extra sign bit before it is summed with the previous result.

The ZSP accumulator is 32 bits, with an additional 8 bit guard register that can be used to extend the accumulator to 40 bits. If the result of a MAC instruction overflows the accumulator does not need to be truncated to a 32 bit number immediately. Instead, the individual products are accumulated using 40 bits, and after all filter taps have been computed the final result is checked for overflow and truncated to a 32 bit number. This method reduces the error resulting from overflow because the guard bits allow overflow to occur in intermediate results.

Prior to storing an output sample, the result is checked for overflow and truncated to a 32 bit value. Based on the value of the guard register and the sign of the 32 bit accumulator, the result may be saturated at the maximum negative or the maximum positive value. The upper 16 bits of the 32 bit accumulator are stored as the final Q15 format output sample. The procedure used to check for overflow is shown in Figure 13.

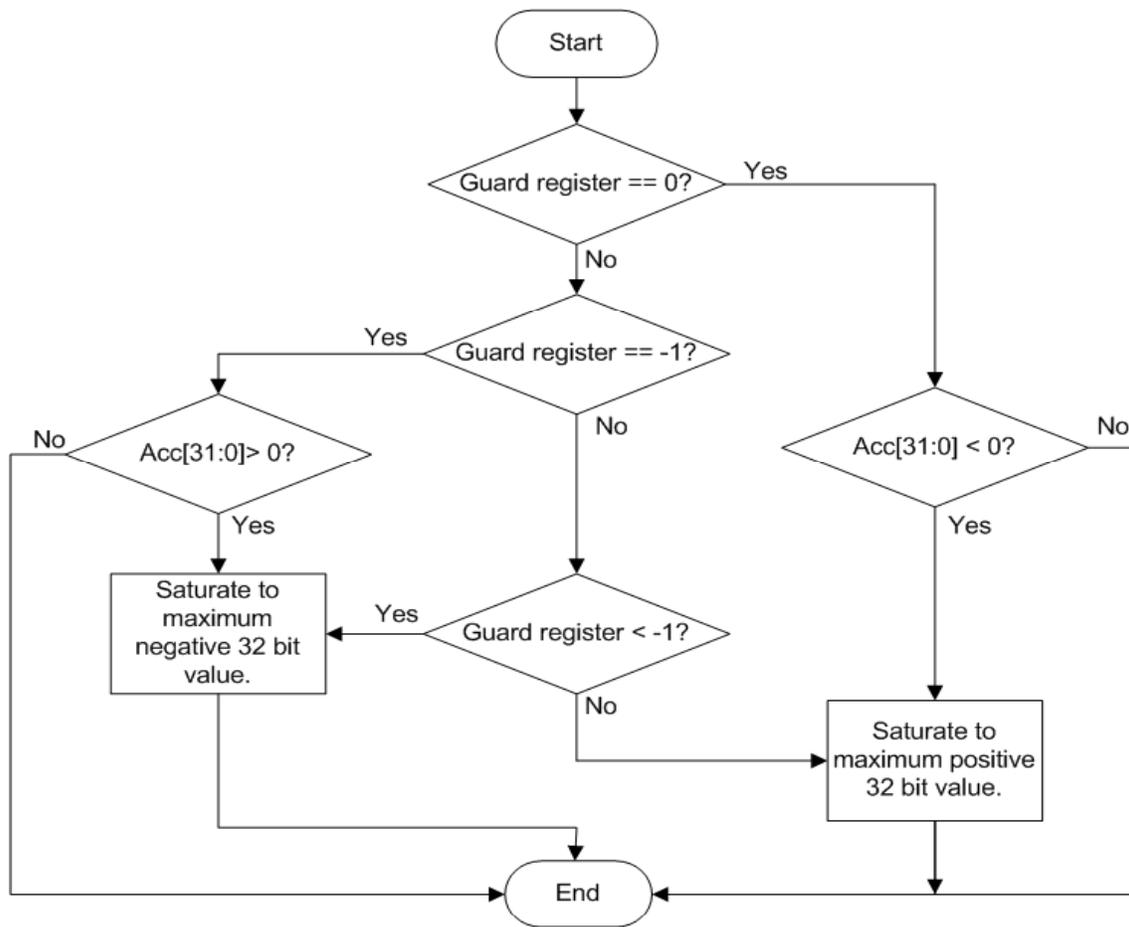


Figure 13 – Algorithm to check for accumulator overflow.

6.2.4 Optimizing the Filter Implementation

The ZSP architecture has specialized features that allow DSP algorithms to be implemented efficiently. The DSP programmer needs to understand how the assembly code can be optimized in order to take full

advantage of the architectural features. For example, the pipeline control unit will attempt to group up to four instructions together for parallel execution. This instruction grouping helps to maximize DSP throughput but is constrained by the processor grouping rules. We optimized the processor throughput of the assembly language implementation of the equalizer filter module by following the ZSP DSP grouping rules.

We also optimized the equalizer filter module by reducing the probability of branch misprediction. For every branch instruction, the processor will predict the branch result and will load the pipeline with the predicted instructions for execution. If the branch prediction is incorrect, the pipeline must be flushed and the correct instructions loaded, which usually requires 5 extra processor cycles. We designed the equalizer filter so that the branch prediction assumes overflow will not occur. Since the filter should be designed to minimize the likelihood of saturation, this implementation will often result in optimal processor usage.

Similarly, there is a processor penalty if the stored data straddles cache lines in memory. The on chip memory is stored in 64 bit cache lines. If the data to be read spans two cache lines, than an extra cycle is required to read the second cache line. For this reason, we implemented the equalizer filter so that all data buffers are evenly aligned in memory (they start on even addresses). This implementation requires fewer processor cycles than if the data buffers were not evenly aligned.

6.3 Equalizer PxD Function

6.3.1 Application Program Interface

The equalizer filter function is a generic FIR filter software module. The PxD equalizer function integrates this filter as a PxD function into the existing PhonexChange system. As discussed in Section 3, a PxD is a software abstraction of a physical audio device, such as a telephone handset. An analog voice signal passing through an external audio device is sampled and converted to a digital signal by a codec. The PxD functions operate on the digital voice signal. The equalizer filter is one of the PxD functions. The PxD functions operate on voice signals passing in both Rx and Tx directions.

The HausWare software architecture provides a standard method of passing information in the form of events and commands to and from the Host software. There are two types of commands: standard and extended. Each type of command has two associated parameters, referred to as op1 and op2, that can be used to pass data parameters. For standard commands, the two parameters are unsigned 16 bit integers. For extended commands the second parameter is a void* pointer, while the first parameter is an unsigned 16 bit integer specifying the number of bytes in the data buffer.

The PxD equalizer function provides an API to the Host software so that the Host can configure and control the operation of the equalizer filter module. There are two separate equalizer filters operating within the same PxD: one filter for ingress (Tx) signals and one filter for egress (Rx) signals. The two filters are controlled individually using separate commands.

The API commands for the equalizer filter allow the Host software to enable or disable the filters, to set the filter order, and to set the filter coefficients. The commands that form the equalizer PxD function API are summarized in Table 2.

Command Name	Op1	Op2	Description
HAPI_FIR_TX_ENABLE_CMD	0 to disable, 1 to enable	Not used	Enables/disables Tx filter.
HAPI_FIR_RX_ENABLE_CMD	0 to disable, 1 to enable	Not used	Enables/disables Rx filter.
HAPI_FIR_SET_TX_ORDER_CMD	Filter order.	Not used	Sets Tx filter order.
HAPI_FIR_SET_RX_ORDER_CMD	Filter order.	Not used	Sets Rx filter order.
HAPI_FIR_SET_TX_COEFF_CMD	Size of coefficient buffer in bytes.	Pointer to coefficient buffer.	Sets Tx filter coefficients.
HAPI_FIR_SET_RX_COEFF_CMD	Size of coefficient buffer in bytes.	Pointer to coefficient buffer.	Sets Rx filter coefficients.

Table 2 – Equalizer PxD Function API Commands.

The commands to enable or disable the filter use a single input parameter. The commands to set the filter order also use a single parameter to specify the filter order. The filter order is restricted to be an integer between 0 and a predefined maximum value. The filter order does not need to be an even integer even though the equalizer filter module API specifies that the filter order must be an even integer. If the order is set to an odd integer, the equalizer PxD function will internally round the filter order up to the next even integer and store this value. An extra filter tap with a value of zero is also stored. In this manner, the

restriction that the equalizer filter module places on the filter order is hidden from the user without changing the filter behaviour.

The commands to set the filter coefficients are extended commands and require the second parameter to be a pointer to a data buffer containing the coefficients. The first parameter is used to specify the length of the data buffer in bytes. The size of the coefficient set should be the same as the filter order.

6.3.2 PxD Function Implementation

The equalizer PxD function has instance memory associated with it. It is used to store the state of the equalizer filter. The filter parameters that an instance of the equalizer PxD function must store are: the enabled state, the order, the coefficients, and the history samples. These parameters must be stored separately for the Rx and the Tx equalizer filters. The instance memory has the following structure:

```
typedef struct FIRPXD_Instance
{
    FIRPXD_Data tx;    /* tx (ingress) data */
    FIRPXD_Data rx;    /* rx (egress) data */
} FIRPXD_Instance;
```

The data structure for each of the Rx and Tx filters has the following structure:

```
typedef struct FIRPXD_Data
{
    SINT16  coeff[ FIRPXD_MAX_ORDER ];
    SINT16  history[ FIRPXD_MAX_HIST ];
    BOOL    enabled;
    UINT16  order;
    UINT16  pad;
} FIRPXD_Data;
```

The constant `FIRPXD_MAX_ORDER` specifies the maximum allowed order of the filter module. The constant `FIRPXD_MAX_HIST` is defined as one less than `FIRPXD_MAX_ORDER`. These constants are defined at compile time.

The `pad` parameter is required to ensure correct memory alignment of the Rx and Tx filter coefficient buffers. There are an odd number of bytes in the `FIRPXD_Data` structure because the filter order N must be an even number and there are N coefficients and $N-1$ history samples. An additional byte is added to the structure so that the number of bytes in the data structure is guaranteed to be even. Hence, within an instance of the `FIRPXD_Instance` structure, if the Tx data structure begins on an even address then the Rx data structure will also begin on an even address. The use of even memory alignment to optimize processor usage when fetching data from memory was discussed in Section 6.2.4

There are separate threads to process the ingress and egress data. However, the equalizer PxD processing function is essentially identical for both Rx and Tx data. There are separate instance memory parameters for the Rx and Tx filters, and the input and output data buffers are different. Nevertheless, the method for processing the Rx and Tx data is the same. Figure 14 summarizes this method for processing the voice data.

The equalizer PxD function copies the filter history samples and the new input samples to one buffer. A pointer to the start of the new input samples within this buffer is passed to the equalizer filter function. This is necessary because the equalizer filter API specifies that the input samples must be immediately preceded in memory by the filter history samples.

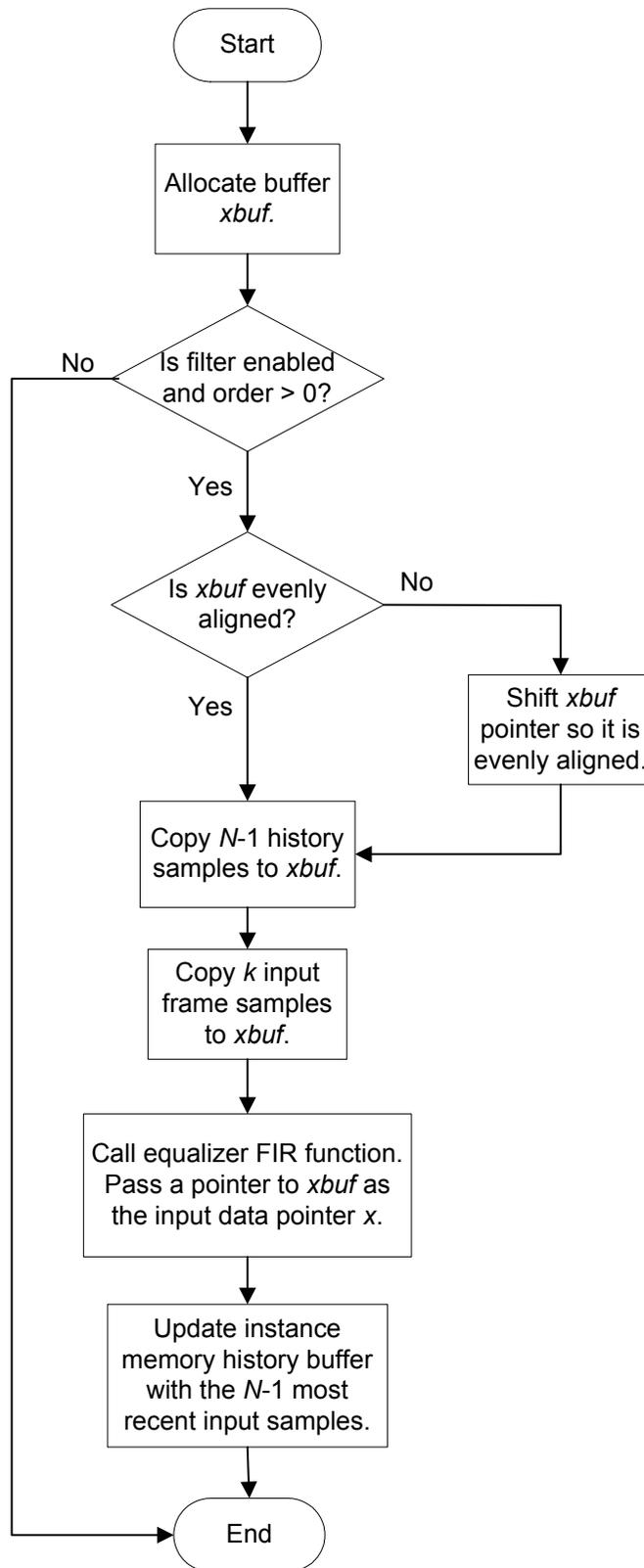


Figure 14 – Equalizer PxD function implementation.

7 Software Testing and Verification

We tested the equalizer filter software by performing unit and system level tests. In this section, we discuss the test procedure we used and some example test results.

7.1 Test Platform

We developed and tested the equalizer filter using the development environment and tools that were already in place for testing the PhonexChange software. The system level testing was performed on the Broadcom IP phone reference platform for the BCM1101 chip.

The unit level testing was done using a test application we wrote specifically for the equalizer filter module. The existing HausWare build environment allows the test application to be built as either a Microsoft Visual C++ executable for a Win32 platform, or as a binary image for the ZSP platform. We tested the C language implementation of the equalizer filter module on the Win32 platform. We tested the assembly language implementation on the ZSP platform.

The LSI Logic ZSP Software Development Kit (ZSP SDK) was used to debug and characterize the equalizer filter module on the ZSP platform. The ZSP SDK allows for the application to be simulated using a simulator on the host PC. The ZSP SDK also allows the application to be tested on the target hardware, connected through a JTAG controller. Features of the ZSP SDK include support for profiling the processor usage of the assembly code and viewing the pipeline information. These features are especially useful for optimizing assembly code.

7.2 Unit Testing

7.2.1 Test Procedure

The purpose of unit testing was to verify and characterize the functional behaviour of the equalizer filter module. The unit test application reads the provided input data, runs the equalizer filter function on the data, and writes the output data to a file. The input and output data is specified as 8 kHz sampled 16 bit linear data. Certain test parameters, such as the filter frame rate, the filter order, and the input file size are defined at compile time.

The test application allows the output data to be compared to a “golden” reference file that has previously been verified by the user to be correct. This comparison feature was especially useful when verifying that the C language implementation and the assembly language implementation of the equalizer filter module were bit exact with respect to each other.

7.2.2 Example Unit Test Results

Figure 15 shows the equalizer filter frequency response for the coefficients used in the two example test cases discussed in this section. The equalizer filter was a 40-tap filter for these examples.

We generated the input PCM data using the Syntrillium Software Corporation’s Cool Edit program. For both example test cases, the input PCM data consists of a frequency sweep ranging from 0 Hz to 4 kHz, with a 50 Hz step size. Each frequency tone is 20 ms long, followed by 10 ms of silence. The silence between each frequency step was added to ensure

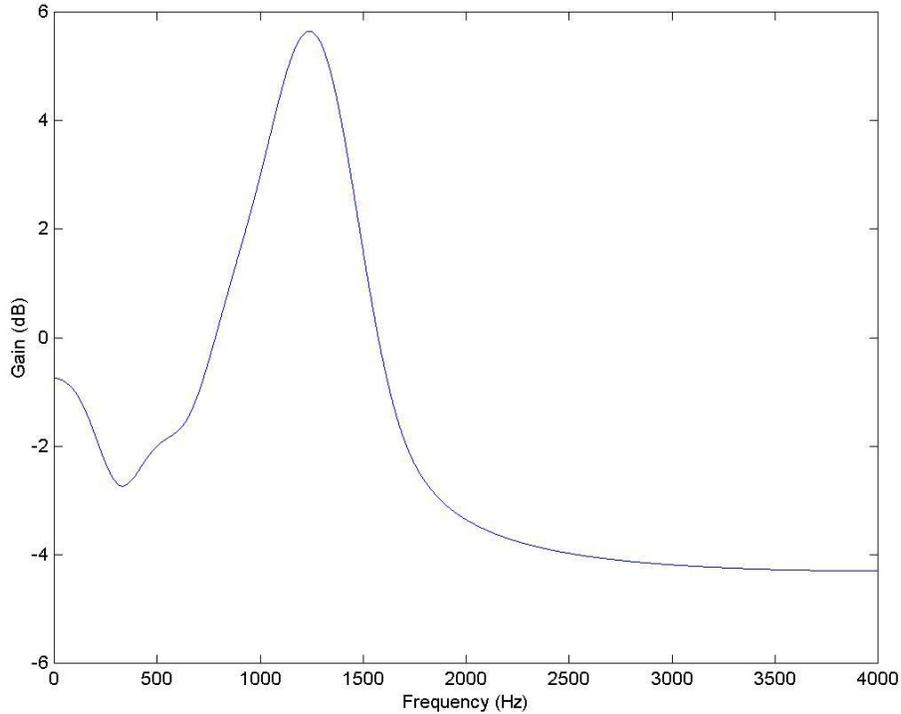


Figure 15 – Equalizer filter frequency response for the example test cases.

the filter had cleared its internal history before processing the next frequency step. The amplitude of the frequency sweep is -12 dB for the first test case, and -1 dB for the second test case.

The filter input signal was a frequency sweep over time. Hence we expect that at each frequency step in time, the appropriate filter gain for that frequency will be applied to the input signal. The envelope of the output signal for each of our test cases should, therefore, reflect the filter frequency response applied to the -12 dB or -1 dB input signal.

Figure 16 and Figure 17 show the filtered output data for each test case. The horizontal axis is in seconds, and the vertical axis is in dB. In the second test case, between approximately 0.5 and 0.9 s, the filter gain is large and the output saturates at the maximum values due to

numerical overflow. The spikes are caused by the transition between the silence and the next frequency step, and hence should be ignored.

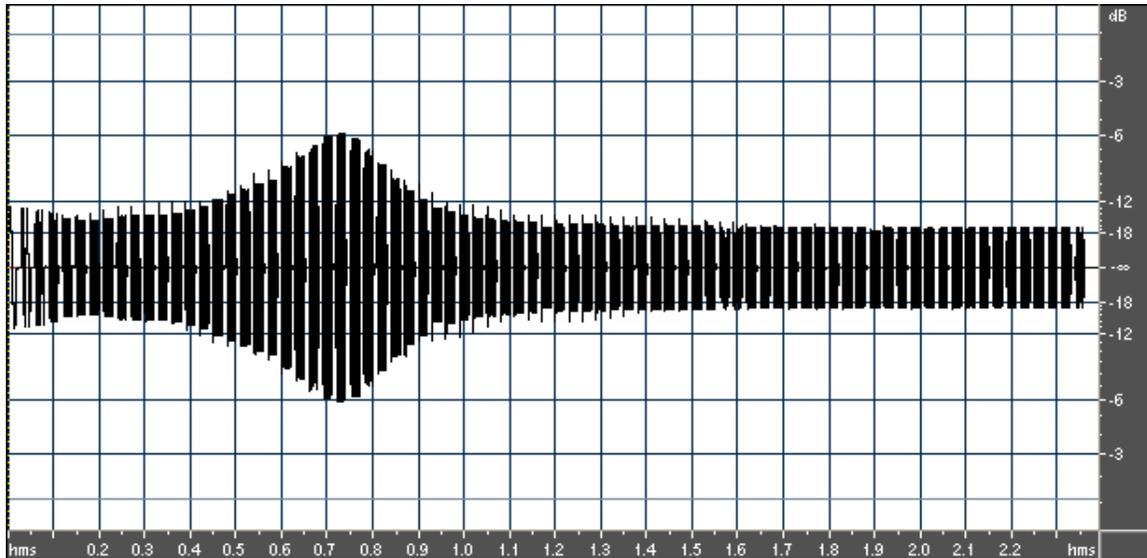


Figure 16 – Filter output for test case with no saturation.

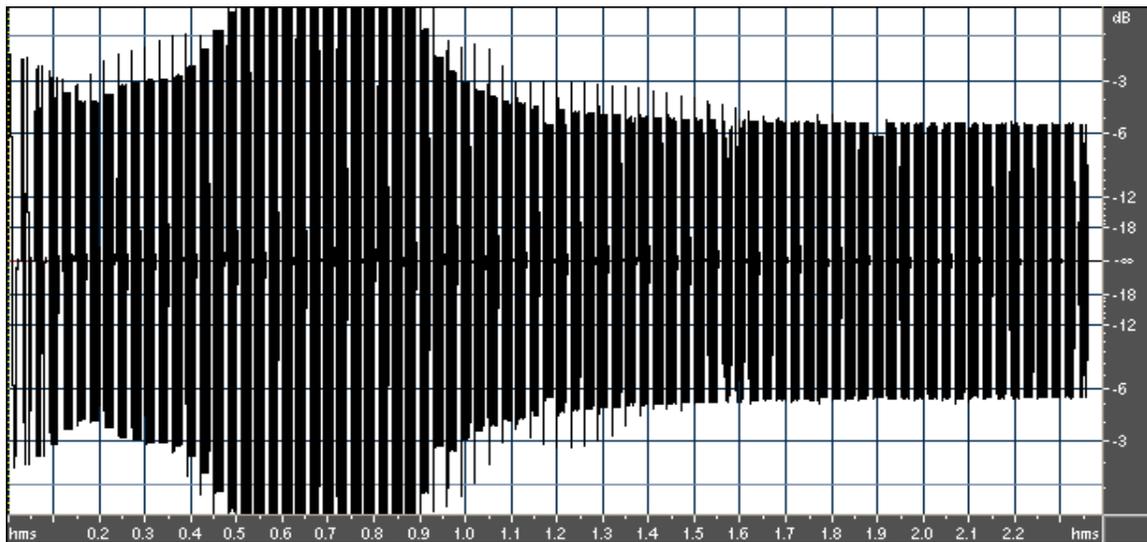


Figure 17 – Filter output for test case with saturation.

We created a separate test application to quantitatively analyze the results for each test case. We compared the actual gain to the expected Matlab generated gain at each step in the input frequency sweep. To compute the actual gain, we calculated the root-mean-square (RMS)

value for a window of samples in each frequency step for both the input and output PCM files. The difference of the dB RMS values is the actual filter gain:

$$rmsValue = \sqrt{\frac{1}{M} \sum_{i=0}^M \left(\frac{sample[i]}{32767} \right)^2} \quad (9)$$

$$actualGain = 20\log(rmsOutput) - 20\log(rmsInput). \quad (10)$$

The actual frequency response for both test cases is plotted in Figure 18 and Figure 19. A qualitative comparison of these figures with Figure 15 confirms that the frequency response of the equalizer filter approximately matches the expected frequency response. In Figure 19 we can see that between 1000 Hz and 1500 Hz the actual filter gain differs considerably from the expected filter gain. This error in the second test case is the result of filter saturation over the 1000 Hz to 1500 Hz frequency range.

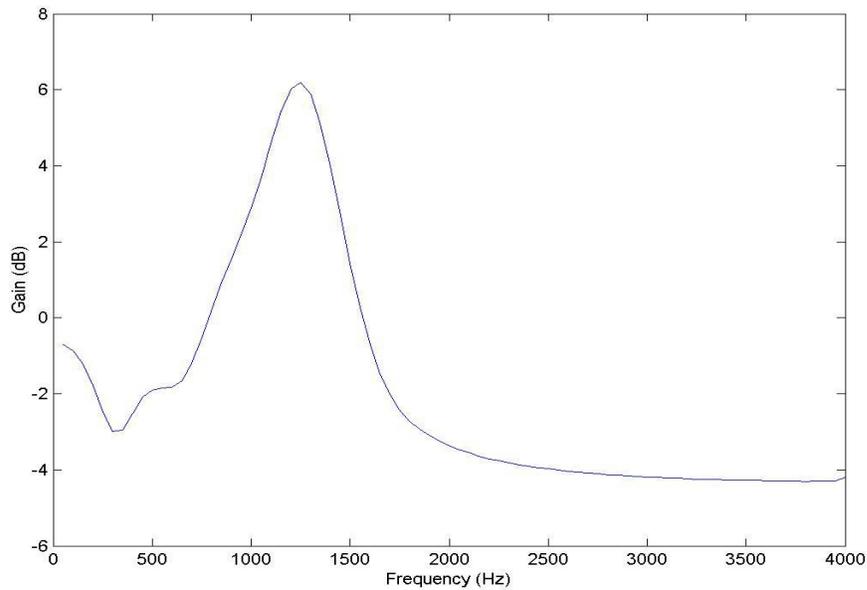


Figure 18 – Actual gain for test case with no saturation

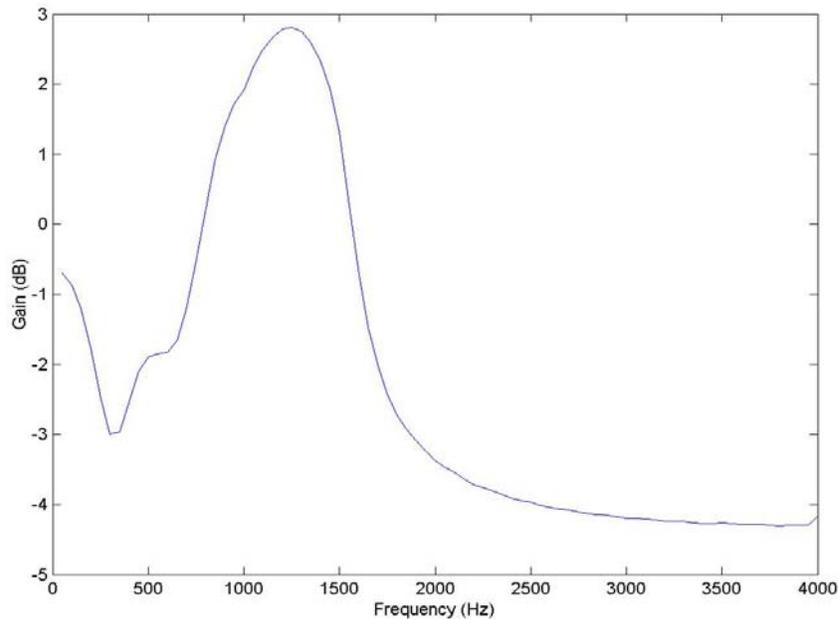


Figure 19 - Actual gain for test case with saturation

The error between the actual filter gain and the expected filter gain is due to numerical imprecision in the filter implementation. Figure 20 and Figure 21 show the error at each frequency step for each test case. In the first case, the maximum numerical error is always less than 1 dB. In the second case, the numerical error is significant in the frequency range where the filter output saturates.

To analyze whether the error in these test cases is acceptable we must consider the specified frequency requirements. As previously discussed, the equalizer filter is used to compensate for frequency distortions introduced by the audio transducers. The IP phone developer must measure this frequency distortion and then determine the compensating frequency response required so the IP phone system meets the relevant standards.

The standards for the frequency response requirements of the handset, headset, and handsfree audio devices are defined in TIA/EIA 810-A. Separate requirements are specified for the transmitted and received signals of each audio device. The frequency response

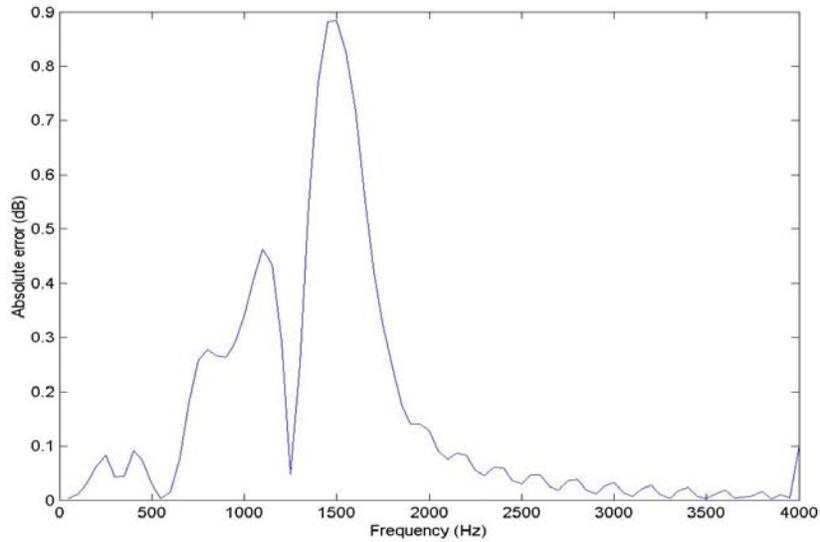


Figure 20 – Error between the actual and expected gains for test case with no saturation.

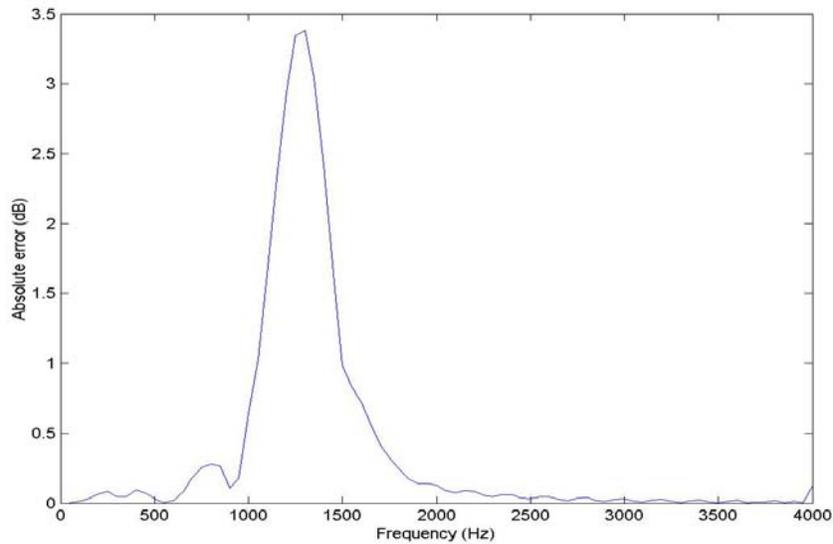


Figure 21 - Error between the actual and expected gains for test case with filter saturation.

requirements are defined in terms of an upper limit frequency response and a lower limit frequency response. The actual frequency response is expected to fall within the upper and lower limits. For example, Table 3 shows the advised frequency specification for IP phone signals received by the handsfree audio device, as defined in TIA/EIA 810-A [1]. The coordinates for both the upper and lower limits are plotted in Figure 22 on a logarithmic frequency scale. Note that frequency response mask is specified as a floating, or ‘best-fit’ mask, so the absolute dB level is arbitrary.

Limit Curve	Frequency Band (Hz)	Receive Response Limit (dB) [arbitrary level]
Upper limit	100	-9
	125	-6
	160	-3
	200 to 4000	0
	5000	-7
	6300	-14
	8000	-20
Lower limit	250	-infinity
	315	-12
	400	-11
	500 to 2500	-10
	3150	-13
	4000	-infinity

Table 3 – Coordinates of handsfree receive response limit curves as specified in TIA/EIA-810-A

As illustrated in Figure 22, the minimum difference between the upper and lower limits is 10dB in the 0 to 4 kHz frequency band. The example frequency response shown in Figure 15 was specified by a particular IP phone developer as the response required for them to meet the specifications in Table 3. The error shown in Figure 20 and Figure

21 is within acceptable limits because of the relatively large tolerance between the upper and lower limit requirements.

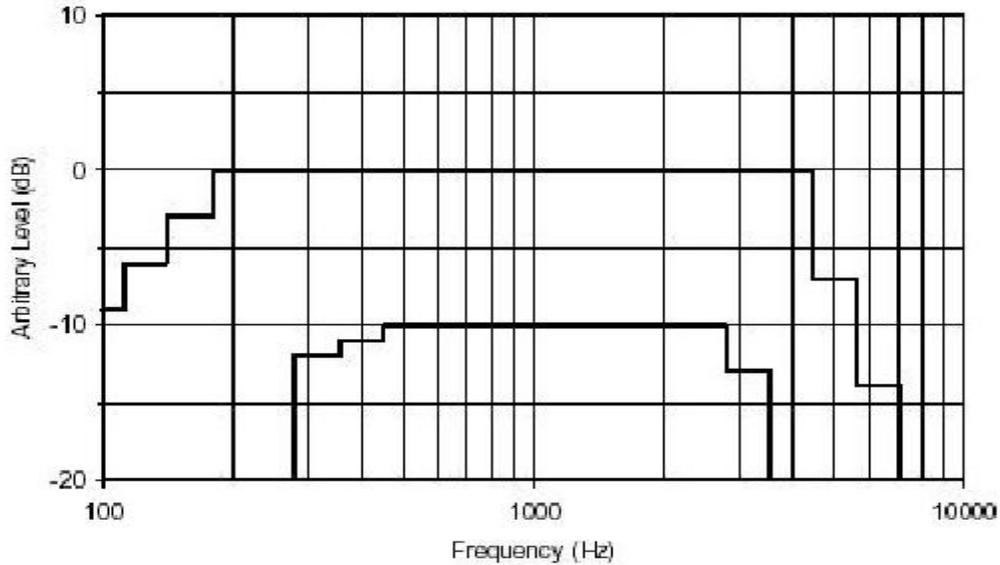


Figure 22 – Handsfree receive frequency response mask

7.2.3 Example Software Resource Usage

The software resources consumed by the equalizer filter module vary with factors such as the filter order and processing frame size. For example, the processor consumption increases with an increase in the filter order or the data frame size. Also, additional instructions must be executed if numerical overflow occurs. Hence, for a given filter order and frame size, more processor cycles are required if the equalizer filter saturates than if the filter does not saturate. Table 4 compares the average number of processor cycles executed by the equalizer filter module for the same frequency response and input signal, but with different filter order and frame size values.

Frame Size	Filter Order	Average Processor Cycles Executed
8	40	396
8	80	556
40	40	1675
40	80	2470

Table 4 – Comparison of processor requirements for different filter order and frame size values.

The memory requirements of the equalizer filter also vary with parameters such as the filter order. As the filter order increases, so does the amount of memory required to store the coefficient values. The code size of the FIR filter was 63 words of program memory.

The process of optimizing the equalizer filter software implementation for performance was a continuous one. Throughout the development of the software we made modifications to the implementation in order to improve efficiency. For example, as discussed in Section 6.2.4, one step we took to minimize processor requirements was to reduce the probability of branch misprediction. The results in Table 5 provide an example of the benefit of optimizing the equalizer filter for efficient software performance. The measurements in Table 5 were made before and after we modified the filter implementation to reduce the probability of an incorrect branch prediction. The measurements

	Filter Order = 40	Filter Order = 80
Avg. Processor Cycles Before Optimizing	1780	2580
Avg. Processor Cycles After Optimizing	1675	2470
% Reduction in Avg. Processor Cycles	5.9	4.3

Table 5 – Example results for optimized branch prediction.

show the average number of processor cycles executed for two test cases where the only difference was the filter order.

7.3 System Testing

7.3.1 Test Procedure

The purpose of the system testing was to ensure that the equalizer filter PxD function was operating correctly after integration into the entire PhonexChange software system. For our system level tests, we used a test application called RTP Blaster, which sends and receives G.711 compressed PCM data across the IP network. The basic test setup is shown in Figure 23.

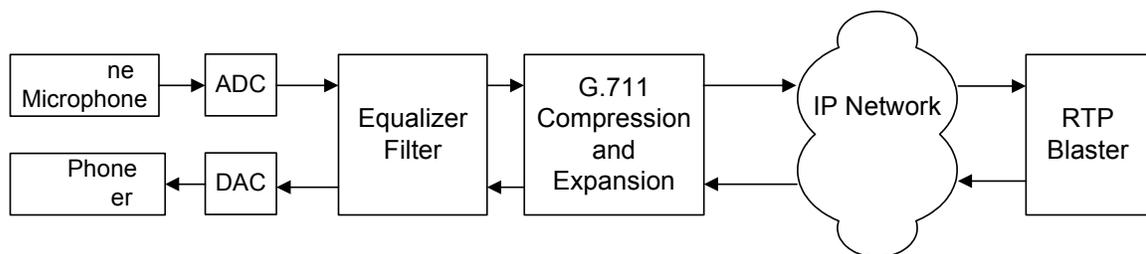


Figure 23 – System level test setup.

As shown in Figure 23, one of the existing PhonexChange components provides G.711 compression and expansion of the voice data. However, it is important to note that Figure 23 shows only the isolated PhonexChange components relevant to the system test setup. Other PhonexChange components, such as gain blocks and filters, are also present in the signal path and may modify the signals.

Using a software application to create the input vectors and record the output data, we tested the system to ensure that frequency response of the voice signals is shaped as expected when the filter is enabled.

7.3.2 Example System Test Results

The example discussed in this section qualitatively shows that the expected equalizer filter frequency response is being applied to the voice signals that pass through the PhonexChange IP phone system. This example uses the equalizer filter frequency response shown in Figure 15. The input data was a frequency sweep from 0 Hz to 4 kHz. The RTP Blaster application was used to send this G.711 compressed PCM format input data into the network. The output at the IP phone speaker was recorded. We repeated the test once with the equalizer filter disabled, and once with the equalizer filter enabled, and we compared the recorded output data to visually verify that the equalizer filter PxD function is functioning correctly. The filtered output data is shown in Figure 24 and Figure 25. The response shown in Figure 24 with the equalizer filter is disabled can be attributed to other hardware and software system components. For example, the speaker device used to play and record the signal has an effect on the system frequency response. In addition to the analog circuitry there are software components, such as a high pass filter, that also shape the frequency characteristics of the voice signals. However, with the equalizer filter enabled, it is clear that the filter response shown in Figure 15 is being applied to the voice signals. These figures show that the equalizer filter shapes the frequency characteristics of the voice signals passing through the entire IP phone system.

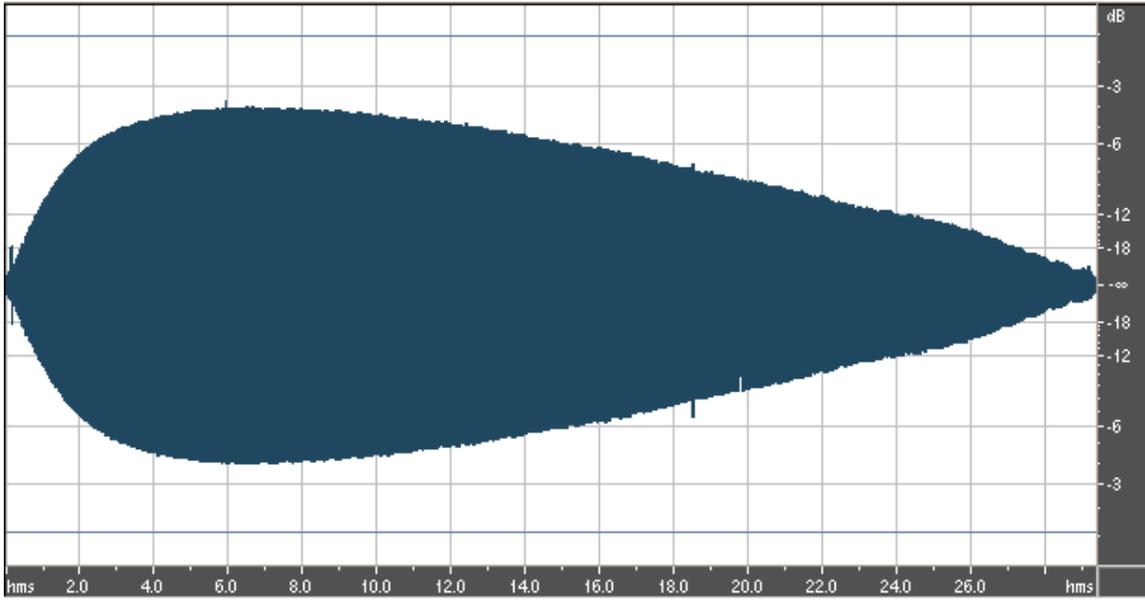


Figure 24 – System response to frequency sweep with equalizer filter disabled.

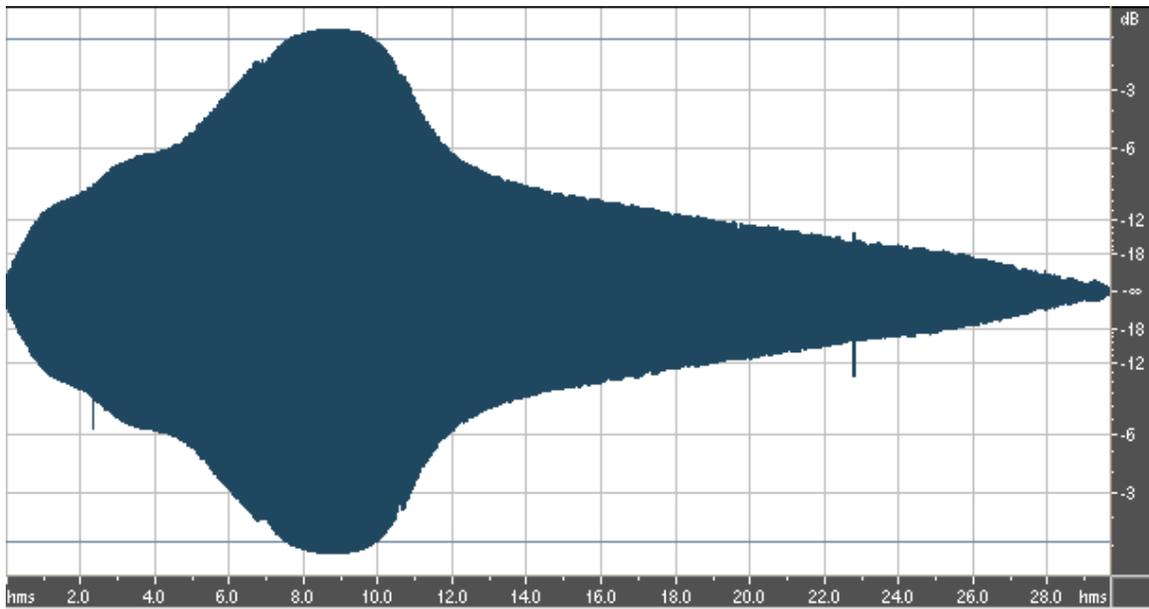


Figure 25 – System response to frequency sweep with equalizer filter enabled.

8 Conclusion

Our equalizer filter provides a flexible solution allowing IP phone users to modify the frequency characteristics of voice signals transmitted through the system. Our design method allows the equalizer filter to be easily adapted to meet various frequency response requirements. Implemented in software as a minimum phase FIR filter, the equalizer filter may be programmed through the provided application programming interface (API).

This thesis was completed within the context of the Broadcom IP phone software. We invested considerable effort to maximize the efficiency of the equalizer filter software implementation. We developed special test applications in order to characterize and verify the functional behaviour of the equalizer filter software. We also ensured that the software functioned correctly as an integrated component in the IP phone software system.

Future equalizer filter solutions could be modified and implemented in silicon instead of in software. Future enhancements of the equalizer filter could also include adding an IIR filter component to allow a finer approximation of the desired frequency response.

9 References

- [1] ANSI/TIA/EIA-810 - Transmission Requirements for Narrowband Voice over IP and Voice over PCM Digital Wireline Telephones, December 2000.
- [2] A. Oppenheim, and R. Schafer. *Discrete-Time Signal Processing*, 2nd Edition. New Jersey: Prentice-Hall, 1999.
- [3] CommWeb. Voice Over IP (VoIP) Tutorial, 2001.
<http://www.commweb.com/article/COM20010709S0016>
- [4] E. C. Ifeachor, and B. Jervis. *Digital Signal Processing, A Practical Approach*, 1st Edition. New York: Addison-Wesley Publishers Ltd. 1993.
- [5] G. Vink. *Programming DSPs using C: Efficiency and Portability Trade-offs*. May 2000.
<http://www.tasking.com/technology/papers.html>
- [6] LSI Logic Corporation. *Implementing FIR Filters on the ZSP Architecture Application Note*. September 2000.
- [7] R. J. Ridder. *Efficient Programming Techniques for Digital Signal Processing*. Oct. 1998.
<http://www.tasking.com/technology/papers.html>