Master's Thesis

Hardware Implementation of MPEG Audio Real-Time Encoder

Joakim Enerstam and Jan Peman

September 1998

Supervisors

Anders Hansson / Glenn Jennings,
Luleå University of Technology

Björn Wesén,
Axis Communications AB

Abstract

Audio compression is a requirement for efficient transmission and storage of high fidelity digital audio. Recently developed algorithms are able to compress audio up to 12 times, without losing any audible quality. In this thesis we look at different audio compression standards with regards to both compression rate and sound quality. The chosen algorithm, MPEG-1 Layer III, is discussed in detail concerning complexity and structure. We have also looked at different hardware technologies, especially digital signal processor solutions, suitable for an implementation of real-time encoders. A software prototype of a real-time encoder was built, based on MPEG-1 Layer III standard. The encoder was tested and verified in a simulator.

Table of Contents

1 Introduction

1.1 Why real-time audio encoding?

The use of computers and digital equipment for different multimedia applications has increased tremendously over the last years. The same goes for the use of different networks to transmit multimedia data such as video and audio. Since neither the storage capacity in the computers, nor the bandwidth in the networks is infinite, there is a limit for how much multimedia data it is possible to manage. To be able to handle more data, some kind of compression might be in place. To manage high fidelity audio with CD quality you would need approximately 1.4 Mbit/s bandwidth. Using modern audio compression algorithms, it is possible to decrease the bandwidth up to 12 times. This explains the need for an encoder to compress the data, but not the need for real-time compression. The real-time demand is easy to explain. Imagine an encoder that should be able to compress data to deliver a bitstream that never stops, an Internet radio station, for instance. If it should be able to send 24h/day and not being able to compress the input in advance, the need for a real-time encoder is quite obvious.

1.2 The thesis background

The idea of this project came up after Björn Wesén did a similar thesis about a “DSP based decompressor unit for high fidelity MPEG-Audio over TCP/IP networks” also at Axis Communications in 1997. Since they had this work about a decompressor, there was a need for a compressor.

There exists already real-time hardware encoders in the market, but all of them are based on Fraunhofer’s own developed DSP (Digital Signal Processor) hardware. Licencing Fraunhofer’s DSP core is quite expensive, which reflects the products, using their code, price [1].

Since Björn´s thesis the new AAC standard has finished its development, and that was also one of the compression algorithms we looked closer on.

1.3 The goal of the project

The goal of this master thesis was to make a hardware implementation of an MPEG audio encoder.

Specifically the parts of the thesis was:

- Examine and chose an MPEG audio encoding standard that is possible to implement on hardware, with regard to compression ratio, performance, complexity, sound quality, parallelizing-ability etc.
- Implement and optimize the compressor in software using the audio-standard and reference compressor to make a first analyze of how much computation power and what types of operations that is needed to do a good real-time implementation.
- Chose a computational architecture to implement the encoder in hardware.
- Implement the compressor on the chosen hardware, with needed assembler-optimizations and verify that it fulfills the real-time demands.
- The implementation will be on self-developed hardware. If shortage of time occur, some kind of developer card or simulator can be used. Lower compressor quality might be necessary if hardware performance is too low.

The optimum goal was to produce a card with analog and digital (S/PDIF) input, and output from the hardware would be an MPEG compressed bitstream.

2. Digital Audio and Compression Algorithms

2.1 Introduction to Digital Audio

The sound that a human ear detects is basically changes in the air pressure. We are able to hear frequencies from 20 Hz to 20 kHz. This is the audio range that is interesting to save, in order to later reproduce the original sound and air pressure. Saving audio digitally has been very common lately, much thanks to the popular CD format introduced by Philips in 1982. The digital format has many advantages compared to the analog, such as better audio quality, processing abilities and less bandwidth consuming. The fact that computers and digital networks can not use analog sound makes digital audio a requirement.

The most common format for storing digital sound is the Pulse Code Modulation (PCM) technique. PCM is simply the sound sampled at a fixed rate and with a fixed number of bits representing the signal's amplitude. The PCM technique was first invented in 1937 by A.H. Reeves [2].

A PCM sound can use various numbers of bit and sampling frequencies, but 16 bit 44.1 kHz is by far the most common format for high fidelity audio, this is also the format recommended for "consumer applications". 48 kHz is recommended for "origination, processing, and interchange of program material" (used on DAT-players) and for "transmission-related applications" is 32 kHz the standard frequency (common in digital radio; see [3], p. 782). Using the Nyquist theorem 44.1 kHz allows reconstruction of sound with frequencies up to 22.05 kHz. This allows some imperfection in filters and other components and still having more than 20 kHz left for audio. The sampling frequency is linear to the number of bits needed for the signal, thus a 48 kHz sample uses 1.5 times more bandwidth than a 32 kHz.

Bits per sample.

Using n bits for each sample allows for 2^n quantization levels. The quantization can be done uniform (linear) or non-uniform (non-linear). In the uniform case, if a sample with the amplitude A and quantization level 1, a sample with 2*Amplitude will have a quantization of level 2 and so on. For each bit the signal to noise ratio (SNR) increases by 6 dB, thus a 16-bit sample has 96 dB SNR. This scheme is the most frequently used. The non-uniform case has a lot of different algorithms to quantize the digital sample, all of them are more complicated than the uniform scheme, but with its own set of advantages and disadvantages. Most of the non-uniform quantization schemes are based on the fact that one knows the signal characteristics and/or how the sound is percepted by the human ear.

Audio that is coded in 44.1 kHz 16 bit stereo uses 44100*16*2 bits per second. This means to be able to play a PCM file over a network, you need more than 1.4 Mbits/s in bandwidth. Storing one song, consisting of three minutes (180 seconds) audio coded in 44.1 kHz 16 bit stereo PCM format, uses 180*44100*16*2 bits, or over 30 MB computer storage.

These two examples give a hint that some kind of sound compression would be useful.

2.2 Audio Compression Techniques

2.2.1 Human Hearing and Auditory Systems

The human is not a perfect creature and the same applies for our ears and hearing. There are lots of different passages that sound has to travel before we can percept it in our mind. The first thing is the ear and it's physical limitations. This is for instance where our frequency range limit of 20Hz-20 kHz comes from. After the ear, the sound travels through the nerves and to the auditory cortex of the brain. Here the brain transforms the sound to different perceptions that we become aware of in our mind.

Loudness
Two sounds with the same amplitude can sound differently loud to the human depending on what frequencies the sounds have. This effect occurs because the human perception of determining a

sound loudness is not constant to the frequency. Looking at the curve in Figure 2.1 we can see that the human ear is most sensitive to sounds between 1000 Hz and 5000 Hz. Figure 2.1 shows the "loudness curve", that changes with the amplitude.

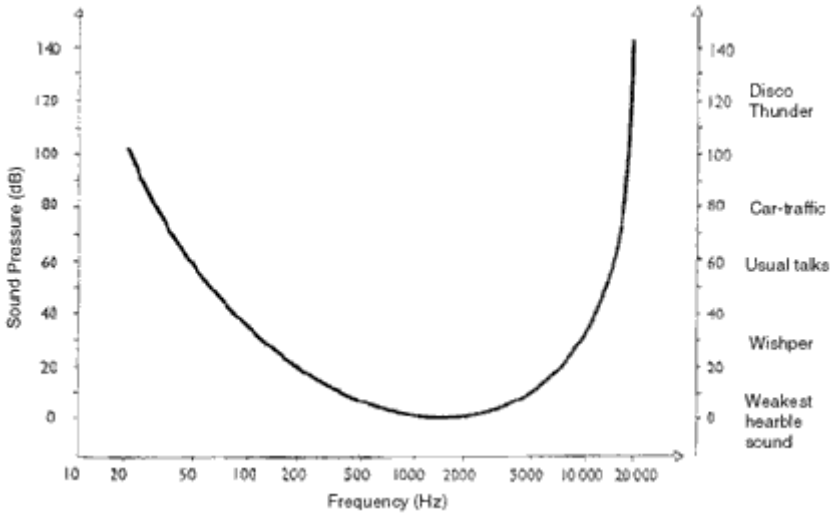


Figure 2.1. The human loudness [6]

Frequency Range

The 20-20kHz audio range mentioned before, has it's physical explanation. In the inner ear we have the cochlea. It consists of small sensory hair cells that sit on the basilar membrane. The hairs have different length that absorbs different frequencies. The cochlea in a cat is more sensitive, and is able to recognize frequencies from 100 Hz up to 40 kHz. The frequency range changes with age, you lose the ability to hear higher frequencies when aging.

Dynamic Range

The lowest air pressure variation a human can detect is 20 micro Pascal, this is measured at the frequencies we are most sensitive to (see Figure 2.1). [4] This sound pressure level is often used as a reference when describing acoustic sound pressure. Based on this reference as 0 dB we measure how strong other sounds is. A normal conversation is around 50-60 dB and the sound from motor traffic is about 80 dB. The ear can tolerate sounds at around 130 dB maximum, which gives a human the dynamic range from quietest to loudest, 0 to 130 dB.

Auditory Masking

Auditory masking is defined as "decreased audibility of one sound due to the presence of another" [5] p. 283. Auditory masking consists of frequency masking and temporal masking, which being described below.

- Frequency Masking

Frequency masking, also called simultaneous masking, is best explained with an example. For instance, if you have a strong tone with a frequency of 1000Hz, and also a tone a nearby at 1100Hz, which is 18 dB lower. The tone at 1100Hz is inaudible because it's being masked by the stronger tone at 1000Hz. This is due to the 1000Hz tone sound is louder and rather close in frequency. A louder sound masks a weaker and the closer in frequency you come, the louder other frequencies you can mask away. [4,6]

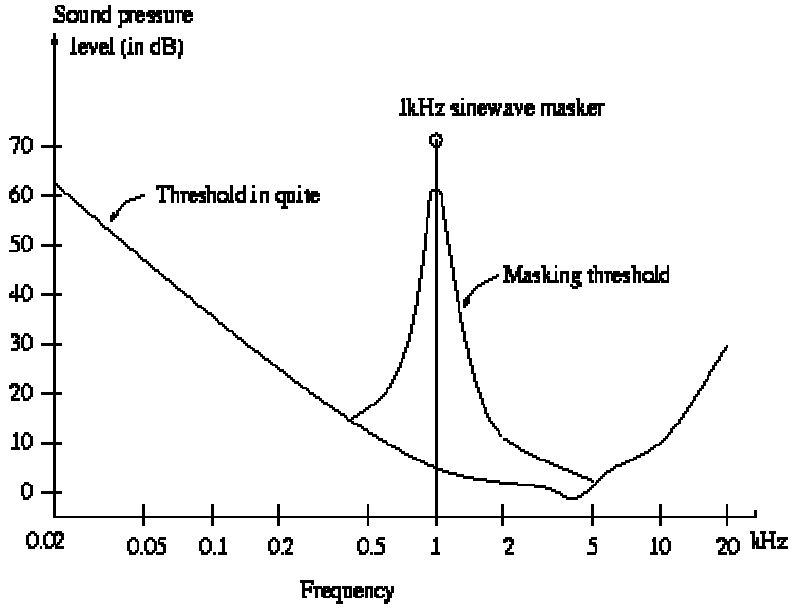


Figure 2.2. Shows how nearby frequencies being masked by 1kHz tone. [7]

Temporal Masking

Temporal masking effects occurs before and after a strong sound. If a sound is masked after a louder sound it's called post-masking, and if it's masked ahead in time it's called pre-masking. This may sound strange, but the pre-masking phenomena actually exist, even if it's only for a short moment (20ms). The post-masking, on the other hand, can be in effect up to 200ms. [4]

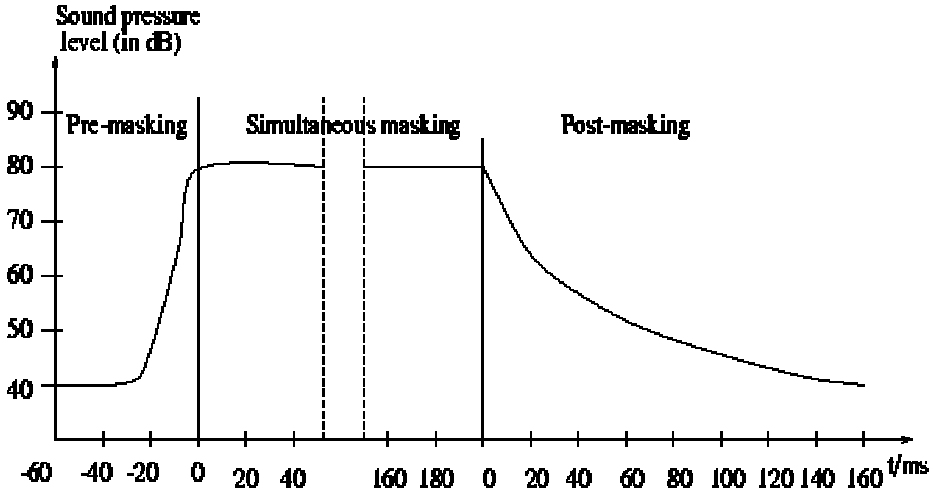


Figure 2.3. temporal masking [9]

Exploiting both frequency and temporal masking makes it possible to reduce substantial audio information, without any audible change.

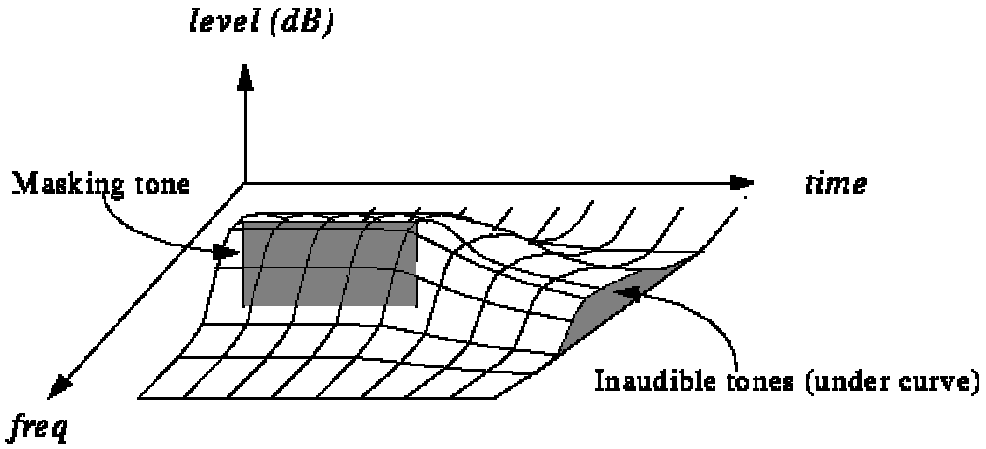


Figure 2.4 shows the combination of frequency and temporal masking. [8]

Stereophonic Redundancy
The fact that human ear not is able to detect the direction of low frequencies, is called Stereophonic Redundancy. For instance, this property is used in audio system with tree loudspeakers, two satellite speakers and one Sub Woofer.

All these imperfections, or limitations in the hearing, make it possible to leave out certain audio information, without affecting what we hear.

2.2.2 Waveform and Perceptual Encoders

There exist two types of audio encoders. First we have the so-called waveform encoders, which try to reconstruct the signal as exactly as possible after encoding and decoding. Differential Pulse Code Modulation (DPCM) is one example of waveform encoders. Each sample is compared to the previous sample, and only the difference between the values is stored, while a regular PCM stores the current value.

Adaptive DPCM (ADPCM) further refines this technique. If the signal remains at a low level, the ADPCM provides more resolution to the quantization steps near this level. When the level changes from the current range, the step size also changes. Compared to other compression techniques ADPCM requires very little processing power. ADPCM was later standardized as the G.721 standard. The perceptual encoders do not attempt to retain the input signal exactly as it was before the encoding and decoding stage. Instead it tries to ensure that the output sounds like the original to the human ear. Using knowledge about the ear properties and the human hearing limitations, the perceptual encoder removes parts of the signal that we are unable to notice.

Virtually all perceptual encoders transform the sound from the time-domain to the frequency domain, and then it splits the different frequencies into subbands. After this, the encoder uses its knowledge about how the ear works, to remove unneeded data. The masking effect is the most commonly exploited ear-phenomena.

Good examples of perceptual encoders are all the MPEG audio encoders and Sony's ATRAC [7] encoder, used in their MiniDisc systems. We will talk some more about MPEG encoders in the following chapters.

2.3 What is MPEG?

Motion Picture Experts Group (MPEG) is a working subgroup of the International Standards Organization (ISO) that generates generic standards for digital video and audio compression. MPEG defines the syntax of low bit rate video and audio bit stream and how to decode it. The algorithms for encoding is not specified by MPEG. This means that you always can develop new methods or algorithms to optimize the use of bits in digital audio, as long as you stick to the bitstream syntax. This is very important for further improvements of MPEG encoders.

2.4 MPEG-1 Layer III

2.4.1 History

In 1987, the Fraunhofer Institute started to work on perceptual audio coding. Together with other companies and institutes this finally, 1993, ended in a very powerful algorithm that was standardized as the MPEG-1 Layer III. [10] This standard is based on modern perceptual audio coding techniques that exploit the properties of the human ear, thereby made it possible to shrink down the original sound data from example a CD, by a factor of 12 without losing any sound quality. Even factors of 24 and more still maintain the sound quality that is better than reducing the sample rate with half. This technique works very well for high quality, low bit rate applications like Internet audio, digital audio broadcasting, etc.

2.4.2 Compression Modes

MPEG-1, also known as phase one, support one (mono) or two (stereo) audio channels with a sampling rate of 32, 44.1, or 48kHz. The compressed bitstream varies with fixed bitrates in a range from 32 to 224kbits/sec per channel. This gives a compression grade ranging from 2.7 to 24 times, depending on the sampling rate.

The MPEG-1 audio standard has three independent layers for compression. These three layers differ in matter of complexity, compression and audio quality. Layer I forms the most basic algorithm, while each successive layer increases the compression rate and gets more complex, both in encoder and decoder.

The Layer I algorithm is the simplest one and is best suited for bit rates above 128kbits/sec per channel. 384 audio samples are coded into every frame. Layer I use the basic filter bank found in each Layer of MPEG-1. For example, Philips Digital Compact Cassette (DCC), not produced anymore, used layer I at 192kbits/sec and channel.

Layer II is a bit more complex and improves the compression rate by coding data in larger groups. Layer II use 1152 samples/frame, which is the same as in Layer III. For example, Digital Audio Broadcasting (DAB) and Video CD use Layer II. MPEG layer 2 is also known as the MUSICAM standard in America.

Layer III is even more complex and is described more in chapter 2.4.3.

The second phase, MPEG-2 audio [11], was completed and became an international standard in 1994. This standard extends the first phase of standard with a set of additional features. The big difference is the support for multichannel and multilingual support.

MPEG-2 support up to five high fidelity audio channels and one low frequency enhancement channel. This is perfectly suited for digital movies where you want surround sounds. The standard also has support for up to seven additional commentary channels. This means for instance that you only need to send one channel of video along with seven different languages, thereby save lot of bandwidth.

Another feature is the additional support for lower, compressed bitrates down to 8kbits/sec. MPEG-2 also introduce support for 16, 22.05, 24kHz as well. The commentary channels are allowed to have a sampling rate that is half the high fidelity channel.

2.4.3 The Algorithm

An overview of the MPEG-1 layer III encoder algorithm is described by a block diagram in figure 2.5. The input audio stream passes trough a filter bank that divides the input into multiple subbands. The filtering is done in parallel with the psychoacoustic analysis that determines the signal-to-mask ratio of each subband. The noise allocation block uses the signal-to-mask ratios to decide how to divide the total number of code bits available. Finally the last block take the quantified and coded samples and format them into a valid MPEG bitstream.

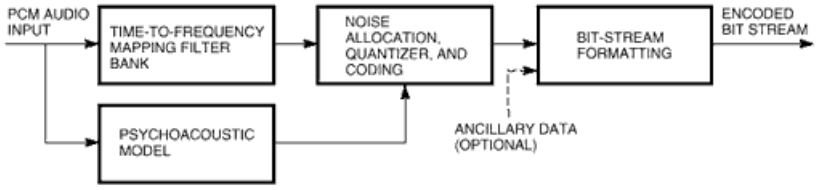


Figure 2.5 Block diagram of an MPEG-1 layer III encoder [12]

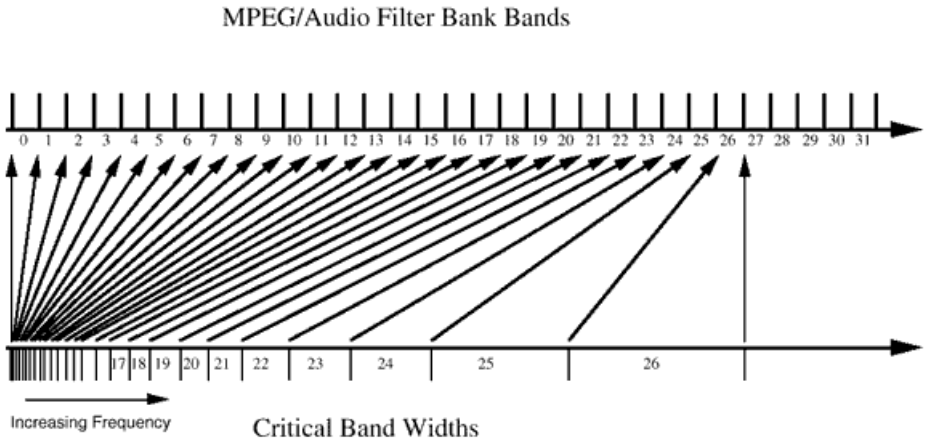
Psychoacoustic Model

There are two psychoacoustic models used in MPEG/Audio. Psychoacoustic model one has been used in Layer I and Layer II, while model two is used in Layer III. The latter has further improvements that better suit the human ear properties. The model two psychoacoustic analysis has two tasks to perform. Decide what blocktype to use, and calculating the signal to mask ratio.

First the model converts the audio to the spectral domain, using a Fast Fourier Transform (FFT) to get a good frequency resolution for a correct calculation of the masking thresholds [12]. The output from the FFT is first used to analyze what type of signal that is being processed. A stationary signal makes the model choose long blocks and a more transient signal results in a short block. The block type is later used in the MDCT part in the layer 3 algorithm. After this, the psychoacoustic model calculates the minimum masking threshold for each subband. These threshold values are then used to calculate the signal-to-mask ratio (SMR). The model then passes SMR to the noise allocation section of the encoder for further usage.

Polyphase Filter Bank

One of the most important block, is the polyphase filter bank which is used in all layers of an MPEG/Audio encoder. Its function is to split the audio signal into 32 subbands. These subbands are equally spaced in frequency, and do not accurately reflect the ear's critical bands, which is shown in figure 2.6 below. The bandwidth is too wide for the lower frequencies and too narrow for the higher frequencies, so the number of quantizer bits cannot be optimized for the noise sensitivity within each



critical band.

Figure 2.6 MPEG/Audio Filter Band Widths vs. Critical Band Widths [12]

The ear has a limited frequency selectivity that varies in exactness from less than 100 Hz for the lowest audible frequencies to more than 4 kHz for the highest [13]. Thereby the audible spectrum can be partitioned into critical bands that reflect the human ear frequency selectivity. Table 2.1 shows how wide these critical bandwidths are at most.

Approximate Critical Band Boundaries			
Band nummer	Frequency (Hz)[1]	Band nummer	Frequency (Hz)[1]
0	50	14	1,970
1	95	15	2,340
2	140	16	2,720
3	235	17	3,280
4	330	18	3,840
5	420	19	4,690
6	560	20	5,440
7	660	21	6,375
8	800	22	7,690
9	940	23	9,375
10	1,125	24	11,625
11	1,265	25	15,375
12	1,500	26	20,250
13	1,735		
[1] Frequencies are at the upper end of the band			

Table 2.1. Approximate bandwidth of the ears critical bands [12]

The filter is relatively simple, but gives a good time resolution with a reasonable frequency resolution. The polyphase filter bank is not lossless, even without quantization there is no possibility to recover the input signal exactly. Fortunately, the human ear is not able to hear the error, introduced by the filter bank. There also exists a frequency overlap between adjacent filter bands. Therefore a signal at a single frequency can effect two adjacent filter bank outputs.

Modified Discrete Transform

The Layer III process the outputs from the filter bank with a Modified Discrete Cosine Transform (MDCT), to compensate for the poor precision of the filter bank. This is done by further subdividing, the spectral output in frequency to provide better resolution. The MDCT transform is lossless compared to the polyphase filter banks.

Noise Allocation

Noise allocation is done in Layer III, while Layer I and Layer II use bit allocation. Bit allocation only approximates the amount of noise caused by quantization, while the noise allocation actually calculates the noise.

The allocation is done in an iteration loop that consists of one outer and one inner loop. The inner loop quantifies the spectral values from the MDCT using a certain step size and then Huffman codes the output values. If the number of bits required for coding the values exceeds the number of bits available for the chosen bitrate, the loop starts over with a new quantizer step size and runs the quantizing and Huffman coding again. The loop finishes when the quantized and Huffman coded values uses less or equal number of bits than the maximum amount allowed.

Now the outer loop has to check if the scalefactor for each subband has more than the allowed distortion and compares each scalefactor band with the data previously calculated in the psychoacoustic analysis. If any of the scalefactor bands has too much noise than the maximum allowed, the loop amplifies that scaleband and runs the inner loop again. The outer loop stops when one of the following conditions is fulfilled:

1. None of the scalefactor bands has too much noise.
2. The next iteration would amplify one of the bands more than allowed.
3. All scale factor bands has been amplified at least once already.

Since the loop is very time consuming, there might also be a fourth condition in a real-time application, that stop the loop and prevent the encoder to run out of time. [12]

Bitstream Formatting

The last step in the encoding process is to produce a MPEG compliant bitstream.
The bitstream formatter stores the encoded audio and some additional data in frames, where each frame holds information of 1152 audio samples per channel. A frame consists of header and audio data together with an optional error check and customizable ancillary data. The header describes among others, which layer, bit rate and sampling frequency that is being used for the encoded audio. The Huffman coded data and its side information are placed in the audio data part, where the side information tells what block type, Huffman tables and subband gain factors to use.

2.5 MPEG-2 Layer 7 (AAC)

AAC is an acronym for Advanced Audio Coding and is a new state-of-the-art MPEG audio encoding standard, IS 13818-7 [17]. The AAC development was initialized in 1994 by the MPEG group and was at first called NBC, which stands for Non Backwards Compatible because it wasn't compatible with the old layers I, II and III. Thanks to this, the AAC format doesn't need to take consideration for any old formats that might otherwise add unwanted limitations and complexity to the standard.

AAC will be one of the sound algorithms used in the new MPEG-4 standard. The AAC standard supports up to 48 channels, of which 0 to 16 are Low frequency element (LFE) channels, 0-16 “coupling channels” that can do either efficient multilingual/voice over and 16 single channels. The sampling frequency of the LFE channel corresponds to the sampling frequency of the main channels divided by a factor of 96. This provides 12 LFE samples within one audio frame. The LFE channel is capable of handling signals in the range from 15 Hz to 120 Hz.

If you compare the encoding techniques for AAC with it's predecessor, Layer III, you'll find that they have much in common. The major differences is the optional tools introduced in AAC, like temporal noise shaping, prediction and gain control.

The AAC standard uses 10 different "tools" which some are required and other are optional [15]. Each tool adds more complexity to the algorithm and thus more processing power. In addition to those tools, an encoder needs also a pshychoacoustic phase.

Gain Control
* Filter Bank
Temporal Noise Shaping (TNS)
Intensity/Coupling
Prediction
M/S
* Scalefactors
* Quantizer (the only lossy part of AAC)
* Noiseless Coding (Huffman coding)
* Bitstream Formatter

* means the tool is required, other optional.

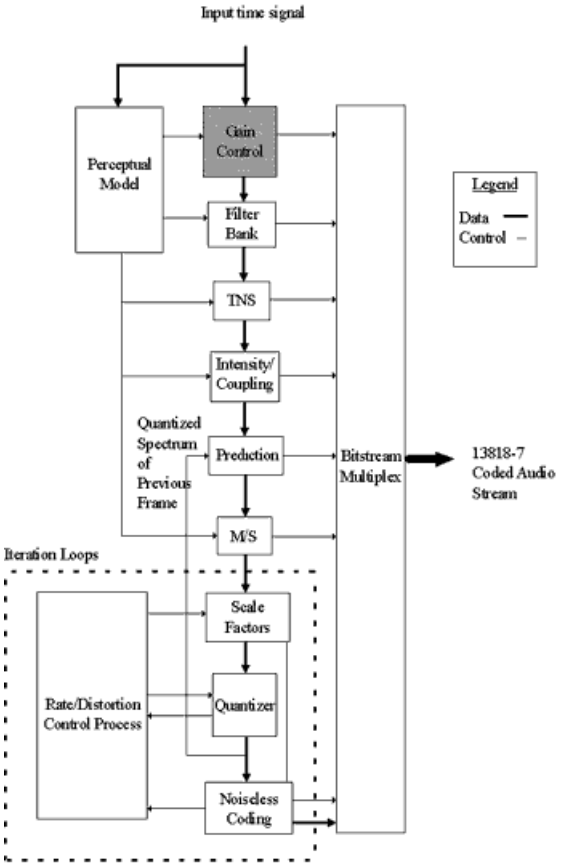


Figure 2.7 Block diagram of AAC encoder [19]

There exist three different profiles for AAC to allow tradeoff between quality and processing power/memory requirements:

Main Profile
The main profile encoder/decoder uses all the tools and is the profile with the highest quality and complexity.

Low Complexity Profile
The low complexity profile is used when available memory and processing power is limited. In this profile prediction and gain control tool are not permitted and the temporal noise shaping order are limited.

Scalable Sampling Rate Profile
In the scalable sampling rate profile the gain-control tool is required, but prediction and coupling channels are not permitted. Like the low complexity profile the TNS-order are limited.

Tests by the MPEG group shows that a Main Profile decoder uses 40% CPU time of an 133 MHz Pentium for a 2 channel 48kHz at 64kbit/channel and 25% using a low complexity profile decoder [16].

Some of the new tools:

- Gain control tool

This tool is only used in the SSR profile. It consists of a Polyphase Quadrature Filter (PQF) and a gain detector/modifier [14]. The PQF divides the input signal in four frequency bands with equal width. Then the detector produces data used for the modifier. The modifier then controls the gain for each frequency band from the PQF. Output from the tool is a gain-controlled signal, still in the time domain, and gain control data, a field added to the side-information in the bitstream. The detector has one frame delay.

- Temporal Noise Shaping tool

The Temporal Noise Shaping (TNS) tool is used for encoding "pitch-based" signals such as speech and other transient signals, without decreasing the coding efficiency. These kinds of signals are hard to encode in an ordinary perceptual encoder, like mp3 and other standards [18]. By using this tool, the quantization noise within each transform window will be decreased for transients. When the tool finds a transient, the encoder codes the time domain data, instead of the spectral data. This is because a transient signal in the time-domain requires fewer bits to represent than a transient signal in the frequency domain (see table below).

Input Signal		Optimum Coding
Time Domain	Frequency_Domain	Direct Coding
Sinussignal	Diracpuls	Coding of spectral data
Diracpuls	Sinussignal	Coding of time domain data

- Prediction tool

Prediction is used for an improved redundancy reduction and is especially effective in a case of more or less stationary parts of a signal. Prediction is a technique commonly established in the area of speech coding systems. Prediction can be applied to every channel using an intra channel predictor, which exploits the auto-correlation between the spectral components of consecutive frames.

2.6 Other compression techniques

There are other audio compression techniques for high fidelity audio than the ones MPEG has developed. Sony ATRAC (used for MiniDisc systems), Dolby AC-3 and TwinVQ to mention some of them. Those have all similar techniques for their compression, but they also differ in some encoding areas, and we will mention what's specific for each of them.

- Sony ATRAC

In the early 90:s Sony realized that there was a need for a portable, recordable digital audio system. This resulted in their first MiniDisc system presented in 1992, using their ATRAC [20] sound compression algorithm. ATRAC, which is an acronym for Adaptive Transform Acoustic Coding, operates exclusively at 16 bit samples with a sampling frequency of 44.1 kHz and provides a compression rate of 1:5.

Algorithm

The sound is first divided into three frequency bands with frequencies, 0-5.5125 kHz, 5.5125-11.025 kHz and 11.025-22.05 kHz. After this, the bands are transformed into the frequency domain, using a Modified Discrete Cosine Transform (MDCT). Like the MPEG Layer-III algorithm ATRAC uses an adaptive block length to avoid pre-echo effects in transient signals. A long block is 11.6 ms and a short block is 1.45 ms or 2.9 ms, depending on frequency band. The high frequency band uses 1.45 ms short blocks, and the two lower frequency bands uses 2.9 ms. Unlike the Layer-III algorithm, the blocksize can be selected individually for each bands. After the MDCT the spectral values produced are quantized and saved by a bit allocation process. This is also very similar to the MPEG algorithm.

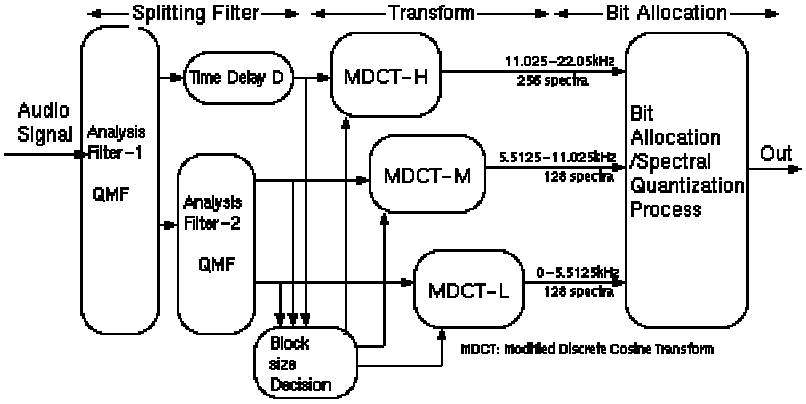


Figure 2.6 Block diagram of ATRAC encoder.

- Dolby AC-3

In 1989 Dolby introduced their AC-2 audio compression system, being one of the pioneers in the fields of audio compression [21]. This system later emerged to the AC-3 system, that "aervde, inherited?" many of the AC-2 ideas. The AC-3 system is commonly used in cinemas, but has also been adopted for the North American HDTV standard and is used as one of the standards in DVD.

Algorithm

The AC-3 bitstream consists of synchronization frames [22]. A frame consists of 6 audio blocks with 256 samples, making a total of 1536 samples per frame (32 ms @ 48kHz). No information is shared between different frames, making it possible to decode one frame independently, without information from any other.

First there is a transient detect phase, to decide block-switching flags. The input signal is segmented into a hierarchical tree with different block lengths and is then examined for transient detection Using the block switch flags decided in the transient detect phase, the signal is transformed into the frequency domain using a MDCT like many other algorithms. The MDCT is one 512 point or two 256-point transforms, depending on the switch flags. The spectral values are then divided into 50 bands, not equally spaced but tried to match the human ears critical bandwidth. The spectral values are then fed to a bit allocation algorithm, analyzing the audio and coding it with respect to masking effects. The coded values consist of an exponent and a mantissa. The exponents are 5 bit values, and are transmitted by using delta values based on the previous exponent. There are three different exponent strategies. The mantissa bits depends on the audio masking analysis.

- TwinVQ

Twin VQ (Transform-domain Weighted Interleave Vector Quantization) is a transform coding like MP3, AAC or AC-3. It will like AAC, be used in the upcoming MPEG-4 standard and uses some tools of AAC like interframe backward prediction, but the encoding of music is totally diffrent. The Twin VQ standard uses a pattern library wich is prepared in advanced and compared against the the bits in the audio stream. The standard patterns that provides the closest match is selected, and a number associated with that pattern is transmitted as the compression code.

3.0 Hardware technologies

To do heavy calculations, like mpeg encoding, there are many different hardware solutions to chose from. We have looked at a few and will discuss them in regards to cost, availability, power consumption etc.

3.1 DSP

The DSP technology is known to be a fast, cheap and not consuming much power. We have looked at two of the fastest floating point DPS:s known today, Analog Devices (ADI) SHARC ADSP-21065L and Texas Instruments (TI) TMS320c6701, and compared the different technologies and other aspects of the DSP:s.

3.1.1 SHARC ADSP-21065L

The SHARC ADSP-21065L [23] is a new member in the 2106x family featuring high speed floating point DSPs. It is fully function and code compatible with its predecessor ADSP-21061. The main difference is that ADSP-21065L have 544Kbits of internal memory, which is about half the size that ADSP-21061 have. This makes it able to clock the ADSP-21065L in 60MHz and thereby a performance increase of 50%. The reduced size in memory also gives a lower price, only $10 per chip (september 1998). The Super Harvard Architecture Computer, (SHARC) is able to execute three instructions in parallel, one ALU, one multiply and one shift instruction. This makes 180 MFLOPS in theory (peak performance). The processor can simultaneously fetch two operands and one instruction from the instruction cache, all in one cycle. This makes it able to clock the 21065L in 60 MHz. Figure 3.1 shows the block scheme of ADSP-21065L.

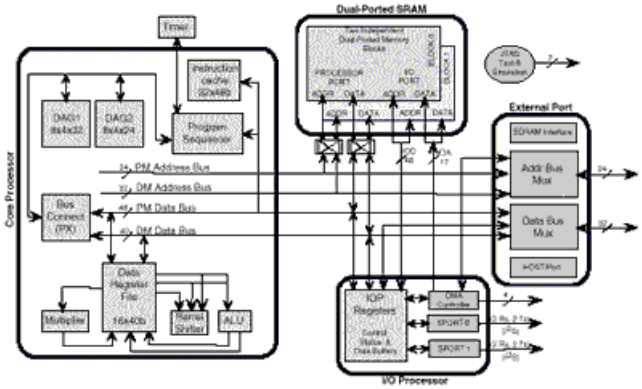


Figure. 3.1 Block scheme of ADSP-21065L

ADSP-21065L calculates a 1024-points complex FFT, radix 2 in 18.221 cycles (0.310ms) and computes a division in 6 cycles. ADSP-21065L has built in interface to standard SDRAM, which operate in 60MHz.

3.1.2 TMS320C6701 DSP

The Texas instruments TMS320C6701 [24] DSP, member of the C6x family, achieve high performance trough the increased level of instruction parallelism. TI C6701 can process up to eight instructions in parallel, which is made possible by the 256 bits wide instruction fetch.
Since you probably not always can use all eight functional units at the same time, every fetch packet may contain of multiple execute packets, which make it possible to reduce the number of NOP:s in program memory. An execution packet consists of those instructions that actually can execute in parallel. If two or more execution packets together contain no more then 8 instructions, those execution packets can all be in same instruction fetch. C6701 runs on 167MHz and is able to execute 6 floating point operations per cycle, of which 2 is multiply instruction. This means 1 GFLOPS peak performance and makes a 1024-points complex FFT in 20,716 cycles (0.124ms), with Texas Instruments hand optimized assembler code. The CPU has two data paths (A and B) which operate in parallel. Each data path has four functional units (.L1, .L2, .S1, .S2, .M1, .M2, .D1, .D2), which execute logic, shifting, multiply and address operations. One register file of 16 32-bits registers is also connected to each data path. There is one single data bus connected between the two register files.

This means that only one register read and write access is allowed across the register files each cycle. A functional unit in data path A, for instance, is only able to work with the registers in register file A, except for reading from one register on side B. Figure below shows the block scheme of C6701.

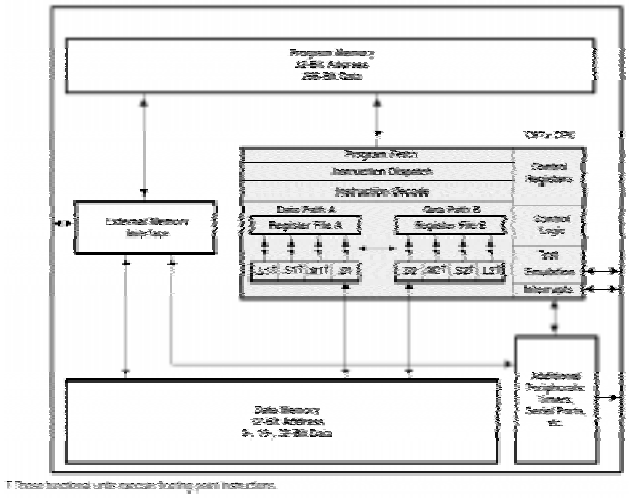


Fig 3.2 Block scheme of TMS320C6701

The C6701 has 1Mbit On-Chip SRAM divided into one 512Kbit internal program/cache and one 512Kbit internal data memory. The 512Kbit data memory is totally user configurable in any way. C6701 also has 32bit External Memory Interface (EMIF), which built in support for synchronous SDRAM, SBSRAM and also asynchronous SRAM and EPROM. Both little-endian byte/half-word addressing and big-endian addressing are supported by C6701. The today price for TMS320C6701 DSP is $99, but the DSP is not yet available in silicon.

3.3 Custom chip

3.3.1 Trimedia TM-1000

The Philips TriMedia[25] TM-1000 is a custom microprocessor, which have the usual DSP instructions, but also special audio, video, graphics and telephony peripheral blocks. TM-1000 is a DSP/CPU 32-bit VLIW core processor with 32 KB instruction cache and 16 KB data cache. TM-1000 has 5 issue slots and 27 functional pipelined units. At each cycle up to five of the 27 functional units can be used simultaneously. TM-1000 has both integer and floating-point arithmetic units and parallel DSP-like units, which all can access 128 32-bit registers. While the registers not are separated into banks, like the c6701, any operation can use any register for any operand. This Chip is available for $50.

4 Choosing Algorithm and Hardware

4.1 Choosing compression algorithm

To decide what compression algorithm we should choose, we looked especially at the AAC and Layer 3, both developed under the wings of MPEG. There are some important factors to examine when you need to encode in real-time. Many of them are closly related to each other, so it is hard to just look at them one at a time. We have looked at some of the factors in the discussion below.

Complexity
For a realtime application, the complexity is a very important factor when choosing a compression algorithm. If too many calculations are needed, the hardware demand might be too much, resulting in extremely complex and expensive hardware solutions.

The MPEG group has done some extensive tests about the complexity of both AAC and Layer 3 decoding. Using the main profile version of the AAC decoder, it is about 2-3 times more complex than Layer 3. Since no comparasion for the encoders were found, we did one ourselfves, based on the reference encoders. We found out that the AAC encoder took over twice as much time as the Layer 3 to encode one test sample.

Memory
The memory requirement for the different encoders was hard to find out. We planned to analyze the requirement of RAM, ROM and program memory, but could not find out any of them. We can only guess that AAC uses more memore because of the extra tools it employs compared to Layer 3.

Sound quality and compression ratio
Sound quality is very close related to the bitratio. You can achieve a very good sound quality with MPEG-1 Layer 1 for instance, but then you need a bitrate of about 384 Kbps, wich is a compression ratio of only about 4 compared to the original input (44.1 Khz 16 bit stereo PCM in this case.). If we look at a high-fidelity sound quality without any noticeable degradation in soundquality, and use that for comparing the compression ratio between the different encoders, we find that AAC has a better sound quality than Layer 3 at the same bitratio. This has been shown in tests performed by the MPEG audio working group and presented at the different mpeg conventions around the world. The tests shows that at 96kbit/s gives a better sound quality than Layer 3 at 128kbit/s. This means that the same quality as layer 3 can be achieved at 70% of the bitrate if using AAC. [26]

Readable and well arranged standard?
We have read the two ISO standards for Layer III and AAC quite much. To understand the algorithms used in the different audio standards, the readability is very important. The standards formal names in terms of ISO are ISO/IEC 11172-3 for Layer III and ISO/IEC 13818-7 for AAC. The AAC standard is a bit more comprehensive and explains the standard pretty good. It's logically divided in parts about each tool in the standard. The Layer 3 standard, which also includes the layer 1 and 2, is somewhat more difficult to read and understand.

Reference source code
Before we choosed a compression standard, we looked very closely on the quality of the reference code that existed for the two MPEG standards. Since it's virtually impossible to write your own encoder only looking on the ISO standards, good reference code is an important factor when choosing between the standards. The Layer 3 code is publically available and is called dist10 [27]. The AAC code, on the other hand, is not released to the public yet, since the standard is not approved by ISO yet.

We tried the two encoders and found out that AAC was more than two times slower than Layer 3. We also listened to the output from the encoders and found out that the Layer 3 encoder produced better sound quality at 128 Kbps than AAC at 96 Kbps, for stereo audio samples. According to MPEG tests AAC at 96 Kbps should sound better, but this is not the case for the reference code.

Conclusion
AAC has better compression for the same sound quality and a more readable standard, but the Layer 3 is less complex, probably consumes less memory and has better reference code.
Based on the discussion above, we chosed Layer 3 for our encoder.

4.2 Choosing hardware technology

We have looked very briefly to the PPC and found that it is very good on floating point operations. The PPC are also cost effective and low power consuming. While there already exist many encoders for main purpose processors, the PPC solution would not be any challenge. Trimedia TM-1000 has very nice MPEG functions, but we discovered this chip late in our project. If we had known about the Trimedia chip in the beginning of our master thesis, we surely had looked closer to this interesting chip. With the reasons above we have choose to go for the DSP solution.

4.3 Looking at DSP tools

When choosing DSP we have also looked at the tools that are included in the development kits.

4.3.1 SHARC EZ-Kit Lite

SHARC EZ-Kit Lite board comes with documentation and software development tools at a relatively low price. The board has an ADSP-21061 that is running at 40MHz and is code and function compatible with the ADSP-21065L, which at this moment not yet is released. The board has an RS-232 interface for program download. You can also access a 16bit stereo audio codec, which support both input and output trough a serial port that connects directly to the SHARC processor. The software tools included with the EZ-Kit are a C compiler, assembler, linker and simulator. The simulator is Windows based and is integrated with the CBUG C source debugger. It models system memory and I/O according to the contents of the systems architecture file. With the architecture file you should be able to specify how to divide the total amount of memory into different parts, like heap, stack etc. This was not the case in EZ-KIT Lite. The architecture file was hard coded into the program. Thereby we are where stuck to the one memory configuration which not fit us at all. All hardware registers and memory are displayed in separate windows. It support single step execution, break points and break conditions. It is also possible to alter and make changes in runtime to the register and memory contents. Plotting memory contents is also possible, useful when verifying functionality of the application.

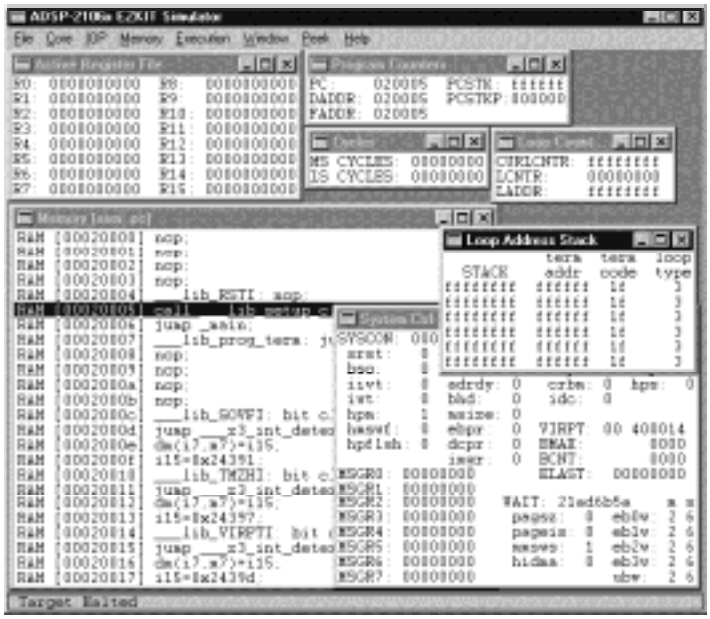


Figure. 4.1 picture of ADSP-2106x EZKIT Simulator

The C compiler has a set of ANSI-standard functions, including functions used in signal processing. The library also includes signal processing functions developed by Analog Devices, like DSP filters and FFT:s. All libraries are supplied with source code. Assembler programming is usually rather hard, but with the assembler used in SHARC DSP, Analog Devices has tried to make this stage easier. A usual algebraic statement like, r = x + y is coded in assembly language as (f0 = f1 + f2), instead of (add f1, f2, f0). This is supposed to make the assembler

programming more natural and intuitive. This may be for a beginner, but for an experienced programmer this could be even more confusing.

4.3.2 TMS320C62x Evaluation Module (EVM)

Texas Instruments (TI) provide the TMS320C62x Evaluation Module (EVM) that comes with the fixed-point TMS320C6201 DSP on a PCI-card. This was really not an option since we did not plan to make a fixed-point version of the encoder. TI also provides a full functional, 30-day time-limited versions of compiler and simulator software. The simulator supports all the features that the simulator in the SHARC EZ-Kit Lite supported. But with a big difference, you are able to specify external ram. The internal memory are also totally user configurable, i.e. in mean of stack size, heap size, etc. A useful command line window is also provided. A C-compiler, assembler optimizer and linker are also included.

4.4 Choosing DSP

When choosing DSP there are some properties that we thought are rather important.

- The DSP must work with at least 24bits data operations.
- It must support floating-point, while a big part of the algorithm work with floating point, and we don’t want to spend our time converting to fixed-point, while we know that the time will be short.
- If the DSP not is available in silicon, there should at least be a good simulator to use.
- A C-compiler is a requirement, because the MPEG-1 Layer III encoder algorithm is very big and complex, so assemble writing the whole encoder is not to think of at all. While there are no C-compilers today which produce enough good optimized assembler code, the DSP should also be easy to program in assembler.
- Of course it should be as inexpensive as possible.

Fraunhofer IIS [28] has managed to construct a real-time encoder, which use only two DSP320C31. Mostly of the MPEG-1 Layer 3 algorithm is developed by Fraunhofer IIS themselves, and therefore we probably need more computational power than just two TMS320C31 to reach the real-time demand.

All DSP:s we looked at supported both 32bits and floating-point operations. Now the only DSP:s left of interest was Texas Instruments TMS320C6701 and Analog Devices ADSP-21065L. TI did not provide any development board for there floating-point version c6701. We then ordered the SHARC EZ-Kit that comes with the ADSP-21061, which supported floating-point and was code and function compatible with the ADSP-21065L. After verifying that the development board was working and some manual readings, we discovered that the simulator wasn't able to simulate extended memory. This is a very important feature, while the different part of the MPEG-algorithm is dependent on data from previous part. Without that feature we would have to make a lot of extra work when verifying that different part really works. Analog Devices also provided a simulation tool called "Visual DSP", but this was in a beta and cost about $3000(too expensive). So we went for the second option, the free (30 days evaluation time) simulator from TI.

5 Implementation

5.1 Developing Environment

The MPEG audio reference sourcecode from ISO was written primary for UNIX. Since Linux is both free and has a very good and neat developer environment, including powerful tools such as gcc, gdb and emacs, we chosed Linux as operating system for algorithmical changes and optimizing of the mp3 code. Our Linux-box is running a standard RedHat 5.0 installation on a Pentium Pro 200MHz CPU with 64MB memory.

For DSP development we used WindowsNT as operating system since all DSP tools we used were made for the Windows platform like the TI 6701 and the ADI 21x environment. The machine we were running the simulator on was a PentiumII 266MHz with 128MB memory.

5.2 Encoder/Decoder

A decoder task is reversed the encoder, with one a big difference. The side information in the MPEG-1 bit stream gives information of i.e. what table to use when Huffman decoding, and therefore no calculations are needed when choosing table.

Layer	Complexity	
	Encoder	Decoder
I	1.5-3	1.0
II	2-4	1.25
III	>7.5	2.5

Table 5.1 Complexity value by Philips [29]

In table 5.1 the Layer I decoder is used as a complexity reference and is given index 1.0. When running the encode/decoder on a general-purpose machine, the complexity has to do with the amount of processing power and memory, while the complexity in case of a dedicated IC is the silicon area. Observe that the complexity value “>7.5” in Layer III encoder, is the minimum theoreticle value. The Layer III is so complex that it is impossible to calculate a maximum complexity value.

5.3 Implementation and Optimizing

The reference source code for MPEG-1 layer III was implemented with regards to understanding the algorithm, and not for speed. This was very obvious to us when we first tried it. It took 128 seconds to encode a 10 seconds sample on our Linux box. This code was used as the framework and reference to verify the functionality of our MPEG encoder.

To reduce the code size, complexity and also memory consumption, we decided early in the project to limit our encoder to 44.1 kHz 16 bit at 64kbps per channel (mono or stereo). This is the most common audio format, with a quality comparable to a music CD. Another limit is that we don’t support joint stereo.

We will discuss some of the major optimizing we have done in the sections below.

The profiling facility gave us a good hint about what functions that we should concentrate our efforts on, to be able to build a real-time encoder. Note that all numbers from the reference source are run on the Linux system with a Pentium Pro CPU. Most certainly those are all different when running on the target DSP, but anyhow the numbers give a hint about what functions that need to be rewritten and optimized.

Psychoacoustic analysis

The psychoacoustic analysis and filter/window subband is in theory done in parallel, with no connection to each other, but since a computer is sequencial, one of them has to be calculated prior to the other.

The most demanding function in the analysis is the 1024-fft, using over 8% of all cpu-time on Linux. Since all major DSP manufacturers provide hand optimized efficient assembler code for fft-calculations, we haven't worried about optimizing the fft-function. The rest of the psychoacoustic analysis takes about 3 % of the total cpu-time and is very memory consuming since it needs lots of precalculated tables to generate the minimum masking threshold and calculating the signal-to-mask ratio (SMR). A pre-calculated hann-window for both long and short blocks is also needed for the fft.

> One FFT on C6701 takes 20.665 cycles. Two fft:s is calculated per channel.
> 38 frames per second gives 20.665*152 = 1.9% CPU time for realtime encoding.

In the our design, figure 5.1, of the MPEG encoder, we don't use the psychoacoustic analysis.By ignoring that, we don't calculate any blocktype and the SMR ratio is never set. Not using the SMR is equivalent with only running the outer loop one time in the noise allocation iteration loop, this because the scalefactor bands will never have too much noise. (see the noise allocation section in 3.4.) This saves alot of computations and memory, but sacrifices some of the high quality in theory. Using non-scientific lissening tests we got no noticeable change in sound quality.

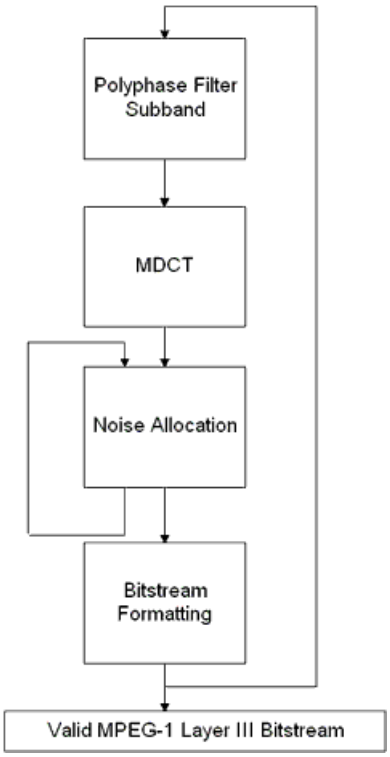


Figure 5.1. Design of the encoder

Modified Discrete Cosinus Transform (MDCT)

The MDCT processes each subband output from the polyphase filterbank. The MDCT was originally called 18*32 or 12*6*3 times, depending on long or short block is used. Since we only use long blocks, the MDCT is always called 18*32 times. The orinially MDCT calculated 18*36*2=1296 floating point multiplications, we used a modified variant of Lee's Fast DCT algorithm [30] that uses 244+36=280 multiplications. See appendix A for source code.

These changes is purely done in the algorithmical stage, thus the earned speedup will be in effect for both DSP and other architectures. With regard to the overhead in start and end of function calls for saving registers etc, we get a speedup of almost 5 times.

Noise allocation (inner loop, Huffman coding and quantization)

The noise allocation part is by far the most demanding part of the whole encoder. This is mainly because of the quantization part, but the Huffman encoding has also a share of the complexity. Since the quantization and Huffman code calls exists in a loop within another loop, there are three ways to optimize the noise allocation part. The first is to optimize the quantization and Huffman coding, the other two is to reduce the number of loops, decreasing the calls to the functions.

```
BEGIN outer_loop
  BEGIN inner_loop
    quantize
    Huffman code
  END inner_loop
END outer_loop
```

The outer_loop has been reduced to only run one time, this because we have no SMR calculation and no psycho acoustic analyze. The inner_loop, on the other hand, can't be optimized in the same way, since it would violate the number of bits required and increase the bitrate from the allowed 128kbit. Instead we use a "trimmed" quantization step size, to get a "good enough" quantization as soon as possible. The step size is chosed so the inner_loop runs only one or maximum two times.

The Quantization

The quantization formula:

$$ix(i) = \text{nint}\left(\left(\frac{|xr(i)|}{\sqrt[4]{2}^{\text{qquant}+\text{quantanf}}}\right)^{0,75} - 0,0946\right)$$

which run 576 times and the power function uses alot of cpu time, in our case on the DSP a power() takes way too much time. The "nint" function round the the calculated value to the nearest int. We optimized the Quantize function by using precalculated tables instead of the slow power function, se appendix b for source-code.

5.4 DSP Specific Optimizations

Since the TMS320C6701 is highly parallell you have to take that in mind when writing your C-code. A small change in the C-code can produce code that exploits the parallell execution stages much better. Note that some of the changes in the code would even increase the execution time on an ordinary CPU, but run faster on the 6701 DSP.

Example optimization of ix_max():

ix_max() is a function in our encoder that returns the maximum value from a vector. Using the original C-code we got an output from the TI compiler that only was able to compare one value from the vector every second clockcycle.

```
for ( i = begin; i < end; i++ )
{
  if ( ix[i] > max )
    max = ix[i];
}
```

Original C-code for ix_max() function.

```
LOOP:       ; PIPED LOOP KERNEL

            CMPGT   .L1     A3,A0,A1    ;compare the value
||  [ B0]   B       .S2     LOOP        ;branch to beginning of loop

    [ A1]   MV      .S1     A3,A0       ;move into another register for storing
||  [ B0]   SUB     .L2     B0,1,B0     ;decrease the counter value
||          LDH     .D1T1   *A4++,A3    ;fetch value from vector

; The || means that that the instruction runs
  in parallell with the previuos instruction.
```

Generated DSP assembler from original C-code

Knowing that not all resources are fully used, we can improve performance by unrolling the loop. This is done by accumulating the even elements ix[i], into max1and the odd elements, ix[i+1] into max2. After the loop max1 and max2 are compared to each other.

```
for ( i = begin; i < end; i+=2 )
{
  if ( ix[i] > max1 )
    max1 = ix[i];
  if ( ix[i+1] > max2 )
    max2 = ix[i+1];
}
```

C6701 optimized unrolled loop.

By unrolling the loop, the C-compiler produce assembler code that are able to do one compare each cycle. This means that the loop now is twice as fast as before.

```
LOOP:       ; PIPED LOOP KERNEL

   [ A1]  MV        .S1    A4,A0
||        CMPGT     .L1    A5,A3,A1
|| [ B0]  B         .S2    LOOP
||        LDH       .D1T1  *++A6(4),A4

   [ A1]  MV        .S1    A5,A3
||        CMPGT     .L1    A4,A0,A1
|| [ B0]  SUB       .L2    B0,1,B0
||        LDH       .D1T1  *+A6(2),A5

; The || means that that the instruction runs
  in parallell with the previuos instruction.
```

> DSP assembler generated from C6701 optimized C-code.

Giving the compiler small hints about how you want your code to compile and information about the datatypes beeing used, helps alot to get more efficient code. The rewriting of the code above is just one example. Other hints to the compiler include declaring what variables that are constant, minimum loop count information and optimize data flow bandwidth to and from memory [31]. When giving all these hints, the c6x compiler is rather efficient.

To optimize the code further in the ix_max() example, you need to write linear assembler. This means that you use the 6701 assembler language. In this example we won't get any faster code by using linear assembler. This is due to more units (see section 3.4) are needed then available, when doing more then one compare instruction per cycle. All these instructions work with the either the .L, .S or .D unit. C6701 got two of each unit, this makes 6 available units. The .M1and .M2 units are only used for multiply instructions. Making two compare instruction each cycle, demands 2 LDH, 2 CMPGT, 2 MV, 1 SUB and 1 B instruction. Thereby we need eight units when only sex units are available. This shows

that the C-compiler produced fully optimized assembler code, and there is no need for linear assembler writing.

If you don't reach the performance you hoped for, the last step is then to write "real" C6701 assembly code. This is not recommended by TI and the manuals they provide tells nothing about this topic, but it should definitively be possible.The problem is that with the architecture and behavioural of the DSP you have to keep too much in mind, but it might be worth the trouble.

5.5 DSP Timing Analysis

For verifying that our code in the simulator runs in realtime, we timed the different functions in terms of clockcycles. Since the sampling rate we use is 44100 samples/second and one frame processes 1152 samples, one frame represents 1152/44100 (26.12E-03) seconds. The TMS320C6701 runs at 167 MHz, and one frame should then take 167*1152/44100 miljon cycles maximum, for real-time performance. That gives us 4.362.449 cycles to spend on each frame.

Cycles for each frame	stereo	mono	
window +filter subband	871.789	548.161	50%
MDCT	281.005	140.564	62%
iteration_loop	2.454.325	1.649.332	67%
bitstream formatting	365.350		
theoretical mem copy	12.305		
	3.984.775		

Note that no function for input/output is in the calculation. With 3.984.775 cycles we use 91% of the DSP capacity, this gives us 377.674 cycles to copy samples from the input buffer, which should be enough. Note that some functions may take different long time depending on what input data (sound) that is beeing processed. Its important to have enough time margin to cover the worst-case frame, otherwise one might violate the real time constraints.

6 Results

We found out that MPEG encoding was not a simple algorithm. It requires lots of computational power for realtime, but it was not impossible to implement.

The parts of the thesis are repeated here, along with a short summary of the results:

- Examine and chose an MPEG audio encoding standard that is possible to implement on hardware, with regard to compression ratio, performance, complexity, sound quality, parallelizing-ability etc.
  - We looked at MPEG-1 Layer-3 and AAC. The Layer 3 was chosen.

- Implement and optimize the compressor in software using the audio-standard and reference compressor to make a first analyze of how much computation power and what types of operations that is needed to do a good realtime implementation.
  - We analyzed it and the result was that fast floating point multiplications was needed.
- Chose a computational architecture to implement the encoder in hardware.
  - A DSP has powerful and fast floating point multiplications. Texas Instruments 6701 DSP was chosen.
- Implement the compressor on the chosen hardware, with needed assembler-optimizations and verify that it fulfills the real-time demands.
  - We implemented it in dsp simulator. The real-time demands and output audio quality are verifyed.
- The implementation will be on self-developed hardware. If shortage of time occur, some kind of developer card or simulator can be used. Lower compressor quality might be necessary if hardware performance is too low.
  - We implemented it in a simulator since shortage of time occured. We also lowered the compressor quality in theory by ignoring the psychoacoustic analysis step in the encoder.
- The optimum goal was to produce a card with analog and digital (S/PDIF) input and output from the hardware would be an MPEG compressed bitstream.
  - We used a simulator thus no card was produced. This mainly because lack of time.

Other topics not included in the thesis specification:

- Memory requirements
  -The current implementation of our encoder uses 43 KB of program memory, 73 KB of RAM and 10 KB of ROM, this fit in the DSP which have 128KB of internal memory. Input buffers not included. (Approx need data for two frames, which needs 9 KB memory)

7 Discussion

Problems related to the thesis

- When developing DSP applications it's very important to have good tools. Unfortunatly the big DSP manufacturers take high prices for the tools, a fact that influenced our choise of DSP very much.

- The Layer III algorithm is very complex and uses a lot of memory (tables). This is a problem since most of the DSP:s are develped with the idea that the code should fit in the DSP memory so external memory bandwidth can be a limiting factor for big algorithms. This problem has been noticed lately, by Texas Instruments for instance. They have developed a new chip, TMS320C6211, with L1 and L2 cache for external memory, addressing the external memory bandwidth problem.

- Since our code could not fit in the DSP internal memory, and external memory access was slow, we solved the problem by copying sections of external memory to the internal, when the external memory was needed. The memory was then copied back to external memory after it had been used. By doing this we had some trouble with memory fragmentation, but after some changes in the code the problem was solved.

Possible improvements

Next step in this project, would be to let an A/D or digital in write to a circular buffer in the external memory via DMA. The CPU then copies one frame, 1152 samples, from the external memory and processes it, frame by frame.

Improve audio quality
To improve the audio quality there is need for a reimplementation of the psychoacoustic analysis. To include this, much more memory is required. This might be a problem since the DSP has limited internal memory. The bandwidth to external memory could also be a problem.

Use another DSP
Replace the TMS320C6701 with TMS320C6711, when (if) it comes (TMS320C6211 with floating point support). Another interesting DSP is the ADSP-21160 which is code compatible with ADI's popular ADSP-2106x SHARC DSPs, and has 4 Mbit internal memory. Samples of the ADSP-21160 will be available through ADI in the fourth quarter of 1998.

Use another technology
The TM-1000 MPEG chip from Philips, aimed for set-top-box applications, is very interesting and alternative architechtures like this chip is certainly worth taking a look at in future MPEG audio encoder developing. We found out about this chip too late in our project, unfortunately.

Use another algorithm
The AAC algorithm will probably be something to take a closer look at in the future.

8 References

[1] http://www.audioactive.com/products/prod_hwe.html $2450, september 1998.

[2] Audio Engineering Handbook, K. Blair Benson. Benson, McGraw-Hill Book Company,1988 ISBN 0-07-004777-4

[3] "AES Recommended Practice for Professional Digital Audio Applications Employing Pulse-Code Modulation-Preferred Sampling Frequencies", AES5-1984 (ANSI S4.28-1984),J. Audio Eng. Soc., 32(10), 781-785 (October 1984)

[4] "Cochlear Mechanics", S. T. Neely, Boys Town National Research Hospital http://www.boystown.org/cel/cochmech.htm neely@boystown.org

[5] National Bureau of Standards Reports of Calibration, TN-181701, July 8, 1964.

[6] "Människans fysiologi", Haug, Sand, Sjaastad, Universitetsförlaget Liber Utbildning, ISBN 91-634-0052-9

[7] "ATRAC: Adaptive Transform Acoustic Coding for MiniDisc",Sony Corporate Research Laboratories, Reprinted from the 93rd Audio Engineering Society Convention in San Fransisco, 1992 October 1-4 http://www.hip.atr.co.jp/~eaw/minidisc/aes_atrac.html

[8] http://www.cs.sfu.ca/CourseCentral/365/li/material/notes/Chap4/Chap4.4/Chap4.4.html

[9] Wide band speech and audio coding, http://www.umiacs.umd.edu/~desin/Speech1/new.html

[10] ISO/IEC International Standard IS 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"

[11] ISO/IEC International Standard IS 13818-3 "Information Technology - Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbits/s - Part 3: Audio"

[12] D. Y. Pan,"Digital Audio Compression", Digital Technical Journal Vol. 5/ No. 2, Spring 1993

[13] D. Y. Pan,"A Tutorial on MPEG/Audio Compression", Institute of Electrical and Electronics Engineers, 1996

[14] "An Experimental High Fidelity Perceptual Audio Coder", Bosse Lincoln http://www-ccrma.stanford.edu/~bosse/proj/proj.html

[15] IS13818-7, N1650

[16] wg11-N2005, San Jose -98, Revised Report on Complexity of MPEG-2 AAC Tools

[17] ISO/IEC MPEG-2 Advanced Audio Coding. AES Preprint November 1996, 4382(N-1) Bosi et al.

[18] Enhancing the Performance of Perceptual Audio Coders by Using Temporal Noise Shaping (TNS). AES Preprint November 1996, 4384(N-3) Jurgen Herre, James D. Johnston (Tyskt Y i Jurgen)

[19] ISO/IEC International Standard IS 13818-3 "Information Technology - Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbits/s - Part 3: Audio"

[20] ATRAC: Adaptive Transform Acoustic Coding for MiniDisc, Tsutsui, Suzukiet al. Sony Corporate Research Laboratories, Reprinted from the 93:rd Audio Engineering Society Convention in San Fransisco, 1992 October 1-4, http://www.amulation.co.uk/minidisc/minidisc/aes_atrac.html

[21] Low-Bitrate Audio Coding: A Comparison, Thomas Gleerup, Lecture Course C5126 Audio Engineering, 1997, http://www.it.dtu.dk/~tmg/c5126/c5126_paper.htm

[22] AC3: Flexible Perceptual Coding for Audio Transmission and Storage
Todd et al, Dolby Laboratories San Fransisco, Paper presented in the 96th AES Convention, Preprint 3796. http://www.dolby.com/tech/ac3flex.html

[23],ADSP-21065L Datasheet, ADSP21065_L_P3.pdf, http://www.adi.com

[24] TMS320C6701 Datasheet, sprs067.pdf, http://www.ti.com

[25] Multimedia Processing on the TriMedia TM-1000, www.trimedia.philips.com

[26] "ISO/IEC MPEG-2 Advanced Audio Coding: Overview and Applications", Brandenburg, Bosi. Preprint presented at 103rd AES convention (Audio Engineering Society) (4641 C-1)

[27] IS13818-5, dist10

[28] What is Fraunhofer IIS?, http://www.iis.fhg.de/index.html

[29] Background on MPEG Audio coding, http://www.sv.philips.com/mpeg/html/mpeg_general.html

[30] http://developer.intel.com/drg/mmx/AppNotes/AP528.HTM

[31] C6x Optimimization Checklist, http://www.ti.com

Glossary

AAC	Advanced Audio Coding
ADPCM	Adaptive Differential Pulse Code Modulation
ATRAC	Adaptive TRansform Acoustic Coding
DAB	Digital Audio Broadcasting, known as digital radio
DPCM	Differential Pulse Code Modulation
DSP	Digital Signal Processor
DVD	Digital Versatile Disk
Filter bank	A set of band-pass filters covering the entire audio frequency Range.
FFT	Fast Fourier Transform. An algorithm for computing the Fourier transform of a given set of discrete data values. The FFT expresses the data in terms of its component frequencies. It also solves the essentially identical inverse problem of reconstructing a signal from the frequency data.
HDTV	High Definition Television
Hi-Fi Audio	High Fidelity Audio
IC	Integrated Ciruit
Joint stereo coding Stereophonic	Any method that exploits stereophonic irrelevance or redundancy.
Layer III	MPEG-1 consist of three layers I, II and III.
LFE	Low Frequency Element channel
MDCT	A transform which has the property of time domain aliasing cancellation.
MP3	popular name of the MPEG-1 Layer III standard
MPEG	Motion Picture Experts Group
Multilingual	A presentation of dialogue in more than one language.
NBC	Non Backward Compatibility, old name for AAC
PCM	Pulse Code Modulation
PQF	Polyphase Quadrature Filter
Psychoacoustic model	A mathematical model of the masking behaviour of the human auditory system.
RAM	Read And Write Memory
ROM	Read Only Memory
Side information	Information in the bitstream necessary for controlling the decoder
SMR	Signal to mask ratio
S/PDIF	Sony/Phillips Digital InterFace
SSR	Scalable Sampling Rate
TCP/IP	TCP/IP Internet Protocol Suite
TNS	Temporal Noise Shaping
Twin VQ	Transform-domain Weighted Interleave Vector Quantization

Appendix A

The original MDCT code:

```
void mdct( double *in, double *out, int block_type )
{
/*------------------------------------------------------------------*/
/*                                                                  */
/*   Function: Calculation of the MDCT                              */
/*   In the case of long blocks ( block_type 0,1,3 ) there are      */
/*   36 coefficents in the time domain and 18 in the frequency      */
/*   domain.                                                        */
/*   In the case of short blocks (block_type 2 ) there are 3        */
/*   transformations with short length. This leads to 12 coefficents */
/*   in the time and 6 in the frequency domain. In this case the    */
/*   results are stored side by side in the vector out[].           */
/*                                                                  */
/*   New layer3                                                     */
/*                                                                  */
/*------------------------------------------------------------------*/

  int l,k,i,m,N;
  double sum;
  static double win[4][36];
  static double cos_s[6][12], cos_l[18][36];

  if ( block_type == 2 )
  {
    N = 12;
    for ( l = 0; l < 3; l++ )
    {
      for ( m = 0; m < N / 2; m++ )
      {
        for ( sum = 0.0, k = 0; k < N; k++ )
          sum += win[block_type][k] * in[k + 6 * l + 6] * cos_s[m][k];
        out[ 3 * m + l] = sum;
      }
    }
  }
  else
  {
    N = 36;
    for ( m = 0; m < N / 2; m++ )
    {
      for ( sum = 0.0, k = 0; k < N; k++ )
        sum += win[block_type][k] * in[k] * cos_l[m][k];
      out[m] = sum;
    }
  }
}
```

modified version of Lee´s Fast DCT algorithm:

```
static void mdct( float *in, float *out )
{
    int i;
    float sum;
    float c[18];

   /* do the f(i)*in(i) first. and save the results */

      for(i=0;i<9;i++){
        c[i]= in[i]*win[i]-in[17-i]*win[17-i];
        c[i+9]= in[i+18]*win[i+18]+in[35-i]*win[35-i];
      }

      /* 0 */
      sum =  c[0]  * cos_l[0][0];
      sum += c[1]  * cos_l[0][1];
      sum += c[2]  * cos_l[0][2];
      sum += c[3]  * cos_l[0][3];
      sum += c[4]  * cos_l[0][4];
      sum += c[5]  * cos_l[0][5];
      sum += c[6]  * cos_l[0][6];
      sum += c[7]  * cos_l[0][7];
      sum += c[8]  * cos_l[0][8];
      sum += c[9]  * cos_l[0][9];
      sum += c[10] * cos_l[0][10];
      sum += c[11] * cos_l[0][11];
      sum += c[12] * cos_l[0][12];
      sum += c[13] * cos_l[0][13];
      sum += c[14] * cos_l[0][14];
      sum += c[15] * cos_l[0][15];
      sum += c[16] * cos_l[0][16];
      sum += c[17] * cos_l[0][17];
      out[0]=sum;

      /* 1 */
      sum  = (c[2]+c[3]+c[17]) * cos_l[1][2];
      sum += (c[1]+c[4]+c[16]) * cos_l[1][1];
      sum += (c[0]+c[5]+c[15]) * cos_l[1][0];
      sum += (c[6]-c[9]+c[14]) * cos_l[1][6];
      sum += (c[7]-c[10]+c[13]) * cos_l[1][7];
      sum += (c[8]-c[11]+c[12]) * cos_l[1][8];
      out[1]=sum;

      /* 2 */
      sum =  c[0] * cos_l[2][0];
      sum += c[1] * cos_l[2][1];
      sum += c[2] * cos_l[2][2];
      sum += c[3] * cos_l[2][3];
      sum += c[4] * cos_l[2][4];
      sum += c[5] * cos_l[2][5];
      sum += c[6] * cos_l[2][6];
      sum += c[7] * cos_l[2][7];
      sum += c[8] * cos_l[2][8];
      sum += c[9] * cos_l[2][9];
      sum += c[10] * cos_l[2][10];
      sum += c[11] * cos_l[2][11];
      sum += c[12] * cos_l[2][12];
      sum += c[13] * cos_l[2][13];
      sum += c[14] * cos_l[2][14];
      sum += c[15] * cos_l[2][15];
      sum += c[16] * cos_l[2][16];
      sum += c[17] * cos_l[2][17];
      out[2]=sum;
```

```
/* 3 */
sum =  c[0] * cos_l[3][0];
sum += c[1] * cos_l[3][1];
sum += c[2] * cos_l[3][2];
sum += c[3] * cos_l[3][3];
sum += c[4] * cos_l[3][4];
sum += c[5] * cos_l[3][5];
sum += c[6] * cos_l[3][6];
sum += c[7] * cos_l[3][7];
sum += c[8] * cos_l[3][8];
sum += c[9] * cos_l[3][9];
sum += c[10] * cos_l[3][10];
sum += c[11] * cos_l[3][11];
sum += c[12] * cos_l[3][12];
sum += c[13] * cos_l[3][13];
sum += c[14] * cos_l[3][14];
sum += c[15] * cos_l[3][15];
sum += c[16] * cos_l[3][16];
sum += c[17] * cos_l[3][17];
out[3]=sum;

/* 4 */
sum =   (c[0]-c[1]-c[4]+c[5]+c[8]-c[11]+c[12]+c[15]-c[16]) * cos_l[4][0];
sum += (-c[2]-c[3]+c[6]+c[7]-c[9]-c[10]+c[13]+c[14]-c[17]) * cos_l[4][6];
out[4]=sum;

/* 5 */
sum =  c[0] * cos_l[5][0];
sum += c[1] * cos_l[5][1];
sum += c[2] * cos_l[5][2];
sum += c[3] * cos_l[5][3];
sum += c[4] * cos_l[5][4];
sum += c[5] * cos_l[5][5];
sum += c[6] * cos_l[5][6];
sum += c[7] * cos_l[5][7];
sum += c[8] * cos_l[5][8];
sum += c[9] * cos_l[5][9];
sum += c[10] * cos_l[5][10];
sum += c[11] * cos_l[5][11];
sum += c[12] * cos_l[5][12];
sum += c[13] * cos_l[5][13];
sum += c[14] * cos_l[5][14];
sum += c[15] * cos_l[5][15];
sum += c[16] * cos_l[5][16];
sum += c[17] * cos_l[5][17];
out[5]=sum;

/* 6 */
sum =  c[0] * cos_l[6][0];
sum += c[1] * cos_l[6][1];
sum += c[2] * cos_l[6][2];
sum += c[3] * cos_l[6][3];
sum += c[4] * cos_l[6][4];
sum += c[5] * cos_l[6][5];
sum += c[6] * cos_l[6][6];
sum += c[7] * cos_l[6][7];
sum += c[8] * cos_l[6][8];
sum += c[9] * cos_l[6][9];
sum += c[10] * cos_l[6][10];
sum += c[11] * cos_l[6][11];
sum += c[12] * cos_l[6][12];
sum += c[13] * cos_l[6][13];
sum += c[14] * cos_l[6][14];
sum += c[15] * cos_l[6][15];
sum += c[16] * cos_l[6][16];
sum += c[17] * cos_l[6][17];
out[6]=sum;
```

```
/* 7 */
sum =  (c[0]+c[5]+c[15]) * cos_l[7][0];
sum += (c[1]+c[4]+c[16]) * cos_l[7][1];
sum += (c[2]+c[3]+c[17]) * cos_l[7][2];
sum += (c[6]-c[9]+c[14]) * cos_l[7][6];
sum += (c[7]-c[10]+c[13]) * cos_l[7][7];
sum += (c[8]-c[11]+c[12]) * cos_l[7][8];
out[7]=sum;

/* 8 */
sum =  c[0] * cos_l[8][0];
sum += c[1] * cos_l[8][1];
sum += c[2] * cos_l[8][2];
sum += c[3] * cos_l[8][3];
sum += c[4] * cos_l[8][4];
sum += c[5] * cos_l[8][5];
sum += c[6] * cos_l[8][6];
sum += c[7] * cos_l[8][7];
sum += c[8] * cos_l[8][8];
sum += c[9] * cos_l[8][9];
sum += c[10] * cos_l[8][10];
sum += c[11] * cos_l[8][11];
sum += c[12] * cos_l[8][12];
sum += c[13] * cos_l[8][13];
sum += c[14] * cos_l[8][14];
sum += c[15] * cos_l[8][15];
sum += c[16] * cos_l[8][16];
sum += c[17] * cos_l[8][17];
out[8]=sum;

/* 9 */
sum =  c[0] * cos_l[9][0];
sum += c[1] * cos_l[9][1];
sum += c[2] * cos_l[9][2];
sum += c[3] * cos_l[9][3];
sum += c[4] * cos_l[9][4];
sum += c[5] * cos_l[9][5];
sum += c[6] * cos_l[9][6];
sum += c[7] * cos_l[9][7];
sum += c[8] * cos_l[9][8];
sum += c[9] * cos_l[9][9];
sum += c[10] * cos_l[9][10];
sum += c[11] * cos_l[9][11];
sum += c[12] * cos_l[9][12];
sum += c[13] * cos_l[9][13];
sum += c[14] * cos_l[9][14];
sum += c[15] * cos_l[9][15];
sum += c[16] * cos_l[9][16];
sum += c[17] * cos_l[9][17];
out[9]=sum;

/* 10 */
sum =  (c[0]+c[5]+c[15]) * cos_l[10][0];
sum += (c[1]+c[4]+c[16]) * cos_l[10][1];
sum += (c[2]+c[3]+c[17]) * cos_l[10][2];
sum += (c[6]-c[9]+c[14]) * cos_l[10][6];
sum += (c[7]-c[10]+c[13]) * cos_l[10][7];
sum += (c[8]-c[11]+c[12]) * cos_l[10][8];
out[10]=sum;

/* 11 */
sum =  c[0] * cos_l[11][0];
sum += c[1] * cos_l[11][1];
sum += c[2] * cos_l[11][2];
sum += c[3] * cos_l[11][3];
sum += c[4] * cos_l[11][4];
sum += c[5] * cos_l[11][5];
sum += c[6] * cos_l[11][6];
```

```
sum += c[7] * cos_l[11][7];
sum += c[8] * cos_l[11][8];
sum += c[9] * cos_l[11][9];
sum += c[10] * cos_l[11][10];
sum += c[11] * cos_l[11][11];
sum += c[12] * cos_l[11][12];
sum += c[13] * cos_l[11][13];
sum += c[14] * cos_l[11][14];
sum += c[15] * cos_l[11][15];
sum += c[16] * cos_l[11][16];
sum += c[17] * cos_l[11][17];
out[11]=sum;

/* 12 */
sum =  c[0] * cos_l[12][0];
sum += c[1] * cos_l[12][1];
sum += c[2] * cos_l[12][2];
sum += c[3] * cos_l[12][3];
sum += c[4] * cos_l[12][4];
sum += c[5] * cos_l[12][5];
sum += c[6] * cos_l[12][6];
sum += c[7] * cos_l[12][7];
sum += c[8] * cos_l[12][8];
sum += c[9] * cos_l[12][9];
sum += c[10] * cos_l[12][10];
sum += c[11] * cos_l[12][11];
sum += c[12] * cos_l[12][12];
sum += c[13] * cos_l[12][13];
sum += c[14] * cos_l[12][14];
sum += c[15] * cos_l[12][15];
sum += c[16] * cos_l[12][16];
sum += c[17] * cos_l[12][17];
out[12]=sum;

/* 13 */
sum =  (c[0]-c[1]-c[4]+c[5]+c[8]-c[11]+c[12]+c[15]-c[16]) * cos_l[13][0];
sum += (c[2]+c[3]-c[6]-c[7]+c[9]+c[10]-c[13]-c[14]+c[17]) * cos_l[13][2];
out[13]=sum;

/* 14 */
sum =  c[0] * cos_l[14][0];
sum += c[1] * cos_l[14][1];
sum += c[2] * cos_l[14][2];
sum += c[3] * cos_l[14][3];
sum += c[4] * cos_l[14][4];
sum += c[5] * cos_l[14][5];
sum += c[6] * cos_l[14][6];
sum += c[7] * cos_l[14][7];
sum += c[8] * cos_l[14][8];
sum += c[9] * cos_l[14][9];
sum += c[10] * cos_l[14][10];
sum += c[11] * cos_l[14][11];
sum += c[12] * cos_l[14][12];
sum += c[13] * cos_l[14][13];
sum += c[14] * cos_l[14][14];
sum += c[15] * cos_l[14][15];
sum += c[16] * cos_l[14][16];
sum += c[17] * cos_l[14][17];
out[14]=sum;

/* 15 */
sum =  c[0] * cos_l[15][0];
sum += c[1] * cos_l[15][1];
sum += c[2] * cos_l[15][2];
sum += c[3] * cos_l[15][3];
sum += c[4] * cos_l[15][4];
sum += c[5] * cos_l[15][5];
sum += c[6] * cos_l[15][6];
```

```
        sum += c[7] * cos_l[15][7];
        sum += c[8] * cos_l[15][8];
        sum += c[9] * cos_l[15][9];
        sum += c[10] * cos_l[15][10];
        sum += c[11] * cos_l[15][11];
        sum += c[12] * cos_l[15][12];
        sum += c[13] * cos_l[15][13];
        sum += c[14] * cos_l[15][14];
        sum += c[15] * cos_l[15][15];
        sum += c[16] * cos_l[15][16];
        sum += c[17] * cos_l[15][17];
        out[15]=sum;

        /* 16 */
        sum =  (c[0]+c[5]+c[15]) * cos_l[16][0];
        sum += (c[1]+c[4]+c[16]) * cos_l[16][1];
        sum += (c[2]+c[3]+c[17]) * cos_l[16][2];
        sum += (c[6]-c[9]+c[14]) * cos_l[16][6];
        sum += (c[7]-c[10]+c[13]) * cos_l[16][7];
        sum += (c[8]-c[11]+c[12]) * cos_l[16][8];
        out[16]=sum;

        /* 17 */
        sum =  c[0] * cos_l[17][0];
        sum += c[1] * cos_l[17][1];
        sum += c[2] * cos_l[17][2];
        sum += c[3] * cos_l[17][3];
        sum += c[4] * cos_l[17][4];
        sum += c[5] * cos_l[17][5];
        sum += c[6] * cos_l[17][6];
        sum += c[7] * cos_l[17][7];
        sum += c[8] * cos_l[17][8];
        sum += c[9] * cos_l[17][9];
        sum += c[10] * cos_l[17][10];
        sum += c[11] * cos_l[17][11];
        sum += c[12] * cos_l[17][12];
        sum += c[13] * cos_l[17][13];
        sum += c[14] * cos_l[17][14];
        sum += c[15] * cos_l[17][15];
        sum += c[16] * cos_l[17][16];
        sum += c[17] * cos_l[17][17];
        out[17]=sum;
}
```

Appendix B

```
void quantize( DOUBLE xr[576], short ix[576], gr_info *cod_info )
{
short i;
DOUBLE step;
DOUBLE ostep;
float temp;

if ( cod_info->quantizerStepSize == 0 )
          step = 1.0;
else
   step = pow ( 2.0, cod_info->quantizerStepSize * 0.25 );

ostep = 1.0/step;
for ( i = 0; i < 576; i++ ){
  temp=ostep*fabs(xr[i]);
  if (temp<4.999960e-01)
    ix[i]=0;
  else {if(temp<1.862955e+00) ix[i]=1;
      else if(temp<3.565282e+00) ix[i]=2;
      else if(temp<5.506396e+00) ix[i]=3;
      else if(temp<7.638304e+00) ix[i]=4;
      else if(temp<9.931741e+00) ix[i]=5;
    else
      ix[i] = nint( pow(fabs(xr[i]) / step, 0.75) - 0.0946);
  else
    ix[i] = pow_nint(fabs(xr[i]) * ostep);
  }
}
}
```

* The vector ix[] holds our precalculated values.