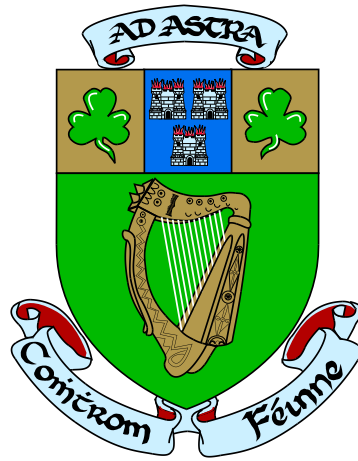


Port of a Fixed Point MPEG-2 AAC Encoder on a ARM Platform

by

Romain Pagniez

romain@felinewave.com



A Dissertation submitted in partial fulfillment of the requirements for the Degree of

Master of Science in Computational Science

University College Dublin

Department of Computer Science

Information Hiding Laboratory

University College Dublin, Ireland

August 2004

Abstract

This dissertation is the result of a three months work carried out at the Information Hiding Laboratory (University College Dublin, Ireland) submitted in partial fulfillment of the requirements for the Degree of of Master in Science in Computational Science. My work on porting a fixed point MPEG-2 AAC Encoder on a ARM platform is detailed.

MPEG like encoders are quite complex and involve a huge amount of computations. Computing the encoding algorithm on a dedicated hardware chip could be an efficient solution allowing fast processing at reduced cost. Some research have been done in the Information Hiding Laboratory in that field providing an efficient and precise algorithm using fixed point number representation and arithmetic.

The port of the encoder on the EPXA1 development board has been quite straight forward from the simulation files developed by Keith Cullen. The port is working just as expected giving the exact same result as the simulations. For convenience, I have implemented some communication functions which allow to encode complete audio files from a host PC.

The main part of my work has been to implement the communication functions across the board and a host PC. I have also looked for some fast extended precision multiplication algorithm in order to faster the encoding time on the board.

Using `long long` integers, the double precision multiplication algorithm has been highly boosted. The computing time of the filter bank block has been decreased of 60% and the computing time of the encoding has been decreased by 40%.

The communication functions have also been optimized by overlapping the communications and the computations on the board. This overlapping has led to save more than 40% of the overall encoding process compared to the naive implementation.

The latest implementation of the encoder (including communications) is a bit more than 50% faster than the original non-optimized one.

The next optimization is to implement the communications on the Ethernet port of the board as computations only represent 66% of the overall encoding time, the remaining time been used by communications.

Acknowledgements

First, I would like to thank Dr Guéno lé Silvestre and Dr Neil Hurley for having me come at UCD while I am finishing my fifth year of electrical engineering at ISEP (Institut Sup rieur d'Electronique de Paris, Paris - France) in parallel.

This work would not have been possible without the source code of the fixed point MPEG2-AAC encoder implemented by Keith Cullen. I would like to thank him particularly for the impressive work he has done to facilitate my work on the encoder.

Finally I would like to thank Alexis Gu erin for his continuous help along the project and his friendship throughout the year.

Contents

Abstract	i
Acknowledgements	ii
Content	iii
Figures	vi
Acronyms	viii
Introduction	1
1 MPEG-2 AAC: State of the Art in Perceptual Audio Compression Technology	6
1.1 Basic Principles of Psychoacoustic	6
1.1.1 The Absolute Threshold of Hearing	7
1.1.2 The Masking Phenomenons	8
1.1.3 The Critical Bands Concept	9
1.2 General Structure of Perceptual Audio Encoders	12
1.3 MPEG-2 AAC Encoding Process Explained	14
1.3.1 Filterbank	16
1.3.2 Psychoacoustic model	18
1.3.3 Temporal Noise Shaping	19
1.3.4 Joint Stereo Coding	19
1.3.5 Prediction	20
1.3.6 Quantization and Noiseless Coding	20
1.3.7 Bitstream Formatting	24

2	Basis of Fixed Point Arithmetic	25
2.1	Introduction	25
2.1.1	Floating Point Number Representation	25
2.1.2	The need for fixed point algorithms	26
2.2	Fixed Point Numerical Representation	26
2.2.1	Unsigned Fixed Points	26
2.2.2	Two's Complement Fixed Points	27
2.3	Basic Fixed Point Arithmetic Operations	29
2.3.1	Addition	29
2.3.2	Multiplication	31
2.4	Position of the binary point and error	32
2.5	Summary of fixed point arithmetic	33
3	Development Toolset	35
3.1	EPXA1 Development kit	35
3.1.1	Content of the kit	35
3.1.2	EPXA1 development board	36
3.1.3	Quartus II	38
3.1.4	GNUpro	40
3.2	Usage guide	41
3.2.1	Software compilation doesn't work	41
3.2.2	Do not declare any long arrays into a function	41
3.2.3	no file system	41
4	Implementation	42
4.1	Port of the filter bank only	42
4.2	Port of the full encoder	43
4.3	Communication functions	44
4.3.1	Communication functions - host PC side	44
4.3.2	Communication functions - EPXA1 side	45
4.3.3	Communication protocol for the filter bank only	46

4.3.4	Communication protocol for the encoder	47
4.4	Double precision multiplication algorithms	49
4.4.1	Original Code	49
4.4.2	A restricted use of the Karatsuba algorithm	51
4.4.3	The basic shift and add algorithm	53
4.4.4	Some little changes in the original code	54
4.4.5	Using long long integers	56
5	Results - Evaluation	57
5.1	Filter Bank	57
5.1.1	Timing	57
5.1.2	Precision	58
5.2	Complete Encoder	59
5.2.1	Non-optimized version	59
5.2.2	Communication overlapping computations version	60
5.2.3	Communication overlapping computations version and high speed multi- plication	60
	Conclusion	62
	Bibliography	63

List of Figures

1	IHL hardware encoding workflow	2
2	Analog voltage line	3
3	Discrete time and amplitude sampling	3
4	Uncompressed audio formats	4
1.1	Absolute threshold of hearing	7
1.2	Frequency or simultaneous masking	8
1.3	Temporal masking	9
1.4	Overall masking threshold	9
1.5	Overall masking threshold along time	10
1.6	Critical bands concept	11
1.7	Critical bands concept	11
1.8	Critical bands and bark unit	13
1.9	Overview of perceptual audio encoders	14
1.10	Block diagram of the AAC encoding process	15
1.11	Sine and KBD windows	16
1.12	AAC filterbank	17
1.13	Basic idea in data rate reduction schemes	21
1.14	SNR, SMR & NMR	21
1.15	Block diagram of MPEG-2 AAC noise allocation loop	23
2.1	Representation of signed fixed point numbers	28
2.2	Two's complement fixed point representation	29
2.3	Examples of 4-bit addition	29
2.4	Examples of 8-bit addition	30
2.5	Examples of 4-bit multiplication	31
2.6	Range and precision of the 4-bit unsigned fixed point system	32

2.7	Determining the best radix position	33
3.1	Overview of the EPXA1 Development Board	36
3.2	Connections of the EPXA1 Development Board	37
3.3	Connecting the LCD module to the EPXA1 development board	38
3.4	Blank main window of Quartus II software	39
3.5	Implementation architecture	39
3.6	Flash programming files flow	40
4.1	Fixed point filter bank simulation	42
4.2	Master - slave architecture	43
4.3	Fixed point encoder simulation	43
4.4	Communication frame for the filter bank block only	46
4.5	Communication sequence	47
4.6	Serial and overlapped communications	48
4.7	Typical N-bit integer ALU	51
4.8	Fixed point variables are limited to half the word length of integer variables	52

Acronyms & Abbreviations

AAC	Advanced Audio Coding
AES	Audio Engineering Society
AHB	Altera Hardware Bus
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ARM	Advanced RISC Machines
ATH	Absolute Threshold of Hearing
AWL	Arbitrary Word Length
bit	Binary digit
CD	Compact Disc
CPU	Central Processing Unit
DAT	Digital Audio Tape
DCT	Discrete Cosine Transform
DSP	Digital Signal Processor
DTFT	Discrete Time Fourier Transform
DVD	Digital Video Disc or Digital Versatile Disc
EBU	European Broadcasting Union
EDA	Event Driven Architecture
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FWL	Fractional Word Length
HAS	Human Auditory System
Hz	Hertz
IEC	International Electrotechnical Commission
IHL	Information Hiding Laboratory
IS	Intensity Stereo
ISEP	Institut Supérieur d'Electronique de Paris
ISO	International Organization for Standardization
ITU	International Telecommunication Union
IWL	Integer Word Length
JTAG	Joint Test Action Group
KBD	Kaiser Bessel Derived
kHz	kilo Hertz / 1,000 Hertz
LCD	Liquid Crystal Display
LED	Light-Emitting Diode

LSB	Least Significant Bit
MDCT	Modified Discrete Cosine Transform
MPEG	Motion Picture Experts Group
MS	Middle/Side Stereo
MSB	Most Significant Bit
MSc	Master of Science
NBC	Non Backward Compatible
NIC	Network Interface Card
NMR	Noise to Mask Ratio
PC	Personal Computer
PCM	Pulse Code Modulation
PhD	Doctor of Philosophy
PLD	Programmable Logic Device
RISC	Reduced Instruction Set Computer
SMR	Signal to Mask Ratio
SNR	Signal to Noise Ratio
SOPC	System On a Programmable Chip
TDAC	Time Domain Aliasing Cancelation
TNS	Temporal Noise Shaping
UCD	University College Dublin
WL	Word Length

Introduction

During the past decade, the demand for digital audio compression has increased drastically. The explosion of internet has caused a boom in the utilization of audio compression technologies. The Internet audio market is one of the world's fastest growing markets and all analysts have predicted dramatic growth over the next few years. As well as internet distribution, many other applications have arisen such as hand-held players, DVD, cable distribution, satellite and digital television. It is vital for most applications to use compression technologies providing both high quality audio and high compression ratios.

Perceptual audio compression

Perceptual encoding appears to be the only technology that may be able to match the severe requirements stated. Perceptual audio compression can be defined as a lossy but perceptually lossless compression, meaning that although the exact original signal cannot be retrieved from the compressed stream, almost no degradation of the audio quality can be perceived by a human observer. The limitations of the human auditory system are taken into account to make the degradations in the signal precision imperceptible. MPEG/audio standard was accepted by the International Organization for Standards and the International Electrotechnical Commission (ISO/IEC) in 1992. It contains several techniques for perceptual audio compression and includes the popular Layer III, which manages to compress CD audio from 1.4 Mbit/s to 128 kbits/s with very little audible degradation. Advanced Audio Coding (AAC) is referred to as the next generation of MPEG audio algorithms. It is part of both the MPEG-2/audio and MPEG-4/audio standards. By removing 90% of the signal component without perceptible degradation, MPEG-2 AAC achieves a compression rate about 30% higher than MP3. Compressed audio at 128 kbit/s is indistinguishable from the original 1.4 Mbit/s CD audio.

hardware targetting

MPEG-like perceptual encoding algorithms are quite complex and involve a significant amount of calculations. The processing power required to compress the audio is an important issue, notably for real-time applications. In many applications, computing the algorithm on a dedicated hardware chip could be an efficient solution allowing fast encoding at a reduced cost. As most hardware ships only performs fixed point arithmetic, research has been conducted in the Information Hiding Laboratory of UCD by Keith Cullen on the implementation of a fixed point MPEG2-AAC encoder that would be used on chips like FPGA or simple RISC architecture processors.

An FPGA implementation of the encoder is developed by Alexis Guérin. FPGAs stands for Field Programmable Gate Array, they are a type of logic chip that can be programmed. FPGAs support thousands of gates. They are especially popular for prototyping integrated circuit designs. Their architecture differs significantly from traditional sequential CPUs in allowing substantial parallel processing power.

The final goal of my project is to port the code of the encoder on a ARM microprocessor platform. ARM processors are quite simple sequential RISC processors designed for embedded applications. In particular our development board has a 32-bit ARM9 which runs at 100 MHz with 8k cache. Figure 1 reports the research workflow led in the Information Hiding Laboratory and the position of my project.

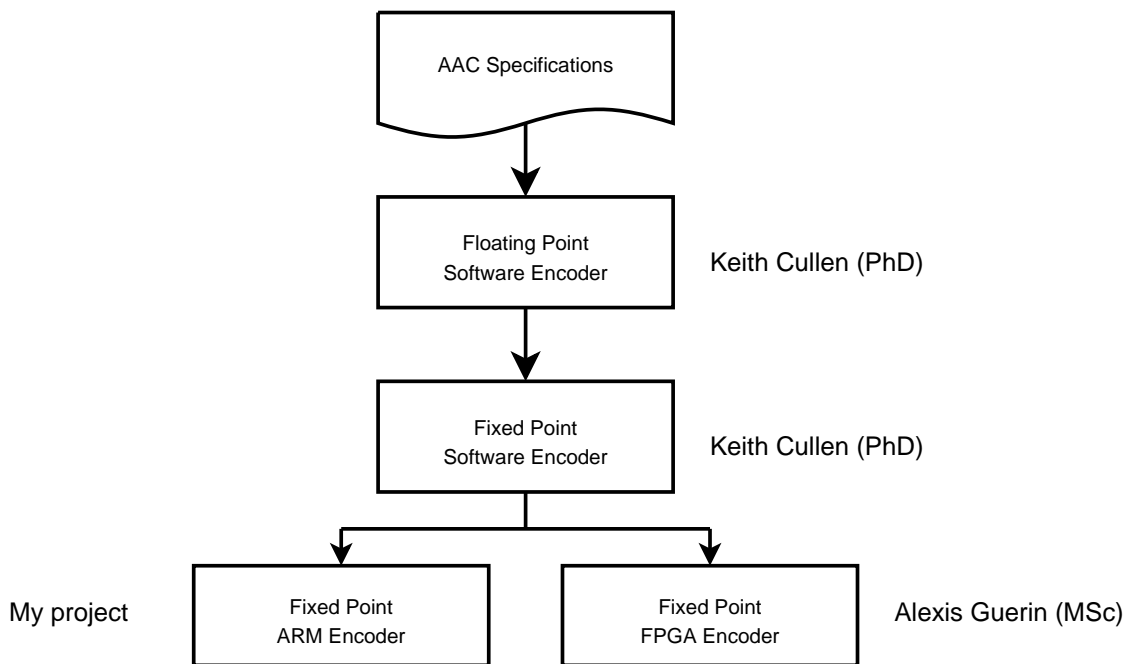


Figure 1: *Hardware encoding workflow in the Information Hiding Laboratory.*

Digital Audio Data

Before talking about audio compression, here is a quick overview of natural uncompressed digital audio data.

At the beginning of audio electronics, analog representation were used in order to amplify or record sounds. The analog representation makes voltage of an electrical line directly proportional to the correspondent sound pressure level. To a particular acoustic wave in the pressure scale corresponds the exact *analog* in the voltage scale. Transcription from one domain to the other is usually performed by microphones and speakers.

In the seventies, appeared the digital representation of audio which offers many advantages compared to the analog one: high noise immunity, stability and reproducibility. Though storage

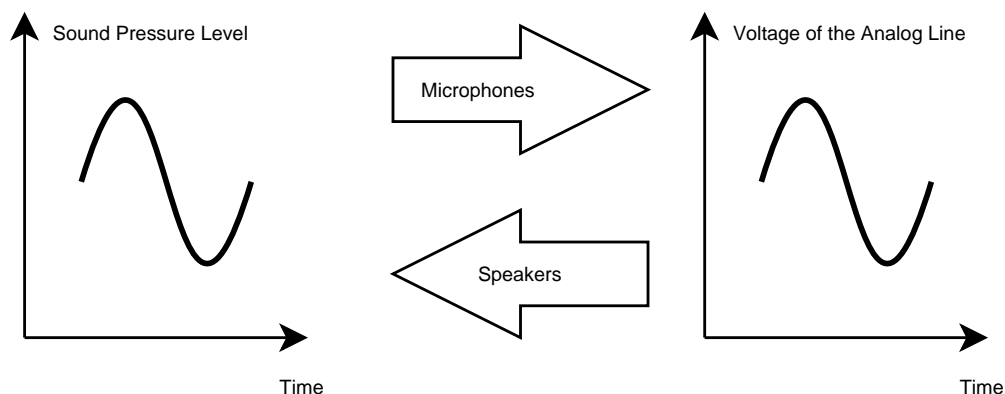


Figure 2: *The voltage of the analog line is the exact replication of the sound pressure level.*

and broadcast of digital audio is easier and may benefit of progress in other digital technologies such as computers, networks and digital communications.

The conversion from the analog to the digital domain is achieved by sampling the audio input in regular, discrete intervals of time. Those samples are then quantized into a discrete number of usually evenly spaced levels. Though the digital audio data consists of a sequence of binary words representing discrete amplitudes of consecutive time-discretized samples of the correspondent analog audio signal. The method of representing each sample with an independent code word is called pulse code modulation (PCM). Figure 3 shows the analog to digital conversion process.

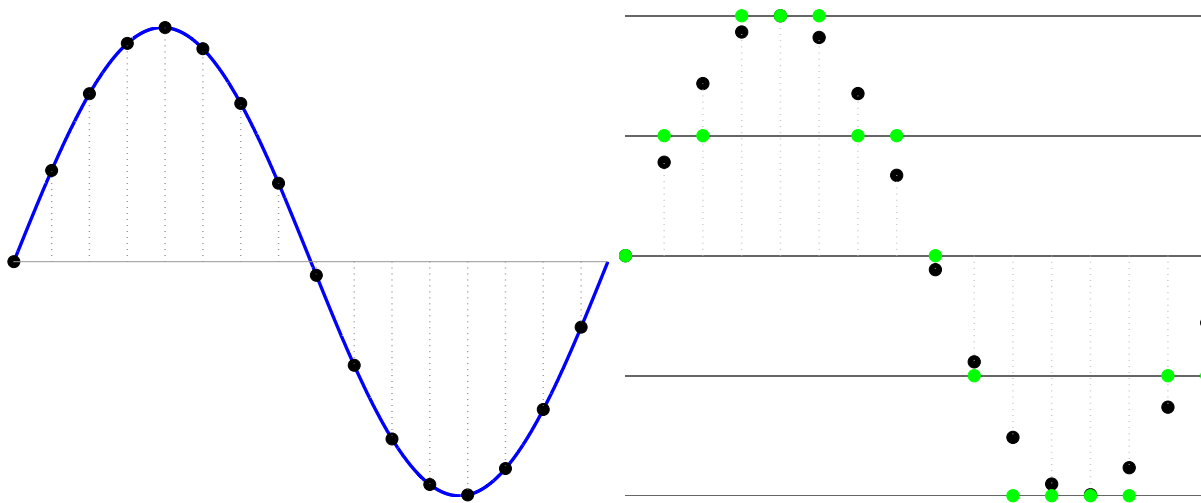


Figure 3: *Time and amplitude discretisation. The input signal in blue is first regularly sampled in time giving the black time-discretized points. Amplitudes of those black points are then discretized into a discrete set of possible values giving the green points. the digital audio data is the sequence of green points.*

As stated below, the two main stages of analog-to-digital audio conversion are based on time and amplitude discretization. Those operations are dependent of important parameters: the sampling frequency and the number of possible values for the amplitude discretization. Parameters of some of the most popular formats are reported on figure 4.

According to the Nyquist theory, a time-sampled signal can faithfully represent signals up to

half the sampling rate. As human hearing is usually considered below the limit of 20 kHz, most popular "high fidelity" formats (such as audio CD) have sampling rates around 44 kHz; more restrictive formats such as telecommunications can use sampling rates as low as 8 kHz. At the opposite, higher sampling rates (up to 192 kHz) have recently appeared with audio and video DVDs. A low sampling rate is dangerous because it leads to spectrum replications also called aliasing. To overcome this constraint, a low sampling rate imposes to severely low-pass filter the input. A high sampling rate doesn't suffer from this consideration but generates much more and possibly unuseful audio data.

The number of quantizer levels is typically a power of 2 to make full use of a fixed number of bits per audio sample to represent the quantized data. With uniform quantizer step spacing, each additional bit has the potential of increasing the signal-to-noise ratio, or equivalently the dynamic range, of the quantized amplitude by roughly 6 dB. The typical number of bits per sample used for digital audio ranges from 8 to 24. The dynamic range capability of these representations thus ranges from 48 to 144 dB, respectively. The useful dynamic for a listener is roughly 100 dB. A 16-bit per sample coding is then the more accurate definition. Using less than 16 bits per sample increase the level of audible noise due to quantization.

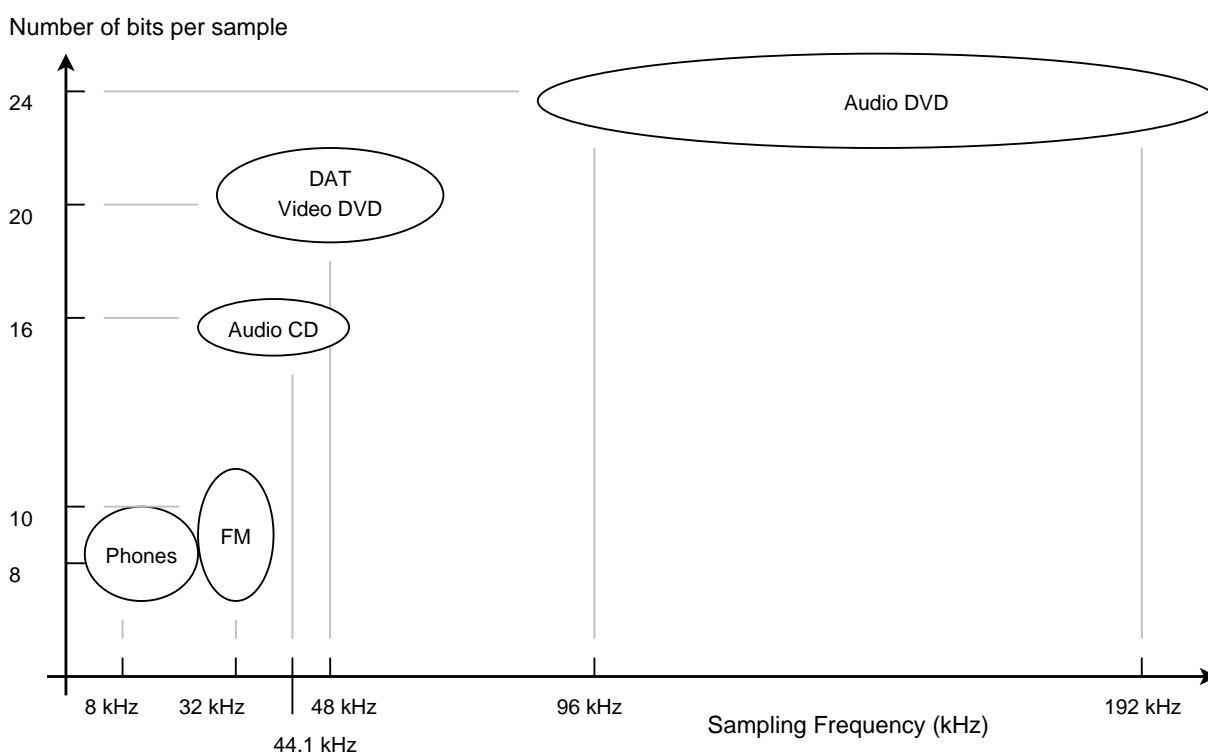


Figure 4: *Parameters of some uncompressed audio formats.*

The data rates associated with uncompressed digital audio are substantial. For example, the audio data on a compact disc (2 channels of audio sampled at 44.1 kHz with 16 bits per sample) requires a data rate of about 1.4 megabits per seconds. There is clear need for some form of compression to enable a more efficient storage and transmission of this data.

Outline

The goal of this dissertation is to report my work on porting an AAC encoder code on an ARM platform. This work is organised as follows:

- Chapter 1 gives the background knowledge necessary to understand the principle of perceptual audio encoding and more particularly how the MPEG-2 AAC encoding works.
- Chapter 2 deals with fixed point number representation and operations, as we are porting a fixed point implementation of the MPEG-2 AAC encoder.
- Chapter 3 details the development toolset used for the port. This part may be a good starting point for any future student willing to use the EPXA1 development toolkit for his applications.
- Chapter 4 highlights how I have ported the code and the modifications I've made. It also deals with the communication functions I've implemented in order for the board to get audio data from a host PC.
- Chapter 5 finally provides a bench of measures and an evaluation of the port.

Chapter 1

MPEG-2 AAC: State of the Art in Perceptual Audio Compression Technology

The Motion Picture Experts Group (MPEG) audio compression algorithm is an International Organization for Standardization (ISO) standard for high-fidelity audio compression. It is one of a three-part compression standard. With the other two parts, video and systems, the composite standard addresses the compression of synchronized video and audio at a total bit rate of roughly 1.5 megabits per seconds.

The MPEG/audio compression is lossy; however, the MPEG algorithm can achieve transparent, perceptually lossless compression. The MPEG/audio committee conducted extensive listening tests during the development of the standard. The tests showed that even with a 6-to-1 compression ratio (stereo, 16 bits per sample, audio sampled at 48 kHz compressed to 256 kilobits per second) and under optimal listening conditions, expert listeners were unable to distinguish between coded and original audio clips with statistical significance.

The high performance of this compression algorithm is due to the exploitation of auditory masking. This masking is a perceptual weakness of the ear that occurs whenever the presence of a strong audio signal makes a spectral neighborhood of weaker audio signals imperceptible.

1.1 Basic Principles of Psychoacoustic

The most important characteristic of an audio compression system is the perceived fidelity to the original signal by the human ear. The noise introduced by the compression must be as inaudible as possible. Perceptual audio codecs take advantage of the inability of the human auditory system to hear quantization noise under certain conditions. Psychoacoustics describes the characteristics of the human auditory system on which perceptual audio compression is based. It is particularly concerned with time and frequency analysis capabilities of the inner ear. The human ear is sensitive to a restricted range of frequencies. Single frequency tones in this range can only be perceived if they are above a certain intensity. The characteristics of this lower limit called the Absolute Threshold of Hearing are described in 1.1.1. The phenomenon whereby one sound becomes inaudible when another sound occurs is called masking and is described in

1.1.2. The capability of the human ear to differentiate similar frequencies is not constant in the audible frequency range. This is the basis of critical bands theory as explained in 1.1.3.

1.1.1 The Absolute Threshold of Hearing

The Absolute Threshold of Hearing (*ATH*) is the most intuitive property of the human auditory system. We simply cannot hear sounds which are too weak. The minimum amount of energy that a pure tone with frequency f (sinusoidal) must have to be detected by the listener in a noiseless environment is called the absolute threshold of hearing for f ($ATH(f)$) or threshold in quiet. This value varies a lot within the ear sensitivity frequency range as shown in figure 1.1.

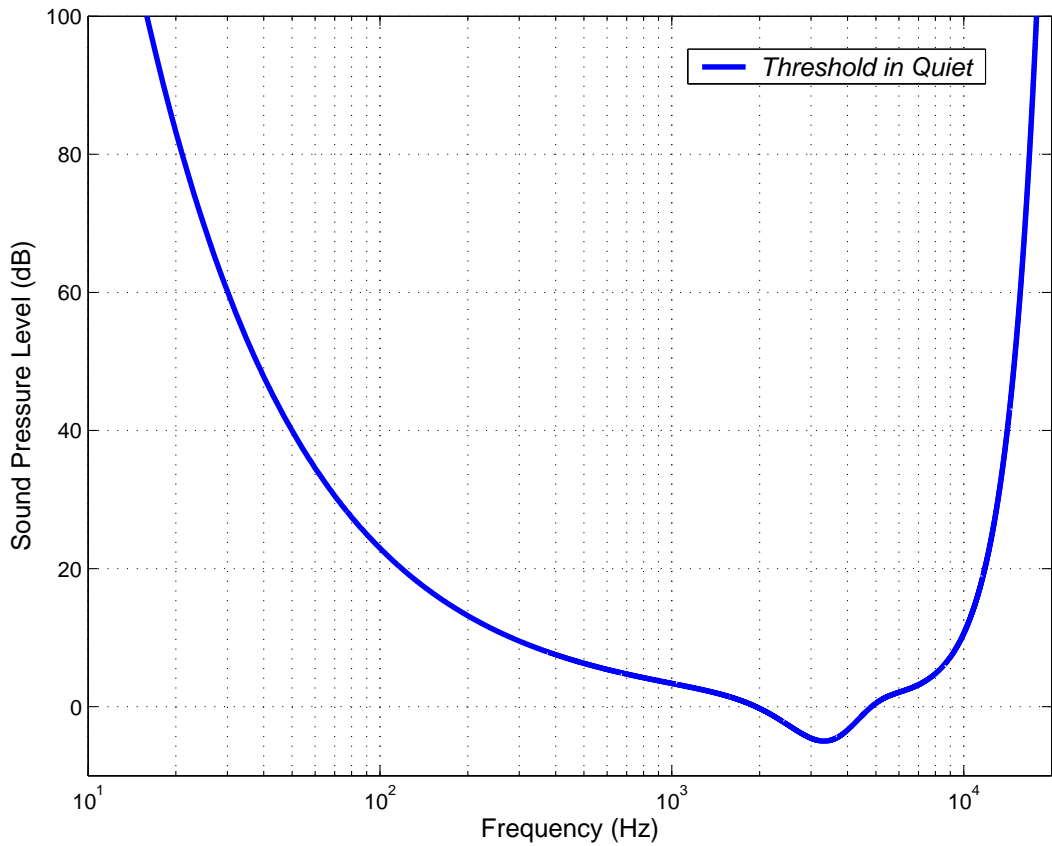


Figure 1.1: Absolute threshold of hearing. A single tone is inaudible if its pressure level is below the absolute threshold of hearing.

An approximation of the *ATH* is given in the MPEG Layer 1 ISO standard as:

$$ATH_{dB}(f) = 3.64 \left(\frac{f}{1000} \right)^{-0.8} - 6.5e^{-0.6 \left(\frac{f}{1000} - 3.3 \right)^2} + 10^{-3} \left(\frac{f}{1000} \right)^4$$

Although the absolute threshold of hearing is a very subjective characteristic of the ear and depends on many factors including the age of the subject, it is a widely used approximation. It is interesting to note that the region where the human ear is most sensitive is around 3000-4000 Hz, which is the average frequency range of the human voice.

1.1.2 The Masking Phenomenons

Masking effects of the human ear are heavily exploited in perceptual audio encoding to make coding noise inaudible.

Frequency Masking (or Simultaneous Masking):

Frequency masking occurs when a louder tone (masker) makes a softer tone (maskee) inaudible. If a tone of a certain frequency and amplitude is present, then other tones of similar frequency but of much lower amplitude are not perceived by the human ear. Thus, there is no need to transmit or store the softer tones. Furthermore, if some additional frequency components have to be added to the signal (ie: watermark, noise), they can be ‘shaped as softer tones’ so that they will be inaudible. The minimum perceptible amplitude level of the softer tone is called the masking threshold. It is usually plotted as a function of the softer tone frequency as shown in figure 1.2. When the masker tone amplitude decreases, the masking threshold generally also decreases, until it reaches a lower limit. This limit is the absolute threshold of hearing (*ATH*) described above.

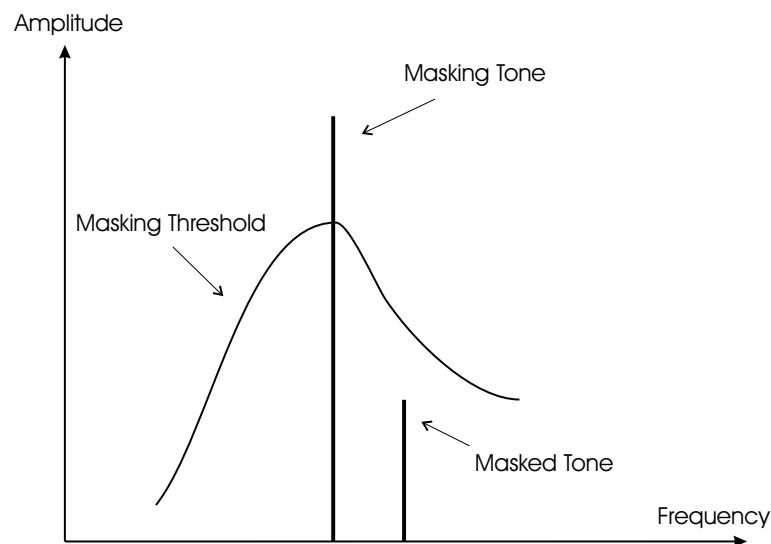


Figure 1.2: *Frequency or Simultaneous Masking. Similar frequency tones are masked by the loud tone.*

Temporal Masking :

When a loud tone with a finite duration occurs, similar frequency softer tones are masked not only during the time of the tone as stated above (simultaneous masking), but also after the masker stops, and even before it starts as shown on figure 1.3. The effect of masking after a strong sound is called post-masking and can last up to 200ms after the end of the masker, depending on masker amplitude and frequency . The effect of masking before a strong noise occurs is called pre-masking and may last up to 20ms.

Overall Masking Threshold :

In normal audio signal, many maskers are usually present at different frequencies. Taking into account the frequency masking, temporal masking, and absolute threshold of hearing, and summing the masking thresholds for every masker in the signal gives the overall masking threshold

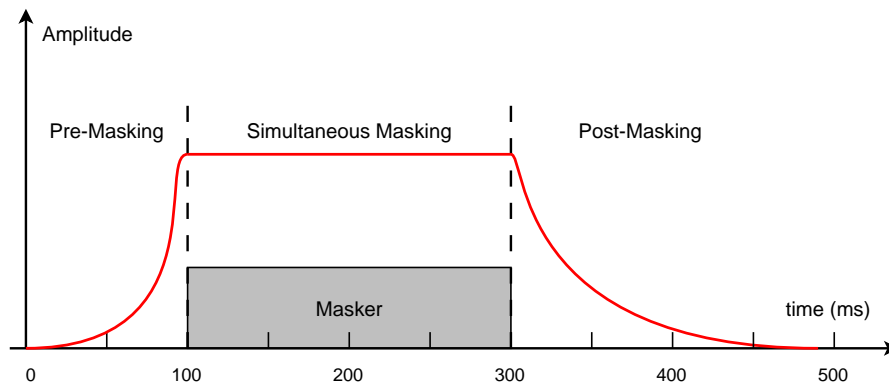


Figure 1.3: *Temporal Masking.* The masking effect appears before the masker starts and last after the end of the masker.

as shown on figure 1.4. It is a time varying function of frequency that indicates the maximum inaudible noise for every frequency at a given time. The calculation of the overall masking threshold can be performed on an audio frame (ie : 2048 consecutive audio samples for MPEG-2 AAC encoding). Longer frames give better frequency resolution while shorter frames give better time resolution.

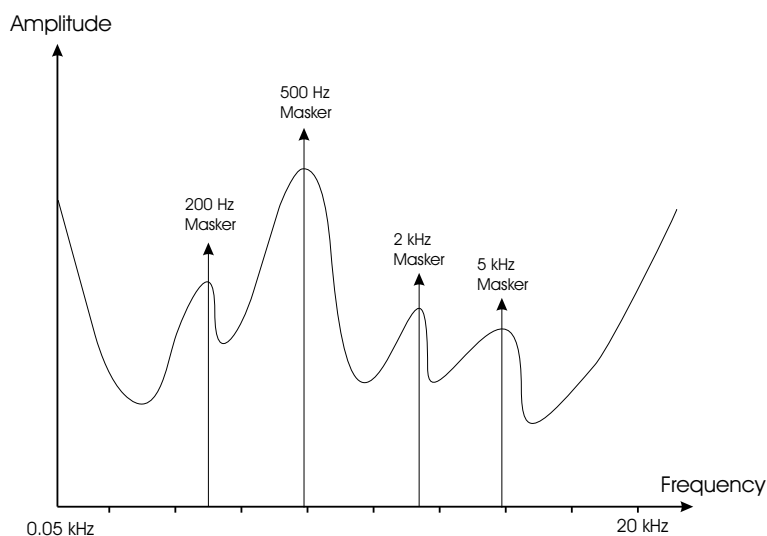


Figure 1.4: *Overall Masking Threshold.* This is a time varying function of frequency that indicates the maximum inaudible noise at each frequency at a given time.

1.1.3 The Critical Bands Concept

The human auditory system has limited frequency resolution. The distinction between two very close single frequency tones cannot be made. This frequency resolution is also frequency-dependent. Listeners tell more easily the difference between 500 Hz and 600 Hz tones than between 17,000 Hz and 18,000 Hz tones. The critical band concept was first introduced by Fletcher (1940) and can be seen as an approximation of the human auditory system's ability to separate sounds of different frequencies. His measurements and assumption led him to model the auditory system as an array of band-pass filters with continuously overlapping pass-bands

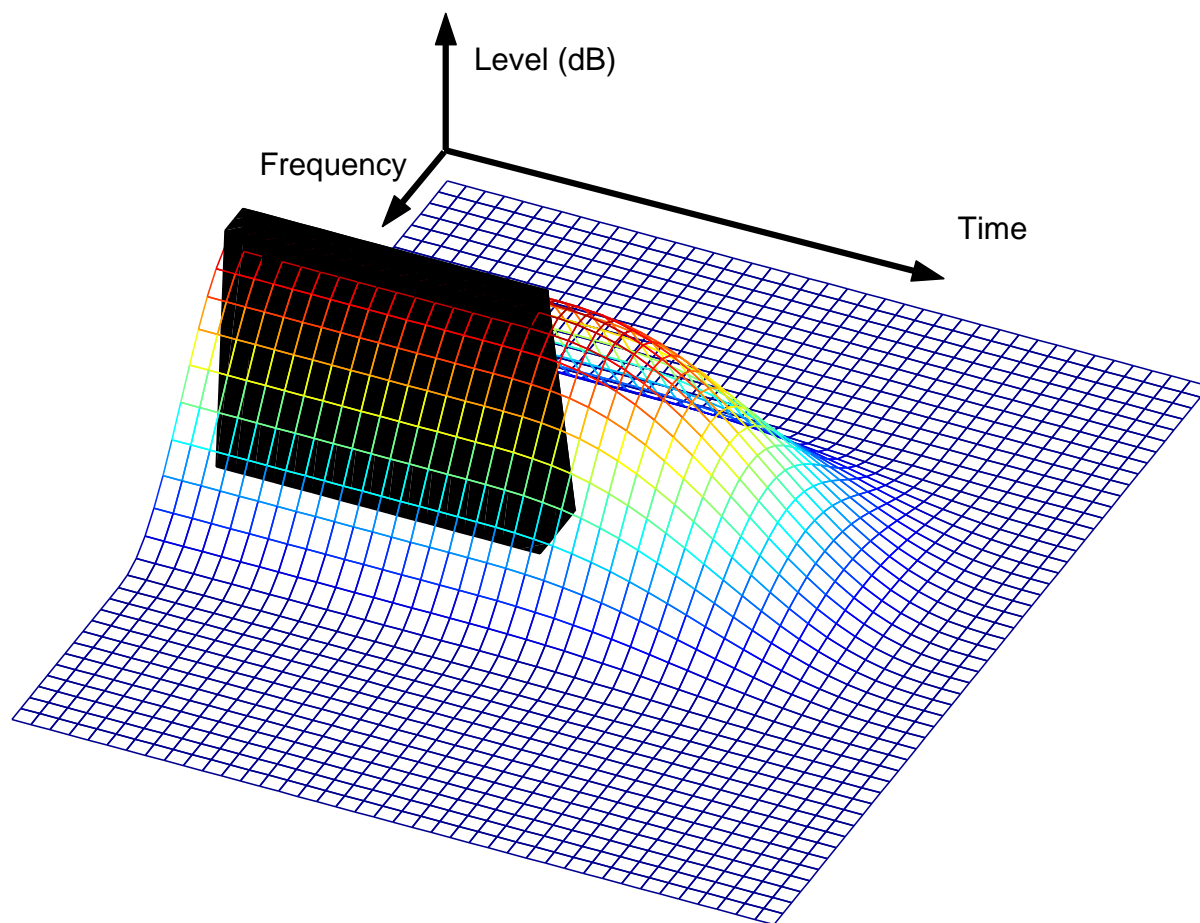


Figure 1.5: Overall masking threshold along time. The single black masker tone makes inaudible any tone of level under the mesh.

of bandwidths equal to critical bandwidths. It has then been admitted that the audibility frequency range, from around 20 Hz to 20,000 Hz, can be broken up into critical bands which are non linear, non uniform, and dependent on the heard sound. Tones within a critical band are difficult to differentiate for a human observer.

The notion of critical band can be explained by the masking of a narrow-band signal (sinusoidal tone of frequency f) by a wide-band noise. In figure 1.6, the narrow-band signal and the wide-band signal are distant in term of frequency so the noise doesn't affect the threshold of hearing of the frequency f .

With the noise centered on the sinusoidal signal as on figure 1.7, the threshold of hearing of the tone has been increased. As the noise bandwidth ΔF becomes larger, the threshold of hearing for the tone will also increase. There will come a point where an increase of ΔF gives no increase in the threshold. At this point, we can say that ΔF is the *critical bandwidth* centered at frequency f .

The critical bands are continuous such that for every audible frequency there is a critical band centered on that frequency. The bandwidth of a critical band centered on a frequency f is given

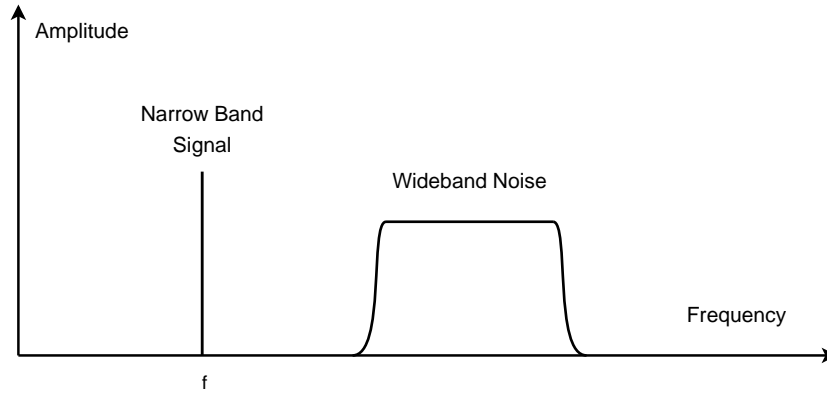


Figure 1.6: *The noise is distant from the signal, it does not affect the threshold of hearing of the sinusoidal tone.*

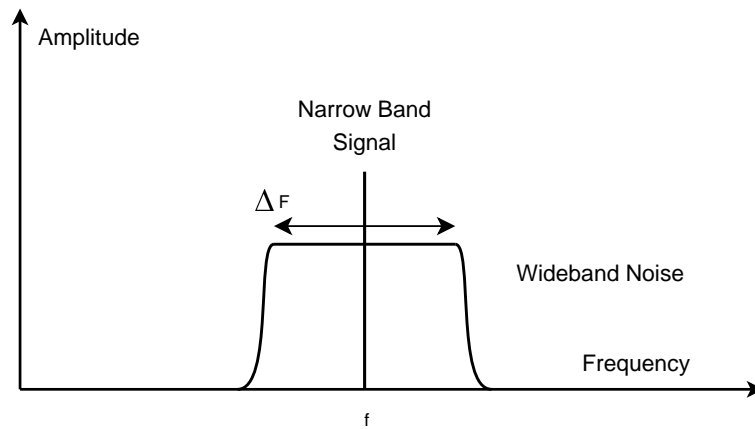


Figure 1.7: *The wideband noise is centered on the sinusoidal frequency, the tone threshold of hearing is increased.*

by

$$\Delta F(f) = 25 + 75 \left(1 + 1.4 \left(\frac{f}{1000} \right)^2 \right)^{0.69}$$

According to the definition of critical bands, a special psychoacoustic unit was introduced: the *bark*. It is a non linear scale on which one bark correspond to the width of one critical band. The rate in bark for a frequency f is given by

$$z = 13 \arctan(0.76f) + 3.5 \arctan \left[\left(\frac{f}{7.5} \right)^2 \right]$$

where z is the rate in bark and f the frequency in kHz.

The audible frequency range can be divided into consecutive frequency bands so that the center frequency of each band corresponds to an integer bark value as shown in table 1.1. The latter division of the audible frequency range does not mean that the critical bands have fixed boundaries. They depends on the sound heard. Two frequencies at 1900 Hz and 2100 Hz belong to

the same critical band because the critical bandwidth around 2 kHz is 300 Hz. The relationship between the bark scale and the frequency scale is approximately linear at low frequencies ($\leq 500\text{Hz}$) and logarithmic at higher frequencies (table 1.1).

Table 1.1: *Division of the audible frequency range into consecutive bands. Each band correspond to an integer bark.*

Band (Bark)	Lower (Hz)	Center (Hz)	Upper (Hz)	Band (Bark)	Lower (Hz)	Center (Hz)	Upper (Hz)
1	0	50	100	14	2000	2150	2320
2	100	150	200	15	2320	2500	2700
3	200	250	300	16	2700	2900	3150
4	300	350	400	17	3150	3400	3700
5	400	450	510	18	3700	4000	4400
6	510	570	630	19	4400	4800	5300
7	630	700	770	20	5300	5800	6400
8	770	840	920	21	6400	7000	7700
9	920	1000	1080	22	7700	8500	9500
10	1080	1170	1270	23	9500	10500	12000
11	1270	1370	1480	24	12000	13500	15500
12	1480	1600	1720	25	15000	19500	
13	1720	1850	2000				

1.2 General Structure of Perceptual Audio Encoders

The main known limitations of human hearing have been described in 1.1. These masking properties of hearing already appears to be kind of a natural reduction of audio content performed by the human auditory system. The main idea of perceptual audio coding is to exploit these limitations by removing audio components that cannot be heard. As a result we will save precious bits just by removing masked frequencies. An overview of the general structure of perceptual audio encoders is given in this section.

The key to audio compression is re-quantization of the digital signal and is a lossy process. Perceptual encoding techniques reduces the bit-rate in such a way that the noise introduced by quantization is inaudible. A perceptually lossless compression is thus achieved from a lossy scheme.

While various codecs use different techniques in the details, the underlying principle is the same for all, and the implementation follows the common plan illustrated on figure 1.9.

There are four major subsections which work together to generate the coded bitstream:

- The *Filterbank* performs a time to frequency mapping. It divides the audio into spectral components. This can be done by passing the input data through a bank of time domain

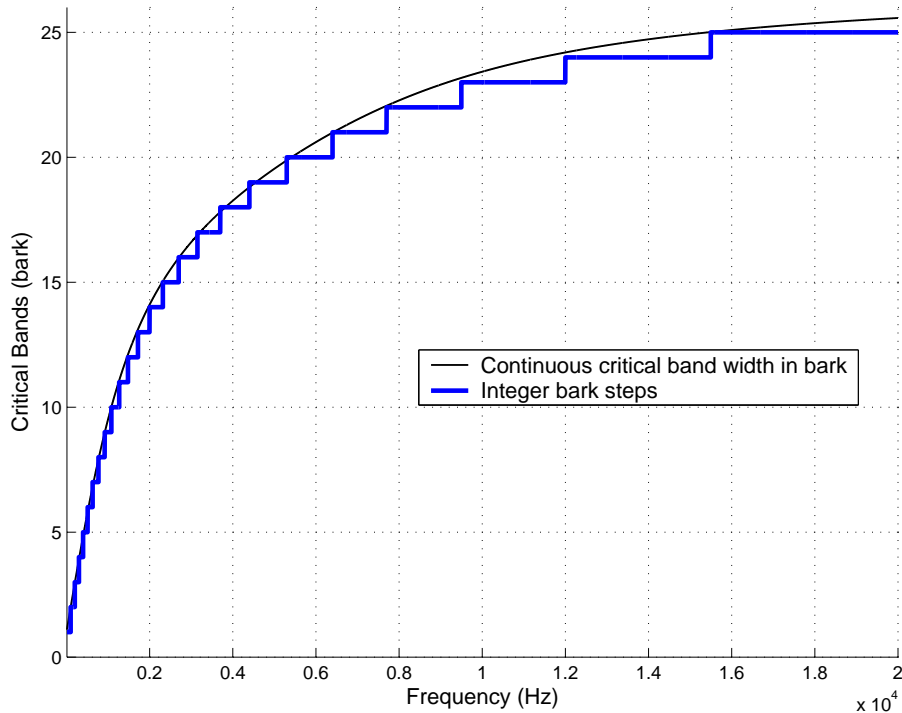


Figure 1.8: *Critical Bands and Bark Unit.*

filters (MPEG-1 Layer-1, MPEG-1 Layer-2), or computing the spectrum (MPEG-2 AAC), or both (MPEG-1 Layer-3). The frequency resolution must be at least such that it exceeds the width of the ear’s critical bands which is 100Hz below 500Hz and up to 4000Hz at higher frequencies (table 1.1) (ie: there should be at least one spectral component per band). The data output by the filterbank is in the format that will be used for the remainder of the encoding process. It is time-domain for MPEG-1 Layer-1, MPEG-1 Layer-2, and frequency-domain for MPEG-1 Layer-3, MPEG-2 AAC. The main part of the coded bitstream will be made of a quantized version of this data.

- The *Psychoacoustic* block models the human auditory system to compute the masking curve, under which introduced noise must fall. The spectrum is first calculated with a higher frequency resolution than the resolution of the filterbank. Then, the spectral values are partitioned into bands related to the critical-band widths. And finally, the masking level is computed for each band, considering the distribution of masker components in the band. All current MPEG encoders use a Fourier transform (FFT) to compute the spectrum.
- It is during the *Bit/Noise Allocation* process that the audio bit-rate is actually reduced. This reduction is achieved by re-quantizing the data. On one hand the quantization must be sufficiently coarse in order to fit the targeted bit-rate, and on the other hand the quantization noise must be kept under the limits set by the masking curve. The frequency representation of the audio previously computed by the filterbank is partitioned into the same frequency bands that were used in the psychoacoustic model. Bits are allocated to each band in respect to the amount of masking computed in the psychoacoustic block.

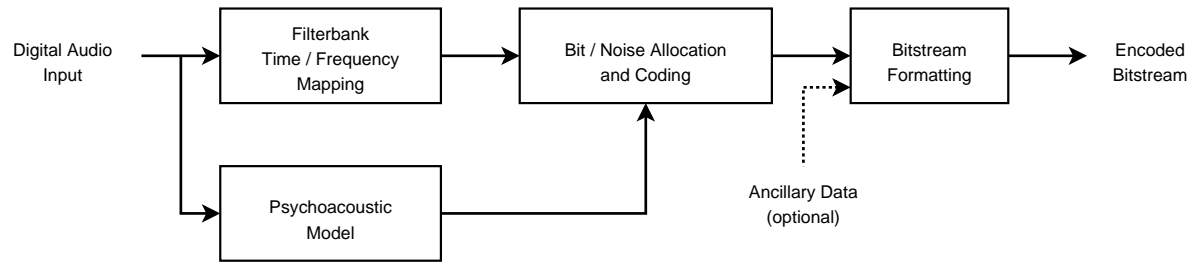


Figure 1.9: Overview of Perceptual Audio Encoders. The filter bank divides the input stream into multiple subbands of frequency. The psychoacoustic model simultaneously determines the overall masking threshold (1.1.2) for each subband. The bit/noise allocation block uses the masking threshold to decide how many bits should be used to quantize the subband to keep the audibility of the quantization noise minimal.

- Information necessary to reconstruct the original signal is added to the quantized values in the *bitstream multiplex* block to form the coded bitstream. Ancillary data can also be inserted to the stream at this stage.

1.3 MPEG-2 AAC Encoding Process Explained

Advanced Audio Coding is part of the MPEG-2 standard (ISO/IEC 13818-7) [1]. It is also known as MPEG-2 NBC (non backward compatible) because it is not compatible with MPEG Audio Layers 1,2 and 3. It was built on a similar structure to MPEG-1 Layer-3 and thus retains most of its features. MPEG-2 AAC is intended to provide very high audio quality at a rate of 64 kb/s/channel for multichannel signals (up to 48 channels). According to the definition of the International Telecommunication Union (ITU), compressed stereo audio has an indistinguishable quality from the original CD signal at a bit-rate of 128 kb/s.

The AAC algorithm makes use of a set of tools, some of which are required and others optional. These are Huffman coding, Non Linear Quantization and Scaling, M/S Matrixing, Intensity Stereo, Frequency Domain Prediction, Temporal Noise Shaping (TNS), Modified Discrete Cosine Transform (MDCT). The idea is to match specific application requirements and present performance/complexity tradeoffs. This have the additional advantage that it is possible to combine various components from different developers, taking the best pieces from each.

The standard describes three profiles in order to serve different requirements:

- The Main profile uses all tools available and delivers the best audio quality of the three profiles.
- The Low Complexity (LC) profile comes with a limited temporal noise shaping and without prediction to reduce the computation complexity and the memory requirement.
- The Scalable Sampling Rate (SSR) profile is a low complexity profile intended for use when a scalable decoder is needed.

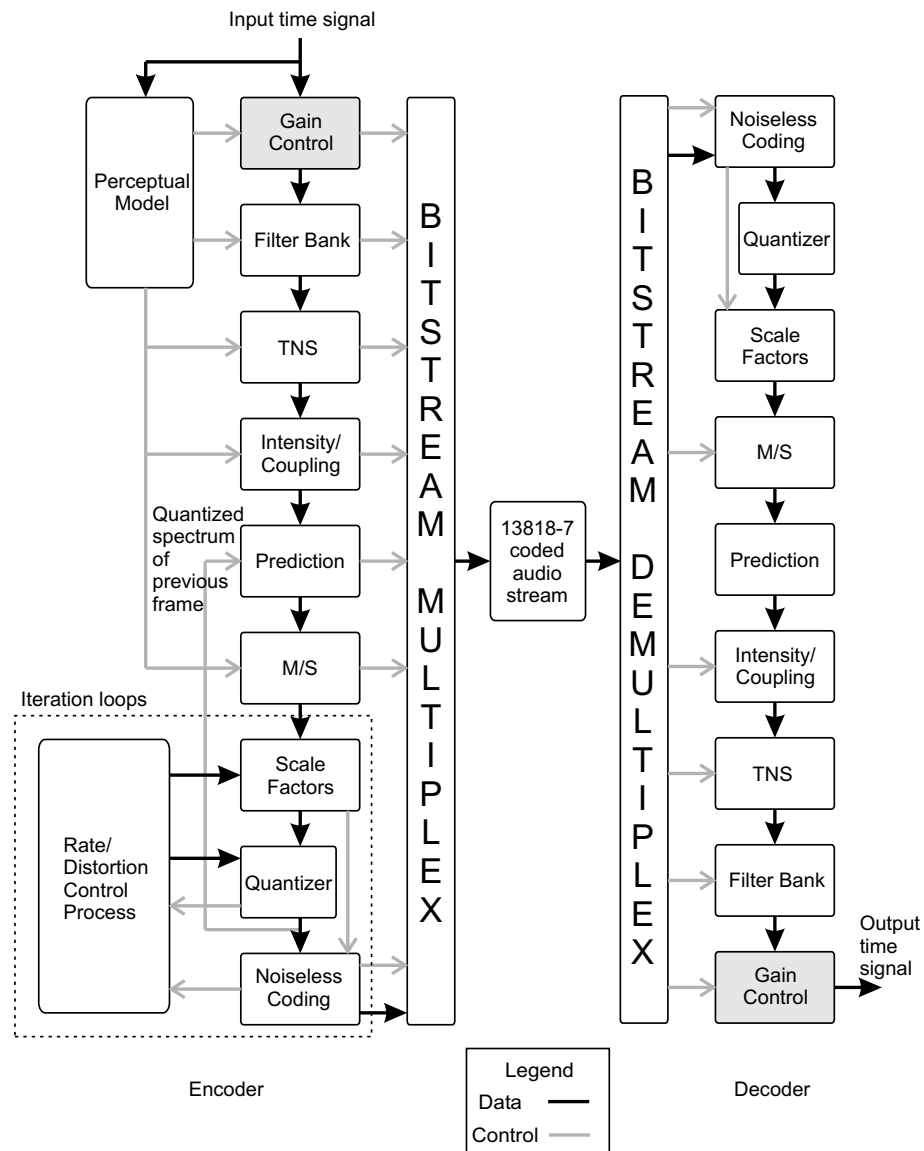


Figure 1.10: Block diagram of the MPEG-2 AAC Encoding-Decoding process.

Figure 1.10 is a block diagram of the MPEG-2 AAC encoding-decoding process.

Compared to the previous MPEG layers, AAC benefits from some important new additions to the coding toolkit:

- An improved filter bank with a frequency resolution of 2048 spectral components, nearly four times more than the layer 3.
- Temporal Noise Shaping, a new and powerful element that minimizes the effect of temporal spread. This benefits voice signals, in particular.
- A prediction module guides the quantizer to very effective coding when there is a noticeable signal pattern, like high tonality.
- Perceptual Noise Shaping allows a finer control of quantization resolution, so bits can be used more efficiently.

1.3.1 Filterbank

The filterbank performs a time to frequency mapping by computing the spectrum of the input signal with rather high frequency resolution compared to previous MPEG codecs. A MDCT transform is used to compute the spectrum. The AAC filterbank implements the following steps:

- Shift in 1024 new samples into 2048 FIFO buffer X.
- Window samples : for $i = 0$ to 2048 do $Z_i = C_i * X_i$, where C_i is one of the analysis window coefficient defined in the standard.
- Compute MDCT of 2048 windowed samples.
- Output 1024 frequency components.

Different factors comes into play in the design of the filter bank stage in AAC coding. Firstly we would like to optimally separate the different spectral components so that the perceptual coding gain can be maximized. Since we are performing short-time analysis of the signals, we would like to minimize the audibility of blocking artefacts both in terms of boundary discontinuities and pre-echo effects. The window shape plays an important role in the spectral separation of the signals and blocking artefacts. While no single window provides optimal resolution for all signals, AAC supports two different window shapes that can be switched dynamically. They are Kaiser-Bessel derived (KBD) window and sine shaped window. The KBD window achieves better stop band attenuation while the sine window has better pass band selectivity.

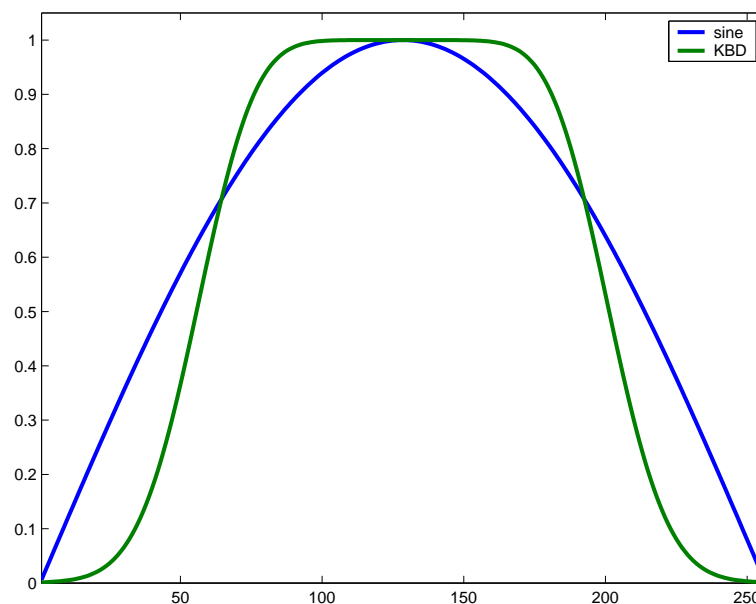


Figure 1.11: *Sine and KBD windows.*

The Modified Discrete Cosine Transform (MDCT) is a frequency transform based on the Discrete Cosine Transform (DCT), with the additional property of being lapped. Unlike other frequency

transforms that have as many outputs as inputs, a lapped transform has half many outputs as inputs. It is designed to be used on consecutive blocks of data where blocks are 50% overlapped. In addition to the energy-compaction of the DCT, this overlapping helps avoiding the aliasing due to block boundaries. This makes the MDCT suitable for signal compression applications. The n frequency components f_0, f_1, \dots, f_{n-1} are computed from the $2n$ input samples $x_0, x_1, \dots, x_{2n-1}$ by :

$$f_j = \sum_{k=0}^{2n-1} x_k \cos \left[\frac{\pi}{2n} \left(j + \frac{1}{2} \right) \left(k + \frac{1}{2} + \frac{n}{2} \right) \right]$$

A fast scheme for computing the MDCT using FFT is proposed in [4]. Thus, the MDCT can be computed using only a $n/2$ FFT and some pre and post rotation of the samples points.

AAC specifies two different MDCT block lengths : a long block of 1024 samples and a short block of 128 samples. Since there is a 50% overlap between successive transform windows, the window sizes are 2048 and 256 respectively. Figure 1.11 reports the shape of sine and KBD windows for blocks of 256 samples. Dynamic switching between long windows and short windows occurs during the encoding to adapt the time-frequency resolution to the input signal. The long block length allows greater frequency resolution for signals with stationary audio characteristics while the short block provides better time resolution for varying signals. In short block mode, eight short blocks replace a long block so that the number of MDCT samples for a frame of audio samples remains 1024. To switch between long and short blocks, a long-to-short window and a short-to-long window are used. Figure 1.12 shows the overlapping of the MDCT windows and the transition between short and long blocks.

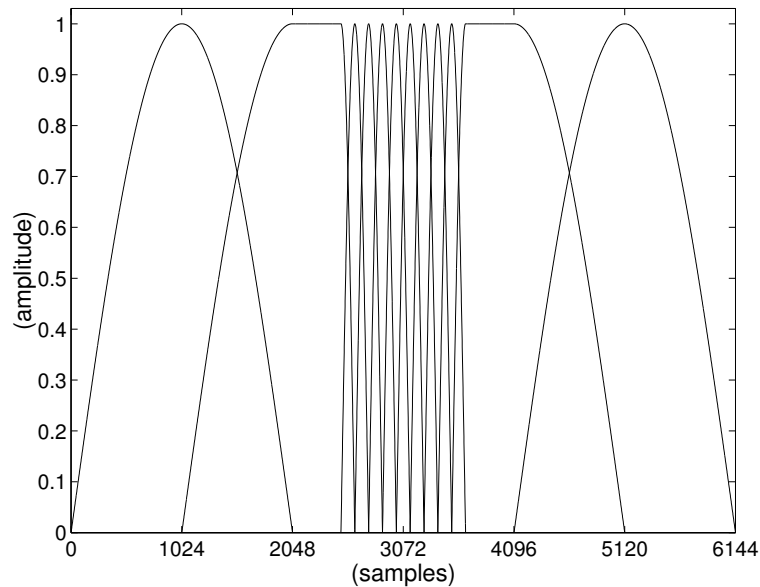


Figure 1.12: AAC Filterbank. *Overlapping MDCT windows and transition between long and short blocks.*

The decision of switching between long and short blocks and KBD and sine windows is made in the psychoacoustic model, according to the results obtained for previous frames. Finally, for further processing of the frequency components in the quantization part, the spectrum is

partitioned into so called scalefactor bands, related to the human auditory system critical bands (1.1.3).

1.3.2 Psychoacoustic model

Considerable freedom is given to designers in the implementation of the psychoacoustic model since the only requirement is to output a set of Signal to Masking Ratios, SMR_n , to be used in the bit/noise allocation stage. Nevertheless, a psychoacoustic model is proposed in Annex A of the MPEG-2 AAC standard. This model is based on "Psychoacoustic Model 2" that was found in MPEG-1 Layer-1,2, and 3 standards [5]. The following description of the psychoacoustic model proposed in the standard gives a good idea of what can be achieved by the psychoacoustic block. However, the complexity of the perceptual model is dependent on the output bit-rate requirements. If no restrictions are placed on the bit-rate, the psychoacoustic block may even be bypassed. The model is described for long blocks mode (1.3.1), but the short block mode works exactly in the same way, and the switching between long blocks and short blocks occurs simultaneously for the filterbank and the perceptual model.

The characteristics of the MDCT makes it inappropriate for the calculation of the masking threshold. This is mainly due to the fact that the basis functions of the transform are purely real. To overcome this limitation, the psychoacoustic model usually uses a Discrete Time Fourier Transform (DTFT). The input frame is the same 2048 samples frame that simultaneously pass through the filterbank block (1.3.1). A standard 2048 point Hann weighting is applied to the audio before the Fourier transform to reduce the edge effects on the transform window. The FFT is then computed and the polar representation of the complex frequency coefficients is calculated so that 2048 magnitude components r_ω and 2048 phase components f_ω are now available.

Both tonal (sinusoidal-like) components and noise-like components are usually present in an audio signal. Masking abilities of tonal and noise-like components differs considerably. Thus, it is essential to identify and separate these components in order to properly approximate the human auditory system in the calculation of the masking threshold . The model computes a tonality index (from 0 to 1) as a function of the frequency. This index gives a measure of whether the component is more tonal-like or more noise-like. It is based on the predictability of the component. A predicted magnitude \hat{r}_ω and phase \hat{f}_ω are linearly interpolated from the magnitudes and phases of the two previous frames :

$$\hat{r}_\omega = 2r_\omega [t - 1] - r_\omega [t - 2]$$

$$\hat{f}_\omega = 2f_\omega [t - 1] - f_\omega [t - 2]$$

And the so called, unpredictability measure c_ω , is calculated :

$$c_\omega = \frac{\left(\left(r_\omega \cos f_\omega - \hat{r}_\omega \cos \hat{f}_\omega \right)^2 + \left(r_\omega \sin f_\omega - \hat{r}_\omega \sin \hat{f}_\omega \right)^2 \right)^{0.5}}{r_\omega + |\hat{r}_\omega|}$$

The more predictable components are tonal components and thus will have higher tonality indices.

The spectrum is divided into partitions related to critical-band width of the human auditory system (1.1.3). Higher frequencies partitions are wider than lower frequencies partitions. For each partition, a single SMR has to be calculated. The masking threshold generated by a given signal spreads across its critical band. The model specifies a spreading function to approximate the noise masking threshold spread in the band by a given tonal component. This lead to the computation of a single masking threshold per band, taking into account the multitude of masking components within the band and their tonality indices. The absolute threshold of hearing is used as a lower bound for the threshold values. The signal to mask ratio (SMR) is computed as the ratio of the signal energy within the sub-band to the masking threshold for that sub-band.

Finally, a mapping from the bark scale used so far by the psychoacoustic model to the scalefactor bands of the filterbank is carried out and the bit allocation, between 0 and 3,000 bits, is calculated for each scalefactor band.

1.3.3 Temporal Noise Shaping

Temporal Noise Shaping (TNS), first introduced in 1996 [6], is a completely new concept in the area of time/frequency coding. It is designed to deal with problems often encountered in conventional coding scheme for signals that varies heavily in time, especially voice signals. While the quantization noise distribution is well controlled over frequency by the psychoacoustic model, it remains constant in time over a complete transform block. If the signal characteristics change abruptly within such a block, without leading to a switch to shorter blocks transform, this may lead to audible artifacts. With a long analysis filterbank window, the quantization noise may spread over a period of 46ms (assuming a sampling rate of 44,1 kHz), which is annoying when the signal to be coded contains strong components only in short parts of the analysis window (speech signals). The idea of TNS relies on time/frequency duality considerations. The correlation between consecutive input samples is exploited by quantizing the error between the unquantized frequency coefficients generated by the filterbank, and a predicted version of these coefficients. To achieve this the output coefficients of the filterbank (original spectrum of the signal) pass through a filter, and the filtered signal is quantized and sent in the bitstream. The coefficients of the filter are also quantized and transmitted to the bitstream. They are used in the decoder to undo the filtering operation. A more rigorous explanation of the temporal noise shaping theory was published by Jürgen Herre in [7]. In this paper, the combination of filterbank and TNS is described as a ‘continuously adaptive filterbank’ opposed to the classic ‘switched filterbank’ used so far.

1.3.4 Joint Stereo Coding

Joint stereo increases the compression efficiency by reducing irrelevancies in the right and the left channel of a stereo signal. Psychoacoustic results show that above 2 kHz, and within each critical

band the perception of stereo is based on the energy-time envelope of the signal. Magnitude and phase are less important. MPEG-2 AAC supports two types of stereo redundancy coding: Intensity Stereo Coding, and Middle/Side Stereo Coding. Both types exploit this property of the human auditory system.

In intensity stereo (IS) coding mode (also called channel coupling), the encoder codes a single summed signal for upper frequency scalefactor bands, instead of sending independent left and right channel for each subband. At the decoder stage, higher frequency bands are reconstructed such that the spectral shape of the right and left channel is the same, but the magnitude differs. Up to 50% data reduction is possible in high frequency bands, but some audible distortions may occur, which make the intensity stereo useful only for low bit-rate.

For coding at higher bitrate, only Middle/Side Stereo (MS) coding should be used. In this mode, the encoder transmits the left and right channel in certain frequency ranges as middle (sum of left and right) and side (difference of left and right).

Care should be taken with the use of joint stereo coding as it can be destructive and not suitable for certain types of signals.

1.3.5 Prediction

Frequency domain prediction is a tool introduced in the standard to enhance the compression efficiency of stationary parts of the signal or very predictable components like high tonality. This tool is only supported in MPEG-2 AAC Main Profile and only in long block mode since it is where stationary signals can be found. A prediction gain is calculated, and the decision of using a predicted value instead of the real value is based on this gain. The processing power of frequency domain prediction and its sensitivity to numerical error makes this tool difficult to use on fixed point platforms... Moreover, the backward structure of the predictor makes the bitstream quite sensitive to transmission error.

1.3.6 Quantization and Noiseless Coding

After the parallel filterbank and psychoacoustic processes, the quantization and coding process is the next major block of the perceptual encoder. Its goal is to achieve a representation of the spectral data from the filterbank which uses as few bits as possible, and which at the same time introduces as little perceptible distortion as possible. To do so the signal is quantized in the frequency domain and the total bit pool is allocated dynamically depending on the energy of each spectrum component and its relevancy.

Quantizing the data firstly allows a reduction of the bit-rate at the cost of introducing quantization noise. This is the basis of all lossy audio compression schemes. With perceptual encoding, the idea is to control the quantization precision according to the masking ability of the signal estimated in the perceptual model. Then, noiseless coding achieves further compression in a lossless way by reducing the redundancies in the quantized representation of the data.

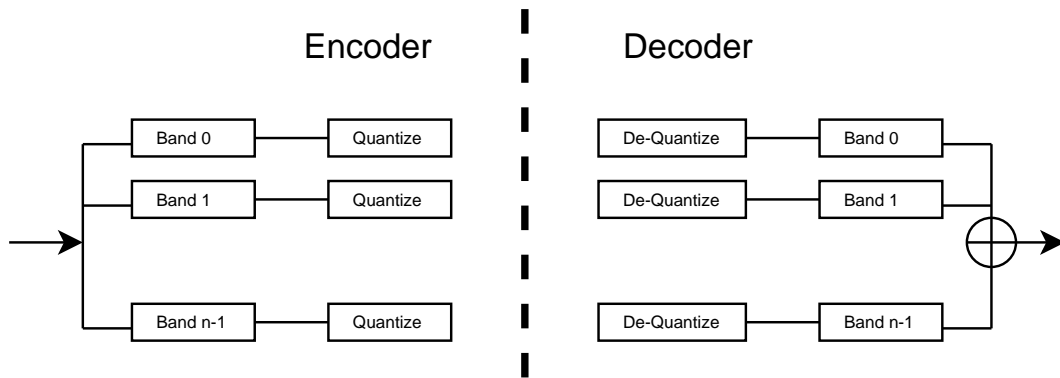


Figure 1.13: *Quantization role in the encoding process.*

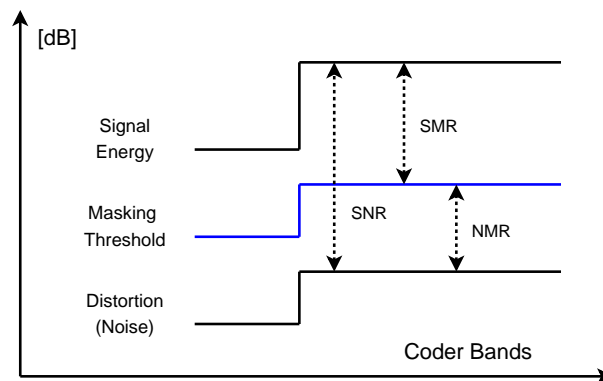


Figure 1.14: *SNR, SMR & NMR.*

Figure 1.14 introduces a number of important terms in the context of perceptual coding which are typically evaluated within groups of spectral coefficients in a coder:

- The Signal-to-Noise-Ratio (SNR) denotes the ratio between signal and quantization noise energy and is a commonly used distortion measure based on a quadratic distance metric. Please note that this measure in itself *does not* allow predictions of the subjective audio quality of the decoded signal. Clearly, reaching a higher local SNR in the encoding process will require a higher number of bits.
- The Noise-to-Mask-Ratio (NMR) is defined as the ratio of the coding distortion energy with respect to the masking threshold and gives an indication for the perceptual audio quality achieved by the coding process. While it is the goal of a perceptual audio coder to achieve values below 0 dB ('transparent coding'), coding of 'difficult' input signals at very low bitrates is likely to produce NMR values in excess of this threshold, i.e. a perceptible quality degradation will result from the coding/decoding process.
- The Signal-to-Mask-Ratio (SMR) describes the relation between signal energy and masking threshold in a particular coder band. This parameter significantly determines the number of bits that have to be spent for transparent coding of the input signal. As one extreme case, if the signal energy is below the masking threshold, no spectral data needs to be transmitted at all and there is zero bit demand. A generalization of this concept is called

perceptual entropy which provides a theoretical minimum bit demand (entropy) for coding a signal based on a set of perceptual thresholds.

Since the quantization step size of each frequency component needs to be transmitted in the bitstream for decoding, it is necessary to group the data into bands called scalefactor bands, to reduce the amount of side information required. While simpler coders use a uniform quantizer which produces the same amount of quantization noise ($q^2/12$ where q is the quantization step) for every coefficient in the scalefactor band, AAC quantization is non-uniform. A power law quantizer is used, which tends to distribute the quantization noise toward coefficients with higher amplitude, where the masking ability is better.

$$X_{quant}(k) = \frac{X(k)^{\frac{3}{4}}}{A(s)}$$

where $A(s)$ is the scalefactor for the subband s .

In doing so, some additional noise shaping is already built in the quantization process. The scaled values are quantized at a fixed quantization step size so that the quantization resolution is controlled by the scalefactors. Hence, the only side information value to send to the decoder is the set of scalefactor values.

In contrast to quantization, Noiseless Coding is a lossless process. AAC achieves rather high compression gain by using Huffman coding. Huffman coding is a popular entropy coding technique introduced in 1952 [11]. Its goal is to provide a representation of the quantized data with as few redundancies as possible. The idea is to assign variable length binary representations to each data coefficient in the frame, giving the most frequently occurring coefficient the shortest binary code word, and the least frequently occurring coefficient the longest code word. Higher compression gain can be achieved using multi-dimensional Huffman coding. In this approach, several data coefficients are combined into a vector which is coded using Huffman coding. MPEG-2 AAC comes with several Huffman code books, and a very flexible mechanism allowing the assignment of a specific Huffman code table to each scalefactor band. Thus, efficient coding can be achieved when the local characteristics of the coefficients are changing.

The actual structure of AAC quantization and coding block is referred to as Noise Allocation. It can be seen as an iteration process which does not control the number of bits allocated to each band, but only the two global parameters of the encoder: the amount of quantization noise introduced in each band, and the total number of bits allocated to the frame. The noise allocation based encoding process described in the informative annex part of the MPEG-2 AAC standard [1] is composed of two nested loops : the outer loop called the distortion control loop, and the inner loop called the rate control loop (figure 1.15).

In the distortion control loop (outer loop), the quantization precision is first set to an equal value corresponding to a white quantization noise for all bands. After quantization, the noise spectrum is computed band by band by comparing the non-quantized scaled values and the quantized values. In the band where quantization noise exceeds the masking threshold specified

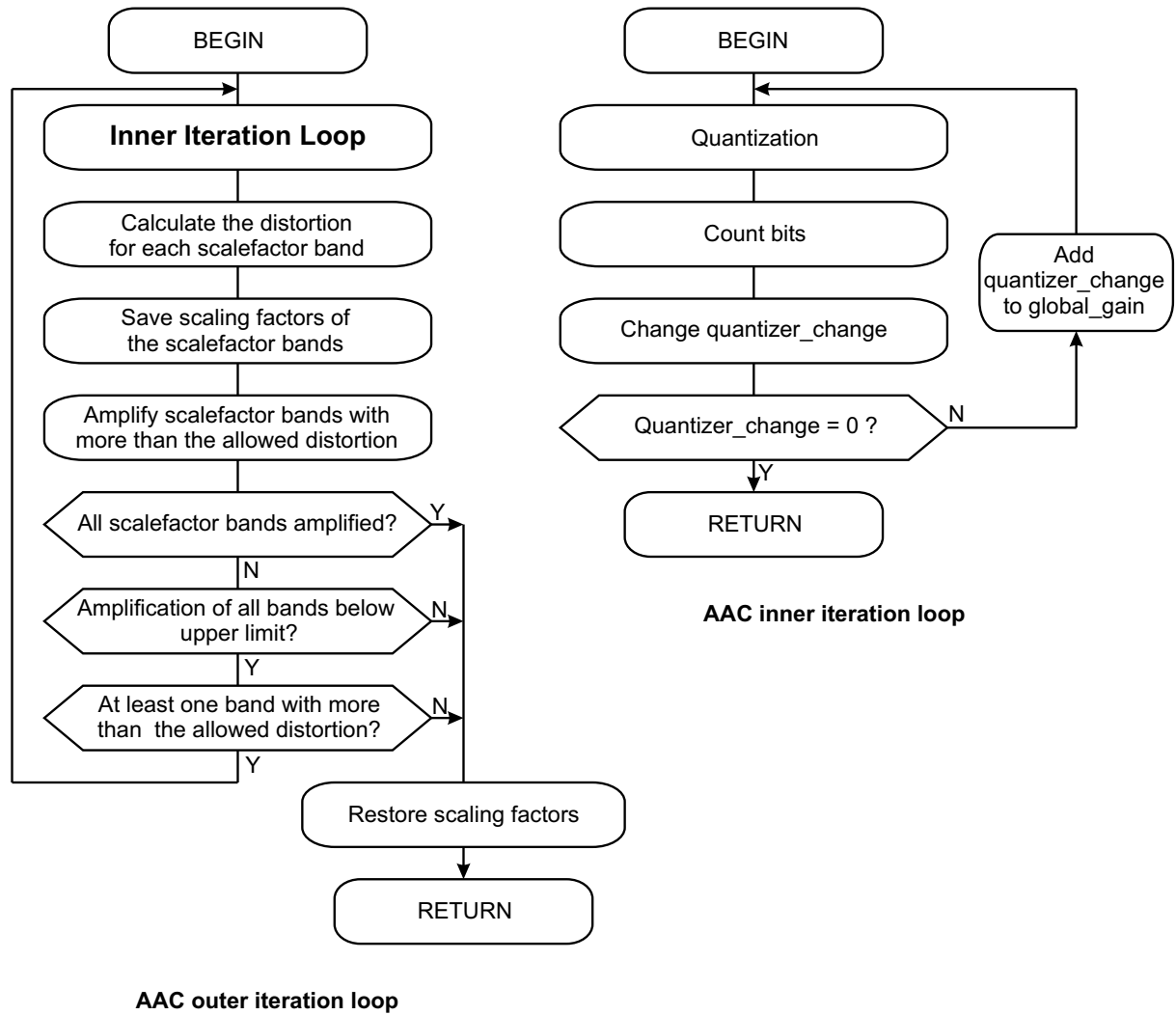


Figure 1.15: AAC Filterbank. Block diagram of MPEG-2 AAC noise allocation loop.

by the psychoacoustic model, the quantization precision is increased by increasing the value of the scalefactor.

The non uniform quantization is actually performed in the rate control loop (inner loop). The calculation of the number of bits using Huffman tables is also done in this loop.

Three extra conditions are adopted to stop the iterative process:

- The quantization noise in all scalefactor bands is below the maximum allowed distortion.
- The next iteration would cause any of the scalefactors to be over the maximum value allowed in the standard.
- The next iteration would cause all the scalefactors to be increased.

Even though this approach leads to the optimal audio quality for a given bit-rate, it is often slow to converge. An important part of the processing power is consumed at the noise allocation stage.

The noise allocation method described above is a method for coding at constant bit-rate. This usually results in a time dependent quality of the coded audio. AAC is able to deliver a so-called *Constrained Variable Rate* bit-rate, which approaches a constant quality output over time. This is achieved by using the concept of a *bit reservoir*. If a frame is easy to code, then not all the available bits are used, but some bits are put into a bit reservoir. If a frame needs more than the allowed number of bits to meet the perceptual quality requirements computed by the psychoacoustic model, the extra bits are taken from the bit reservoir. The maximum deviation allowed for the bit demand is constrained and defines the size of the bit reservoir. Thus, this approach can be seen as local variation in bitrate and helps the coder during times of peak bit demands, e.g. for coding of transient events, while still maintaining a constant average bitrate as required for applications.

1.3.7 Bitstream Formatting

The role of the bitstream formatting stage of the encoder is to multiplex all the data to be transmitted to the decoder into a 13818-7 coded bitstream as described in the standard. The main data to be transmitted the bitstream are the Huffman mapping of the quantized frequency components. Scalefactors are also transmitted as side information. Some ancillary data can be added to the stream.

Chapter 2

Basis of Fixed Point Arithmetic

2.1 Introduction

There are several ways to represent real numbers on computers. Fixed point places a radix point somewhere in the middle of the digits, and is equivalent to using integers that represent portions of some unit. For example, one might represent 1/100ths of a unit; with four decimal digits, 10.82 or 00.01 can be represented.

Fixed point has a fixed window of representation, which limits it from representing very large or very small numbers. Floating-point, on the other hand, employs a sort of ‘sliding window’ of precision which is more appropriate to the scale of numbers.

2.1.1 Floating Point Number Representation

The floating point number system is similar to the *scientific notation*, i.e., $x = \pm s \times b^e$ (e.g., -0.3662×10^{-3}). All floating point number systems represent a real number x as

$$fl(x) = \pm s \times b^e = \pm \left(\frac{s_1}{b^1} + \frac{s_2}{b^2} + \dots + \frac{s_p}{b^p} \right) \times b^e = \pm (s_1 s_2 \dots s_p)_b \times b^e,$$

where s is called the significand, e is called the exponent, p is the precision and the number of digits in the significand, and b is called the base. For example, the decimal number 321.456 has the floating point representation $+ .321456 \times 10^3$.

Each digit s_i of the significand is in the range $0 \leq s_i \leq b - 1$. The exponent must lie in the range $e_{min} \leq e \leq e_{max}$. Both the significand and the exponent are represented as integers.

A floating point number system is thus characterized by 4 parameters:

- The base b .
- The precision p .
- The minimum of the exponent e_{min} .
- The maximum of the exponent e_{max} .

The magnitude of floating point numbers have the range $.min(s) \times b^{e_{min}} \leq |x| \leq .max(s) \times b^{e_{max}}$. Most computers use Floating Point Units (FPU) to deal with floating point calculations. FPUs, also called numeric coprocessors, are special circuitry with a specific set of instructions to carry large mathematical operations out quickly.

2.1.2 The need for fixed point algorithms

However, many devices including small microcontrollers, DSP, and configurable devices (FPGA) do not have floating point coprocessors. As these devices use fixed point arithmetic, computations can be managed by fixing the position of the decimal point and using integer operations. A wide range of numbers available with floating point representation are then lost. A description of the fixed point representation of real numbers that we use for the fixed point implementation of the encoder is given in this section. The basic arithmetic operations using this representation are also explained here.

2.2 Fixed Point Numerical Representation

Recall that an N -bit word has 2^N possible states and that the rational numbers are the set of numbers x such that $x = a/b$ where $a, b \in \mathcal{Z}$ and $b \neq 0$. We can consider the subsets of rational numbers for which $b = 2^n$. And we can further constrain these subsets to elements that have the same number N of bits and that have the binary point at the same position (binary point is fixed). These representation sets contain numbers that can be exactly represented by the integer a . They are called fixed point representations.

2.2.1 Unsigned Fixed Points

A N -bit word interpreted as an unsigned fixed point can represent positive values from a subset P of 2^N elements given by :

$$P = \left\{ \frac{p}{2^b} \mid 0 \leq p \leq 2^N - 1, p \in \mathcal{N} \right\}$$

Such a representation is denoted $U(a, b)$ where $a = N - b$. In this representation, the first a bits (counting from left to right) represent the integer part of the rational number, while the last b bits ($a + b = N$) represent the fractional part. Thus, a is often called the *integer word length (iwl)*, b the *fractional word length (fwl)*, and N the *word length (wl)*. The value of a number $x = x_{N-1}x_{N-2} \dots x_1x_0$ of the subset $U(a, b)$ is given by :

$$x = \frac{1}{2^b} \sum_{i=0}^{N-1} 2^i x_i$$

The $U(a, b)$ interpretation of a N -bit word can represent numbers from 0 to $\frac{(2^N - 1)}{2^b}$. As shown in the examples below, a can be positive, negative or equal to 0. In the case of $a = 0$, the unsigned fixed point rational representation is identical to the N bits integer representation.

Example 1 : $U(5, 3)$ is a 8-bit representation. The value 1000 1010 is :

$$\frac{1}{2^3} (2^1 + 2^3 + 2^7) = 17.25$$

Example 2 : $U(-3, 19)$ is a 16-bit representation. The value 0010 1100 1011 1100 is :

$$\frac{1}{2^{19}} (2^2 + 2^3 + 2^4 + 2^5 + 2^7 + 2^{10} + 2^{11} + 2^{13}) = 0.02184295654296875$$

Example 3 : $U(16, 0)$ is a 16-bit representation identical to the 16-bit integer representation. The value 0001 1100 1011 1100 is :

$$\frac{1}{2^0} (2^2 + 2^3 + 2^4 + 2^5 + 2^7 + 2^{10} + 2^{11} + 2^{12}) = 7356$$

Example 4 : The value 3.625 is mapped into the $U(2, 6)$ representation (8-bit) by :

$$3.625 \times 2^6 = 232 = 1110 1000$$

Example 5 : The value 2.786 can not be represented exactly in the $U(2, 6)$ representation. There is a loss of accuracy due to the limited precision of the fixed point representation. The closest representation is given by :

$$\text{int} (2.786 \times 2^6) = \text{int} (178.304) = 178 = 1011 0010$$

The exact value represented by 1011 0010 is in fact $\frac{178}{2^6} = 2.78125$

2.2.2 Two's Complement Fixed Points

The two's complement representation is a very convenient way to express negative numbers. The two's complement of the N -bit word x is determined by inverting every bits of x and adding 1 to the inverted word.

Example: The two's complement of 0000 0101 is 1111 1011.

A nice feature of the two's complement representation of negative numbers is that the normal rules used for binary integer addition still work. Furthermore, it is also easy to negate any number and the leftmost bit gives the sign of the number (0 = positive, 1 = negative). Since a fixed point value is an integer representation of a rational number, the two's complement operation for fixed points is exactly the same as for integers. The 2^N possible values represented by an N -bit word interpreted as a signed two's complement fixed point are the values of the subset P such that :

$$P = \left\{ \frac{p}{2^b} \mid -2^{N-1} \leq p \leq 2^{N-1} - 1, p \in \mathcal{Z} \right\}$$

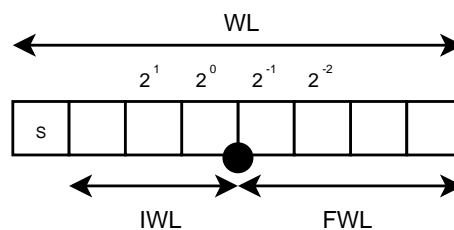
Similarly to the unsigned representation, this representation is denoted $A(a, b)$, where $a = N - b - 1$. The most left bit is often referred to as the *sign bit*, a the *integer word length (iwl)*,

Table 2.1: Three-bit two's-complement binary fixed point numbers.

Binary	Decimal
000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1

b the *fractional word length* and N the *word length (wl)*(figure 2.1). The value of a number $x = x_{N-1}x_{N-2} \dots x_1x_0$ of the subset $A(a, b)$ is given by :

$$x = \frac{1}{2^b} \left[-2^{N-1}x_{N-1} + \sum_{i=0}^{N-2} 2^i x_i \right]$$

**Figure 2.1:** Representation of signed fixed point numbers.

The range of signed fixed point numbers is $[-2^{IWL-1}; 2^{IWL-1} - 2^{-FWL}]$ which are the maximum positive and negative values and the precision is 2^{-FWL} (difference between successive values).

example 1 : $A(9, 6)$ is a 16-bit representation. The value 1000 1010 0011 0001 is :

$$\frac{1}{2^6} (2^0 + 2^4 + 2^5 + 2^9 + 2^{11} - 2^{15}) = -471.234375$$

The value 0000 1010 0011 0001 is :

$$\frac{1}{2^6} (2^0 + 2^4 + 2^5 + 2^9 + 2^{11}) = 40.765625$$

example 2 : The value -235.625175 is mapped into the $A(9, 6)$ representation (16-bit) by :

$$\text{int}(-235.625175 \times 2^6) = \text{int}(-15080.0112) = -15080 = 1100 0101 0001 1000$$

Figure 2.2 illustrate several ways to imagine the two's complement fixed point representation of signed numbers.

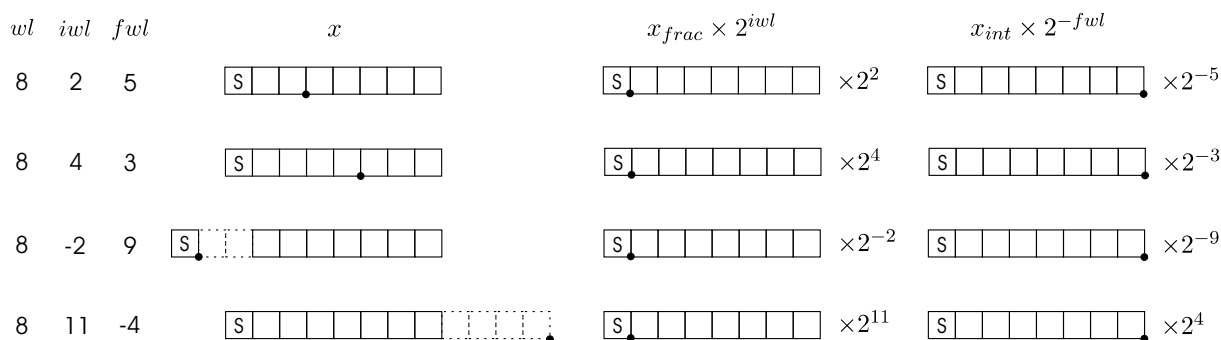


Figure 2.2: Three ways to represent the two's complement fixed point representation of signed numbers. The bit pattern is the same for the three cases.

2.3 Basic Fixed Point Arithmetic Operations

2.3.1 Addition

The addition algorithm for signed and unsigned fixed point numbers is the same as for integers. The two operands x and y must have the same word length N in order to be added. Furthermore, they must be aligned to the same integer word length. In other words, x and y must both belong to the same subset $U(a, b)$ or $A(a, b)$ where $U(a, b)$ and $A(a, b)$ refer to the unsigned and two's complement fixed point representation described above. The result of the addition of two N -bit numbers is an $(N + 1)$ -bit number due to a carry from the left most digit. In most applications, it is suitable to keep the size of the variables constant. Thus, when an addition of two N -bit numbers is performed, the result is expected to fit in a N -bit register.

One solution is to keep only the N less significant bits (lsbs) of the $(N + 1)$ -bit result as shown in figure 2.3.

$\begin{array}{r} (-1) \quad 1110 \\ (7) \quad +0111 \\ \hline \cancel{X}0110 \end{array}$	$\begin{array}{r} (-8) \quad 1001 \\ (-6) \quad +1010 \\ \hline \cancel{X}0011 \end{array}$	$\begin{array}{r} (4) \quad 0100 \\ (2) \quad +0010 \\ \hline \cancel{X}0110 \end{array}$
\uparrow Valid 4 bits result	\uparrow Overflow	\uparrow Valid 4 bits result

Figure 2.3: Examples of 4-bit addition. The result is made of the 4 lsbs of the 5 bits temporary result.

There are three cases to consider for two's complements addition of two numbers:

- 1 - If both x and y are > 0 and the result has a sign bit of 1, then overflow has occurred. Otherwise the result is correct.
- 2 - If both x and y are < 0 and the result has a sign bit of 0, then overflow has occurred. Otherwise the result is correct.
- 3 - If x and y have different signs, overflow cannot occur and the result is always correct.

The result, if valid, is always exact. There is no loss of accuracy due to the addition operation. However, the major restriction of this system is that an overflow can occur if the operands have the same sign. This means that the range of values allowed to be represented by a N -bit number has to be restricted.

A more efficient way to perform additions of fixed point numbers is to take into account the $N + 1$ bits of the temporary result and the integer word length (iwl) of the result as shown in figure 2.4.

<p>Example 1: signed variables : $wl = 8$ $x = 2.3125$ $iwl_x = 2$ $fwl_x = 5$ $y = 3.40625$ $iwl_y = 2$ $fwl_y = 5$ $iwl_z = 3$ $fwl_z = 4$</p>	<p>Example 2: signed variables : $wl = 8$ $x = 1.125$ $iwl_x = 2$ $fwl_x = 5$ $y = 2.75$ $iwl_y = 3$ $fwl_y = 4$ $iwl_z = 2$ $fwl_z = 5$</p>
<p>1- x and y are already aligned : No shift needed</p>	<p>1- x is right shifted by $3 - 2 = 1$ and iwl_x is now 3</p>
<p>2- $\begin{array}{r} (2.3125) \quad 01001010 \\ (3.40625) \quad +01101101 \\ \hline \quad \quad \quad 010110111 \end{array}$ $tmp = 010110111$</p>	<p>2- $\begin{array}{r} (1.125) \quad 00100100 \\ (2.75) \quad +00101100 \\ \hline \quad \quad \quad 000111110 \end{array}$ $tmp = 000111110$</p>
<p>3- tmp is left shifted by $2 + 1 - 3 = 0$</p>	<p>3- tmp is left shifted by $3 + 1 - 2 = 2$ $tmp = 011111000$</p>
<p>4- $z = tmp[8 \text{ downto } 1] = 01011011 = 5.6875$</p>	<p>4- $z = tmp[8 \text{ downto } 1] = 01111100 = 3.875$</p>

Figure 2.4: Examples of 8-bit addition. z is always the best possible approximation of the result, and there is no need to restrict the range of values of x and y to avoid overflow.

The steps we use to perform an addition are the followings :

- 1 - Either x or y or neither (but not both x and y) is right shifted to align the operands to the same iwl . The common iwl will be the biggest of the two original $iwls$.

```
x >> (MAX(iwlx, iwly) - iwlx); y >> (MAX(iwlx, iwly) - iwly); iwlx =
MAX(iwlx, iwly); iwly = MAX(iwlx, iwly);
```

- 2 - The addition is performed and the result is stored into a temporary ($N + 1$)-bit register.

```
tmp = x + y;
```

- 3 - The temporary register is shifted so that the most significant bit is a 1 if the result is positive and a 0 if the result is negative. This way, the final register will be used as efficiently as possible.

```
tmp << MAX(iwlx, iwly) + 1 - iwlz;
```

The $+1$ makes room for the carry. The shift is either a null shift or a left shift because $iwl_z \leq \max(iwlx, iwly) + 1$.

- 4 - The N bits of the final result are the N msbs of the temporary register.

```
z = tmp[N downto 1];
```

2.3.2 Multiplication

Similarly to the addition, the multiplication algorithm for signed and unsigned fixed points is the same as for integers. It is a succession of left shift of the multiplicand x and a sum of the partial products. The sign of each partial product is duplicated to the left, and if the multiplier y is negative, the two's complement of x must be added to the sum of partial products. The computation of an N -bit multiplication requires the computation of N N -bit additions for the sum of the partial products, and N shift operations. Thus, an N -bit multiplication is about N times more complex to perform than an N -bit addition. Figure 2.5 illustrates the multiplication algorithm with two 4-bit multiplication examples.

<p>Example 1: signed integers : $N = 4$ bits $x = -3 = 1101$ $y = 5 = 0101$</p> $ \begin{array}{r} 1101 \\ * 0101 \\ \hline 11111101 \\ 00000000 \\ 11110100 \\ 00000000 \\ \hline 11110001 = -15 \end{array} $	<p>Two's complement of -6</p> <p>→</p>	<p>Example 2: signed integers : $N = 4$ bits $x = -6 = 1010$ $y = -3 = 1101$</p> $ \begin{array}{r} 1010 \\ * 1101 \\ \hline 11111010 \\ 00000000 \\ 11101000 \\ 11010000 \\ 01100000 \\ \hline 00010010 = 18 \end{array} $
--	--	---

Figure 2.5: Examples of 4-bit multiplication. In example 2, the multiplier y is negative so the two's complement of the multiplicand x must be added to the sum of partial products.

The two operands x and y must have the same word length N but they do not need to have the same integer word length. The result of the multiplication of two N -bit numbers is a $2N$ -bit number. As in the addition case, we need to get a N -bit result from this number. This result must be as accurate as possible. We use the following steps to perform a multiplication:

- 1 - The multiplication is performed and the result is stored into a temporary $2N$ -bit register.

```
tmp = x * y;
```

- 2 - The temporary register is shifted so that the most significant bit is a 1 if the result is positive and a 0 if the result is negative. In doing so, the final register will be used as efficiently as possible.

```
tmp << iwlx + iwly + 1 - iwlz;
```

The iwl of the result of a multiplication is the sum of the two iwl s and the $+1$ makes room for the carry. The shift is either a null shift or a left shift because $iwl_z \leq iwl_x + iwl_y + 1$.

- 3 - The N bits of the final result are the N msbs of the temporary register.

```
z = tmp[(2N-1) downto N];
```

Note that a multiplication by a power of two is a simple shift of the binary point and requires no actual operation.

2.4 Position of the binary point and error

It is important to understand that the position of the binary point (ie: integer word length) is not the same for every variable in a given algorithm. Simulations must be carried out to determine the optimal integer word length of every variables. Then, the appropriate shifting rules must be respected for each addition or multiplication in the algorithm. There are two types of error in fixed point number representation:

- Overflow error occurs when the result of an operation exceeds the maximum representable value in the system. Overflow errors reflect an insufficient range of the system. The range of a representation system gives the limits of representable numbers in this system.
- Quantisation error occurs when the result of an operation is not representable by the system. Quantization errors reflects an insufficient precision of the system. The precision gives the distance between successive representable numbers in this system.

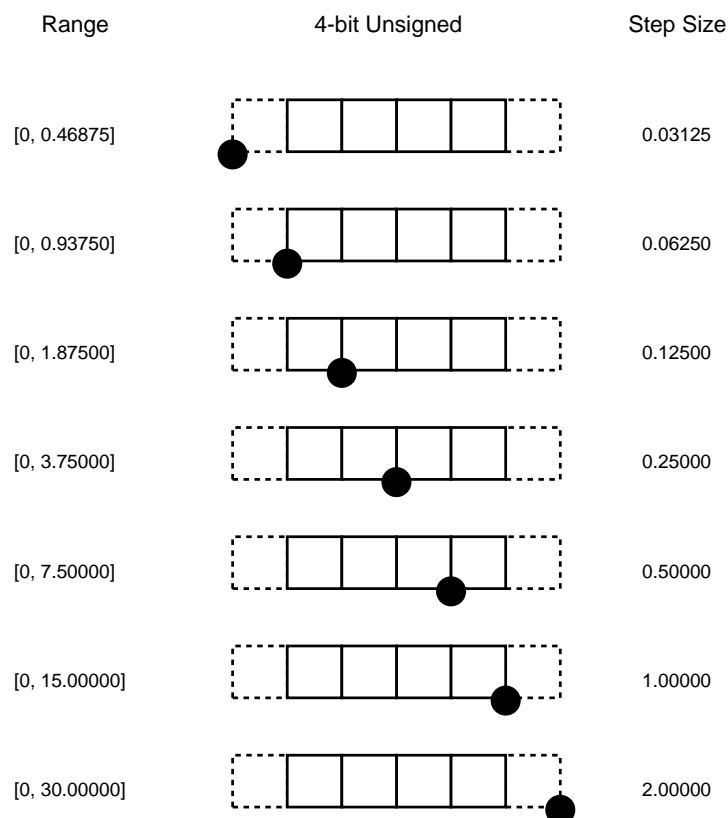


Figure 2.6: Range and precision (step size) of the 4-bit unsigned fixed point system.

Figure 2.6 shows how the range and precision of a floating point number system is dependent of the position of the binary point. In order to choose the optimal binary point for a specific fixed

point data type and a particular operation, one must pay attention to the precision and range in order to reduce errors.

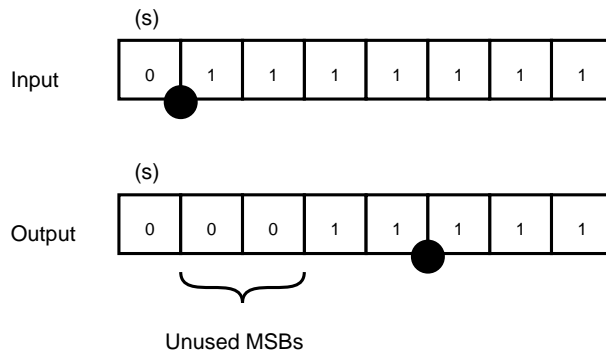


Figure 2.7: *Determination of the best point position.*

The basic requirement of porting a floating point algorithm to a fixed point algorithm is to set the binary point positions of every variable. To do so several tools have been developed in the IHL laboratory by Keith Cullen to optimize the position of the radix of the variables used in the MPEG2-AAC encoder. He carried out many simulations using real audio data to find the better position for the binary point of each variable of the encoder. The first requirement is to locate binary points such that the range of each variable is high enough to prevent overflow. To balance this statement, the range have to be minimized to ensure maximum precision for a given word length. This operation goes through the search of unused bits in the output of each stage (figure 2.7). In practice, the maximum value at each stage uses all available bits

This method results in maximum possible accuracy for the studied algorithm but will only be valid for a particular input format (16-bit PCM stereo CD quality audio, for exemple). For a new input format (say PCM 24-bit / 44.1kHz), the simulations would have to be done again to optimize the positions of binary points.

2.5 Summary of fixed point arithmetic

Let $U(iwl, fwl)$ be the subset of unsigned fixed point numbers with integer word length iwl and fractional word length fwl . $S(iwl, fwl)$ is the subset of two's complement fixed points with integer word length iwl and fractional word length fwl . Let u_1, u_2, s_1, s_2 be such that :

$$u_1 \in U(iwl_1, fwl_1), u_2 \in U(iwl_2, fwl_2), s_1 \in S(iwl_1, fwl_1), s_2 \in S(iwl_2, fwl_2)$$

1- Word Length:

$$wl_{u_1} = iwl_1 + fwl_1$$

$$wl_{s_1} = iwl_1 + fwl_1 + 1$$

2- Range:

$$0 \leq u_1 \leq 2^{iwl_1} - 2^{-fwl_1}$$

$$-2^{iwl_1} \leq s_1 \leq 2^{iwl_1} - 2^{-fwl_1}$$

3- Resolution:

This is the smallest non-zero value representable.

$$\delta_{u_1} = 2^{-f_{wl_1}}$$

$$\delta_{s_1} = 2^{-f_{wl_1}}$$

4- Accuracy:

This is the maximum difference between a real value and its fixed point representation.

$$acc = \frac{\delta}{2}$$

5- Addition:

$z_u = u_1 + u_2$ is valid only if $i_{wl_1} = i_{wl_2}$ and $f_{wl_1} = f_{wl_2}$

$$i_{wl_{z_u}} = i_{wl_1} + 1 \quad f_{wl_{z_u}} = f_{wl_{f_{wl_1}}}$$

$z_s = s_1 + s_2$ is valid only if $i_{wl_1} = i_{wl_2}$ and $f_{wl_1} = f_{wl_2}$

$$i_{wl_{z_s}} = i_{wl_1} + 1 \quad f_{wl_{z_s}} = f_{wl_{f_{wl_1}}}$$

6- multiplication:

$$z_u = u_1 \times u_2$$

$$i_{wl_{z_u}} = i_{wl_1} + i_{wl_2} \quad f_{wl_{z_u}} = f_{wl_1} + f_{wl_2}$$

$$z_s = s_1 \times s_2$$

$$i_{wl_{z_s}} = i_{wl_1} + i_{wl_2} \quad f_{wl_{z_s}} = f_{wl_1} + f_{wl_2}$$

Chapter 3

Development Toolset

The code of the encoder has been ported on a Altera EPXA1 Development Kit which features the lowest-cost member of the excalibur family, the EPXA1. The EPXA1 device contains an ARM922T 32-bit RISC microprocessor combined with an FPGA. For the project we will only use the ARM to encode audio.

This chapter details the development board as well as the specific software tools provided by Altera. It can be considered by any future user as a useful starting point for developing an application on the board.

3.1 EPXA1 Development kit

3.1.1 Content of the kit

The content of the development kit is the following:

- The EPXA1 development board.
- LCD module kit.
- power supply and several power cords (US, UK, European and Japan).
- The ByteBlaster II cable and a parallel extension cable.
- A 9-pin null modem cable.
- An ethernet cable with and a crossover adaptor.
- Quartus II software CD.
- Excalibur Devices Development Kit CD ROM.
- Introduction to Quartus II Documentation book

3.1.2 EPXA1 development board

3.1.2.1 EPXA1 development board features

The EPXA1 development board features:

- EXPA1F484C device (which embeds the ARM processor).
- Two RS-232 ports.
- 8-Mbyte flash memory (boot from flash supported).
- 32-Mbyte single data rate (SDR) SDRAM on the board.
- 10/100 Ethernet MAC/PHY with full- and half-duplex modes .
- ByteBlaster IEEE 1149.1 Joint Test Action Group (JTAG) connector.
- Two expansion headers for daughter cards (one standard- and one long-format).
- One user-definable 8-bit dual in-line package (DIP) switch block.
- Four user-definable push-button switches.
- Eight user-definable LEDs

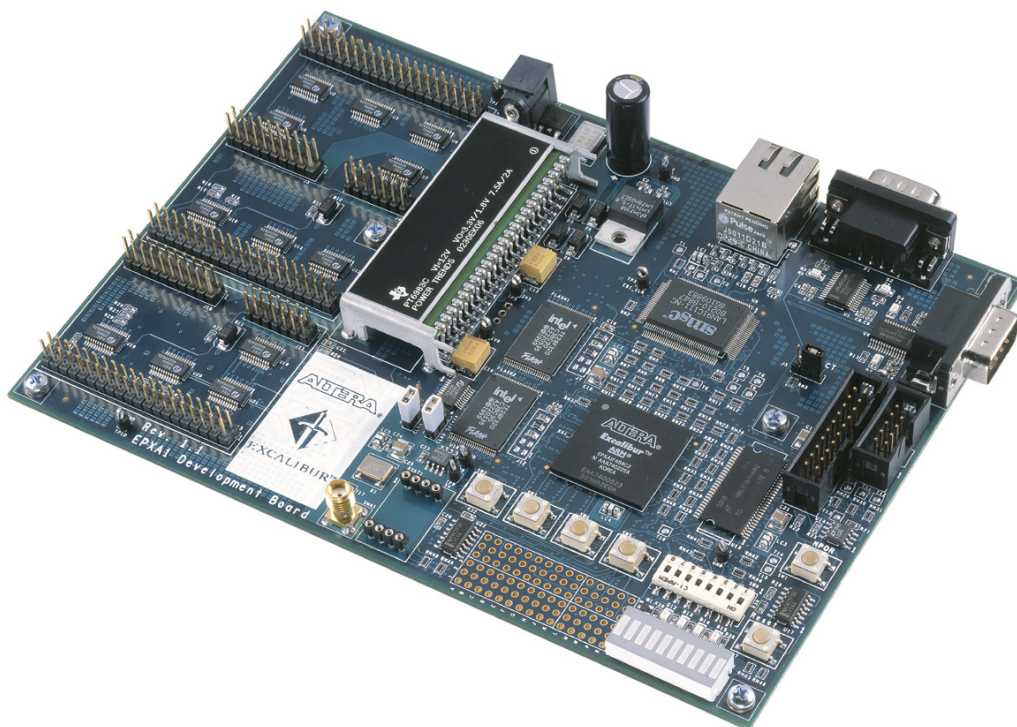


Figure 3.1: *Overview of the EPXA1 Development Board.*

Figure 3.1 gives an overview of the board and of its different features. From left to right, we can see:

- the two expansion headers for daughter cards (The LCD device is connected on one of them).
- the power manager (big black box in the center) and the power jack just behind.
- the two 4-Mbyte flash memories (marked Intel).
- the EXPA1F484C device (marked Altera).
- the RS-232 ports on the upper right corner.
- the JTAG header on the middle right edge.
- and the 32-Mbyte RAM just next.

3.1.2.2 Connections of the EPXA1 development board

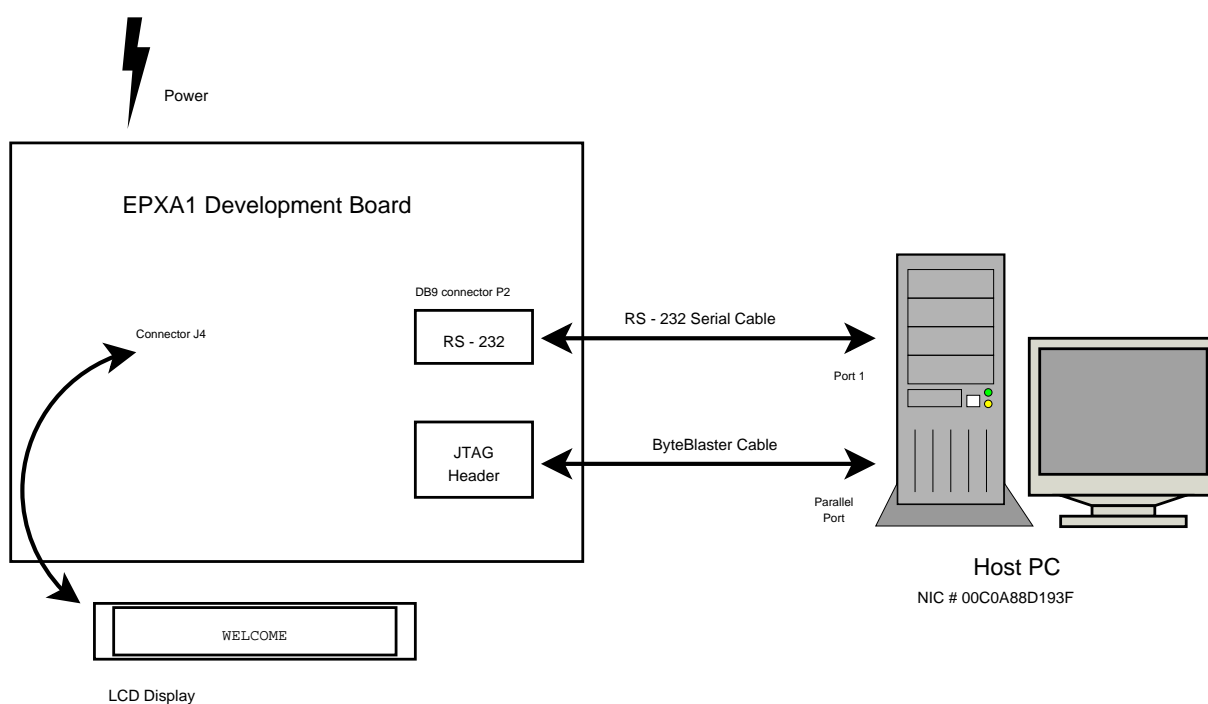


Figure 3.2: *Connections of the EPXA1 Development Board.*

Figure 3.2 reports the connections used to work on the port and necessary to run the encoder from a host PC. The four main connections are:

- The LCD display. It is used to display some information about the current state of the encoder. It should be wired to header J4 ensuring that pin numbers matches: pin 1 is mentioned on both the development kit board and the lcd assembly, and the first pin on the dedicated cable connectors is represented by a small triangle (see figure 3.3). On the board, the cable lies toward the power supply jack and on the display the cable is at the opposite of the display.

- The JTAG header. The ByteBlaster cable, connected to the host PC parallel port, it should be plugged into the 10-pin JTAG header on the board.
- The serial port connector. A NULL-modem cable is connected between the DB9 connector on the XA1 board labeled P2, and the COM port 1 on the host PC.
- The power supply.

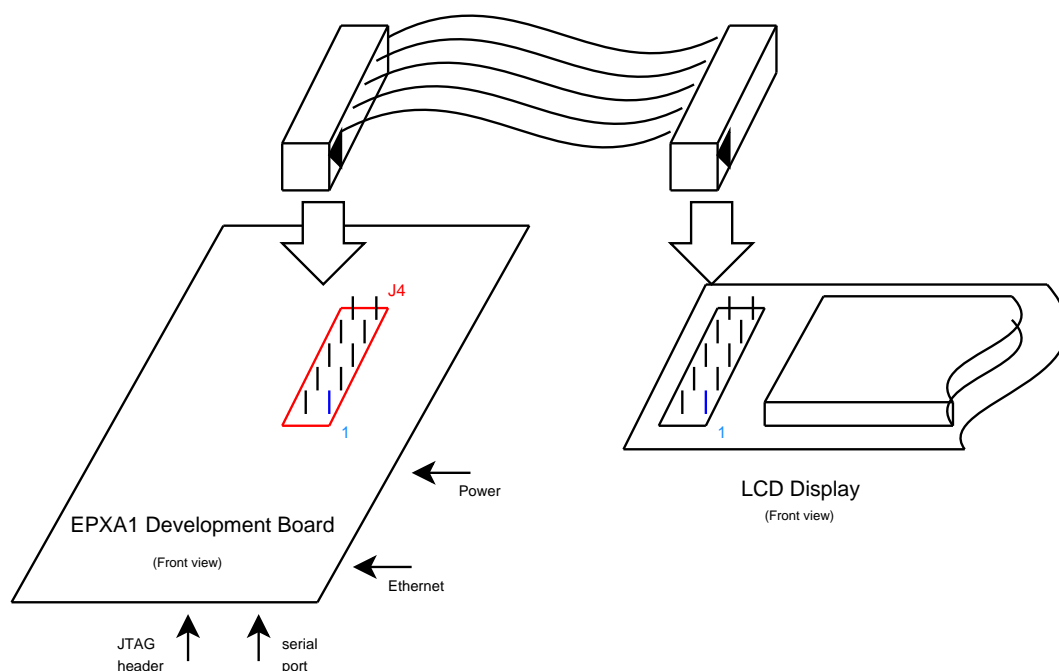


Figure 3.3: *Connection of the LCD module.*

3.1.3 Quartus II

The Altera Quartus II design software provides a complete, multiplatform design environment. Quartus is the unique software to address every Altera's devices. Excalibur devices as the one we are using are quite exotic devices from Altera because they embed an ARM microprocessor. Though, software development for this processor is really a tiny subpart of the possibilities offered by Quartus.

Figure 3.5 shows the structure of the board and of course, the structure that as to be developed in order to run the encoder. As we can see, the heart of the board is the FPGA, it is interfaced with the ARM processor via an AHB (Altera Hardware Bus) high speed bus. In our particular case, we will only consider the PLD as an interface between the processor and the peripheral devices such as LCD or serial communications. This specific architecture of Excalibur devices is usually used in dynamic reconfiguration contexts: this allows the FPGA to be reprogrammed with new hardware at any time, as required by an application. This can be really cost, space and power saving for the industry in order to implement multi-purpose devices with less hardware.

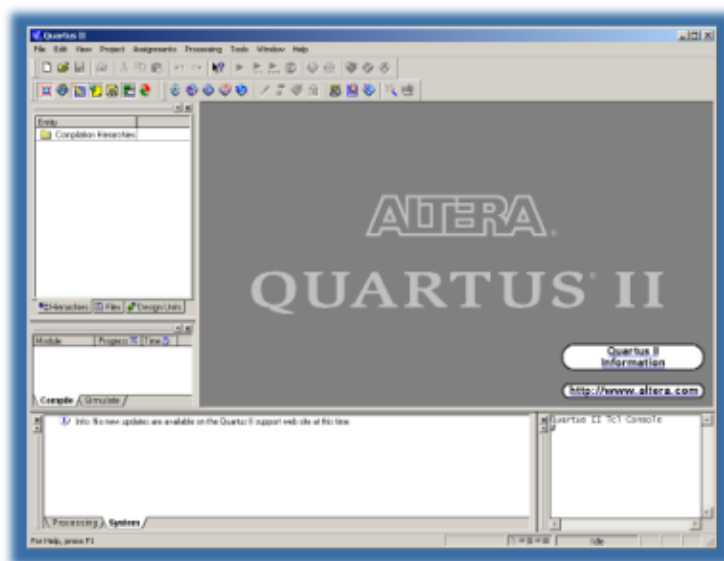


Figure 3.4: *Blank main window of Quartus II software.*

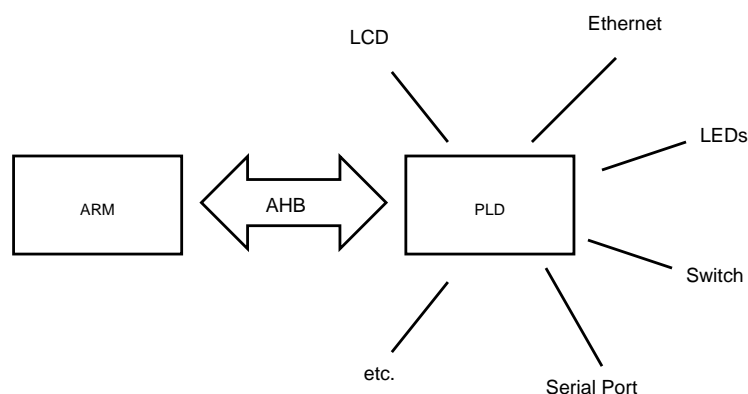


Figure 3.5: *Implementation architecture.*

Figure 3.6 shows the programming flow used in Quartus. As we can see, whatever the project, two compilations are necessary: an hardware compilation and a software compilation. The hardware compilation needs some verilog files (*.v) and returns a slave binary image file (*.sbi). A slave binary image file contains the PLD configuration data from the design.

The software compiler needs this slave binary image file, a system build descriptor file and the software files. The system build descriptor file (*.sbd) describes the interface between the hardware image and the processor. Software files must be coded in pure C and will be compiled by GNUpro. At the end of the software compilation, an image file of the flash memory (*_flash.hex) is output and can be sent to the development board via the ByteBlaster cable using:

```
exc_flash_programmer -g *_flash.hex
```

The -g option will reboot the board after completion of the flash upload. The exc_flash_programmer

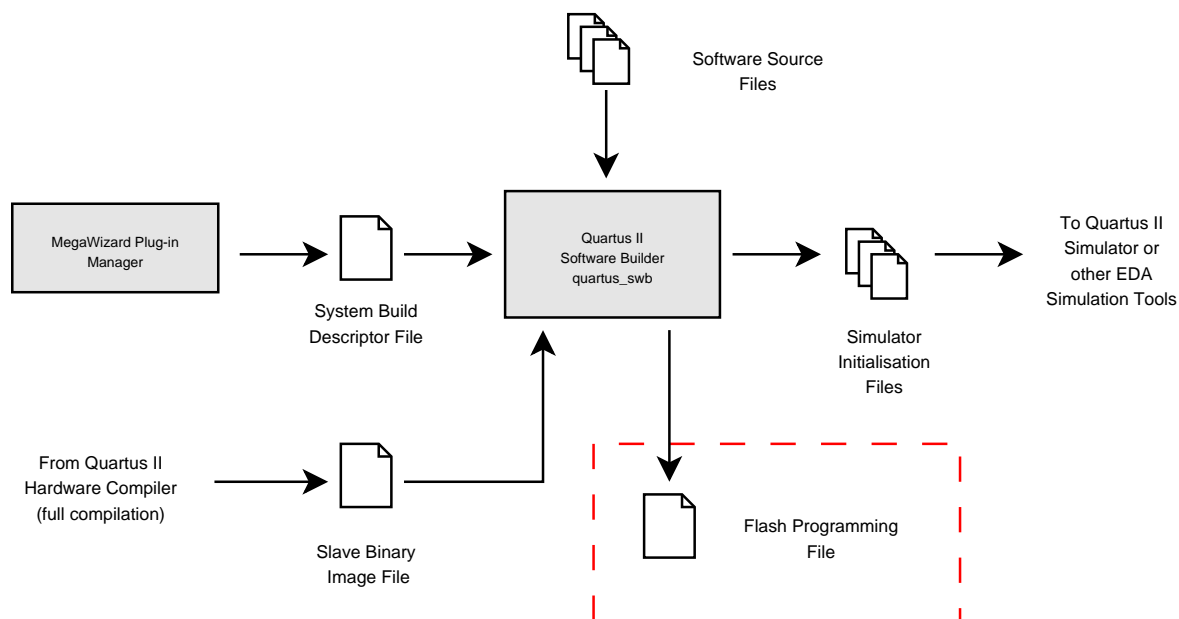


Figure 3.6: *Flash programming files flow.*

instruction is part of the ByteBlaster driver. Appendix A of the *EPXA1 Development Kit Getting Started User Guide* [19] explains how to install the driver on any PC.

For convenience, and as we are mainly interested in software work, I have chosen to modify (from a software point of view only) a pre-existing build offered by Altera. This build is the LCD demonstration that came along with the kit, it uses LCD driver logic in the FPGA driven by the processor to print text to the external LCD module. This choice suffers from the non-use of the Ethernet port (as it wasn't implemented in this demonstration) which would highly faster audio data transfers with the board. However, serial communication is already sufficient to demonstrate that the encoder works properly on the board and allows to encode real audio files.

3.1.4 GNUpro

As mentioned previously, software files are compiled using GNUpro compiler. GNUpro is the compiler provided with the development kit. It is a Red Hat commercial software development suite built around the open source GNU standard. GNUpro is specifically designed for commercial developers of embedded products.

The use of this compiler is really restrictive. First of all it only allows pure C code. It won't allow any C++ like style. On the other hand compilation may be successful but the board may be more restrictive than the compiler and return errors and not run the code at all. That is why I have reported some strange facts in the next section.

3.2 Usage guide

3.2.1 Software compilation doesn't work

When opening Quartus it is good to have look to the settings to check that the compiler chosen is really the GNUpro one and not anything else.

3.2.2 Do not declare any long arrays into a function

This was really difficult to find but long arrays must be declared only in a global context. For example the following code will compile but will not run on the board and send continuous error messages on the serial port (if not using hyperterminal one can notice that a small surface-mount led next to the serial port header is continuously on indicating that data is continuously send):

```
#include <stdio.h>

main {

long window1[1024];
long window2[1024];

/* do what you want */

}
```

The solution is simply to declare the variables as global variables, that is declare them outside of any function, even the *main* function of the application.

3.2.3 no file system

The environment doesn't provide any file system support. Consequently it is not possible to save files or data. Each read or write function such as `getchar`, `printf` or `putchar` is directly routed to the serial port and communication functions on the host PC will have to be implemented to send or receive these data.

Other functions such as `sprintf` which might be used in order to prepare a string to be displayed on the LCD has some influences on the serial port as well. Such functions must be banned from the code if we don't want to mess up the communication protocols.

Chapter 4

Implementation

4.1 Port of the filter bank only

The port of the fixed point filter bank was quite straight forward given the simulation files developed by Keith Cullen. The architecture of these simulations is reported on figure 4.1.

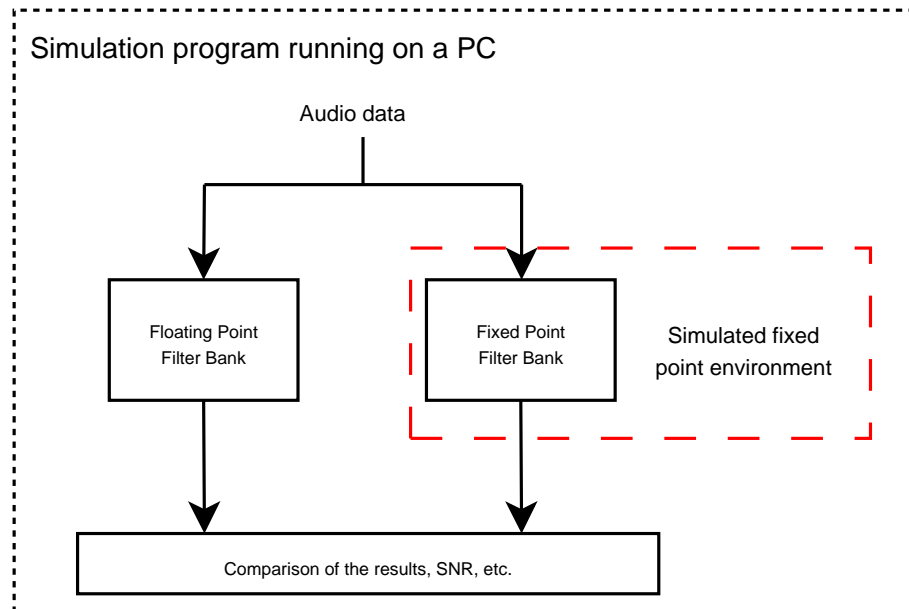


Figure 4.1: *Fixed point filter bank simulation on a PC.*

These simulations are really useful for the project as they give a reference of fixed point results that should be obtain for the port on the EPXA1.

The main idea of the port has been to derive the architecture used for the simulations to a master - slave architecture across a host PC and the development board. As stated on figure 4.2, this master - slave architecture rely on communication functions and protocols that have to be jointly developed. These communication functions are detailed in the next part of this chapter.

From a code point of view, very little modification has been made. The first requirement was to ensure that the code for the board was pure C. The original code wasn't extensively using C++ properties; only some type casts and some overloading functions have been changed. Because of linking problems during the compilation, some tables and arrays have also been moved to more appropriate places.

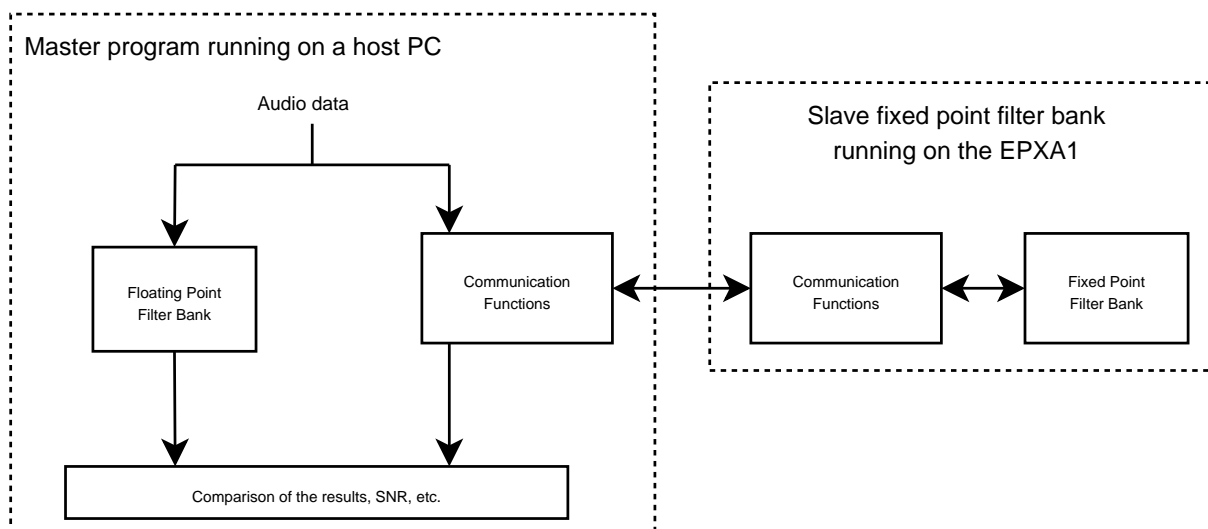


Figure 4.2: *Master - slave architecture.*

4.2 Port of the full encoder

The port of the full encoder has followed the same scheme except the architecture of the simulation files were a bit different. Figure 4.3 reports this structure: the floating point encoder is no longer used for comparison since it makes no sense trying to compute a difference between those two results. At this level, the encoded bitstreams may be quite different while the resulting audio files are very close to one another. Only a listener can judge the difference.

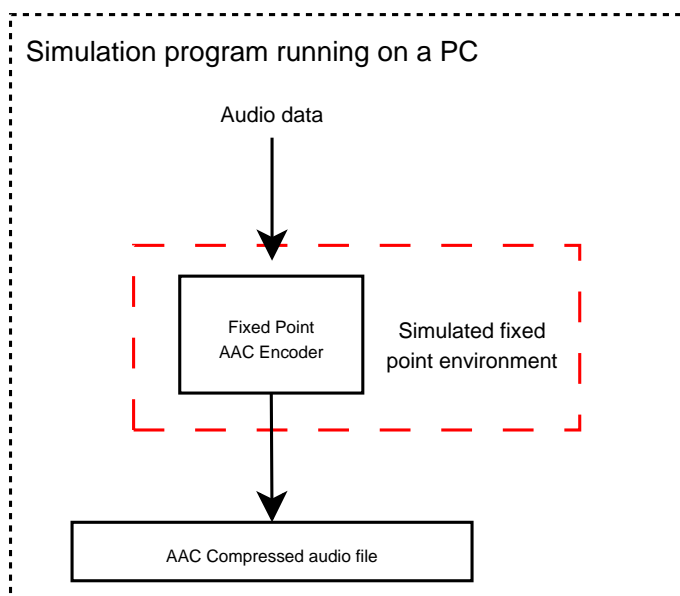


Figure 4.3: *Fixed point encoding simulation on a PC.*

The only modification that has been done to the code was to move some long array variables used in the psychoacoustic block into global variables to comply with the rules described in 3.2.

4.3 Communication functions

4.3.1 Communication functions - host PC side

The host PC is running Windows 2000 so the Windows' serial port API will be used. The following class CSerial has been implemented to make transfers easy:

```
class CSerial
{
public:
    CSerial();
    ~CSerial();

    bool Open( int nPort = 2, int nBaud = 9600 );
    bool Close( void );

    int ReadData( void *, int );
    int SendData( const char *, int );
    int ReadDataWaiting( void );

    void PutFrame(short*,int);
    void GetFrame(short*,int);
    void GetFrame(long*,int);
    void GetFrame(int*,int);
    void PutInt(int);
    void GetExactCharFrame(char*, int);

    bool IsOpened( void ){ return( m_bOpened ); }

protected:
    bool WriteCommByte( unsigned char );
    HANDLE m_hIDComDev;
    OVERLAPPED m_OverlappedRead, m_OverlappedWrite;
    bool m_bOpened;
};
```

`Open` and `Close` functions are called at the beginning and at the end of the program to open and close the port. For our application the `Open` functions has to be called with `nPort` set to 1 because the host PC has only one serial port and `nBaud` set to 115200 (that is 115.2 kb/s or 14.4 kB/s). 115200 bauds is the highest rate that could be achieved with the board. If one want to change this rate, the corresponding rate has to changed in the function `uart_init` of the file `uartcomm.c` in the software folder of the Quartus project as well. This class doesn't allow concurrent use of the port so an application willing to open the port while the encoder is

used won't be provided access. On the other hand if an other application is already using the port (such as hyperterminal for example), the `Open` function will return false and the encoding process won't start.

`ReadData` and `SendData` are used to read data from and send data to the serial port. They return the number of bytes read or sent. The data is stored or read in the array pointed by the first argument which has to be a pointer to characters (1 byte long). `ReadData` is an interrupt-driven function, it will wait for an event occurring on the serial port, gather the required length of data and finally return. `ReadDataWaiting` returns the number of bytes waiting in the receive buffer of the port waiting to be read.

`PutFrame` and `GetFrame` are provided for convenience to directly address real used data types which are shorts (2 bytes) and long (4 bytes) integers. The second argument is the length of the array, usually 1024 for an audio data frame and 1 for a control flag.

`PutInt` is used to send a single integer. It is mainly used in the encoder to send wave file information such as sample rate or number of audio samples for instance.

`GetExactCharFrame` is used in the encoder to get the precise number of bytes (second argument) in the receive buffer of the serial port on the host PC.

4.3.2 Communication functions - EPXA1 side

As presented previously, each read or write function on the board is routed towards the serial port. Communication function through the serial port will then be quite easy to implement because they will be based on the standard `getchar` and `printf` functions. The top-of-the-iceberg functions used are:

```
void get_char_frame (char *buf, int size, int LED_display, int *LED);
void get_frame (short *in_buff, int size, int LED_display, int *LED);
void put_int(int x);
void put_frame (FILTER_CHANNEL *fch, short win_seq, int LED_display, int *LED);
void put_char_frame(char *frame, int length);
```

The `get_char_frame` function is used in the main function to get control flags and is also used by the `get_frame` function. The first argument `buf` is used to store the data extracted from the serial port. `size` corresponds to the length of data we are to receive. The function won't return unless all specified data has been received. `LED_display` is used as a boolean to require that the progression of the data acquisition is simultaneously shown on the led meter of the development board. `LED` is a pointer to the led meter. To display something on this meter, one just have to assign any number with this pointer. This meter displays the binary transcript of the assigned number; for example if we want to light the first and third led of the meter, the following code has to be used:

```
/* Initialize LED address */
```

```

int *LED;
LED = (int*) na_pio_0;

/* Light first and third led of the meter */
*LED = 5;

```

The `get_frame` function is similar to the `get_char_frame` one except it deal with short integers instead of char. This function is mainly used to receive frames of CD quality audio data as this data is quantized over 16 bits.

The `put_int` function is used to send a single integer to the host PC. This function is mainly used to send the number of bytes of the bitstream computed by the encoder.

The `put_frame` function has been implemented to return to the host PC the data computed by the filter bank for in the beginning of the project. This data is composed of long integers (4 bytes) either in a single frame of 1024 elements for a long window transform or 8 consecutive frames of 128 elements for a short window transform. One or the other of these two possibilities must be passed to the function by the `win_seq` parameter.

The `put_char_frame` function is similar to the `put_frame` one but is used to send the bitstream buffer computed by the encoder.

4.3.3 Communication protocol for the filter bank only

The first part of the encoder that I had to deal with was the filter bank. In this simulation it is the user that has to decide wether to use long or short block window whereas this choice is usually made by the psychoacoustic block in the complete encoder. Consequently, this parameter has to be sent only once to the board before any new coding work and doesn't need to be actualized for each 1024 audio sample block as the simulation doesn't support windows size changes during compression of a single audio file.

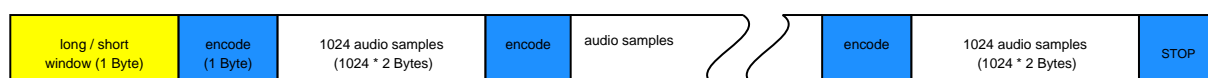


Figure 4.4: *Communication pattern of the communication host PC → board for the filter bank only. The communication board → host PC is just the sequence of transform results.*

As a slave, the development board is operating some kind of *blind* coding, that is it successively receives 1024 audio samples, runs it through the filter bank and returns the result to the PC. It is then necessary to tell the board which kind of work it will have to do next and more importantly which kind of data it will receive next. From this point of view it is necessary to tell the board whether the job is done or not and if it has to go back in the initial state. To do so, a stop flag is sent before any new audio frame as illustrated on figure 4.4.

4.3.4 Communication protocol for the encoder

The communication protocol for the complete encoder is slightly more complicated but is still based on a sequence of sending an audio frame to the board and getting the computed bitstream back.

4.3.4.1 EPXA1 side

figure 4.5 shows the communication protocol seen from the development board side. Before encoding a new audio file some information is sent: the required bit rate and information about the file.

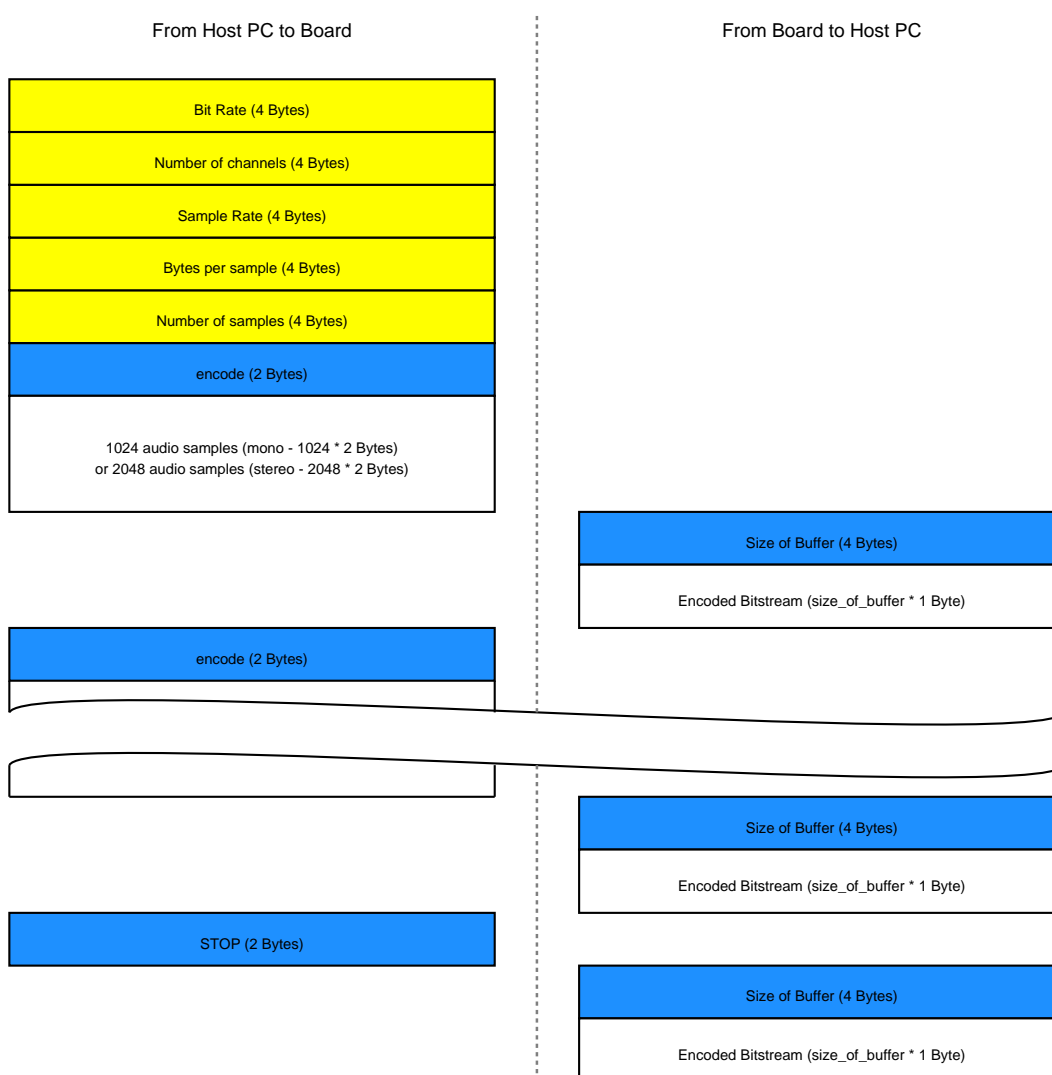


Figure 4.5: *Communication sequence seen from EPXA1 side.*

The information about the file is:

- The number of channels of the file. It corresponds to `wf.num_channels` in the code. This is useful to know how many mono frames we must receive and encode.

- The sample rate of the file. It corresponds to `wf.sample_rate` in the code.
- The number of bytes per sample of the file. It corresponds to `wf.bytes_per_sample` in the code.
- The number of audio samples in the file. It corresponds to `wf.num_sample` in the code. It is used to display the progression of the encoding on the LCD in percent.

After receiving a new audio frame, the encoding function is called on the board and returns the length (not constant) of the encoded bitstream produced. This length is sent back to the host PC in order for it to appropriately gather the stream.

We can see that an extra encoded bitstream is sent after the stop flag. This is due to the encoding function that has to be called one extra-time at the end of the file as there is a one frame delay between input and output.

4.3.4.2 Host PC side

From the host PC side, we can consider communications mirroring the protocol described on figure 4.5. This is illustrated on the left hand side of figure 4.6.

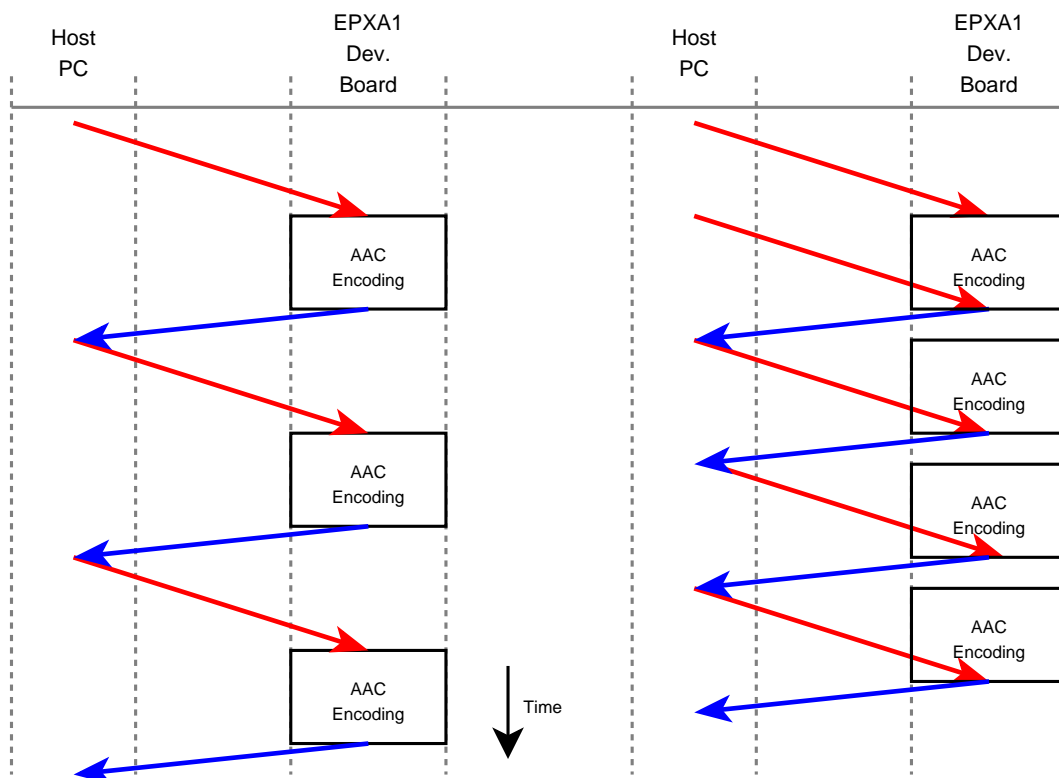


Figure 4.6: *Serial (left) and Overlapped (right) communications. Overlapping communication and computations drastically fasters the encoding process.*

Red arrows represent communications from the host PC towards to development board and blue ones represent communications on the other side. Communications from the PC to the board

are much longer than the others, indeed they should be around 12 times longer for an encoding at 128 kbits/s.

It appeared that the implementation of the serial port on the development board has its own hardware buffer. That is really useful because data can be sent to or received by the board without involving the processor in any way. When the processor requires data, it looks for it in the buffer of the serial port. If there is data in this buffer, it immediately gets it from there, otherwise it hangs here, waiting for data from the host PC and decreasing the time efficiency of the program. It is then possible to have the processor computing the encoding of frame n while frame $n+1$ is being received by the board. This idea of overlapping communication and computation is illustrated on the right hand side of figure 4.6.

Finally the only change that has to be done in order to implement this overlapping it to send consecutively two audio frames at the beginning of the encoding without waiting for the result of the first one which will be received after the second frame is sent. Note that the protocol seen by the board is not changed, its program doesn't need any modification to run this improvement.

The speedup gain by this overlapping is dependant on the relative times of communication and computations. For convenience, the encoding time has been drawn shorter than communication time on figure 4.6 (where communication time represents the addition of all communications for the encoding of a single frame). This is not necessarily true, for instance the processing time of the non-optimized encoder was a bit longer than communication time.

4.4 Double precision multiplication algorithms

In the encoder code is implemented a double precision multiplication. At the beginning of my work, Keith suggested that I look for a faster algorithm than the one he used. I have then implemented different algorithms aiming to faster the process as this multiplication is extensively used. Indeed, this extended precision multiplication is called 15 times per audio sample in a long window transform and 12 times per sample in a short window transform in the filterbank block. These multiplications represent more than 60% of the time spent for the transform in the original code.

4.4.1 Original Code

Here is the original code:

```
void fixedpoint_dmpy(fixedpoint x, fixedpoint y, fixedpoint *zh, fixedpoint *zl)
{
    int          x_sign, y_sign;
    unsigned long acc, u, v;
    unsigned short u0, u1, v0, v1,
                  w0, w1, w2, w3;
```

```
x_sign = x < 0;
y_sign = y < 0;
u = abs(x);
v = abs(y);

u0 = (unsigned short)u;
u1 = (unsigned short)(u >> 16);
v0 = (unsigned short)v;
v1 = (unsigned short)(v >> 16);

acc = (unsigned long)u0 * (unsigned long)v0;
w0 = (unsigned short)acc;
acc = (unsigned long)u1 * (unsigned long)v0 + (acc >> 16);
w1 = (unsigned short)acc;
w2 = (unsigned short)(acc >> 16);

acc = (unsigned long)w1 + (unsigned long)u0 * (unsigned long)v1;
w1 = (unsigned short)acc;
acc = (unsigned long)w2 + (unsigned long)u1 * (unsigned long)v1 + (acc >> 16);
w2 = (unsigned short)acc;
w3 = (unsigned short)(acc >> 16);

if (x_sign ^ y_sign)
{
    w3 = ~w3;
    w2 = ~w2;
    w1 = ~w1;
    w0 = ~w0;
    acc = (unsigned long)w0 + 1;
    w0 = (unsigned short)acc;
    acc = (acc >> 16) + (unsigned long)w1;
    w1 = (unsigned short)acc;
    acc = (acc >> 16) + (unsigned long)w2;
    w2 = (unsigned short)acc;
    acc = (acc >> 16) + (unsigned long)w3;
    w3 = (unsigned short)acc;
}

*zh = (fixedpoint)((((unsigned long)w3 << 16) | (unsigned long)w2);
*zl = (fixedpoint)((((unsigned long)w1 << 16) | (unsigned long)w0);
}
```

This algorithm performs the multiplication of 32-bit **fixedpoint** operands x and y and returns the 32-bit MSB and LSB of the result in zh and $z1$ respectively. To understand this code, one have to recall that a typical N -bit integer ALU will only give the programmer access to the N LSBs from the result of any arithmetic operation (figure 4.7).

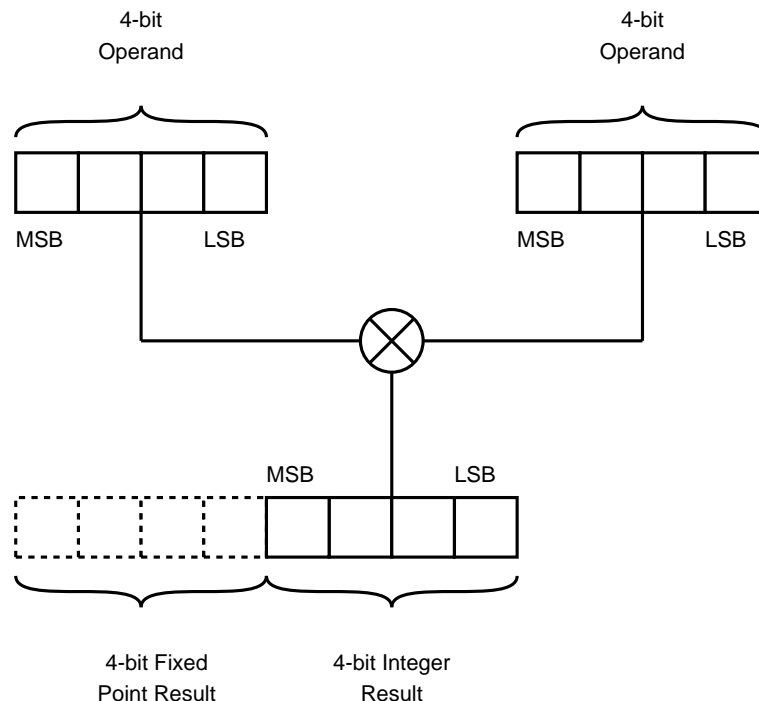


Figure 4.7: Typical N -bit integer ALU. The programmer has only access to the N LSBs of the result.

Consequently, fixed point variables are limited to half the word length of integer variables (e.g. 16-bit on a 32-bit machine - figure 4.8). In order to overcome this limitation, fixed point variables have to be split into two $N/2$ word length subvariables.

The algorithm presented above directly uses this idea by splitting the input variables x and y into $u0, u1$ and $v0, v1$.

4.4.2 A restricted use of the Karatsuba algorithm

As we can see on the above algorithm, four multiplications are performed. Multiplications are usually considered much slower than other simpler operations such as additions or shifts. It may then be a good idea to try to limit the number of multiplication in the algorithm. As discovered by Karatsuba and Ofman (1962), multiplication of two N -digit numbers can be done with a bit complexity of less than n^2 using identities of the form:

$$(a + b \cdot 2^N) (c + d \cdot 2^N) = ac + [(a + b)(c + d) - ac - bd] \cdot 2^N + bd \cdot 2^{2N}$$

This algorithm is usually used for very long integers in a recursive manner with a *divide and conquer* philosophy. Nevertheless, I thought it would be interesting to implement the identity

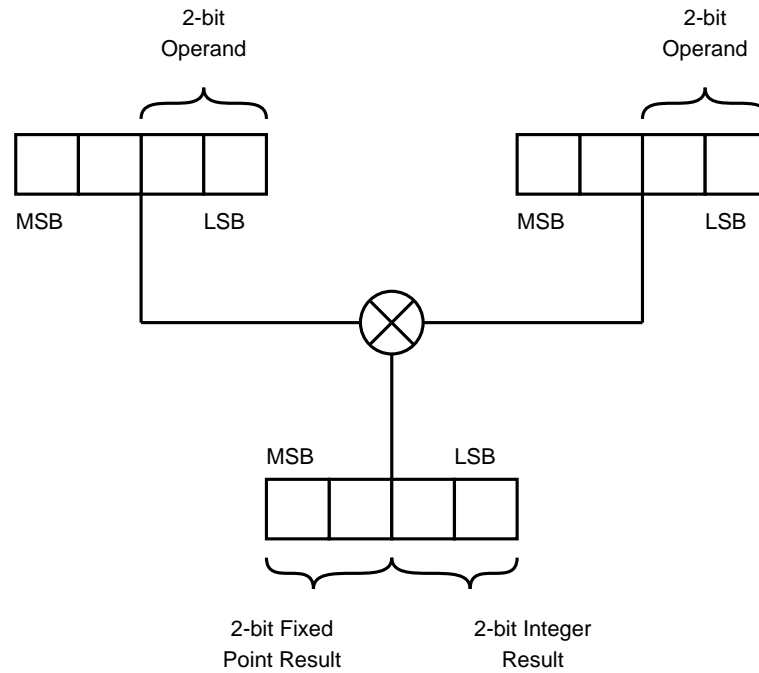


Figure 4.8: *Fixed point variables are limited to half the word length of integer variables .*

described above in order to perform only 3 multiplications instead of 4. For our particular application, our two operands are written in the base $w = 2^{16}$:

$$x = u_0 + u_1 \cdot w$$

$$y = v_0 + v_1 \cdot w$$

and their product is:

$$P = x \cdot y$$

$$P = u_0 \cdot v_0 + (u_0 \cdot v_1 + u_1 \cdot v_0) \cdot w + u_1 \cdot v_1 \cdot w^2$$

$$P = p_0 + p_1 \cdot w + p_2 \cdot w^2$$

Instead of evaluating products of individual digits, we can write:

$$q_0 = u_0 \cdot v_0$$

$$q_1 = (u_0 + u_1) \cdot (v_0 + v_1)$$

$$q_2 = u_1 \cdot v_1$$

The key term is q_1 , which can be expanded, regrouped, and written in terms of the p_j as

$$q_1 = p_0 + p_1 + p_2$$

However, since $p_0 = q_0$, and $p_2 = q_2$, it immediately follows that

$$p_0 = q_0$$

$$p_1 = q_1 - q_0 - q_2$$

$$p_2 = q_2$$

so the three ‘digits’ of the product P have been evaluated using three multiplications rather than four at the presumably small cost of extra additions.

In practice, this algorithm has shown to be slower than the original one. The calculation of the MDCT with this algorithm is about 8% longer than the MDCT with the original algorithm.

4.4.3 The basic shift and add algorithm

This multiplication algorithm is just a basic sequence of shifts and additions of the multiplicand ($A = a_{n-1}a_{n-2} \cdots a_0$) according to the bit sequence of the multiplier ($X = x_{n-1}x_{n-2} \cdots x_0$). At step j , the multiplier bit x_j is examined and the product $x_j \cdot A$ is added to $P^{(j)}$ which is the previously accumulated partial product ($P^{(0)}$).

$$P^{(j+1)} = (P^{(j)} + x_j \cdot A) \cdot 2^{-1} \quad ; \quad j = 0, 1, 2, \dots, n-2$$

Multiplying by 2^{-1} corresponds to a shift by one position to the right, this alignment is necessary since the weight of x_{j+1} is double that x_j .

Table 4.1: *Example of a positive multiplier.*

A	1 0 1 1		-5
X	× 0 0 1 1		3
$P^{(0)} = 0$	0 0 0 0		
$x_0 = 1 \rightarrow$ Add A	+ 1 0 1 1		
	1 0 1 1		
Shift	1 1 0 1	1	
$x_1 = 1 \rightarrow$ Add A	+ 1 0 1 1		
	1 0 0 0	1	
Shift	1 1 0 0	0 1	
$x_2 = 0 \rightarrow$ Shift only	1 1 1 0	0 0 1	-15

Table 4.2: *Example of a negative multiplier.*

A		1 0 1 1		-5
X	\times	1 1 0 1		-3
$x_0 = 1 \rightarrow$ Add A		1 0 1 1		
Shift		1 1 0 1	1	
$x_1 = 0 \rightarrow$ Shift only		1 1 1 0	1 1	
$x_2 = 1 \rightarrow$ Add A	$+$	1 0 1 1		
		1 0 0 1	1 1	
Shift		1 1 0 0	1 1 1	
$x_3 = 1 \rightarrow$ Correct	$+$	0 1 0 1		
		0 0 0 1	1 1 1	15

For this multiplication algorithm, the distinction must be made between multiplication with a negative multiplicand A and multiplication with a negative multiplier X . If only the multiplicand is negative, there is no need to change the algorithm. But if the multiplier is negative, a correction step is added by subtracting the multiplicand (ie: adding its two's complement).

Although the implementation of this multiplier seems to be quite straight forward, the programmer must pay attention to possible overflow errors and deal with the carry generated by the addition steps.

Finally, this algorithm has shown to be the worse multiplier I have tried with a run time of the MDCT about 10% longer than the original algorithm.

4.4.4 Some little changes in the original code

Given that two algorithms I have tried to speedup the process have turned out to be slower than the original one, I have tried to apply some little tricks on the original code hoping to give a little boost. These little changes have only saved 2.6% and 3.1% of the time spent on the MDCT for the long and short window analysis respectively.

The main ideas used are to avoid as much as possible type casts and to regroup $w0$, $w1$, $w2$ and $w3$ in a single array.

```
void fixedpoint_dmpy(fixedpoint x, fixedpoint y, fixedpoint *zh, fixedpoint *zl)
{
    int          x_sign, y_sign;
    unsigned long acc, u, v;
    unsigned short w[4];
    unsigned long u01,u11,v01,v11;

    x_sign = x < 0;
    y_sign = y < 0;
```

```
    u = abs(x);
    v = abs(y);

    u0l = u & 0x0000ffff;
    u1l = (u >> 16) & 0x0000ffff;
    v0l = v & 0x0000ffff;
    v1l = (v >> 16) & 0x0000ffff;

    acc = u0l * v0l;
    w[0] = acc;
    acc = u1l * v0l + (acc >> 16);
    w[1] = acc;
    w[2] = acc >> 16;

    acc = w[1] + u0l*v1l;
    w[1] = acc;
    acc = w[2] + u1l * v1l + (acc >> 16);
    w[2] = acc;
    w[3] = acc >> 16;

    if (x_sign ^ y_sign)
    {
        w[3] = ~w[3];
        w[2] = ~w[2];
        w[1] = ~w[1];
        w[0] = ~w[0];
        acc = w[0] + 1;
        w[0] = acc;
        acc = (acc >> 16) + w[1];
        w[1] = acc;
        acc = (acc >> 16) + w[2];
        w[2] = acc;
        acc = (acc >> 16) + w[3];
        w[3] = acc;
    }

    *zh = (fixedpoint)((((unsigned long)w[3] << 16) | (unsigned long)w[2]));
    *zl = (fixedpoint)((((unsigned long)w[1] << 16) | (unsigned long)w[0]));
}
```

4.4.5 Using long long integers

Finally, since GNUpro (the compiler of the development kit) supports `long long` integers, the simpler solution is to use them. `long long` integers are 64-bit integers. The following code has been implemented and has shown to faster the filter bank block by more than 60% and the complete encoder by more than 40%.

```
void fixedpoint_dmpy(fixedpoint x, fixedpoint y, fixedpoint *zh, fixedpoint *zl)
{
    long long res;

    res = ((long long)x) * ((long long)y);

    *zh = (fixedpoint)(res >> 32);
    *zl = (fixedpoint)res;
}
```

Since the ARM 9 is a 32-bit processor, it is very likely that the use of `long long` integers calls some optimized assembly routines that does the exact same job as the one described in the original algorithm.

Chapter 5

Results - Evaluation

5.1 Filter Bank

As I have first worked on the filter bank only, here are the results for this block of the encoder. In this section, a frame corresponds to a single channel frame, that is 1024 audio samples.

5.1.1 Timing

Long window:	8.96	milliseconds per frame
Short window:	7.09	milliseconds per frame

Table 5.1: *Average performing time of the original filter bank only.*

Table 5.1 reports the performing time of the original filter bank. For both long and short window transforms, an audio frame of 1024 samples is considered. That is, 7.09 milliseconds corresponds to the computing of 8 transforms of 128 samples each for the short window transform. As we are dealing with the same number of sample, one would expect to measure the same performing time. This is due to the non-linearity of the transform: it is faster to compute 8 short transforms than 1 long one.

Long window:	4.09	milliseconds per frame
Short window:	3.75	milliseconds per frame

Table 5.2: *Average performing time of the filter bank only with the high speed extended precision multiplication.*

Table 5.2 reports the measures of the filter bank using the fast multiplication algorithm. `long long` integers based multiplication algorithm divides by more than two the computing time of the filter bank compared to table 5.1, it's really a huge improvement. To have an idea of what represents the milliseconds scale, one can remember that a frame is about 23 milliseconds for a sampling rate of 44,1 kHz (CD sampling rate).

Table 5.3 and 5.4 are measures of the filter bank and communications for the original filter bank and the filter bank using the fast multiplication algorithm. Communication time is about 100 times longer than computing filter bank block with the fast multiplication algorithm! In these

Long window:	567	milliseconds per frame
Short window:	564	milliseconds per frame

Table 5.3: *Average performing time of the original filter bank and communications.*

Long window:	563	milliseconds per frame
Short window:	560	milliseconds per frame

Table 5.4: *Average performing time of the filter bank and communications with the high speed extended precision multiplication.*

conditions we are completely dependent on communication time: neither fast computations or overlapping can significantly help us to reduce the overall performing time.

Track	Size	Time	Window	Run Time
Buffy.wav	10.9 MB	1 min 04 s	long window:	1710.4 s 28.5 min
			short window:	1699.9 s 28.3 min

Table 5.5: *Original filter bank run time for a real track using the serial port.*

Table 5.5 gives an idea of the time required for some *real length* audio track. About 28 minutes are needed to deal with a 1 minute track whereas the actual computing time is roughly 28 seconds.

For this particular test, the communications are longer than the ones of the final encoder. Indeed, to compute the filter bank only we need to send one audio frame (1024 audio samples of 2 bytes) and get the result back (1024 transformed samples of 4 bytes). That is 6144 bytes per mono frame. This is much more than the actual need for the complete encoder. For the encoder we still need to send one audio frame (1024 audio samples of 2 bytes) but the encoded result is much shorter (around 175 bytes). That is only around 2200 bytes for the complete encoder dealing with one frame. Consequently, one can expect the complete encoder to be much faster than performing the filter bank only as communications will be faster.

5.1.2 Precision

Table 5.6 reports the precision of computing the filter bank with the fixed point implementation. The column *simulation* makes reference to the result computed on the PC by the simulation program.

The signal to noise ratios are calculated by comparing the result given by the fixed point version with the floating point one.

The SNR calculated with the results of the board and the theoretical results are the exact same. Indeed, the outputs of the board and simulation have been compared using Matlab and were absolutely matching.

			Simulation	ARM	
In.wav	86.1 kB	0.48 s	long window:	130.6365 dB	130.6365 dB
			short window:	129.5466 dB	129.5466 dB
Buffy.wav	10.9 MB	1 min 04 s	long window:	127.8283 dB	127.8283 dB
			short window:	126.6140 dB	126.6140 dB

Table 5.6: Precision (SNR) obtained by the simulation and the ARM implementation for the filter bank. The results of the board are the exact same as the simulation ones.

5.2 Complete Encoder

In this section a frame corresponds to a stereo frame, that is 2 times 1024 audio samples.

			Encoding Time		Real Time
Buffy8s.wav	1.52 MB	8.45 s	38 s	0.105 s/frame	× 0.22
Buffy.wav	10.9 MB	1 min 04 s	286 s	0.103 s/frame	× 0.23

Table 5.7: Encoding time of the original fixed point encoder running on a Pentium 4 2.5 GHz - 256 MB RAM - Windows 2000.

Table 5.7 gives an idea of the amount of computation to perform the compression. Of course the code is not optimized to run on the PC and provides all the extra fixed point environment but still 280 seconds of computation is a lot of work. The PC is running at 2.5 GHz whereas the ARM is only running at 100 MHz.

Real Time column gives the ratio of the track length to encoding time.

5.2.1 Non-optimized version

			Encoding Time		Real Time
In.wav	86.1 kB	0.48 s	17.51 s	0.845 s/frame	× 0.029
Buffy8s.wav	1.52 MB	8.45 s	284.5 s	0.783 s/frame	× 0.029

Table 5.8: Encoding time for the non-optimized encoder (including communications).

Table 5.8 gives the encoding times of the original non-optimized encoder on two wav files. Compared to the PC run, it is about 10 times slower and 5 times slower if we except the communication time.

Table 5.9 highlights the proportion of communication over the encoding process. Computations represents a bit more than 51% of the overall performing time. Computing the whole encoder algorithm is about 50 times longer than the filter bank only. This non-optimized version of the encoder doesn't use the overlapping of communications and computation, that's why the overall encoding time is the sum of communications and computations.

With this non-optimized version of the encoder, we can see that overlapping will greatly faster the encoding process. If the overlap is well timed, the communication time will be faster than computations and the inconvenient of using serial port transmission will totally disappear.

			Proportion	Real Time
Communications	8.63 s	0.411 s/frame	48.6 %	× 0.056
Computations	8.88 s	0.434 s/frame	51.4 %	× 0.053
Total	17.51 s	0.845 s/frame	100 %	× 0.027

Table 5.9: *Computation time vs. communications for the non-optimized encoder running In.wav.*

5.2.2 Communication overlapping computations version

			Bit rate	Encoding Time		Real Time
In.wav	86.1 kB	0.48 s	128 kbit/s	10.76 s	0.512 s/frame	× 0.045
Buffy8s.wav	1.41 MB	8.45 s	96 kbit/s	168.2 s	0.463 s/frame	× 0.050
Buffy8s.wav	1.41 MB	8.45 s	128 kbit/s	167.9 s	0.462 s/frame	× 0.050
Buffy8s.wav	1.41 MB	8.45 s	192 kbit/s	158.0 s	0.453 s/frame	× 0.053
Buffy.wav	10.9 MB	1 min 04 s	128 kbit/s	1261 s	0.445 s/frame	× 0.051

Table 5.10: *Encoding time for the communication overlapped with computations encoder.*

Table 5.10 shows the encoding time for several files using the original encoder with communications overlapped with computations. The first line is the file In.wav which non-overlapping encoding time is already detailed in table 5.8 and 5.9. For this particular file, the overall encoding time was 17.51 seconds and is now 10.76 seconds thanks to the overlap. The speedup of overlapping is not as high as expected but it's nonetheless a huge improvement.

For the file Buffy8s.wav, three bit rates are reported: 96, 128 and 192 kbit/s. We can see that the slowest the bit rate is, the longer it takes to encode the file.

For the file Buffy.wav, the overall encoding time is 1261 seconds. That is much less than the time required to perform the filter bank only (around 1700 seconds in table 5.5) because communications are faster.

5.2.3 Communication overlapping computations version and high speed multiplication

In this last part we will see how using the fast multiplication algorithm fasters the encoding process.

			Bit rate	Encoding Time		Real Time
In.wav	86.1 kB	0.48 s	128 kbit/s	8.30 s	0.415 s/frame	× 0.058
Buffy8s.wav	1.41 MB	8.45 s	128 kbit/s	139.3 s	0.385 s/frame	× 0.063
Buffy.wav	10.9 MB	1 min 04 s	128 kbit/s	1065 s	0.384 s/frame	× 0.060

Table 5.11: *Encoding time for the communication overlapped with computations encoder (high speed multiplication).*

Table 5.11 gives some overall encoding times for several audio files using the final version of the

encoder. This final version uses overlapped communications and the high speed multiplication algorithm.

For the particular file Buffy.wav the overall encoding time is now reduced to 1065 seconds. This reduction of the encoding time was quite predictable since the use of the fast multiplication algorithm fasters the computations which were longer than the overlapped communications.

			Proportion	Real Time
Communications	140 s	0.392 s/frame	100 %	× 0.059
Computations	93 s	0.260 s/frame	66.3 %	× 0.089
Total	140 s	0.392 s/frame	100 %	× 0.059

Table 5.12: *Computation time vs. communications for the overlapping and high speed multiplication encoder running buffy8s.wav.*

Table 5.12 compares the computation and communication times for the last version of the encoder; it can be balanced with table 5.9 which doesn't use any optimization. As we are using overlapped communications, the overall encoding time is just the higher of the computation or communication times; it is no longer the sum as it was in table 5.9.

Let's have a look to the computation time: it was 434 milliseconds without the fast multiplication algorithm (table 5.9) and is now 260 milliseconds for the optimized version. The computing time is reduced by 40% only by using the fast multiplication algorithm detailed in 4.4.5.

Since computations are now much faster than the original version, the communication part become the new bottleneck of the encoder. 100% of the encoding time is spend by communications whereas only 66% of the encoding time is used for actual computing.

Conclusion

The very first assignment of my project was to have the encoder ported on the EPXA1 development board. That is just what have been done and the encoder has shown to perform exactly like the simulation does. Furthermore the original code is quite good and hardly no modifications have been done to it for the port.

The other subject of my work was the implementation of communication functions. The latest version of the encoder works using serial port communication at a rate of 14.4 kB/s which is very slow compared to the rate that could be achieved using Ethernet communications (theoretically up to a 1,000 times faster). I have worked on implementing communications using the Ethernet port but without any success mainly because of hardware considerations in Quartus. Nevertheless, the overlapping of communications and computations has allowed to drastically reduce the overhead due to communications on the serial port. Indeed, the original code was slower than the communications. Serial communications are then already sufficient to encode a real audio track from a host PC and perform a demonstration of the encoder on a ARM platform.

I have finally worked on the extended precision multiplication algorithm of the encoder. The use of `long long` integers has made the encoding time drastically faster. As the computations are now faster. The new bottleneck of the encoding process are the communications.

Anyway, even with Ethernet communications it would be quite impossible to reach real time encoding as computation of a frame requires around 260 milliseconds whereas an audio frame last only 23 milliseconds. Only real time decoding may be achievable.

Bibliography

- [1] ISO/IEC International Standard 13818-7, *Information technology - Generic coding of moving pictures and associated audio information - Part 7: Advanced Audio Coding (AAC)*
- [2] Princen J, Bradley A, *Analysis/Synthesis Filter Bank Design Based on Time Domain Aliasing Cancellation*, IEEE Transactions, ASSP-34, No.5, Oct 1986, pp. 1153-1161.
- [3] Princen J, Johnson A, Bradley, A, *Subband/Transform Coding Using Filter Bank Designs Based on Time Domain Aliasing Cancellation*, Proc. of the ICASSP 1987, pp 2161-2164.
- [4] P. Duhamel, Y. Mahieux, and J. Petit, *A fast algorithm for the implementation of filter banks based on time domain aliasing cancellation*, in Proceedings of the Int. Conf. on Audio Speech and Sig. Proc., pp. 2209–2212, May 1991.
- [5] ISO/IEC International Standard 11172-3, *Information technology - Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbits/s - Part 3: Audio*
- [6] J. Herre, J. D. Johnston, *Enhancing the Performance of Perceptual Audio Coders by Using Temporal Noise Shaping (TNS)*, Proc. 101st AES Conv., Los Angeles, Nov 1996
- [7] J. Herre, *Temporal Noise Shaping, Quantization and Coding Methods in Perceptual Audio Coding: A Tutorial Introduction*, AES 17th Int. Conf. on High Quality Audio Coding
- [8] M. Bosi, *Digital Audio Compression*, Digital Technical Journal Vol. 5 No. 2, Spring 1993
- [9] D. Pan, *Audio Coding: Basic Principles and Recent Developments*, 6th HC International Conference, Aizu 2003
- [10] S. Church, *On Beer and Audio Coding*,
- [11] Huffman, D.A., *A method for the construction of minimum-redundancy codes*, Proc. IRE, Vol. 40, pp. 1098-1101 (1952)
- [12] Official website for MPEG audio
<http://www.tnt.uni-hannover.de/project/mpeg/audio/>
- [13] Computer Laboratory - University of Cambridge, *ECAD + Architecture main page*
<http://www.cl.cam.ac.uk/Teaching/current/ECADArch>
- [14] Altera, *EPXA1 Development Kit*
<http://www.altera.com/products/devkits/altera/kit-epxa1.html>

-
- [15] Altera, *Literature: Excalibur*
<http://www.altera.com/literature/lit-exc.jsp>
- [16] ARM, *ARM922T Technical Reference Manual*
http://www.altera.com/literature/third-party/ddi0184a_922t_trm.pdf Rev 0
- [17] Altera, Technical Support Web Interface
<http://www.altera.com/mysupport>
- [18] Altera, *Introduction to Quartus II*
version 3.0, June 2003
- [19] Altera, *EPXA1 Development Kit Getting Started User Guide*
February 2003
- [20] Altera, *Excalibur Hardware Design Tutorial*
version 1.5, August 2002
- [21] Altera, *Using SOPC Builder with Excalibur Devices Tutorial*
version 1.1, June 2003
- [22] Altera, *Excalibur Web Server Demonstration*
Application Note 285, version 1.0, December 2002
- [23] ARM, *ARM Instruction Set Quick Reference Card*
http://www.arm.com/pdfs/QRC0001H_rvct_v2.1_arm.pdf
version 2.1, October 2003
- [24] RedHat, *GNUPro Toolkit Users Guide for ARM and ARM/Thumb Development*
http://www.altera.com/literature/third-party/gnupro_userguide.pdf