University of Kentucky
UKnowledge

University of Kentucky Master's Theses | Graduate School

2006

POWER REDUCTION BY DYNAMICALLY VARYING SAMPLING RATE

Srabosti Datta
University of Kentucky, sdatt1@ENGR.UKY.EDU

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation

Datta, Srabosti, "POWER REDUCTION BY DYNAMICALLY VARYING SAMPLING RATE" (2006). *University of Kentucky Master's Theses*. 275.
https://uknowledge.uky.edu/gradschool_theses/275

This Thesis is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Master's Theses by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

ABSTRACT OF THESIS

POWER REDUCTION BY DYNAMICALLY VARYING SAMPLING RATE

In modern digital audio applications, a continuous audio signal stream is sampled at a fixed sampling rate, which is always greater than twice the highest frequency of the input signal, to prevent aliasing. A more energy efficient approach is to dynamically change the sampling rate based on the input signal. In the dynamic sampling rate technique, fewer samples are processed when there is little frequency content in the samples. The perceived quality of the signal is unchanged in this technique. Processing fewer samples involves less computation work; therefore processor speed and voltage can be reduced. This reduction in processor speed and voltage has been shown to reduce power consumption by up to 40% less than if the audio stream had been run at a fixed sampling rate.

KEYWORDS: Digital signal processors, Audio applications, Dynamic voltage scaling, frequency scaling, sampling rate

Srabosti Datta

08/24/2006

POWER REDUCTION BY DYNAMICALLY VARYING SAMPLING RATE

By

Srabosti Datta

Dr. William Dieter
(Director of Thesis)

Dr. Yu Ming Zhang
(Director of Graduate Studies)

08/24/2006

RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate school of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name	Date
Srabosti Datta	08/24/2006

THESIS

Srabosti Datta

The Graduate School
University of Kentucky
2006

POWER REDUCTION BY DYNMICALLY VARYING SAMPLING RATE

THESIS

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering at the University of Kentucky

By

Srabosti Datta

Lexington, Kentucky

Director: Dr. William Dieter, Asst. Professor

Electrical Engineering, Lexington, Kentucky

2006

Copyright © Srabosti Datta 2006.

ACKNOWLEDGMENTS

The following dissertation, while an individual work, benefited from the insights and direction of several people. First, my Thesis Chair, Dr. William Dieter, exemplifies the high quality scholarship to which I aspire. In addition, he provided timely and instructive comments and evaluation at every stage of the dissertation process, allowing me to complete this project on schedule. Next, I wish to thank the complete Thesis Committee: Dr. Dieter, Dr. Dietz and Dr. Lumpp. Each individual provided insights that guided and challenged my thinking, substantially improving the finished product.

In addition to the technical and instrumental assistance above, I received equally important assistance from family and friends. My husband, Siddhartha, provided on-going support throughout the dissertation process, as well as technical assistance critical for completing the project in a timely manner. My father, (Nilmoni Datta) and mother (Sarmistha Datta) instilled in me, from an early age, the desire and skills to obtain the Master's. Finally, I wish to thank the respondents of my study (who remain anonymous for confidentiality purposes). Their comments and insights created an informative and interesting project with opportunities for future work.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iii

TABLE OF CONTENTS.. iv

LIST OF FIGURES vi

LIST OF TABLES.. vii

LIST OF FILES viii

CHAPTER 1 1
INTRODUCTION 1

CHAPTER 2 4
PSYCOACOUSTIC MODEL 4
- 2.1 EQUI-LOUDNESS CURVES AND ABSOLUTE HEARING THRESHOLD. 4
- 2.2 CRITICAL FREQUENCY BANDS.. 5
- 2.3 MASKING PRINCIPLE.. 9

CHAPTER 3 10
RELATED WORK 10

CHAPTER 4 14
OVERVIEW OF SYSTEM DESIGN.. 14

CHAPTER 5 19
SIGNAL PROPERTIES 19
- 5.1 DETERMINATION OF HIGHEST FREQUENCY CONTENT..................... 19
- 5.2 MATLAB MODELLING FOR FEASIBILITY OF THE THEORY............... 20
- 5.3 DOWNSAMPLING AND UPSAMPLING 26
- 5.4 DYNAMIC FREQUENCY AND VOLTAGE SCALING............................... 27

CHAPTER 6 29
IMPLEMENTATION.. 29
- 6.1 TIME DOMAIN IMPLEMENTATION.. 31
  - 6.1.1 SELECTION OF FIR COEFFICIENTS.. 32
  - 6.1.2 FREQUENCY DETERMINATION 36
- 6.2 FREQUENCY DOMAIN IMPLEMENTATION 37
- 6.3 RESULTS 39
- 6.4 HARDWARE LIMITATIONS.. 42
- 6.5 AUDIO QUALITY TESTING 44

CHAPTER 7 46
CONCLUSION.. 46

7.1 FUTURE WORK 46

CHAPTER 8 48
APPENDICES 48
8.1 MATLAB CODE 48
8.2 C CODE 49

CHAPTER 9 66
REFERENCES 66

Vita 70

LIST OF FIGURES

Figure 1: Equi-loudness curve 5
Figure 2: Critical Band Responses 6
Figure 3: Relationship between bark and hertz scale 7
Figure 4: ATH curve in kHz 8
Figure 5: ATH curve in bark scale 8
Figure 6: Simultaneous masking curve 9
Figure 7: Hearing Aid Model 14
Figure 8: Scheduling of an audio frame 15
Figure 9: Schedule of an audio frame (lower CPU frequency and slack =0) 16
Figure 10: Enhanced Human Hearing Aid Model 17
Figure 11: Steps in CPU processing 18
Figure 12: Frequency distribution for voice samples 21
Figure 13: Frequency distribution for music sample 22
Figure 14: Calculated power consumption vs. number of load filters for music 24
Figure 15: Calculated power consumption vs. number of load filters for voice 25
Figure 16: Block Diagram for the Implementation Setup 31
Figure 17: Magnitude response of high pass filter (order =35) with cutoff frequency =12 kHz 34
Figure 18: Magnitude response of a lowpass filter (order =20) with cutoff frequency nearing 5 kHz 35
Figure 19: Magnitude response of a bandpass filter (order =60) with cutoff frequency with 35
Figure 20: Flowchart showing f_{max} determination in Time Domain method 36
Figure 21: Graph showing f_{max} determination in Frequency Domain method 38
Figure 22: Power consumption vs. no. of filters for music samples 39
Figure 23: Power consumption vs. No. of filters for voice samples 40
Figure 24: percentage of samples vs. set sampling frequency for music 41
Figure 25: percentage of samples vs. set sampling frequency for voice 41
Figure 26: Two-PLL model for separate clock inputs 43

LIST OF TABLES

Table 1: Frequency Bands 21
Table 2: Power and frequency values used for power calculations................................. 22
Table 3: Distribution of CPU clock cycles and power consumed in three different frequency bands (for Music samples) for different number of filters....................... 23
Table 4: Distribution of CPU clock cycles and power consumed in three frequency bands for voice samples 25
Table 5: Supported frequencies for voltages in TMS320C5510 DSP core processor...... 29
Table 6: Number of coefficients for each type of filter 34
Table 7: Average clock cycles using time domain for maximum frequency determination 37
Table 8: Average clock cycles using frequency domain for maximum determination 38

LIST OF FILES

Thesis.pdf....................684kB

CHAPTER 1

INTRODUCTION

Many modern digital audio applications involve digital signal processing techniques, which increase the performance of consumer-level devices such as cell-phones, hearing aids and portable radios. The development of advanced digital signal processing techniques has allowed the reproduction of accurate sound with little distortion. These processing techniques are achieved through complex algorithms, which are computation intensive. Therefore the devices need to support very high performance *Central Processing Units* (CPUs). Apart from being high performance devices, these portable devices have stringent size and power requirements. Hence small batteries have emerged as the preferred power sources for such portable audio devices. The lifetime of the batteries is roughly proportional to the amount of energy drawn from them. In the case of hearing aids, digital hearing aids require more power than analog hearing aids because they have more complicated signal-processing algorithms than in an analog hearing aid. Typically, battery life can range anywhere from 5-7 days [2]. This not only increases the final cost of the portable digital devices to many times the cost of conventional analog hearing aids but also necessitates frequent replacement of the battery. Therefore the inconvenience of short battery life has limited the popularity of digital hearing aids.

The battery life is a very important consideration for the usefulness of these applications. By controlling the battery life through *Digital Signal Processing* (DSP) techniques, the operating cost of a hearing aid can be significantly reduced, thereby reaching a larger portion of the population with a lower price, while delivering the same sound quality.

The battery life of any device is dependent on the power consumption of all its components. Various methods of minimizing power consumption have been explored at different levels of abstraction from sub-silicon level to application software level.

At the silicon level the power consumed by a CMOS transistor is governed by the following equation [38]:

$$P_{avg} = P_{switching} + P_{short\text{-}circuit} + P_{leakage} = \alpha_{0\text{->}1} \text{ x } (C_L \cdot V_{dd}^{2} \cdot f_{clk} + I_{sc} \cdot V_{dd}) + I_{leakage} \cdot V_{dd} \quad (1)$$

The $\alpha_{0\text{->}1}$ is the probability that a transition occurs (the activity factor). Any transition, whether *high* to *low* or *low* to *high*, involves power consumption. The first term inside the parenthesis represents the switching component of power, where C_L is the load capacitance, f_{clk} is the clock frequency and the second term in the parenthesis is due to the direct-path short circuit current, I_{sc}, which arises when both the NMOS and PMOS transistors are simultaneously active, conducting current directly from supply (V_{dd}) to ground. Finally, leakage current, $I_{leakage}$, which can arise from substrate injection and sub-threshold effects, is primarily determined by fabrication technology considerations.

In current semiconductor devices $P_{switching}$ dominates the other power terms in Equation (1). Since the energy expended for each switching event in CMOS circuits is $C_L \cdot V_{dd}^2 \cdot f_{clk}$, decreasing the frequency of the device can subsequently decrease the power. Reducing the clock frequency also allows the supply voltage to be reduced. Therefore dynamically controlling both frequency and voltage can lead to increased power savings. In audio applications, the workload of the processing in the CPU varies from sample to sample. If the CPU needs to do more computations than average, then it must run at a higher frequency to correctly reproduce the sound. If the frequency and voltage can be dynamically changed with the varying amount of processing, greater power savings may be obtained. Also, since most embedded applications are real-time, they have deadline considerations to ensure good quality samples. There have been Dynamic *Voltage Scaling* (DVS) algorithms, which exploit this principle [11, 18, 21, 31, 37]. DVS algorithms can be applied, ensuring that the jobs can meet their deadlines and the frequencies and voltages can be reduced to meet the deadlines. This approach not only increases the utilization of the processor but also decreases power consumption of the device while ensuring correct reproduction of sound.

In many DSP applications, a set of standard *Finite Impulse Response* (FIR) filters is used to do the processing work. A stream of data goes through the filters at a fixed sampling frequency. This sampling frequency is determined at design time by the Nyquist rate, which states, that DSP applications must sample their inputs at a frequency at least twice the highest frequency in the input signal to accurately reproduce the signal [19]. Typically the sampling rate is set at a rate higher than the Nyquist rate to improve signal quality. With a fixed sampling rate and a fixed set of FIR filters the demand for CPU processing varies little and therefore there is little scope for using DVS algorithms.

Dynamically varying the sampling rate in response to the input signal provides opportunities to vary frequency and voltage and thereby decrease power consumption. When the input signal has little perceptible high frequency content, the sampling rate can be reduced. A lower sampling rate reduces the number of samples to be processed while running at a lower sampling rate, allowing CPU speed to be reduced. When perceptible high frequency content is present, the system samples it at a higher rate, preserving signal quality. Using this *Dynamic Sampling Rate* (DSR) technique in a hearing aid application can reduce power consumption to about 40% of that without DSR.

Chapter 2 describes the psychoacoustic model of the human ear and its use for determination of the highest frequency content in a frame in both frequency and time domain. Chapter 3 deals with the related work, which serves as a background for the thesis. Chapter 4 describes the overall system design. Chapter 5 describes the signal properties of an audio signal. Chapter 6 discusses the software and the hardware implementation details. Chapter 7 holds the conclusion and the direction of future work.

CHAPTER 2

PSYCOACOUSTIC MODEL

The psychoacoustic model is based on the perception of human auditory system. The human ear receives information in the form of audio signals. In a digital hearing aid these signals are digitally represented in form of bits. Audio compression algorithms [22] have been used to obtain a minimum set of such bits representing audio signals. Audio compression is done to reduce processing and storage of data without any perceived distortion of the signal. The purpose of this kind of compression is to be able to reproduce a signal using less storage space or transmission bandwidth. These algorithms are derived from psychoacoustic principles. They identify imperceptible information and then compress the digital information by removing inaudible bits. The psychoacoustic principles that will be discussed in this thesis are equi-loudness curves, absolute threshold of hearing and masking principles associated with audio processing and critical band frequency analysis. Each property provides a way of determining which portions of a signal are inaudible to the average human, and can thus be removed from an incoming signal. These principles are needed to determine the maximum frequency content in a given frame of audio signals [39].

2.1 EQUI-LOUDNESS CURVES AND ABSOLUTE HEARING THRESHOLD

The average human ear does not hear all frequencies equally well. It is most sensitive to frequencies around 4 kHz with less sensitivity at higher and lower frequencies. The standard metric for measuring intensity of an audio signal is *Sound Pressure Level* (SPL). The SPL is defined as the intensity of sound pressure in decibels (dB) relative to a defined reference level, i.e,

$$L_{SPL} = 20 \log_{10} \frac{P}{P_0} \qquad (2)$$

where L_{SPL} is the SPL to be measured, P is the sound pressure of the stimulus in Newton per square meter (N/m^2) and P_0 is the standard reference level of 20 $\mu N/m^2$. In case of the human ear, sound pressure levels that are detectable at 4 KHz may not be heard at other frequencies. In general, two tones with equal SPL but different frequency will not sound equally loud. Equi-loudness curves at different loudness levels are shown in Figure 1. The x-axis shows the frequency in terms of *Barks*. Bark is a unit of measurement for frequency and is based on the perceptibility of a frequency by an ordinary human ear. We will discuss Barks in more details in Section 2.2. The dotted curve is the "hearing threshold in quiet" or the *absolute threshold of hearing* (ATH), which indicates the minimum level at which the ear can detect a tone at a given frequency. These curves indicate that the ear is more sensitive at some frequencies than it is at others. Therefore distortions will be more audible in the sensitive frequency ranges than in other frequency ranges.

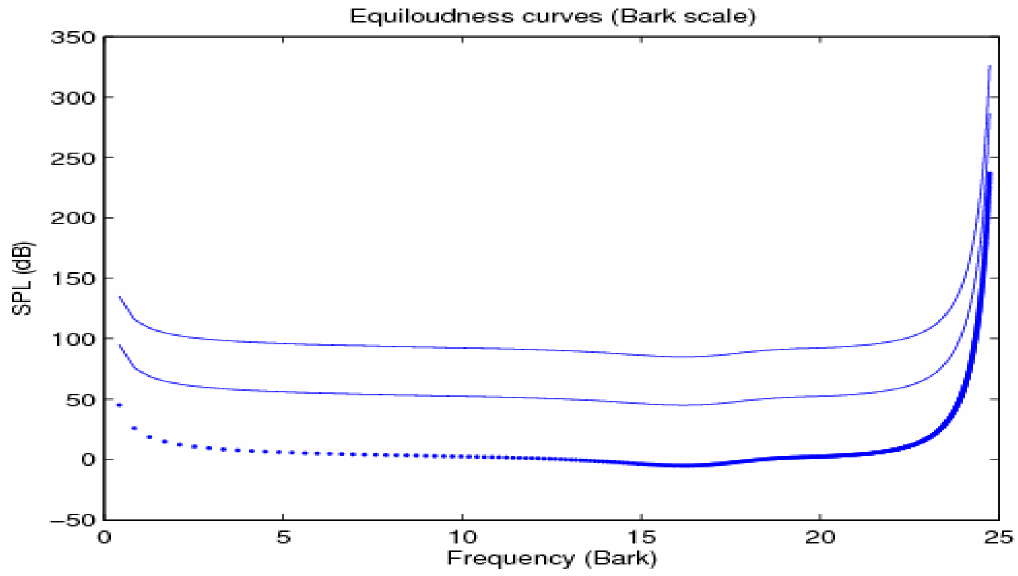

Figure 1: Equi-loudness curve

2.2 CRITICAL FREQUENCY BANDS

At the extreme frequencies, hearing a tone becomes more difficult. The human ear can detect differences in pitch better at lower frequencies than at higher frequencies. For example, a human has an easier time telling the difference between 500 Hz and 600 Hz than between 17,000 Hz or 18,000 Hz. The frequency range ranging from 20Hz to 20,000

Hz can be broken up into critical bands. *Critical bands* are a set of sub-bands of the audible frequency range. Critical bands occur when the sound signal hits the basilar membrane disturbing the membrane over a small area and exciting the nerve endings over that entire area. The entire range of critical bands has been determined according to different experiments. Frequencies within a critical band are similar in terms of the ear's perception, and are processed separately from other critical bands. The critical bands are much narrower at lower frequencies than at high frequencies; about, three quarters of the critical bands are located below 5 kHz. This indicates that the ear is more sensitive for low frequencies than for higher frequencies. In a particular critical band, a higher pitch of one frequency will mask the lower pitch of another frequency. Therefore the critical band can be considered as a "frequency selective channel of psychoacoustic processing" [36]. Any noise falling within the critical bandwidth can contribute to the masking of a narrow band signal. The human ear consists of a whole series of critical bands, each selecting a specific portion of the audible audio spectrum. Figure 2 shows a graph of the responses of several critical bands.

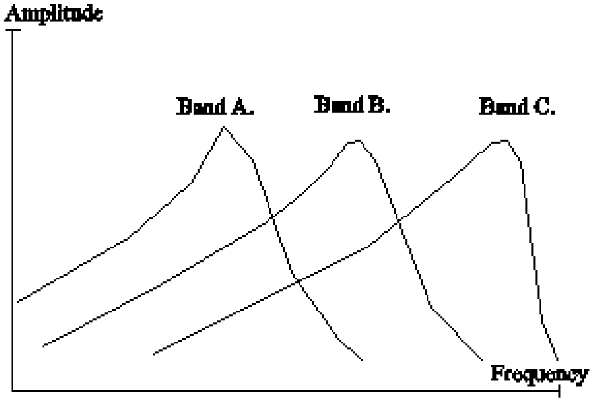

Figure 2: Critical Band Responses

These bands are non-uniform, non-linear, and dependent on the sound heard. Signals within one critical bandwidth are hard to separate for a human ear. A more uniform measure of frequency based on critical bandwidths is the Bark [39]. A Bark bandwidth is smaller at low frequencies (in Hz) and larger at high ones. The Bark scale is a psychoacoustical scale. This scale ranges from 1 to 24 and corresponds to the first 24 critical bands of hearing. The subsequent band edges are (in Hz) 0, 100, 200, 300, 400, 510, 630, 770, 920, 1,080, 1,270, 1,480, 1,720, 2,000, 2,320, 2,700, 3,150, 3,700, 4,400, 5,300, 6,400, 7,700, 9,500, 12,000 and 15,500. The Bark frequency scale can be approximated by the following equation:

$$f_{Barks} = 13 \cdot \arctan(0.00076 \cdot f_{hz}) + 3.5 \cdot \arctan((\frac{f_{hz}}{7500}))^2 \quad (3)$$

where f_{Barks} is the frequency in Barks and f_{hz} is the frequency in Hertz scale. The equation (3) has been plotted in Figure 3.

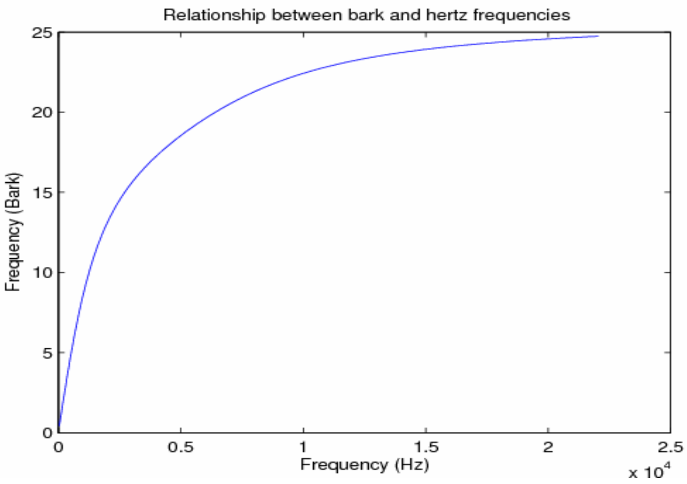

Figure 3: Relationship between bark and hertz scale

The ATH curve is an ideal example where bark scale can be used. Figure 4 shows the ATH curve in Hertz scale. The ATH curve in this figure has 0 dB around 0.4 KHz before rising steeply around 1 KHz.

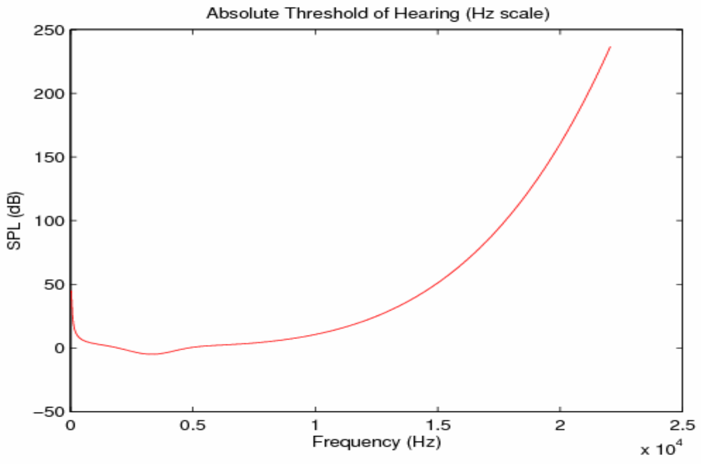

Figure 4: ATH curve in kHz

The ATH curve as shown in Figure 5 is drawn in Bark scale. In this figure it can be observed, that the ATH curve in bark scale expands along the frequency scale for low frequencies and the curve contracts at higher frequencies.

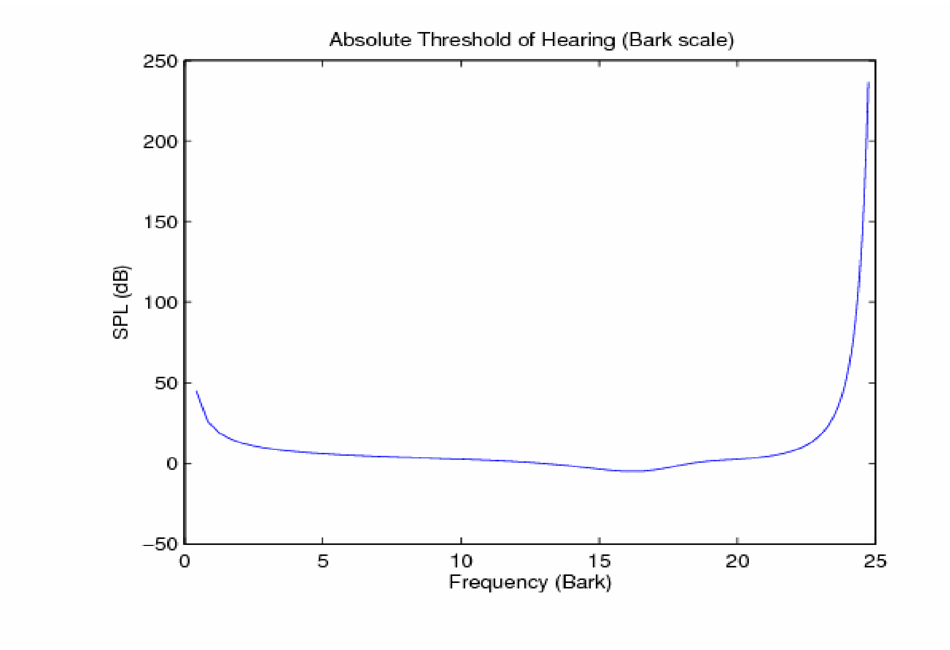

Figure 5: ATH curve in bark scale

The ATH has been determined experimentally [22]. In order to model the ATH into a mathematical equation, a sinusoidal tone was played at a very low power for many different listeners. The power was slowly raised until the tone was heard. This level at which the tone was heard was the threshold. The process was repeated for many frequencies in the human auditory range and with many test subjects. The experimental data gathered above can be modeled by the following equation, where f is frequency in Hertz:

$$\mathrm{ATH}\ (f) = 3.64 \bullet (\frac{f}{1000})^{-0.8} - 6.5 \bullet e^{-0.6 \bullet ((\frac{f}{1000})^{-3.3})^{2}} + 10^{-3} \bullet (\frac{f}{1000})^{4} \quad \text{(dB SPL)} \quad (4)$$

2.3 MASKING PRINCIPLE

Human ear does not have the ability to hear minute differences in frequency, especially when two signals are playing at the same time. This concept is known as simultaneous masking [39]. If one signal is strong, it will mask signals at nearby frequencies, making them inaudible to the listener. From a frequency-domain point of view, the relative magnitude of the masker signal and the maskee signal determine how one sound signal will mask the other. From a time-domain perspective, phase relationships between the audio signals will determine the extent of masking of signals. Therefore masking becomes stronger as the two sounds get closer together in both time and frequency. For a masked signal to be heard, its power will have to be increased to a level greater than that of a threshold that is determined by the frequency of the masker tone and its intensity. Figure 6 shows the example of simultaneous masking. Here the signal *b* at 14 Barks has power level above the threshold of hearing (dotted line) but still it is masked by signal *a* at 13 Barks, which has higher amplitude within the same critical band.

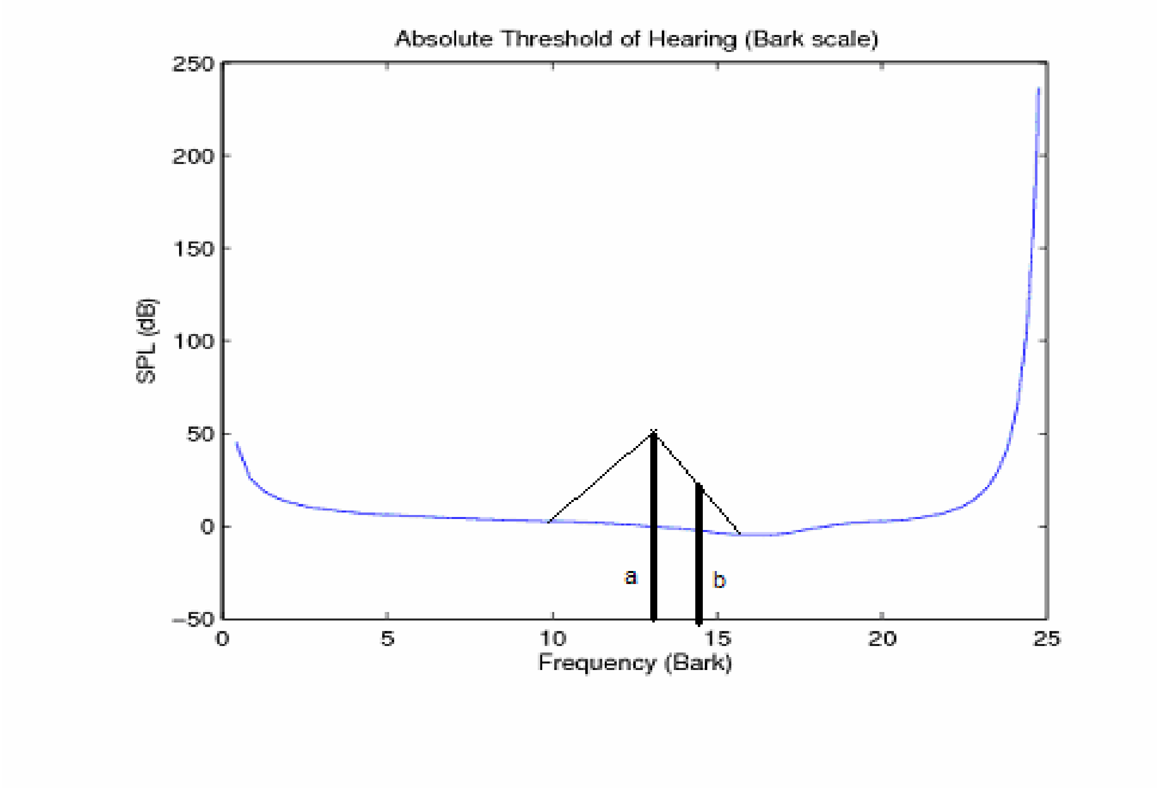

Figure 6: Simultaneous masking curve

CHAPTER 3

RELATED WORK

A number of power management techniques have been used in the audio applications. Most of these techniques are related to the device characteristics of the system or the hardware design of the integrated chips [4]. Some power-aware software algorithms and compiler techniques have been exploited in case of real time embedded applications [30]. Most of these systems have very tight temporal constraints.

In CMOS technology the dynamic component in the power consumption is the switching component from eq (1)

$$\text{Pswitching} = \alpha_{0\text{->}1}\, C_L \cdot V_{dd}^{\,2} \cdot f_{clk} \qquad (5)$$

From the above equation we need to determine which factor needs to be lowered by what amount to achieve maximum power reduction. Assume V_{dd} is scaled by a factor $S_{v,}$ where the S_v can be any positive value between 1 and 0, and F is scaled by a factor S_f, where the S_f can be any positive value between 1 and 0. The dynamic power equation becomes:

$$P_{switching} = \alpha_{0\text{->}1}\, C_L \cdot V_{dd}^{\,2} \cdot f_{clk} = \alpha_{0\text{->}1}\, C_L \cdot S_v^2\, V_{dd}^{\,2} \cdot S_f \cdot f_{clk}$$
$$= S_p^{\,2} \,.\, \alpha_{0\text{->}1}\, C_L \cdot V_{dd}^{\,2} \qquad (6)$$

where dynamic power scaling factor $S_p = S_v^2\, S_f$. From the equation it can be observed that reducing the core operating voltage will be the most effective way to reduce power since a small change in voltage can decrease/increase the power consumed quadratically.

One of the basic approaches of power consumption is shutting down voltage supply or the clock when the processor is idle. But the power saving is insufficient for cases where the CPU does not shut down completely but has reduced processing load. In such cases, reducing the voltage will also result in major power savings. Simunic et al.[31] takes into consideration *dynamic program management* (DPM) policies and tradeoff power consumption with performance by selectively placing components in low power states for MP3 applications.

One of the other power conserving mechanisms that dynamically reduces voltage and clock speed, is DVS [11, 18, 21, 31, 37]. In portable devices, performance and energy are the two tradeoffs in designing the system, considering the area is constant. DVS ensures minimization of energy consumption while meeting performance considerations. In our audio application i.e. the hearing aid, the processing unit, which mainly comprises of filtering operation circuits, is the computational load of the circuit. When the computational load is high, the throughput of the CPU should be high so that the CPU can process the load within the *deadline*. The deadline for a job is the time period within which the execution of the job should be completed. Therefore, the high performance requirement for the CPU will be met but at higher energy cost per computation. However these occurrences of high computational load are few. Therefore the total energy consumed will be less when there is no high computational load. DVS is one mechanism that ensures that the CPU can consume optimum power as well as produce maximum utilization. In order to meet peak computational loads, the processor is operated at its normal voltage and frequency, which is also its maximum frequency. When the load is lower, the operating frequency and the voltage are reduced to meet the computational and deadline requirements. According to Padmanabhan and Shin [21], *Real Time DVS* (RT DVS) algorithms can be used to modify the operating systems' real time scheduler and task management service to provide energy savings while maintaining Real Time deadline requirements. This is achieved by conducting schedulability test at certain intervals of the task.

However, most of the DVS algorithms use *worst-case execution time (WCET)* to determine deadline of a system. The disadvantage of using WCET is that a major portion of time remains unused by the CPU. Therefore the use of *slack* to dynamically determine the voltage and frequency becomes important. At any time t, the slack for any job with deadline d is calculated as time period "d – t". In DVS algorithms, the slack at each of the scheduling points is calculated and accordingly the clock frequency is updated to reduce speed a much as possible without violating any deadlines [13]. In cases where the slack cannot be predicted statically due to run-time variations, this slack is known as the *dynamic slack*. The slack is *static*, when the difference between the deadline and the execution time is fixed for any job [13], considering the release time between jobs to be fixed. Dynamic or static slack is used for reducing the energy consumption for of the computing unit. Manzak and Chakrabarti [14] give us an overview of the different scheduling algorithms, which use the concepts of slack that can be used for energy efficient scheduling. In our method slack is introduced by changing the amount of processing work depending on the input to the CPU.

Audio compression algorithms are used to occupy less memory space. Some of the recent work in multimedia applications takes the advantage of compression to achieve bandwidth reduction and as a result decrease power consumption of the system [17]. Choi, Dantu et al. [11] have proposed a method by which the processor decides the workload depending on the previous history of incoming frames and the computing power associated with each type of frame. Depending on the workload, the voltage and the frequency are scaled accordingly. Weiser et al. [37] describe a method which uses the information of energy consumption in the previous frame to set the future deadline. These are predictive algorithms and the frame is processed within the deadline requirements to maintain the quality of the data. It is a hard constraint for any real time system to maintain the quality of service. Therefore the CPU time to be allocated and the voltage profile on a variable voltage system are determined such that all the applications' requirements are satisfied and the total energy consumption of a system is minimized.

Im et al. proposed a DVS technique for multimedia applications in which idle intervals of the processor are utilized using buffers [11]. The buffers are used so that the workload for many frames can be averaged and the slack time for multiple frame time periods can be used to process multiple input samples. This reduces the total energy consumption of the system. Each task period (frame period) is divided into time slots and the voltage level is adjusted such that the execution time is maximized to the WCET (worst case execution time). Maxiaguine and Chakraborty [18] present another DVS algorithm for processing multimedia streams on architectures with restricted buffer sizes. The main advantage of this scheme is that it provides hard Quality of Service (QoS) guarantees apart from considerable power savings. Buffering of more than 20 ms frame cannot be applied to applications such as hearing aid because it introduces a perceivable delay in sound.

Compression algorithms are generally used to minimize the storage space of samples [20]. In our case, compression is used to minimize the data so that less computation in performed. Nevertheless in our work it is ensured that compression of the data does not introduce perceptible noise in the system and thereby does not distort the perceived sound quality.

CHAPTER 4

OVERVIEW OF SYSTEM DESIGN

A digital hearing aid model demonstrates our power saving mechanism. A simple digital hearing model aid consists of a microphone, A/D converter, DSP processor, D/A converter, and a speaker.

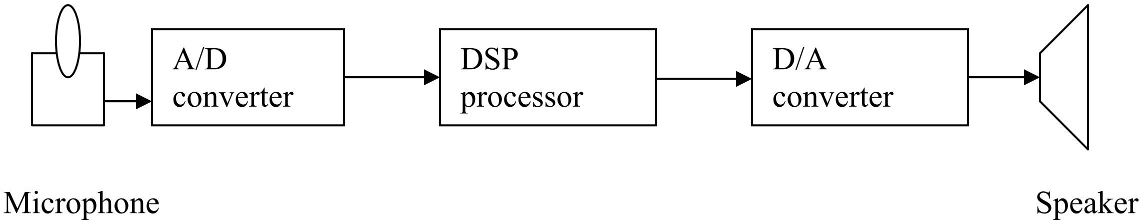

Figure 7: Hearing Aid Model

The function of the microphone is to catch incoming audio signals. The A/D converter converts analog signals from the microphone into digital signals, which are sent to the DSP for processing. Finally the processed digital signals are sent to the D/A converter where they get converted to analog signals and output to the speaker as amplified audio signals.

In modern digital hearing aids, the A/D converter samples the incoming audio signal at a fixed rate; typically 48 kHz or 44.1 kHz. This rate is fixed with consideration of the sampling theorem, which states that for a limited bandwidth signal with maximum frequency f_{max}, the equally spaced sampling frequency f_s must be greater than twice of the maximum frequency f_{max}, i.e.,

$$f_s > 2 \cdot f_{max} \qquad (7)$$

in order to have the signal be uniquely reconstructed without aliasing. The frequency $2 \cdot f_{max}$ is called the Nyquist sampling rate [19].

The human ear can hear sounds across the frequency range of 20 Hz to 20 KHz. According to the sampling theorem, the sound signals should be sampled at least at 40 KHz in order for the reconstructed sound signal to be acceptable to the human ear. Signal components higher than 20 kHz cannot be detected, but they can still pollute the sampled signal through aliasing [26]. Therefore, frequency components above 20 kHz are removed from the sound signal before sampling by a low-pass filter. Ideally all signals above the cutoff frequency would be attenuated to 0, but real filters allow some signals. The sampling rate is typically set at 44 KHz (rather than 40 kHz) in order to avoid signal contamination from the filter rolloff [30].

As discussed in Chapter 0, not all signals within the audible range i.e. from 20Hz to 20 kHz range can be heard equally well, even by a person with normal hearing capability. The signal may be distorted, but the distortion might not be perceptible. Therefore the hearing aid needs to process only those signals that a person with normal hearing capability can perceive based on the psychoacoustic principles of human hearing.

After the A/D converter samples the audio signal at a constant rate, the sampled audio is divided into frames and stored into a temporary input buffer. A frame is a fixed time period, T_{frame}. The CPU processes the data from the input buffer, one frame at a time, and transfers the processed frame to the output buffer. If the time taken to sample the input data is T_{samp}, time required for CPU processing is T_{exec} and the total time taken for data transfer from the A/D to input buffer and output buffer to the D/A converter is represented as $T_{data\ transfer}$, then the total time taken for doing all these tasks must be less than or equal to $T_{frame.}$ Otherwise data transfer buffers will overflow, causing distortion in sound. This relationship is defined by Equation (8) and is shown in Figure 8.

$$T_{samp} + T_{exec} + T_{data\ transfer} \leq T_{frame} \quad (8)$$

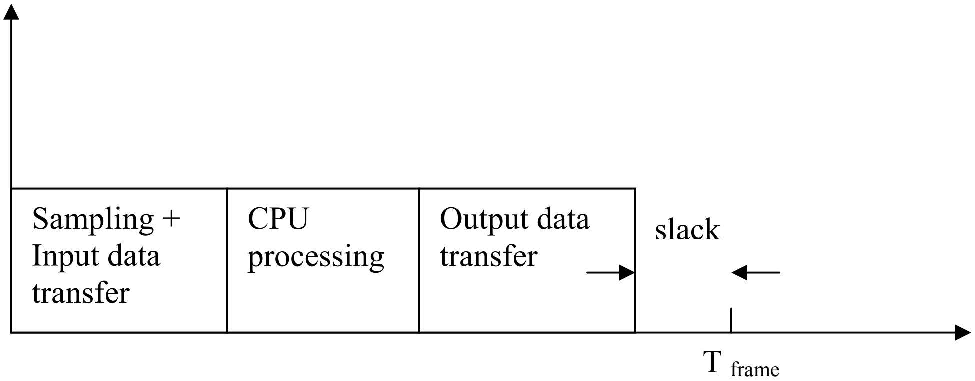

Figure 8: Scheduling of an audio frame

The time difference between T_{frame} and the last byte of output data transferred is known as slack [13]. The slack is shown in Figure 8. Higher slack reduces utilization of the DSP. The slack can be utilized such that the CPU can run at slower speed and complete all the tasks in time. Though the sampling rate in existing audio devices is fixed, it does not need to be for all frames. When no high frequency samples are present in a frame, the frame does not require as many samples to be processed. Fewer samples require less processing time. With less processing to do the processor can reduce its speed. The sampling frequency for that low frequency frame will be adjusted, such that the sum of T_{samp}, T_{exec} and $T_{data\ transfer}$ is equal to that of T_{frame}. This will increase the utilization of the DSP and as per Equation (8), the power consumption for a low speed CPU will be less than that of a normal CPU. Therefore a lower sampling rate should be used with frames containing low frequency components.

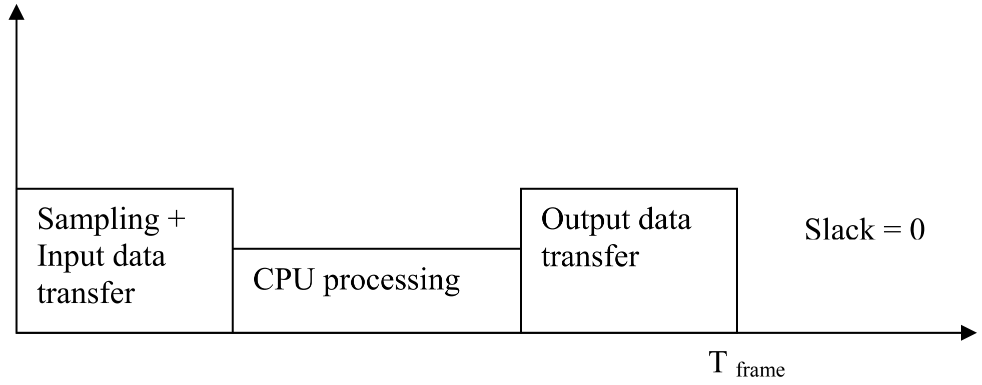

Figure 9: Schedule of an audio frame (lower CPU frequency and slack =0)

Nevertheless, a frame with higher frequency components needs to be sampled at a higher rate so that no aliasing happens on the input frame. In using a variable sampling rate approach, the sampling frequency would be lowered for frames when the audio signal has inaudible high frequency components and increased when high frequency components are present.

The CPU processing will go through the standard hearing aid processing steps like amplification and masking techniques as in normal hearing aid. These steps will be done by a set of digital filters. For each set of frequencies, there will be separate set of filters depending on the sampling frequency. The execution time will increase or decrease depending on the filter coefficients and the number of samples to be processed. Therefore the execution time in the filters is dependent on the sampling frequency determined earlier. By decreasing the sampling frequency, we also reduce the operating frequency of the CPU, which results in less dynamic power consumption as in Equation (1). Due to the reduction in frequency, there is an increase in execution time of CPU. Nevertheless the sum of total execution time and buffer transfer time is below $T_{frame.}$

After the samples have gone through processing steps like filtering, amplification etc., the number of output samples is made equal to the number of input samples by a method called upsampling. The reason is that in our setup the same codec does both A/D conversion and D/A conversion. Therefore the number of output samples to the D/A converter needs to be same as the input samples from the A/D converter. Upsampling ensures that the D/A converter will have enough data to reproduce a new signal at a fixed sampling rate. In cases where the A/D converter is different from D/A converter, the number of output samples may be different from the input samples.

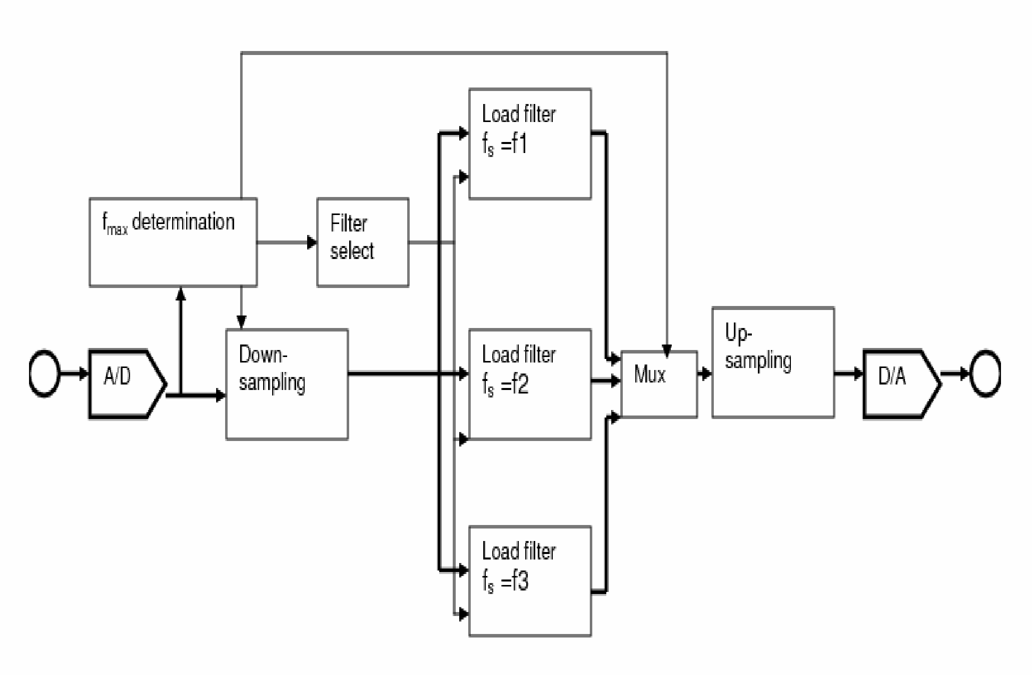

Figure 10: Enhanced Human Hearing Aid Model

Figure 10 shows the model of the variable sampling rate hearing aid. The f_{max} determination process determines the sampling rate of the CPU depending on the frequency content of the signal in a frame of fixed time period. The determined sampling rate is set for the subsequent filtering operations in the CPU. The downsampling and the upsampling stages depend on the sampling rate determined by the f_{max} determination process.

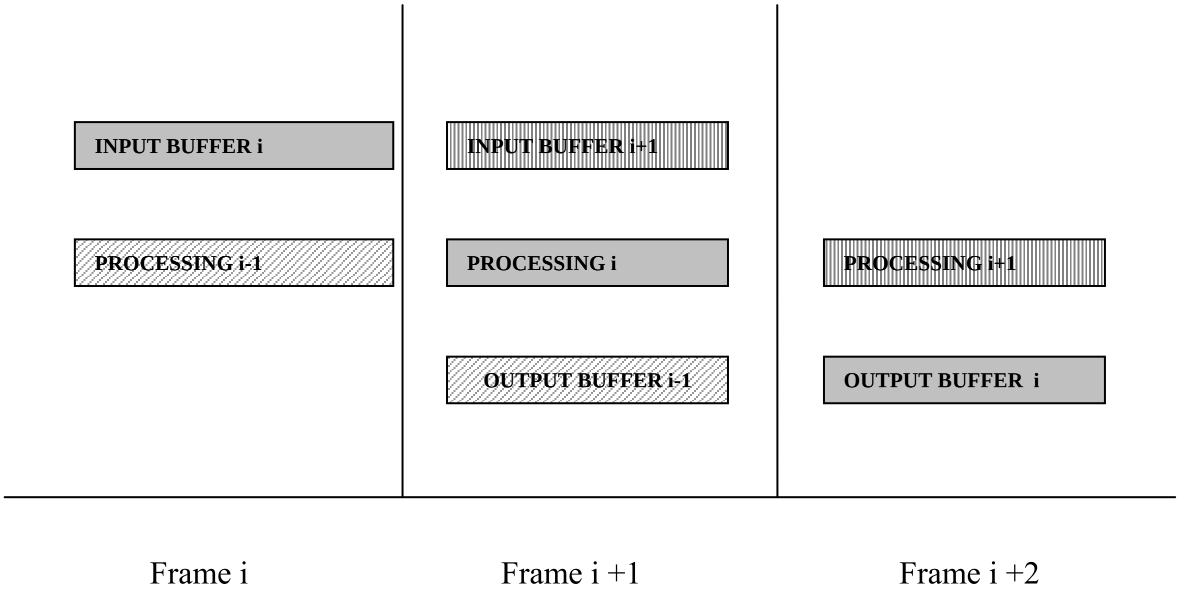

Figure 11: Steps in CPU processing

Figure 11 shows all the steps in which data are processed by the CPU. The different data packets are shown in different shading patterns. The packets with the same pattern have data from the same input frame, which undergoes different steps of operation in different frames. In the first frame period the input data is transferred to the input buffer. In the second frame period, the frame undergoes DSR processing steps. In the third frame period the processed frame is transferred to the output buffer.

CHAPTER 5

SIGNAL PROPERTIES

We propose a method to decrease dynamic power consumption by processing a continuous audio stream after segregating it into frames. Each of these frames is processed separately at a speed depending on the frequency content in that frame. The sampling frequency and the voltage are adjusted according to the frequency content.

The sequence of samples coming from the A/D converter is divided into frames. The number of samples produced by the A/D converter depends on the sampling rate of the codec. Each frame represents the same amount of time. A 20 ms frame size is sufficient to produce stationary signals and does not introduce a delay in the signal quality [34]. The audio signals are transferred from the A/D converter where it had been sampled at the constant speed of 48 KHz. Each 20 ms of a frame should constitute 20 ms x 48 KHz = 960 digitized sound samples.

5.1 DETERMINATION OF HIGHEST FREQUENCY CONTENT

There are two ways in which the maximum audible frequency in a frame is calculated: frequency domain method and the time domain method. In the frequency domain method, the audio frame is transformed into the frequency domain using Fast Fourier Transforms (FFT). In our implementation we use 1024 samples or 21.33 ms frame size since we have wanted to use power of 2-FFT operations as suggested by Cooley-Tukey method [26]. We may have used other FFT operations other than Cooley-Tukey method but the clock cycles required for other FFT operations are more. Therefore power consumption is more for other FFT operations. The FFT determines the *Power Spectral Density* (PSD) of the signal for each frequency. The CPU checks for the highest frequency in the sampled signal with PSD value greater than the ATH curve value. The dynamic sampling rate f_s is set to at least 2 times the highest frequency determined f_{max}, so that no aliasing takes place.

The f_s can be set to a range of frequencies - 48 KHz, 24 KHz, 12 KHz, 6 KHz, and 3 KHz as allowed by our implementation setup. The DSR algorithm selects the lowest available dynamic sampling rate, which is at least two times greater than the maximum audible frequency in a frame. The FIR filters are designed to work at a particular sampling frequency. Depending on the selected sampling frequency, a set of filters is selected which supports operation with this sampling rate. Also, the audio signal has to be downsampled for subsequent filtering operations.

The FFT is an accurate method for calculating PSD in a time frame but it is very expensive in terms of clock cycles. The second method for determining highest frequency content in a frame is by using a cascade of time domain FIR filters. These filters can be used to determine the power for a set of frequencies and will have cutoff frequencies depending on the critical bands of frequency. The frequency responses of the time domain filters are such that they emulate the ATH curve as closely as possible. The signals are first passed through a high pass filter comprising the highest critical band. The power under that curve is calculated using equation:

$$P=\sum_{i=1}^{n} \frac{1}{n}\times X_i^2 \quad (9)$$

Where X_i is the amplitude of a particular sample i and n is the number of samples calculated. In our implementation, n= 1024. This power is compared with the power under ATH curve in that region. If the total power of samples is greater than the ATH power in that critical band, then the computation for highest frequency stops there and the sampling frequency is set to the nearest available frequency which should be approximately equal to double the highest frequency for that filter. If the power is less than ATH power of that critical band, then the power computation and comparison takes place for the band pass filter of the next lower critical band and this process is repeated until the power becomes greater than the ATH curve for that critical band.

Mathematical modeling and implementation tests described later show that the FFT method is a superior method both in terms of accuracy and also power savings.

5.2 MATLAB MODELLING FOR FEASIBILITY OF THE THEORY

To determine the feasibility of the proposed theory of power savings using dynamic sampling rate, Matlab simulations were done to find the distribution of frequency in the audio spectrum. Two different types of audio inputs were chosen, Voice and Music, to determine the difference in frequency spectrum of the two types of audio signals. Each of these audio inputs was categorized into three frequency bands - Low, Medium and High. The three frequency bands were defined as follows:

Table 1: Frequency Bands

Frequency Band	Frequencies
Low	F < 5 KHz
Medium	5 KHz < F < 12 KHz
High	F > 12 KHz

The method of determination of the frequency bands as shown in Table 1 has been described in Section 6.1.1. The audio was divided into frames of 1024 samples. The highest frequency content sample with a PSD above the ATH curve is determined for each frame. The frequency determined for each frame is then categorized into one of the three bands in Table 1.

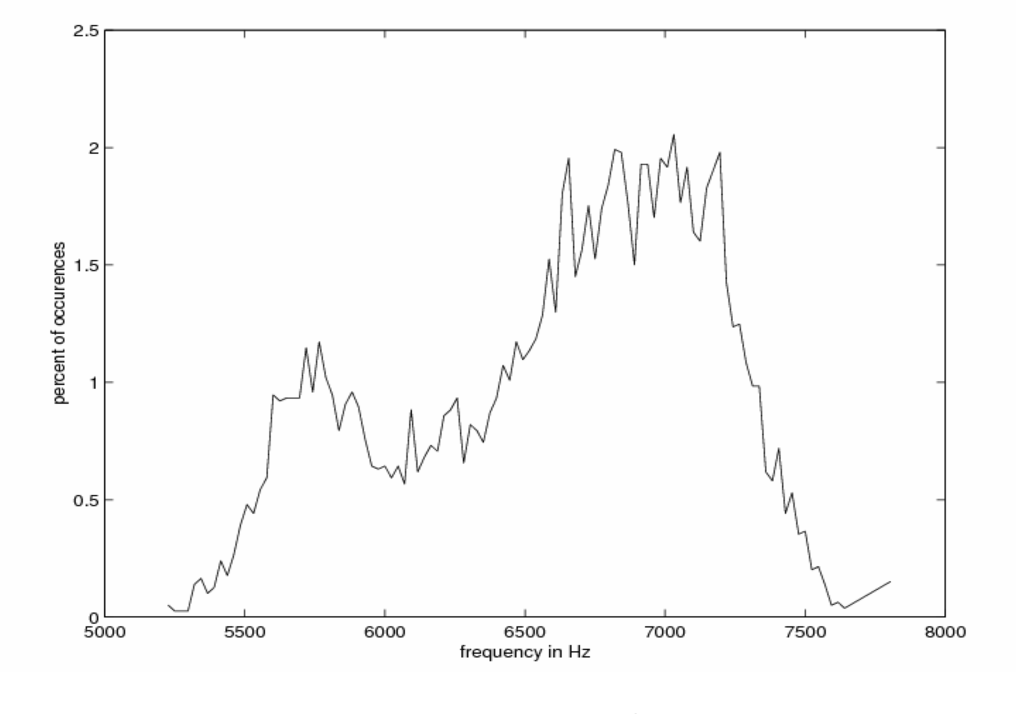

Figure 12: Frequency distribution for voice samples

Figure 12 shows the frequency distribution of highest frequencies of a voice sample [1]. From this test it has been determined that about 19% of the samples are in the low frequency range, 80% in medium frequency and the remaining 1% in the high frequency range. Therefore the CPU needs to run only for 1% of the time with maximum frequency and voltage and rest of the time at reduced frequency and voltage.

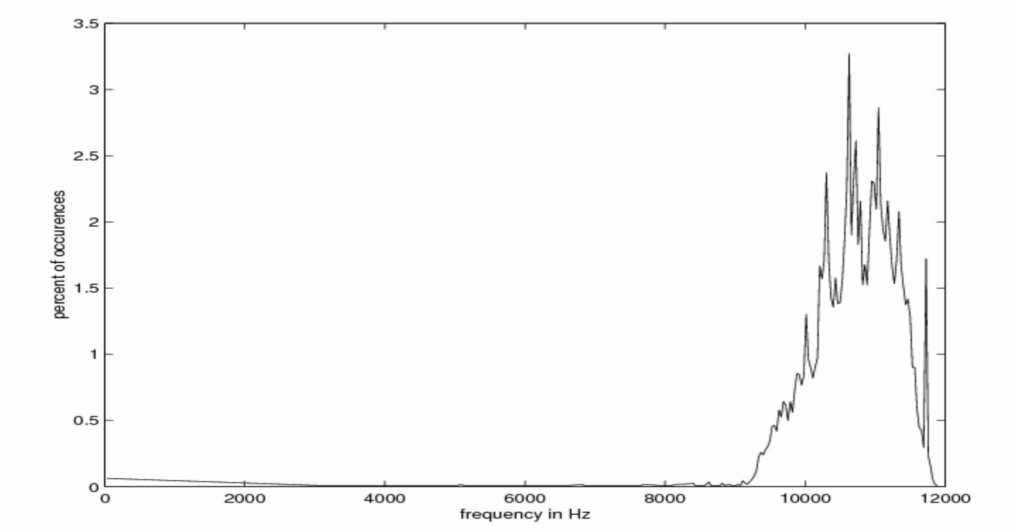

Figure 13: Frequency distribution for music sample

Similarly for the music sample [6] in Figure 13, less than 1% of the samples are in low frequency range, 95% in medium frequency and about 2%-3% samples in the high frequency range. Therefore the CPU needs to run only for 2-3% of the time with maximum frequency and voltage and rest of the time at reduced frequency and voltage in case of music samples. The percent of time the sampling rate is set at low, high or medium have been determined from the frequency distribution of the voice samples (Figure 12) and music samples (Figure 13) and is collated in Table 2.

Table 2: Power and frequency values used for power calculations

Input Type	Band	Percent of time (%)
Music	Low	1
	Medium	95
	High	4
Voice	Low	19
	Medium	80
	High	1

Based on the data sheet of the TMS320C5510, the power is calculated based on the following equation:

$$P_{average} = P_{static} + P_{low} \,.\, T_{low} + P_{med} \,.\, T_{med} + P_{high} \,.\, T_{high} \quad (10)$$

where, $P_{average}$ is the average power consumed by the CPU. P_{static} is the power consumed when all the CPU and the internal memory accesses are idle. P_{static} has been measured as 12.15 mW. P_{low}, P_{med} and P_{high} are the power values in mW consumed at low, medium and high frequency band respectively. T_{low}, T_{med} and T_{high} are the percent to time the frequency is low, medium and high respectively. The values of P_{low}, P_{med} and P_{high} depend on the clock frequency of the CPU which also depends on the number of processed filters and the amount of time to do all the computations in a frame.

The power consumed with worst-case current values for the CPU is calculated for different number of load filters. The number of coefficients required for each of the load filters is determined by analysis, which will be discussed in Section 6.1.1. The values of the clock cycles for different number of load filters are taken from TMS320C5510 datasheet. The power values have been calculated based on the worst-case output load current in the datasheet.

Table 3: Distribution of CPU clock cycles and power consumed in three different frequency bands (for Music samples) for different number of filters

No of filters	Low		Medium		High	
	f_{clk} (Hz)	P_{low}(mW)	f_{clk} (Hz)	P_{med}(mW)	f_{clk} (Hz)	P_{high}(mW)
0	5,672,800	2.8	5,670,000	3.18	5,673,000	5.45
2	5,790,700	8.53	16,670,000	9.67	6,138,000	16.55
4	5,908,700	14.25	27,650,000	16.16	6,603,000	27.65
6	6,026,700	19.98	38,630,000	22.65	7,067,000	38.75
8	6,144,700	25.71	49,620,000	29.14	7,532,000	49.86
10	6,262,600	31.43	60,610,000	35.62	7,997,000	60.96
16	6,616,600	48.61	93,570,000	55.09	9,392,000	94.27
20	6,852,500	60.06	115,550,000	68.06	10,321,000	116.47
27	7,265,400	80.09	154,000,000	90.77	11,949,000	155.33
30	7,442,400	88.68	170,480,000	100.51	12,646,000	171.99
37	7,855,300	108.71	208,940,000	123.21	14,273,000	210.85

Table 3 shows the distribution of CPU clock cycles and power consumption in three different frequency bands for different number of filters for music samples and Figure 14 shows the power consumption calculated for music samples for increasing number of filters.

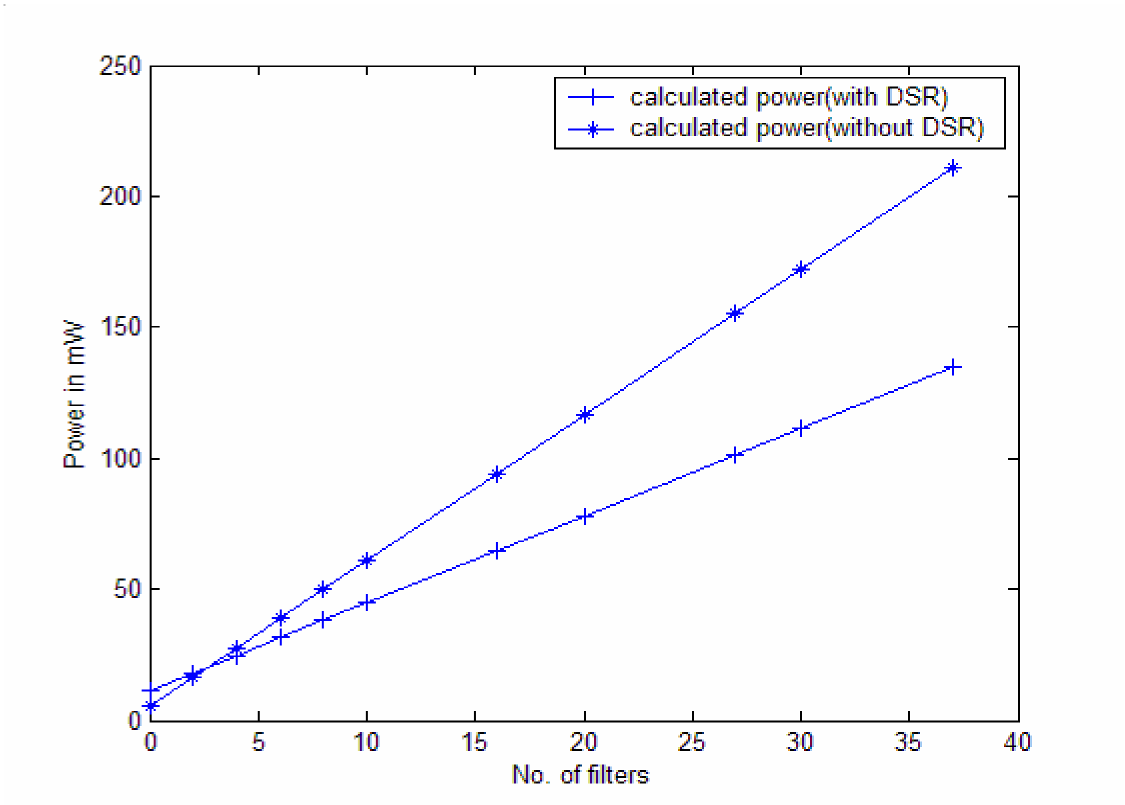

Figure 14: Calculated power consumption vs. number of load filters for music

Table 4 shows the distribution of CPU clock cycles and power consumption in three different frequency bands for different number of filters for voice samples and Figure 15 shows the power consumption calculated for voice samples. It can be observed that the voice samples have fewer frames with high frequency content. Therefore the power consumed for a large number of load filters is less.

Table 4: Distribution of CPU clock cycles and power consumed in three frequency bands for voice samples

No of filters	Low		Medium		High	
	f_{clk} (Hz)	P_{low}(mW)	f_{clk} (Hz)	P_{med}(mW)	f_{clk} (Hz)	P_{high}(mW)
0	5,673,000	2.8	5,670,000	3.18	5,672,800	5.45
2	7,219,000	6.83	12,180,000	7.74	5,756,400	13.25
4	8,766,000	10.86	18,680,000	12.30	5,840,000	21.05
6	10,312,000	14.88	25,190,000	16.87	5,923,600	28.86
8	11,859,000	18.91	31,690,000	21.43	6,007,200	36.66
10	13,406,000	22.93	38,190,000	25.99	6,090,900	44.45
16	18,045,000	35.00	57,710,000	39.67	6,341,700	67.88
20	21,138,000	43.05	70,720,000	48.79	6,509,000	83.49
27	26,551,000	57.14	93,480,000	64.75	6,801,600	110.52
30	28,871,000	63.17	103,240,000	71.60	6,927,100	122.52
37	34,284,000	77.26	126,000,000	87.56	7,219,700	149.83

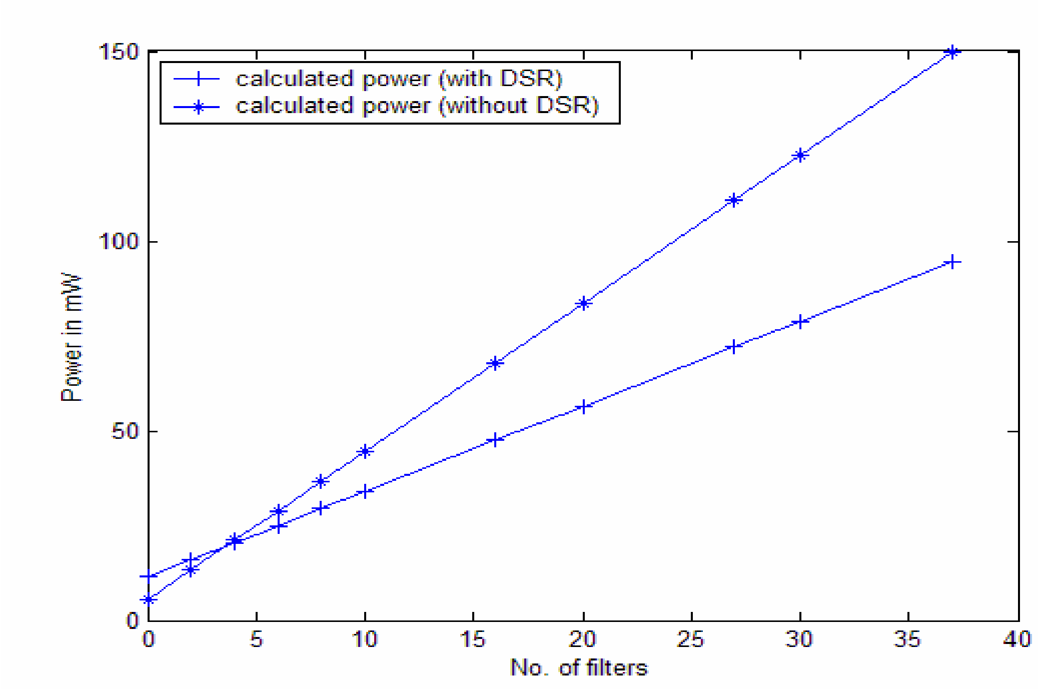

Figure 15: Calculated power consumption vs. number of load filters for voice

5.3 DOWNSAMPLING AND UPSAMPLING

After the frequency with highest audible SPL has been determined for a frame, the input samples to the filters are adjusted based on the output sampling rate of the frequency determination stage. The number of samples to be processed in the subsequent filters should be adjusted so that number of input samples from the A/D converter is same as the number of output samples to the D/A converter. Therefore if a lower sampling rate is set after frequency determination, the clock frequency for subsequent filtering operations is also reduced. The number of samples needs to decrease for CPU processing at a reduced frequency, since processing the original number of samples with a reduced operating frequency will take more execution time for the DSP. The number of samples will be reduced depending on the highest frequency component signal in a frame. The dropping of samples depending on the sampling rate is known as downsampling. The downsampling ratio is equal to A/D converter sampling frequency divided by f_s. The downsampling ratio should be an integer to minimize downsampling overhead. For example, if the downsampling factor is two, then every other sample will be dropped. In this case the number of samples to undergo CPU processing is half of the original number. Similarly, if the sampling rate is decreased by a factor of three, then every third sample will be kept from the original input signal. The above relation has been represented by a general equation

$$B[x] = A[Mx + (M - 1)] \qquad (11)$$

where A [] is the input array which consists of the original number of samples sampled by the A/D converter at 48 kHz sampling frequency, B[] is an array to store the result of the down sampling, M is the factor by which the sampling rate has been reduced, $x = 0$ to K. K represents the size of the output buffer to the CPU. If N is the input buffer size, then $K = N / M$.

After the samples have gone through processing steps like filtering, amplification etc., the number of samples fed into the D/A converter should be equal to the original number of samples produced by the A/D converter. The reason is that in our implementation, the same codec does both A/D and D/A conversion and it requires the input and output rate to be the same. In cases where separate A/D and D/A converters are used, the input frequency need not be equal to the output frequency. Therefore, in our implementation setup if a frame has undergone downsampling before the processing stage, it needs to be upsampled and converted back into analog signals to be transmitted by the speaker. Up sampling ensures that the D/A converter will have enough data to reproduce a new signal at a fixed sampling rate. Upsampling is done by predicting the value between two samples and inserting the predicted value between the two samples in order to increase the number of samples. We use linear interpolation to determine the data between two adjacent signals [9]. There are other interpolation methods like convolution with sinc function or bilinear interpolation that can be used for data interpolation but linear interpolation has been used because it is computationally inexpensive and is easy to use. Linear interpolation averages the two sample values weighting them by the ratio of the distance of the point to each sample. Linear interpolation assumes that the rate of change between any two points is constant.

Linear interpolation may introduce noise into the system because linear interpolation introduces a fairly large amount of aliased signals at higher frequencies [32]. Therefore it must be ensured that upsampling and downsampling does not affect any perceivable quality of the signals.

5.4 DYNAMIC FREQUENCY AND VOLTAGE SCALING

The amount of computation required to filter each frame is directly proportional to the number of samples in a frame. When the samples below the ATH curve are discarded, the difference in the signal output should be indistinguishable from a frame in which all samples were processed, so that the discarded work had zero value. It has been observed that operating the DSP with DSR technique utilizes the slack more effectively than if the DSP runs at a fixed sampling rate [21]. This behavior is governed by equation (8). In this equation T_{exec} can be defined as

$$T_{exec} = \frac{clock\_cycles}{f_{clk}} \qquad (12)$$

In order to utilize the slack, the execution time for that low frequency frame is adjusted, so that the sum of the processing time and frequency determination time uses up the whole 20 ms frame. With change in frequency, the voltage gets dynamically changed.

The actual operating voltages and the frequencies will depend on the operating ranges the hardware can support. In some hardware, a particular voltage will support only a set of frequencies. The CPU should operate such that it fully utilizes the slack and at the same time operates at the lowest possible frequency. This will not only ensure that the CPU consumes less power but also meets the no deadline requirements. Thus, by dynamically scaling both voltage and frequency of the processor based on computation load, DVS can provide the performance to meet the computational demands.

CHAPTER 6

IMPLEMENTATION

A Texas Instrument DSP kit TMS320C5510 was used to simulate the psychoacoustic functions of a hearing aid and do the variable sampling rate measurements [7]. The TMS320C5510 chip is a low power DSP [6]. This kit consists of a TMS320C5510 DSP processor and an AIC23 stereo codec. The codec acts as A/D and D/A converter and supports a range of sampling rates from 32 kHz to 96 kHz. However, adjusting the codec speed takes considerably longer than one frame period; hence the codec was operated at a fixed sampling rate. The core processor allows voltage scaling and frequency scaling. The voltages supported are 1.1 V and 1.6 V. The frequency can be scaled from 6 MHz to 200 MHz. Not all voltages support all the frequencies. Table 5 summarizes the frequencies supported by each of the voltages [33].

Table 5: Supported frequencies for voltages in TMS320C5510 DSP core processor

Voltages	Frequencies
1.1 V	6 MHz to 72 MHz
1.6 V	72 MHz to 200 MHz

The audio data moves from the codec A/D to memory, is processed, and moves back to the codec D/A in a pipelined fashion. The audio data is transferred back and forth between the AIC23 codec and memory using a *Multichannel Bidirectional Serial Port number 2* (McBSP2). The DMA transfers the data from the McBSP2 to the input buffer and from the output buffer to McBSP2. The McBSP2 is a serial port, which is capable of full duplex communication. The McBSP2 has double-buffered transmit and receive data registers, which allow continuous data stream. A DMA controller moves the entire frame, while CPU is processing the data. The DMA takes every 16-bit signed audio sample from the McBSP2 and stores it in a buffer in memory. In parallel, the highest frequency of the last 20ms sampled frame is computed and the audible data is stored in an intermediate buffer, which can be processed by the DSP core. Once the CPU core has processed the input data, it is sent back out through McBSP2 to the codec for output. DMA channel 0 is used for transmitting data to the codec and channel 1 is used to simultaneously receive data from the codec. The *Multichannel Bidirectional Serial Port number 1* (McBSP1) is used to control/configure the codec.

The constant sampling rate of the codec was set to f_{max} = 48 KHz with a frame size of 1024 samples because reconfiguring the codec takes more than the total time between samples. The downsampling was implemented by dividing the f_{max} by factors of 2. Therefore the effective sampling rates used were 48 KHz, 24 KHz, 12 KHz, 6 KHz, 3 KHz, and so on.

The hearing aid uses ping-pong buffering for transmitting and receiving data samples for both left and right ears. This technique is used to make data transfer seem stationary. There are two groups of buffers - “PING” buffer and “PONG” buffer for each of the following combinations- receive, transmit, left and right. Two for transmit left and transmit right channels and two for receive left and receive right channels. Totally there are eight buffers-4 “PING” and 4 “PONG”. The DMA controller is configured such that when data is transferred to or from the codec to the PING buffer, the CPU moves data out of the PONG buffer possibly processing it at the same time. At the end of the frames, the data transfer roles are reversed and the PING buffer is processed by the DSP while the PONG buffer is involved in data transfer and vice-versa. This system of using alternate buffers provides a processing window of time equal to an entire buffer size instead of a single sample time.

One of the major challenges in our implementation was that the frequency change in the hearing aid interferes with the transfer of data samples in the DMA. The workaround for this limitation is to break the transfer of each frame into two parts. During the first part, the DMA controller fills part of the current buffer the next frequency is computed. Then the DMA stops and the frequency is changed. During the remaining time data is removed from the buffer. The hearing aid computations are performed while the second part of the buffer is filled. After frequency scaling, the second state reconfigures the DMA and continues to fill up the rest of the empty buffer space from the point where it stopped previously.

The experimental setup consists of the TMS320C5510 starter kit, a PC sound card, a Tektronix current probe, a Tektronix TDS 3012B digital oscilloscope and the AM503B current probe amplifier. The PC sound card supplies audio inputs to the “line in" on the starter kit. The Tektronix current probe and AM503B current probe amplifier are connected to the digital oscilloscope and measures the current through the DSP supply pins and the voltage at the DSP supply pins. The power consumed by the DSP is computed using the oscilloscope multiply and RMS function. The audio inputs were a CD recording of music [29] and voice [1] data sets. The block diagram for the implementation is shown in Figure 16.

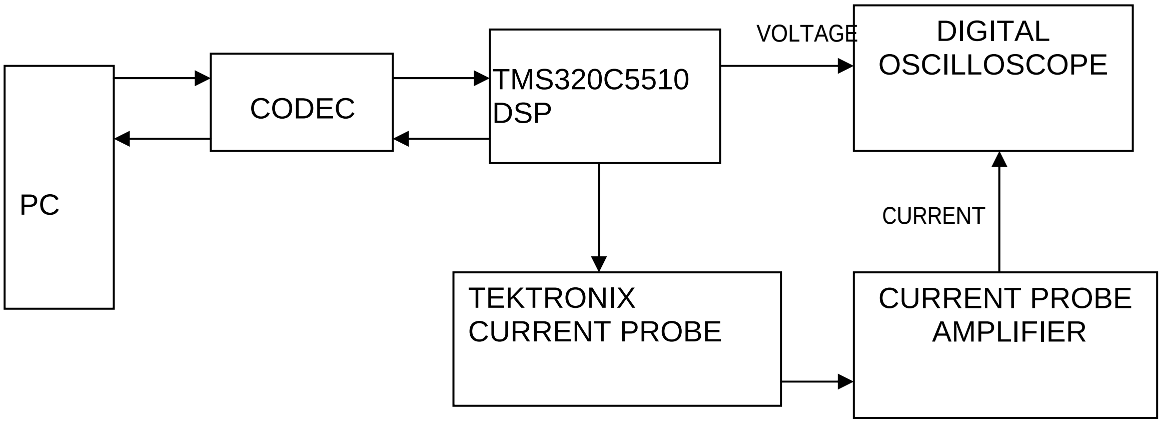

Figure 16: Block Diagram for the Implementation Setup

The filters used for processing the audio signals were a set of dummy filters with similar number of coefficients as those filters used in real hearing aids. This load varies from 1 to 50 filters having 50 to 200 coefficients to emulate the load for a frequency shaping filters [22]. Real hearing aids use adaptive noise reduction, interaural time delay, and multichannel amplitude compression [22], though they should have similar power consumption characteristics.

6.1 TIME DOMAIN IMPLEMENTATION

The frequency determination in time domain implementation is achieved through the use of a set of Finite Impulse Response (FIR) filters. These FIR filters function as bandpass filters for several frequency bands. The highest audible frequency is determined by calculating the cumulative power of the signals in each band and comparing them against the power under the ATH curve in the same frequency range. If the power in any higher frequency band is greater than the ATH power, then the highest audible frequency is set, otherwise the power comparison is done iteratively for lower frequency bands. In order to implement the FIR filter, we must select an optimum set of filter coefficients for each of the filters used. For a higher number of filter coefficients, the frequency response for the filter is very sharp and the filter efficiently removes signals outside the pass band. The execution time increases linearly with the number of filter coefficients. However, increasing execution time leads to more power consumption for the CPU. Ideally the resultant frequency response of all the FIR filters used for frequency determination should be equal to the ATH curve. Therefore there is a tradeoff between number of coefficients and execution time.

We also need to determine the ranges of cutoff frequencies for each of these FIR filters. In this case the range of the cutoff frequencies should be determined based on the critical bands as described in Section 2.2. The following section demonstrates the method used in determining the FIR filter coefficients.

6.1.1 SELECTION OF FIR COEFFICIENTS

FIR filters have several characteristics that make them ideal for DSP implementation.

1. Ease of implementation
2. Linear phase response, which helps prevent distortion.
3. Easy to represent as matrices of coefficients.

When a frame of samples is collected, the highest audible frequency will be determined by applying a cascade of time domain FIR filters. The first filter applied to the incoming audio frame is a high pass filter and the total power of the signals is compared against the ATH power in the high pass band to determine if there is any significant high frequency content present above the ATH curve. If the high frequency content is below the ATH, then a bandpass filter is applied and the power is checked for any frequency content over the ATH curve in the bandpass zone. If only a high pass filter were used instead of a bandpass filter, the number of coefficients would be fewer, but the energy of the filtered signals would include energy of both the high pass zone (>12 KHz) and the medium zone (5 KHz – 12 KHz). The ATH curve rises sharply near the extreme higher frequency end of the hearing range. Therefore the high frequency energy (>12 KHz) would be enough to trigger the medium band high pass filter, even though the high frequency component is inaudible and below the ATH curve.

The coefficients of these time domain FIR filters are determined on the cutoff frequencies. The cutoff frequencies of the filters are determined by concatenating the voice and music samples after sampling them at 48 KHz. A histogram was generated from the merged audio samples. The histogram is the sum of Figure 12 and Figure 13. After adding the cumulative area under the curve of the histogram, the total sample count for the merged audio sample was determined. The area under the curve was divided into equal portions of 3 bands. The boundary values of the 3 equal bands were set to be the cutoff frequencies of the FIR filters. The cutoff frequencies determined by the above method for the three filters are as follows:

- Low pass filter 5 KHz
- Band pass filter 5 KHz – 12 KHz
- High pass filter 12 KHz

Since there were not any samples above 16 KHz, an additional low pass filter with cut off frequency around 16 KHz should be applied. This can additionally serve as an anti-aliasing filter. The frequency response and the coefficients of the filter were determined using the Matlab filter design tool. The filters are of type equiripple because any other type of supported FIR filter needs more coefficients.

The number of coefficients for each filter was determined experimentally. We started by analyzing the response of the filter with very few numbers of coefficients because as mentioned in Section 6.1, the more the number of coefficients a filter has, the more computation overhead it will add. The magnitude responses for filters with fewer coefficients were not very sharp. The number of coefficients was increased to sharpen the falloff response until we got the desired magnitude response. This experiment was done for all the 3 filters - low pass, band pass and high pass. The numbers of coefficients for all the 3 types of filters, found experimentally, are in Table 6.

Table 6: Number of coefficients for each type of filter

Filter type	Number of coefficients
Low pass	20
Band pass	60
High pass	35

Figure 17 shows the response of a high pass filter with order 35. An ideal high pass filter would block all signals below the cutoff frequency but setting closer to ideal behavior requires more coefficients. We do not always require a perfect filter response. Once the signal is attenuated to a level below the ATH curve, it does not need to be suppressed further. Thus the filter response can have any shape that suppresses frequencies below the cutoff frequency to a level below the ATH curve. For lower orders of filter coefficient the magnitude response was not as sharp as the magnitude response of ATH curve. The filter order is adjusted depending on the magnitude response of the filters. The same approach is used in the determination of the response of the Bandpass and low pass filters.

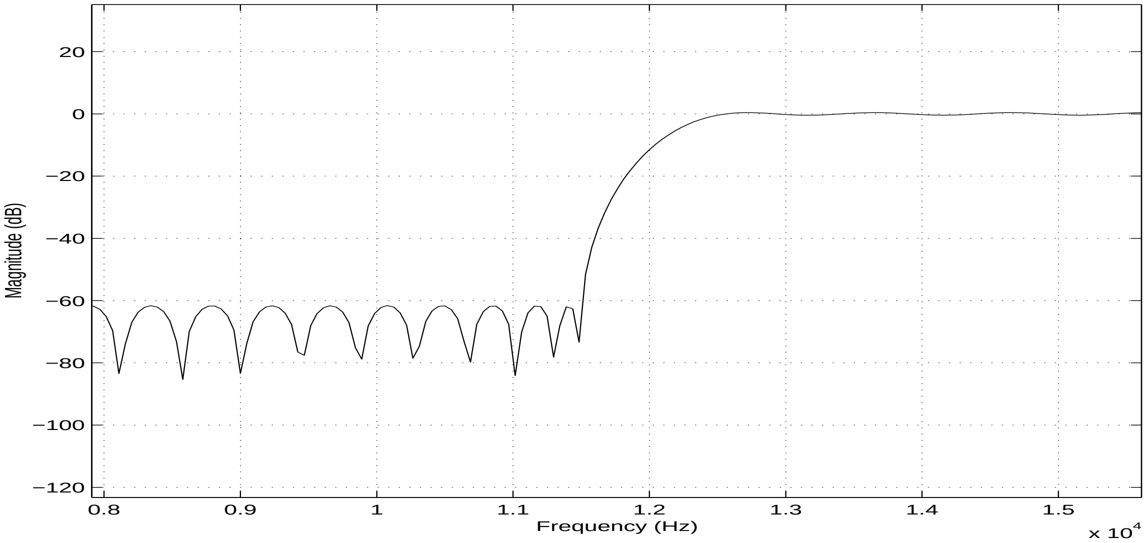

Figure 17: Magnitude response of high pass filter (order =35) with cutoff frequency =12 kHz

Figure 18 shows the frequency response of a low pass filter. In this case the filter order 20 is less since the response is not so steep in this case.

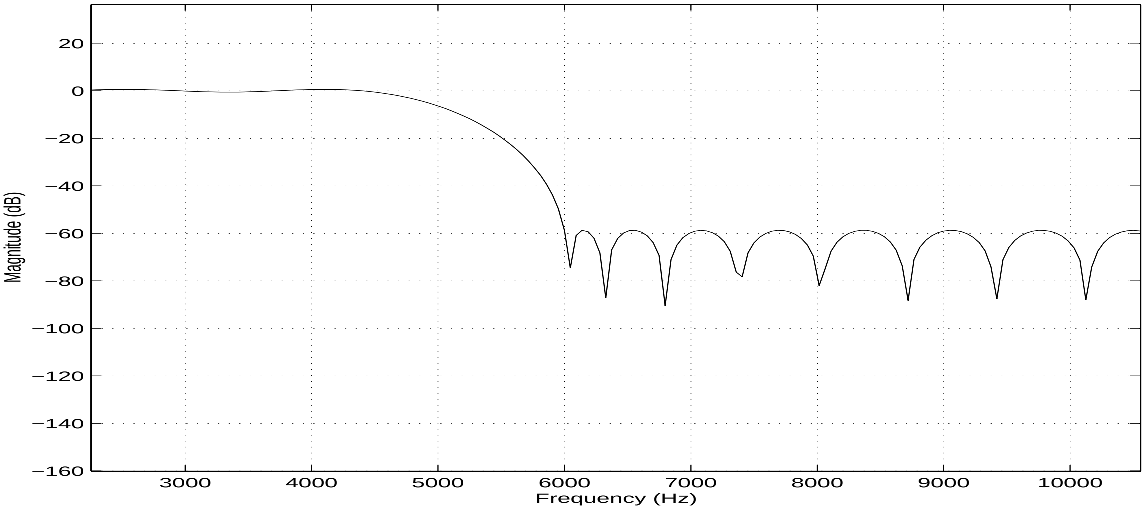

Figure 18: Magnitude response of a lowpass filter (order =20) with cutoff frequency nearing 5 kHz

Figure 19 shows the frequency response of a bandpass filter with pass band being frequency between 5 kHz to 12 kHz. In this case the order of the filter is 60 and is higher than the other two filters because we need to match the frequency responses of the curve at two ends of the band and not at just one end as for other filters.

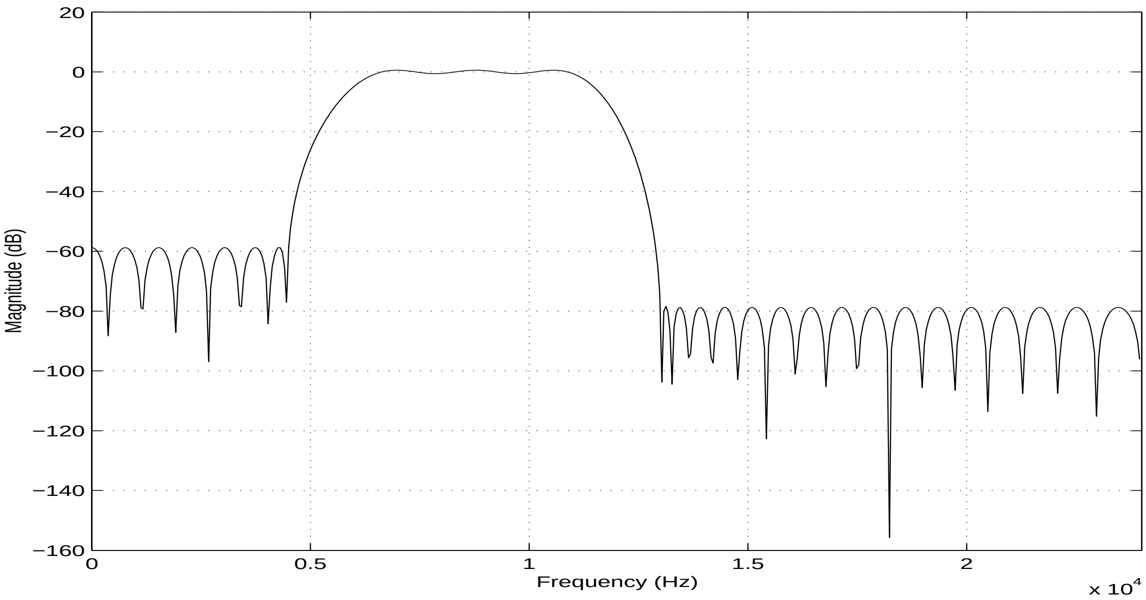

Figure 19: Magnitude response of a bandpass filter (order =60) with cutoff frequency with 5 kHz - 12 kHz

From the above analysis of frequency response curves, we determined the number of filters and their respective coefficients according to the computation load at each of the three sub-bands.

6.1.2 FREQUENCY DETERMINATION

Figure 20 shows the process by which the sampling rate is determined by time domain filters.

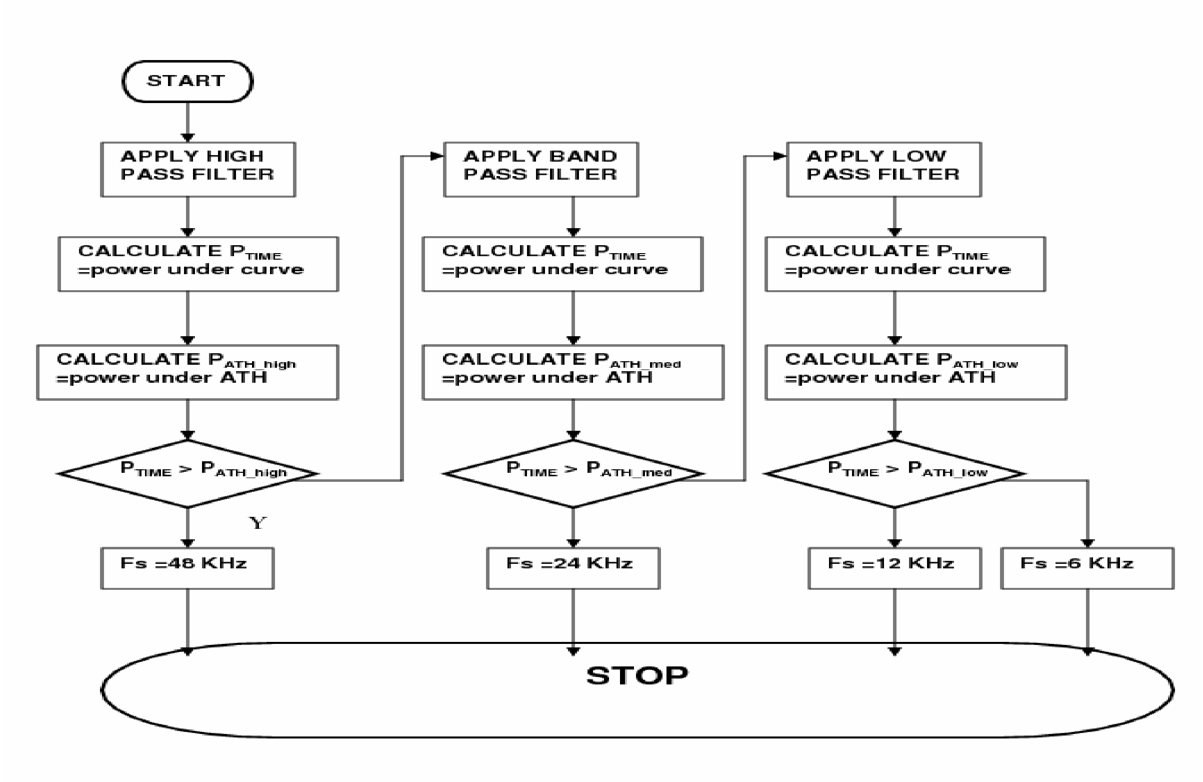
START
APPLY HIGH PASS FILTER
CALCULATE P_TIME =power under curve
CALCULATE P_ATH_high =power under ATH
P_TIME > P_ATH_high
Y
Fs =48 KHz
APPLY BAND PASS FILTER
CALCULATE P_TIME =power under curve
CALCULATE P_ATH_med =power under ATH
P_TIME > P_ATH_med
Fs =24 KHz
APPLY LOW PASS FILTER
CALCULATE P_TIME =power under curve
CALCULATE P_ATH_low =power under ATH
P_TIME > P_ATH_low
Fs =12 KHz
Fs =6 KHz
STOP

Figure 20: Flowchart showing f_{max} determination in Time Domain method

The frequency responses of the time domain filters are kept in such a way that it emulates the ATH as closely as possible. The frequency is determined in the following way. The signals are first passed through a high pass filter comprising the highest critical band. The power under that curve for that band is calculated as:

$$P = \sum_{i=1}^{n} \frac{1}{n} \bullet W_i^2 \qquad (13)$$

where W_i is the i^{th} filtered buffer values. This power is compared with the power under ATH curve in that region. If the total power of samples in that band is greater than the ATH power in that band, then the computation for highest frequency stops there and the sampling frequency is set to the nearest available frequency which should be approximately equal to double the highest frequency for that filter. If the power is less than ATH power of that band, then the power computation and comparison takes place for the band pass filter of the next lower band and this process is repeated until the power becomes greater than the ATH curve for that band.

The number of clock cycles that each of the stages takes has been summarized in Table 7. These number of clock cycles were determined from the TMS320C5510 DSP library reference documentation [35].

Table 7: Average clock cycles using time domain for maximum frequency determination

Operation	**Clock cycles**
Frequency determination using total number of FIRs	37000
CPU processing load/filter (208 coefficients)	110000
Post processing (upsampling of samples)	57000

The FIR filters used as CPU processing load have been coded in assembly instructions. These functions were supplied by library functions. This increases the execution speed of the CPU since it uses efficient pipelining which gives optimized code and low power throughput [35].

6.2 FREQUENCY DOMAIN IMPLEMENTATION

In the frequency domain applications, FFT operation was used to determine the maximum frequency content with audible PSD in the input audio signal. The input audio signal is broken up into frames, each containing 1024 samples. The FFT operation is used to determine the Power Spectral density of the signal for each frequency in a frame. The CPU checks for the highest frequency in the sampled frame with Power Spectral Density greater than the ATH curve value. After determination of the highest frequency, the sampling rate is set to the nearest value greater than twice the highest audible frequency content in that frame.

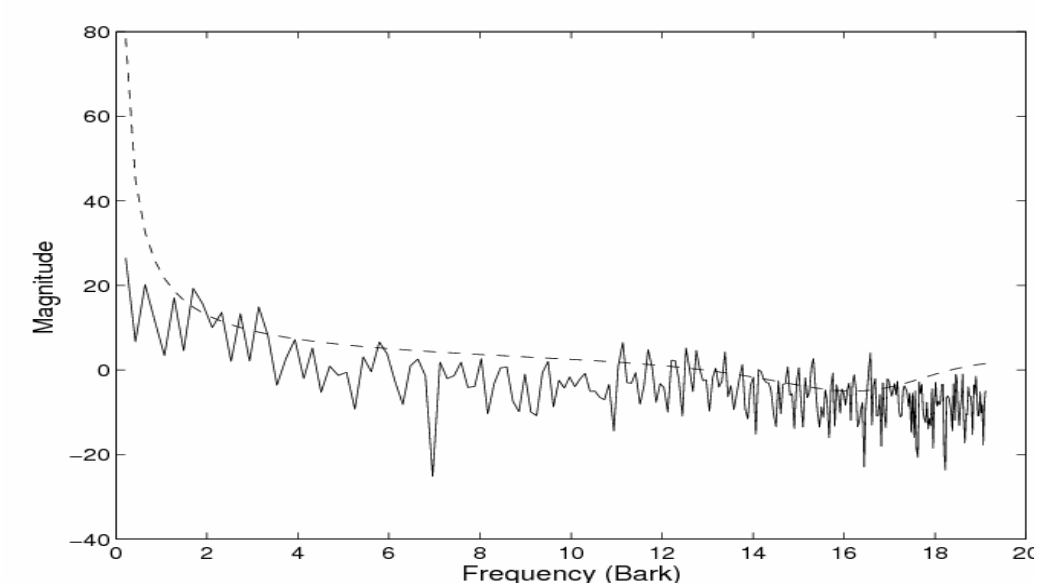

Figure 21: Graph showing f_{max} determination in Frequency Domain method

Figure 21 shows the frequency spectrum of a 20ms frame of audio. The dotted line is the ATH curve and any tone below the ATH curve is suppressed. The highest audible frequency where the tone is above the ATH curve is around 17 Barks (around 4500 Hz) and is determined as the f_{max} for this frame.

The TI TMS320C55x DSP library [35] supports the FFT function. The number of clock cycles that each of the stages took has been summarized in Table 8.

Table 8: Average clock cycles using frequency domain for maximum determination

Operation	**Clock cycles**
Frequency determination using FFT	64000
Constant CPU processing load (208 coefficients)	110000
Post processing (upsampling of samples)	57000

Table 8 shows the number of clock cycles required for FFT operation is more than the frequency determination using time domain FIR filters. In the time domain case the power determination is a cruder method to compress audio signals in order to determine the highest audible frequency in a frame while the FFT method is far more accurate method and therefore more performance intensive.

6.3 RESULTS

The power consumption in a standard hearing aid using both variable sampling rate and constant sampling rate is as shown in Figure 22 for music input. In this figure the power for both the normal and DSR methods is plotted against the number of filters the CPU had processed. In the DSR method, the sampling frequency for each frame was adjusted to twice the highest frequency determined for a frame and then rounded off to the next highest frequency the hardware can support. For the normal constant sampling rate method, the chip was run at a constant sampling rate such that the CPU processing time is within the frame time period.

From the graph, it can be seen that for fewer filters the power consumption for both normal and DSR method is almost equal because the hearing aid runs at a low frequency in both cases. Therefore the core voltage always remains at 1.1 V and never switches to 1.6 V. As the number of filters increase, the difference in power consumption is appreciable. In this case the normal hearing aid runs at 1.6 V almost all the time whereas the DSR hearing aid now switches voltage depending on the frequency content of the frame.

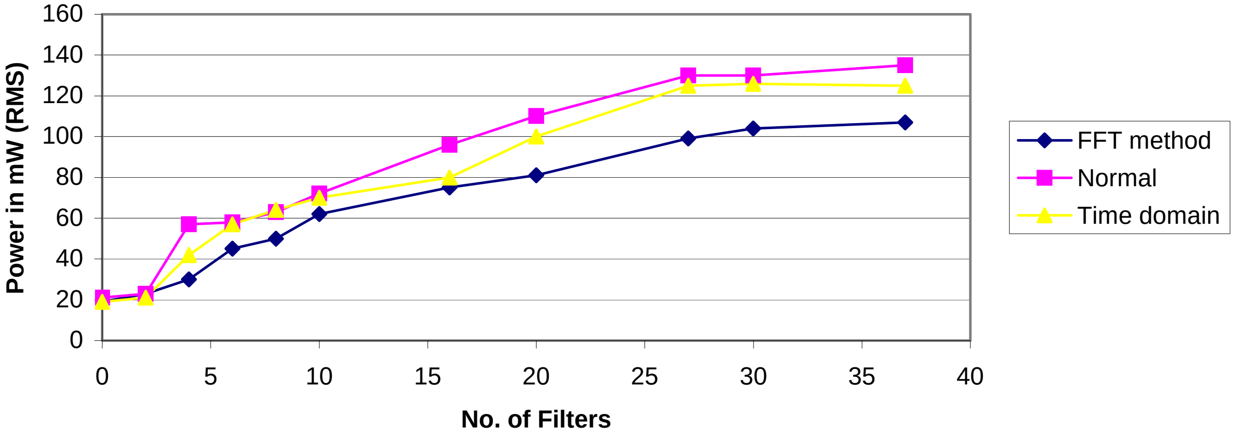

Figure 22: Power consumption vs. no. of filters for music samples

In Figure 23, the comparative power consumption for voice samples is shown. The power consumed in case of voice samples is much lower because most of the frames containing the voice inputs have lower frequency content than frames containing music inputs. The power consumption shown in both the figures will vary depending on the type of music or voice samples but the results shown are indicative of the difference in power consumption for music and voice samples.

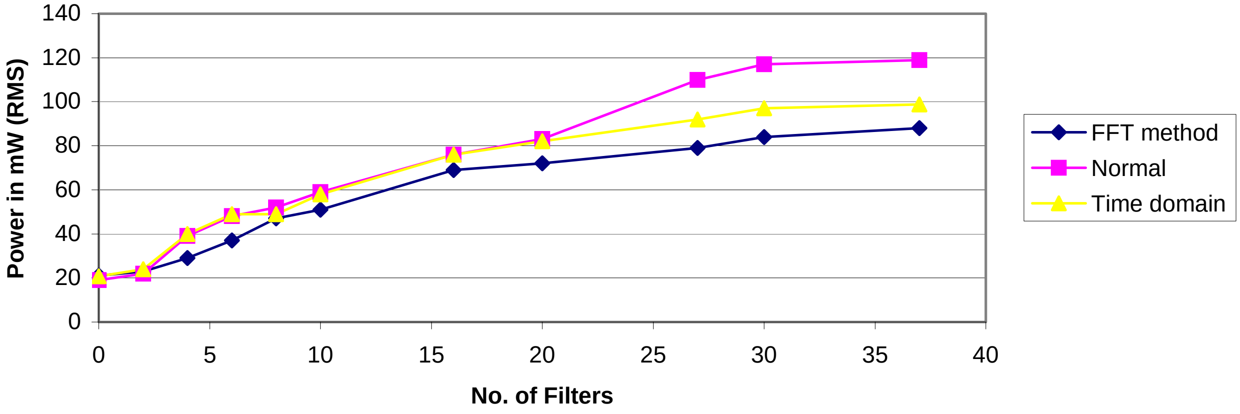

Figure 23: Power consumption vs. No. of filters for voice samples

From Figure 22 and Figure 23, it can be seen that the power saving is more for time domain method than for FFT method. It is quite contradictory from the data in Table 7 and Table 8 since the FIR filters ideally take fewer clock cycles to execute than FFT operations. The contradiction can be explained by the fact that in about 30% of cases, the FIR filters do not accurately measure the highest frequency in a frame as observed from the discrepancy from Figure 24 for music samples and Figure 25 for voice samples. In these figures it can be seen that the distribution of sampling rate determined from the time domain method is different from the distribution in the case of FFT method. In most cases the highest audible frequency determined is higher than actual value, therefore the sampling rate set to a higher value. Therefore in the case of time domain method, the CPU runs at higher clock frequency in most cases and hence the core voltage rarely drops from 1.6 V to 1.1 V. In the case of FFT, the frequency determined is accurate and since the frequency content is in low or medium frequency bands in most of the cases, therefore the DSR hearing aid runs at 1.1 V more frequently than at 1.6 V. This leads to greater power savings. The difference is marked in the case of the music stream where the high frequency component is predominant.

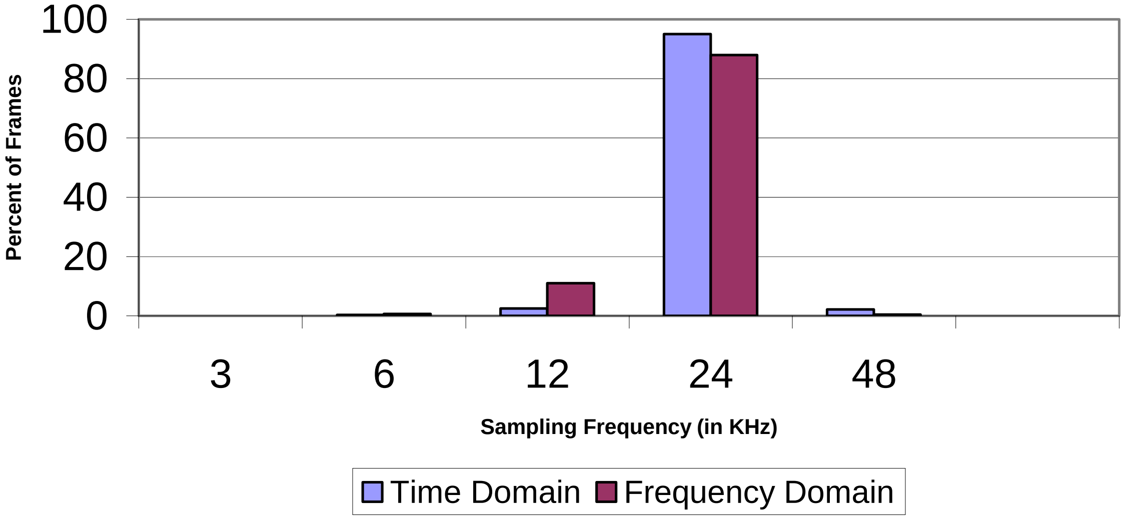

Figure 24: percentage of samples vs. set sampling frequency for music

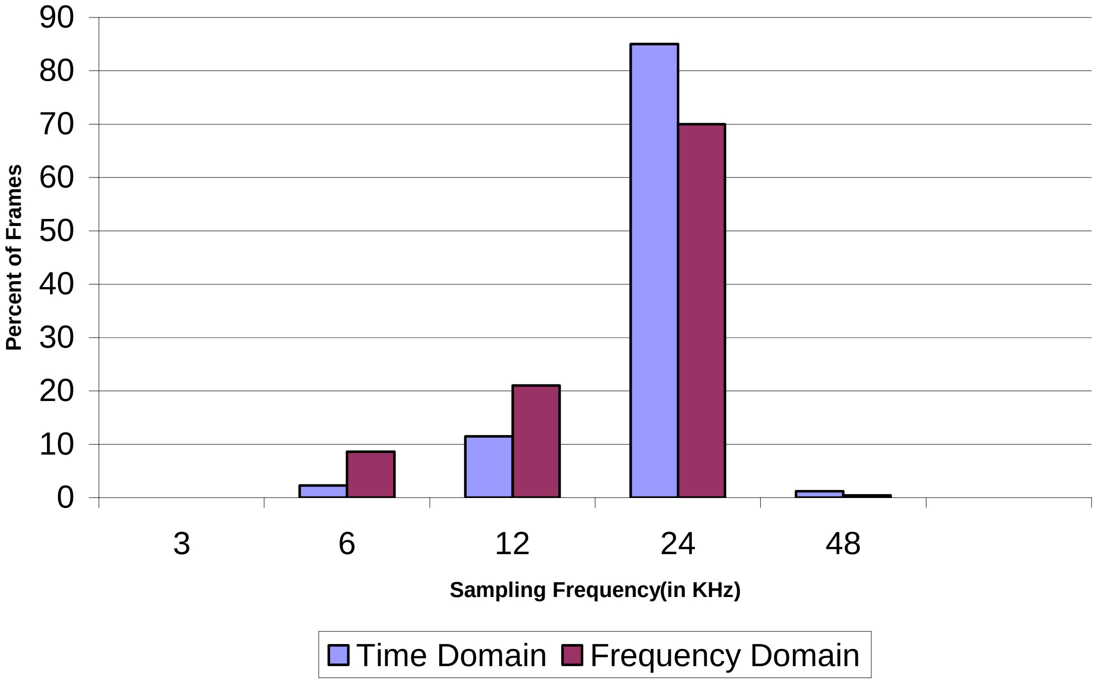

Figure 25: percentage of samples vs. set sampling frequency for voice

Therefore the major advantage of the Frequency domain FFT method is that there is marked improvement in the power consumption with respect to the time domain method but at the cost of higher overhead in clock cycles for FFT operation. It can also been seen that when the number of load filters is 0, the power consumed by normal non-DSR method is around 19mW and for both time-domain DSR and frequency-domain DSR, the power consumed is about 21mW. In this case the load is the high pass anti-aliasing FIR filter which is required to remove the high frequency components from the signal.

Comparing Figure 14 and Figure 22 for music samples and Figure 15 and Figure 23 for voice samples, we observe that the implementation results are different from theoretical power consumption. The first discrepancy is the mismatch between the power consumption without using DSR. The theoretical calculations are higher compared to implementation results. This can be attributed to the fact that we use worst-case values for theoretical calculations where the values are higher than normal. Another discrepancy is that in case of DSR power consumption results, the implementation values are higher than theoretical results. This is because, in case of DSR, a major portion of the power is used up during frequency and voltage switching, which is not accounted for in the theoretical results. For example, switching from 1.1V to 1.6V takes about 10 - 15mW.

6.4 HARDWARE LIMITATIONS

The TMS320C5510 DSP starter kit has several hardware limitations. The first limitation is related to the transfer of data by the DMA controller. In the case of the constant sampling rate method, the frequency remains the same and the DMA controller can work in parallel with the DSP processing of frames. Therefore the rate at which the data is transferred is equal to the rate at which the DSP processes the data. In the case of dynamic sampling rate, the DMA transfers data at a different rate than the DSP processes data. There will be no problem if the frequency switching takes place within the frame time period (around 20 ms in this case) or if the DSP can independently work while the frequency is being changed. But according to experimentation the frequency locking by the *Phase Locked Loop*s (PLL) takes between 20 us to 80 us. If there is a change in voltage along with frequency switching it takes on the order of 500 ms to 2 ms for the system to stabilize. Therefore the time required to stabilize the PLL is more than the time between two samples. This causes some of the audio samples to be dropped causing deterioration in the signal quality.

The frequency switching interferes with the transfer of samples in the DMA from McBSP2 to the buffer or vise-versa. Therefore the DMA needs to stop transferring data while frequency or voltage changes. The library functions are modified such that the ping-pong buffers will break into two states. In the first state the f_{max} is determined while in the second state the clock frequency is switched and the samples are processed. Thus the voltage scaling is made independent of the transfer of buffers such that the DMA continues data transfer during voltage change. This makes the DMA data transfer dependent only on frequency scaling. The DVS function is executed only when the current core voltage is 1.1 V and the computed clock frequency is greater than 72 MHz or when the current core voltage is 1.6 V and the clock frequency becomes less then 72 MHz.

However the problems could be easily solved if the hardware could support a frequency divider instead of a PLL so that the frequency switching can take less time to switch speeds. Also another PLL could be added such that while one PLL acts as clock to the CPU the other can sync to the switched frequency and the CPU can be switched between these two PLLs depending on a multiplexed input as shown in Figure 26.

The TMS320C5510 DSP supports only two levels of voltage 1.1 V and 1.6 V. The DSR method will show better results for a wider range of voltage supporting different frequencies.

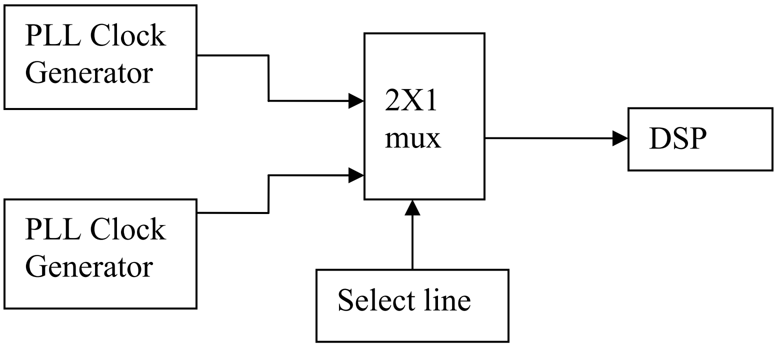

Figure 26: Two-PLL model for separate clock inputs

For a wide range of voltage the power consumed by samples in the intermediate frequency bands will be lesser. This method has a distinct advantage over the constant sampling rate method in case of low frequency content frames where it operates at a lower voltage. Ideally, each sampling rate should have its own voltage, which runs with a fixed set of frequencies.

6.5 AUDIO QUALITY TESTING

In case of frame-based DSR method, if any of the deadlines for the samples are missed, it might cause buffer overflow in the hardware, which will in turn deteriorate the signal quality. It is also possible that errors could cause noise. Therefore it is important to test the quality of the audio that is produced from the test setup. The audio quality was tested two ways:

1. Taking the frequency spectrum of the input and output signal and then subtracting one from the other without any amplification. The resultant spectrum is noise signal caused due to various steps of processing like frequency determination, filtering, upsampling and downsampling. The noise signal should be below the ATH curve showing that the noise added is imperceptible.
2. Using human ear to compare the output without any processing and the output with processing. In this case, the input was a PC sound card and the output was a PC speaker. A continuous stream of music was played to discern any perceptible difference in the audio quality.

In the first case, we use Matlab to read the input and the output signals, carry the subtraction operation and then plot the frequency spectrum of the resultant noise signal. It was verified that the noise introduced in the output signal after subtracting the input signal from the output signal is well below the ATH curve. This test was carried out for both music and voice samples using both time domain method and frequency domain method of frequency determination. For both the time domain and frequency domain method it was ensured that the FIR filters and the FFT operations respectively do not contribute to any noise over the ATH curve in the system.

In the second case, we made an informal quality check on the output music and voice samples from the PC speaker to make sure that there was no perceived quality deterioration compared to the original sound. Two people listened to the output and could not distinguish between the original signal and the DSR output.

CHAPTER 7

CONCLUSION

The DSR technique for an audio application involves dynamically changing the sampling rate in response to the input audio signal. The variations in the input audio are of types - high frequency e.g. music and low frequency e.g. voice. The sampling rate is set based on the highest audible frequency content in an audio frame. This highest frequency has been determined by using two methods: time domain method and frequency domain method. The sampling rate provides opportunities to vary frequency and voltage and thereby decrease power consumption.

In our implementation, the hearing aid prototype using DSR method gives us power savings of up to 26% for Time domain method and 40% for FFT method of its normal power consumption i.e. without using DSR. There is no perceptible output quality degradation. The power consumption for this prototype is based on the filter processing load and the profile of the input signal. If the number of coefficients is higher for a filter, the power consumed is more. Also, in case of music samples the power consumed is more because of higher number of high frequency frames. For higher frequency content frames, the sampling frequency set is high. According to the simulation calculations, the maximum percentage of power saving possible is around 65%. Improvement in the power numbers can be achieved by supporting more voltage options for different frequencies and by reducing the settling time for the frequency and voltage scaling method. This method may apply to any other portable DSP application where battery life is critical.

7.1 FUTURE WORK

In order to determine the highest frequency content of a frame, the compression method used in our method is removal of audio bits, which are otherwise inaudible to a normal human ear and therefore below the ATH curve. There are a lot of other audio compression methods like simultaneous masking, temporal masking, tone masking, noise masking [20], which can help in the determination of a global threshold of hearing. Some of these compression methodologies can be complex and can take more computation clock cycles. This ultimately leads to more power consumption on the part of the DSP. On the other hand the compression methods can reduce the number of samples to be processed in the later stages. Therefore extensive research needs to be performed to determine which of the audio compression methods should be used to get maximum power savings. On average the time required for determining the sampling frequency should be significantly lower than the CPU processing time, otherwise the effectiveness of the technique decreases.

The technique for reducing power using dynamic sampling rate and DVS principles can be used for other DSP applications, which have a repeatable set of inputs. As discussed earlier, this methodology will be very effective for applications where a large percentage of the power savings can be obtained by reducing the number of samples. Also, we can have adaptive filtering based on the sampling rate. There are load filters which may consume less power at particular operating frequencies depending on the hearing aid functions they may be performing. An extensive analysis on the audio algorithms and the coefficients in the filters will give us a better understanding of the scope of power reduction techniques in such audio devices.

There are many interpolation methods, which can be used during upsampling after the CPU has processed the samples [20] that produce superior quality output sound. In this case too, a judicious decision needs to be made on the choice of interpolation algorithms based on complexity of the algorithm and the output sound quality.

CHAPTER 8

APPENDICES

8.1 MATLAB CODE

Main.m

```
clc;
clear;

%s gives the intensity of the sound,fs-sampling frequency
% default sampling rate=48khz //depends on sampling rate of the sound file

[s, Fs,nbits] = wavread('OyeComoVa.wav');

frame_strt=0;       % starting index of a frame
low_freq=0;         %no of times the audio signal goes in low frequency zone
mid_freq=0;         %no of times the audio signal goes in mid frequency zone
high_freq=0;        %no of times the audio signal goes in high frequency zone

point=1024;%FFT points(there will be 513 points between 0khz to 24khz)

%process till end-of-file, whichever comes first
frame_size = 20*Fs*10^-3;
frame_size = point;
hist = zeros(floor(frame_size/2)+1,1);
while( (frame_strt + frame_size) < size(s,1) )
    % extract out a frame of 20 ms
    frame=s(frame_strt + 1 : frame_strt + frame_size);

    psd_val=(abs(fft(frame,point))).^2;
    for psd_size=1:1:1024
       if(psd_val(psd_size)==0.0000)
           Pxx(psd_size)=powernormconst;
       else
          % extract out PSD for particular frequecies
          Pxx(psd_size)=powernormconst + 10.*log10(psd_val(psd_size));
       end
   end

    Pxx1=Pxx(1:512);% extract out PSD for particular frequecies

    audible = Pxx1 >= ath(F);
    freq_content(i)=F(max_freq);
    i=i+1;
    for max_freq = size(F):-1:1
            if (audible(max_freq))
                break
            end
    end
    hist(max_freq) = hist(max_freq) + 1;

    disp(sprintf('max freq = %f', F(max_freq)));

    if  (F(max_freq) < Fs/8)
        low_freq=low_freq+1;
    elseif (F(max_freq) <Fs/4)
         mid_freq=mid_freq+1;
    else
         high_freq=high_freq +1 ;
    end

   frame_strt = frame_strt + frame_size;         %increment to new frame no.
end

disp('Frequency density');

total = low_freq + mid_freq + high_freq;
disp(sprintf('of %d samples %f %% were low, %f %% were mid-range, and %f %% were high
frequency\n', total, 100*low_freq/total, 100*mid_freq/total, 100*high_freq/total));

int_content=round(freq_content);
sat_un = unique(freq_content);
for val = 1 : length(sat_un)
fprintf('Value %d, Occurences %d\n', sat_un(val), ...
sum(freq_content== sat_un(val)));
occur(val)=sum(freq_content== sat_un(val));
end

bar(sat_un,occur)
```

ath.m

```
function a = ath(f)
% ATH  Audible threshold level.  Return the audible threshold at a particular
%      frequency.
%
a  = 3.64*((f/1000).^-0.8) - 6.5*exp(-0.6*((f/1000)-3.3).^2) + 10^-3*(f/1000).^4;
```

8.2   C CODE

Hearing_Aid.c

```
#include "hearing_aid.h"
#include "processing.h"

/*  The 5510 DSK Board Support Library is divided into several modules, each of which has its
own include file.  The file dsk5510.h must be included in every program that uses the BSL. This
example also uses the DIP, LED and AIC23 modules.  */

#include "dsk5510.h"
#include "dsk5510_led.h"
```

```
#include "dsk5510_dip.h"
#include "dsk5510_aic23.h"
#include "psl.h"
#include "ath.h"            //ATH array values imported from ath.m
#include “filter_low.h”    //low pass filter coefficients values imported from MATLAB filter
toolbox
#include “filter_med.h”    //band pass filter coefficients values imported from MATLAB filter
toolbox
#include “filter_high.h”   //high pass filter coefficients values imported from MATLAB filter
toolbox

/* This program uses Code Composer's Chip Support Library to access C55x peripheral registers
and interrupt setup.  The following include files are required for the CSL modules.  */

#include <csl.h>
#include <csl_irq.h>
#include <csl_dma.h>
#include <csl_mcbsp.h>
#include <csl_pwr.h>
#include <csl_icache.h>
#include <csl_pll.h>
#include <dsplib.h>

/* Function prototypes */
void initIrq(void);
void initDma(void);
void copyData(Int16 *inbuf, Int16 *outbuf, Int16 length);
void setDMAdata_addr(int);
void DVS(void);
void downsamples(void);
void upsamples(void);
void processBuffer(void);
void switchfrequency(void);
long Maxfrequency(void);
void computenextfrequency(void);
void dmaHwi(void);

/* Constants for the buffered ping-pong transfer */
#define BUFFSIZE         1024
#define BUFFOFFSETinit           21
#define PING            0x00
#define PONG           0x02
#define LEFT              0x00
#define RIGHT 0x01

/* Initial it for the 200MHz operate */
Uint16 BUFFOFFSET =BUFFOFFSETinit;

/* Ping pong state variable */
Int16 pingPong;
```

```
/* power mode indicator */
int highpower = 1;

/* frequency switch indicator */
int changefrequency = 0;

/* upsampling last samples */
Int16 lastLs = 0;
Int16 lastRs = 0;

/* dynamic sampling frequency scaling rate */
int step;

Int16 switch3;

/* PSL data */
#define SETPOINT72MHZ       5
#define SETPOINT200MHZ      15
#define INITSETPOINT        SETPOINT200MHZ
#define ORDER          208
#define samp_freq           48000

PSL_Setpoint setPoint;
PSL_Setpoint prevSetPoint = 15;
PSL_ClkID clk = PSL_CPU_CLK;

/* Clock cycle computation data */
#define other_CC            30
#define UPDN_sampling       60
#define RTDX_OH                     1000

int timeleft = 0;

// speeds available in MHz, index == setpoint
long gSpeedTbl[] = {
        6, 12, 24, 48, 60, 72, 84,
        96, 108, 120, 132, 144, 156, 168, 180, 200
};

// gFFT must be 2x as big as sample buffers to accomodate complex data
#pragma DATA_SECTION(gFFT, ".input");
Int16 gFFT[BUFFSIZE*2];
int step;

/* Codec configuration settings */
DSK5510_AIC23_Config config = { \
   0x0017,  /* 0 DSK5510_AIC23_LEFTINVOL  Left line input channel volume */ \
   0x0017,  /* 1 DSK5510_AIC23_RIGHTINVOL Right line input channel volume */\
   0x01f9,  /* 2 DSK5510_AIC23_LEFTHPVOL  Left channel headphone volume */  \
   0x01f9,  /* 3 DSK5510_AIC23_RIGHTHPVOL Right channel headphone volume */ \
   0x0010,  /* 4 DSK5510_AIC23_ANAPATH    Analog audio path control */      \
   0x0000,  /* 5 DSK5510_AIC23_DIGPATH    Digital audio path control */     \
   0x0002,  /* 6 DSK5510_AIC23_POWERDOWN  Power down control */             \
   0x0043,  /* 7 DSK5510_AIC23_DIGIF      Digital audio interface format */ \
   0x0081,  /* 8 DSK5510_AIC23_SAMPLERATE Sample rate control */            \
   0x0001   /* 9 DSK5510_AIC23_DIGACT     Digital interface activation */   \
};

/*
 * Data buffer declarations
 * gBufferRcv[PONG|RIGHT] selects the right, pong recieve buffer, or
 * gBufferXmt[PING|LEFT] selects the left, ping transmit buffer.
 */
Int16 gBufferRcv[4][BUFFSIZE];  // Top of receive buffer
Int16 gBufferXmt[4][BUFFSIZE];  // Top of transmit buffer

/* Event IDs, global so they can be set in initIrq() and used everywhere */
Uint16 eventIdRcv;
Uint16 eventIdXmt;

/* initIrq() - Initialize and enable the DMA receive interrupt using the CSL. The interrupt service
routine for this interrupt is hwiDma. The interrupt enable and flag bits of this interrupt is bit 9 of
the DSP's IER0 and IFR0 registers. The transmit interrupt is configured but not enabled so the
program can detect when a block has been fully transmitted. */
void initIrq(void)
{
 // Get Event ID associated with DMA channel interrupt.  Event IDs are a CSL //abstraction that
lets code describe a logical event that gets mapped to a real physical //event at run time.  This
helps to improve code portability.
   eventIdRcv = DMA_getEventId(hDmaRcv);
   eventIdXmt = DMA_getEventId(hDmaXmt);

   // Clear any pending receive channel interrupts (IFR)
   IRQ_clear(eventIdRcv);
   IRQ_clear(eventIdXmt);

   // Enable receive DMA interrupt (IMR)
   IRQ_enable(eventIdRcv);

   // Make sure global interrupts are enabled
   IRQ_globalEnable();
}


/* initDma()- Initialize the DMA controller. The actual DMA register            configuration is
done in the DSP/BIOS configuration under Chip Support Library --> DMA --> DMA
Configuration Manager and loaded at run time in the auto-generated file
dsk_app1cfg_c.  initDma() initializes some registers not normally set in the DSP/BIOS config
like CEI and CFI.  It also sets frame count based on BUFFSIZE.  */

void initDma(void)
{
   volatile Int16 i;

   // Set indices and lengths for receive channel sorting.
   DMA_RSETH(hDmaRcv, DMACEI, (2*BUFFSIZE) - 1);
   DMA_RSETH(hDmaRcv, DMACFI, -((2*BUFFSIZE) - 1));
   DMA_RSETH(hDmaRcv, DMACFN, BUFFSIZE);

   // Set indices for transfer channel unsorting
   DMA_RSETH(hDmaXmt, DMACEI, (2*BUFFSIZE) - 1);
   DMA_RSETH(hDmaXmt, DMACFI, -((2*BUFFSIZE) - 1));
   DMA_RSETH(hDmaXmt, DMACFN, BUFFSIZE);

   // Clear the DMA status registers to receive new interrupts
   i = DMA_RGETH(hDmaRcv, DMACSR);
   i = DMA_RGETH(hDmaXmt, DMACSR);
}

/* copyData() - Copy one buffer with length elements to another.  */
void copyData(Int16 *inbuf, Int16 *outbuf, Int16 length)
{
   Int16 i = 0;

   for (i = 0; i < length; i++) {
      outbuf[i]  = inbuf[i];
   }
}

/* SetDMA () -  Configure the DMA data memory location. Set the receive and transmit buffer
destination addresses */

void setDMAdata_addr(int secondStates)
{
        Uint32 addr;


        if(switch3){
                // Configure the receive channel for second state input data
        addr = ((Uint32)gBufferRcv[pingPong])<<1;
        DMA_RSETH(hDmaRcv, DMACDSAL, addr & 0xffff);
        DMA_RSETH(hDmaRcv, DMACDSAU, (addr >> 16) & 0xffff);
        DMA_RSETH(hDmaRcv, DMACFN, BUFFSIZE);

        // Configure the transmit channel for second state output data
        addr = ((Uint32)gBufferXmt[pingPong])<<1;
        DMA_RSETH(hDmaXmt, DMACSSAL, addr & 0xffff);
        DMA_RSETH(hDmaXmt, DMACSSAU, (addr >> 16) & 0xffff);
        DMA_RSETH(hDmaXmt, DMACFN, BUFFSIZE);
   }
   else{
                if(!secondStates){
                        // Configure the transmit channel for first state input data
                addr = ((Uint32)gBufferRcv[pingPong])<<1;
                DMA_RSETH(hDmaRcv, DMACDSAL, addr & 0xffff);
                DMA_RSETH(hDmaRcv, DMACDSAU, (addr >> 16) & 0xffff);
                //reset the receive channel buffer size
                        DMA_RSETH(hDmaRcv, DMACFN, BUFFOFFSET);

                // Configure the transmit channel for first state output data
                addr = ((Uint32)gBufferXmt[pingPong])<<1;
                DMA_RSETH(hDmaXmt, DMACSSAL, addr & 0xffff);
                DMA_RSETH(hDmaXmt, DMACSSAU, (addr >> 16) & 0xffff);
                        //reset the transmit channel buffer size
                DMA_RSETH(hDmaXmt, DMACFN, BUFFOFFSET);
        }
        else{
                // Configure the receive channel for second state input data
                addr = ((Uint32)gBufferRcv[pingPong]+BUFFOFFSET)<<1;
                        DMA_RSETH(hDmaRcv, DMACDSAL, addr & 0xffff);
                DMA_RSETH(hDmaRcv, DMACDSAU, (addr >> 16) & 0xffff);
                DMA_RSETH(hDmaRcv, DMACFN, BUFFSIZE-BUFFOFFSET);

                // Configure the transmit channel for second state output data
                addr = ((Uint32)gBufferXmt[pingPong]+BUFFOFFSET)<<1;
                DMA_RSETH(hDmaXmt, DMACSSAL, addr & 0xffff);
                DMA_RSETH(hDmaXmt, DMACSSAU, (addr >> 16) & 0xffff);
                DMA_RSETH(hDmaXmt, DMACFN, BUFFSIZE-BUFFOFFSET);
                }
         }

}

/*  DVS() -             Dynamic Voltage Scaling
 *                        It will scale the voltage down to 1.1 V if the current
 *                        frequency is less of equal to 72 MHz. And it will scale the
 *                        voltage back to the 1.6 V before the frequency get higher than
 *                        72 MHz. highpower is a variable to indicate the current power state
 *                        when the highpower = 1, it means 1.6V, otherwise it will be 1.1 V.
 *                        Some delay were added, when the voltage switch from 1.1V to
 *                        1.6V in order to avoid the interference to the frequency switch.
 *                        The frequency switch will happen right after the voltage scaling
 *                        Thus it need make sure that the core voltage is stable before
 *                        running  the switch frequency.
 */
void DVS()
{

        if(highpower){
                if(setPoint<6){
                        PSL_gpioVoltRegScale_DSK5510(1.6,1.1,72000,0);
                        highpower=0;
                }
        }
        else{
                if(setPoint>5){
                        PSL_gpioVoltRegScale_DSK5510(1.1,1.6,gSpeedTbl[prevSetPoint],1);
                        highpower=1;
                }
        }
}

/*
 *  downsamples -        It will reduce the number of samples to be process in order to
 *                       implement the dynamic sampling frequency scaling. It will ensure
 *                       that all the samples will still have the same time interval between
 *                       samples after the number of samples reduction. It will reduce the
 *                       samples base on the rate given by the computefrequency process
 *                       which is step. if step = 1,it mean the samples size shift letf by 1,
 *                       shift then half of the samples will be reduce. if step = 2, it mean the
 *                       samples size shift letf by 2 then the number samples will be reduce
 *                       to the orginal size divided by four. It will take the samples from the
 *                       gBufferRcv. Drop the samples alternately base on the rate if
 *                       nessacery and put it back to gBufferRcv. samples[i-1] =
 *                       samples[i*(div)-1]where div=1<<step, and i = 1 to (samples
 *                       size>>step)
 */
void downsamples()
{
        int j;
        int i;
        int div;

        div=(1<<step);
        if(div>1)
                for(i = div-1, j = 0; i < BUFFSIZE ; i += div, j++) {
                        gBufferRcv[pingPong|LEFT][j]  = gBufferRcv[pingPong|LEFT][i];

        gBufferRcv[pingPong|RIGHT][j]=gBufferRcv[pingPong|RIGHT][i];
                }
}

/*
 *      upsamples -      The upsamples process will ensure that D/A converter will receive
 *                       the same number of samples produced by the A/D converter, since
 *                       both of the converters have the same sampling rate. It will
 *                       responsible to bring back the number of samples back to the
 *                       original number of samples. Upsamples predict the value between
 *                       two samples in order to increase the number of samples by assume
 *                       the rate of change is consistent. This method is called linear
 *                       interpolation. it will increase the number of output result
 *                       which is store in the gBufferXmt, base on the step computed from
 *                       the computenextfrequency function. After increment number of
 *                       output result, they will be stored back to the gBufferXmt array.
 *                       lastLs and lastRs are variables where they store the last left and
 *                              right signal as reference for next upsamples process in next frame
 *                              if needed.
 */
void upsamples()
{
          int j;
          int i;
          int k;
          int div;
          int size = (BUFFSIZE>>step)-1;

          div=(1<<step);

          for(i = BUFFSIZE-1, j = size; i > div ; i -= div, j--) {
                    for(k = 0 ; k < div ; k++) {
                              gBufferXmt[pingPong|LEFT][i-k] = (((div-
k)*gBufferXmt[pingPong|LEFT][j] + k*gBufferXmt[pingPong|LEFT][j-1])) >> step;
                              gBufferXmt[pingPong|RIGHT][i-k] = (((div-
k)*gBufferXmt[pingPong|RIGHT][j] + k*gBufferXmt[pingPong|RIGHT][j-1])) >> step;
                    }
          }

          for(k = 0 ; k < div ; k++) {
                    gBufferXmt[pingPong|LEFT][div-k]  = ((k*lastLs + (div-
k)*gBufferXmt[pingPong|LEFT][0])) >> step;
                    gBufferXmt[pingPong|RIGHT][div-k] = ((k*lastRs + (div-
k)*gBufferXmt[pingPong|RIGHT][0])) >> step;
          }

          lastLs = gBufferXmt[pingPong|LEFT][BUFFSIZE-1];
          lastRs = gBufferXmt[pingPong|RIGHT][BUFFSIZE-1];
}

/*
 *  processBuffer() - Process audio data once it has been received, then
 *                  set the DMA configuration registers up for the next
 *                  transfer.  If DIP switch #3 is up, the audio passes
 *                  straight through.  If DIP switch #3 is down, this will
 *                    be in the power reduction by varying the dynamic
 *                    sampling rate mode
 *
 */

void processBuffer(void)
{
   Int16 switch3;

   // Wait until transmit DMA is finished too
   while(!IRQ_test(eventIdXmt));

          // Determine which ping-pong state we're in
```

```
    // Toggle LED #3 as a visual cue
    DSK5510_LED_toggle(pingPong ? 3 : 2);

    // Read DIP switch 3
    switch3 = DSK5510_DIP_get(3);
    if (switch3) {
                        // Switch 3 is up, normal hearing aid
            hearingAid(gBufferRcv[pingPong|LEFT], gBufferRcv[pingPong|RIGHT],
                            gBufferXmt[pingPong|LEFT],gBufferXmt[pingPong|RIGHT],
                            BUFFSIZE);
    } else {
            // Switch 3 is down, DVS hearing aid
            downsamples();
            LOG_printf(&logTrace,"DVS");
            hearingAid(gBufferRcv[pingPong|LEFT], gBufferRcv[pingPong|RIGHT],
                            gBufferXmt[pingPong|LEFT],gBufferXmt[pingPong|RIGHT],
                            BUFFSIZE>>step);
            upsamples();
    }
}

/*
 *      switchfrequency -       switch to a new frequency according the frequency
 *                              computed by computenextfrequency function. Stop the
 *                              DMA data transfering before the frequency change and
 *                              resume the DMA when the new clock frequency
 *                              is stable. Set the new BUFFOFFSET value, which is the
 *                              execution time for for the first state of each frame.
 *                              BUFFOFFSET has to be big enough, so that        all the thread
 *                              in the first state will met their deadline in time.
 *                              BUFFOFFSET value will be vary according to the clock
 *                              frequency.
 */

void switchfrequency(void)
{
        PSL_Status      status;
        volatile DSK5510_AIC23_CodecHandle hCodec;


        //send a message to message log to indicate frequency switch
        //LOG_printf(&logTrace, "switch frequency");

        //Puase DMA for frequency switch
        DSK5510_LED_on(0);
        DMA_stop(hDmaRcv);
        DMA_stop(hDmaXmt);

        status = PSL_changeSetpoints(1, &clk, &setPoint,
                                                                TRUE, TRUE, NULL,
NULL);
```

```
        //Restort DMA
        DMA_start(hDmaRcv);
        DMA_start(hDmaXmt);
        DSK5510_LED_off(0);

        DVS();


  /*Set a new Buffer size for the first state due to the
   *change of the frequency
   */
        BUFFOFFSET = (( BUFFOFFSETinit * 100 ) / gSpeedTbl[setPoint])*2;
        BUFFOFFSET = BUFFOFFSET + 30;

/* reset the flag to zero */
        changefrequency = 0;

}

/*
 *      Maxfrequency_FD -       Search for the highest audible frequency component in a
 *                              frame base on the ATH. Apply FFT on the input samples
 *                              from the gBufferRcv and store in to gFFT. compute the
 *                              power for each frequency in the frequency domain, and
 *                              look for the highest frequency with the power is greater
 *                              than the ATH value. note: since the highest audible
 *                              frequency is 20KHz, thus those frequency greater than
 *                              20KHz will have an infinite value.
 */
long Maxfrequency_FD(void)
{
        int i;
        long max_freq = samp_freq / 2;
        long pwr;

        //Copy the gBufferRcv to the gFFT
        for(i = 0 ; i < BUFFSIZE ; i++) {
        gFFT[i] =
((long)(int)gBufferRcv[pingPong|LEFT][i]+(long)(int)gBufferRcv[pingPong|LEFT][i])/2;
  }

        // Find the max frequency
        // note: It only work between 6k - 22k
        rfft(gFFT, BUFFSIZE, SCALE);
        for(i=BUFFSIZE-2; i > 0 ; i -= 2) {
                pwr = (long)(int)gFFT[i] * (long)(int)gFFT[i]
                                + (long)(int)gFFT[i+1] * (long)(int)gFFT[i+1];
                if(pwr <= (long)ATH_pwr[i/2]*4) {
                        gFFT[i] = 0;
                        gFFT[i+1] = 0;
                } else {
                        max_freq = i*(samp_freq/2)/BUFFSIZE;
                        break;
                }
        }
        return max_freq;
}

/*
 *      Maxfrequency_FD -       Search for the highest audible frequency component in a
 *                              frame base on the ATH. Apply FFT on the input samples
 *                              from the gBufferRcv and store in to gFFT. compute the
 *                              power for each frequency in the frequency domain, and
 *                              look for the highest frequency with the power is greater
 *                              than the ATH value. note: since the highest audible
 *                              frequency is 20KHz, thus those frequency greater than
 *                              20KHz will have an infinite value.
 */
long Maxfrequency_TD(void)
{
        int i;
        long max_freq = samp_freq / 2;
        long pwr_high, pwr_med, pwr_low;
        Int16 gFIR[BUFFSIZE*2];
        Int16 outFIR_high[BUFFSIZE*2], outFIR_med[BUFFSIZE*2],
                                        outFIR_low[BUFFSIZE*2] ;

        //Copy the gBufferRcv to the gFIR
        for(i = 0 ; i < BUFFSIZE ; i++) {
        gFIR[i] =
((long)(int)gBufferRcv[pingPong|LEFT][i]+(long)(int)gBufferRcv[pingPong|LEFT][i])/2;
   }

        // Find the max frequency
        fir2(gFIR, COEFFS_HIGH, outFIR_high, size, ORDER_HIGH);
        fir2(gFIR, COEFFS_MED, outFIR_med, size, ORDER_MED);
        fir2(gFIR, COEFFS_LOW, outFIR_low,  size, ORDER_LOW);

        for(i=BUFFSIZE-2; i > 0 ; i -= 2) {
                pwr_high = pwr_high + (long)(int) outFIR_high [i] * (long)(int) outFIR_high [i]
                                + (long)(int)outFIR_high[i+1] * (long)(int) outFIR_high [i+1];
                pwr_med = pwr_med + (long)(int) outFIR_med [i] * (long)(int) outFIR_med [i]
                                + (long)(int)outFIR_med[i+1] * (long)(int) outFIR_med [i+1];
                pwr_low = pwr_low + (long)(int) outFIR_low [i] * (long)(int) outFIR_low [i]
                                + (long)(int)outFIR_low[i+1] * (long)(int) outFIR_low [i+1];
        }


                if(pwr_high > (long)ATH_pwr_high[i/2]*4) {
                        max_freq = i*(samp_freq)/BUFFSIZE;
                } else {
```

```
			if(pwr_med > (long)ATH_pwr_med[i/2]*4)
				max_freq = i*(samp_freq/2)/BUFFSIZE;
			else
				max_freq = i*(samp_freq/4)/BUFFSIZE;
		}

	return max_freq;
}


/*
 * computenextfrequency -	compute the new lowest clock frequency for the
 *				next state base on the highest frequency component
 *				in a frame where is fast enough for all the threads
 *				meet their deadline. The new frequency will be compute
 *				base on the fir_OH,fir_CC,ther_CC and UPDN_sampling.
 */

void computenextfrequency(void)
{
	int div;
	long max_freqs;
	long Total_CC2;
	long fir_CC;
	long fir_OH;
	long fremin;			// minimum acceptable frequency


	if(!switch3){
		max_freqs = Maxfrequency();
		//compute the ratio
		for(step = 1 ; (samp_freq >> (step+1)) > max_freqs ; step++);
		step--;

		if (step > 3) {
			step = 3;
		}

		div = (1<<step);

	/* compute the timeleft for state 2 in m second */
	timeleft =  ( BUFFSIZE - BUFFOFFSET )  / 48 - 1;

	/* Note: the  "- 1" in the above equation is the RTDX overhead */
	/* Compute the clock cycle needed for N numbers of fir filter
	 * in 1000 units base on Buffer size, down sample rate and the number of
	 * the coefficient.
	 */

	fir_CC = ( ( ( BUFFSIZE / ( 16 * div ) ) * ( 9 + ( ORDER - 2 ) ) ) / 125 + 1) * NFIRS;
	fir_OH = (32 * NFIRS )/1000 + 1;
```

```
        /* Compute the total clock cycle needed for second state in K unit */
        Total_CC2 = fir_OH + fir_CC + other_CC + UPDN_sampling + RTDX_OH;
  }
        else{

                timeleft = BUFFSIZE    /48 - 1;

      /* Compute the clock cycle needed for N numbers of fir filter
         * in 1000 units base on Buffer size, down sample rate and the number of
           * the coefficient.
         */
        fir_CC = ( ( ( BUFFSIZE / ( 16 * div ) ) * ( 9 + ( ORDER - 2 ) ) ) / 125 + 1) * NFIRS;
        fir_OH = (32 * NFIRS )/1000 + 1;


        /* Compute the total clock cycle needed for second state in K unit */
        Total_CC2 = fir_OH + fir_CC + other_CC + RTDX_OH;
        }

  /* Compute the minimum frequency where all thead can meet their deadline */
        fremin = Total_CC2 / timeleft;
        fremin = fremin;

        /* Compare the list of CPU frequency from the gSpeedTbl to get a new minimum
         * CPU frequency which is greater than fremin
         */
        for(setPoint = 0 ;( gSpeedTbl[setPoint] < (int)fremin) && (setPoint < 16) ;
                        setPoint++);
                if (setPoint > 15) {
                        setPoint = 15;
        }

  LOG_printf(&logTrace,"setpoint = %d",setPoint);

  //check the change of the frequency
  if (setPoint != prevSetPoint){
        DVS();
        changefrequency=1;

        prevSetPoint = setPoint;
  }

}
/* --------------------- Interrupt Service Routines -------------------- */
/*
 *  dmaHwi() - Interrupt service routine for the DMA transfer.  It is triggered
 *           when a DMA complete receive frame has been transferred.   The
 *           hwiDma ISR is inserted into the interrupt vector table at
 *           compile time through a setting in the DSP/BIOS configuration
 *           under Scheduling --> HWI --> HWI_INT9.  dmaHwi uses the DSP/BIOS
 *           Dispatcher to save register state and make sure the ISR
 *           co-exists with other DSP/BIOS functions. If DIP switch #3 is up,
 *            the audio passes straight through.  If DIP switch #3 is down,
 *            this will be in the power reduction by varying the dynamic
 *            sampling rate mode.
 *
 */
void dmaHwi(void)
{
   // Ping-pong state.  Initialized to PING initially but declared static so
   // contents are preserved as dmaHwi() is called repeatedly like a global.
   static Int16 pingOrPong = PING;
   static int secondState = 0;


  /* The ping or pong buffer were broke down to two part in order to
   * do the frequency scaling without affecting the Data transfer from
   * the codec
   * The first part is used to calculate the new frequency for the second
   * state. At the begining of the second state DMA will be stopped
   * before the frequency change. The rest of the second state will do the
   * filtering process.
   */

   if (!secondState){
         // First state
         DSK5510_LED_on(0);
         //Search for max frequency
         computenextfrequency();

         secondState = 1;
   }
         else {
                 // second state
                 DSK5510_LED_on(1);

                 // change frequency
                 if(changefrequency)
                         switchfrequency();


         // Determine if current state is PING or PONG
         if (pingOrPong == PING) {
            // Post SWI thread to process PING data
            pingPong = PING;
            processBuffer();

            // Set new state to PONG
            pingOrPong = PONG;
         }
```

```
            else {

               // Post SWI thread to process PONG data
               pingPong = PONG;
               processBuffer();

                  // Set new state to PING
            pingOrPong = PING;
            }

                     switch3 = DSK5510_DIP_get(3);
                     if(switch3){
                               computenextfrequency();
                     }
            else{
                     //set it back to first state
                     secondState = 0;
            }
            }

            setDMAdata_addr(secondState);
   // Read the DMA status register to clear it so new interrupts will be seen
   DMA_RGETH(hDmaRcv, DMACSR);
   DSK5510_LED_off(0);
   DSK5510_LED_off(1);
}



/* -------------------------- main() function ------------------------- */
/*
 *  main() - The main user task.  Performs application initialization and
 *          starts the data transfer.
 */
void main()
{
   volatile DSK5510_AIC23_CodecHandle hCodec;
   PSL_Status status;
   unsigned initFreqIndex = INITSETPOINT;


   // Initialize the board support library, must be called first
   DSK5510_init();

   // Initialize LEDs and DIP switches
   DSK5510_LED_init();
   DSK5510_DIP_init();


            status = PSL_initialize(1, &clk, &initFreqIndex, 1.6f);
   // Clear buffers
   memset((void *)gBufferRcv, 0, BUFFSIZE * 8);
   processing_init();

   // Start the codec
   hCodec = DSK5510_AIC23_openCodec(0, &config);

   // Start the DMA controller for the receive transfer
   initDma();

   // Set up interrupts
   initIrq();

         CHIP_FSET(ST3_55, CLKOFF, 1);

   // Start the DMA
   DMA_start(hDmaRcv);
   DMA_start(hDmaXmt);

   //set the pong state in order to come back in between the frame;
   pingPong = PONG;
   // configure the DMA data address register
   setDMAdata_addr(0);

}
```

Hearing_Aid.h

```
#ifndef HEARING_AID_H
#define HEARING_AID_H

void SetSpeed(int nx, int nfirs);

#endif
```

Processing.h

```
#ifndef PROCESSING_H
#define PROCESSING_H
#define NFIRS  30

void processing_init(void);

void hearingAid(Int16 *inL, Int16 *inR, Int16 *outL, Int16 *outR, int size);

#endif
```

Processing.c

```
#include <dsplib.h>
```

```
// should #include <time.h>, but headers are broken badly
#include <linkage.h>
typedef unsigned long clock_t;
_CODE_ACCESS clock_t    clock(void);

#include "hearing_aidcfg.h"
#include "processing.h"
#include "highpass.h"        // high pass filter coefficients

#define ORDER       208
#define BUFFSIZE        1024

Int16 delayBufferL[ORDER+2]={0};
Int16 delayBufferR[ORDER+2]={0};

void processing_init(void)
{
}

/*
 *         hearingAid-      suppose the heading aid coding will be adder here
 *                                   but right now the fir filter just act like a load for
 *                                   the normal hearing aid process.
 *                                   the input size might vary from BUFFSIZE  to BUFFSIZE/8
 *                                   provided the max value for step is 3.
 */
void hearingAid(Int16 *inL, Int16 *inR, Int16 *outL, Int16 *outR, int size)
{
        int i;

        // insert real filter code here
        for(i =0; i < (NFIRS/2); i++){
                fir2(inL, COEFFS, outL, delayBufferL, size, ORDER);
                fir2(inR, COEFFS, outR, delayBufferL, size, ORDER);
   }


   copyData(inL, outL, size);
   copyData(inR, outR, size);
}
```

CHAPTER 9

REFERENCES

[1] D. Adams. The hitchhiker's guide to the galaxy, CD Recording. *New Millenium Audio*, 2002.

[2] M. Bille, A. Jensen, E. Kjærbøl, V. Vesterager, P. Sibelle and H. Nielsen. Clinical study of a digital vs. an analogue hearing aid. In *Taylor and Francis Health Sciences, part of the Taylor & Francis Group*, 1999, Vol. 28, No. 2, pp. 127-135.

[3] B. Brock and K. Rajamani. Dynamic power management for embedded systems. In Proceedings *of the IEEE System On Chip Conference,* 2003, pp. 416-419.

[4] T. D. Burd and R. W. Brodersen. Energy efficient CMOS microprocessor design. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences, Volume 1: Architecture,* 1995, p.288.

[5] K. Choi, K. Dantu, W. C. Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a MPEG decoder. In *International Conference on Computer-Aided Design*, 2002, pp. 732-737.

[6] R. Cyran. Using the power scaling library on the TMS320C5510. *Technical Report SPRA848*, Texas Instruments, P.O. Box 655303 Dallas, Texas 75265, October 2002.

[7] W. R. Dieter, S. Datta and W. K. Kai. Power reduction by varying sampling rate. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2005, pp. 227-232.

[8] A. T. Erdogan, M. Hasan and T. Arslan. Algorithmic low power FIR cores. In *Circuits, Devices and Systems, IEEE Proceedings*, 2003, Vol. 50, No. 3, pp. 155-160.

[9] R. A. Gopinath and C. S. Burrus. On upsampling, downsampling, and rational sampling rate filter banks. In *Signal Processing, IEEE Transactions on Acoustics*, 1994, Vol. 42, no. 4, pp. 812-824.

[10] W. Huang, Y. Wang and S. Chakraborty. Power-aware bandwidth and stereo-image scalable audio decoding. In *Proceedings of the 13th Annual ACM International Conference on Multimedia*, 2005, pp. 291 – 294.

[11] C. Im, S. Ha, and H. Kim. Dynamic voltage scheduling with buffers for low-power multimedia applications. *ACM Transactions on Embedded Computing Systems*, 2004, Vol. 3, No. 4, pp. 686–705.

[12] R. Klootsema, O. Nys, E. Vandel and D Aebischer. Battery supplied low power analog-digital front-end for audio applications. In *Solid-State Circuits Conference*, 2000, pp. 118-121.

[13] J. W. S. Liu. Real-Time Systems. *Pearson Education*, 2003.

[14] A. Manzak and C. Chakrabarti. Variable voltage task scheduling algorithms for minimizing energy. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, 2001, pp. 279-282.

[15] E. Meijering. A chronology of interpolation: from ancient astronomy to modern signal and image processing. In *Proceedings of the IEEE*, 2002, Vol. 90, No. 3, pp. 319-342.

[16] D. Mosse, H. Aydin, B. Childers and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low-Power*, 2000.

[17] J. Korhonen and Y. Wang. Power-efficient streaming for mobile terminals. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2005, pp. 39 – 44.

[18] A. Maxiaguine, S. Chakraborty and L. Thiele. DVS for buffer-constrained architectures with predictable QoS-energy tradeoffs. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2005, pp. 111-116.

[19] A. V. Oppenhiem and R. W. Schafer. Digital Signal Processing. *Prentice-Hall,* 1975.

[20] D. Pan. A Tutorial on MPEG/Audio Compression. *IEEE MultiMedia*, June 1995 Vol. 2, No. 2, pp. 60-74.

[21] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Symposium On Operating System Principles,* 2001, pp. 89-102.

[22] T. Painter and A. Spanias. Perceptual Coding of Digital Audio. In *Proc. IEEE*, 2000, Vol. 88, No. 4, pp. 451-515.

[23] T. Painter and A. Spanias, Perceptual segmentation and component selection in compact sinusoidal representations of audio. In *International Conference on Acoustics, Speech and Signal Processing*, 2001, Vol. 13, No. 2, pp. 149-162.

[24] C. Perkins, O. Hardson and V. Hardmon. A Survey of Packet Loss Recovery Techniques for Streaming Audio. *IEEE Network Magazine,* 1998, Vol. 12, No. 5 pp. 40-48.

[25] M. Potkonjak and G. Qu. Energy minimization with guaranteed quality of service. In *International Symposium on Low Power Electronics and Design*, 2000, pp. 43-48.

[26] J. Proakis and D. Manolakis. Digital Signal Processing. *Addison Wesley*, 1995.

[27] L. R. Rabiner and B. Gold. The Theory and Application of Digital Signal Processing. *Prentice-Hall,* 1975.

[28] J. J. K. Ó Ruanaidh and W. J. Fitzgerald. Interpolation of missing samples for audio restoration. In *Electronics Letters*, 1994, Vol. 30, No. 8, pp. 622-623.

[29] C. Santana. The Best of Santana, CD Recording. *Sony Music Entertainment,* 1998.

[30] T. Schneider and R. Brennan. An ultra low-power programmable DSP system for hearing aids and other audio applications. In *Proceedings of International Conference on Signal Processing Applications & Technology*, 1999, pp. 569-572

[31] T. Simunic, L. Benini, A. Acquaviva and P. Glynn. Dynamic voltage scaling and power management for portable systems. In *Proceedings of the 38th conference on Design Automation*, 2001, pp. 524-529.

[32] J. O. Smith. Physical modeling synthesis update. *Computer Music Journal,* 1996 Vol. 20, No. 2, pp. 44-56.

[33] Spectrum Digital, 12502 Exchange Drive, Suite 440, TX 77477. *MS320VC5510 DSK Technical reference*, 2002.

[34] T. Stetzler, N. Magotra, P. Gelabert, P. Kasthuri, and S. Bangalore. Low-power real-time programmable DSP development platform for digital hearing aids. *Technical Report SPRA657, Texas Instruments*, 2000.

[35] Texas Instruments, P.O. Box 655303 Dallas, Texas 75265. *TMS320C55x DSP Library Programmer's Reference*, 2003.

[36] K. Tsutsui, H. Suzuki, O. Shimoyoshi, M. Sonohara, K. Akagiri and R. M. Heddle. ATRAC: Adaptive Transform Acoustic Coding for MiniDisc. *Sony Corporate Research Laboratories, Reprinted from the 93rd Audio Engineering Society Convention*, 1992.

[37] M. Weiser, B. Welch, A. Demers and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI),* 1994, pp. 13–23.

[38] N. Weste and K. Eshragian. Principles of CMOS VLSI Design: A Systems Perspective. *Addison-Wesley, MA*, 1988.

[39] E. Zwicker and H. Fastl. Psychoacoustics: Facts and Models. *Springer-Verlag*, Berlin, Heidelberg, 1990.

[40] E.P. Zwyssig, A.T. Erdogan and T. Arslan. Low power system on chip implementation scheme of digital filtering cores. In *Low Power IC Design Seminar, IEEE Seminar,* 2001, pp. 5/1-5/9.

Vita

- Date and place of birth: 04/13/1979, City: Calcutta, State: West Bengal, Country: India

- Educational institutions attended and degrees already awarded:

  Bachelor of Engineering, Electrical and Electronics Engineering, Birla Institue Of Technology, India, May2001

- Professional positions held:

  CAD Engineer, Cypress Semiconductor, June 2005 to present

  IC Design Engineer, Texas Instruments, India, January 2002 – December 2003

  Project trainee, CMC, Calcutta Summer 2000

  Summer Intern, Calcutta Electrical Supply Company, Summer 1999