mindTunes
W4840 - Project Description

Jonathan Chen jtc2119@columbia.edu UNI:jtc2119
Po-Han Huang ph2252@columbia.edu UNI:ph2252
Michael Kempf mjk2154@columbia.edu UNI:mjk2154
Yen-Liang Tung yt2230@columbia.edu UNI:yt2230
Christos Vezyrtzis chris@cisl.columbia.edu UNI:cv2176

Abstract—This project involves the combination of software and hardware design techniques for the implementation of a voice recording system. The particular system will involve both the interface to record voice clips and to store them in **.wav** format in an external storage (USB flash drive). Playing existing voice clips will also be a feature of the designed system. In this report we give an analytic description of the followed procedure and of the system setup and organization.

1 Introduction

The designed system can serve as a complete voice recording system, capable of storing memos through a line/microphone input to a USB external storage device (flash-type). The entire project was built on the Altera DE2 FPGA Board. The operation of the ”mindTunes” system offers:

- 16-bit quality voice recording
- Storing of recorded voice clips to an external USB flash drive in a wave and mp3 format
- Browsing the contents of the external storing medium and playback of the existing voice clips

The process of voice recording can significantly lower the standards that the system needs to satisfy. The band-limited nature of the speech signals can offer the choice of a wave format storage, which can significantly lower the system's complexity, as well as the produced files' size, while maintaining the vioce-recording functionality.

The choise of an external USB flash drive will offer the possibility of a large storing space, in the **G**igabyte **B**yte (**GB**) region, which can extend the storing capacity to hundreds of hours of voice clips. Futhermore, the USB flash drive storage has a well known interface, which can be implemented through software.

The authors are with Columbia University, New York, NY. This project is associated with Embedded Systems Design CSEE4840. Theu authors would like to thank Professor Steven Edwards and their Teaching Assistant, David Lariviere, for their contribution and comments in this project.

2 Floorplan of the System

The block diagram of the system is illustrated in Figure 1. The nature of each entity (i.e. if its implementation is through hardware or software) is also mentioned Figure 1 and in Table 2. We also note the interactions

Required Block	Implementation
ADC/DAC	Hardware
USB Flash Drive and Interface	Hardware/Software
CPU and OS	Software
Memory Buffer	Hardware
Speaker, Microphone	Hardware

TABLE 1
Required Blocks for mindTunes

between these building blocks, which denote their way of cooperation for the operation of this system.

3 Environment of the System/Sampling and Wave Format

As mentioned previously, the key functionality of this system is the recording and reproduction of voice memos (signals). We, therefore, take advantage of the form of inputs to the system, to set limits to the system's standards. This process will prove beneficial in terms of complexity in designing the system.

More specifically, it is a well-known fact that voice signals are (with a very small approximation) band-limited to a region of 4kHz in the baseband. Moreover, speech signals can be well approximated as band-limited to an even smaller region (namely to a 3.3kHz region in the baseband), making small sacrifices in the speech quality, since small errors are not of primary concern in memo-recording applications. It is, therefore, dictated that a very small sampling rate, namely 8kHz, can very well serve as the sampling rate for such systems, since small degradations due to aliasing can be tolerated in voice-recording applications. Finally, by the same token (of some tolerance in the recorded signal quality), a 16-bit sampling scheme can offer satisfactory performance (we mention here that the aliasing error will be the limiting factor in such systems, since a 16-bit sampling can offer higher performances).

Reduction of both the sampling rate and of the quantization level scheme can make the resulting sampled signal be of a relatively low bitrate, thus enabling us to bypass any encoding scheme and use a very simple recording format, namely **wave**, since the recorded clips will not have the large size that would be caused through a non-compressed highly-oversampled signal sampling. Further (and more sophisticated) encoding of this file into mp3 format can be of low complexity and computational time, due to the reduced size of the wave file. This will also bring the encoding of the recorded clips closer to real-time.

4 Analytic Description of the System Building Blocks

We now proceed by analytically describing each building block and its function, along which some key parameters through which it influences the system's performance.

4.1 Audio Interface

In this voice recording and playback scheme, the ADC and DAC (namely **A**nalog to **D**igital **C**onverter and **D**igital to **A**nalog **C**onverter respectively) are the interfaces of the system's communication with the "outside world". They provide the conversion of the analog inputs to the system (voice) to a digital format ("word") so that they can be processed, as well as the back-conversion of the words to a clean (in terms of its noise performance) analog signal to be used for playback. The setup of these components must be accompanied by the existence of a few other building blocks, which ensure the proper data transfer to the storing/encoding structure. In this section we provide an analytic description and both qualitative and quantitative perspective of the hardware blocks; we must, however note that we mainly emphasize on the description of the input compomnents (ADC,S/P, input FIFO), the corresponding output structures are the symmetric structures of the former and their function can easily be derived and comprehended from the description of the input blocks.

The Altera DE2 board offers a top-level quality ADC and DAC (combined they are named as **"Codec"**), which can be used in numerous (software controlled) settings, depending on the application in hand. These settings include analog and digital pre-and-pro processing filters, along with the sampling rates and the analysis (number of bits) which the ADC and DAC encorporate. These parameters are involved in trade-offs that characterize the system's performance. As a simple example, we refer to the obvious trade-off between speed (and thus power) and audio quality (usually characterized by SQNR or SNR[1]), according to the number of used bits, more bits will lead to more use of resources and power, along with a degradation in speed (due to processing), but improvement in audio quality[2].

The use of the system for voice processing and storing (with a large tolerance in quality degradation and main concern toward storage) indicates the obvious choice for the Mixed Signal (ADC/DAC) blocks: they should be operated in minimum sampling rate and analysis, which is $8kHz$ and 16 bits respectively. A sampling rate of $8kHz$ is enough to handle a voice signal, which is band-limited to less than $4kHz$ (thus the Nyquist criterion is satisfied) and a 16-bit analysis is more than satisfactory for this purpose. This choice will be beneficiary (as mentioned before) in terms of occupied system resources, speed and power consumption and will allow the use of maximum resources in the encoding process (which is clearly more demanding). Furthermore, as is also obvious, the serializing and FIFO structures can also be of minimal space.

The hardware implementation of the audio interface involves the design and setup of the following:

- The Wolfson Audio Chip, operating in slave mode
- A serializer, which handles the serial-to parallel conversion of the data and the inverse structure
- A **F**irst **I**n **F**irst **O**ut (**FIFO**) structure, to act as a buffer structure

This procedure is shown in Figure 2.

The Wolfson Audio Chip contains (among others) the ADC and DAC structures. The proper setup will guarantee the correct sampling rate of the input signal (and playback of the stored clips), for which the chip needs to be provided with the appropriate clock signals. More specifically, for the operation of the audio inteface structures we need the $8kHz$ sampling clock ($16MHz$ if stereo inputs are sampled) for the ADC and DAC, along with the $16 \cdot 8MHz = 128kHz$ clock for the serializing and de-serializing operation (see next following comments). These clocks must be created as fractions of the existing audio clock (which is set by means of a crystal oscillator on the Altera DE2 board used for this project to $18.432MHz$ by means of a crystal type oscillator); more specifically we use a divide ratio of 2304 and 144 to create the above mentioned clocks.

The output of the ADC and the input of the DAC,

1. **SQNR** means **S**ignal to **Q**uantization and **N**oise **R**atio while **SNR** means **S**ignal to **N**oise **R**atio

2. The SNR or SQNR will approximately be $6.02n+1.76$ in dB, where n is the number of employed bits if no post-processing filter is used.

however, deal with a single-bit bit stream. This translates to the fact that the 16-bit samples (either of the input or of the stored clip to be played back) need to be converted to a 16-bit form and from it to a single-bit waveform respectively. This is done by means of a de-serializer and a serializer respectively. Moreover, this block must have comleted its function before the arrival of the next audio sample.

Consider the case of the input from the ADC (the DAC case can be easily considered easily if we explore the symmetry of the structure). Within a period of the ADC-sampling clock the de-serializer must convert the 1-bit produced by the ADC to a single 16-bit word, which will later be processed by the system. It is clear, therefore, that this block must be running at a clock 16 times (or in general to the bit acuracy of our sampling scheme) faster than the one at which the ADC samples. The placement of the bits in the ″word″ is done according to the wav (and mp3) specifications, which is little endian for this case. This function is illustrated in Figure 3.

The most important part of the audio interface (and the one with the more difficulties in the timing specifications - see section on problems) is the ″buffer″ structure, implemented as a FIFO structure. The need for such a structure arises because the internal processor of the system (through μcLinux) is operating at a speed of $100MHz$, which is significantly larger than the sampling rate. Without the presence of the FIFO structure, the above would dictate that the need for input samples would exceed the number provided by the ADC, for a fixed time interval. The solution for this is the implementation of a ″wait″ function, which holds the existing samples, feeding them to the processor in a serial way when they exist, while preventing the encoding/decoding operations from being executed in the absense of input samples (or in the opposite case for the DAC in the abundance of samples to be fed to the output).

The implementation of this was done be means of creating an Avalon component which implemented two FIFO structures, along with some combinational and sequential logic to control their process. While the key idea behind this process is trivial, this structure is prone to errors, mostly due to the tight timing specifications posed by the structures. More specifically, we need to perform the ″pause″ function (described in the previous paragraph), whenever we need to process samples and there are currently none present and when we need to play some samples and there is already an abundance of samples to be played back (two perfectly symmetric cases). We chose to use a FIFO operating in Legacy Mode (as described in the Altera documentations).

For the operation of the control logic, we implemented a Moore state machine. The underlying idea behind its function is the need to hold the processing system, by means of asserting the bus stalling signal ″waitrequest″, when the system is in the cases described above. Furthermore, we also need waitrequest to pause the function of the processor during the first cycle of a sample transfer, due to the delay posed by the FIFO structure before the valid data are presented to the bus. The associated timing diagram and state machine describing images are shown below. As shown in Figure, the key delays associated with this transfer are:

- The delay between the writing procedure in the FIFO and the moment that the corresponding signal declaring if the FIFO is empty (or nearly full) is asserted, which is zero clock cycles (with respect to the ″writing clock″)
- The delay from the assertion of the ″read″ signal of the FIFO to the point at which the valid data are presented at the output, whch is one clock cycle of the ″reading clock″. This is an effect of the choice of the function of the FIFO structure in ″Legacy Mode″, which is recommended by the providing plattform as the top-performance structure.

The state machine was designed so as to keep the bus on hold for the period when the action planned cannot be completed (i.e. the FIFO containing the samples fed from the ADC is empty when a read command is issued or when the FIFO that contains the samples to be fed to the DAC is full). The key issue in implementing this procedure is that it was done asynchronously (with respect to the processor clock), thus giving rise to a Moore state machine. Furthermore, even during the process of a succesful command (either read or write), the processor was paused by means of asserting the waitrequest signal until the data presented to the bus was valid. This gave rise to a two-processor-clock-period time interval at which waitrequest was to be held high, in order for a command to be completed with success and without any invalid data presented to the encoder. The principle of the state machine's function is indicated in Figure 7

The input FIFO is created out of Altera's build-in MegaFunction. Both the input and output are of 16-bit width. It provides status flags which can be used to decide the state of our Moore machine. The important status signals are rdempty, wrfull, and usedw. One clock cycle after rdreq has been asserted, one 16-bit data will be popped out of the FIFO and be presented on the output q at which time usedw decrements. Similarly, one clock cycle after wrreq has been asserted, one 16-bit data will be latched into the FIFO at which time usedw increments. Reading and writing to the FIFO in the same clock cycle is allowed, and usedw will not be updated.

Upon a system reset, it goes into state a0. In this state the Avalon bus is blocked from reading the FIFO. The state machine is allowed to traverse to the next state a1 provided that rdempty signal is deasserted and a read request has been issued. The state machine stays in state a1 for one clock cycle, with waitrequest and rdreq asserted, before checking the rdempty and read request signal from the Avalon bus. It will loop inside a1 if and only if the aforementioned conditions are true. If not, it goes to state a2, in which the bus is again blocked from reading the FIFO. One clock cycle after a2, it goes back to a0.

Because we maintain two independent, concurrently running state machines, the Moore machine on the output side is analogous to that of the input side. The system goes to state d0 upon a reset. Thereafter, it will only go to d1 provided wrfull signal is deasserted and a write request has been issued. The state machine stays in state d1 for one clock cycle, with waitrequest and wrreq asserted, before checking the wrfull and write request signal from the Avalon bus. It will loop inside d1 if and only if the aforementioned conditions are true. If not, it goes to state d2, in which the bus is again blocked from writing into the FIFO. One clock cycle after d2, it goes back to d0.

4.2 USB Flash Drive and Interface

An external Universal Serial Bus (USB) flash drive will be used as the storage medium for recorded voice clip files. The flash drive will also act as the file repository for voice clip playback. It will have a capacity in the GB region at low cost. The DE2 board provides both USB host and device interfaces using the Philips ISP1362 single-chip USB controller. Our flash drive will be mounted to the operating system as a USB host. Typically, the challenge of implementing a USB component is the requirement to design a device driver. Fortunately, uClinux provides a USB driver that we will utilize for writing and reading WAV/MP3 formatted voice clips. Once mounted, the USB provides an easy to use peripheral for software programming and can be treated as a path directory.

4.3 WAV Format

Waveform audio format (WAV) is a file format standard for storing audio data. This format stores data in ”chunks” with a 44 byte header that describes the sound information (sampling rate, sample size, etc.) that characterizes the audio file. Since our sound information was constant, we were able to prepare a header and write it to the file descriptor on the USB flash drive that was awaiting the input data from the ADC. Storing our data in WAV format allowed us to check the recording half of our project by providing us the capability of playing the audio sample through an audio application. During playback the 44 byte header can be safely skipped and the data chunk can be written to the audio base address. Also, the stored WAV file provided a ready to use file input to the MP3 encoder application discussed in the following section. The WAV header source code can be found in the main.c file.

4.4 MP3 Encoding

MP3 is commonly known as the MPEG 1 version Layer 3. However, MPEG 1 standard does not support our ADC or DAC sampling rate, which is 8000Hz in our design. Therefore, we choose to use the version MPEG 2.5, which is the extension version of MPEG 2 . The format and procedure to encode MPEG 2.5 layer 3 and MPEG 1 layer 3 are basically the same. The only difference is that version MPEG 2.5 support the lower sampling rate. The following table shows the sampling rate each version support.

The MP3 header is 32 bit long; the format is shown in Figures 10 11.

The difference between MP3 format files and WAV format file is that mp3 files are composed by successions of frames; each frame has its own header and small size data chunk. It is different from WAV files, which there is only one header for the entire file and a large size of data chunk. Therefore we can cut any part of the MP3 file and be able to decode it successfully.

The steps to encode mp3 data are not trivial but however the concept is actually simple. The concept behind mp3 encoding is that we take discrete cosine transform to a certain amount of audio samples in the time domain and project it to the frequency domain. We then masking out the frequency that human ear cannot easily perceive and only leave the most important frequency. The theory behinds this is that in Fourier analysis, the lowest few frequency coefficients can always well represent the original signal and contains the majority of the energy. Then we do Huffman coding to the result information in order to save the number of bitstream since it encodes the most popular symbols with least bits. And the decoding procedure is exactly the same steps but backwards.

The library we implement encoding on the NiosII CPU is modified based on the ShineFixed Point MP3 open source encoder version 1.09. The program was originally developed on the ARM’s RISC Operating System with inline assembly optimization. Thus in order to make the program to run on the Nios operation system, we remove the inline assembly language and substitute them with the proper C code. And we also undefined the RISC OS in the programs. We then port the program to the NIOS board by add it to be an uClinux user’s application. Later we integrate the program with the wav recorder together to be our main program.

4.5 MP3 Decoding

We choose MP3PLAY as our decoding library because it is built-in uClinux and supports various MPEG standards (MPEG-1,2,2.5 in Layer 1, 2, 3) which can match our encoder. In the MP3PLAY library, it can support both floating point and fixed point operation. There is also Assembly optimization but it conflicts to our platform, so we need to set the code not to turn on the assembly mode. By porting the original code to Nios2, we need to comment some functions because they are used for other devices.

The process of decoding MP3 file is to feed bitstream data into the decoder, unpack the frame , read header info to reconstruct and inverse mapping, as shown in Figure 12.

The bitstream unpacking block does error detection if error-check is applied in the encoder. Then, the bitstream data are unpacked to recover the various pieces of information. The reconstruction block reconstructs the quantized version of the set of mapped samples. The inverse mapping transforms these mapped samples back into uniform PCM.

With the existing MPEG decode library (in /mp3play/mpegdec_lib), we can decode 8K Hz 16 bits MP3 file into raw PCM data, and write it to ”na_audio” which is our FIFO address to buffer the data being decoded and pass to LINE OUT. There is also a function can show what the format of the file is. Because the output DAC is set to play stereo in 8K Hz sampling rate, if we want to test our file match is in 8K Hz format or not.

4.6 Operating System

The need to access (browsing through the files and storing) the USB flash drive data indicates the neccesity to use an operating system, to take advantage of the existing libraries to handle the file system needs. Furthermore (see previous section), we chose to use software to implement the MP3 coding, which again can be found as open-source code for an existing operating system.

A wide-used choice for such applications is clinux. Programming a uClinux OS to the board is a procedure which does is not resouce-hungry and enables the use of a wide variety of libraries for any use, thus wide programmability and function range. A careful design of the Operating System will save any such design of significant hardware complexity, with the single use of some (prefedined) resources (memory and gates).

As our programs are running as user applications on uClinux, it is important to note we are running our program in user space. This means we can not use interrupts or the NiosII Hardware Abstraction Layer that we utilized in the labs. To access peripherals we defined memory pointer access for read and write commands.

5 Problems During Implementation - Key Points and Remarks

The implementation of this project was not done in the absense of design issues and pitfalls; in this section we choose to mention the points which caused the largest obstacles toward the completion of this project. The functionality of the project was limited; we use this section to provide a complete list of all the problems encountered, as well as the possible causes and solutions to tackle them.

5.1 Hardware - Audio Interface

The key issue in implementing the hardware arose in the implementation of the buffer structure. The trivial way of operation becomes, when translated to a state machine and correspoinding logic structure, quite complicated, mostly due to the tight timing diagrams and the numerous cases that have to be examined. Among the different configurations available and the many clock and control signals, the choice of a set of control signals that could give rise to a fast and easy-to-comprehend method of creating the FIFO/Buffer structure was a non-trivial issue.

The large availability of FIFO structures made their vast majority prone to timing issues, due to the asunchronous way in which ”waitrequest” signal had to be set and the presence of two (very different) clocks that operated each FIFO structure. Despite the presence of an asynchronous (again with respect to the processor clock) control signal, most asynchronous control signals could not be used in this project (such as the data flow control signals available to us) due to the tight timing of the FIFO structure and the presence of two clocks. Data corruption and the issue of not efficiently pausing the processor turned out to be the largers sources of error, before the adoption of an asynchronous Moore state machine. It is clear that a more conservative way to deal with the pausing of the processor could not have been employed, due to the further slowing down of the processing system that it would have brought about.

Further problems arose while trying to verify the functionality of our design by means of a simulation testbench. The very nature of the control signals made it very difficult to construct a simple testbench, which would be able to be used as a verification scenario. Furthermore, difficulty arises when we are not careful when writing and reading in a FIFO of the type that we used. To be more speccific, the Legacy mode FIFO needs to be read from one time before the first readrequest, since the first (ever) written word in it will not be available immediately. A FIFO in Show-Ahead mode, which does not suffer from this limitation, comes with a warning from the existing software about performance limitation. We, therefore, chose Legacy Mode in order to enhance the FIFO's performance and simply tolerated this non-immediate existance of the first sample at the output of the FIFO. Simulation showed the success of the FIFO structure in transmitting to the bus all the samples of the input, as shown in Figures 5 - 6. The boundary cases (when the processor is both reading and writing by accident) are covered by the asynchronous logic which controls the ”waitrequest” signal. As expected, this property of the Legacy mode FIFO had no effect on the functionality of our system, which was indeed verified by the success of the encoding process. The initialization of the FIFO is also covered by our logic, since the output of the FIFO is initially set to the zero sample, as Figure proves.

One more limitation factor in (mostly) the decoding process (which would have caused some degradation or lag in the decoding process) is the limited size of the **M4K** blocks, which can serve as the building blocks for the FIFO structures. This project (despite the fact that some peripherals which were not used) were removed, ended up occupying 98/108 memory units, thus limiting the maximum number of frames that could be stored in the FIFO. In this implementation, 90% of the existing memory blocks were allocated to the FIFO structures.

5.2 MP3 Encoding

The encoding program is originally 180% slower than the real-time. By changing the number winder filter subbands or the number of DCT coefficient we can make it as close as a real-time mp3 encode, but the trade off is the quality drops significantly. Moreover, the original thought of the front stage is to read from the ADC into a chunk of data and feed to encoder. Then we found out that with this step added, the encoding program cannot be real-time anymore. And that is the reason we decides not to encode directly from the ADC, instead, break it into two stages. We read from the ADC samples to and encode it to a wav file to the USB, since a wav encoder can be easily real-time. Then the mp3encode encodes from the WAV file on USB to a mp3 file on USB. This way is not real time, but then we don't have to constraint our MP3 encoder and we can have a better quality.

5.3 MP3 Decoding

The decoding problem of MP3PLAY library is that we can successfully ”process” the data, and write to LINE OUT. However, we barely hear the singer's voice and there is distorted background. We try to write PCM data to WAV file instead of writing to FIFO directly to test if the library works. On Matlab, it sounds like random noise. By Realplayer, we cannot hear the voice.

For software verification, it is hard to debug on the board. We try to port MP3PLAY to CentOS so that we can test the result without uploading zImage to Linux Desktop and downloading it to Board back and forth which can help us save lots of time to debug. Also, isolating the problem between software and hardware is important. At first, we discussed the hardware wait-request issues, because since the decoding process is faster than playing process, the FIFO will be exploded without wait-request. We made the PCM write to the FIFO address. But, the wait-request issue is not solved and the MP3PLAY does not provide functions to convert PCM data into WAV, and we met problems to write it into WAV format so it is hard to prove if the library works. The best way to verify the library is writing the decoding material into a wav file and try to test it on laptop. Therefore, we can check and try to find suitable library.

There is another issue about wait-request. We are not sure when the FIFO requests software to feed data into FIFO again if the software is able to write data immediately since the FIFO is empty.

5.4 Team Organization

As our project included some very complex hardware and software design issues, we made the decision to take a divide and conquer approach toward assigning areas of focus. Our project consisted of different serial stages of audio processing. The division of labor resulted in group members being knowledgeable about their particular stage but, affected our ability to collaborate. The major drawback of our organization scheme was that it created a single point of failure.

6 Conclusion

In this project we planned to fully implement a complete audio (mp3 and wav format) recording system, with recording and playback features. Problems in the decoding stage prevented us from fully integrating the system, which offers the possibility for mp3 and wav recording to an external USB Flash Disk drive. During the process of this project, valuable experiences were gained from all the authors, since we encountered the increased complexity of the encoding and decoding mechanisms, along with the numerous standards presented (all of which carry increased complexity). Timing requirements and processing pausing also posed objects. The functionality of the system (to the recording extent) was confirmed by both simulations and through a live demonstration.

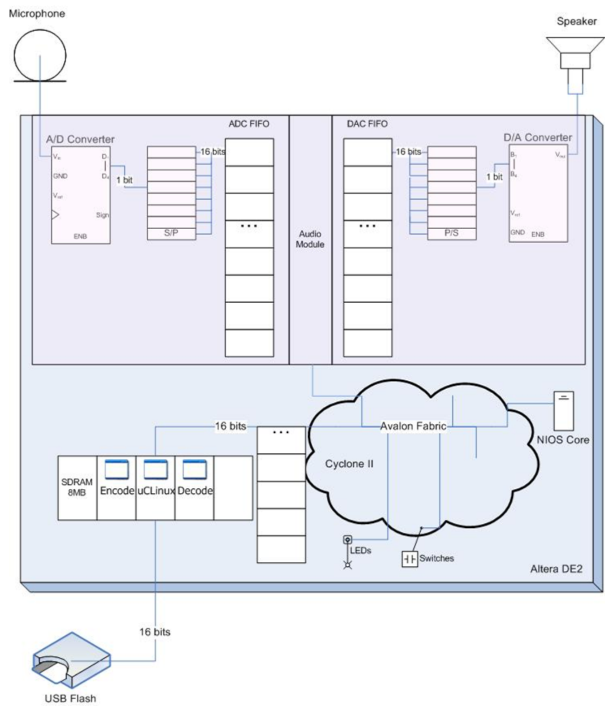


Fig. 1. Top Level Design of mindTunes

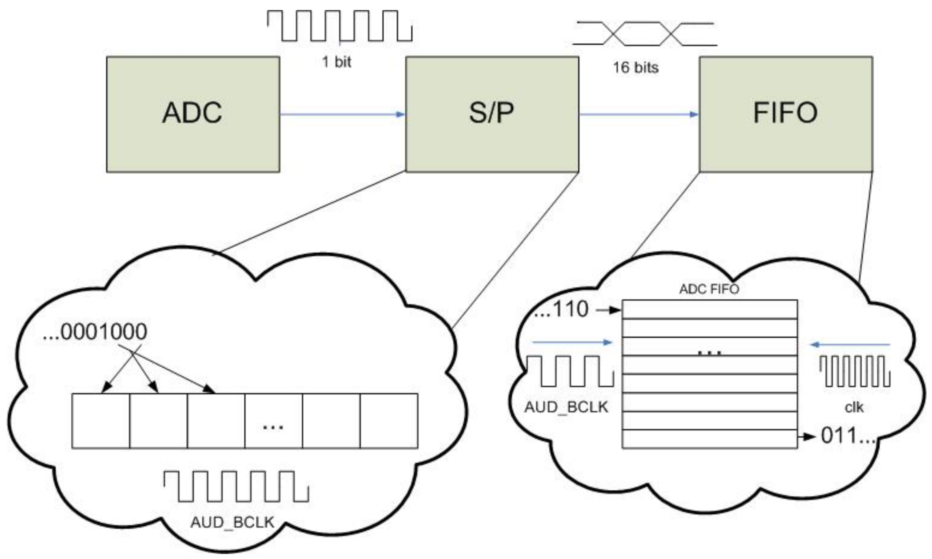


Fig. 2. Block Diagram of mindTunes

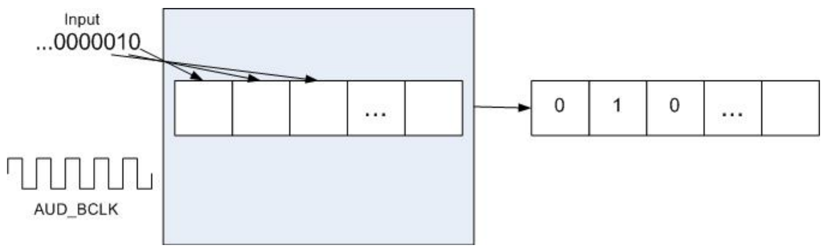


Fig. 3. Serial to Parallel Converter

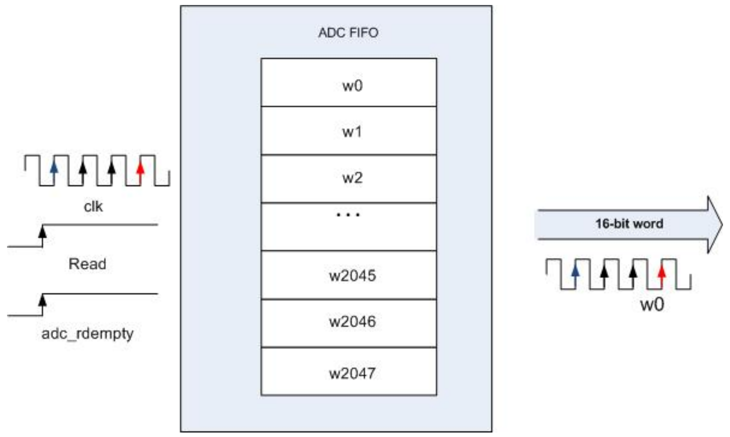


Fig. 4. FIFO structure

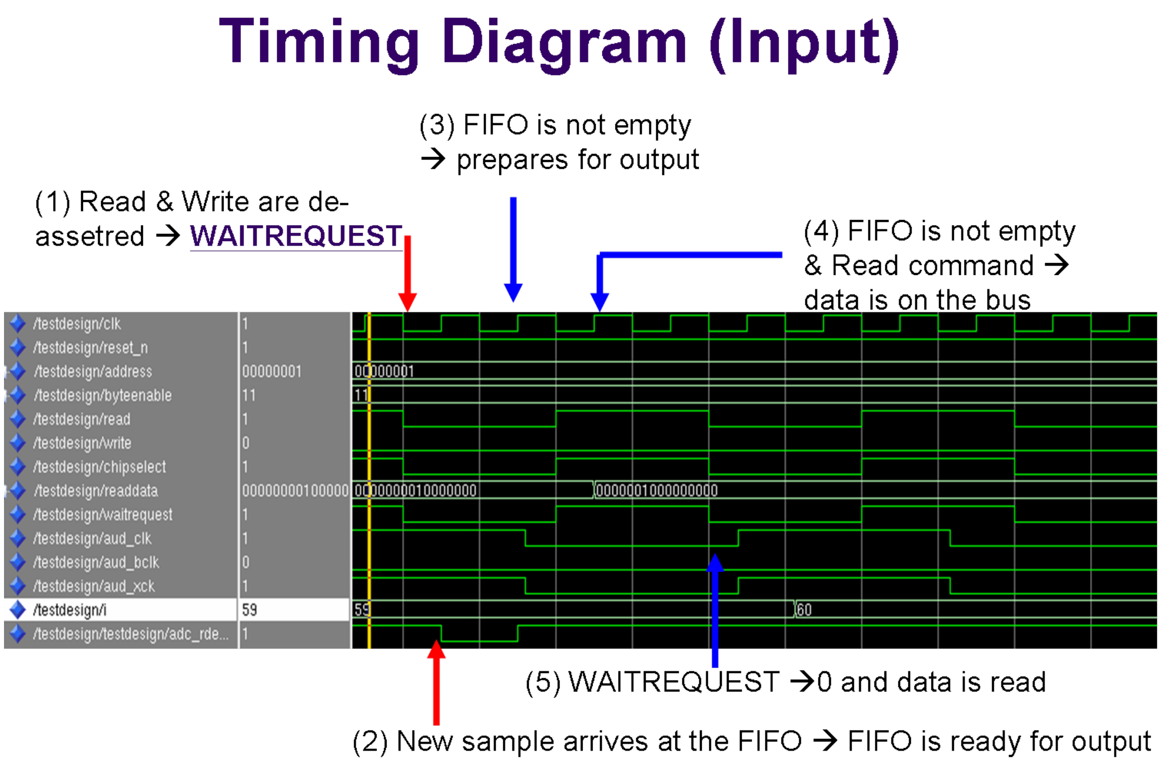

Timing Diagram (Input)
(3) FIFO is not empty → prepares for output
(1) Read & Write are de-assetred → WAITREQUEST
(4) FIFO is not empty & Read command → data is on the bus
/testdesign/clk 1
/testdesign/reset_n 1
/testdesign/address 00000001
/testdesign/byteenable 11
/testdesign/read 1
/testdesign/write 0
/testdesign/chipselect 1
/testdesign/readdata 0000000010000
/testdesign/waitrequest 1
/testdesign/aud_clk 1
/testdesign/aud_bclk 0
/testdesign/aud_xck 1
/testdesign/i 59
/testdesign/testdesign/adc_rde... 1
00000001
11
0000000010000000
0000001000000000
59
60
(5) WAITREQUEST →0 and data is read
(2) New sample arrives at the FIFO → FIFO is ready for output

Fig. 5. Timing Diagram in the process of writing input samples

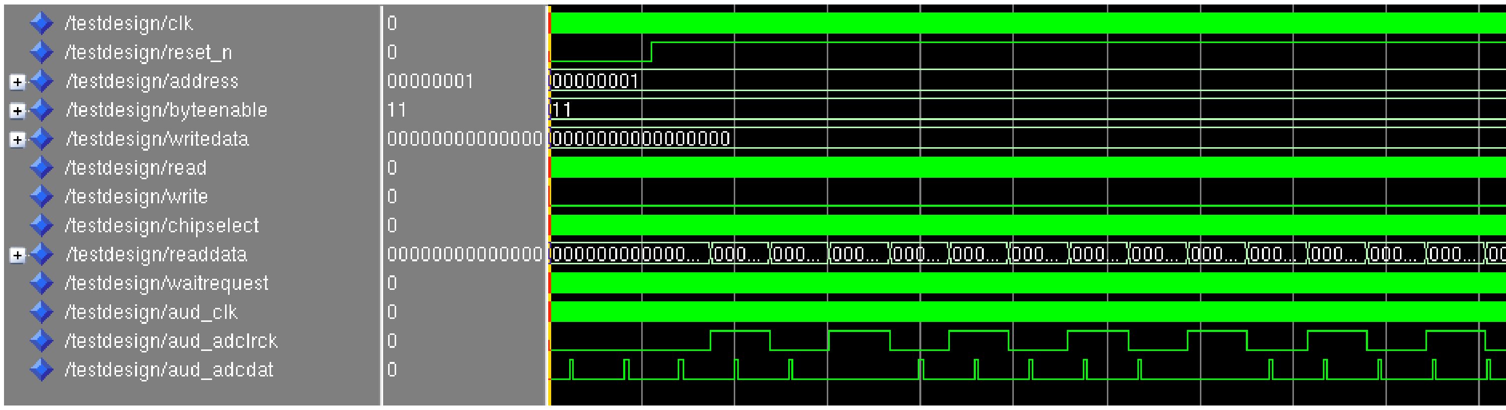


Fig. 6. Timing Diagram in the process of writing input samples

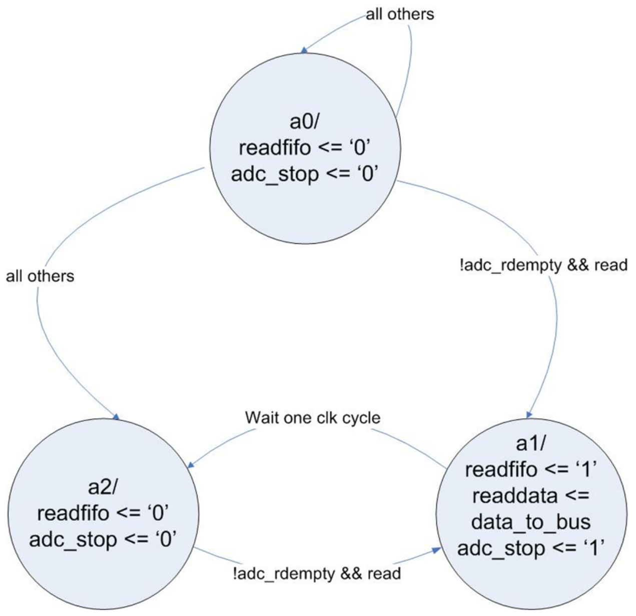
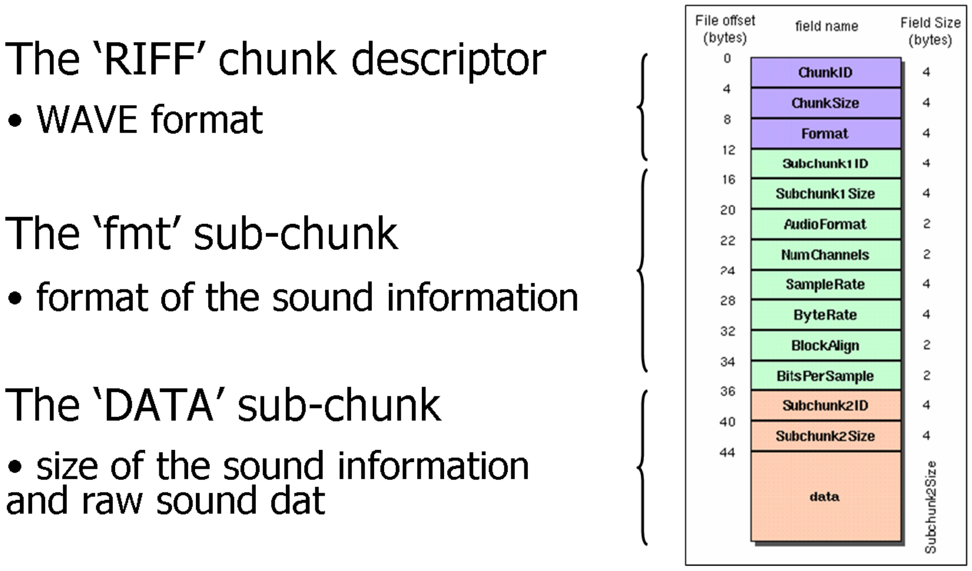


Fig. 7. Function of FIFO controlling state machine


The ‘RIFF’ chunk descriptor

• WAVE format

The ‘fmt’ sub-chunk

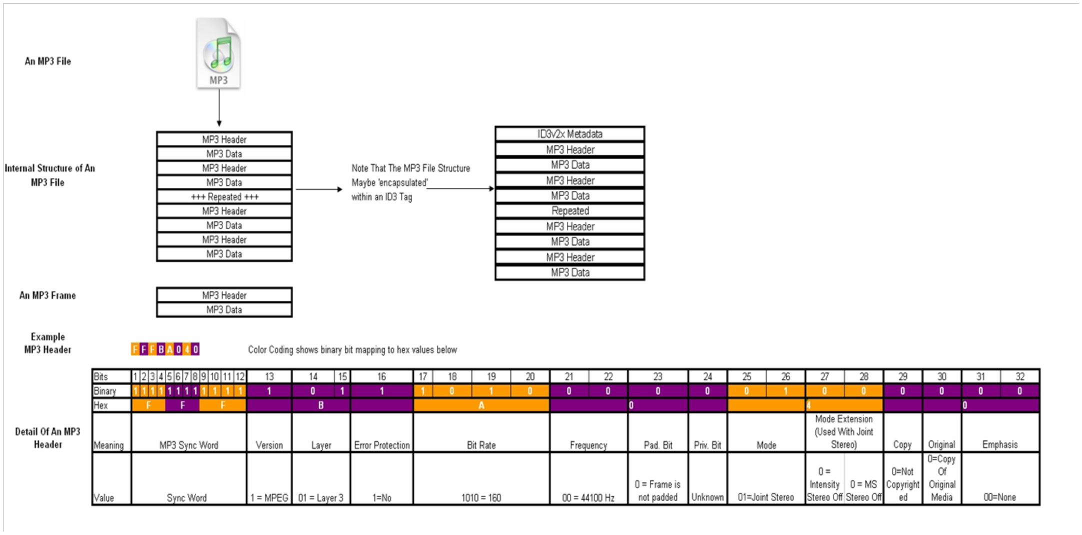
• format of the sound information

The ‘DATA’ sub-chunk

• size of the sound information and raw sound dat

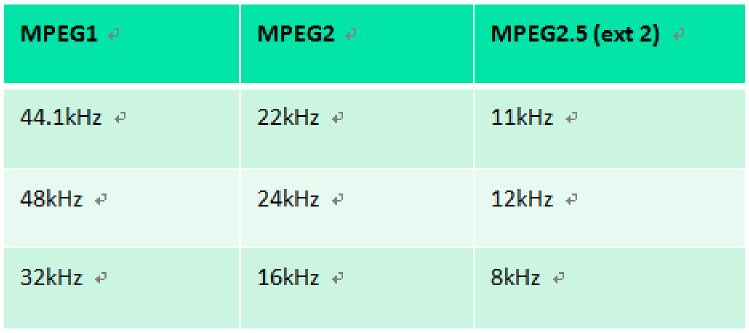
File offset (bytes)	field name	Field Size (bytes)
0	ChunkID	4
4	ChunkSize	4
8	Format	4
12	Subchunk1ID	4
16	Subchunk1Size	4
20	AudioFormat	2
22	NumChannels	2
24	SampleRate	4
28	ByteRate	4
32	BlockAlign	2
34	BitsPerSample	2
36	Subchunk2ID	4
40	Subchunk2Size	4
44	data	Subchunk2Size

Fig. 8. Wav Header



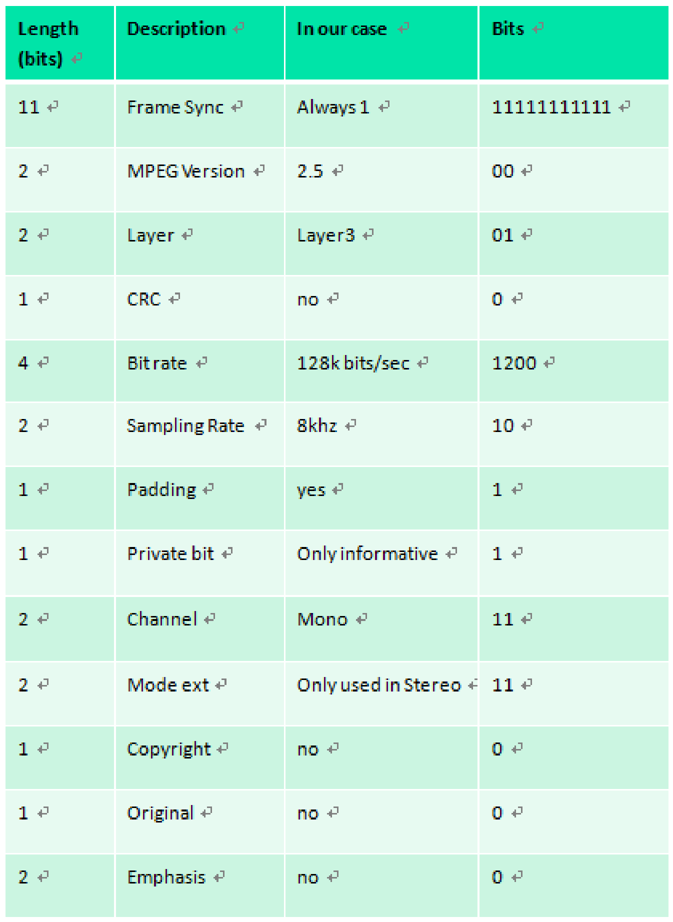
Bits	1 2 3 4 5 6 7 8 9 10 11 12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Binary	1 1 1 1 1 1 1 1 1 1 1 1	1	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
Hex	F F F		B			A						0				4				0	
Meaning	MP3 Sync Word	Version	Layer		Error Protection	Bit Rate				Frequency		Pad. Bit	Priv. Bit	Mode		Mode Extension (Used With Joint Stereo)		Copy	Original	Emphasis	
Value	Sync Word	1 = MPEG	01 = Layer 3		1=No	1010 = 160				00 = 44100 Hz		0 = Frame is not padded	Unknown	01=Joint Stereo		0 = Intensity Stereo Off	0 = MS Stereo Off	0=Not Copyrighted	0=Copy Of Original Media	00=None	

Fig. 9. Mp3 Header and Encoding



MPEG1	MPEG2	MPEG2.5 (ext 2)
44.1kHz	22kHz	11kHz
48kHz	24kHz	12kHz
32kHz	16kHz	8kHz

Fig. 10. Mp3 Header 1



Length (bits)	Description	In our case	Bits
11	Frame Sync	Always 1	11111111111
2	MPEG Version	2.5	00
2	Layer	Layer3	01
1	CRC	no	0
4	Bit rate	128k bits/sec	1200
2	Sampling Rate	8khz	10
1	Padding	yes	1
1	Private bit	Only informative	1
2	Channel	Mono	11
2	Mode ext	Only used in Stereo	11
1	Copyright	no	0
1	Original	no	0
2	Emphasis	no	0

Fig. 11. Mp3 Header 2

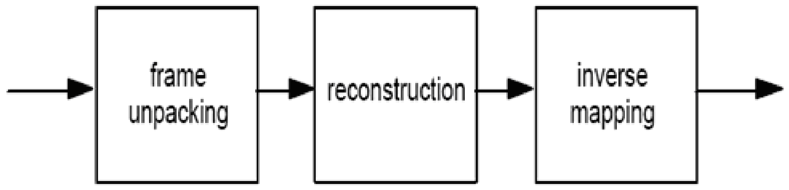


Fig. 12. Mp3 Decoding

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
--use IEEE.STD_LOGIC_ARITH.all;
--use IEEE.STD_LOGIC_SIGNED.all;
use IEEE.numeric_std.all;

-----------------------------------------------------------------------
--------
-- entity
-----------------------------------------------------------------------
--------

entity de2_audio is
port (
    clk : in std_logic;
    reset_n : in std_logic;

    -- Bus master signals
    address      : in std_logic_vector (7 downto 0);
    byteenable   : in std_logic_vector (1 downto 0);
    writedata    : in std_logic_vector (15 downto 0);
    read         : in std_logic;
    write        : in std_logic;
    chipselect   : in std_logic;

     -- Slave signals
    readdata     : out std_logic_vector (15 downto 0);
    waitrequest  : out std_logic;

    -- Audio interface signals
    AUD_CLK         : in std_logic;                  --    18.43MHz
audio clock AUD_XCK
    AUD_ADCLRCK  : out std_logic;                    --    Audio CODEC
ADC LR Clock
    AUD_ADCDAT   : in  std_logic;                    --    Audio CODEC
ADC Data
    AUD_DACLRCK  : out std_logic;                    --    Audio CODEC
DAC LR Clock
    AUD_DACDAT   : out std_logic;                    --    Audio CODEC
DAC Data
    AUD_BCLK     : inout std_logic;                  --    Audio CODEC
Bit-Stream Clock
    AUD_XCK      : out std_logic;
    -- Test signals --
    ledr    : out std_logic_vector (17 downto 0)
  );
end de2_audio;

-----------------------------------------------------------------------
--------
-- architecture
-----------------------------------------------------------------------
--------

architecture imp of de2_audio is

      component de2_fifo
      port (
            data        : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
            rdclk        : IN STD_LOGIC ;
            rdreq        : IN STD_LOGIC ;
            wrclk        : IN STD_LOGIC ;
            wrreq        : IN STD_LOGIC ;
            q            : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
            rdempty        : OUT STD_LOGIC ;
            wrfull        : OUT STD_LOGIC ;
            wrusedw        : OUT STD_LOGIC_VECTOR (12 DOWNTO 0)
      );
      end component;

    component de2_wm8731_audio_in is
    port (
        clk : in std_logic;       --  Audio CODEC Chip Clock AUD_XCK
(18.43 MHz)
        reset_n : in std_logic;
        data_out : out std_logic_vector(15 downto 0);
        audio_req : out std_logic;

        -- Audio interface signals
        AUD_ADCLRCK  : out std_logic;   --    Audio CODEC ADC LR Clock
        AUD_ADCDAT   : in  std_logic;   --    Audio CODEC ADC Data
        AUD_BCLK     : inout std_logic  --    Audio CODEC Bit-Stream
Clock
    );
    end component;

    component de2_wm8731_audio_out is
    port (
        clk : in std_logic;       --  Audio CODEC Chip Clock AUD_XCK
(18.43 MHz)
        reset_n : in std_logic;
        test_mode : in std_logic;       --    Audio CODEC controller
test mode
        data_in : in std_logic_vector(15 downto 0);
        audio_req : out std_logic;

        -- Audio interface signals
        AUD_DACLRCK  : out std_logic;   --    Audio CODEC DAC LR Clock
        AUD_DACDAT   : out std_logic   --    Audio CODEC DAC Data
    );
    end component;

    signal reset : std_logic;

    signal dac_request : std_logic;
    signal adc_request : std_logic;
    signal data_from_bus : std_logic_vector (15 downto 0);
    signal data_to_bus : std_logic_vector (15 downto 0);
    signal adc_data : std_logic_vector (15 downto 0);
    signal dac_data : std_logic_vector (15 downto 0);
      signal adc_rdempty : std_logic;
    signal adc_wrfull : std_logic;
    signal dac_rdempty : std_logic;
    signal dac_wrfull : std_logic;
    signal writefifo   : std_logic;
    signal readfifo       : std_logic;
    signal adc_rdreq   : std_logic;
    signal dac_wrreq   : std_logic;

    signal adc_used_buf : std_logic_vector (12 downto 0);
    signal dac_used_buf : std_logic_vector (12 downto 0);

    signal adc_stop : std_logic;
    signal dac_stop : std_logic;

    --test
    signal ledr_count : std_logic_vector (16 downto 0);
    signal led_ctrl : std_logic;

    --Audio clk dividers
    signal sampling_clk : std_logic;
    signal audio_counter : unsigned (13 downto 0);
    signal counter_aud_adclkrck : unsigned (15 downto 0);
    signal LRCK: std_logic;
    signal bus_state: std_logic_vector (2 downto 0);

      type state_t is (a0, a1, a2, d0, d1, d2);
      signal adc_state, adc_new_state, dac_state, dac_new_state :
state_t;

begin

    adc_rdreq <= readfifo;
    dac_wrreq <= writefifo;

      -- waitrequest is combinational
      waitrequest <= (chipselect and adc_rdempty and read) or
(chipselect and dac_wrfull and write);

      SM_ADC: process(adc_state, adc_rdempty, read, adc_stop)
      begin
            case adc_state is
                  when a0 =>
                        if (adc_rdempty = '0' and read = '1') then
                              adc_new_state <= a1;
                        else
                              adc_new_state <= a0;
                        end if;
                  when a1 =>
                        if (adc_stop = '1') then
                              adc_new_state <= a2;
                        else
                              adc_new_state <= a1;
                        end if;
                  when a2 =>
                        if (read = '1' and adc_rdempty = '0') then
                              adc_new_state <= a1;
                        else
                              adc_new_state <= a0;
                        end if;
                  when others =>
                        adc_new_state <= a0;
            end case;
      end process SM_ADC;

      SM_DAC: process(dac_state, dac_wrfull, write, dac_stop)
      begin
            case dac_state is
                  when d0 =>
                        if (dac_wrfull = '0' and write = '1') then
                              dac_new_state <= d1;
                        else
                              dac_new_state <= d0;
                        end if;
                  when d1 =>
                        if (dac_stop = '1') then
                              dac_new_state <= d2;
                        else
                              dac_new_state <= d1;
                        end if;

                  when d2 =>
                        if (write = '1' and dac_wrfull = '0') then
                              dac_new_state <= d1;
                        else
                              dac_new_state <= d0;
                        end if;
                  when others =>
                        dac_new_state <= d0;
            end case;
      end process SM_DAC;

      ST_ADC: process(clk, reset_n)
      begin
            if reset_n = '0' then
                  adc_state <= a0;
            elsif rising_edge(clk) then
                  if chipselect = '1' then
                        adc_state <= adc_new_state;
                  end if;
            end if;
      end process ST_ADC;

      ST_DAC: process(clk, reset_n)
      begin
            if reset_n = '0' then
                  dac_state <= d0;
            elsif rising_edge(clk) then
                  if chipselect = '1' then
                        dac_state <= dac_new_state;
                  end if;
            end if;
      end process ST_DAC;

      BusComm: process(clk, reset_n)
      begin
            if reset_n = '0' then
                  readdata <= (others => '0');
                  --waitrequest <= '0';
                        adc_stop <= '0';
                        dac_stop <= '0';
                        readfifo <= '0';
                        writefifo <= '0';
                  elsif rising_edge(clk) then
                        if adc_new_state = a0 then
                              if chipselect = '1' then
                                    readfifo <= '0';
                                    --waitrequest <= '0';
                                    adc_stop <= '0';
                              end if;
                        elsif adc_new_state = a1 then
                              if chipselect = '1' then
                                    --if (address(0) = '1' and read = '1')
      then
                                    if (read = '1') then
                                          readdata <= data_to_bus;
                                          readfifo <= '1';
                                          --waitrequest <= '1';
                                          adc_stop <= '1';
                                    end if;
                      end if;
                        elsif adc_new_state = a2 then
                              if chipselect = '1' then
                                    --waitrequest <= '0';
                                    adc_stop <= '0';
                                    readfifo <= '0';
                              end if;
                        end if;

                        if dac_new_state = d0 then
                              if chipselect = '1' then
                                    writefifo <= '0';
                                    --waitrequest <= '0';
                                    dac_stop <= '0';
                              end if;
                        elsif dac_new_state = d1 then
                              if chipselect = '1' then
                                    --if (address(0) = '1' and write = '1')
      then
                                    if (write = '1') then
                                          data_from_bus <= writedata;
                                          writefifo <= '1';
                                          --waitrequest <= '1';
                                          dac_stop <= '1';
                                    end if;
                              end if;
                        elsif dac_new_state = d2 then
                              if chipselect = '1' then
                                    --waitrequest <= '0';
                                    dac_stop <= '0';
                                    writefifo <= '0';
                              end if;
                        end if;

                  end if;
```

```
  end process BusComm;

process(AUD_BCLK)
begin
if rising_edge(AUD_BCLK) then
    counter_aud_adclkrck <= counter_aud_adclkrck + 1;
end if;

if (counter_aud_adclkrck = "0000000000000000") then
    led_ctrl <= not led_ctrl;
end if;
end process;

AUD_XCK <= AUD_CLK;

ADC : de2_wm8731_audio_in
port map (
    clk => AUD_CLK,
    reset_n => reset_n,
    data_out => adc_data,
    audio_req => adc_request,

    AUD_ADCLRCK => AUD_ADCLRCK,
    AUD_ADCDAT => AUD_ADCDAT,
    AUD_BCLK => AUD_BCLK
);

DAC : de2_wm8731_audio_out
port map (
    clk => AUD_CLK,
    reset_n => reset_n,
    test_mode => '0',
    audio_req => dac_request,
    data_in => dac_data,

    AUD_DACLRCK  => AUD_DACLRCK,
    AUD_DACDAT   => AUD_DACDAT
);

ADC_FIFO : de2_fifo
port map (
    data        => adc_data,
    rdclk        => clk,
    rdreq        => adc_rdreq,
    wrclk        => AUD_CLK,
    wrreq        => adc_request,
    q            => data_to_bus,
    rdempty        => adc_rdempty,
    wrfull        => adc_wrfull,
    wrusedw        => adc_used_buf
);

DAC_FIFO : de2_fifo
port map (
    data        => data_from_bus,
    rdclk        => AUD_CLK,
    rdreq        => dac_request,
        wrclk        => clk,
        wrreq        => dac_wrreq,
        q            => dac_data,
        rdempty        => dac_rdempty,
        wrfull        => dac_wrfull,
        wrusedw        => dac_used_buf
    );

      ledr(1)  <= led_ctrl;
      ledr(2)  <= dac_wrfull;
      ledr(3)  <= adc_rdempty;
      ledr(0)  <= '0';

end architecture imp;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- de2_wm8731_audio_in : generate clock and get the samples from device

entity de2_wm8731_audio_in is
port (
    clk : in std_logic;         --   Audio CODEC Chip Clock AUD_XCK (18.43 MHz)
    reset_n : in std_logic;
    data_out : out std_logic_vector(15 downto 0);
    audio_req : out std_logic;

    -- Audio interface signals
        AUD_ADCLRCK   : out std_logic;     --      Audio CODEC ADC LR Clock
    AUD_ADCDAT    : in   std_logic;     --      Audio CODEC ADC Data
    AUD_BCLK       : inout std_logic   --      Audio CODEC Bit-Stream Clock
  );
end   de2_wm8731_audio_in;


architecture Behavioral of   de2_wm8731_audio_in is

    signal lrck : std_logic;
    signal bclk : std_logic;
    signal xck : std_logic;

    signal lrck_divider : std_logic_vector (7 downto 0);
    signal bclk_divider : std_logic_vector (3 downto 0);

    signal set_bclk : std_logic;
    signal set_lrck : std_logic;
    signal lrck_lat : std_logic;
        signal clr_bclk : std_logic;
    signal datain : std_logic;

    signal shift_in : std_logic_vector ( 15 downto 0);
        signal shift_counter : integer := 15;

        -- Second clock divider

        signal lrck_div2 : std_logic_vector (11 downto 0);
        --signal set_lrck2 : std_logic;
        signal bclk_divider2: std_logic_vector (7 downto 0);

begin
    -- LRCK divider
    -- Audio chip main clock is 18.432MHz / Sample rate 48KHz
    -- Divider is 18.432 MHz / 48KHz = 192 (X"C0")
    -- Left justify mode set by I2C controller

    process(clk, reset_n) -- loops Another divider to slow down the LRclk
    begin
        if ( reset_n = '0' ) then
            lrck_div2 <= (others => '0');
        elsif ( clk'event and clk='1' ) then
            if ( lrck_div2 = X"47F")   then        -- 8FF = 900 - 1
                lrck_div2 <= X"000";
            else
                lrck_div2 <= lrck_div2 + '1';
            end if;
        end if;
    end process;

    process(clk, reset_n) -- loops second bclk_divider -- we only need one of the 2
    begin
        if ( reset_n = '0' ) then
            bclk_divider2 <= (others => '0');
        elsif ( clk'event and clk='1' ) then
            if ( bclk_divider2 = X"47" or set_lrck = '1')   then      -- 8F = 90-1
                bclk_divider2 <= X"00";
            else
                bclk_divider2 <= bclk_divider2 + '1';
            end if;
        end if;
    end process;

    process ( lrck_div2 )
    begin
        if ( lrck_div2 = X"47F") then
            set_lrck <= '1';
        else
            set_lrck <= '0';
        end if;
    end process;

-- Here we just have to change set_lrck to set_lrck2 to change the Sampling rate to 8kHz

    process ( clk, reset_n)
    begin
        if ( reset_n = '0') then
            lrck <= '0';
        elsif ( clk 'event and clk = '1') then
            if ( set_lrck = '1') then
                lrck <= not lrck;
            end if;
        end if;
end process;

-- BCLK divider
process ( bclk_divider2 )
begin
        if ( bclk_divider2   = X"23") then -- x5 -- why 5 and B?
                set_bclk <= '1';
        else
                set_bclk <= '0';
        end if;

        if ( bclk_divider2 = X"47") then -- xB
                clr_bclk <= '1';
        else
                clr_bclk <= '0';
        end if;
end process;

process ( clk, reset_n)
begin
        if ( reset_n = '0') then
                bclk <= '0';
        elsif ( clk 'event and clk = '1') then
                if ( set_lrck = '1' or clr_bclk = '1') then
                        bclk <= '0';
                elsif ( set_bclk = '1') then
                        bclk <= '1';
                end if;
        end if;
end process;

    process (clk)
    begin
    if ( clk 'event and clk = '1') then
                if (set_bclk = '1') then
                        shift_in(shift_counter) <= AUD_ADCDAT;
                        if (shift_counter = 0) then
                                shift_counter <= 15;
                        else
                                shift_counter <= shift_counter - 1;
                        end if;
                end if;
    end if;
    end process;
    process(clk)
begin
```

```
            if ( clk'event and clk='1' ) then -- why??
                lrck_lat <= lrck;
            end if;
        end process;
           process (clk)
        begin
            if ( clk'event and clk = '1') then
                if (( lrck_lat = '1' and lrck = '0') or ( lrck_lat = '0' and lrck = '1')) then
                    audio_req <= '1';
                else
                    audio_req <= '0';
                end if;
            end if;
        end process;
        -- Audio data shift output
        process ( clk, reset_n)
        begin
            if ( clk 'event and clk = '1') then
                if ( set_lrck = '1') then
                   data_out <= shift_in;
                end if;
            end if;
        end process;

        -- Audio outputs

        AUD_BCLK        <= bclk;
           AUD_ADCLRCK       <= lrck;

    end architecture;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- de2_wm8731_audio_in : generate clock and set the samples from device

entity de2_wm8731_audio_out is
port (
    clk : in std_logic;         --  Audio CODEC Chip Clock AUD_XCK (18.43 MHz)
    reset_n : in std_logic;
    test_mode : in std_logic;          --    Audio CODEC controller test mode
    data_in : in std_logic_vector(15 downto 0);
        audio_req : out std_logic;

    -- Audio interface signals
    AUD_DACLRCK   : out std_logic;    --     Audio CODEC DAC LR Clock
    AUD_DACDAT    : out std_logic    --     Audio CODEC DAC Data
  );
end   de2_wm8731_audio_out;


architecture Behavioral of   de2_wm8731_audio_out is

    signal lrck : std_logic;
    signal bclk : std_logic;
    signal xck : std_logic;

    signal lrck_divider : std_logic_vector (11 downto 0);
    signal bclk_divider : std_logic_vector (7 downto 0);

    signal set_bclk : std_logic;
    signal set_lrck : std_logic;
    signal clr_bclk : std_logic;
    signal lrck_lat : std_logic;

    signal shift_out : std_logic_vector ( 15 downto 0);

    signal sin_out : std_logic_vector ( 15 downto 0);
    signal sin_counter : std_logic_vector ( 5 downto 0);


begin

    -- LRCK divider
    -- Audio chip main clock is 18.432MHz / Sample rate 48KHz
    -- Divider is 18.432 MHz / 48KHz = 192 (X"C0")
    -- Left justify mode set by I2C controller
```

```
process(clk, reset_n)
begin
    if ( reset_n = '0' ) then
        lrck_divider <= (others => '0');
    elsif ( clk'event and clk='1' ) then
        if ( lrck_divider = X"47F")   then            -- "C0" minus 1
            lrck_divider <= X"000";
        else
            lrck_divider <= lrck_divider + '1';
        end if;
    end if;
end process;

process(clk, reset_n)
begin
    if ( reset_n = '0' ) then
        bclk_divider <= (others => '0');
    elsif ( clk'event and clk='1' ) then
        if ( bclk_divider = X"47" or set_lrck = '1')   then
            bclk_divider <= X"00";
        else
            bclk_divider <= bclk_divider + '1';
        end if;
    end if;
end process;


process ( lrck_divider )
begin
    if ( lrck_divider = X"47F") then
        set_lrck <= '1';
    else
        set_lrck <= '0';
    end if;
end process;

process ( clk, reset_n)
begin
    if ( reset_n = '0') then
        lrck <= '0';
    elsif ( clk 'event and clk = '1') then
        if ( set_lrck = '1') then
            lrck <= not lrck;
        end if;
    end if;
end process;
```

```
-- BCLK divider
process ( bclk_divider )
begin
    if ( bclk_divider = X"23") then
        set_bclk <= '1';
    else
        set_bclk <= '0';
    end if;

    if ( bclk_divider = X"47") then
        clr_bclk <= '1';
    else
        clr_bclk <= '0';
    end if;
end process;

process ( clk, reset_n)
begin
    if ( reset_n = '0') then
        bclk <= '0';
    elsif ( clk 'event and clk = '1') then
        if ( set_lrck = '1' or clr_bclk = '1') then
            bclk <= '0';
        elsif ( set_bclk = '1') then
            bclk <= '1';
        end if;
    end if;
end process;

-- Audio data shift output
process ( clk, reset_n)
begin
    if ( reset_n = '0') then
        shift_out <= (others => '0');
    elsif ( clk 'event and clk = '1') then
        if ( set_lrck = '1') then
            if ( test_mode = '1') then
                shift_out <= sin_out;
            else
                shift_out <= data_in;
            end if;
        elsif ( clr_bclk = '1') then
            shift_out <= shift_out (14 downto 0) & '0';
        end if;
    end if;
end process;

-- Audio outputs
```

```
AUD_DACLRCK   <= lrck;
AUD_DACDAT    <= shift_out(15);

-- Self test with Sin wave

process(clk, reset_n)
begin
    if ( reset_n = '0' ) then
        sin_counter <= (others => '0');
    elsif ( clk'event and clk='1' ) then
        if ( lrck_lat = '1' and lrck = '0')   then
            if (sin_counter = "101111") then
                sin_counter <= "000000";
            else
                sin_counter <= sin_counter + '1';
            end if;
        end if;
    end if;
end process;

process(clk)
begin
    if ( clk'event and clk='1' ) then
        lrck_lat <= lrck;
    end if;
end process;
   process (clk)
begin
    if ( clk'event and clk = '1') then
        if (( lrck_lat = '1' and lrck = '0') or ( lrck_lat = '0' and lrck = '1')) then
            audio_req <= '1';
        else
            audio_req <= '0';
        end if;
    end if;
end process;
process ( sin_counter )
begin
    case sin_counter is
        when "000000" => sin_out <= X"0000";
        when "000001" => sin_out <= X"10b4";
        when "000010" => sin_out <= X"2120";
        when "000011" => sin_out <= X"30fb";
        when "000100" => sin_out <= X"3fff";
        when "000101" => sin_out <= X"4deb";
        when "000110" => sin_out <= X"5a81";
        when "000111" => sin_out <= X"658b";
```

```
                when "001000" => sin_out <= X"6ed9";
                when "001001" => sin_out <= X"7640";
                when "001010" => sin_out <= X"7ba2";
                when "001011" => sin_out <= X"7ee6";
                when "001100" => sin_out <= X"7fff";
                when "001101" => sin_out <= X"7ee6";
                when "001110" => sin_out <= X"7ba2";
                when "001111" => sin_out <= X"7640";
                when "010000" => sin_out <= X"6ed9";
                when "010001" => sin_out <= X"658b";
                when "010010" => sin_out <= X"5a81";
                when "010011" => sin_out <= X"4deb";
                when "010100" => sin_out <= X"3fff";
                when "010101" => sin_out <= X"30fb";
                when "010110" => sin_out <= X"2120";
                when "010111" => sin_out <= X"10b4";
                when "011000" => sin_out <= X"0000";
                when "011001" => sin_out <= X"ef4b";
                when "011010" => sin_out <= X"dee0";
                when "011011" => sin_out <= X"cf05";
                when "011100" => sin_out <= X"c001";
                when "011101" => sin_out <= X"b215";
                when "011110" => sin_out <= X"a57e";
                when "011111" => sin_out <= X"9a74";
                when "100000" => sin_out <= X"9127";
                when "100001" => sin_out <= X"89bf";
                when "100010" => sin_out <= X"845d";
                when "100011" => sin_out <= X"8119";
                when "100100" => sin_out <= X"8000";
                when "100101" => sin_out <= X"8119";
                when "100110" => sin_out <= X"845d";
                when "100111" => sin_out <= X"89bf";
                when "101000" => sin_out <= X"9127";
                when "101001" => sin_out <= X"9a74";
                when "101010" => sin_out <= X"a57e";
                when "101011" => sin_out <= X"b215";
                when "101100" => sin_out <= X"c000";
                when "101101" => sin_out <= X"cf05";
                when "101110" => sin_out <= X"dee0";
                when "101111" => sin_out <= X"ef4b";
                when others => sin_out <= X"0000";
            end case;
        end process;


    end architecture;
```

```
/* main.c
 * Command line interface.
 *
 * This fixed point version of shine is based on Gabriel Bouvigne's original
 * source, version 0.1.2
 *
 * 09/02/01 PRE Ported to Acorn computers running RISC OS.
 * 19/06/03 PRE MPEG2/2.5 support added. RISC OS build conditional.
 */

#include "types.h"

int timelimit = 40;
int dsp_speed = 16000;
int dsp_stereo = 0;
int samplesize = 16;
int flag=0;

FILE * pFile;

/* write a WAVE-header */
void start_wave(int fd, unsigned long cnt)
{
  WaveHeader wh;

  wh.main_chunk = RIFF;
  wh.length     = cnt + sizeof(WaveHeader) - 8;
  wh.chunk_type = WAVE;
  wh.sub_chunk  = FMT;
  wh.sc_len     = 16;
  wh.format     = PCM_CODE;
  wh.modus      = dsp_stereo ? 2 : 1;
  printf("stereo 2 or mono 1: %d\n", wh.modus);
  wh.sample_fq  = dsp_speed;
  wh.byte_p_spl = ((samplesize == 8) ? 1 : 2) * (dsp_stereo ? 2 : 1);
  wh.byte_p_sec = dsp_speed * wh.modus * wh.byte_p_spl;
  wh.bit_p_spl  = samplesize;
  wh.data_chunk = DATA;
  wh.data_length= cnt;
  fwrite (&wh, sizeof(WaveHeader), 1, fd);
}

unsigned long calc_count(void)
{
  unsigned long count;
  if (!timelimit)
    count = 0x7fffffff;
  else {
    count = timelimit * dsp_speed;
    if (dsp_stereo)
      count *= 2;
    if (samplesize != 8)
      count *= 2;
  }
  return count;
}

void record(FILE* fd)
{
            //WAVE header calls
                        int i = 0;
                        unsigned long cnt;
                cnt = calc_count();

                start_wave(fd, cnt);

                        short int * samples = (short *) malloc(sizeof(short) * numSamples);

                        for (i=0; i<numSamples; i++)
                {
                                    //volatile unsigned int audioIn = inw(na_audio);
                                    samples[i] = IORD(na_audio, 1);
                                    switch(i)
                                    {
                        case 8000:
                                       printf("15 sec left\n");
                                       break;
                        case 16000:
                                       printf("14 sec left\n");
                                       break;
                                    case 24000:
                                       printf("13 sec left\n");
                                       break;
                                    case 32000:
                                       printf("12 sec left\n");
                                       break;
                                    case 40000:
                                       printf("11 sec left\n");
                                       break;
                                    case 48000:
                                       printf("10 sec left\n");
```

```
                  break;
                case 56000:
                  printf("9 sec left\n");
                  break;
                case 64000:
                  printf("8 sec left\n");
                  break;
                case 72000:
                  printf("7 sec left\n");
                  break;
                case 80000:
                  printf("6 sec left\n");
                  break;
                case 88000:
                  printf("5 sec left\n");
                  break;
                case 96000:
                  printf("4 sec left\n");
                  break;
                case 114000:
                  printf("3 sec left\n");
                  break;
                case 122000:
                  printf("2 sec left\n");
                  break;
                case 130000:
                  printf("1 sec left\n");
                  break;
                case 138000:
                  printf("0 sec left\n");
                  break;
        //default:

       }
        }

        int numSamplesWritten = fwrite(samples, sizeof(short), numSamples, fd);
        printf("numSamplesWritten=%d\n", numSamplesWritten);
        if (numSamplesWritten!=numSamples) {
          printf("Error: only wrote %d of %d samples\n", numSamplesWritten,
numSamples);
        }

   //printf("samples[128]=%d\n", samples[128]);
        //printf("samples[129]=%d\n", samples[129]);
```

```
                fflush(fd);
        free(samples);
        fclose(fd);
                flag = 1;


}


/*
 * main:
 * -----
 */
int main()
{
  static int j=0;
  static int i=0;




  printf("Program Starts!!");

  while(1)
  {

    if ((inw(na_switch_pio) == 0x3) && (flag==0))
    {
        char str1[20];
        char str2[20];
        //const char* saxiao = "/mnt/test";
        //const char* WAVE = "wav";
        const char* name = "/mnt/Mike";
        const char* EMPEE3 = "mp3";
       //for (i=0;i<10;i++){
       //strcpy(str1, saxiao);
       sprintf(str1, "/mnt/test_%d.wav", j);

        FILE * fd = fopen(str1, "w");
        if (fd==NULL)
        {
             printf("Error opening file\n");
             exit(-1);
        }

            printf("Start Recording......\n");
            record(fd);
            printf("Done recording!\n");
        //The ADC to wac code
        strcpy(str2, name);
        sprintf(str2, "%s_%d.%s", name, j, EMPEE3);
        j++;
        mp3encode(str2, str1);
        printf("Done Encoding %s\n",str2);
    }
    if ((inw(na_switch_pio) == 0x1) &&  (flag==1))
    {
      flag=0;
      printf("Ready to Record\n");
    }
    if ((inw(na_switch_pio) == 0x5)  && (flag==0))
       exit(EXIT_SUCCESS);
  }
}
```

```
/* main.c
 * Command line interface.
 *
 * This fixed point version of shine is based on Gabriel Bouvigne's original
 * source, version 0.1.2
 *
 * 09/02/01 PRE Ported to Acorn computers running RISC OS.
 * 19/06/03 PRE MPEG2/2.5 support added. RISC OS build conditional.
 */

#include "types.h"


//filetypes
#define AMPEG   0x1ad
#define DATA    0xffd
#define WAVE    0xfb1
#define CD_DATA 0x0cd

config_t config;
int cutoff;


void (*L3_window_filter_subband)(unsigned long **buffer, long s[SBLIMIT], int k);
void (*L3_mdct_sub)(long sb_sample[2][3][18][SBLIMIT],
                long mdct_freq[2][2][samp_per_frame2]);
int (*quantize)(int ix[samp_per_frame2], int stepsize);

extern void L3_window_filter_subband_a(unsigned long **buffer, long s[SBLIMIT], int
k);
extern void L3_mdct_sub_a(long sb_sample[2][3][18][SBLIMIT],
                long mdct_freq[2][2][samp_per_frame2]);
extern int quantize_a(int ix[samp_per_frame2], int stepsize);



/*
 * error:
 * ------
 */
void error(char *s)
{
  printf("[ERROR] %s\n",s);
  exit(1);
}
```

```
/*
 * print_usage:
 * ------------
 */
/*
static void print_usage()
{
  printf("\nUSAGE   :  Shine [options] <infile> <outfile>\n"
         "options : -h             this help message\n"
         "          -b <bitrate>  set the bitrate [32-320], default 128kbit\n"
         "          -c            set copyright flag, default off\n"
         "          -o            reset original flag, default on\n"
         "          -r [sample rate]  raw cd data file instead of wave\n"
         "          -m            mono from stereo, raw mono with -r\n"
         );

  exit(0);
}
 */
/*
 * set_functions
 * -------------
 * Initialises the function pointers for strongarm or normal arm
 * function versions.
 */
static void set_functions()//int strongarm)
{

  L3_window_filter_subband = L3_window_filter_subband_a;
  L3_mdct_sub = L3_mdct_sub_a;
  quantize = quantize_a;

}

/*
 * set_defaults:
 * -------------
 */
static void set_defaults()
{
  config.mpeg.type = MPEG2;    //Original MPEG1 Yen
  config.mpeg.layr = LAYER_3;
  config.mpeg.mode = MODE_MONO;
  config.mpeg.bitr = 128;
  config.mpeg.psyc = 0;
  config.mpeg.emph = 0;
  config.mpeg.crc  = 0;
  config.mpeg.ext  = 0;
  config.mpeg.mode_ext  = 0;
  config.mpeg.copyright = 0;
  config.mpeg.original  = 0;
  config.mpeg.channels = 1;         //2 to 1 Yen
  config.mpeg.granules = 1;           // originally 2, but since it is MPEG2.5 so granules=1
Yen
  cutoff = 418; // 16KHz @ 44.1Ksps   original 418  Yen
  config.wave.samplerate = 8000;  //original 41000 Yen
}

/*
 * parse_command:
 * --------------
 */
/*
static int parse_command(int argc, char **argv, int *raw, int *mono_from_stereo)
{
  int i = 0, x;

  if(argc<3) return 0;

  while(argv[++i][0]=='-')
    switch(argv[i][1])
    {
      case 'b':
        config.mpeg.bitr = atoi(argv[++i]);
        break;

      case 'c':
        config.mpeg.copyright = 1;
        break;

      case 'o':
        config.mpeg.original = 0;
        break;

      case 'r':
        *raw = 1;
        x = atoi(argv[i+1]);
        if(x > 7999)
        {
          i++;
          config.wave.samplerate = x;
        }
        break;

      case 'm':
        *mono_from_stereo = 1;
        break;

      default :
        return 0;
    }

  /*if((argc-i)!=2) return 0;
  config.infile  = argv[i++];
  config.outfile = argv[i];
  return 1;
}
*/
/*
 * find_samplerate_index:
 * ----------------------
 */
static int find_samplerate_index(long freq)
{
  static long sr[4][3] = {{11025, 12000,  8000},   /* mpeg 2.5 */
                          {    0,     0,     0},   /* reserved */
                          {22050, 24000, 16000},   /* mpeg 2 */
                          {44100, 48000, 32000}};  /* mpeg 1 */
  int i, j;

  for(j=0; j<4; j++)
    for(i=0; i<3; i++)
      if((freq == sr[j][i]) && (j != 1))
      {
        config.mpeg.type = j;
        return i;
      }

  error("Invalid samplerate");
  return 0;
}

/*
 * find_bitrate_index:
 * -------------------
 */
static int find_bitrate_index(int bitr)
{
  static long br[2][15] =
    {{0, 8,16,24,32,40,48,56, 64, 80, 96,112,128,144,160},   /* mpeg 2/2.5 */
     {0,32,40,48,56,64,80,96,112,128,160,192,224,256,320}};  /* mpeg 1 */
  int i;

  for(i=1; i<15; i++)
    if(bitr==br[config.mpeg.type & 1][i]) return i;

  error("Invalid bitrate");
  return 0;
}

int set_cutoff(void)
{
  static int cutoff_tab[3][2][15] =
  {
    { /* 44.1k, 22.05k, 11.025k */
      {100,104,131,157,183,209,261,313,365,418,418,418,418,418,418}, /* stereo */
      {183,209,261,313,365,418,418,418,418,418,418,418,418,418,418}  /* mono */
    },
    { /* 48k, 24k, 12k */
      {100,104,131,157,183,209,261,313,384,384,384,384,384,384,384}, /* stereo */
      {183,209,261,313,365,384,384,384,384,384,384,384,384,384,384}  /* mono */
    },
    { /* 32k, 16k, 8k */
      {100,104,131,157,183,209,261,313,365,418,522,576,576,576,576}, /* stereo */
      {183,209,261,313,365,418,522,576,576,576,576,576,576,576,576}  /* mono */
    }
  };

  return cutoff_tab[config.mpeg.samplerate_index]
                   [config.mpeg.mode == MODE_MONO]
                   [config.mpeg.bitrate_index];
}

/*
 * check_config:
 * -------------
 */
static void check_config()
{
  static char *mode_names[4]    = { "stereo", "j-stereo", "dual-ch", "mono" };
  static char *layer_names[4]   = { "", "III", "II", "I" };
  static char *version_names[4] = { "MPEG 2.5", "", "MPEG 2", "MPEG 1" };
  static char *psy_names[3]     = { "none", "MUSICAM", "Shine" };
  static char *demp_names[4]    = { "none", "50/15us", "", "CITT" };
  config.mpeg.samplerate_index = find_samplerate_index(config.wave.samplerate);
  config.mpeg.bitrate_index    = find_bitrate_index(config.mpeg.bitr);
  cutoff = set_cutoff();

  printf("%s layer %s, %s  Psychoacoustic Model: %s\n",
           version_names[config.mpeg.type],
           layer_names[config.mpeg.layr],
           mode_names[config.mpeg.mode],
           psy_names[config.mpeg.psyc] );
  printf("Bitrate=%d kbps  ",config.mpeg.bitr );
  printf("De-emphasis: %s   %s %s\n",
           demp_names[config.mpeg.emph],
           (config.mpeg.original) ? "Original" : "",
           (config.mpeg.copyright) ? "(C)" : "" );
}

/*
 * main:
 * -----
 */
int mp3encode(const char* name, const char* saxiao)
{
  int mono_from_stereo = 0;
  int raw =0;
  config.infile=saxiao;
  config.outfile=name;

  set_defaults();

  set_functions();

  wave_open(raw,mono_from_stereo);

  check_config();

  printf("Encoding \"%s\" to \"%s\"\n", config.infile, config.outfile);


  L3_compress();


  wave_close();

  return 0;
}
```

```
#include "defs.h"
#include "mpegdec.h"
#include "genre.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/uio.h>
#include <sys/file.h>
#include <nios2_system.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <getopt.h>
#include <signal.h>
#include <linux/soundcard.h>
#include <sys/resource.h>
#include <config/autoconf.h>
#include "fmtheaders.h"

int timelimit = 85;
int dsp_speed = 8000;
int dsp_stereo = 0;
int samplesize = 16;

/* write a WAVE-header */
void start_wave(int fd, unsigned long cnt)
{
  WaveHeader wh;

  wh.main_chunk = RIFF;
  wh.length     = cnt + sizeof(WaveHeader) - 8;
  wh.chunk_type = WAVE;
  wh.sub_chunk  = FMT;
  wh.sc_len     = 16;
  wh.format     = PCM_CODE;
  wh.modus      = dsp_stereo ? 2 : 1;
  wh.sample_fq  = dsp_speed;
  wh.byte_p_spl = ((samplesize == 8) ? 1 : 2) * (dsp_stereo ? 2 : 1);
  wh.byte_p_sec = dsp_speed * wh.modus * wh.byte_p_spl;
  wh.bit_p_spl  = samplesize;
  wh.data_chunk = DATA;
  wh.data_length= cnt;
  fwrite (&wh, sizeof(WaveHeader), 1, fd);
```

```
}

FILE *f;

#ifdef CONFIG_USER_SETKEY_SETKEY
#include <key/key.h>
#endif

/**********************************************************************
*****/

int     verbose;
int     quiet;
int     http_streaming;
int     lcd_line, lcd_time;
int     prebuflimit;
int     lcdfd = -1;
int     gotsigusr1;
char    key[128];

/*
 *      Keep track of start and end times.
 */
struct timeval  tvstart, tvend;

/*
 *      Global settings per decode stream. Used to control the final
 *      PCM to raw driver data conversion.
 */
static int      stereo;
static int      bits;
static int      testtone;
static int      quality;

/*
 *      Master MP3 decoder settings.
 */
static MPEGDEC_STREAM *mps = NULL;

static MPEGDEC_CTRL     mpa_ctrl;
static MPEGDEC_CTRL     mpa_defctrl = {
        NULL,   // Bitstream access is default file I/O
        // Layers I & II settings (#3)
        { FALSE, { 1, 2, 48000 }, { 1, 2, 48000 } },
        // Layer III settings (#3)
        { FALSE, { 1, 2, 48000 }, { 1, 2, 48000 } },
	0,		// #2: Don't check mpeg validity at start
			// (needed for mux stream)
	2048		// #2: Stream Buffer size
};

static char *modes[] = { "stereo", "j-stereo", "dual", "mono" };

/***********************************************************************
*****/
/*
 *	MP3 data stream support (could be file or http stream).
 */
#define MP3_BUF_SIZE	(4*1024)

static char	*mp3_filename;
static int	mp3_fd;
static INT8	*mp3_buffer;
static UINT32 mp3_buffer_size;
static UINT32 mp3_buffer_offset;
static UINT32 mp3_buffer_next_block;
static UINT32 mp3_stream_size;

static char	*rawbuf;
static char	*prebuffer;

int		prebufsize;
int		prebufcnt;
int		prebufnow;

/***********************************************************************
*****/

/*
 *	MP3 file TAG info. Output when in verbose mode. This structure is
 *	designed to match the in-file structure, don't change it!
 *	Nice printable strings are generated in the other vars below.
 */
struct mp3tag {
	char		tag[3];
	char		title[30];
	char		artist[30];
	char		album[30];
	char		year[4];
	char		comments[30];
	unsigned char	genre;
};
```

```
static struct mp3tag    mp3_tag;
static int              mp3_gottag;

static char             mp3_title[32];
static char             mp3_artist[32];
static char             mp3_year[8];
static char             mp3_album[32];
static char             mp3_comments[32];
static char             *mp3_genre;

unsigned long calc_count(void)
{
  unsigned long count;

  if (!timelimit)
    count = 0x7fffffff;
  else {
    count = timelimit * dsp_speed;
    if (dsp_stereo)
      count *= 2;
    if (samplesize != 8)
      count *= 2;
  }
  return count;
}
/***********************************************************************
*****/
/*
 *      Trivial signal handler, processing is done from the main loop.
 */

void usr1_handler(int ignore)
{
        gotsigusr1 = 1;
}

/***********************************************************************
*****/

/*
 *      Get stream size (just file size).
 */

static int getstreamsize(void)
{
	struct stat	st;
	if (stat(mp3_filename, &st) < 0)
		return(0);
	return(st.st_size);
}

/**********************************************************************
*****/
/*
 *	Get another chunk of data into RAM.
 */

static UINT32 getnextbuffer()
{
	int	rc;

	lseek(mp3_fd, mp3_buffer_next_block, SEEK_SET);
	rc = read(mp3_fd, mp3_buffer, MP3_BUF_SIZE);
	mp3_buffer_size = (rc < 0) ? 0 : rc;
	mp3_buffer_next_block += mp3_buffer_size;
	return(mp3_buffer_size);
}

/**********************************************************************
*****/

/*
 *	Start our own bitstream access routines.
 */

INT32 bs_open(char *stream_name, INT32 buffer_size, INT32 *stream_size)
{
#if 0
	printf("bs_open: '%s'\n", stream_name);
#endif

	if (!mp3_buffer)
		return(0);

	mp3_buffer_offset = 0;

	/* We know total size, we can set it */
	*stream_size = mp3_stream_size;

	/* Just return a dummy handle (not NULL) */
	return(1);
```

```
}

/************************************************************************
*****/

void bs_close(INT32 handle)
{
#if 0
	printf("bs_close\n");
#endif
	/* Don't need to do anything... */
}

/************************************************************************
*****/

INT32 bs_read(INT32 handle, void *buffer, INT32 num_bytes)
{
	INT32 read_size;

	if (!handle )
		return(-1);

tryagain:
	read_size = mp3_buffer_size - mp3_buffer_offset;
	if (read_size > num_bytes)
		read_size = num_bytes;

	if (read_size > 0) {
		if(!buffer)
			return(-1);
		memcpy(buffer, &mp3_buffer[mp3_buffer_offset], read_size);
		mp3_buffer_offset += read_size;
	} else {
		if (getnextbuffer() > 0) {
			mp3_buffer_offset = 0;
			goto tryagain;
		}
		read_size = -1; /* End of stream */
	}

	return(read_size);
}

/************************************************************************
*****/
```

```
int bs_seek(INT32 handle, INT32 abs_byte_seek_pos)
{
	if (!handle)
		return(-1);

	if (abs_byte_seek_pos <= 0)
		mp3_buffer_offset = 0;
	else if (abs_byte_seek_pos >= mp3_buffer_size)
		return(-1);
	else
		mp3_buffer_offset = abs_byte_seek_pos;
	return(0);
}

/**********************************************************************
*****/

MPEGDEC_ACCESS bs_access = { bs_open, bs_close, bs_read, bs_seek };

/**********************************************************************
*****/

void mkstring(char *str, char *buf, int size)
{
   int i, j = 0;
	char save;
	for (i = size - 1; i >= 0; i--) {
		if (buf[i] != ' ') {
			while (buf[j] == ' ')
				j++;
			strncpy(str, &buf[j], i - j + 1);
			str[i-j+1] = '\0';
			return;
		}
	}
}

/**********************************************************************
*****/

/*
 *	Get TAG info from mp3 file, if it is present. No point doing a
 *	fatal exit on errors, just assume no tag info is present.
 */
```

```
void getmp3taginfo(void)
{
	long	pos;
	int	size;

	mp3_gottag = 0;
	size = sizeof(mp3_tag);
	pos = mp3_stream_size - size;
	if (pos < 0)
		return;

	if (lseek(mp3_fd, pos, SEEK_SET) < 0)
		return;
	if (read(mp3_fd, &mp3_tag, size) != size)
		return;
	if (strncmp(&mp3_tag.tag[0], "TAG", 3) != 0)
		return;

	/* Return file pointer to start of file */
	lseek(mp3_fd, 0, SEEK_SET);

	/* Construct fill NULL terminated strings */
	mkstring(&mp3_title[0], &mp3_tag.title[0], sizeof(mp3_tag.title));
	mkstring(&mp3_artist[0], &mp3_tag.artist[0], sizeof(mp3_tag.artist));
	mkstring(&mp3_album[0], &mp3_tag.album[0], sizeof(mp3_tag.album));
	mkstring(&mp3_year[0], &mp3_tag.year[0], sizeof(mp3_tag.year));
	mkstring(&mp3_comments[0], &mp3_tag.comments[0],
sizeof(mp3_tag.comments));
	mp3_genre = (mp3_tag.genre >= genre_count) ? "Unknown" :
		genre_table[mp3_tag.genre];

	mp3_gottag = 1;
}

/*********************************************************************
*****/

/*
 *	Print out everything we know about the MP3 stream.
 */

void printmp3info(void)
{
	if (quiet)
		return;
```

```
        if (verbose == 0) {
                printf("New");
                printf("%s: MPEG%d-%s (%ld ms)\n", mp3_filename, mps->norm,
                        (mps->layer == 1)?"I":(mps->layer == 2)?"II":"III",
                        mps->ms_duration);
                return;
        }

        /* This is the verbose output */
        printf("%s:\n", mp3_filename);
        printf("    MPEG%d-%s %s %dkbps %dHz (%ld ms)\n", mps->norm,
                (mps->layer == 1) ? "I" : (mps->layer == 2) ? "II" : "III",
                modes[mps->mode], mps->bitrate, mps->frequency,
                mps->ms_duration );
        printf("    Decoding: Channels=%d Quality=%d Frequency=%dHz\n",
                mps->dec_channels, mps->dec_quality, mps->dec_frequency );

        if (mp3_gottag) {
                printf("    Title:    %s\n", mp3_title);
                printf("    Artist:   %s\n", mp3_artist);
                printf("    Album:    %s\n", mp3_album );
                printf("    Year:     %s\n", mp3_year);
                printf("    Comments: %s\n", mp3_comments);
                printf("    Genre:    %s\n", mp3_genre);
        }
}

/***********************************************************************
*****/

/*
 *      Print out the name on a display device if present.
 */

void lcdtitle(void)
{
        char    ctrl, *name;
        int     ivp;
        struct  iovec iv[4];
        char    prebuf[10];
        char    postbuf;
        char    *p;
        int     namelen;

        /* Install a signal handler to allow updates to be forced */
        signal(SIGUSR1, usr1_handler);
```

```
/* Determine the name to display.  We use the tag if it is
 * present and the basename of the file if not.
 */
if (mp3_gottag) {
	name = mp3_title;
	namelen = strlen(name);
} else {
	name = strrchr(mp3_filename, '/');
	if (name == NULL)
		name = mp3_filename;
	else
		name++;
	p = strchr(name, '.');
	if (p == NULL)
		namelen = strlen(name);
	else
		namelen = p - name;
}

if (lcd_line) {
	/* Lock the file so we can access it... */
	if (flock(lcdfd, LOCK_SH | LOCK_NB) == -1)
		return;
	if (lcd_line == 0) {
		prebuf[0] = '\f';
		prebuf[1] = '\0';
	} else if (lcd_line == 1) {
		strcpy(prebuf, "\003\005");
	} else if (lcd_line == 2) {
		strcpy(prebuf, "\003\v\005");
	}

	/*
	 * Now we'll write the title out.  We'll do this atomically
	 * just in case two players decide to coexecute...
	 */
	ivp = 0;
	iv[ivp].iov_len = strlen(prebuf) * sizeof(char);
	iv[ivp++].iov_base = prebuf;

	iv[ivp].iov_len = namelen * sizeof(char);
	iv[ivp++].iov_base = name;

	//postbuf = '\n';
	//iv[ivp].iov_len = sizeof(char);
```

```
		//iv[ivp++].iov_base = &postbuf;
		writev(lcdfd, iv, ivp);

		/* Finally, unlock it since we've finished. */
		flock(lcdfd, LOCK_UN);
	}
}

/***********************************************************************
*****/

/*
 *	Output time info to display device.
 */

void lcdtime(time_t sttime)
{
	static time_t	lasttime;
	time_t		t;
	char		buf[15], *p;
	int		m, s;

	t = time(NULL) - sttime;
	if (t != lasttime && flock(lcdfd, LOCK_SH | LOCK_NB) == 0) {
		p = buf;
		*p++ = '\003';
		if (lcd_time == 2)
			*p++ = '\v';
		*p++ = '\005';
		m = t / 60;
		s = t % 60;
		if (s < 0) s += 60;
		sprintf(p, "%02d:%02d", m, s);
		write(lcdfd, buf, strlen(buf));
		flock(lcdfd, LOCK_UN);
	}
	lasttime = t;
}

/***********************************************************************
*****/

/*
 *	Configure DSP engine settings for playing this track.
 */
```

```
void setdsp(int fd, int playstereo, int playbits)
{
        if (ioctl(fd, SNDCTL_DSP_SPEED, &mps->dec_frequency) < 0) {
                fprintf(stderr, "ERROR: Unable to set frequency to %d, "
                        "errno=%d\n", mps->dec_frequency, errno);
                exit(1);
        }

        /* Check if data stream is stereo, otherwise must play mono. */
        stereo = (mps->channels == 1) ? 0 : playstereo;
        if (ioctl(fd, SNDCTL_DSP_STEREO, &stereo) < 0) {
                fprintf(stderr, "ERROR: Unable to set stereo to %d, "
                        "errno=%d\n", stereo, errno);
                exit(1);
        }

#if BYTE_ORDER == LITTLE_ENDIAN
        bits = (playbits == 16) ? AFMT_S16_LE : AFMT_U8;
#else
        bits = (playbits == 16) ? AFMT_S16_BE : AFMT_U8;
#endif
        if (ioctl(fd, SNDCTL_DSP_SAMPLESIZE, &bits) < 0) {
                fprintf(stderr, "ERROR: Unable to set sample size to "
                        "%d, errno=%d\n", bits, errno);
                exit(1);
        }
}

/*****************************************************************************
*****/

/*
 *      Generate a tone instead of PCM output. This is purely for
 *      testing purposes.
 */

int writetone(int fd, INT16 *pcm[2], int count)
{
        static int      ramp = 0;
        unsigned char   *pbufbp;
        unsigned short  *pbufwp;
        int             i;

        if (count <= 0)
                return(count);
        if (stereo) {
                if (bits == 8) {
                        /* 8bit stereo */
                        pbufbp = (unsigned char *) rawbuf;
                        for (i = 0; (i < count); i++) {
                                *pbufbp++ = ramp;
                                *pbufbp++ = ramp;
                                ramp = (ramp + 0x80) & 0x7ff;
                        }
                        i = count * 2 * sizeof(unsigned char);
                } else {
                        /* 16bit stereo */
                        pbufwp = (unsigned short *) rawbuf;
                        for (i = 0; (i < count); i++) {
                                *pbufwp++ = ramp;
                                *pbufwp++ = ramp;
                                ramp = (ramp + 0x80) & 0x7ff;
                        }
                        i = count * 2 * sizeof(unsigned short);
                }
        } else {
                if (bits == 8) {
                        /* 8bit mono */
                        pbufbp = (unsigned char *) rawbuf;
                        for (i = 0; (i < count); i++) {
                                *pbufbp++ = ramp;
                                ramp = (ramp + 0x80) & 0x7ff;
                        }
                        i = count * sizeof(unsigned char);
                } else {
                        /* 16bit mono */
                        pbufwp = (unsigned short *) rawbuf;
                        for (i = 0; (i < count); i++) {
                                *pbufwp++ = ramp;
                                ramp = (ramp + 0x80) & 0x7ff;
                        }
                        i = count * sizeof(unsigned short);
                }
        }

        write(fd, rawbuf, i);
        return(i);
}

/*************************************************************************
*****/
```

```
//writepcm(dspfd, pcm, pcmcount);
/*
 *      Write out the PCM data to the file descriptor, translating to
 *      the specified data format.
 */
/*

int writepcm(int fd, INT16 *pcm[2], int count)
{
        unsigned short *pcm0, *pcm1;
        unsigned char  *pbufbp;
        unsigned short *pbufwp;
        char           *buf;
        int            i;

        if (count <= 0)
                return(count);
        if (testtone)
                return(writetone(fd, pcm, count));

        buf = rawbuf;

        if (stereo) {
                if (bits == 8) {
                        // 8bit stereo //
                        pcm0 = pcm[0];
                        pcm1 = pcm[1];
                        pbufbp = (unsigned char *) buf;
                        for (i = 0; (i < count); i++) {
                                *pbufbp++ = (*pcm0++ ^ 0x8000) >> 8;
                                *pbufbp++ = (*pcm1++ ^ 0x8000) >> 8;
                        }
                        i = count * 2 * sizeof(unsigned char);
                } else {
                        // 16bit stereo
                        pcm0 = pcm[0];    //pcm[0] store pcm0 address
                        pcm1 = pcm[1];
                        pbufwp = (unsigned short *) buf;
                        for (i = 0; (i < count); i++) {
                                *pbufwp++ = *pcm0++;
                                *pbufwp++ = *pcm1++;
                        }
                        i = count * 2 * sizeof(unsigned short);
                }
        } else {
                if (bits == 8) {
```

```
				// 8bit mono
				pcm0 = pcm[0];
				pbufbp = (unsigned char *) buf;
				for (i = 0; (i < count); i++)
					*pbufbp++ = (*pcm0++ ^ 0x8000) >> 8;
				i = count * sizeof(unsigned char);
			} else {
				// 16bit mono, no translation required!
				i = count * sizeof(unsigned short);
				buf = (char *) pcm[0];
			}
		}

		if (prebufnow) {
			memcpy(&prebuffer[prebufcnt], buf, i);
			prebufcnt += i;
			if (prebufcnt > prebufnow) {
				write(fd, &prebuffer[0], prebufcnt);
				prebufnow = prebufcnt = 0;
			}
		} else {
			write(fd, buf, i);
		}

		return(i);
}
*/

/*********************************************************************
*****/

/*
 *	Flush out any remaining buffered PCM data. This is really to allow
 *	for prebuffing of files smaller than the prebuffer.
 */

void flushpcm(int fd)
{
		if (prebufnow) {
			write(fd, &prebuffer[0], prebufcnt);
			prebufnow = prebufcnt = 0;
		}
}

/*********************************************************************
*****/
```

```
void usage(int rc)
{
	printf("usage: mp3play [-hmvqz8RPTZ] [-g <quality>] [-s <time>] "
		"[-d <device>] [-w <filename>] [-B <prebuf>] "
		"[-l <line> [-t]] mp3-files...\n\n"
		"\t\t-h	this help\n"
		"\t\t-v	verbose stdout output\n"
		"\t\t-q	quiet (don't print title)\n"
		"\t\t-m	 mix both channels (mono)\n"
		"\t\t-8	play 8 bit samples\n"
		"\t\t-R	 repeat tracks forever\n"
		"\t\t-z	shuffle tracks\n"
		"\t\t-Z	 psuedo-random tracks (implicit -R)\n"
		"\t\t-P	 print time to decode/play\n"
		"\t\t-T	 do decode, but output test tone\n"
		"\t\t-g <quality>  decode quality (0,1,2)\n"
		"\t\t-s <time>     sleep between playing tracks\n"
#ifdef SWAP_WD
		"\t\t-w <device>   audio device for playback\n"
		"\t\t-d <filename> write output to file\n"
#else
		"\t\t-d <device>   audio device for playback\n"
		"\t\t-w <filename> write output to file\n"
#endif
		"\t\t-l <line>     display title on LCD line (0,1,2) (0 = no title)\n"
		"\t\t-t <line>     display time on LCD line (1,2)\n"
		"\t\t-B <prebuf>   size of pre-buffer\n");
	exit(rc);
}

/*******************************************************************
*****/

int main(int argc, char *argv[])
{
	unsigned long	us;
	INT16		*pcm[MPEGDEC_MAX_CHANNELS];
	int		pcmcount, rawcount;
	int		c, i, j, dspfd, dsphw, slptime;
	int		count = 0;
	int		pcmsize=0;
	int		k=0;
	int		playbits, playstereo, repeat, printtime;
	int		argnr, startargnr, rand, shuffle;
	char		*device, *argvtmp;
```

```
time_t          sttime;

verbose = 0;
quiet = 0;
playstereo = 1;
playbits = 16;
quality = 2;
shuffle = 0;
rand = 0;
repeat = 0;
printtime = 0;
slptime = 0;
prebuflimit = 64000;
device = "/dev/dsp";
dsphw = 1;

while ((c = getopt(argc, argv, "?hmvqzt:8RZPTg:s:d:w:l:B:V")) >= 0) {
        switch (c) {
        case 'V':
                printf("%s version 1.0\n", argv[0]);
                return 0;
        case 'v':
                verbose++;
                break;
        case 'q':
                verbose = 0;
                quiet++;
                break;
        case 'm':
                playstereo = 0;
                break;
        case '8':
                playbits = 8;
                break;
        case 'R':
                repeat++;
                break;
        case 'z':
                shuffle++;
                break;
        case 'Z':
                rand++;
                repeat++;
                break;
        case 'P':
                printtime++;
```

```
			break;
		case 'T':
			testtone++;
			break;
		case 'g':
			quality = atoi(optarg);
			if ((quality < 0) || (quality > 2)) {
				fprintf(stderr, "ERROR: valid quality 0, 1 "
					"and 2\n");
				exit(1);
			}
			break;
		case 's':
			slptime = atoi(optarg);
			break;
		case 'd':
			device = optarg;
#ifdef SWAP_WD
			dsphw = 0;
#endif
			break;
		case 'w':
			device = optarg;
#ifndef SWAP_WD
			dsphw = 0;
#endif
			break;
		case 'l':
			lcd_line = atoi(optarg);
			break;
		case 't':
			lcd_time = atoi(optarg);
			break;
		case 'B':
			prebuflimit = atoi(optarg);
			if ((prebuflimit < 0) || (prebuflimit > (1*1024*1024))){
				fprintf(stderr, "ERROR: valid pre-buffer range "
					"0 to 1000000 bytes\n");
				exit(1);
			}
			break;
		case 'h':
		case '?':
			usage(0);
			break;
		}
```

```
	}

	argnr = optind;
	if (argnr >= argc)
		usage(1);
	startargnr = argnr;

	mp3_buffer = (INT8 *) malloc(MP3_BUF_SIZE);
	if (!mp3_buffer) {
		fprintf(stderr, "ERROR: Can't allocate MPEG buffer\n");
		exit(0);
	}

	for (i = 0; (i < MPEGDEC_MAX_CHANNELS); i++) {
		pcm[i] = malloc(MPEGDEC_PCM_SIZE * sizeof(INT16));
		if (!pcm[i]) {
			fprintf(stderr, "ERROR: Can't allocate PCM buffers\n");
			exit(1);
		}
	}

	if ((rawbuf = malloc(MPEGDEC_PCM_SIZE * sizeof(short) * 2)) == NULL) {
		fprintf(stderr, "ERROR: Can't allocate raw buffers\n");
		exit(1);
	}
	if (prebuflimit) {
		prebufsize = prebuflimit + (MPEGDEC_PCM_SIZE*sizeof(short)*2);
		if ((prebuffer = malloc(prebufsize)) == NULL) {
			fprintf(stderr, "ERROR: Can't allocate pre-buffer\n");
			exit(1);
		}
	}

	/* Make ourselves the top priority process! */
	setpriority(PRIO_PROCESS, 0, -20);
	srandom(time(NULL));

	/* Open the audio playback device */
	/*
	if ((dspfd = open(device, (O_WRONLY | O_CREAT | O_TRUNC), 0660)) < 0) {
		fprintf(stderr, "ERROR: Can't open output device '%s', "
			"errno=%d\n", device, errno);
		exit(0);
	}
*/
	/* Open LCD device if we are going to use it */
```

```
/*
        if ((lcd_line > 0) || (lcd_time > 0)) {
                lcdfd = open("/dev/lcdtxt", O_WRONLY);
                if (lcdfd < 0) {
                        lcd_time = 0;
                        lcd_line = 0;
                }
        }
*/
nextall:
        /* Shuffle tracks if slected */
        if (shuffle) {
                for (c = 0; (c < 10000); c++) {
                        i = (((unsigned int) random()) % (argc - startargnr)) +
                                startargnr;
                        j = (((unsigned int) random()) % (argc - startargnr)) +
                                startargnr;
                        argvtmp = argv[i];
                        argv[i] = argv[j];
                        argv[j] = argvtmp;
                }
        }

nextfile:
        if (rand) {
                argnr = (((unsigned int) random()) % (argc - startargnr)) +
                        startargnr;
        }

        mpa_ctrl = mpa_defctrl;
        mpa_ctrl.bs_access = &bs_access;
        if (playstereo == 0)
                mpa_ctrl.layer_3.force_mono = 1;
        mpa_ctrl.layer_3.mono.quality = quality;
        mpa_ctrl.layer_3.stereo.quality = quality;

        mp3_buffer_offset = 0;
        mp3_buffer_next_block = 0;

        mp3_filename = argv[argnr];

        /* Open file or stream to mp3 data */
        if (strncmp(mp3_filename, "http://", 7) == 0) {
                mp3_stream_size = 0; /*HACK*/
                mp3_fd = openhttp(mp3_filename);
        } else {
```

```
		mp3_stream_size = getstreamsize();
		mp3_fd = open(mp3_filename, O_RDONLY);
	}

	if (mp3_fd < 0) {
		fprintf(stderr, "ERROR: Unable to open '%s', errno=%d\n",
			mp3_filename, errno);
		http_streaming = 0;
		goto badfile;
	}

	getmp3taginfo();

	/* Get first part of the stream into a ram buffer */
	getnextbuffer();

mp3_restream:
	mps = MPEGDEC_open(mp3_filename, &mpa_ctrl);
	if (!mps) {
		fprintf(stderr, "ERROR: Unable to open MP3 Audio "
			"stream '%s'\n", mp3_filename);
		http_streaming = 0;
		goto badfile;
	}

#ifdef CONFIG_USER_SETKEY_SETKEY
	if ((i = getdriverkey(&key, sizeof(key))) > 0)
		MPEGDEC_setkey(mps, &key, i);
#endif

	printmp3info();

	gettimeofday(&tvstart, NULL);
	sttime = time(NULL);

	/* Restart pre-buffering for next track */
	prebufnow = prebuflimit;
	prebufcnt = 0;

	//short *audioOut = na_audio;	//AUDIO_BASE : 0x01b02000
	//f=fopen("/mnt/Huang2.wav", "wb");

	//WAVE header calls
	//	unsigned long cnt;
	//	cnt = calc_count();
	//	start_wave(f, cnt);
```

```
/* write the 'RIFF' header */
//fwrite("RIFF", 1, 4, f);
/* write the total size */
//temp32 = 0;
//wav->total_offs = ftell(wav->file);
//fwrite(&temp32, 1, 4, wav->file);

while ((pcmcount = MPEGDEC_decode_frame(mps, pcm)) >= 0) {

        /*
        for (i = 0; i < pcmcount; i++) {
                //audio_data = inw(na_sram_0);
                int addr = pcm[0] + (i * 0x00000010);
                printf("ADDR=%d\n",addr);
                //printf("i=%d\n",i);
                outw(inw(addr), na_audio);
                for (count = 0; count < DELAY2; count++);
}
*/
volatile INT16 *wavOutput = na_audio+2;
for (i=0; i<pcmcount; i++) {
        *wavOutput = pcm[0][i];  //>>16
        //pcm[0][i]= ( (pcm[0][i]>>8) & 0xFF ) | ((pcm[0][i]<<8) & 0xFF );
        // *wavOutput = pcm[0][i];
        // *wavOutput = pcm[1][i];
        //printf("&pcm[0][i]=%X pcm[0][i]=%X \n",&pcm[0][i],pcm[0][i]);
        //printf("&pcm[1][i]=%X pcm[1][i]=%X \n",&pcm[1][i],pcm[1][i]);
        //printf("channels=%d",channels);
        //f=fopen("/mnt/freeza.wav", "wa");
        //fwrite(pcm[0][i], sizeof(INT16), 1, f);

}
     //fwrite(pcm[0],1152,1,f);

                //printf("pcmaddr=%d\n",&pcm);
                //printf("pcm[0]=%d\n",pcm[0]);
                //printf("k=%d\n",k);
                //printf("Stereo=%d\n",stereo);
                //printf("bits=%d\n",bits);
                //printf("pcmcount=%d\n",pcmcount);
                //printf("rawcount=%d\n",rawcount);
                //pcmsize=sizeof(pcm);
                //printf("pcmsize=%d\n",pcmsize);
                //k=k+1;
```

```
	}
	//fclose(f);
	//printf("Finsih");



	/* Flush out any remaining buffer PCM data */
	//flushpcm(dspfd);

	gettimeofday(&tvend, NULL);
	if (printtime) {
		us = ((tvend.tv_sec - tvstart.tv_sec) * 1000000) +
		   (tvend.tv_usec - tvstart.tv_usec);
		printf("Total time = %d.%06d seconds\n",
			(us / 1000000), (us % 1000000));
	}

badfile:
	if (slptime)
		sleep(slptime);

	close(mp3_fd);
	MPEGDEC_close(mps);
	mps = NULL;

	if (++argnr < argc)
		goto nextfile;

	if (repeat) {
		argnr = startargnr;
		goto nextall;
	}
/*
	close(dspfd);
	if (lcdfd >= 0)
		close(lcdfd);
*/
	exit(0);
}

/**********************************************************************
*****/
```