OGG VORBIS DECODER FOR MOTOROLA DSP56002

Master thesis in Electronics Systems at Linköping Institute of Technology
by

Niklas Barsk

LiTH-ISY-EX--05/3700--SE
Linköping 2005-01-14
OGG VORBIS DECODER FOR 
MOTOROLA DSP56002

Master thesis in Electronics Systems
at Linköping Institute of Technology
by

Niklas Barsk

LiTH-ISY-EX--05/3700--SE

Supervisor: Patrik Thalin
Examiner: Kent Palmkvist
Ogg Vorbis avkodare för Motorola DSP56002

Ogg Vorbis decoder for Motorola DSP56002

Niklas Barsk

Ogg Vorbis is a rather new audio format with some similarities with other more known formats such as MP3 and WMA. It is generally accepted to have a better audio quality than most competing formats and it is in contrast to many of its competitors totally licence and royalty free. The goal with this thesis is to port the existing fixed point decoder Tremor, which is written in C, to Motorola's DSP56002. The DSP has a very limited amount of memory so some optimizations has to be made to be able to run Tremor successfully. The report presents the necessary steps taken to port Tremor to the DSP and the difficulties of this process. It also describes the memory and CPU usage of the DSP when running Tremor and other results of the port. A description as well as examples and workarounds of bugs found in the compiler g56k is attached to this report.

Nyckelord
Keyword
Tremor, Ogg, Vorbis, DSP, DSP56002, C, Audio Coding
ABSTRACT

Ogg Vorbis is a rather new audio format with some similarities to other more known formats such as MP3 and WMA. It is generally accepted to have a better audio quality than most competing formats and it is in contrast to many of its competitors totally licence and royalty free.

The goal of this thesis is to port the existing fixed point decoder Tremor to Motorola’s DSP56002. The DSP has a very limited amount of memory so some optimizations has to be made to be able to run Tremor successfully.

The report presents the necessary steps taken to port Tremor to the DSP and the difficulties of this process. It also describes the memory and CPU usage of the DSP when running Tremor and other results of the port.

A description as well as examples and workarounds of bugs found in the compiler g56k is attached to this report.
ACKNOWLEDGMENTS

I would like to thank my supervisor Patrik Thalin and the staff of ISY for all the help and support they have provided. I would also like to give thanks to my family and friends who have helped me by proofreading this thesis.

Finally I would like to pay my tributes to the Xiph.Org foundation and all the members of the Vorbis mailing lists.
# TABLE OF CONTENTS

1 Introduction. .............................................. 1  
1.1 Background ....................................... 1  
1.2 Purpose............................................ 1  
1.3 Notation ........................................... 2  
1.4 Limitation ......................................... 2  
1.5 Outline ............................................ 2  

2 Audio coding .......................................... 3  
2.1 Lossless encoding .................................. 3  
2.2 Lossy encoding .................................... 3  
2.3 Psychoacoustic encoding ......................... 4  
2.3.1 Critical bands ................................ 5  
2.3.2 Masking ....................................... 6  
2.3.3 MDCT and windows ............................ 8  

3 Ogg Vorbis.............................................. 9  
3.1 Vorbis............................................. 9  
3.2 Background ....................................... 9  
3.3 Encoding ......................................... 10  
3.4 Decoding ......................................... 10  
3.5 Tremor ............................................ 14  
3.6 License ........................................... 15  
3.6.1 BSD license .................................. 15  
3.6.2 GNU GPL .................................... 15  

4 Digital signal processing. ......................... 17  
4.1 DSP processors .................................... 17  
4.1.1 Multiply-accumulate ......................... 17  
4.1.2 Harvard architecture ....................... 18  
4.1.3 Hardware loops ............................... 18  
4.1.4 Address generation ......................... 18  
4.2 Motorola DSP56002 ............................. 18  
4.3 Software ......................................... 19  

5 Porting Tremor........................................ 21  
5.1 Memory allocation ............................... 21  
5.2 Callbacks for I/O ............................... 21
5.3  64 bit arithmetics ........................................... 22
5.4  Word size ................................................... 23
5.5  Floor 0 and window sizes ................................. 23
5.6  Making Tremor run correctly .............................. 24

6  Results and future work ................................. 25
   6.1  Results ......................................................... 25
       6.1.1 Decoding time ........................................... 25
       6.1.2 Memory usage ........................................... 27
       6.1.3 Output .................................................. 28
       6.1.4 Audio quality ........................................... 28
   6.2  Future work ................................................... 28
       6.2.1 Input file ................................................. 28
       6.2.2 Memory allocation ................................. 29
       6.2.3 Audio quality ........................................... 29
       6.2.4 Assembler optimization ................................ 29
       6.2.5 Replace the IMDCT with a FFT ................. 30
       6.2.6 Word size ................................................. 30
       6.2.7 64 bit arithmetics ...................................... 30
       6.2.8 Split header and audio decoder .............. 31

7  Conclusion .................................................. 33

8  References .................................................... 35

Appendix A: List of abbreviations ......................... 37
Appendix B: Tremor license ................................. 39
Appendix C: Bugs found in g56k ......................... 41
INTRODUCTION

1.1 BACKGROUND

As the use of digital audio and its distribution over the Internet has increased rapidly over the last couple of years the need for good compression techniques is great. The most known compression technique of today, which is almost a de facto standard for compressed audio, is MP3. In spite of its popularity it still suffers from drawbacks, such as being limited to only two channels. Several codecs have been implemented to replace MP3 but like MP3 most of them have quite restrictive licenses that involve royalty fees.

Ogg Vorbis is a relatively new audio codec that is totally free in terms of licenses, patents and royalties. Due to its BSD like license it can be used freely by anyone for almost anything whether they are open source projects or commercial close source projects.

1.2 PURPOSE

The purpose of this thesis is to explore the possibilities to decode an Ogg Vorbis stream with Motorola’s DSP56002 and to implement a system that does this. The Xiph.Org foundation has implemented an integer only decoder named Tremor that is very suitable for processors without any hardware for floating point arithmetics and with low memory. This makes Tremor a good choice and it is the basis of the work in this thesis.
1.3 NOTATION

Since the smallest addressable unit of the DSP56002 is words of 24 bits this unit will be used when mentioning sizes if nothing else is stated. As an example 1 KWord would be 1024 Words which is 1024*24 bits or 1024*3 bytes.

1.4 LIMITATION

This thesis is only focusing on making Tremor run successfully on the DSP56002. To do this some optimizations have to be done but other than that further optimizations is left for a future project due to time limitations.

1.5 OUTLINE

Chapter 2 gives a brief overview of audio coding and a short introduction to psychoacoustic encoding. Chapter 3 introduces Ogg Vorbis and the Xiph.Org foundation. It also briefly describes the encoding and decoding of Vorbis files. The fourth chapter gives an introduction to DSP processors in general and mentions the features and tools of the DSP used in this thesis. The information given in these three chapters is not strictly needed for the understanding of this thesis but they give a broader view of the surroundings of the thesis.

Chapter 5 describes the problems that occurred during the work and how they were solved. This is followed by a chapter that presents the result of the thesis and suggestions on how the decoder can be improved in the future.

The thesis is ended with a conclusion in chapter 7 and a list of references in chapter 8.

There are also three appendices included: one containing a list of abbreviations used in this thesis, a second containing the licence of Tremor and finally, the third which presents the bugs found in the compiler during the work.
This chapter gives a short introduction to audio coding and more specifically psychoacoustic encoding. For more information on this subject [4, 5, 6] is recommended.

2.1 LOSSLESS ENCODING

One category of compressing techniques is lossless coding which means that the original file can be perfectly reconstructed from the compressed copy. Due to the complex nature of audio in general it is difficult to get a good compression with lossless techniques. Examples of lossless codecs are FLAC and SHN (shorten).

2.2 LOSSY ENCODING

To achieve a much better compression rate than with lossless encoding lossy compression methods are often used and in most cases they are combined with lossless algorithms for best results.

The basic idea of lossy encoding is to remove certain parts of the sound to save even more space. Since parts of the sound that are not redundant are removed it is impossible to reconstruct the original from the encoded copy. In other words the sound loses some information during the encoding which can not be reconstructed.

Lossy encoding is actually not compression (i.e. redundancy reduction) but instead irrelevance coding (i.e. irrelevance reduction). The main objective is to remove perceptual redundancy, in other words to avoid coding sound that cannot be heard or code them with lower
accuracy to save space. If all sounds in the audio stream that are inaudible to the human ear were removed the result would have a much smaller size but still sound exactly the same. To do this a psychoacoustic model that models the human ear is needed. See figure 2.1 for an illustration of ideal lossless and lossy encoding.

![Figure 2.1: Illustration of lossless and lossy encoding.](image)

### 2.3 PSYCHOACOUSTIC ENCODING

Psychoacoustic encoding is a set of lossy compression methods that uses a psychoacoustic model to remove sounds inaudible to the human ear. Most people have a hearing range from 20Hz to 20kHz and are most sensitive to sounds around 3kHz, see figure 2.3. Sounds that have to low volume or are outside the human hearing spectrum should be safe to remove. Some sounds are masked by other sounds and these can be removed as well.

The majority of the perceptual audio encoders are based on the same basic architecture, shown in figure 2.2. First the input signals are sent to both the psychoacoustic analyser and the time/frequency analyser. The result is dynamically adjusted based on the result from the psychoacoustic analysis. The output is sent to bit allocation and quantization and these steps are also dependent of the result of the psychoacoustic analysis. Finally the data is entropy coded and multiplexed into a single bit stream.
Sound pressure level, SPL, is the standard metric for measuring sound level in decibels. It is defined as \( L_{LSP} = 20 \log_{10}(p/p_0) \) dB, where \( p \) is the sound pressure in Pascals (Pa) and \( p_0 \) is the standard reference level of 20\( \mu \)Pa.

The absolute threshold of hearing, ATH, is the sound pressure level that is needed for a single tone to be audible in a noiseless environment. It can be described by the curve plotted in figure 2.3.

2.3.1 CRITICAL BANDS

In human hearing, only those frequency components within a narrow band, called the critical band, will mask a given tone. The length of
these bands varies with the frequency and is called critical bandwidth. The unit of critical band rate, CBR, is Bark and the Bark scale is a linearization of the human ear’s frequency response. The conversion from linear frequency to CBR is given by

\[ z(f) = 13 \arctan(0.00076 f) + 3.5 \arctan \left( \frac{f}{7500} \right) \] (Bark) \hspace{1cm} (2.1)

where \( f \) is the frequency in Hz. Experiments show that the human auditory system can be approximated with the 25 bands in Table 2.1.

**Table 2.1:** Band and bandwidth for the 25 critical bands.

<table>
<thead>
<tr>
<th>Band (Bark)</th>
<th>BW (Hz)</th>
<th>Band (Bark)</th>
<th>BW (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-100</td>
<td>14</td>
<td>2000-2320</td>
</tr>
<tr>
<td>2</td>
<td>100-200</td>
<td>15</td>
<td>2320-2700</td>
</tr>
<tr>
<td>3</td>
<td>200-300</td>
<td>16</td>
<td>2700-3150</td>
</tr>
<tr>
<td>4</td>
<td>300-400</td>
<td>17</td>
<td>3150-3700</td>
</tr>
<tr>
<td>5</td>
<td>400-510</td>
<td>18</td>
<td>3700-4400</td>
</tr>
<tr>
<td>6</td>
<td>512-630</td>
<td>19</td>
<td>4400-5300</td>
</tr>
<tr>
<td>7</td>
<td>630-770</td>
<td>20</td>
<td>5300-6400</td>
</tr>
<tr>
<td>8</td>
<td>770-920</td>
<td>21</td>
<td>6400-7700</td>
</tr>
<tr>
<td>9</td>
<td>920-1080</td>
<td>22</td>
<td>7700-9500</td>
</tr>
<tr>
<td>10</td>
<td>1080-1270</td>
<td>23</td>
<td>9500-12000</td>
</tr>
<tr>
<td>11</td>
<td>1270-1480</td>
<td>24</td>
<td>12000-15500</td>
</tr>
<tr>
<td>12</td>
<td>1480-1720</td>
<td>25</td>
<td>15500-</td>
</tr>
<tr>
<td>13</td>
<td>1720-2000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**2.3.1 MASKING**

Masking is the event where one sound, the maskee, is made inaudible by another sound, the masker. Since the maskee falls outside of the human realm of perception it is irrelevant. If the maskee can be
removed from the audio stream it will have a reduced size but the same perceived quality as the original.

The different types of masking can be divided into two categories: spectral masking and temporal masking.

**Spectral Masking**

Spectral masking is when a high level tone masks a lower level tone at frequencies in the vicinity of the masker’s frequency. Figure 2.4 illustrates how a 1kHz tone masks the surrounding frequencies.

![Figure 2.4: Spectral masking.](image)

**Temporal masking**

A faint sound can be masked by a stronger sound if it is played slightly before or after the masking sound. Post-masking effects can last up to 200 milliseconds while pre-masking effects is present in about a tenth of that duration. Figure 2.5 illustrates the principles of temporal masking.

![Figure 2.5: Temporal masking.](image)
2.3.2 MDCT AND WINDOWS

To transform the audio data to the frequency domain without too much distortions a transform combined with overlapping and windowing is used. In most cases an MDCT and sine window is used. There exists various window shapes but the basic criterion is that must be followed is the Princen-Bradley condition, $w_k^2 + w_{k+N}^2 = 1$. Which means that the square sum of the overlapping window parts must be one. The reason for this is that the windows are applied twice, during both encode (MDCT) and decode (IMDCT).
OGG VORBIS

This chapter gives an introduction to Ogg Vorbis and the Xiph.Org foundation. It also briefly describes the encoding and decoding of Vorbis files.

3.1 VORBIS

Vorbis is a completely open, non-proprietary, patent-and-royalty-free, general-purpose audio compression format from the Xiph.org foundation. It is often used together with Ogg, which is Xiph.org’s container format for multimedia, and is then called Ogg Vorbis. Ogg Vorbis is a lossy audio codec tuned to, but not limited to bit rates of 16 to 128 kbit/s per channel. It is a growing audio format used mainly for online distribution. Many online radio stations have begun streaming in Ogg Vorbis or are examining the possibilities, much because of it being licence and royalty free. But also because it is generally accepted to be in the same league as MPEG-4 (AAC) and similar to, but with better performance than MPEG 1/2-audio layer 3 (MP3), MPEG-4 audio (TwinVQ), WMA and PAC.

3.2 BACKGROUND

The development of Vorbis began immediately after the copyright holders of MP3, Fraunhofer IIS, began demanding licensing fees for its product. In September of 1998 Fraunhofer IIS sent a letter of infringement to several small open source and commercial MP3 development projects. They stated that because of their patents related to MP3 they were entitled royalties for any commercial player and all encoders,
free and commercial alike. This included a fee for every file encoded.

Vorbis is based on an earlier project, OggSquish, started 1993 by Christopher Montgomery and version 1.0 was released on June 19th 2002. The purpose of Vorbis is to open up the field of audio codecs which is controlled by several dominant and completely incompatible closed “standards”.

The Xiph.Org foundation was founded by Christopher Montgomery and is a collection of open source multimedia projects with Ogg, a multimedia container format, as the most prominent one. Some of the other projects include Vorbis (audio codec), Theora (video codec), FLAC (lossless audio codec) and Speex (audio codec for speech data).

For more information about Xiph.Org and its projects see [7].

3.3 ENCODING

The Vorbis specification only describes how the decoding of a Vorbis file is done. This because Vorbis is not a CODEC specification but instead a bitstream specification that describes how a certain file should be played. How the file is encoded is not, and does not have to be, part of the standard. It does not matter how the file is created as long as it can be played using a Vorbis compliant decoder. The intention is to avoid placing restrictions on the encoder that might inhibit new methods that would increase the performance. Xiph.Org has however provided a reference encoder as an example on how an encoder can work.

3.4 DECODING

Before the decoding process can begin, the decoder must initialize the stream to be decoded. This is done using the three header packets in the Vorbis stream. The Vorbis stream begins with the three header packets and all following packets are audio packets.

The three header packets are the identification header, the comments header and the setup header.
Identification header

The identification header identifies the stream as a Vorbis stream and contains information about the Vorbis version and simple audio characteristics such as sample rate and number of channels.

Comment header

The comment header contains user comments in form of text “tags” and a vendor string for the application/library that encoded the stream. Xiph.org has proposed a minimal list of standard field names, but arbitrary comments can be used.

Setup header

The setup header contains CODEC setup information, Huffman codebooks and VQ needed for decode. The setup header packet must be followed by the first audio packet and when this is detected the setup process is terminated and the decode process begins.

Audio packets

The general decode process for the audio packets can be described as follows:

1) decode packet type flag
2) decode mode number
3) decode window shape (long windows only)
4) decode floor
5) decode residue into residue vectors
6) inverse channel coupling of residue vectors
7) generate floor curve from decoded floor data
8) compute dot product of floor and residue, producing audio spectrum vector
9) IMDCT
10) overlap/add left-hand output of transform with right-hand output of previous frame
11) store right-hand data from transformation of current frame for future lapping
12) if not first frame, return results of overlap/add as audio result of current frame.

Mode

Each Vorbis frame is coded in a special mode and a bitstream may use one or many modes. The mode specifies frame size setting, window type, transform type and a mapping number. In Vorbis I both window type and transform type is always 0 (the Vorbis window and MDCT).

The mapping number specifies with mapping configuration to use for low-level packet decode and synthesis. The configuration contains information about which floor and residue vectors to use. Different channels in the audio stream can use different mapping configurations, thus allowing the best suited configuration to be used for each channel.

Window

Vorbis frames may be one of two PCM sample sizes specified during setup. The legal frame sizes in Vorbis I are powers of two from 64 to 8192 samples per channel but the only frame sizes used in practice are those of 256 and 2048.

Vorbis uses an overlapping transformation to blend one frame into the next. The transformation output is windowed (output of current frame overlapped 50% with the output of the previous frame and added) to get a seamless reconstruction.

Floor

Vorbis encodes a spectral floor vector for each PCM channel. This is a low-resolution representation of the audio spectrum in the current frame.

There are two types of floor functions, floor 0 and floor 1, but only floor 1 is used today. The reasons for this is the more stable inter-frame behaviour of floor 1 and the fact that it is considerably less expensive to decode floor 1 than floor 0. No known encoder past Xiph.Org’s own beta 4 uses floor 0 but it is not to be considered deprecated.

Floor 0 uses Line Spectral Pair (LSP, alternatively know as Line Spectral Frequency or LSF) to encode a smooth spectral envelope curve as the frequency response of the LSP filter. This representation is equivalent to a traditional all-pole IIR filter used in linear predictive coding.
Floor 1 uses a piecewise linear function to encode a spectral envelope curve. The function plots this curve on a linear frequency axis and a logarithmic (dB) axis. The plotting algorithm used is similar to Bresenham’s line-drawing algorithm[1].

Residue

The spectral residue is the fine structure of the audio spectrum once the floor curve has been subtracted. It is coded using multi-pass VQ according to one of three specific packing algorithms.

Inverse channel coupling

Vorbis coupling is done at pairs of residue vectors and decoupling is done in-place a pair at the time. The vectors are specified in the current mapping configuration. The decoupling operation is the same for all pairs and it converts from a square polar representation (where one vector is magnitude and the other is angle) back to a Cartesian representation.

After decoupling the resulting residue vectors represent the fine spectral detail of each output channel.

Generate floor curve

The floor curve can be generated at any appropriate time, whether it is generated when the floor data is decoded from the raw packet or after inverse coupling and applied to the spectral residue directly.

Both floor 0 and floor 1 generates a linear-range, linear-domain output vector.

Compute floor/residue dot product

For each channel the floor curve and residue vectors are multiplied element by element to produce the finished audio spectrum.

IMDCT

The audio spectrum is converted back to time domain PCM audio with an IMDCT. For more information about the IMDCT algorithm used in Tremor see [9].

The produced PCM is not yet finished audio, it must first be overlapped with the surrounding frames using an appropriate window.
Overlap/add data
The windowed IMDCT output is overlapped and added with the right-hand data of the previous window. This is done so that the 3/4 point of the previous window is aligned with the 1/4 point of the current window. See figure 3.1 for an illustration of how long, short and transitions windows are overlapped.

![Figure 3.1: Example of long, short and transition windows.](image)

Return finished audio data
Now the audio data between the centre of the previous frame and the centre of the current frame is finished and can be returned. The remaining data of the current window is saved for overlapping with the next window.

In the case of same-sized windows the returned data is one half block consisting of only the overlapped portions. When overlapping windows of different size a large part of the returned data is not over-lapped.

3.5 TREMOR
Tremor is an integer-only, fully Ogg Vorbis compatible, decoder written by Xiph.Org. There are three different versions of Tremor available, one general purpose version aimed at ARM based decoding and two “low mem” versions aimed at DSP decoding. The difference between the regular version and the low mem versions is, as suggested by the name, that the low mem version uses much less memory.
but with a penalty of higher CPU usage. The two low mem versions of Tremor are one normal version and one “no-byte” version which is intended for platforms with a 16 bit short as its smallest storing unit.

All versions of Tremor can be compiled in a “low accuracy” mode which limits the size of integers from 64-bit to 32-bit to increase the portability and performance at the cost of audio fidelity.

Tremor is written in ANSI C with portability in mind and uses only the standard C library so no external libraries are needed. The I/O for reading the Vorbis files are done with a callback interface which makes it easy for the user to use their own functions for file I/O.

3.6 LICENSE

The Ogg Vorbis specification [8] is in the public domain and is therefore available to anyone for both commercial and non-commercial use. Both the libraries, the SDK and Tremor are available under a BSD-like licence. See Appendix B for the licence of Tremor. Most of Xiph.Org’s utility software for Ogg Vorbis is released under the terms of GNU GPL.

3.6.1 BSD LICENSE

The BSD license is one of the most popular licenses for free software and it has very few restrictions. The only requirements are preservation of copyright and a standard disclaimer of liability so any BSD-licensed software can be freely modified and used in proprietary or commercial software.

3.6.2 GNU GPL

GPL is the most popular licence for free software and is used among others by the GNU Linux operating system and the Linux kernel. Any derivative work must also be GPL licensed, so in contrast to the BSD licence GPLed software can not be part of a commercial closed source software.
DIGITAL SIGNAL PROCESSING

This chapter gives a brief description of DSP processors and outlines the features of the DSP56002 processor used in this project. It also describes the software used for compiling and debugging C code for the processor.

Digital signal processing or DSP is the arithmetical processing of, often real-time, signals sampled at regular intervals and digitized. Popular uses of DSP are signal filtering, mixing of signals, comparison of signals and transformation of signals. Traditionally this have been done with analog circuits but as semiconductor technology has improved and processing power of digital solutions has increased the use of DSP processors has increased rapidly.

4.1 DSP PROCESSORS

A DSP processor is a specialized microprocessor designed for digital signal processing. The processor can be used for general-purpose computations, but since it is not optimized for this it will be limited in its efficiency. The processor instead has an architecture and an instruction set optimized for the task of rapid signal processing.

4.1.1 MULTIPLY-ACCUMULATE

One very important feature of DSP processors is the multiply-accumulate unit. It allows for multiplication and accumulation in one instruction cycle, thus expressions such as \( a = a + bc \) can be computed in a single cycle. This is very useful for any algorithm that involve the computation of vector products, such as FIR filters.
4.1.2 HARVARD ARCHITECTURE

In contrast to the most general purpose processors, which mostly use the von Neumann architecture, most DSP processors use the Harvard architecture. The Harvard architecture separates the program memory and the data memory so simultaneous access of data and instructions can be made. This enables the processor to fetch the next instruction at the same time it is computing the current one, even if it needs access to the data memory.

4.1.3 HARDWARE LOOPS

Since most DSP algorithms include looping, many DSP processors have special assembler instructions for looping. This increases the performance since no time has to be spent on updating a loop counter and jumping to the beginning of the loop.

4.1.4 ADDRESS GENERATION

Another architectural feature common for DSP processors is to have a special address generation unit dedicated for address calculations. Special addressing modes such as modulo and bit-reversed addressing are often available too. Modulo addressing allows for fast circular buffers that reduce the overhead in recursive algorithms.

For further reading on DSP processors see [2, 3].

4.2 MOTOROLA DSP56002

The Motorola DSP56002 is a part of the DSP56K family which is a series of 24-bit general purpose DSP processors with a triple-bus Harvard architecture and a high degree of parallelism. Its features include:

- 20 MIPS at 40MHz
- Harvard architecture
- single cycle 24 x 24 bit parallel multiply-accumulator
- zero overhead nested do loops
• on-chip 512 x 24 program RAM (expandable to 64K off-chip)
• two on-chip 256 x 24 data RAM (both expandable to 64K off-chip)
• two on-chip 256 x 24 data ROM
• full speed memory expansion port with 16-bit address and 24-bit data buses.

For more information about the DSP56k family and the DSP56002 see [10, 11].

4.3 SOFTWARE

Suite56 DSP56000 Software Development Tools by Motorola Inc. has been used during this project. It is an environment that supports the development and debugging of software for the DSP56K family. Included in this package are code generation and code execution tools as well as documentation for all parts of the environment.

Suite56 also includes a GNU based C cross-compiler called DSP56KCC. The compiler implements the full C language defined in the ANSI C standard from 1989. The compiler generates COFF files which can be imported into the DSP simulator for debugging.
PORTING TREMOR

This chapter describes the different problems that occurred and solutions to them during the work to port Tremor to Motorola's DSP56002 processor.

5.1 MEMORY ALLOCATION

Tremor handles memory by using the standard C library functions `malloc()`, `calloc()`, `free()` and `alloca()`. This causes some problems since the compiler for the DSP56k processors does not implement `alloca()`. It is solved by using a version of D A Gwyn's implementation of `alloca()` [12] which is available in the public domain. The general concept is to keep track of all alloca-allocated blocks and reclaim any found to be deeper in the stack than the current invocation. It does not reclaim storage as soon as it becomes invalid, but it does that eventually. Garbage collection can be forced by doing a `alloca(0)` call. This will reclaim storage without allocating any new and should be used in loops containing function calls to functions using `alloca()`.

Solving it this way will unfortunately slow down the memory allocation where `alloca()` is used compared to a system which has `alloca()` implemented.

5.2 CALLBACKS FOR I/O

To decode a Vorbis file, there must be one available. The Vorbis file to be decoded is placed in an array of `chars`. This was done with the help of a Perl script that converts a file of binary data to a C-file con-
taining a char array with the binary data. This file can then be compiled and accessed quite easily from the program.

Since the standard I/O functions for reading and seeking operates on files these cannot be used, custom functions had to be written to replace the standard I/O functions \texttt{read()}, \texttt{seek()}, \texttt{close()} and \texttt{tell()}.

Since the Vorbis data is stored in memory the \texttt{close()} function is trivial to implement. With the decision that the Vorbis data should not be seekable and thus is treated strictly as a stream, the implementation of \texttt{seek()} and \texttt{tell()} also becomes trivial. Another advantage of this decision is that the open operation becomes cheaper. If a seekable stream is wanted only \texttt{seek()} and \texttt{tell()} have to be changed.

Tremor provides special functions for opening and testing Vorbis files with custom I/O functions. The only thing that has to be done is to populate an \texttt{ov_callbacks} structure with the custom functions and call the special open and/or test functions instead of the regular ones.

### 5.3 64 BIT ARITHMETICS

Even when Tremor is run in low accuracy mode, which limits the decode calculations to 32 bit precision, there are a few operations in the header decode process that needs 64 bit arithmetics. Since the DSP56002 processor’s largest data type is 48 bit, special functions to handle all 64 bit arithmetics had to be used and all arithmetics on the 64 bit data type had to be changed to use these functions. An implementation of these functions where provided by Christopher Montgomery on the Tremor mailing list [14].

But with this added, the total amount of program memory needed exceeds the maximum amount of program memory the processor can have by slightly less than 0.5 KWords. Thus, it is impossible to use 64 bit arithmetics without optimizing or in some other way reduce the code size of Tremor.

It is however possible to run Tremor without any 64 bit precision arithmetics but only if nothing more complicated than direct playback of a stream with no nonlinear access and no timecode interpretation is done.
Fortunately, this is the case in this project so Tremor can be run without 64 bit arithmetics. However the functions are implemented and can easily be used by defining a macro at compile time if wanted and if the proper reduction in code size has been done.

5.4 WORD SIZE

The smallest word size on the DSP56k platform is 24 bits. This means that the C data types `char`, `short` and `int` all are 24 bit and `long` is 48 bit. Unfortunately at some places in the Tremor code the assumption that `char`, `short`, `int` and `long` are 8, 16, 32 and 32 bits respectively has been made. Some of these assumptions makes the program produce incorrect output. One example where this is a problem is when you are assigning a `char` to its maximum value. With the assumption that a `char` is 8 bits the C-code would be:

```
ch = 0xff;
```

But on a platform where a `char` is 24 bits it should be `0xffffffff` and thus the value `0xff` is wrong. A way to do this without any assumption on the size of a char is to use the one’s complement operator `~`:

```
ch = ~0;
```

To remedy this the code had to be searched through for any assumptions like this and replace them with word size independent versions.

5.5 FLOOR 0 AND WINDOW SIZES

As mentioned in section 3.4, the use of floor 0 and other window sizes than 256 and 2048 are practically non existent today. This means that support for floor 0 and other window sizes can be removed and the decoder would still be able to decode virtually all existing vorbis files. However, when doing this the decoder is not fully compliant with the standard but since this will significantly reduce the memory usage it is an acceptable sacrifice to do, especially on a platform which has as limited memory as the DSP56k.

Section 6.1.2 will discuss the amount of memory saved in detail.
5.6 MAKING TREMOR RUN CORRECTLY

Unfortunately the compiler used to compile Tremor is a bit buggy and generates incorrect assembler at some times. This has drastically slowed down the work on this project since perfectly valid C code that should work did not work. While running Tremor on the simulator it would suddenly do something wrong which caused the program to not recognise the input file as a Vorbis file or to generate invalid data.

The only thing to do was to step through the program line by line in the simulator and on another computer running Tremor and try to find out where the simulator goes wrong. When that point was found I had to figure out what caused the problem and then rewrite the C-code to avoid that bug in the compiler.

Appendix C contains the bugs found in the compiler during the project and short code examples of the bug as well as workarounds for them.
This chapter presents the results of this thesis as well as some suggestions on ways to improve the decoder in the future.

6.1 RESULTS

Unfortunately it was impossible to test Tremor on the actual DSP since the memory requirements were much larger than the memory available on the DSP used. But the memory requirements are lower than the maximum allowed memory on the DSP so it should be possible to run Tremor on a DSP56002 with the maximum amount of memory installed. Since it was not possible to run it on the real thing making it run on the simulator had to suffice.

When running Tremor the test file that was used were encoded at quality -1 (on a scale from -1 to 10 where 3 is the default) from a one-channel wave file with a sample rate of 11 kHz and a file size of 824 bytes. The length of the file is 0.054 seconds. All data presented in this chapter are based on this test file.

6.1.1 DECODING TIME

Since Tremor has only been run on the simulator it was not possible to measure the time needed to decode the test file. Instead a theoretical value based on the number of clock-cycles used to execute the program and the frequency of the processor (40 MHz) has been calculated.

Table 6.1 shows the needed time to decode the test file with and without support for floor 0 and window sizes other than 256 and 2048. The
first column shows the memory usage of the original Tremor, in the second the support for floor 0 has been removed and in the third also the support for window sizes other than 256 and 2048 has been removed. The open row shows how much time had to be spent on opening the Vorbis file, i.e. decoding the headers and preparing for decode. The decode column shows the time for the actual decode.

### Table 6.1: Time needed to decode the test file.

<table>
<thead>
<tr>
<th></th>
<th>Tremor</th>
<th>w/o floor 0</th>
<th>only 256/2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>15.5 s</td>
<td>17.8 s</td>
<td>2.3 s</td>
</tr>
<tr>
<td>Decode</td>
<td>1.5 s</td>
<td>1.7 s</td>
<td>0.23 s</td>
</tr>
<tr>
<td>Total time</td>
<td>17 s</td>
<td>19 s</td>
<td>2.5 s</td>
</tr>
</tbody>
</table>

The reason why the opening of the file is such a large part of the total is that with as small files as the test file, the header is much larger than the audio data. The header is about 90% of the file size, which is consistent with the ratio between the execution time for opening and decoding the file. In fact, the compressed file is about twice the size of the uncompressed file due to the relatively large header.

This is a well known problem among game developers who often have many very small files that they want encoded. One solution is to make sure that all files have the same codec setup so they can share one header and only store one copy of it instead of having one copy in the header of every file.

When doing a comparison between the decode time of the file and the actual file length it becomes clear that it is nowhere near real time decoding, especially in the two first cases. The decode must be from four times up to 30 times faster for real time decoding to be possible depending on which version is used. This would be an even larger number if the file was encoded with a higher quality setting.

The time required for opening the file will create a delay before the decode can start. But this delay should remain constant if longer files of the same quality were decoded. If files with higher quality or more channels were decoded this delay, as well as the decode time, should increase due to the more complex header and more data that needs to
be decoded.

When comparing the results from the different versions of Tremor it is quite obvious that limiting the decoder to only support floor 1 and window sizes of 256 and 2048 is a good choice. The large save in decode time is very advantageous since it puts the decode time within reach of real time decoding with extensive optimizations.

6.1.2 MEMORY USAGE

Table 6.2 shows the memory usage for Tremor with and without the support of floor 0 and window sizes other than 256 and 2048. The fist column shows the memory usage of the original Tremor, in the second the support for floor 0 has been removed and in the third also the support for window sizes other than 256 and 2048 has been removed.

<table>
<thead>
<tr>
<th></th>
<th>Tremor</th>
<th>w/o floor 0</th>
<th>only 256/2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>61 359</td>
<td>57 478</td>
<td>57 409</td>
</tr>
<tr>
<td>Constants</td>
<td>16 403</td>
<td>15 693</td>
<td>9 837</td>
</tr>
<tr>
<td>Run-time Data</td>
<td>23 717</td>
<td>23 720</td>
<td>23 717</td>
</tr>
<tr>
<td>(heap + stack)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>101 479</td>
<td>96 891</td>
<td>90 963</td>
</tr>
</tbody>
</table>

The table shows that a very large amount of memory is saved by dropping the support for other window sizes. It is also a noticeable difference in code size when floor 0 is dropped.

Since the encoded file is stored in an array it contributes to the size of the constants in table 6.2.

In addition to the figures in the table above the DSP needs as much free memory space as the size of the decoded in the second data memory unit.

When looking at the memory saved by these two optimizations it shows once again that, this is worth the loss in compliance with the standard. Especially since the amount of data memory saved can be used to decode larger files.
6.1.3 OUTPUT

The output of Tremor are placed in the second memory unit since it is not used by the compiler. But this can easily be changed so the output is placed elsewhere or with some small modifications sent to an appropriate output port of the processor.

Since the no-byte version of Tremor is used and no effort has been made on changing it to take full advantage of the 24 bit word length the output will only be stored in the 16 lower bits of each memory cell. The upper 8 bits contains a garbage value that is to be ignored.

6.1.4 AUDIO QUALITY

The audio quality of the output from the decoder is the same as when running an unaltered version of Tremor in low accuracy mode on any other machine. The reason for this is that no “lossy” optimizations have been made to make Tremor run on the target platform and therefore the output is identical to that generated by the unaltered Tremor.

6.2 FUTURE WORK

During this project several things has been noticed that could have been done to improve the decoder in some way. There were not enough time to do all these improvements so there are still a number of things that could be done to this decoder.

6.2.1 INPUT FILE

To get the Vorbis file into the DSP a Perl script is used to create a C-file containing an array with the Vorbis file as binary data. The array has one byte of data in each position, but on the DSP56002 each position can hold three bytes. So if the Perl script was changed to generate an array that took advantage of the 24 bit word length the memory used to store the Vorbis file would be reduced to a third. This would also require some changes in Tremor since it only reads a byte at a time now.
6.2.2 MEMORY ALLOCATION

As stated in section 5.1 the need for the custom `alloca()` slows down the decode process. One solution could be to do all memory management manually by allocating one large array at compile time and using assembler optimized functions for allocation and deallocation of memory. This would also give a much better control over how much memory is available and the decoder could halt with an error if it runs out of memory. As it is now when the decoder runs out of memory it gets a NULL pointer as a pointer to the newly allocated memory. This causes Tremor to over write the global data stored at the bottom of the data memory and the effects are quite catastrophic.

6.2.3 AUDIO QUALITY

When Tremor is run in low accuracy mode the look up tables for the windows and for the IMDCT are converted from 32 bits to 8 bits and stored in arrays of `chars`. But on the DSP56002 platform a `char` is 24 bits wide so there are 16 bits unused for each entry in the look up tables.

It should be possible to improve the audio quality to near that of files generated by Tremor run in normal mode if the values of the look up tables where rounded to 24 bits instead of 8.

Another possibility is to stick with the 8 bit values and instead somehow store three values in each position of the look up tables. This would maintain the original audio quality of the low accuracy mode but instead cut the size of the look up tables to one third. However, this method would most likely need special functions for reading values instead of just accessing the wanted element from an array and thus slow down the decoder further. Since additional code would be needed this alternative would result in an increase in needed program memory which already is what the target platform lacks the most.

6.2.4 ASSEMBLER OPTIMIZATION

A great increase in performance as well as a reduction in code size can be achieved by rewriting and optimizing an algorithm in assembler, but this makes debugging more tedious than with a high level lan-
guage. In this decoder the IMDCT part should be the part that would benefit the most from this kind of optimization since this is where the most of the calculations during decode are done. This code is already rather assembler-like so it should be relatively easy to rewrite it to assembler.

6.2.5 REPLACE THE IMDCT WITH A FFT

Another option to assembler optimization of the IMDCT is to replace it with an equivalent FFT. Even though they have the same mathematical complexity, a DSP processor benefits more from the regular structure of the FFT. This change combined with assembler optimization could result in an improvement in efficiency of as much as 800%. For further information about this optimization and its results see [13].

6.2.6 WORD SIZE

The no-byte branch of Tremor that this work is based on is targeted on platforms with 16 bits as the smallest storage unit and makes efficient use of the extra width to save some space. However since the DSP56002 has 24 bits as its smallest storage unit there are still one third unused. Tremor should be modified so this last third is used.

6.2.7 64 BIT ARITHMETICS

As stated in section 5.3 I solved the problem with the lack of a 64 bit data type by using an implementation that substitutes the 64 bit integer with a structure containing two longs instead. But since a long is 48 bits wide this structure can hold $48 + 48 = 92$ bits. This could fairly easy be changed so that the structure contains one long and one int thus capable of holding $48 + 24 = 72$ bits. Unfortunately this only reduces the data memory usage and not the program memory usage so some reduction of the code size has to be done before these functions can be used.
6.2.8 SPLIT HEADER AND AUDIO DECODER

One way of reducing the requirements of the processor could be to divide the decoding on two processors and let one decode the header and then send the decoded header data and the remainder of the file to the other processor. This would drastically cut the requirements on memory and processing power on the decoding processor. See [13] for more information on splitting the decoder on two processors.
Ogg Vorbis decoder for Motorola DSP56002
CONCLUSION

During this project the integer-only Ogg Vorbis decoder Tremor written by the Xiph.Org foundation has been successfully ported to the Motorola DSP56002. It can be run in a low accuracy mode where it generates the same output with the same input as it does on, for example, a Sun workstation.

With this work it has been shown that it is possible to decode a Vorbis file with a Motorola DSP56002, even though it is not suited for this task at all. Due to the memory restrictions of the DSP it can only decode very small files, around a second in length at most, which were encoded at a very low quality setting. Due to the limited execution speed it is impossible to decode any files in real time.

The performance could be boosted a great deal with extensive optimization, possibly enough to decode larger files of better quality. Perhaps even enough to decode files encoded with the lowest quality setting in real time.
REFERENCES


http://www.mibsoftware.com/userkt/inn/dev/inn2.0-beta/inn/lib/alloca.c

[13] Erik Montémery, Johannes Sandvall, *Ogg/Vorbis in embedded systems*
Lunds Tekniska Högskola, 2004

[14] Christopher Montgomery, 64 bit math contribution to the Tremor mailing list,
# Appendix A

## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAC</td>
<td>Advanced Audio Coding</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>ARM</td>
<td>Advanced RISC Machine</td>
</tr>
<tr>
<td>ATH</td>
<td>Absolute Threshold of Hearing</td>
</tr>
<tr>
<td>BSD</td>
<td>Berkeley Software Distribution</td>
</tr>
<tr>
<td>CODEC</td>
<td>Coder-Decoder</td>
</tr>
<tr>
<td>COFF</td>
<td>Common Object File Format</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor / Digital Signal Processing</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FLAC</td>
<td>Free Lossless Audio Codec</td>
</tr>
<tr>
<td>GNU</td>
<td>Gnu's Not Unix</td>
</tr>
<tr>
<td>GPL</td>
<td>General Public License</td>
</tr>
<tr>
<td>IIR</td>
<td>Infinite Impulse Response</td>
</tr>
<tr>
<td>IMDCT</td>
<td>Inverse Modified Discrete Cosine Transformation</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>LSF</td>
<td>Line Spectral Frequency</td>
</tr>
<tr>
<td>LSP</td>
<td>Line Spectral Pair</td>
</tr>
<tr>
<td>MDCT</td>
<td>Modified Discrete Cosine Transformation</td>
</tr>
<tr>
<td>MIPS</td>
<td>Million Instructions Per Second</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>MP3</td>
<td>MPEG-1 audio layer 3</td>
</tr>
<tr>
<td>MPEG</td>
<td>Moving Pictures Expert Group</td>
</tr>
<tr>
<td>PAC</td>
<td>Perceptual Audio Coding</td>
</tr>
<tr>
<td>PCM</td>
<td>Pulse-Code Modulation</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SPL</td>
<td>Sound Pressure Level</td>
</tr>
<tr>
<td>VQ</td>
<td>Vector Quantization</td>
</tr>
<tr>
<td>WMA</td>
<td>Windows Media Audio</td>
</tr>
</tbody>
</table>
TREMOR LICENSE

Copyright (c) 2002, Xiph.org Foundation

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of the Xiph.org Foundation nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
BUGS FOUND IN G56K

This appendix presents the bugs found in the compiler for DSP65002. It also presents short code examples where the bugs are shown and workarounds to avoid the bugs are given.

In the code examples given it is assumed that `<stdio.h>` has been included and that the following macro has been defined:

```c
#define dprint(expr) fprintf(stdout, #expr " = 0x%i\n", expr)
```

FUNCTIONS RETURNING LONG

If a function that returns a `long` is added with an `int` there will be an incorrect result. Since the return value is `long` both the upper and the lower part of the a register is used to store it. The `int` is stored in register b, but only in its upper part since it only needs one register. When these two are added only the upper part of both registers will be used instead of adding the upper half of the b register with the whole a register a. See table C.1 for a code example.

SHIFTING LONG TO INT

When a `long` variable first is shifted and added to an `int` which is then stored as an `int` the same thing as in the previous case happens. The solution to this is to temporary store the result of the shift in a `volatile int`. If `volatile` is not used the compiler will optimize the workaround code so it becomes the same code as it was before and thus rendering it useless. See table C.2 for a code example.
SHIFTING INT TO LONG

When left shifting an `int` it is not automatically typecasted to a `long`. This means that any `int` shifted 24 bits to the left will give the result 0. One solution is to typecast the `int` that is being shifted to a `long` before doing the shift. See table C.3 for a code example.

Table C.1: Example of function returning a long.

<table>
<thead>
<tr>
<th>Code</th>
<th>Bug</th>
<th>Workaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>long one(){</td>
<td></td>
<td>long one(){</td>
</tr>
<tr>
<td>return 1;</td>
<td></td>
<td>return 1;</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td>int main(){</td>
</tr>
<tr>
<td>int main(){</td>
<td></td>
<td>long temp = 1;</td>
</tr>
<tr>
<td>i = 1 + one();</td>
<td></td>
<td>int i = temp + one();</td>
</tr>
<tr>
<td>dprint( i );</td>
<td></td>
<td>dprint( i );</td>
</tr>
<tr>
<td>return 0;</td>
<td></td>
<td>return 0;</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

Output: i=0x1

Table C.2: Example of shifting a long to an int.

<table>
<thead>
<tr>
<th>Code</th>
<th>Bug</th>
<th>Workaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>int main(){</td>
<td></td>
<td>int main(){</td>
</tr>
<tr>
<td>long l = 0x7000000;</td>
<td></td>
<td>long l = 0x7000000;</td>
</tr>
<tr>
<td>int i = (l&gt;&gt;24)+1;</td>
<td></td>
<td>volatile int temp =</td>
</tr>
<tr>
<td>dprint( i );</td>
<td></td>
<td>(l&gt;&gt;24);</td>
</tr>
<tr>
<td>return 0;</td>
<td></td>
<td>int i = temp + 1;</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td>dprint( i );</td>
</tr>
</tbody>
</table>

Output: i=0x1
When compiling with g56k the preprocessor macro DSIZE is supposed to automatically have the value of the first unused memory address in the memory used by the compiler. If for example the program is being compiled has 15 words of static and global data DSIZE should have the value 16. This value is used to place the run time stack at the bottom of the free memory.

But for some unknown reason DSIZE is always 0 which means that the run time stack is placed at the bottom of the memory. This will cause the run time stack to over write the global and static data and cause very strange and hard to find bugs.

One way to solve this is to make a copy of the *crt.asm* file found in the *DSP\dsp\src\56000\y* folder and replace the DSIZE with a value that is known to be in the free memory space. To find out how much of the memory is used a map can be generated with the compiler command `-j "-m<mapname>"`. Then the modified *crt.asm* file has to be compiled by running *g56k -c *crt.asm*. Finally the modified and now compiled *crt.cln* file must be used when compiling the program by using the option `-crt *crt.cln`

<table>
<thead>
<tr>
<th>Code</th>
<th>Bug</th>
<th>Workaround</th>
</tr>
</thead>
</table>
| int main(){
  long l = 0x17<<24;
  if(l == 0x17000000);
  printf("correct");
  return 0;
} | int main(){
  long l = 0x17UL<<24;
  if( l == 0x17000000 )
  printf("correct");
  else
  printf("incorrect,";
  " l=%x", l );
  return 0;
} | incorrect, l=0 | correct |

### Table C.3: Example of shifting an int to a long.

**PREPROCESSOR MACRO DSIZE**

When compiling with g56k the preprocessor macro DSIZE is supposed to automatically have the value of the first unused memory address in the memory used by the compiler. If for example the program is being compiled has 15 words of static and global data DSIZE should have the value 16. This value is used to place the run time stack at the bottom of the free memory.

But for some unknown reason DSIZE is always 0 which means that the run time stack is placed at the bottom of the memory. This will cause the run time stack to over write the global and static data and cause very strange and hard to find bugs.

One way to solve this is to make a copy of the *crt.asm* file found in the *DSP\dsp\src\56000\y* folder and replace the DSIZE with a value that is known to be in the free memory space. To find out how much of the memory is used a map can be generated with the compiler command `-j "-m<mapname>"`. Then the modified *crt.asm* file has to be compiled by running *g56k -c *crt.asm*. Finally the modified and now compiled *crt.cln* file must be used when compiling the program by using the option `-crt *crt.cln`
NESTED STRUCTURES AND UNIONS

The compilation will fail when compiling with the -g flag if the code being compiled has a structure containing an unnamed union which contains a pointer to the same structure. The same will happen with the opposite, i.e. a union containing a structure containing a pointer to the union. This can be solved by giving the unnamed union (or structure) an arbitrary name. Remember that this only happens when compiling with the -g flag. See table C.4 for a code example.

Table C.4: Example of nested structures and unions.

<table>
<thead>
<tr>
<th>Bug</th>
<th>Workaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct s { union { struct s *next; } ptr; } s</td>
<td>typedef struct s { union name{ struct s *next; } ptr; } s;</td>
</tr>
</tbody>
</table>

FUNCTION CALLS

Another bug was found but what is causing it is a mystery. Some function calls (but not all) having an argument which is a function call has strange behaviour. The solution to this was simply to store the result of the function call from the argument in a variable and use that in the other function call. See table C.5 for a code example taken from Tremor.

Table C.5: Example of nested function calls.

<table>
<thead>
<tr>
<th>Bug</th>
<th>Workaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>ogg_int64_t llret = _get_next_page(vf,&amp;og, set64_int(CHUNKSIZE));</td>
<td>ogg_int64_t chunk = set64_int(CHUNKSIZE); ogg_int64_t llret = _get_next_page(vf,&amp;og, chunk);</td>
</tr>
</tbody>
</table>
In this example the value of the third argument is wrong inside `get_next_page()` if this is written without a temporary variable. All attempts to find the reason for this bug has failed since it only occurs in a few places and because no reconstruction of the bug with own code has been successful.
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommerciell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Niklas Barsk