Technical notes on using Analog Devices DSPs, processors and development tools
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

Implementing an Ogg Vorbis Decoder on SHARC® Processors

Contributed by Jeyanthi Jegadeesan and Kulin Seth *Rev 1 – June 10, 2008*

Introduction

This EE-Note discusses the real-time implementation of an Ogg-Vorbis decoder on SHARC® processors. The Ogg Vorbis decoder is implemented on the ADSP-21364 and ADSP-21369 SHARC processors, although the same concepts apply for all ADSP-2136x SHARC processors. The ADSP-21364 and ADSP-21369 EZ-KIT Lite® boards are used as the hardware platform, and the VisualDSP++® development tools are used for the software development of the application. The VisualDSP++ source code for the decoder application is provided with this EE-Note in the associated `.ZIP` file.

Throughout this document, details about the following items will be broadly addressed:

- Ogg Vorbis codecs
- Visual C++ as a cross-reference tool
- Architectural advantages of ADSP-2136x SHARC processors
- Decoder workflow
- Implementation on ADSP-2136x SHARC processors
- Features of VisualDSP++ tools
- Optimization
- MIPS calculation

Ogg Vorbis Codecs

Ogg is the Xiph.org foundation's container format that holds multimedia data. Vorbis is a fully open, patent-free, royalty-free audio compression format, which uses Ogg format to store its bit streams as files. In many respects, Vorbis is similar in function to the MPEG-1/2 layer 3 (MP3) format and the newer MPEG-4 (AAC) format. This codec was designed for mid-to-high quality (8-kHz to 48-kHz bandwidth, >16-bit, polyphonic) audio at variable bit rates from 16 to 128 Kbps/channel, so it is an ideal format for music. The Ogg transport bit stream is designed to provide framing, error protection and seeking structure for higher-level codec streams that consist of raw, unencapsulated data packets, such as the Vorbis audio codec or Tarkin video codec. Vorbis encodes short-time blocks of PCM data into raw packets of bit-packed data. These raw packets may be used directly by transport mechanisms that provide their own framing and packet-separation mechanisms (such as UDP datagrams). For stream-based storage (such as files) and transport (such as TCP streams or pipes), Vorbis uses the Ogg bit stream format to provide framing/sync, sync recapture after error, landmarks during seeking, and enough information to properly

separate data back into packets at the original packet boundaries without relying on decoding to find packet boundaries. Refer to Appendix A — Ogg Details for more Ogg details. For information about Vorbis packets, refer to Appendix B – Vorbis Details.

Real-Time Implementation

The real-time implementation of the decoder includes understanding the existing non-real-time open source code and modification of it to use the SHARC processor architectural features. The existing open source code reads the `.ogg` files from the hard disk of the PC and stores the decoded PCM samples to the output `.dat` file. Visual C++ tools are used as a cross-reference tool for understanding and verifying the open source code.

Once the code has been validated using the Visual C++ tools, the code is ported to a SHARC processor using the VisualDSP++ tools. Initially, the decoder application is validated using the file I/O operation. Later, the code is modified to read the input `.ogg` file from the parallel flash on the EZ-KIT Lite board. The decoded PCM samples are sent using the serial port to the DAC of the AD1835 codec on the EZ-KIT Lite board. The decoder is implemented in such a way that it utilizes the architectural features of the processor more efficiently. Once the decoder is implemented, it is optimized further to improve performance. The decoder application uses some VisualDSP++ features for optimization. The following sections discuss in detail the implementation of the decoder on SHARC processors.

Using Visual C++ as a Reference Platform

The open source code for the Ogg Vorbis decoder is available from Xiph.org. This EE-Note uses the Vorbis Library 1.1.2 (libvorbis) source code, which is a floating-point implementation. This code is tested using the Visual C++ tools. The open source code is implemented with generic platforms in mind. The Visual C++ source code is modified to run and be tested on the PC. This code is used as the reference for the code implemented in the VisualDSP++ development tools. This stage includes modifying some header files and defining the required macros for the Visual C++ project. The code is also modified to read the `.ogg` file from the PC's hard disk and to write the PCM data to a `.dat` file on the hard disk. The PCM data file is converted into a `.wav` file and is played using the standard player for testing.

Advantages of using the Visual C++ tools as a reference include:

- Validation of results when testing the same code on other platforms. After running the code successfully on Visual C++ tools, it is easier to compare the results, and solve the problems. (In this case, it is VisualDSP++ tools.)
- Visual C++ Debugger features, like stepping through code, helps the user to understand the program and data flow. The behavior of various functions and variables can be determined easily in the Visual C++ tools, which can be used as the reference.
- Unwanted code and functions that are not used by the decoder can be removed and tested easily.
- The primary level optimization which is not related to the processor architecture can be implemented in Visual C++ tools conveniently.
- Debugging and code execution is faster with the Visual C++ tools.
- Last but not least, Visual C++ tools provide confidence that the code works correctly and can be implemented on other platforms.

Architectural Advantages of SHARC Processors

ADSP-2136x SHARC processors are highly integrated, 32/40-bit, floating-point processors optimized for high-performance audio processing. The processor core runs at higher speed up to 400 MHz. The I/O processor runs in the background without interrupting the core processor. Internal memory is divided in to 4 blocks in such a way that the processor core and I/O processor can access internal memory at the same time. The following architectural advantages of the processor are used for the algorithm development:

High Speed

The processor core runs at up to 400 MHz to achieve maximum performance.

I/O Processor

The I/O processor is used to get the input data from the flash memory and to send the decoded PCM data over the serial port. DMA is used for the input and output parts. The I/O processor, which runs in the background, fetches data from the external flash to the internal memory for the decoder to process and also sends the decoded data to the DAC via the serial port. This allows the core to run without interruption. The Vorbis decoder algorithm runs on the core, and the DMA controller takes care of the input and output data of the decoder in parallel.

On-chip Memory Blocks

The internal memory of the ADSP-2136x SHARC processor has 4 blocks. The core has the PM and DM data buses for core accesses, and the I/O processor has a dedicated bus for DMA accesses. The code is placed in the internal memory block 0, the data used by the core is placed in block 1, and the data used by the DMA controller is placed in block 2. Block 3 has some of the data used by the core and DMA controller. This effectively ensures that the core can simultaneously execute code from one block and access data from other block, and at the same time the I/O processor accesses the other blocks for DMA.

External Port on the ADSP-21369 Processor

The external port of the processor supports both the AMI and SDRAM interfaces. The AMI interface allows the processor to connect to asynchronous memory devices like flash and SRAM. The on-chip SDRAM controller allows the processor to interface with any SDRAM device without additional glue logic. The processor can interface up to a maximum of 254 MB SDRAM to 4 external banks or up to 62 MB asynchronous memory using the AMI interface. The SDRAM can be interfaced up to a maximum speed of 166 MHz, and the AMI devices can be interfaced up to the speed of 66 MHz. The DMA controller supports reading the data into the internal memory with packing enabled or disabled.

The decoder uses external port DMA with packing disabled. This allows the data to be read as 8-bit data into internal memory, and the decoder directly operates on this data. The AMI interface connects to the parallel flash on the EZ-KIT Lite board. The input `.ogg` file, which is programmed on the flash, is read using external port DMA. The SDRAM on the EZ-KIT Lite board is used for placing part of the data used by the decoder. Since the internal memory of the processor is not sufficient for the dynamic memory allocated by the decoder, the heap is placed on the external SDRAM.

Parallel Port on the ADSP-21364 Processor

The parallel port on the ADSP-21364 processor provides the interface to asynchronous 8-bit and 16-bit memory. The parallel port supports a 56-Mbytes per second transfer rate (`CCLK`/6) and 256-word page boundaries. The on-chip DMA controller automatically packs external data into the appropriate word

width during transfers. The parallel port supports packing of 32-bit words into 8- or 16-bit external memory and programmable external data access duration from 3 to 32 clock cycles.

Serial Ports (SPORTs)

SPORTs provide an inexpensive interface to a wide variety of digital and mixed-signal peripheral devices. The SPORTs can operate at up to one-eighth the core clock (`CCLK`) rate. They allow the processor to interface with codecs and high-speed data converters. On the EZ-KIT Lite board, the SPORT is connected to the DAC of the AD1835 codec. The decoded PCM data is transferred to the DAC over the SPORT. The output of the DAC, available on some of the headers, can be connected to speakers or a head phone.

Digital Application Interface (DAI)

The Signal Routing Unit (SRU) on the processor connects the peripherals to the pins or to each other. This allows the peripherals to be suited for a wide variety of systems. On the ADSP-21369 EZ-KIT Lite board, the SRU interface connects serial port signals to the DAC using DAI pins.

Development Tools

VisualDSP++ is the integrated software development platform used to develop applications on SHARC processors. It allows users to move easily between editing, building, and debugging activities. It also helps users to perform various levels of optimization to the generated code. The code profiling tool of VisualDSP++ helps users to identify the amount of time spent on a function.

The EZ-KIT Lite board is the hardware platform used for developing the application. The following ADSP-21364 and ADSP-21369 EZ-KIT Lite board's features are used for the real-time implementation.

Parallel Flash

The EZ-KIT Lite board has 1 MB of 8-bit parallel flash (AM29LV08IB) which is connected to the external port of the ADSP-21369 processor on the ADSP-21369 EZ-KIT Lite board and to the parallel port of the ADSP-21364 processor on the ADSP-21364 EZ-KIT Lite board. This parallel flash is used to store the compressed audio song in `.ogg` format. Since the flash on the EZ-KIT Lite board is 1 MB, the current implementations of the Ogg Vorbis decoder are tested only for the song size of 1 MB.

AD1835A Codec

The AD1835A is a high-performance, single-chip codec, featuring four stereo digital-to analog converters (DACs) for audio output and one stereo analog-to-digital converter (ADC) for audio input. The codec can input and output data at a sample rate of up to 96 kHz on all channels. A 192-kHz sample rate can be used with the one of the DAC channels. The serial port of the processor is interfaced with the AD1835A via the DAI port. The master input clock (`MCLK`) for the AD1835A can be generated by the on-board 12.288-MHz oscillator or can be supplied by one of the DAI pins of the processor. The AD1835A codec can be configured as a master or as a slave, depending on DIP switch settings. In master mode, the AD1835A drives the serial port clock and frame sync signals to the processor. The AD1835A audio codec's internal configuration registers are configured using the SPI port of the processor. The DACs of the codec were used for the output of the decoded song.

Figure 1 explains the block diagram of the Ogg Vorbis decoder application on the EZ-KIT Lite board.

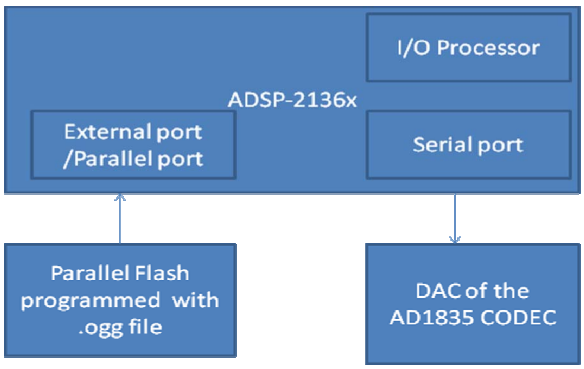

Figure 1. Overview of the Ogg Vorbis application

Dataflow of the Decoder

Vorbis I uses four types of packets in the compressed audio stream (three headers and one audio packet). Before decoding can begin, a decoder must initialize using the bit stream headers matching the stream to be decoded. Once set up, decoding may begin at any audio packet belonging to the Vorbis stream. In Vorbis I, all packets after the three initial headers are audio packets. The header packets are (in order) the identification header, the comments header, and the setup header. The decoder initially decodes the header information from the incoming stream and then starts decoding the audio packets. Figure 2 shows the generic algorithm for the Ogg Vorbis decoder workflow.

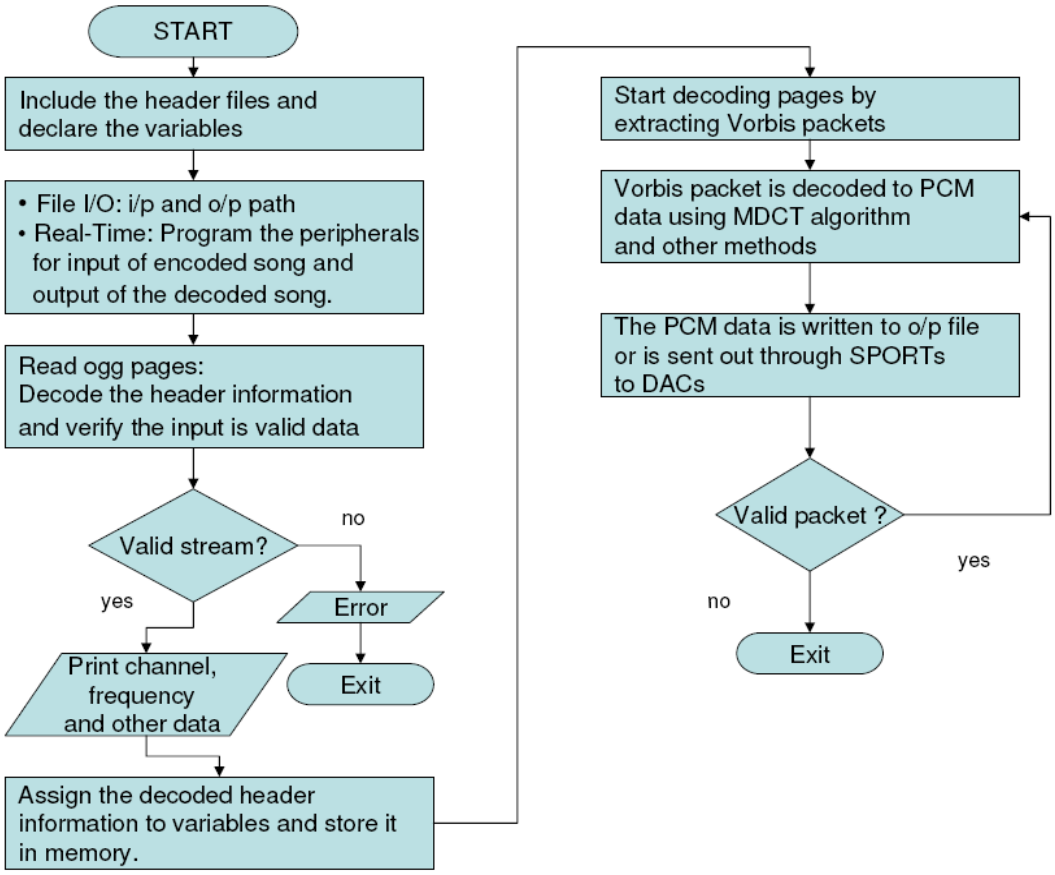

Figure 2. Flow of the Ogg Vorbis code

Implementation on SHARC Processors

VisualDSP++ File I/O

The Visual C++ code is directly ported to the VisualDSP++ tools to check for compatibility. The following modifications are added to make the code work with the VisualDSP++ tools:

- Since SHARC internal memory is 32-bit, the data types supported by the VisualDSP++ tools are 32-bit only. Hence, 8- and 16-bit data types used by the open source code are modified to 32-bit.
- The file I/O functions provided by the VisualDSP++ tools return the data as 32-bit words. The data read from the input `.ogg` file needs to be divided into four 8-bit words before processing since the decoder algorithm assumes that the data to be decoded is in 8-bit format. Similarly, the PCM data is combined to get a complete 32-bit data and written into the output file. The code snippet in Listing 1 shows how to unpack the data before passing it to the decoder.

```
for( temp = 0; temp < bytes; temp++ )
{
buffer[j++] = buffer1[temp] & 0x000000FF;
buffer[j++] = (buffer1[temp] & 0x0000FF00) >> 8;
buffer[j++] = (buffer1[temp] & 0x00FF0000) >> 16;
buffer[j++] = (buffer1[temp] & 0xFF000000) >> 24;
}
```

Listing 1. Data unpacking logic

- The Ogg Vorbis codec application is an ongoing project written from a code-scalable perspective. Some code portions were written so that they can be used in future upgrades. These redundant code portions were removed after we got hold of the data flow of the application. For example, there was code whose results were not used in any other part.
- The floating-point implementation of the Ogg Vorbis decoder requires the heap memory of about 2.8 Mbits. On ADSP-21369 processors, the heap is placed in external SDRAM since the internal memory is not sufficient for the source code and complete heap allocation. The PLL initialization function is added to the project to run the core at a higher frequency. SDRAM initialization is added to initialize the on-chip SDRAM controller.
- For ADSP-21364 processors, the internal memory is not sufficient for the source code and complete heap allocation. The external memory of the processor also cannot be used for placing some data directly. Byte packing logic is added to use the 32-bit memory of the processor efficiently. This was done by creating a data structure called `DATA_PTR` and defining a few functions for byte read/write functionality in word-addressed memory environments, basically a set of wrapper functions to obtain byte addressability in word-addressed memory (see Listing 2). This data structure contained 2 fields: one that points to the word address and the other is a modulus value that points to the corresponding byte in the word. This reduces the amount of memory required by the decoder.

```
typedef struct {
  unsigned int *data;
  int mod;
} DATA_PTR;
```

Listing 2. Data structure used for byte packing logic on an ADSP-21364 implementation

- For ADSP-21364 processors, with the byte packing logic, the internal memory was not sufficient for the heap memory needed by the decoder. Hence, the reference application was rewritten according to the decoder specification. It was observed that the major portion of the heap usage was due to the vector tables that were decoded during header decode. This problem was overcome by building the vector table when it was required during audio decode phase. This resulted in a reduction in the memory requirements (to about 0.7 Mbits of heap) of the code.

With these modifications the worked fine with the VisualDSP++ tools. Since the file I/O operation supported by the VisualDSP++ tools takes more cycles, the code takes a very long time to execute.

Modifications to the Real-Time Code

The following modifications are added to the file I/O code to make it real-time:

Input/Output

The input part is modified to read the data from the parallel flash on the external port/parallel port. The output part is modified to send the data to the DAC of the AD1835A codec over a serial port. The SRU initialization code is added to route the serial port signals to the DAC of the AD1835 and to route the `MCLK` for the AD1835 codec on the EZ-KIT Lite board.

DMA Controller

The DMA controller is used for reading data from the parallel flash and sending it to the DAC over a serial port. This allows the core to operate only on the decoder algorithm. External port DMA is used on ADSP-21369 processors and parallel port DMA is used on ADSP-21364 processors to read the data from the parallel flash. DMA chaining is used for the serial port on the output end for continuous data transfer.

Memory Management on ADSP-21369 Processors

ADSP-2136x processors have four blocks of internal memory. The ADSP-21369 processor has the overall internal memory of 2 Mbits of RAM and 6 Mbits of ROM. Block 0 and block 1 have 0.75 Mbits of SRAM and 3 Mbits of ROM. Block 2 and block 3 have 0.25 Mbits of SRAM. The memory requirement of the floating-point implementation of the Ogg Vorbis decoder is too high.

The Ogg stream has a variable page size which can be a maximum of 64 Kbytes. The Ogg Vorbis decoder open source code reads one complete page from the input file to the page buffer, which is dynamically allocated according to the page size of the incoming stream. Then, from the page buffer, one complete packet of data is extracted and the data is copied to the packet buffer. The Vorbis decoder operates on the packet buffer and decodes the data. This approach uses more memory and also includes core activity that copies data from the page buffer to packet buffer. Since the same data is used by the Vorbis decoder function, the library functions to extract the page and packet data are modified to use the input buffer directly. Some Ogg library functions are rewritten to reduce code size. This reduces the amount of memory used for the input buffer and also reduces the core activity used for copying from one buffer to other buffers by the Ogg library functions.

The open source code example processes the data as 8-bit data across the application. Since VisualDSP++ tools support only 32-bit data types, the data is initially read from parallel flash without enabling packing. This avoided the unpacking logic needed in the file I/O code but added more memory requirement for the input buffer. This way the minimum input buffer size needed was 3072 words which increased the internal memory requirement. Hence, external port DMA is used with packing enabled. The 32-bit data in the input buffer is accessed byte-wise using specific functions. Decoder functions were modified to use this

approach. This allowed part of the internal memory to be used for heap, which speeds up algorithm execution. Multiple heaps were used in the decoder. One heap is placed in internal memory, and the other heap is placed in external memory. The internal memory heap is used for placing the decoder output data and some of the codebook tables used by the decoder. The external memory heap is used for the rest of the dynamic memory used by the application.

```
unsigned char Get_Byte_Data(unsigned char *buffer_ptr,int position)
{
    if( position == 0 )
         return (*buffer_ptr) & 0x000000FF;
    else if( position == 1 )
         return ((*buffer_ptr) & 0x0000FF00) >> 8;
    else if( position == 2 )
         return ((*buffer_ptr) & 0x00FF0000) >> 16;
    else
         return ((*buffer_ptr) & 0xFF000000) >> 24;
}
```

Listing 3. Function used for accessing the byte data from a word

In the real-time decoder application, the code and the data used by the decoder are placed in internal memory block 0. The look-up tables used by the decoder are placed in internal memory block 2. The buffers used by the DMA controller and the stack memory are placed in internal memory block 3. One of the heaps is placed in internal memory block 1, and the other heap is placed in external SDRAM. The external SDRAM interface is run at a speed of 133 MHz.

```
MEMORY
{
  seg_rth  { TYPE(PM RAM) START(0x00090000) END(0x000900ff) WIDTH(48) }

  seg_init { TYPE(PM RAM) START(0x00090100) END(0x0009011f) WIDTH(48) }
  seg_int_code { TYPE(PM RAM) START(0x00090120) END(0x0009012F) WIDTH(48) }
  seg_pmco { TYPE(PM RAM) START(0x00090130) END(0x00093bff) WIDTH(48) }
  seg_dmda { TYPE(PM RAM) START(0x0009da00) END(0x0009dFFF) WIDTH(32) }

  seg_pmda { TYPE(PM RAM) START(0x000C0000) END(0x000c1FFF) WIDTH(32) }

  seg_heaq { TYPE(DM RAM) START(0x000b8000) END(0x000bdfff) WIDTH(32) }

  seg_dmda1{ TYPE(DM RAM) START(0x000e0000) END(0x000e0FFF) WIDTH(32) }
  seg_stak { TYPE(DM RAM) START(0x000e1000) END(0x000e1FFF) WIDTH(32) }

  seg_heap { TYPE(DM RAM) START(0x08000000) END(0x08FEFFFF) WIDTH(32) }
  seg_sram { TYPE(DM RAM) START(0x00200000) END(0x0027FFFF) WIDTH(8) }
}
```

Listing 4. .LDF file for the decoder on ADSP-21369 processors

Memory Management on ADSP-21364 Processors

The ADSP-21364 processor has the overall internal memory of 4 Mbits of RAM and 4 Mbits of ROM. Block 0 and block 1 have 1 Mbit of SRAM and 2 Mbits of ROM. Block 2 and block 3 have 0.5 Mbits of SRAM. The code is placed in block 0. The data used by the DMA controller and core are placed in block 3. Block 1 is used for heap memory, and block 2 is used for stack memory space.

```
MEMORY
{
  seg_rth {TYPE(PM RAM) START(0x00090000) END(0x000900ff) WIDTH(48)}
  seg_init {TYPE(PM RAM) START(0x00090100) END(0x000901ff) WIDTH(48)}
  seg_int_code {TYPE(PM RAM) START(0x00090200) END(0x000902ef) WIDTH(48)}
  seg_pmco {TYPE(PM RAM) START(0x000902f0) END(0x00095554) WIDTH(48)}
  seg_heap {TYPE(DM RAM) START(0x000b8000) END(0x000bffff) WIDTH(32)}
  seg_stak {TYPE(DM RAM) START(0x000e17bf) END(0x000e3fff) WIDTH(32)}
  seg_dmda {TYPE(DM RAM) START(0xc0000) END ( 0xc3fff )WIDTH(32)}
  seg_sram {TYPE(DMAONLY DM) START(0x01200000) LENGTH(0x3fff) WIDTH(8)}
}
```

Listing 5. .LDF file for the decoder on ADSP-21364 processors

Buffer Management

In the file I/O code, the data is read sequentially from the input file, and the decoded data is also written sequentially. But for the real-time case, when the input buffer is processed by the decoder, the decoded data in the output buffer is sent out in parallel. The buffer management must ensure that the same data is not processed twice and that the data is not overwritten before the decoder processes it. The real-time implementation uses the ping-pong buffers for this purpose. Figure 3 explains the ping-pong buffering used by the decoder.

The Ogg Vorbis decoder implementation code uses two buffers in the input part. On ADSP-21369 processors, initially both input buffers are filled with the data from the flash. The decoder starts processing the data on input buffer 1. Once input buffer 1 is processed, the decoder starts processing the data from input buffer 2 and the DMA controller fills input buffer 1 with next set of data in the background. When input buffer 1 is processed next, the DMA controller starts filling input buffer 2 in parallel. Before starting the next DMA, the processor core ensures that the previous DMA is completed. If the DMA transfer is not completed, the core waits for the current DMA transfer to finish before the next one is started. This is explained with the code of Listing 6.

```
if( read_ptr >= PAGE_SIZE_IN )
{
    read_ptr = read_ptr - PAGE_SIZE_IN;
    data_process_count++;

    if( page_ptr < databuffer2 )
    {
         while (data_ready == 0);
              data_ready = 0;
         FillBufferFromFlash(2);
    }
    else
    {
         while (data_ready == 0);
              data_ready = 0;
         FillBufferFromFlash(1);
    }
}
```

Listing 6. Code logic for reading from flash

External port or parallel port DMA is not continuous, since the next set of data can be read only when the previous data read is processed.

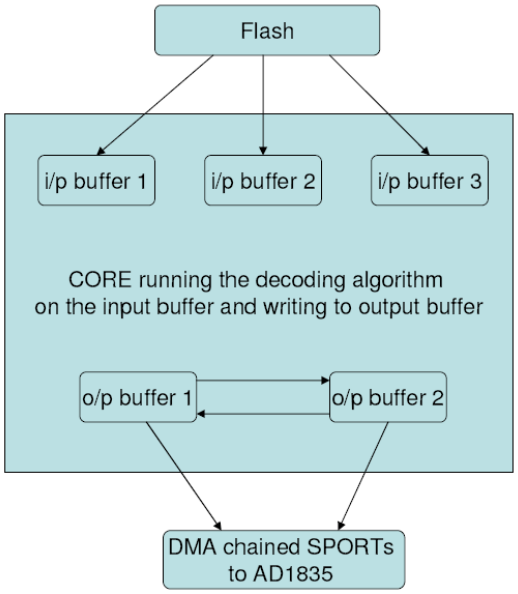

Figure 3. Ping-pong buffer mechanism

The output part uses two ping-pong buffers with DMA chaining. The decoder starts filling the decoded data on output buffer 1. Once output buffer 1 is filled, serial port DMA is started to send the data out. In the meantime, the decoder starts filling output buffer 2 in parallel. Once output buffer 2 is filled, the decoder ensures that the previous DMA transfer is over before overwriting output buffer 1. This is done by checking the flag which is set inside the serial port interrupt service routine. If the DMA is not over, it waits for the DMA to complete before writing output buffer 2 with the new set of PCM data. When output buffer 2 is transferred by the DMA, output buffer 1 is filled with decoded data in parallel.

```
if( output_filled >= (PAGE_SIZE_OUT) )
{
    output_filled = output_filled - (PAGE_SIZE_OUT);
    fill_count++;

    if( init_sport == 0 )
    {
        init_sport = 1;
        InitSPORT();
    }

    if( xmit_count < fill_count )
    {
         while( xmit_done == 0 )
              wait_count++;

         xmit_done = 0;
    }
}
```

Listing 7. Code logic to wait for the previous SPORT DMA completion

With these modifications, the code worked in real-time.

VisualDSP++ Features

LDF Programming

Linker Description File (`.ldf`) programming is used for efficient placement of the data and code to internal memory blocks. The default `.ldf` file is modified to allocate the memory to use the processor architecture effectively. When the application is built with the default `.ldf` file, the generated memory map provides the details of the used memory and free memory on each block. Based on the size of the code and data memory used by a `.doj` file, the code and data can be placed in the free memory blocks.

The VisualDSP++ Expert Linker utility has a GUI that allows you to drag and drop an `.obj` file from one memory block onto another memory block. This way you can view the memory blocks and place the code and data accordingly. The `.ldf` file can also be edited using the VisualDSP++ editor. The `.ldf` file is re-programmed so that the data is placed in multiple memory locations to use the dual-fetch feature of the core and to allow the core and DMA to access different memory blocks at the same time. The processor core also executes instructions in parallel. There are many source files in the application. Due to internal memory block size constraints, the sizes of the various memory sections (such as seg_init, seg_init_code, seg_pmco, and seg_dmda) are changed.

Figure 4 shows the ADSP-21369 decoder `.ldf` file viewed from the Expert Linker utility.

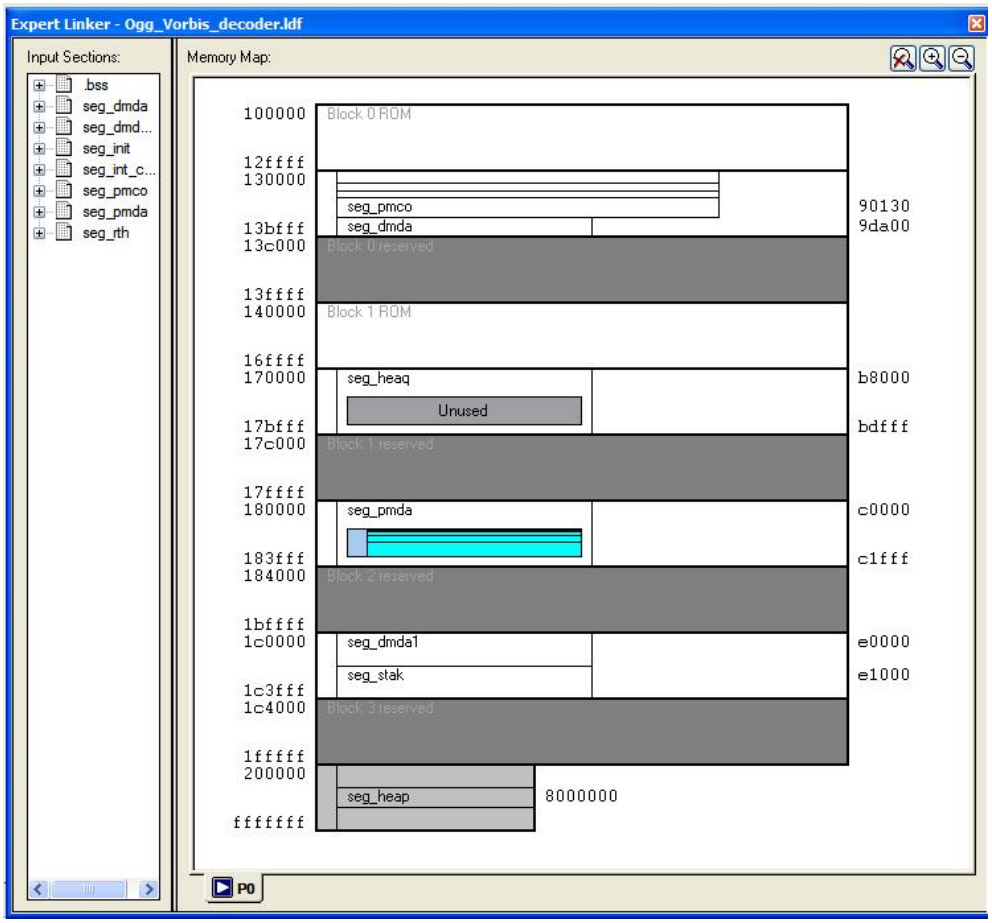

Figure 4. Memory organization of an .ldf file with multiple heaps

Multiple Heaps and Heap Management

The SHARC C/C++ run-time library supports the standard heap management functions `calloc`, `free`, `malloc`, and `realloc`. By default, these functions access the default heap, which is defined in the standard Linker Description File (`.ldf`) and the run-time header. We can define any number of additional heaps, which can be located in any SHARC processor memory block. These additional heaps can be accessed either by the standard `calloc`, `free`, `malloc`, and `realloc` functions, or via the Analog Devices extensions (`heap_calloc`, `heap_free`, `heap_malloc`, and `heap_realloc`). The primary use of alternate heaps is to allow dynamic memory allocation from more than one memory block. This feature is used by the ADSP-21369 decoder implementation.

Declaring a Heap

Each heap must be declared with a `.VAR` directive in the `seg_init.asm` file, and the `.ldf` file must declare memory and section placement for the heaps. The default `seg_init.asm` file declares one heap (`seg_heap`). To use a custom `seg_init.asm`, assemble it using the `easm21k.exe` application and use it to replace the default `seg_init.doj` in the `libc.dlb` archive using the VisualDSP++ `elfar.exe` application. Alternatively, add the `seg_init.asm` file to the project by copying it to the project folder.

Listing 4 provides a simple example to create a secondary heap. The following changes (Listing 8) have to be made to `seg_init.asm` (default path: `C:\Program Files\Analog Devices\VisualDSP 5.0\213xx\lib\src\libc_src`) to include a second heap called "`seg_heaq`".

```
.global ___lib_heaq_space;
.var    ___lib_heaq_space[5] =
    0x7365675F6865, // 'seg_he'
    0x6171FFFFFFFF, // 'aq'
    0,              // 0
    ldf_heaq_space,
    ldf_heaq_length;
___lib_heaq_space.end:
```

Listing 8. Multiple-heap declaration

The `.ldf` file is modified to incorporate the following changes for memory space allocation for second heap '`seg_heaq`'.

In memory section:

```
seg_heaq { TYPE(DM RAM) START(0x000b8000) END(0x000bdfff) WIDTH(32) }
```

In processor section:

```
/* section placement for additional custom heap */
heaq
{
// allocate a heap for the application
   ldf_heaq_space = .;
   ldf_heaq_length = MEMORY_SIZEOF(seg_heaq);
   ldf_heaq_end = ldf_heaq_space + ldf_heaq_length - 1;
} > seg_heaq
```

Listing 9. Programming an .ldf file for multiple heaps

Using Alternate Heaps

Alternate heaps can be accessed by the standard functions (`calloc`, `free`, `malloc`, and `realloc`). The run-time library keeps track of a current heap, which initially is the default heap. The current heap can be changed any number of times at runtime by calling the `set_alloc_type` function with the new heap name as a parameter, or by calling `heap_switch` with the heap ID as a parameter.

In the current implementation, the default heap is accessed using the `malloc`, `calloc`, and `realloc` functions. The secondary heap is accessed using the `heap_malloc`, `heap_calloc`, and `heap_realloc` functions with the heapID value of 1. The secondary heap is used mainly for placing the decoded code books, vectors decoded by the floor 0 and mapping functions. The data used by the decoder function is also placed in this heap.

Flash Programming Utility

The flash programmer utility available with the VisualDSP++ tools cannot be used to program the `.ogg` file on the flash. But the flash programmer driver files can be used for this purpose. For the ADSP-21364 processor, the flash programmer driver uses parallel port DMA to send the data to the flash. For the ADSP-21369 processor, the flash programmer driver uses external port DMA for the same purpose. The `adi_am29lv081b.c` file available with the VisualDSP++ tools has the driver functions for erasing the flash, reading the flash, and programming the flash.

The AMD Flash Programmer source code included in the `.ZIP` file associated with this EE-Note uses the driver functions provided by the VisualDSP++ tools. This application uses the following flash programmer driver functions:

- Allocate `AFP_Buffer` using `AllocateAFPBuffer()`
- Get sector map using `GetSectorMap()`
- Set up the device so the DSP can access it using `SetupForFlash()`
- Get flash manufacturer & device codes, title & description using `GetFlashInfo()`
- Erase flash
- Reset flash
- WriteData

The application works as follows:

- Initialize the processor to access the flash
- Erases the flash
- Opens the `.ogg` file using file I/O operation
- Reads 4 kBytes of data from the `.ogg` file into a buffer. The 32-bit word read from the file is split into 8-bit words and copied into another buffer.
- This data is programmed in the flash using `WriteData()` driver function
- The above two steps are repeated up to the size of the flash (1 MB)

Code Optimization

The code is optimized further in order to improve performance. Code profiling was implemented to understand how different parts of the code consume the processor computing speed (core clock cycles). This percentage break-up of the processing time (MIPS) indicates which functions are to be optimized for better performance. After identifying the functions, they were worked upon and optimization techniques were applied.

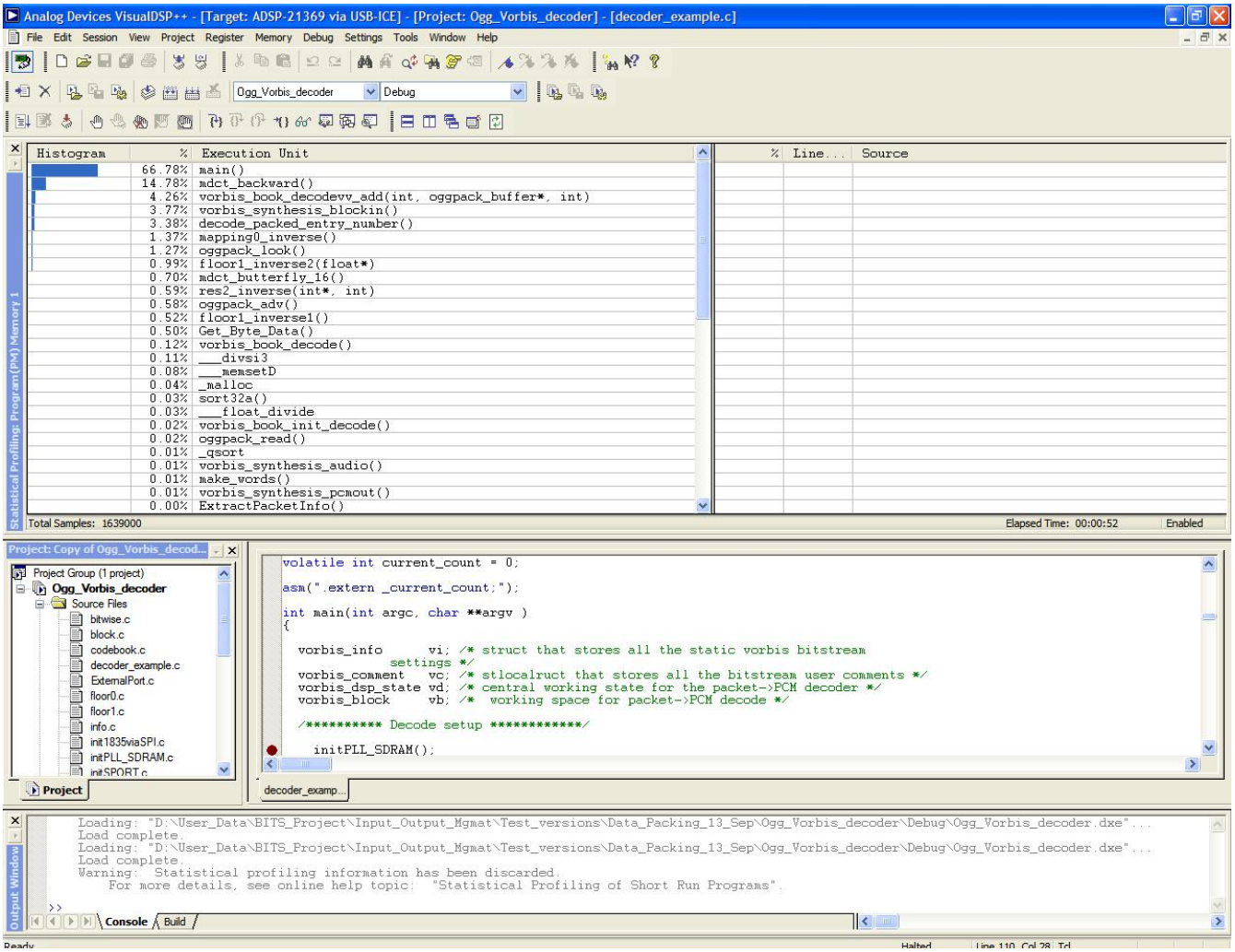

Figure 5. A screen capture of code-profiling

Figure 5 provides the profile created for the decoder application. It shows the percentage distribution of the core time among all the executed functions.

After using the VisualDSP++ profiler to identify the part of the code that takes more time, the following optimization techniques are used in the application:

Built-in optimization

The VisualDSP++ compiler has an internal built-in optimizer. This can optimize the code for speed or size. The optimization mode for generating the code for high speed is used. This helped reduce the processor MIPS consumption.

Parallel data fetch

The code is modified to add the `PM` keyword to enable the parallel data fetch operation. Whenever the compiler sees this keyword, it generates code to use the dual data access instructions. The memory management ensures that both data arrays accessed by the code are placed in two different internal memory banks.

SIMD mode

The project settings are modified to allow the compiler to enable SIMD mode whenever possible. This way the compiler generates code that uses SIMD mode. When SIMD mode is enabled, the number of times taken by a loop is reduced by one half.

Using built-in functions

The VisualDSP++ compiler supports intrinsic (built-in) functions that enable efficient use of hardware resources. These functions provide a means to use the processor's hardware efficiently. In the Ogg Vorbis decoder, the `circ_buf` and `circ_ptr` built-in functions are used. This instructs the compiler to use circular buffer indexing for these buffers. Circular buffer indexing is used by the decoder to access the input and output ping-pong DMA buffers.

Inline functions

The C run-time manager must save/restore context information across function calls. The context information is pushed onto the stack while calling a new function and is popped from the stack when returning from the function call. If frequent function calls are made to a relatively small function, large overheads are required. These overheads can be eliminated by replacing such function calls with inline code.

The VisualDSP++ 5.0 compiler also provides built-in versions of some C library functions. The compiler immediately recognizes them and replaces them with inline assembly code instead of a function call. Inline assembly code is faster than an average library routine, and it does not incur the calling overhead.

MIPS Calculation

The MIPS calculation was done taking into account the following features:

- The input and output buffer sizes
- The data transfer rate from the parallel port to the input buffers
- The time (core clock cycles) it takes for the core to apply the decoding algorithm on the input buffer and write the result into output buffer
- The output data transfer rate is fixed as the DAC is programmed with frame-sync frequency of 48 kHz. The buffer management is implemented to meet this criterion so as to make the decoding algorithm realtime.

For the ADSP-21364 implementation, on the input side, two buffers of 1024 bytes were chosen into which the data from the flash is to be written. Here the parallel port DMA speed is 55 MBytes/sec for `PPDUR` = 3 and a core clock speed = 333 MHz. So on the input side, there is no timing constraint. Here the necessary condition is:

Input data transfer rate + Core processing cycles < Time to transfer output buffer

On the output side, the timing requirement is that the output buffer should be written before it is sent out. With a 48-kHz frame sync, it takes around 11 ms for the SPORT to send out a data buffer of 1024 bytes. So the next output buffer should be written within this time limit. To meet this criterion, core clock cycles were calculated for the processor to write 1024 bytes of output data, and it was found to be well within this range. The current ADSP-21364 implementation takes approximately 65 MIPS.

For the ADSP-21369 implementation, the input side has two buffers of 1024 bytes. The output side also uses the buffers of 1024 bytes. With a 48-kHz frame sync, it takes around 11 ms for the SPORT to send out a data buffer of 1024 bytes. It is verified that the time taken by the input DMA and the decoder core operation is less than the time taken by the SPORT. The current ADSP-21369 implementation takes approximately 100 MIPS.

The `EMUCLK` method was used to calculate the MIPS. The number of core clock cycles taken by the decoder is calculated by reading the `EMUCLK` register for different number of samples.

```
asm("r15=EMUCLK;");
```

Based on this number, the MIPS consumed by the processor can be calculated.

Summary

Ogg Vorbis is a highly competitive codec and due to its extremely flexible nature, it is on the verge of becoming one of the leading audio codecs in the near future. This EE-Note implements this codec on the SHARC architecture.

The following areas were explored:

- Porting and real-time implementation of the Vorbis decoder was achieved using the SHARC architectural features and VisualDSP++ tools
- The codec used DSP concepts such as MDCT and vector quantization, and the implementation of this codec is provided as a practical application of DSP used to develop embedded systems. This provides a complete system-level demonstration, showcasing processor capabilities.
- Interfacing the DSP with other peripherals like parallel flash and AD1835 codec was achieved. The song programmed in the flash was read at varying speeds.

The source code provided with the EE-Note demonstrates the real-time implementation of the Ogg Vorbis decoder on SHARC processors. This is not completely optimized code; it can be further optimized to reduce the MIPS taken.

Software Code

The source code for the Ogg Vorbis decoder on ADSP-21364 and ADSP-21369 processors is provided in the `.ZIP` file associated with this EE-Note. The AMD flash programmer code, which is used for programming the flash on the EZ-KIT Lite boards, is also provided. The source code has been tested using VisualDSP++ release 5.0.

Appendix A — Ogg Details

The Ogg transport bitstream is designed to provide framing (logical bitstreams), error protection and seeking structure for higher-level codec streams that consist of raw, un-encapsulated data packets, such as the Vorbis audio codec or Theora video codec. It does not know any specifics about the codec data that it encapsulates and is thus independent of any media codec.

Design Constraints

- Streaming capability (no seeking is needed to build a 100% complete bitstream)
- Small overhead (using no more than approximately 1-2% of bitstream bandwidth for packet boundary marking, high-level framing, sync, and seeking)
- Simplicity to enable fast parsing and concatenation mechanism of several physical bitstreams

Logical and Physical Bitstream

Raw packets are grouped and encoded into contiguous pages of structured bitstream data called *logical bitstreams*. A logical bitstream consists of pages, in order, belonging to a single codec instance. Each page is a self-contained entity (although it is possible that a packet may be split and encoded across one or more pages); that is, the page decode mechanism is designed to recognize, verify, and handle single pages at a time from the overall bitstream.

Multiple logical bitstreams can be combined (with restrictions) into a single *physical bitstream*. A physical bitstream consists of multiple logical bitstreams multiplexed at the page level and may include a 'meta-header' at the beginning of the multiplexed logical stream that serves as identification magic. The decoder reconstructs the original logical bitstreams from the physical bitstream by taking the pages in order from the physical bitstream and redirecting them into the appropriate logical decoding entity.

Bitstream Structure

An Ogg stream is structured by dividing incoming packets into segments of up to 255 bytes and then wrapping a group of contiguous packet segments into a variable-length page preceded by a page header. Both the header size and page size are variable; the page header contains sizing information and checksum data to determine header/page size and data integrity.

The bitstream is captured (or recaptured) by looking for the beginning of a page, specifically the capture pattern. Once the capture pattern is found, the decoder verifies page sync and integrity by computing and comparing the checksum. At that point, the decoder can extract the packets themselves.

Packet Segmentation

Packets are logically divided into multiple segments before encoding into a page. The segmentation and fragmentation process is a logical one; it’s used to compute page header values and the original page data need not be disturbed, even when a packet spans page boundaries.

The raw packet is logically divided into 255-byte segments and a last fractional segment of less than 255 bytes. A packet size may well consist only of the trailing fractional segment, and a fractional segment may be zero length. These values, called *lacing values*, are then saved and placed into the header segment table.

Ogg Encapsulation

- The first Vorbis packet (the identification header), which uniquely identifies the stream as Vorbis audio, is placed alone in the first page of the logical Ogg stream. This results in a first Ogg page of exactly 58 bytes at the very beginning of the logical stream.
- This first page is marked “beginning of stream” in the page flags.
- The second and third Vorbis packets (comment and setup headers) may span one or more pages beginning on the second page of the logical stream. However many pages they span, the third header packet finishes the page on which it ends. The next (first audio) packet must begin on a fresh page

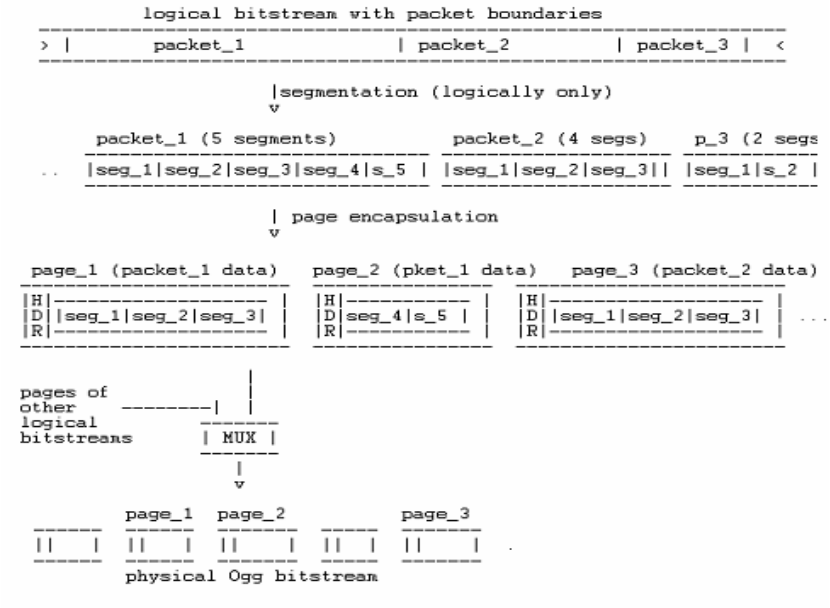

Figure 6. Ogg bitstream

Ogg Decoding

This decoding is based around the Ogg synchronization layer. We read data into the synchronization layer, submit the data to the stream, and output raw packets to the decoder. The bitstream is captured (or recaptured) by looking for the beginning of a page, specifically the capture pattern. Once the capture pattern is found, the decoder verifies page sync and integrity by computing and comparing the checksum. At that point, the decoder can extract the packets themselves. Decoding through the Ogg layer follows a specific logical sequence (Figure 7). The steps to be followed are as follows:

- Expose a buffer from the synchronization layer in order to read data.
- Read data into the buffer, using `fread()` or a similar function.
- Call a function to tell the synchronization layer how many bytes you wrote into the buffer.
- Write out the data by outputting a page from the synchronization layer.
- Submit the completed page to the streaming layer (using appropriate function).
- Output a packet of data to the codec-specific decoding engine.
- Ogg also must handle headers, incomplete or dropped pages, and other errors in input.

- The buffer exposure is performed by `ogg_sync_buffer()`.
- The data reading is done using `fread()` or a similar function.
- `ogg_sync_wrote()` tells the synchronization layer how many bytes you wrote into the buffer.
- `ogg_sync_pageout()` writes out the data by outputting a page from the synchronization layer.
- `ogg_stream_pagein()` submits the completed page to the streaming layer.
- `ogg_stream_packetout()` outputs a packet of data to the codec-specific decoding engine.

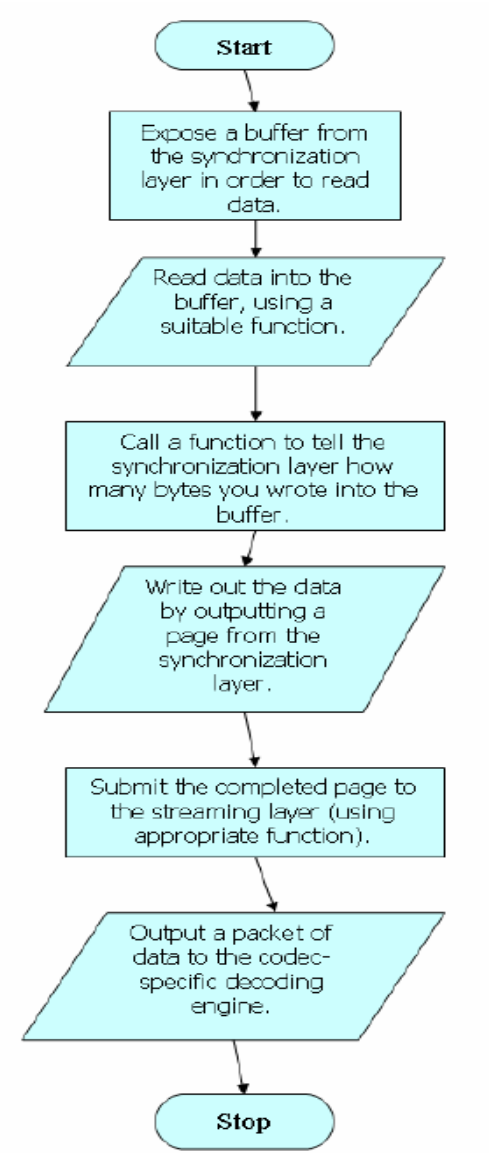

Figure 7. Ogg decoding

Appendix B – Vorbis Details

General Overview

Vorbis

- A general-purpose perceptual audio codec
- Allows maximum encoder flexibility
- Can be scaled over exceptionally wide range of bit rates both lower and high bit rates
- Is of the same league as MPEG-2 and MPC

Classification

- Vorbis I – A forward adaptive monolithic transform codec based on Modified Discrete Cosine Transform (MDCT)
- Vorbis II – A codec structured to allow addition hybrid wavelet filter banks to offer better transient response and reproduction using a transform better suited to localized time events

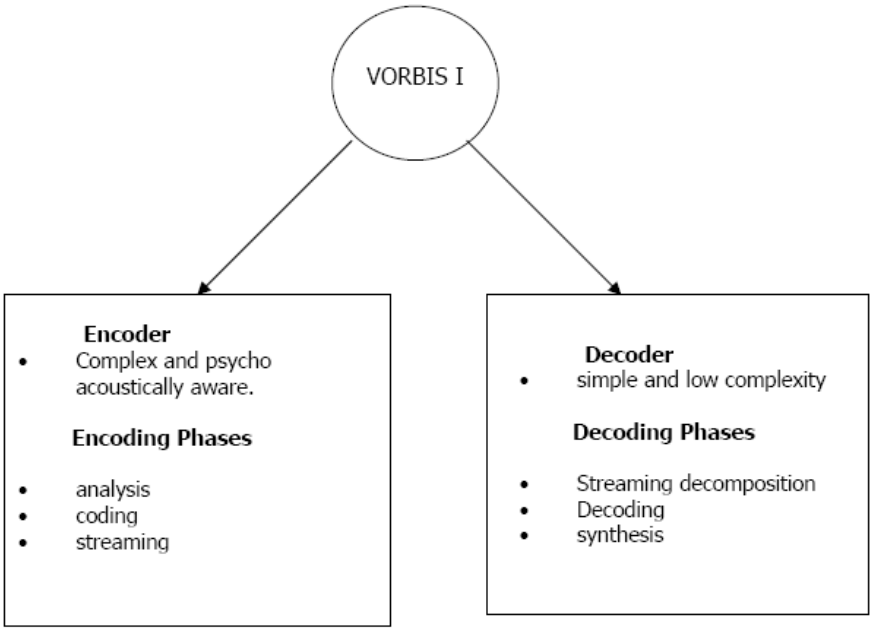

Figure 8. Vorbis overview

Vorbis Encoding Overview

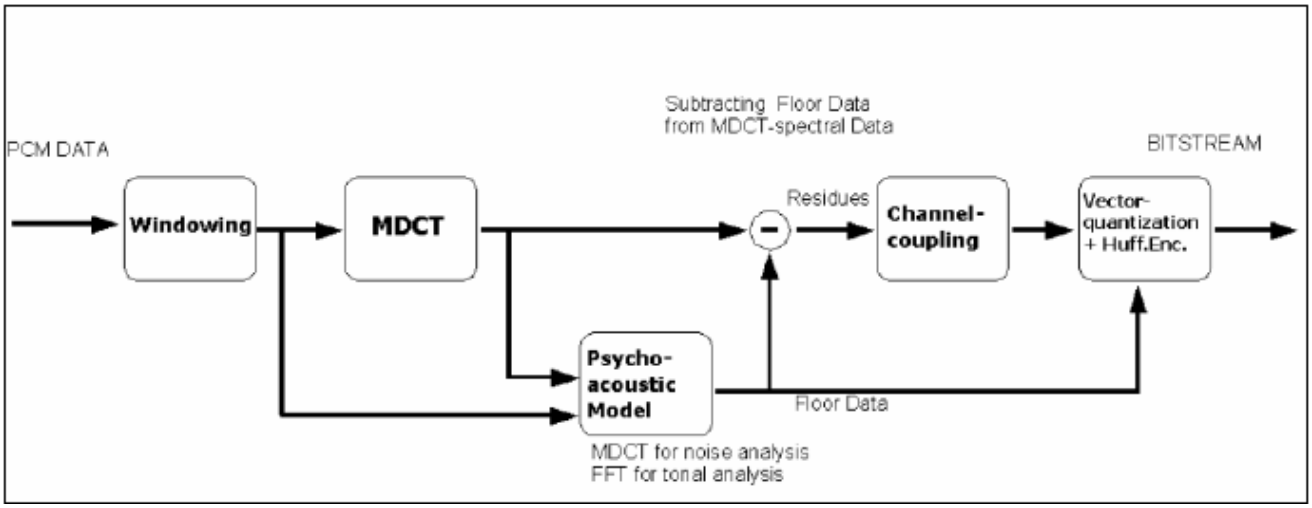

Figure 9. Vorbis encoder

The Vorbis encoder involves the following stages:

1. Analysis Stage
   - Block switching
   - MDCT
   - Psychoacoustic model
2. Coding
   - Floor generation
   - Channel coupling/ residue generation
   - Encoding (vector quantization)
3. 3) Streaming
   - Pack to Ogg stream

Vorbis Decoding

Decoder Setup

Before decoding can begin, a decoder must initialize using the bitstream headers matching the stream to be decoded. Vorbis uses three header packets; all are required, in-order, by this specification. Once set up, decode may begin at any audio packet belonging to the Vorbis stream. In Vorbis I, all packets after the three initial headers are audio packets. The header packets are, in the order, as follows:

- **Identification Header:** Used to identify the bitstream, Vorbis version, sample rate, and number of channels
- **Comment Header:** Includes user text comments (tags) and a vendor string for the application/library that produced the bit stream

- **Setup Header:** Includes extensive codec setup information as well as the complete VQ and Huffman codebooks needed for decode

Decode Procedure

Decoding and synthesis procedure for all audio packets is essentially the same and are shown in Figure 10. With the advantage of the symmetry of MDCT and store right-hand transform data of a partial 50% inter-frame buffer space savings, we can complete the transform later before overlap/add with the next frame.

Packet Decode Process

Vorbis I has 4 packets – first three types are header packets and are as described above, and the fourth type is an audio packet. All other types are marked as reserved and must be ignored.

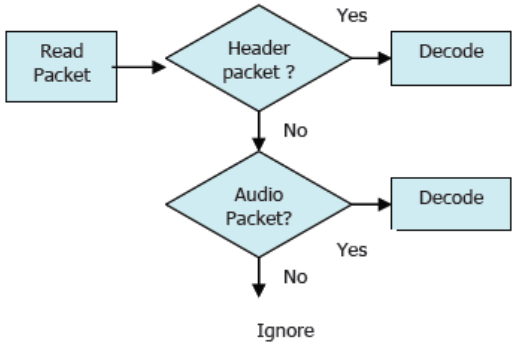

Figure 10. Vorbis packet decode process

Mode Decode

Vorbis allows multiple, numbered packet (modes).

Window Shape Decode (Long Windows Only)

Vorbis uses an overlapping transform, namely the MDCT, to blend one frame into the next, avoiding most inter-frame block boundary artifacts. The MDCT output of one frame is windowed according to MDCT requirements overlapped 50% with the output of the previous frame and added. The window shape assures seamless reconstruction.

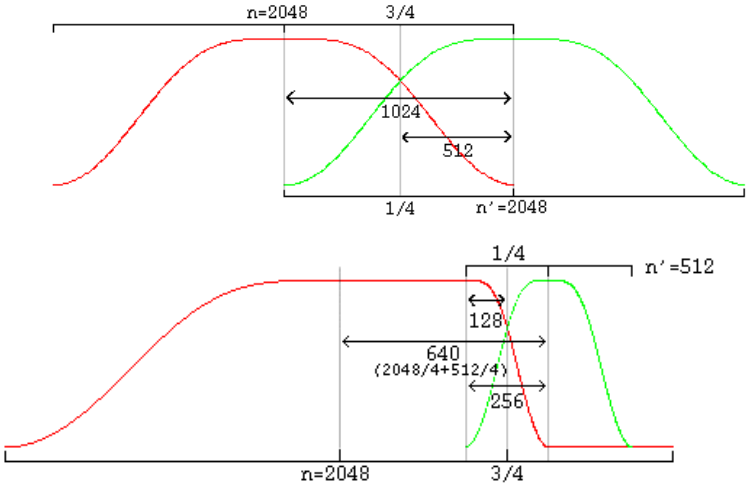

Figure 11. Overlapping windows

In un-equal sized overlap, window shape must be modified for seamless lapping. Vorbis also codes two flag bits to specify pre- and post-window shape.

Floor Decode

Each floor is encoded/decoded in channel order; however, each floor belongs to a sub-map that specifies which floor configuration to use. All floors are decoded before residue decode begins.

Residue Decode

Although the number of residue vectors equals the number of channels, channel coupling may mean that the raw residue vectors extracted during decode do not map directly to specific channels. When channel coupling is in use, some vectors will correspond to the coupled magnitude or angle. The coupling relationships are described in the codec setup and may differ from frame to frame, due to different mode numbers. Vorbis codes residue vectors in groups by sub-map; the coding is done in sub-map order from sub-map 0 to n-1. This differs from floors which are coded using a configuration provided by a sub-map number, but are coded individually in channel order.

Inverse Channel Coupling

Vorbis coupling applies to pairs of residue vectors at a time; decoupling is done in-place, a pair at a time in the order and using the vectors specified in the current mapping configuration. The decoupling operation is the same for all pairs, converting square polar representation (where one vector is magnitude, and the second is angle) back to Cartesian representation. After decoupling, in order, each pair of vectors on the coupling list, the resulting residue vectors represent the fine spectral detail of each output channel.

Compute Floor/Residue dot Product

The decoder multiplies the floor curve and residue vectors element by element, producing the finished audio spectrum of each channel.

Inverse Monolithic Transform (MDCT)

The audio spectrum is converted back into the time domain PCM audio via an inverse Modified Discrete Cosine Transform (MDCT). Note that the PCM produced directly from the MDCT is not yet finished audio; it must be lapped with surrounding frames using an appropriate window (such as the Vorbis window) before the MDCT can be considered orthogonal.

References

[1] *ADSP-2136x SHARC Processor Hardware Reference (includes ADSP-21362/3/4/5/6)*, Revision 1.0, October 2005, Analog Devices, Inc.

[2] *ADSP-21368 SHARC Processor Hardware Reference (includes ADSP-21367, ADSP-21369, ADSP-21371 and ADSP-21375)*, Revision 1.0, September 2006, Analog Devices, Inc.

[3] *VisualDSP++ 5.0 Linker and Utilities Manual*, Revision 3.0, August 2007, Analog Devices, Inc.

[4] *VisualDSP++5.0 C/C++ Compiler Manual for SHARC Processors*, Revision 1.0, August 2007, Analog Devices, Inc.

[5] *ADSP-21364 EZ-KIT Lite Evaluation System Manual*, Revision 3.2, July 2007, Analog Devices, Inc.

[6] *ADSP-21369 EZ-KIT Lite Manual*, Revision 2.1, August 2006, Analog Devices, Inc.

[7] *http://xiph.org/vorbis/ :* Ogg-Vorbis documentation, code, and other details.

[8] Final year Project Report at NITK (07-08).

Document History

Revision	Description
Rev 1 – June 10, 2008 *by Kulin Seth and Jeyanthi Jegadeesan*	Initial release