LUNDS TEKNISKA
HÖGSKOLA
Lunds universitet

Ogg/Vorbis in embedded systems

Master thesis

Erik Montnémery
Johannes Sandvall

February 2004

Abstract

Ogg/Vorbis is currently a growing audio format, mainly used for online distribution of music. The number of available encoded audio files is quickly increasing even though MP3 still is the most used format. Many internet radio stations have begun streaming in Ogg/Vorbis and even more are examining the possibilities. In contrast with other renown formats such as AAC and MP3, Ogg/Vorbis is totally license and royalty free. For embedded platforms the licensing and royalty cost for supporting commercial formats can be quite taxing as payments are often per device.

The aim of this thesis is to implement an embedded Ogg/Vorbis system under strict memory and CPU usage constraints. As opposed to most other audio formats, Ogg/Vorbis includes codebooks and other data structures in the data stream, thus greatly increasing dynamic memory usage. Furthermore, the reference decoder is based on floating point math, albeit a fixed-point implementation also exists. These problems paired with the short time elapsed since Ogg/Vorbis was introduced has had the implications that very few embedded implementations have been done so far.

Since a reference fixed-point math decoder exists under a generous BSD-like license the efforts have been concentrated at optimizations of memory and CPU usage of the available code. Data structures and lookup tables have been optimized and the largest CPU consuming block, the IMDCT, has been replaced with a mathematically equivalent FFT that is more than eight times faster.

Contents

Contents 5

1 Introduction 7
- 1.1 Motivation . . . 7
- 1.2 Objectives . . . 8
- 1.3 Outline . . . 8
- 1.4 Acknowledgments . . . 8

2 Digital audio coding 9
- 2.1 Lossy coding . . . 9
- 2.2 Psycho acoustic audio coding . . . 10
  - 2.2.1 Critical bandwidth . . . 11
  - 2.2.2 Masking . . . 12
  - 2.2.3 MDCT and window functions . . . 14
  - 2.2.4 Temporal artifacts . . . 15
  - 2.2.5 Wavelets . . . 16
  - 2.2.6 Vector Quantization (VQ) . . . 16
  - 2.2.7 Decoding . . . 16

3 Ogg 19
- 3.1 Encoding . . . 20
- 3.2 Decoding . . . 22
- 3.3 Tremor . . . 27
- 3.4 Future . . . 28
- 3.5 Comparison with other audio formats . . . 28
- 3.6 License . . . 29
- 3.7 Ogg/Vorbis in embedded systems . . . 29

4 Implementation and Partitioning 31
- 4.1 Dynamic memory handling . . . 31
- 4.2 Host – DSP . . . 32
- 4.3 Codebook cache . . . 33
- 4.4 Host – DSP packet format . . . 33
- 4.5 Host – DSP communication error handling . . . 34

4.5.1 Reentrance 34
4.6 Word size 34

5 Optimization **37**
5.1 Target hardware 37
5.2 Memory 37
5.2.1 Lookup tables 38
5.2.2 Codebook cache 38
5.2.3 Data structures 39
5.2.4 Reducing code size 39
5.2.5 Further memory reduction 39
5.3 Complexity 40
5.3.1 Initial profiling 40
5.3.2 IMDCT / FFT 40
5.3.3 Assembler / C 43
5.3.4 Profiling 44

6 DSP **47**
6.1 DSP Processors 47
6.2 Platform 55xx 49
6.3 TI – XDAIS 50

7 Results and future improvements **51**
7.1 Audio quality 51
7.2 Memory 52
7.3 CPU usage 54
7.4 Future improvements 54

8 Conclusion **57**

A List of abbreviations **59**

B Tremor license **61**

Bibliography **63**

Chapter 1

Introduction

During the last couple of years lossy compression of audio data has gained widespread acceptance, especially for on-line distribution of music. The most renown compression technique in use today is without comparison MPEG I layer III, or "MP3", which is even used as a synonym for compressed music data. Even though MP3 has gained wide spread acceptance, it suffers from various flaws and limitations. One notable limitation is that MP3 does not support more than two audio channels.

Several new codecs have been developed to replace MP3, featuring both better compression using more modern techniques (e.g. more refined psycho acoustic model, temporal noise shaping, etc.) and less restrictions on the audio streams being processed. These codecs are often referred to as "second generation audio codecs". In widespread use are AAC, which is used to compress the audio data in DVD, AC3 from Dolby Laboratories used in HDTV and WMA which is a codec developed by Microsoft.

The development of Ogg/Vorbis started as a reaction to the aggressive license policy run by the Fraunhofer institute, owner of the MP3 standard. Ogg/Vorbis is a second generation codec technically on par with new standards such as AAC and WMA, while remaining totally free. As the name suggests Ogg/Vorbis is a combination of two formats, namely Ogg, a general purpose container stream format and Vorbis, a psycho acoustic audio codec.

1.1 Motivation

Ogg/Vorbis is currently gaining widespread acceptance as a free audio format and several internet radio stations have begun streaming using Ogg/Vorbis. However most products and implementations are focused on PC based decoding and very few embedded Ogg/Vorbis implementations have been introduced on the market so far.

The goal of this project is to explore the possibilities of implementing a working high quality Ogg/Vorbis decoder in an embedded environment with strict memory constraints. Two decoder versions have been implemented and will be presented. In the first all decoding and stream parsing is done on a stand alone DSP target while

in the second version the decoding is split up between parsing and pre-decoding on a general purpose CPU (e.g. an ARM) and decoding on a DSP. Implementing a split decoder is motivated by the fact that many embedded systems feature a stripped DSP core intended for signal processing tasks combined with a RISC-based host CPU.

1.2 Objectives

- Explore the possibilities of running an Ogg/Vorbis decoder implementation under strict memory requirements in an embedded environment.
- Implement a stand-alone DSP optimized decoder.
- Implement a multi processed decoder where stream parsing is done externally, and only the actual decoding computations are optimized for DSP execution.
- Investigate possible optimizations in terms of memory and CPU-usage.

1.3 Outline

The report starts with a brief introduction to psycho acoustic models focused on the methods used by Ogg/Vorbis. The next chapter discusses the Ogg/Vorbis format and the basic design of the decoder. The implementation follows the design of the format and only design decisions that optimize the design or introduce aspects that are not covered in the format description are further discussed. The implementation and partitioning chapter describe design considerations, memory handling and partitioning of the decoder into parsing and decoding on separate processes. The next chapter, Optimization, describes the optimization strategies used in the project, followed by a brief chapter on DSP processors in general. The last two chapters present the results obtained, what to expect from future work and finally conclude the project achievements.

1.4 Acknowledgments

The authors would like to thank:

Thomas Lenart and Viktor Öwall at the department of Electroscience at Lund Institute of Technology.

Peter Eneroth and Per Nilsson at Ericsson Mobile Platforms, Lund

Texas Instruments for their donation of the DSP development board.

The Xiph.org Foundation

Chapter 2

Digital audio coding

The transmission of uncompressed 16-bit PCM stereo audio sampled at 44.1 kHz over the internet would require over 1.4 Mbps and over 50 Mbyte in storage area for a 5 minute data set. This motivates the need for audio compression in order to reduce both the bandwidth and storage requirements. Available compression techniques can be categorized as either lossless or lossy coding. A lossless coder is able to perfectly reconstruct the samples of the original signal. Examples of lossless codecs are FLAC[1] and SHN[2] (shorten). Most lossy codecs in use today combine lossy coding techniques with the methods used for lossless coding in order to achieve a better compression rate while at the same time maintaining acceptable audio quality.

This chapter puts focus on the perceptual coding methods that are used in Ogg/Vorbis, for a more complete overview of available methods see [1].

2.1 Lossy coding

Lossy audio data compression methods are methods where high compression rates are achieved by the removal of perceptual irrelevancies and statistical redundancies in the original data. It is often referred to as transparent coding when the reconstructed signal is perceptually impossible to distinguish from the original. Compared to lossless coding of audio where a normal data rate is about 10 bits/sample, lossy coding can achieve transparency at bit rates of less than 1 bit/sample.

There are many techniques available for lossy coding, some of which need very small computational efforts to compress data while the more refined coders need a substantial amount of digital signal processing. At the low end of the scale are trivial methods such as simply decreasing the number of bits used to represent each sample, used e.g. when transmitting telephony at a sample rate of 8kHz with 8 bits/sample (compared to the sample rate of 44.1kHz with 16 bits/sample used for CD-audio). Such techniques can obviously not maintain transparent coding. At the high end

[1] FLAC – http://flac.sourceforge.net
[2] SHN (shorten) – http://www.softsound.com/Shorten.html

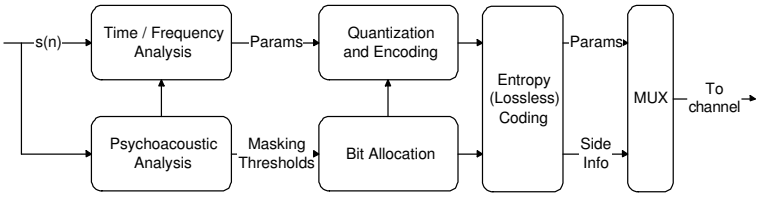


Figure 2.1: Generic perceptual audio encoder

of the scale are coders that use a psycho acoustic model to remove redundant audio information while maintaining transparency.

2.2 Psycho acoustic audio coding

Psycho acoustic coding is a set of lossy compression methods that intend to remove sound that is inaudible to the human ear, based on a generalized models of the human auditory system. Most people have a hearing range from 20Hz to 20kHz and a frequency resolution in the middle range of about 2Hz. The sensitivity to sounds differs from a frequency to another and the human ear is typically most sensitive at about 3kHz, see figure 2.2. Sounds that are too low in amplitude to be percepted when considering the spectral content can thus simply be removed. The presence of a faint but normally audible sound that is either very close in frequency or with a much lesser amplitude than surrounding, stronger, sounds may also render the faint sound inaudible, this effect is called masking.

The psycho acoustic analysis is generally done in the frequency domain after transformation from the time domain by the use of an FFT. The inaudible parts of the spectrum are removed before quantization and transmission of the spectral data, thus greatly reducing the needed data rate. A model of a generic perceptual encoder is shown in figure 2.1. First, the input data is sent both to the time/frequency analysis and the psychoacoustic analyzer. The time/frequency analysis is adjusted dynamically according to the results of the psychoacoustic analysis. The output of these steps is then sent to quantization and bit allocation, according to the figure. Quantization and encoding bit allocation decisions are based on the results of the psycho acoustic analysis. The remaining data is then entropy coded and multiplexed into a single bit stream.

Sound pressure level (SPL) is the standard metric for measuring the sound level or intensity in decibels (dB). It is defined as $L_{SPL} = 20log_{10}(p/p_0)dB$, where p is the sound pressure in Pascals (Pa) and p_0 is the standard reference level of 20 μPa. 150 db SPL spans the human range of hearing, even though 140 dB SPL is above the pain threshold for most people.

The absolute threshold of hearing (ATH) is the sound pressure level that is needed for a pure tone to be audible in a noiseless environment. The threshold in dB SPL can

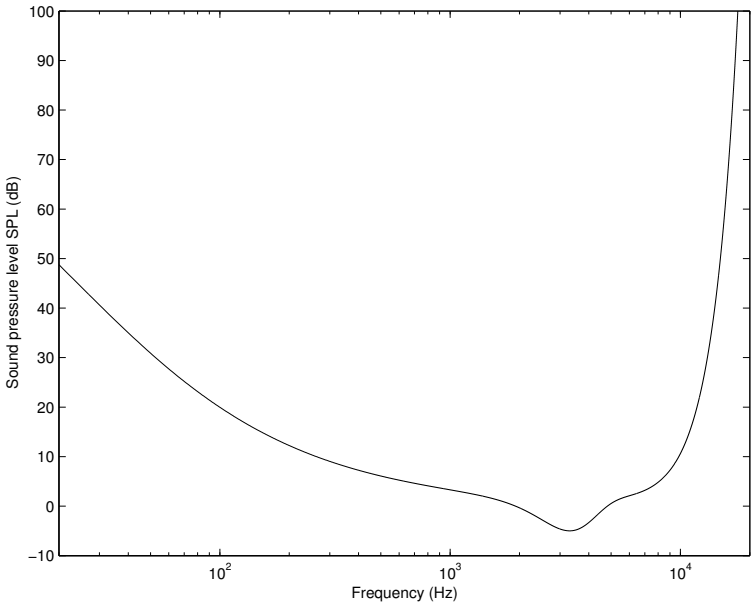


Figure 2.2: Absolute threshold characteristics of the human auditory system

be approximated using the following equation:

$$T_q(f) = 3.64(f/1000)^{-0.8} - 6.5e^{-0.6(f/1000-3.3)^2} + 10^{-3}(f/1000)^4 \quad (2.1)$$

which is plotted in Figure 2.2.

A few other important terms needs to be defined before proceeding: signal to noise ratio (SNR) which denotes the signal strength related to background noise and is usually measured in decibels (dB), and signal to mask ratio, SMR, which is the strength of a signal related to the masking threshold, also in dB.

2.2.1 Critical bandwidth

A definition of critical bands is "The bandwidth at which subjective responses change rather abruptly" [2]. In other words it means the ear's ability to identify and separate sounds of different frequencies. The established unit of critical band rate (CBR) is 'Bark' and corresponds to the physical structure of the ear. The Bark scale is a linearization of the human ear's frequency response which has near linear properties for low frequencies and more logarithmic for higher frequencies [2]. The linear properties makes CBR useful for psycho acoustic models. The conversion from linear frequency to CBR is given by

$$z(f) = 13\arctan(0.00076f) + 3.5\arctan[(\frac{f}{7500})^2] \quad (2.2)$$

where z is in Barks and f is the frequency in Hz. Experiments have shown that the 25 bands enumerated in table 2.1 corresponds closely to the human auditory system.

Bark	Center (Hz)	BW (Hz)	Bark	Center (Hz)	BW (Hz)
1	50	0-100	14	2150	2000-2320
2	150	100-200	15	2500	2320-2700
3	250	200-300	16	2900	2700-3150
4	350	300-400	17	3400	3150-3700
5	450	400-510	18	4000	3700-4400
6	570	510-630	19	4800	4400-5300
7	700	630-770	20	5800	5300-6400
8	840	770-920	21	7000	6400-7700
9	1000	920-1080	22	8500	7700-9500
10	1175	1080-1270	23	10500	9500-12000
11	1370	1270-1480	24	13500	12000-15500
12	1600	1480-1720	25	19500	15500-
13	1850	1720-2000			

Table 2.1: Band(Bark) and bandwidth for the 25 critical bands

Some codecs use a different frequency decomposition though. MPEG-1 layer I,II and III, for instance divides the frequency spectrum into 32 subbands instead of 25. The reason for not using the 25 bands given in table 2.1 is of computational nature, i.e. faster calculations.

2.2.2 Masking

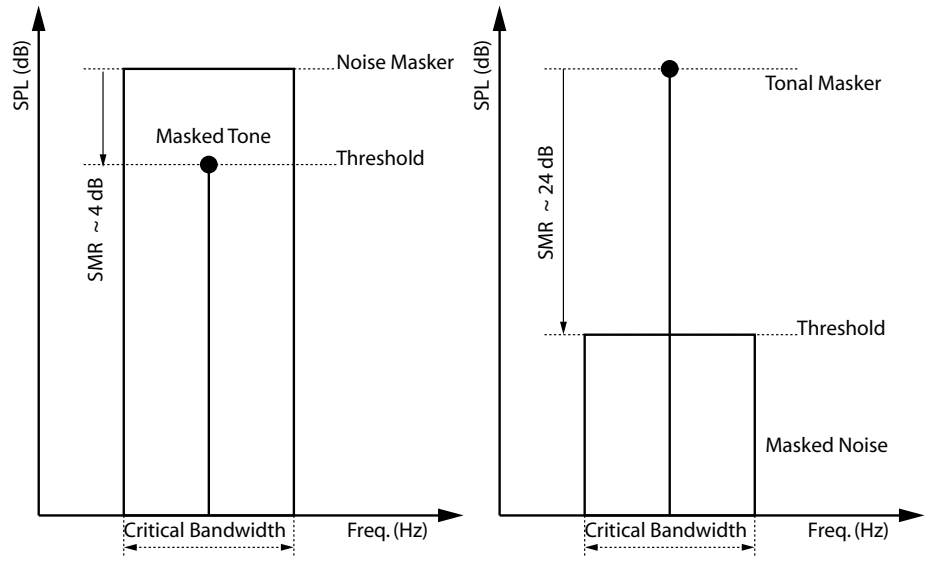
Masking refers to one sound made inaudible in the presence of another stronger sound, either from a pure spectral viewpoint or combined with time aspects. Both frequency and phase relationships affect masking. Even though complex masking scenarios may occur, for the purpose of psycho acoustic modeling it is convenient to classify masking effects as one of four typical categories.

1. Noise Masking Tone (NMT)

A narrow-band noise will mask a tone of lower amplitude provided that it is within the same critical band, see figure 2.3a. A predictable masking threshold is determined with regard to the noise intensity and the tone can be removed if its amplitude is lower than the threshold. NMT has been known since the 1930's and several studies have been performed to characterize the effect [3].

2. Tone Masking Noise (TMN)

In the opposite scenario of noise masking tone, a pure tone masks noise of less amplitude in its spectral vicinity, see figure 2.3b. Not as many studies have characterized TMN as NMT, but it is generally accepted that the signal to mask ratio is much greater than in the case of NMT (e.g. in [4]). TMN has its strongest effect when the center frequencies of masker and maskee are equal (as is the case with NMT).



(a) Noise masking tone

(b) Tone masking noise

Figure 2.3: The two most pronounced non-temporal masking effects

3. Noise Masking Noise (NMN)

In the noise masking noise case a narrow-band noise masks another narrow-band noise. This scenario is harder to characterize than noise masking tone or tone masking noise because of phase relationships between masker and maskee. One study attempting to characterize NMN for wide-band noises suggests SMRs of nearly 26 dB [5].

4. Temporal masking

Apart from a pure frequency view, time must also be considered since masking effects can be observed after the actual disappearance of the masker and more curious, even before the onset of the masker. An example of this phenomenon can be seen in figure 2.4. Pre masking effects lasts only a few milliseconds while post masking effects are significant for as long as 200 milliseconds after the masker has disappeared.

Asymmetry, spread and modeling

An adequate psycho acoustic model is the key to maintaining perceptual transparency in highly bit reduced codecs. The model must take into account and blend the effect of all masking effects presented. The basic idea is to compute the maximum allowed quantization noise in each critical band or subband while preserving a fixed bit rate or maintaining a subjective audio quality. The former approach is the most

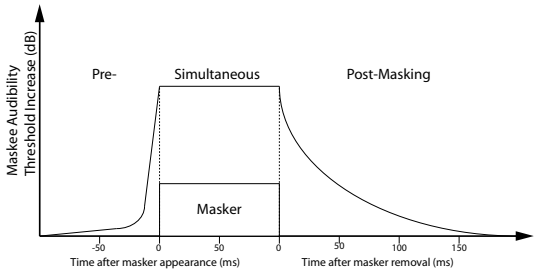


Figure 2.4: Temporal masking properties of the human ear.

(a) Original image

(b) 20x compression using the DCT

(c) 20x compression using the DFT (FFT)

Figure 2.5: Example image to show block artifacts of the DCT and DFT

common but the latter is also used, e.g. in Ogg/Vorbis (it is unclear whether the available Ogg/Vorbis encoder actually does this or if it rather makes use of a non strict form of VBR). More information on psycho acoustic models used in commercial and experimental codecs can be found in e.g [6], [7] and [8].

2.2.3 MDCT and window functions

To facilitate the audio data quantization according to the results of the psycho acoustic analysis, the audio data must be transformed to the frequency domain. Although the FFT is adequate during the psycho acoustic analysis it can not be used to transform blocks of data intended for later reconstruction due to pronounced block artifacts. An informative example of block artifacts when transforming image data can be seen in figure 2.5. Although an audio example would have been more appropriate, it is hard to describe audio artifacts in words. For this reason a transform with less pronounced block artifacts combined with overlapping and windowing (e.g. the MDCT with a sine window) is used.

Most coders use an MDCT based on DCT^{IV} with 50% lapping. Various window shapes have been used of which a few will be described briefly. The basic criterion

on any window shape is that it must follow the Princen-Bradley condition, $w_k^2 + w_{k+N}^2 = 1$, i.e. the square sum of the overlapping window parts must be equal to one. It is worth noting that the window functions differ from other types used in signal analysis. The reason for this is that the windows are applied twice both during encoding (MDCT) and decoding (IMDCT). The target when choosing the window shape is to achieve good passband selectivity while keeping ringing effects at a minimum. MP3 uses a sine window, equation (2.3), which is also used by a number of other coders.

$$sine\ window:\ w_k = \sin((\frac{\pi}{2N}(k+0.5))) \tag{2.3}$$

The advantages of this window function is that the dc energy is concentrated to a single coefficient and that the window is asymptotically optimal in terms of coding gain (minimizing the mean square reconstruction error) [9]. However recent decoders often use a slightly different window function. AAC and AC3 use a Kaiser-Bessel derived (KBD) window. Vorbis uses the window defined in equation (2.4).

$$Vorbis\ window:\ w_k = \sin(\frac{\pi}{2}\sin^2(\frac{\pi}{2N}(k+0.5))) \tag{2.4}$$

The advantages of this window is better stopband attenuation at the expense of lesser passband selectivity.

2.2.4 Temporal artifacts

When using MDCT or related filterbanks there will always be a tradeoff between high spectral resolution and temporal artifacts, which typically appear as a "smearing" of the sound. This effect is very pronounced when coding recordings of e.g. the castanets and cymbals. High spectral resolution is achieved by using a longer window size, e.g. 2048 samples, at the cost of temporal integrity. Temporal integrity on the other hand is achieved in the opposite way, using a shorter window size, e.g. 256 samples.

Several methods intended to reduce temporal artifacts have been proposed, most notably adaptation of the transform size over time and allowing a variable bit rate (VBR). Adaptation of transform size, window switching, works by characterizing signal blocks as either transient or stationary, choosing a shorter window for transients and a longer window for stationary segments. The main drawback of window switching is that unwanted artifacts appear due to transition windows (i.e. when switching from a short to a long window or vice versa).

If the bit rate is allowed to vary over time, even at a "fixed" bit rate, more bits can be spent to allow better reconstruction of transient blocks. A drawback of this scheme is that perceptual quality cannot be guaranteed if the signal contains long blocks of transients. Most standard codecs (e.g. MPEG 1, AAC, AC3, Ogg/Vorbis) use both techniques.

The difference between true variable bit rate (used by Ogg/Vorbis) and "pseudo" variable rate is that in the true variable bit rate scenario the encoder can allocate bits for a certain part of the bit stream in an arbitrary fashion, while in the "pseudo" VBR

scenario the allocation must be kept within some constraints. As an example the limited size bit reservoir used in MP3 results in a fixed upper limit of the deviation from the nominal bit rate.

2.2.5 Wavelets

Discrete Wavelet Transforms (DWT) have very desirable properties for high quality audio coding of transient sounds, e.g. the castanets and cymbals. DWTs are not limited to stationary sinusoids used by the FFT/MDCT classes, as discrete wavelet based transforms use a class of base functions that are well localized in both the time and frequency domains. Furthermore, a DWT can use a non balanced filterbank tree for decomposition. Several attempts at using DWT-based audio coders have shown good results at overcoming the main shortcomings of DCT-based coders such as pre-echo artifacts and unsatisfactory representation of transient sounds, but have unfortunately been found to introduce problems with stationary signals (e.g. in [8]). There have also been interesting attempts at using both DWT and sinusoidal transforms in hybrid coders. The results are very promising, with nearly transparent coding of CD-quality audio at bit rates as low as 44kbit/s, i.e. approximately 1bit/sample [10].

2.2.6 Vector Quantization (VQ)

Vector quantization is a lossy data compression method based on approximation. An obvious way to compress data is to simply decrease the data bit width. The trivial approach to this method is *scalar quantization* which basically means that only the most significant bits are preserved. A more refined approach useful for coding highly correlated data (e.g. audio or video) is to map values into small groups coding adjacent values. A typical example is a colormap, used when converting an RGB image to an image with a smaller subset of colors (e.g. 256). A scalar method uses a fixed transformation based only on the most significant bits of the RGB value, whereas VQ allocates colors based on the image information. E.g. a picture of a forest tends to have most pixels in shades of green and brown and by allocating a larger subset of the map to those colors the overall quantization error of the image can be reduced.

This method can be applied to any large set of data, and is particularly beneficial if consecutive points are related or if the precision required varies over the input range. If the data is highly uncorrelated the resulting submap will be close to a scalar quantization.

A brief introduction to vector quantization can be found in [11].

2.2.7 Decoding

It may be worthwhile to point out that the time consuming calculations involved with the psycho acoustic model analysis are only performed at the time of encoding – during decoding these steps are not needed. Hence the psycho acoustic model in a typical encoder can be improved over time, even though the actual standard is long since established. This is the case with e.g. MP3, where recent encoder versions show

significant improvements in terms of audio quality. Of course decoder hardware also benefits from this in terms of simplicity and thus cost.

Chapter 3

Ogg

Development of the source code used in the current versions of Ogg/Vorbis started in the fall of 1998, shortly after Fraunhofer decided to strengthen its control of MP3 and threatened to sue free MP3 projects. Initially the Ogg/Vorbis source code was based on an earlier project initially started by Christopher Montgomery in 1993. The first stable codec version, 1.0, was released on the 19th of July 2002. This version complies to the Vorbis I standard.

The idea was to develop a general purpose high quality perceptual audio codec truly free in terms of licenses, patents and royalties. It was also a reaction to the ongoing commercialization of MP3 and other non-free audio codecs.

Christopher Montgomery founded the Xiph.org foundation that is now the home of Vorbis. It is a collection of open source multimedia related projects. Most notable are Vorbis (audio), Theora (video, in early stage) and Ogg, a container format for audio and video capable of including e.g. Vorbis and Theora. Xiph.org is named after the "Xiphophorus helleri", or Swordtail, an aquarium fish which also decorates the official Xiph.org logo. Vorbis is named after a character that appears in the Terry Pratchett novel "Small gods".

The main project branch uses floating point arithmetics exclusively, which obviously is not suitable for embedded systems and other platforms with restricted resources. A fixed point implementation of the Ogg/Vorbis decoder standard called "Tremor" has been developed and was initially released in September 2002. Both versions are released under a BSD like license, included in appendix B.

The audio quality of data encoded according to the format is generally accepted to be in the same league as MPEG-4 (AAC) and somewhat higher than MPEG-1/2 layer 3 (MP3), TwinVQ, WMA and PAC, even though no extensive studies have been done, i.e. listening tests.

At the moment Ogg/Vorbis is tuned for bit rates of 16-128 kbit/s per channel but can still be used to encode audio at any arbitrary bit rate. It is currently not viable to encode audio at a fixed bit rate since the Vorbis encoder currently only supports VBR (variable bit rate) – while forcing a fixed bit rate is possible if the subjective audio quality is lowered. The reason is that Ogg/Vorbis does not have a method for evening

out bursts of plosive sounds or other features requiring a temporary raised bit rate available in other formats, e.g. the bit reservoir used by MP3 or the constrained VBR used by AAC.

3.1 Encoding

The Ogg/Vorbis standard only specifies the decoding procedure, leaving more flexibility to the encoder. That is, the behavior of a compliant decoder is specified down to the bit level of the processed packets, but nothing is stated about how the packets are to be generated. This might seem counter productive at the first glance, but the intent is to allow improvements at the encoder side over time, not only by tuning parameters but also by incorporating entirely new methods for e.g. psycho acoustic modeling and analysis, quantization approach and generation of codebooks.

Some interesting aspects of the encoding process, which to a large extent follows the generic encoding process described in figure 2.1, will be briefly discussed.

1. Window

Vorbis uses two window sizes, called short and long. The window lengths must be a power of two between 64 and 8192, typically 256 and 2048. When plosive sounds are processed through an MDCT, spreading artifacts like pre- and post- echoes are introduced. The effect appears as a faint echo that can be heard before and after the actual sound. A shortened window size reduces this effect, and therefore shorter windows are used to suppress it. Otherwise long windows are used to achieve better spectral resolution. The human ear is not as sensitive to post-echoes as it is to pre-echoes, so short windows are not used for cutoffs.

Vorbis I uses the following window shapes for lapping:

$$y = \sin(\pi/2 * sin^2(((x + 0.5)/n) * \pi)) \tag{3.1}$$

A plot of typical long, short and transition windows can be seen in figure 3.1.

This window achieves a good trade off between good stop band attenuation and low ringing effects, see section 2.2.3.

2. Time - frequency transform (MDCT)

Although the standard has room for several transform types, only type 0 which is the MDCT is currently defined. A probable candidate in future versions is the Discrete Wavelet Transform (DWT), either as a stand alone transform or as a hybrid variant.

3. Psycho acoustic masking

Ogg/Vorbis uses a different method than most codecs in its model of absolute threshold of hearing(ATH). The most common approach is to use a model with fixed vol-

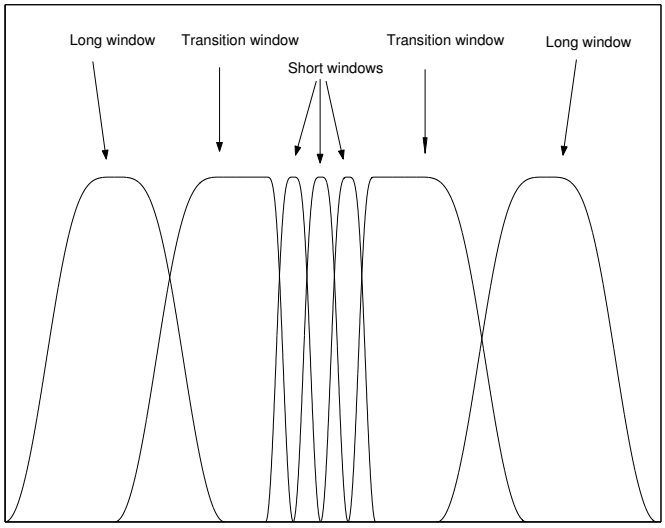


Figure 3.1: Short, Long and Transition windows as used by Ogg/Vorbis

ume for the duration of the song, where Vorbis assumes that the volume is adjusted dynamically with a maximum at the pain threshold. That is, the listener may adjust the volume during playback, e.g. raise the volume during faint sections and lower the volume during loud sections.

The masking follows the same ideas as described in 2.2.2. For each band, the audio energy, the tonality and the masking threshold is calculated followed by removal of inaudible sounds.

4. Codebooks

As opposed to the approach taken by most other audio codecs Ogg/Vorbis does not define a set of static codebooks (used for lossless entropy coding of quantized spectral data) known in advance; codebooks are instead defined and transmitted as a part of the audio stream although they can only be sent prior to any actual audio data. The obvious advantage is that codebooks may be adapted to the actual data being encoded while the big disadvantages are that codebooks can not be stored in ROM. Therefore the require bit rate needs to be increased due to the initial transmission of codebooks. Current versions of the Ogg/Vorbis encoder includes roughly 16kB of codebooks in an audio stream, which at a data rate of 128kbps means a delay of 1s before playback can begin.

5. Floor generation and encoding

The frequency spectrum can be thought of as a floor function with accompanying residues. The floor function is an approximation of the spectral envelope, in Vorbis I

represented as a piecewise polynomial. The floor curve is calculated during encoding and then subtracted from the spectrum leaving the residues.

6. Floor removal

The floor is subtracted from the spectrum leaving the uncoded residues.

7. Residue generation and encoding

The residues are VQ encoded (2.2.6) and Huffman packed. The corresponding codebooks used for decoding are sent in the setup header.

3.2 Decoding

An Ogg/Vorbis stream consists of Vorbis packets embedded in an Ogg bit stream. An Ogg bit stream is a container stream format able to hold data of most kinds. The Vorbis packets can be one of four types. The characteristics and function of the four defined Vorbis packet types will be very briefly discussed. For an in-depth description of the packets, down to a bit level precision, as well as the Ogg bit stream format which will not be discussed at all, see [12].

Identification packet

Identifies the stream as Vorbis, and specifies version and audio characteristics, sample rate and the number of channels. Sample rate can be specified using maximal, minimal and nominal bit rate. An identification packet must be directly followed by a comment packet or another identification packet (which will reset the setup process).

Comment packet

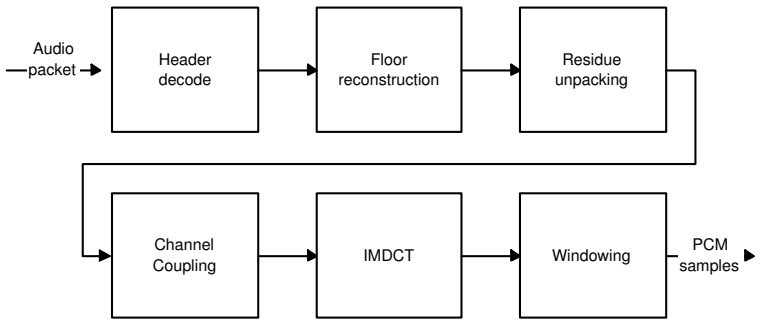
Contains title, artist, album and other meta information. A comment packet must be directly followed by a setup packet or an identification packet (which will reset the setup process).

Setup packet

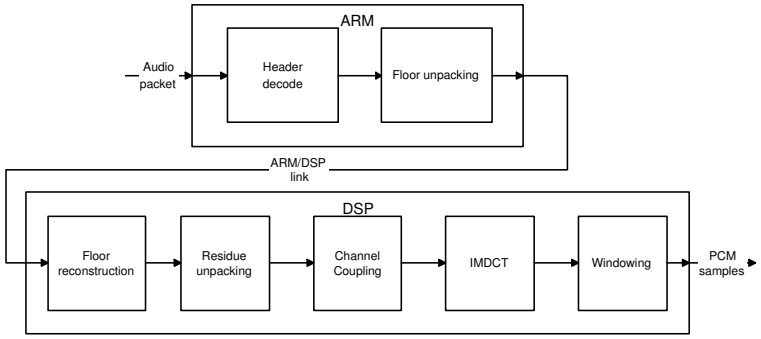
Specifies codec setup information, VQ and Huffman codebooks. A setup packet must be directly followed by an audio packet or an identification packet (which will reset the setup process).

Audio packet

Contains audio data. After the three mandatory setup packets are received, in the given order, the stream is setup and only audio packets or an identification packet (marking the beginning of a new stream) must follow. If the setup packets does not appear in the given order or audio packets appear before the three setup packets, the



(a)



(b)

Figure 3.2: (a) High-level audio decoding process, (b) Partitioned version of the decoder

stream is considered undecodable. Audio packets are self contained, and lost packets is not a problem in terms of decoder integrity. In fact coarse seeking (i.e. seeking at the packet level) is done by simply skipping packets either backward or forward. It should be noted however that playback can not be resumed until two sequential audio packets have been received, since frame lapping has to be done (this applies to most transform coders such as MP3, AAC etc.). The Ogg/Vorbis standard does not specify what actions should be taken in the case of occasional missing packets (e.g. repeated playback of the last decoded packet or interpolation).

Audio packets are basically decoded as shown in figure 3.2a, albeit the figure is somewhat simplified. A more thorough description of the process can be found in [12]. Figure 3.2b describes how the decoding steps are split between host and slave in the split decoder which is described in section 4.

The split decoder specifies its own intermediate frame format which is described in 4.4.

Audio packet decode procedure:

1. decode packet type flag (ensure that packet is of audio type)
2. decode mode number
3. decode window shape
4. decode residue into residue vectors
5. generate floor curve from decoded floor data
6. compute dot product of floor and residue
7. IMDCT, transform from audio spectrum to time domain samples
8. overlap/add
9. store right hand-data from transform of current frame for future lapping
10. return PCM-samples from overlap/add, if not first frame

Mode and mapping

The format allows the bit stream to setup a number of modes for decoding. Each mode specifies frame size, transform type, window type and mapping number. Vorbis I only allows window and transform type 0 (Window shape as given by equation 3.1 and IMDCT). The mapping number corresponds to a defined submap, specifying floor and residue functions. Submaps are also set up by the bit stream. Each submap includes information about which floor function and residue packing style that are to be used. Channels in the audio stream may use different submaps, thereby allowing different spectral characteristics, e.g. a dedicated bass channel with increased low frequency spectrum resolution.

Window

During parsing of the setup header, two window sizes are specified in PCM samples per channel. The legal window sizes are all powers of two from 64 to 8192 samples, although only window sizes of 256 and 2048 are currently used.

Floor

Vorbis I specifies two floor functions, floor0 and floor1, of which only floor 1 is used.

Floor0 uses Line Spectral Pair (LSP, alternatively known as Line Spectral Frequency, LSF) to encode a spectral envelope curve as the frequency response of the LSP filter. This is equivalent to LPC (linear predictive coding) and can be converted to an LPC representation. Decoding the floor curve is done in two steps. First curve amplitude and filter coefficients are extracted from the bit stream, and then the floor

curve is computed which is defined as the frequency response of the decoded LSP filter. This floor type is not used anymore, due to poor performance and high decoder complexity compared to floor 1. Floor 0 was replaced in an early beta, but is still part of the standard end hence needs to be supported by a compliant decoder.

Floor1 uses a piecewise linear function to encode the spectral envelope curve, which is represented on a linear frequency axis and a logarithmic (dB) axis. The floor curve is coded iteratively as described by figures 3.3 a–d. The example uses a list of selected X values {0,16,32,48,64,80,96,112,128} which are sent in interleaved form {0,128,64,32,96,16,48,80,112}. The corresponding Y-values are {110,20,-5,-45,0,-25,-10,30,-10}. The first step is to draw the line between the start and end point, and then iteratively change the Y-values for the X-values. In a) the mid point is lowered five points from 65 to 60, and continuing the the following X-values in b) and c) until the points have been processed in d). The Bresenham line drawing algorithm is used for line interpolation [13].

Huffman entropy coding is used to compress the floor coefficients in both methods.

Residue

The residues are VQ encoded according to one of three possible packing modes, which mainly differ in how the values are interleaved. Although the vectors may be encoded monolithically (i.e. encoded with a single pass) they are often encoded as an additive sum of several passes through the residue vector using more than one VQ codebook, to accomplish efficient codebook design. The VQ codebooks are of the same format as the floor codebooks, and some codebooks are used in both stages. The multi pass decoding is the main reason for the high time complexity of this step, see section 5.3.1.

Spectral reconstruction

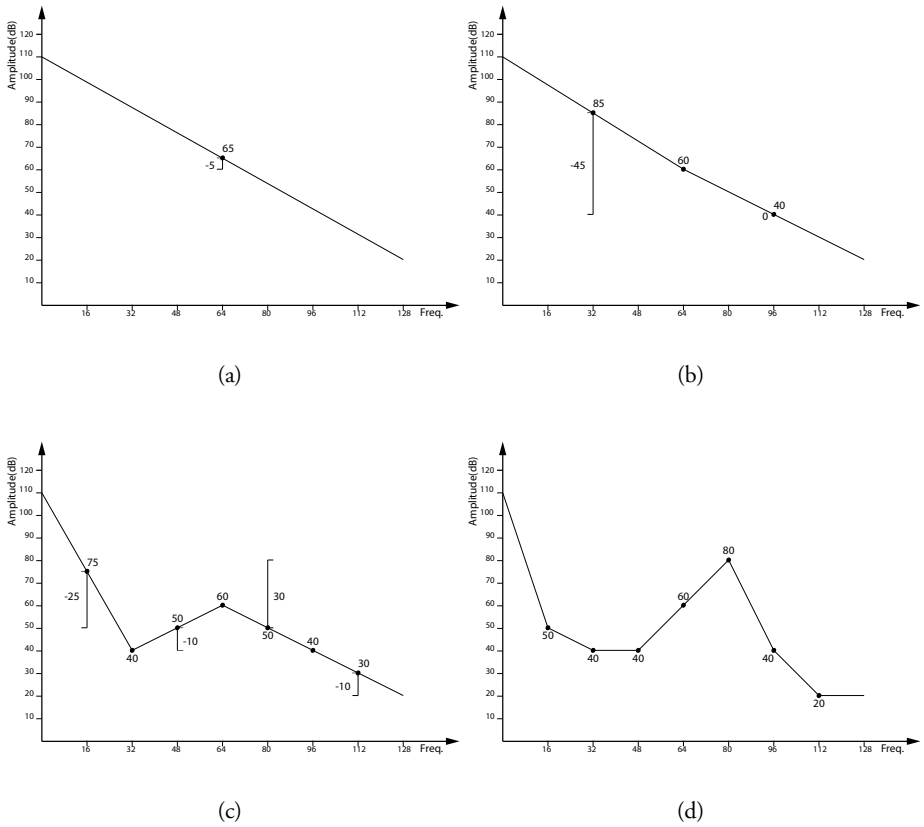
The residues are added to the previously generated spectral envelope to finalize the spectral reconstruction. The resulting output of this step can then be inverse transformed to the time domain in the next decoding step. The results of the floor and residue decoding steps as well as the final reconstruction can be seen in figure 3.4 a–c.

IMDCT

Vorbis I specifies that the frequency to time domain transformation of the restored frequency domain data is to be done using the IMDCT discussed in 2.2.3.

Overlap/add

The first half of the frame currently being decoded and the stored second half of the preceding frame are windowed with the window function specified by equation 3.1 and added to form the PCM output. Windowing is done on the time domain output of the IMDCT step.



(a) (b)

(c) (d)

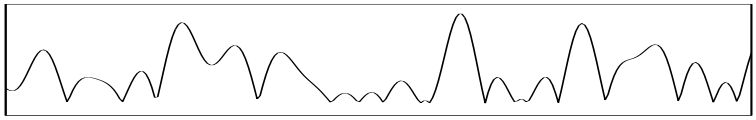
Figure 3.3: Example of floor reconstruction

The windows are critically lapped and as earlier noted may be one of two different sizes. When switching between short and long windows special transition windows are used, see figure 3.1.

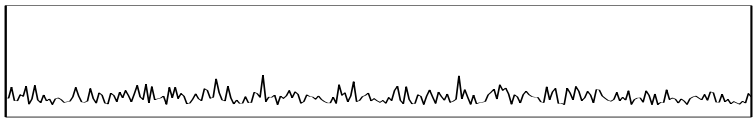
Since the MDCT windows are 50% critically overlapped, the second half of the current window has to be stored in a buffer in order to be lapped with the following frame (i.e. during the next decoding pass).

PCM-samples

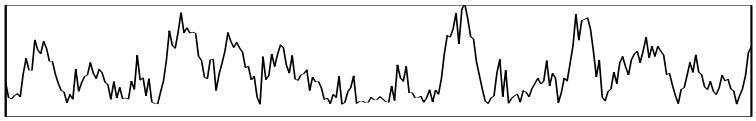
After lapping with the preceding window the result is decoded PCM samples, ready to be output in any desired fashion (e.g. sent to a D/A converter, written to file, etc.). Since every frame has to be lapped, the output of the first decoder pass is zeroed out as only the right part of the first frame contains valid audio data.



(a) Floor curve



(b) Residues



(c) Reconstructed spectral data

Figure 3.4: Spectral reconstruction during audio decoding

3.3 Tremor

This work is based on Tremor, which as described earlier is an open-source fixed point implementation of a Vorbis decoder, written by Xiph.org. There are two Tremor versions available: a general purpose implementation primarily aimed at ARM based decoding, and a low mem version aimed at DSP decoding. As the name suggests the low mem branch uses substantially less memory than the standard version does but with a CPU usage penalty. Tremor does all calculations using 32-bit fixed point arithmetics, and rounds the result down to 16 bit PCM samples.

Both Tremor versions can be compiled in a “low accuracy” mode where no 64 bit values are used to store intermediate 32-bit multiply results, which leads to a performance increasement at the cost of decoder fidelity.

Tremor is written in portable C, designed for easy porting and integration in larger projects. Ogg stream parsing is built in and the Tremor Ogg/Vorbis decoder does not need any libraries other than the standard C library. Stream I/O is done with a callback interface, and thus the decoder does not need any knowledge of the nature of the decoded stream.

3.4 Future

Ogg Vorbis is today considered to be an "immature" codec in the sense that there is need for tweaking in various parts of the encoder (e.g. the psycho acoustic model). Although the quality is maybe not as good as AAC, it surpasses other formats like MP3, WMA and might have the potential to be on par with AAC. . All coders used today are based on the MDCT transform, and are thus comparable in terms of CPU usage.

Wavelets, as opposed to Fourier transforms that only have good locality in the frequency domain, also have good locality in the time domain. As described in 2.2.5 this reduces the problems of attacking sound and temporal artifacts. There have been proposals suggesting that a hybrid Wavelet/MDCT transform should be included in future Vorbis versions. According to this scheme Wavelets are to be used for attacking sounds as the DWT has better time-resolution suitable for transients and IMDCT will be used for tonal sounds since it has better characteristics for stationary signals.

3.5 Comparison with other audio formats

Comparing the audio quality with other audio formats is never an easy task. However it is generally accepted that Ogg/Voribs offers a higher quality than MP3 and probably slightly lesser that AAC. This is very subjective since the formats use different techniques for audio compression and each format has its strengths and weaknesses. The different encoders are also differently tuned for different bit rates. One of Ogg/Vorbis strengths is low bit rates where it clearly surpasses MP3 and is equal to AAC.

MP3 [6] was one the first generally widespread codecs and is still today the most common format for music encoding. It suffers from a few drawbacks, e.g. too low time resolution for transient signals. Some of the features:

- 32 band polyphase quadrature filter (PQF)
- 36 or 12 tap MDCT.
- alias reduction postprocessing
- bitreservoir (pseudo VBR)

MPEG-2 AAC [7], MPEG-4 AAC [14] is the successor of MP3. The main drawback is the restrictive and expensive licensing. It uses three different profiles: main, low complexity (LC) and scalable sampling rate (SSR), where the most commonly used is LC. Each profile provides a different decoding process. Some of the features:

- window size 256, 2048
- M/S matrixing stereo.
- Long term prediction, LTP.

- Temporal noise shaping, TNS.
- Perceptual noise substitution, PNS.

Explanation of the methods used in MP3 and AAC can be found in [6, 7, 14].

3.6 License

There exist over 30 licenses that are approved by the Open Source Initiative (OSI) but the two most commonly used are the BSD- and the GPL-license. The Tremor code is released under a BSD like license (appendix B), which seems like the best choice for quick acceptance of the format in both software and embedded systems.

BSD license

BSD, or Berkley Source Distribution originated from the university of California, Berkley. Compared to GPL it is more business friendly since it allows commercial closed source projects derived from the code. It is also a good choice for reference implementations of standards.

GPL license

The most popular of the open source licenses. All derivates must also remain open source under the GPL license. One of the advantages is that even though it won't prevent competing companies to use that code, it keeps them from adding proprietary extensions.

3.7 Ogg/Vorbis in embedded systems

Ogg/Vorbis was unfortunately not designed with an embedded market in mind, which is quite evident in certain parts of the specification.

No limit of codebook size

The total allowed codebook size is virtually infinite (the total allowed size is 4G of codewords). This is unrealistic in terms of compliance of embedded system implementations, and in effect no really compliant system can be implemented with today's hardware.

No realistic fixed bit rate

Unlike other VBR coders (e.g. AAC) Vorbis does not currently support fixed bit rates in an acceptable way. If a Vorbis stream is encoded with a fixed bit rate the result will be unnecessary poor audio quality, while a stream encoded with a given nominal bit rate does not guarantee a maximum upper bit rate deviance as AAC and other codecs does. This is unfortunate since it is not known in advance how much of the stream

needs to be pre-cached in an input buffer when streaming at the nominal bit rate to guarantee continuous playback.

Multiple window sizes supported

The permitted window sizes includes all powers of two from 32 to 8192 in length, albeit only 256 and 2048 are used in the currently available encoder. In a memory constrained environment this is a considerable waste of resources as memory has to be reserved for storage of IMDCT data. With a long window size of 2048 points, 2048words + 1024words per channel are needed in our implementation to decode any currently available stream, but to support the standard 12kwords per channel must be allocated, i.e. 24kwords for a stereo stream. This means that 9kwords per channel are never used and thus wasted. This is clearly unacceptable for a small DSP-based or other memory constrained system.

Uncertain future

The flexibility of the standard is one of Vorbis' biggest strengths but also one of its weakest points. It allows the encoder to mature and increase the audio quality but at the same time imposes the threat of rendering current implementation useless since compromises have to be made and as mentioned several times no fully compliant decoder can be implemented.

Possible solutions

It's the strong belief of the authors that the shortcomings of Ogg/Vorbis with regards to embedded systems could easily be remedied by a more strict standard that keeps the flexibility but deals with the above mentioned shortcomings. This could be done either by updating the Vorbis standard or specifying an "embedded profile".

Chapter 4

Implementation and Partitioning

All code developed in this project is based on the low-memory branch of the Tremor Ogg/Vorbis decoder implementation, see section 3.3. Two optimized versions of the decoder have been developed. The first one is a DSP only version that runs as a stand-alone program, consuming an Ogg/Vorbis stream and producing PCM samples. In the second version the decoder is split in two processes of which the first one handles high-level Ogg/Vorbis stream parsing and the second process does the actual audio decoding of the pre-decoded output. Typically the first process runs on a CPU-based host while the second runs on a DSP slave.

The motive for splitting the decoder process is to make Vorbis decoding in a severely memory constrained DSP environment possible, e.g. with less than 64kB free RAM. Except for the assembly language optimizations all code is written in portable C, with as few assumptions about the target platform as possible. Since there are C implementations of the assembly optimized functions it should be trivial to port the decoder to other platforms. As a matter of fact the implementation compiles and is verified to execute correctly not only on the intended DSP target but also on a Pentium class PC or Sun workstation.

4.1 Dynamic memory handling

Tremor, as available from Xiph.org, handles memory by calling standard libc functions (i.e. `malloc()`, `calloc()`, `free()` and `alloca()`). Since an important goal of the project is to have strict control over the amount of memory used by the decoder processes, functions providing decoder internal memory management are added. This is done by providing the decoder with as much memory as it is allowed to use at decoder instantiation. The stand alone decoder features a full set of standard C memory allocation functions while the slave part of the split decoder only implements `malloc()`, `calloc()` and `alloca()`. It is not necessary to implement `free()` in this case as all memory is allocated when a stream is setup for decoding and freed on the beginning of a new stream. It should be noted that in both decoder versions the specialized version of `alloca()` allocates memory on a decoder internal stack instead

of on the system stack. As opposed to the automatic freeing of memory when using the standard `alloca()` function handling of the "stack pointer" has to be done manually upon returning from a function where temporary memory has been allocated. Furthermore, `free()` features debugging facilities for detection of memory leaks.

When the decoder is instantiated the creator provides pointers to the memory chunks that will used as heap (by `malloc()` and `calloc()`), stack (by `alloca()`) and in the case of the slave DSP process, the memory used as codebook cache. The sizes of these blocks are decided at runtime by the creator. Pointers to the working buffers are also passed to the algorithm upon instantiation. Figure 4.1 shows the memory model used.

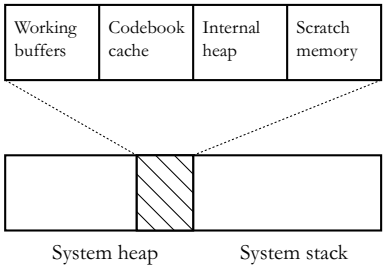


Figure 4.1: The various memory segments allocated upon decoder instantiation

Since the decoder thus knows the exact amount of memory that it is allowed to use, and all memory is allocated by the user prior to decoder execution, the goal of strict control over memory usage is achieved. If the available memory is too small (e.g. when trying to open a stream with a very large set of codebooks) the decoder will simply report this to the user, instead of trying to allocate more memory on it's own – potentially with catastrophic results.

4.2 Host – DSP

The goal when designing the split decoder was to keep memory usage on the slave (DSP) at a minimum while keeping host CPU usage as well as the data rate between the processes low. Thus all functions regarding Ogg bit stream handling and parsing, error checking, etc. are done in the host process. This is motivated by the fact that some codebooks are used exclusively for encoding floor coefficients, as discussed in section 5.2.2, only residue decoding is done on the slave while floor coefficients are decoded on host. This reduces the number of codebooks that have to be stored on the DSP, since the floor codebooks are kept on the host. The drawback is that the data rate between host and slave is increased by approximately a factor of two, e.g. a typical song encoded at a data rate of 120kbit/s results in a data rate of 240kbit/s between host and slave.

If the residues were decoded on the host too, even more memory would be freed on the DSP but at the cost of a very high data rate – comparable with sending an uncompressed PCM stream, i.e. 1.4Mbit/s for 44.1kHz 16-bit stereo.

A high level representation of the split decoder processes is available in figure 3.2 a–b.

4.3 Codebook cache

Management of the codebook cache (described in section 5.2.2)is done exclusively by the host process – the DSP process does not even check for possible overflows of the cache. When receiving a new codebook the host has to inform the slave of where in the cache the new codebook should be stored, and possibly how to move old codebooks. The current cache algorithm only considers codebook age i.e. the oldest codebooks are replaced when a cache miss occurs. It may be worthwhile to also consider codebook length since it varies between just a few tens of entries to more than a thousand.

4.4 Host – DSP packet format

To ease communications between the host and the DSP, a new packet format, inspired by the original Vorbis format, was developed. Four different packet types are defined, of which three are setup packets and the last packet type is a pre-decoded audio frame.

The three available setup packets used for communication between host and DSP are:

1. First setup packet
This packet type contains basic setup information, and may appear anytime in a stream. Upon receiving a packet of this type the decoder will tear down all data structures and free all allocated memory. Two or more succeeding initial setup packets are not considered an error condition. After parsing the packet the decoder will report whether it will be able to decode the stream or not.

2. Second setup packet
The second setup packet holds all information needed to setup data structures that are not described by the first packet type – except for code books. The second packet must not appear anywhere in a stream but directly after a first setup packet. If this condition fails an error is reported and the stream is rendered undecodable until another packet of the first setup type is encountered.

3. Codebook packet
As the name suggests, codebook packets holds codebook data. Codebook packets can appear anywhere in a stream that has been setup successfully. Since the codebook cache is managed remotely on the host instead of locally on the DSP the codebook packet also contains detailed information needed for cache management, e.g. the address at which the codebook data is to be stored.

4. Audio packet
The audio packets are slightly modified pre-decoded Vorbis audio packets. As discussed earlier the floor curve coefficients are sent unpacked. If a codebook needed during residue unpacking has not already been received, or if it is no longer cached, the audio frame is undecodable and an error will be reported. As with Vorbis audio

packets, packet skipping is not a problem. It should be noted however that if seeking is done in an already pre-decoded stream, as opposed to seeking in the actual Ogg/Vorbis stream, codebook packets appearing in the stream during seeking must be handled carefully.

All packets begin with a mandatory header to facilitate resynchronization in the case of synchronization loss. There is no checksum or other error checking information available as the original Ogg/Vorbis stream has already been CRC checked on host. The decoder will check every frame for the synchronization header, and report an error to the host if synchronization appears to be lost. In the case of loss of synchronization no audio data will be output (the user may output zeroes if need be), and internal decoder state will not be altered.

A low-level description of the intermediate packet format together with the set of possible error codes returned by the decoder is available on request.

4.5 Host – DSP communication error handling

In the case of streaming Ogg/Vorbis over an unreliable link it is very important to be able to detect missing or broken packets to ensure stable execution of the decoder. Data corruption in the floor part of an audio packet, for instance, might have fatal consequences if the data corruption is not detected (the decoder might e.g. try to read and write at unsafe memory locations). Fortunately Ogg/Vorbis provides very good means for detecting data corruption (e.g. by CRC checksums in Vorbis packets). In the split decoder the processes are expected to communicate over a reliable hardware link, and provided that the Ogg/Vorbis stream is properly checked for errors by the host, it is safe to assume that the slave receives proper data. Thus the only error detection available is the checking for frame headers to ensure that synchronization can be re-established e.g. in the case of abnormal shut-down of the host process.

4.5.1 Reentrance

To make the decoder implementations XDAIS compliant (see section 6.3) all functions must be reentrant in a pre-emptive environment, to make it possible to execute two decoder instances in the same memory space. The need for reentrance basically infers the same limitations as does thread safe coding, except for data synchronization which is not an issue when only reentrance is required. When programming in C the most evident limitation is that no `static` variables or data structures may be used, instead pointers to data structures must be passed to all functions that need knowledge about decoder state. This results in some CPU and memory bandwidth overhead due to the extra pointer dereferencing and argument passing needed.

4.6 Word size

The smallest available word size on the intended target DSP platform (TI C55xx) is 16 bits. This means that when programming in the C programming language `char`,

`short` and `int` all refer to the same 16-bit integer data type. The Tremor code base unfortunately mixes declaration of variables and data structures of explicit widths with using standard C types and making assumptions of the width of these. The assumptions made are that a `char`, `short` and `int` are 8, 16 and 32 bits respectively. These assumptions leads to two different types of problems:
1. In a lot of places the assumption is made that for instance an `int` is 32 bits wide and thus able to keep integer values between -2^{31} and $2^{31} - 1$, which clearly is not true on the target platform. This could easily have been avoided by always using data types with known widths when there is a need to store large values. The trivial approach is to replace all occurrences of `int` with `long` (32 bits on the TI C55xx). This is not a good solution since many of the used `int`s can safely be left as is since a 16-bit integer will be enough to store all possible values (replacing such a variable with a 32-bit integer would waste both memory and CPU-time). Fortunately it is possible to determine the actually needed data width by carefully reading the source code.
2. Vectors of the type `char[]` are extensively used to keep data (e.g. when reading from the input stream and to keep codebook entries). This does not affect the output of the decoder as the former problem does, but instead wastes memory. For example, Tremor by default uses a `char[1024]` as input buffer, assuming that this will allocate a 1024 entries long vector with 8-bit entries. When compiled for a TI 55xx a vector with 1024 16-bit entries will instead be allocated, using twice as much memory to hold the same amount of data. This problem is not as easily remedied as the former since functions that processes the vectors assumes 8-bit entries and thus have to been rewritten to properly work with 16-bit entries. Another problem, partly related to word size, is that the C55xx has strict requirements on word alignment (i.e. even though byte-level addressing is used 16-bit integers must be aligned on a 2-byte boundary and 32-bit integers must be aligned on a 4-byte boundary). The original code was rewritten to solve these problems and inconsistencies, resulting in a memory usage on par with that on a byte-addressable platform, e.g. ARM.

Chapter 5

Optimization

5.1 Target hardware

Optimization is targeted at two platforms, motivated by the project goal of implementing two optimized decoder versions. The first one is a stand alone decoder running on a DSP based platform and the second is partitioned between parsing and pre-decode on a host system, and only actual audio decoding on a DSP slave. Most optimization efforts, in terms of both CPU time and memory usage reduction, is spent on the audio decoding functions shared by both decoder versions. Platform dependent optimizations (i.e. assembly language function implementation) is targeted at a the TI TMS320C55xx DSP described in section 6.2.

5.2 Memory

Memory requirements is one of the most critical factors when building an embedded system. Thus the initial step during the optimization process was to generate exhaustive memory profiling data, see table 5.1. Since the main Tremor branch freely allocates memory and also assumes that memory consumption is not critical, it uses far more memory than acceptable. Fortunately some work has already been put into developing another branch of the Tremor code, a low memory branch suited for DSP implementation. Memory requirements are reduced in this version, albeit with a MIPS penalty. As table 5.1 shows memory usage is reduced to less than half of that used by the main branch. Therefore the low memory branch was used for further development in this project.

The three main goals upon reducing memory footprint are to reduce code size, data structures and buffer sizes. These three areas are further discussed in the following sections.

Decoder version	Tremor	Tremor lowmem
Code	90kB	80kB
Lookup tables	52kB	52kB
Data (heap usage)	256kB	70kB

Table 5.1: Memory usage when compiled for IA32 with MSVC++

5.2.1 Lookup tables

The lookup tables used by the decoder process are used to keep window shapes, as well as pre- and post-rotation factors (used before and after FFT) and FFT twiddle factors. Since Ogg/Vorbis specifies window lengths up to 8192 samples these tables use quite a lot of memory, see table 7.3. As discussed in section 3.3 the Tremor code uses 32-bit fixed point data rounded to 16-bit PCM samples at the end of the decoding pipeline. Reducing the lookup table precision to 16-bit while keeping the data path of 32-bit halves the lookup table memory consumption at the expense of a typical added mean error of less than one LSB and a typical worst case error of two LSBs. The added error is without doubt acceptable considering the gain in memory usage. Reducing the width of the lookup table data also improves overall execution time, most notable in the FFT kernel, since the multiplications are done on one 32 bit and one 16 bit integer instead of on two 32 bit integers. This leads to that each multiplication can be done with 2 multiplications and one addition instead of with 3 multiplications and two additions.

In the case of the FFT only one lookup table of the same length as the longest supported transform size is needed. To compute shorter transforms the same table can be used by simply skipping values in the LUT. Unfortunately, this method can not be used for windowing and FFT-rotations since the window look up tables do not share the same symmetrical properties. In both cases the reason is that a constant divided with the window length is added in the cosine and sine functions in such a way that the different windows needs their own lookup tables.

Additional memory can be reserved by storing lookup tables compressed rather than uncompressed, either lossy or non-lossy.

In the split decoder version only lookup tables in actual use needs to be kept in DSP RAM since they can be stored on the host system and inserted into the bit stream upon opening a new file. It is also possible to generate lookup tables at run time, but this introduces some delay before the first sample can be produced.

5.2.2 Codebook cache

Ogg/Vorbis audio streams show a strong temporal locality in codebooks used for Huffman decoding of both residue and floor data. In the split version of the decoder on-chip DSP memory can be conserved by exploiting the temporal locality of used codebooks by keeping only the most recently used codebooks on the DSP. This requires all codebooks to be kept in the memory of the host and also increases the overall bit

rate from host to DSP since codebooks have to be inserted into the bit stream during actual decoding.

Codebooks are used both during floor and residue decoding. Some codebooks are used only when decoding the floor function and vice versa. This allows the codebook storage to be decreased further on the DSP if floors are sent unpacked, but adds some load to the host as it has to perform Huffman decoding and also further increases bit rate between host and slave.

5.2.3 Data structures

To further reduce memory usage, all data structures are optimized by determining the actual needed member sizes as well as removing unnecessary or redundant members. Some data structures that are used to keep data of different types are split in separate versions. For instance, the data structure used to keep codebooks is replaced with two separate versions, one used for keeping codebooks of type 0 and the other for keeping codebooks of type 1 and 2. This is possible due to that some members in the original struct are not needed to describe a codebook of type 0 while others are not needed when describing a codebook of the other types. Due to the alignment restrictions of the C55xx, as described in 4.6, struct members are rearranged when necessary to assure that memory is not wasted due to padding (e.g. when an odd number of `int`s is followed by a `long`).

5.2.4 Reducing code size

Although significant code size reduction is certainly possible by either carefully rewriting the C code or implementing functions in assembly language not much time has been spent on this, due to time limitation.

5.2.5 Further memory reduction

Since the current Ogg/Vorbis standard includes features that are not currently used, some very unlikely to ever be used (e.g. window lengths of up to 8192 samples), and others that are simply outdated (e.g. floor type 0), it is possible to remove support for these features from the decoder while still being able to play back almost every available Ogg/Vorbis stream. Of course the decoder can no longer be considered fully compliant with the standard. The features that can safely be reduced from the decoder process are: code for support of floor type 0, look up tables for floor type 0, look up tables for IMDCT support of window lengths other than 256 and 2048 and look up tables for windows of length other than 256 and 2048. In our implementations support for floor 0 as well as window sizes of more than 2048 can easily be removed at compile time. The memory gain is significant, especially when considering the slave decoder process, see tables 7.2 and 7.3.

5.3 Complexity

5.3.1 Initial profiling

Profiling is the basis of the optimizations. Without knowledge about how much CPU time each block uses, it is easy to concentrate optimization efforts on blocks that offer very low overall performance improvement. Initial profiling showed that a few function blocks used a disproportionate large share of the execution time.

One reference Ogg/Vorbis stream (test vector) was used during profiling to make profiling data consistent and comparable over time. The full set of test vectors was mainly used to verify compliance of the optimized decoder, but some profiling data was also gathered to check for difference in relative CPU time usage when decoding at various bit rates. Decoding at higher bit rates spends more CPU time in function blocks where data is Huffman decoded (i.e. floor and residue unpacking).

Table 5.2 shows the profiling results of decoding a 39.5s test vector before optimization, using the original Tremor code, and with the DSP master clock running at 200MHz.

Block	%	time
MDCT	65.1	13.8
unroll	13.2	2.8
inverse MDCT	51.9	11.0
floor	8.0	1.70
coupling	1.4	0.30
inverse residue	24.1	5.11
various	1.4	0.29
All blocks	100	21.2

Table 5.2: Before optimization

It is quite clear that the MDCT transform is the main candidate for optimization efforts, and that the Huffman transform in the inverse residue function is next in line.

5.3.2 IMDCT / FFT

In the original Tremor code, the audio spectrum to time domain PCM-samples conversion is done using an IMDCT algorithm based on the paper by T. Sporer, K Brandenburg and B. Edler [15]. This is one of the most wide-spread algorithms used in audio decoding.

As seen in table 5.2, the IMDCT is by far the most time consuming block of the decode process when running the original Tremor code. There exists a few different approaches to optimization of the block. One is to assembler optimize the existing IMDCT functions for the DSP. Even though the IMDCT algorithm is not more mathematically complex than the FFT kernel, the IMDCT involves more steps and does not benefit from the same regular structure as the FFT does. Another approach

is to replace the IMDCT with an FFT kernel combined with accompanying pre- and post-sort functions to achieve mathematical equivalence with the IMDCT. The FFT which executes very fast on all DSPs since the FFT has beneficial regular properties. This method has proved successful in terms of computational complexity as reported by e.g. A.D. Duenas et al [16] even though some additional steps are needed as discussed later.

Based on the algorithm by Rolf Gluth from his paper "Regular FFT-related transform kernels for DCT/DST-based polyphase filter banks" [17], the algorithm exploits the symmetry of FFT-data and the relationship between discrete Fourier transform (DFT) and discrete cosine transform (DCT).

Definition of DCT^{IV}:

$$DCT^{IV}_{[K]_k} = C^{IV}_k = \sum_{r=0}^{K-1} x_r cos(\frac{\pi}{4K}(2k+1)(2r+1)) \tag{5.1}$$

The DCT can be rewritten as an odd-time odd-frequency discrete Fourier transform (O^2DFT):

$$O^2DFT_{[N]_k}\{\underline{u}\} = \mathcal{U}_k = \sum_{r=0}^{N-1} u_r e^{-j\frac{\pi}{2N}(2k+1)(2r+1)} \tag{5.2}$$

Using a 2K-point zero padded vector $\underline{x}'$

$$x'_r = \begin{cases} x, & 0 \le r \le K-1 \\ 0, & K \le r \le 2K-1 \end{cases} \tag{5.3}$$

the DCT is the real part of the $O^2DFT_{[2K]}$ of $\underline{x}'$

$$C^{IV}_k = \Re\{O^2DFT_{[2K]_k}\{\underline{x}'\}\} \tag{5.4}$$

Using the symmetry of O^2DFT:

$$\begin{aligned} \mathcal{P}_k &= \mathcal{U}_{2k} + j\mathcal{U}_{N/2+2k} = \\ &= 2\sum_{r=0}^{N/4-1}(u_{2r} - ju_{N/2+2r})e^{-j\frac{2\pi}{N}(2k+\frac{1}{2})(2r+\frac{1}{2})} \\ &= 2\underbrace{e^{-j\frac{2\pi}{N}(k+\frac{1}{8})}}_{rotation}\underbrace{\sum_{r=0}^{N/4-1}((u'_r e^{-j\frac{2\pi}{N}(r+\frac{1}{8})})e^{-j\frac{2\pi}{N/4}rk})}_{N/4\ point\ FFT}, \end{aligned} \tag{5.5}$$

where

$$u'_r = u_{2r} - ju_{N/2+2r}, N = 4i; i = 1, 2, \ldots;\ r, k = 0, 1, \ldots, \frac{N}{4} - 1$$

From the definition of DCT^{IV} (5.1), MDCT (5.6) and IMDCT (5.7), for even length input (always the case in Vorbis) the IMDCT is essentially equal to the DCT^{IV} and can be calculated by shifting and rearranging the input order.

$$MDCT : f_k = \sum_{r=0}^{2K-1} X_r cos(\frac{\pi}{2K}(2k+1)(2r+1+K)) \tag{5.6}$$

$$IMDCT : y_r = \frac{1}{K} \sum_{k=0}^{K-1} X_k cos(\frac{\pi}{2K}(2k+1)(2r+1+K)) \tag{5.7}$$

Using this conclusion and the results from equation 5.5 it is possible to calculate the IMDCT using the following algorithm.

- Pre-sort
- Pre-rotation
- N/4-point complex FFT
- Post-rotation
- Post-sort

The pre-sort step reorders the inputs from an N/2-point vector to an N/4-point complex vector.

$$\begin{aligned} \Re u'_{r_i} &= 0.5 * in_{2i} \\ \Im u'_{r_i} &= 0.5 * in_{N/2-1-2i} \\ i &= 0, \ldots, N/4 \end{aligned} \tag{5.8}$$

The pre-rotation is formulated from equation (5.5).

$$p_k = e^{-j\frac{2\pi}{N}(u'_k+\frac{1}{8})} \tag{5.9}$$

N/4-point complex FFT

$$\mathcal{P}_k = FFT(p_k) \tag{5.10}$$

The post-rotation uses exactly the same rotation as the pre-rotation. The post-sorting restores the order of the vectors to the same format as the original IMDCT output

vector.

$$\begin{aligned}
out_{2i} &= \Re\,\mathcal{P}_{i+N/8} \\
out_{2i+1} &= -\Im\,\mathcal{P}_{N/4-1-i} \\
out_{2i+N/4} &= \Im\,\mathcal{P}_{i} \\
out_{2i+1+N/4} &= -\Re\,\mathcal{P}_{N/8-1-i} \\
out_{2i+N/2} &= \Im\,\mathcal{P}_{i+N/8} \\
out_{2i+1+N/2} &= -\Re\,\mathcal{P}_{N/8-1-i} \\
out_{2i+3N/4} &= -\Re\,\mathcal{P}_{i} \\
out_{2i+1+3N/4} &= \Im\,\mathcal{P}_{N/4-1-i} \\
i &= 0, 1, \ldots, N/8
\end{aligned} \tag{5.11}$$

The IMDCT processes an N/2 point input vector and generates an N point output vector that is 50% critically lapped with adjacent windows. Equation (5.11) shows that 50% of the output data is redundant and thus only the first half of the output vector needs to be stored. The second half only contains elements from the first half but in negated and/or reversed form. This also applies to each half window and the storage needed for all buffers is N/2 data points for the FFT buffer and N/4 data points for the right hand side of the window stored for future lapping. The sorting step is actually not implemented as part of the IMDCT, but as a part of the unroll function. The sorting, lapping with previous window and window functions are combined in that function.

Another important property of the algorithm is that all calculations can be performed in-place, i.e. no intermediate buffers are needed for calculations.

By using this method for calculating the IMDCT the calculations execute more than 8 times faster than when using the original C code, see table 5.4.

5.3.3 Assembler / C

Optimized assembly algorithm implementation can yield a tremendous performance gain, but might on the other hand be very time consuming as the task of debugging assembly is more tedious than debugging a high level language. Thus, only the most computation intense function blocks are hand optimized in assembler. This includes the FFT, post and pre rotation before and after the FFT and overlap/add of FFT data combined with windowing (unroll). This approach proved very successful yielding an overall performance improvement of 242%. Another problem with assembler optimizing code is that code portability is no longer achieved.

The target for the optimization is a DSP, and a reference PC-version FFT was implemented in C. The C version is a standard three nested loops, radix-2 decimation in frequency, FFT with the last two stages unrolled. This version was only used for initial algorithm validation and profiling. The DSP version is the 32-bit FFT supplied by Texas Instruments [18].

Table 5.2 shows profiling of the original Tremor code, and in table 5.3, 5.4 shows profile data of the optimized FFT and optimized FFT and post- and pre-rotations. It is quite clearly an impressive performance boost.

5.3.4 Profiling

The switch from an MDCT based frequency to time domain transform to an FFT discussed in 5.3.2 significanlty improves the performance. Table 5.3 shows the profiling data generated after initial FFT optimization, with the FFT kernel used is written in assembler but pre/post-rotations as well as sorting are still not optimized (same test vector as when profiling the original code).

Block	%	time	gain
MDCT(FFT)	48.9	7.04	1.96
unroll	18.5	2.67	1.04
inverse MDCT	30.3	4.37	2.51
post rotation	11.0	1.59	
pre rotation	12.2	1.75	
bit reverse	2.2	0.31	
FFT	4.9	0.70	
sort	0.1	0.02	
floor	6.7	0.97	1.75
coupling	2.8	0.41	0.73
inverse residue	39.8	5.73	0.89
various	1.7	0.25	1.16
All blocks	100	14.40	1.47

Table 5.3: FFT replacement, only FFT-kernel assembler optimized

The pre/post-rotation, windowing, and the lapping in the unroll function (sorting of final FFT data) all have a very regular structure of multiplications, additions and sorting, making them very suitable for assembly language implementation. This is verified by table 5.4 which shows the profiling data when these functions are replaced with assembly optimized versions (same test vector as when profiling the original code).

Block	%	time	gain
MDCT(FFT)	18.9	1.65	8.36
unroll	2.9	0.25	11.2
inverse MDCT	16.0	1.40	7.8
post rotation	3.1	0.27	
pre rotation	1.6	0.14	
bit reverse	0.9	0.08	
FFT	10.1	0.88	
sort	0.3	0.03	
floor	11.7	1.03	1.65
coupling	4.6	0.40	0.75
inverse residue	61.9	5.42	0.95
various	2.7	0.25	1.16
All blocks	100	8.75	2.42

Table 5.4: FFT, replacement, optimized assembler, 39.5s long test vector

This shows a substantial improvement for the optimized blocks, and also highlights blocks that are candidates for future optimizations, e.g. Huffman unpacking. It should also be noted that some blocks have increased in time. This is mainly due to the switch from simply using a `char[]` vector for storing buffered input data and codebooks, without packing two 8-bit values in each vector entry (remember that the smallest word size on the target platform is 16 bits), to using vectors of half length with packed data as described in section 4.6.

Chapter 6

DSP

6.1 DSP Processors

The use of DSP processors in embedded system designs is increasing rapidly. The market for general purpose DSP processors reached $6.0 billion in 2003, while the market for embedded DSP processors reached $10.4 billion, both increasements of about 15% from 2002.

DSP processors are a specialized class of CPUs optimized for signal processing tasks, i.e. highly repetitive and numerically intensive tasks. Most DSP processors use a Harvard architecture approach to it's design, accomplished either by separate data and instruction memories or by separate buses to a single, multiported, memory. The most important features will be briefly discussed. For a more thorough review of DSP processors see [19].

Fast Multiply-Accumulate

Clearly the most important feature of any DSP processor is the ability to perform a multiply–accumulate (MAC) in a singly instruction cycle. The MAC operation is useful in any algorithm that involve computing a vector product, such as FIR filters, folding and FFTs. To accomplish the single-instruction MAC a multiply-accumulate engine is integrated as a part of the processor.

Multiple-Access Memory Architecture

Most DSP processors share the ability to perform several accesses to a single memory in a single instruction cycle. Most can also fetch an instruction word while at the same time store or fetch at least one operand, due to the Harvard architecture nature. To support simultaneous access of multiple memory locations, DSP processors feature multiple on-chip memory busses, and sometimes even multiple off-chip memory buses. In contrast with a general purpose CPU where on-chip memory is automatically filled by hardware cache algorithms, on-chip memory is explicitly addressed and used by the programmer.

Specialized Addressing Modes

To keep the MAC units working and to allow specification of multiple memory addresses in a single instruction word, DSP processors come with dedicated address generation units. The address generation units typically support a number of addressing modes useful for DSP algorithms. The most common is register-indirect addressing with post-increment, which useful in situations where a repetitive computations is performed on a series of data stored sequentially in memory. Other special addressing modes (called circular or modulo addressing) are often available so simplify the use of circular data buffers. Some processors also support bit–reversed addressing to avoid the bit–reversing step of the FFT algorithm.

Specialized Execution Control

Because of the highly repetitive nature of many DSP algorithms, most DSP processors provide support for efficient looping. Often special instructions are available that allows the programmer to implement loops without the need for explicit testing and updating of the loop counter or spending instructions on jumping back to the start of the loop.

Data path

As already mentioned most DSP processors feature at least one MAC unit in the data path, which is typically 16–32 bits wide. Although floating point DSP processors are available the lion's share of the market is attributed to processors limited to integer arithmetics. To facilitate computations DSP processors typically have hardware features to interpret integer calculations as fixed point calculations (e.g. by automatic shifting of multiply results).

Accumulator registers hold intermediate and final results of MAC operations and are typically wider than the multiplier output width by several bits. The extra bits, or guard bits, allows subsequent accumulation without risk for overflow, thus removing the need for scaling of intermediate results. An accumulator with n guard bits allows for up to 2^n values to be accumulated without the possibility of overflow.

The ALU of a typical DSP processor implements the same basic operations as that of a general purpose CPU (e.g add, subtract, increment, negate, *and*, *or* and *not*. Some DSP processors use the ALU to perform the addition in MAC operations while others have separate adders for this purpose.

In contrast with general purpose CPUs, DSP processors often incorporate saturation logic to lessen the impact of overflow errors. This special logic detects an overflow situation and replaces the erroneous output with the largest positive or negative value possible depending on overflow direction.

Limitations and advantages

There is a wide variety of DSP chips available on the market with different configurations and features. For high quality audio processing where the output is typically 16

bit PCM samples, the computations have to be done with higher intermediate precision (e.g. 24 or 32 bits) due to quantization effects. This does not imply that only high precision DSPs can be used since high precision can be accomplished by the execution of several lower precision instructions. Although higher clock speed is required, a less complex chip can be selected, reducing cost, chip area and power consumption.

6.2 Platform 55xx

The DSP processor used in this project is the Texas Instruments TMS320C5510, which is a member of the 55xx set of DSP processors featuring:

- 200 MHz clock speed
- Modified Harvard architecture
- Three address generation units
- 16-bit internal data path
- Dual 16-bit ALU
- Dual multiply–accumulate (MAC) units
- Four accumulators
- 40-bit accumulator width (8 guard bits)
- 64 kB on-chip DARAM (dual access RAM)
- 256 kB on-chip SARAM (single access RAM)
- Up to 11 RISC equivalent instructions executed in a single cycle

The C55xx family of DSP processors was introduced in 2000 and is an improved version of the 54xx family, providing full backwards compatibility while introducing a large set of new features (e.g. dual 17x17-bit MAC units, a second 16-bit ALU, four 40-bit accumulators and four new data registers) as well as higly extended power save features.

The development board used, a Spectrum Digital 5510 DSK[1], was donated by Texas Instruments. This board features 16MB SDRAM, onboard audio codec, debug facilities (e.g. USB/JTAG) and the possibility to install expansion boards.

[1] Technical information about the development board is available at http://www.spectrumdigital.com

6.3 TI – XDAIS

The TMS320 DSP Algorithm Standard, also known as XDAIS [20, 21], is a standard of coding conventions and application interface (API) that all compliant algorithms must follow. The goal of the standard is to define an interface for third-party algorithms. Non-standardized algorithms often need to be re-engineered in order to integrate into different systems. Compliant algorithms must implement a standard caller-algorithm interface which makes it easy in integrate in other systems. Another aspect is that memory, stack and CPU usage must be characterized, which makes shopping algorithms from third-parties somewhat easier.

The most important rules of the standard are:

- C callable.
- Reentrant, within preemptive environment.
- No memory allocation. Memory allocated by caller.
- Code must be fully relocatable.
- Implementing the IALG interface (algorithm interface).
- No direct access to any peripheral device.
- Characterize the memory, stack and cpu usage.

Both optimized decoder versions (DSP only and host/DSP) are compliant with the XDAIS standard.

Chapter 7

Results and future improvements

7.1 Audio quality

Objective audio fidelity of a perceptive audio codec can be measured only in terms of mathematical deviation between input and output. In this case between the original PCM-samples that are used as input to the encoder and the output PCM-samples of the implemented decoder. Different audio coders with the same deviation as calculated by a conformance algorithm may however show different subjective audio fidelity, which has to be measured as an average over a large population listening test. This only applies to the quality of the encoder. In order to verify the correctness of a decoder implementation the output is simply compared to the output of a reference decoder. In this case, the floating point implementation of Ogg/Vorbis is used.

The conformance of an audio decoder can be tested using a number of different tests. One of the most common tests is to use the root mean square (RMS) error combined with the maximum deviation from a reference vector. This method is used for qualifying implementations of for instance MPEG-1 and AAC as compliant decoders. The RMS error is calculated as follows:

$$RMS = \sqrt{\frac{1}{N}\left(\sum_{i=1}^{N}(t_i - r_i)^2\right)} \tag{7.1}$$

where N is the number of samples, t_i is the decoder sample and r_i is the reference sample.

In this project a small set of seven test vectors are used to verify the correctness of the optimized decoder, i.e. that it generates the same output as the chosen reference decoder. To accomplish the needed calculations Matlab scripts are used extensively, performing e.g. the calculations of RMS and maximum deviation as well as the simple, yet very helpful, method of plotting the difference between outputs.

Maximum deviation and RMS error computed by comparing the outputs of the optimized decoder as well as the various Tremor flavours with the output of the reference decoder can be seen in table 7.1.

Decoder version	RMS	Maximum deviation
Original Tremor	$2.67 \cdot 10^{-5}$	$9.15 \cdot 10^{-5}$ (3 LSB)
low-accuracy Tremor	$1.96 \cdot 10^{-3}$	0.0171 (565 LSB)
Optimized Tremor	$2.15 \cdot 10^{-5}$	$9.15 \cdot 10^{-5}$ (3 LSB)

Table 7.1: RMS error and maximum deviation of decoder output

The Ogg/Vorbis standard does not state any limits within which a decoder can be considered compliant, nor does it provide a set of test vectors or reference output signals. To give an apprehension of the figures it is thus useful to compare with part 4 of ISO/IEC 11172 [22] which states that a compliant MPEG-1 decoder "shall provide an output such that the RMS level of the difference signal between the output of the decoder under test and the supplied reference output is less than $2^{-15}/\sqrt{12}$ $[8.81 \cdot 10^{-6}]$ for the supplied sine sweep (20Hz–10kHz) with an amplitude of -20dB relative to full scale. In addition, the difference signal shall have a maximum absolute value of at most 2^{-14} relative to full-scale." It later states that "To be called a limited accuracy ISO/IEC 11172-3 audio decoder, the decoder shall provide an output for a provided test sequence such that the RMS level of the difference signal between the output of the decoder under test and the supplied reference output is less than $2^{-11}/\sqrt{12}$ $[1.41 \cdot 10^{-4}]$ for the supplied sine sweep (20Hz–10kHz) with an amplitude of -20dB relative to full scale."

It is interesting to note that our implementation, which uses only 16-bit constants, produces more accurate output than does the original Tremor decoder, and performs much better than the low accuracy version of Tremor, which uses 23-bit data and 9-bit constants. The low accuracy version yields a maximum error of 565LSSB, i.e. with a maximum error of 11 bits. The reason for the bad results of the original Tremor code is most likely sub optimal rounding from intermediate 32-bit data to 16-bit PCM samples.

7.2 Memory

The project goal of running an Ogg/Vorbis decoder on a platform with less than 64kB of memory is unfortunately not met for the full decoder, even if support for floor0 and unused windows are omitted. The DSP part of the split decoder meets the goal with good measure though, needing as little as 50kB of memory including the decoders share of the system stack when support for unused windows and floor0 is dropped.

Memory usage figures for the optimized decoder versions are shown in tables 7.2 and 7.3 below. There is obviously a very large gain in memory usage when omitting support for large window lengths. It is also worth pointing out that the code size of the slave decoder is less than half of the full decoder.

All memory figures are from the final code when compiled with full optimization

and no symbols, the needed heap size assumes at most two audio channels.

Version	DSP	DSP w/o floor0	DSP 256/2048
Code	15448	13404	13404
Constants	43334	41920	7680
Floor 0	1412	0	0
Floor 1	1024	1024	1024
Twiddle factors	24544	24544	4352
Window shapes	16352	16352	2304
Data	46080	46080	27648
Local heap	34816	34816	16384
Local stack	1024	1024	1024
Codebook cache	10240	10240	10240
Sum	104862	101404	48732

Table 7.2: Memory usage for the slave decoder (bytes)

Version	Full decoder	Full decoder w/o floor0	Full decoder 256/2048
Code	31650	29036	29036
Constants	43334	41920	7680
Floor 0	1412	0	0
Floor 1	1024	1024	1024
Twiddle factors	24544	24544	4352
Window shapes	16352	16352	2304
Data	83968	83968	71680
Local heap	77824	77824	65536
Local stack	6144	6144	6144
Sum	158952	154924	108396

Table 7.3: Memory usage for the full decoder (bytes)

The “DSP 256/2048” version is the slave decoder compiled without support for any windows other than 256 and 2048 and without support for floor 0. As discussed in section 5.2.5 supporting only these window lengths makes it possible to decode any stream coded with the current encoder version albeit with an uncertain future.

“DSP w/o floor 0” supports all window lengths but does not support floor 0. Dropping support for only floor0 can be considered almost safe as it is very unlikely that any decoder will ever use floor0 again as floor1 is superior in terms of both bit rate and complexity, see section 3.2.

The memory usage of the full featured slave decoder process (denoted “DSP”) is showed in the third column of table 7.2. The large increase of the needed local heap size when all window lengths are allowed is mainly due to the need for a much larger working buffer (12kB / channel).

The same naming scheme is used for the various versions of the full decoder.

The host decoder process uses approximately 50kB memory, 15kB of which is code while the rest is used for storing data, mainly codebooks.

7.3 CPU usage

Decoder CPU usage has been reduced significantly in the optimized decoder versions compared with the original non optimized source. Table 7.4 shows the MIPS usage when decoding a 112kbit test vector at various stages of optimization. The presented data applies to the slave decoder. CPU usage of the full decoder is slightly higher, adding about 2-3 MIPS depending on content bit rate. Note that MIPS figures are a rough estimate, derived from the DSP clock rate of 200MHz and the time needed to decode 39.5s of data.

Decoder version	MIPS
Original code	107
Optimized FFT kernel	72.9
All optimizations turned on	44.5

Table 7.4: MIPS usage of the slave decoder at different stages of optimization

As the content bit rate increases, so does the CPU usage. Typical MIPS usage values for the same content encoded at various bit rates when decoded with the optimized slave decoder are printed in table 7.5. The main reason for the increase of CPU usage is that more residue data has to be Huffman decompressed – as discussed earlier this is the most computationally complex block after optimization.

Bit rate	MIPS
92	41.4
112	44.5
158	52.6
226	58.5

Table 7.5: MIPS usage of the slave decoder at various bit rates (all optimizations turned on)

7.4 Future improvements

Optimized Huffman

During initial profiling the most CPU consuming block by far was the frequency to time domain transform (IMDCT) and thus a lot of effort was spent on optimizing that block. As seen in table 5.4, after replacing the IMDCT implemented in C with the assembler optimized FFT 63% of the DSP CPU time is spent on Huffman unpacking (inverse residue). This suggests that further optimization efforts should target this block.

RTP streaming

The number of internet radio stations transmitting Ogg/Vorbis audio streams is rapidly increasing (e.g. BBC, http://www.icecast.org) and many of the PC based audio players (e.g. Winamp, XMMS) are now shipped with plugins to support decoding and internet streaming of Ogg/Vorbis.

All current Ogg/Vorbis internet streaming is done using HTTP over TCP which is not an ideal solution, for instance multicast is unsupported and ACKs must be sent back to the transmitter. A better solution would be to use RTSP or RTP combined with RTCP, both of which are targeted at real time distribution of multimedia.

Streaming Ogg/Vorbis is somewhat problematic as each logical audio stream begins with setup headers including codebooks then followed by audio packets. When connecting to a multicast audio stream the decoder first has to get the proper set of codebooks before playback can begin. The Ogg/Vorbis standard has not yet specified a proper method of doing this, but several proposals exists:

- Broadcast a codebook only stream as a second multicast stream as well as the Vorbis audio stream.
- Periodically retransmit the headers in the stream.
- Let the receiver request the codebooks via RTCP.

More efficient codebook cache

The current host based codebook cache algorithm does not take into account the size of the codebooks, which varies from just a few tens of bytes to several kBytes. By exploiting the properties of size a more efficent cache algorithm could be implemented, to reduce the increased bit rate caused by transmission of codebooks.

Further reduction of memory usage

As seen in tables 7.2 and 7.3 memory usage is as low as 105kB for the single processed decoder and 48kB for the DSP part of the split decoder. While this is a significant improvement over the original memory usage it still might not be enough.

Motivated by this a few possible methods for further reducing memory usage in future implementations will be presented.

Reducing the size of look up tables
The main part of memory in the DSP part of the split decoder is used for storing look up tables. Thus reducing the size of look up tables is highly desirable. A method to accomplish this is to store the LUTs compressed, e.g. by storing the difference from the last value or storing only a subset of the needed values, using interpolation to estimate the missing points. Unfortunately this method can only be achieved at the cost of some added complexity, and in the case of interpolation, precision will be reduced.

Storing look up tables on host
In the split decoder it would be possible to permanently store LUTs on the host where memory usage is assumed to be non critical and transmit the needed tables to the decoder as a part of the setup header.

Generate look up tables at run time
The calculations used for building the look up tables are well defined and thus the needed tables could be built at run time. It is doubtful if this would provide any gain in terms of memory usage however as a number of math library functions have to be included (e.g. `sinf()`, `logf()` etc). Computing the needed values on the fly (i.e. not using LUTs at all) is not an option as the added complexity in terms of computations is much too big.

The DSP part of the split decoder can use as little as less than 50kB of memory at the cost of not supporting all defined window lengths or floor type 0. As discussed in section 5.2.5 this can safely be done as no encoder generates files using longer windows or the removed floor type. If an encoder using these features would be implemented and thus decoder support has to be added, it might be possible to use a combination of the described improvements to keep memory usage below 50kB instead of further reducing it.

Chapter 8

Conclusion

To conclude the work done in this project we have shown that it is possible to run an Ogg/Vorbis decoder in a strictly memory constrained embedded environment. We have also shown that Ogg/Vorbis is not as suitable for integration in embedded systems as are most other coders, mainly because of memory requirement issues.

Two decoder versions were implemented. One stand-alone decoder and one split in two processes, where the first process handles high-level stream parsing, typically on an ARM based host, and the second does actual audio decoding on a DSP slave. Both decoder versions run and generate correct output when executed on the target platform (TI TMS320C55xx).

Implemented optimizations have resulted in significant performance improvements, both in MIPS and memory usage, and it is our opinion that it is realistic to run an Ogg/Vorbis decoder in an embedded environment with limited hardware resources.

The major problem with Ogg/Vorbis decoders in embedded systems is without doubt the lack of a maximal codebook size. The Ogg/Vorbis standard unfortunately also contains unused features, which in our opinion should either be dropped from the standard or declared optional instead of mandatory. Most problems would be sorted out by reducing the maximal window length and codebook size, this would ideally be done by defining an embedded Ogg/Vorbis profile.

There is still much room for further optimization improvements of both memory and CPU usage, and some possible methods to accomplish this have been introduced.

Appendix A

List of abbreviations

AAC	Advanced Audio Codec
AC3	Audio Compression 3 (Dolby)
ALU	Arithmetic and Logic Unit
ATH	Absolute Threshold of Hearing
CBR	Constant Bit Rate, Critical Band Rate
CRC	Cyclic Redundancy Check
DARAM	Dual Access RAM
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
DWT	Discrete Wavelet Transform
DSP	Digital Signal Processor / Digital Signal Processing
DST	Discrete Sine Transform
FFT	Fast Fourier Transform
GPL	General Public Licence. www.gnu.org
IMDCT	Inverse Modified Discrete Cosine Transform
LSB	Least Significant Bit
LSF	Line Spectral Frequency
LSP	Line Spectral Pair
LTP	Long Term Prediction
LUT	Lookup Table
MAC	Multiply–ACcumulate
MDCT	Modified Discrete Cosine Transform
MIPS	Million Instructions Per Second
MP3	MPEG I layer III
MPEG	Moving Picture Expert Group
NMN	Noise Masking Noise
NMR	Noise to Mask Ratio
NMT	Noise Masking Tone
PCM	Pulse Coded Modulation
PNS	Perceptual Noise Substitution

PQF	Polyphase Quadrature Filter
RISC	Reduced Instruction Set Code
RMS	Root Mean Square
RTCP	Real-Time Control Protocol
RTP	transport protocol for real-time Applications (RFC 1889)
RTSP	Real-Time Streaming Protocol
SARAM	Single Access RAM
SDRAM	Synchronous DRAM
SMR	Signal to Mask Ratio
SNR	Signal to Noice Ratio
SPL	Sound Pressure Level
TMN	Tone Masking Noise
TNS	Temporal Noise Shaping
VBR	Variable Bit Rate
VQ	Vector Quantization
WMA	Windows Media Audio
XDAIS	TMS320 DSP algorithm standard

Appendix B

Tremor license

Bibliography

[1] T. Painter and A. Spanias. Perceptual coding of digital audio. *Proc. IEEE*, 88(4), april 2000.

[2] B. Scharf. Critical bands. In J. Tobias, editor, *Foundations of Modern Auditory Theory*, volume 1, chapter 5, pages 159–202. New York Academic Press, 1970.

[3] H. Fletcher and W. Munson. Relation between loudness and masking. *J. Acoust Soc. Amer.*, 9:1–10, 1937.

[4] R. P. Hellman. Asymmetry of masking between noise and tone. *Perception & Psychophysics*, 11(3):241–246, 1972.

[5] J. L. Hall. Auditory psychophysics for coding applications. In V. Madisetti and D. Williams, editors, *The Digital Signal Processing Handbook*, pages 39.1–39.25. CRC Press, Boca Raton, FL, 1998.

[6] Technical report, ISO/IEC MPEG 11172-3. *Information technology - coding of moving picture and associated audio for digital storage media at up to about 1.5 Mbit/s, part 3: Audio*, 1993.

[7] Technical report, ISO/IEC MPEG 13818-7. *Information technology - generic coding of moving pictures and associated audio information, part 7: Advanced audio coding (AAC)*, 1997.

[8] Deepen Sinha and Ahmed H. Tewfik. Low bit rate transparent audio compression using adapted wavelets. *IEEE Transaction on Signal Processing*, 41(12):3463–3479, 1993.

[9] H. Malvar. Lapped transforms for efficient transform/subband coding. *IEEE Transactions of acousticsm speech and signal processing*, 38(6), 1990.

[10] K. Hamdy, A. Ali, and A. Tewfik. Low bit rate high quality audio coding with combined harmonic and wavelet representations, 1996.

[11] Data compression. *Vector Quantization*. http://data-compression.com/vq.html.

[12] Xiph.org foundation. *Vorbis I specification*. http://www.xiph.org/ogg/vorbis/doc/Vorbis_I_spec.pdf.

[13] J. E. Bresenham. Algorithm for computer control of digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[14] Technical report, ISO/IEC MPEG 14496-3. *Information technology - very low bitrate audio-visual coding, part 3: Audio*, 1998.

[15] T. Sporer, K. Brandenburg, and B. Edler. The use of multirate filter banks for coding of high quality digital audio. In *Proceedings of the 6th European Signal Processing Conference*, pages 211–214, 1992.

[16] Alberto D. Duenas et al. A robust and efficient implementation of mpeg-2/4 aac natural audio coders. AES-convention, 2002.

[17] Rolf Gluth. Regular fft-related transform kernels for dct/dst-based polyphase filter banks. *ICASSP*, 3:2205–8, 1991.

[18] Texas Instruments. *TMS320C55x DSP Library Programmer's Reference*. SPRU422F.

[19] Amit Shoham Phil Lapsley, Jeff Bier and Edward A. Lee. *DSP Processor Fundamentals*. IEEE Press, Piscataway, NJ, 1997.

[20] Texas Instruments. *TMS320 DSP Algorithm Standard Rules and Guidelines*. SPRU352E.

[21] Texas Instruments. *TMS320 DSP Algorithm Standard API Reference*. SPRU360C.

[22] Technical report, ISO/IEC MPEG 11172-4. *Information technology - coding of moving picture and associated audio for digital storage media at up to about 1.5 Mbit/s, part 4: Compliance testing*, 1995.