Approach to Real Time Encoding of Audio Samples

A DSP Realization of the MPEG Algorithm

ME 235
Professor Steve Kraft
University of California Berkeley
Spring 2002

Brian Loker ? Robin Liu

1 INTRODUCTION

Since the invention of high speed microprocessors in the early 1990's, MPEG-1 layer 3 encoding (MP3) has become the most prolific encoding process for audio transferred over high speed networks. The Fraunhofer Institute and Thomson Multimedia created the popular coding scheme in the late eighties. The International Standards Organization (ISO) then standardized the process into three parts known as MPEG-1. This standard includes details about the bit streams for audio, video, and packaged systems, along with the tools used to encode and decode these bit streams.

Today, MP3 encoded files are as common as CDs, and most people connected to the Internet have had experience downloading and listening to MP3 files. This following makes the use of MP3 files in any transportable audio intriguing. The amazing compression ratios that can be achieved using the algorithm are indispensable to keeping memory requirements low. Smaller memory requirements can mean larger capacity for music or cost savings in hardware. Both of which are huge advantages to a commercial application of this technology.

We had proposed to create a real-time MP3 encoder using the Analog Devices DSP and audio codec provided on the EZ-Kit lab board. We began writing code that could do this, but soon found that to process the audio samples in real-time might not be possible using this board and the algorithms specified in the ISO specification of MPEG layer 3 audio. We then changed our implementation to be non real-time. Using the board we chose to collect audio samples and store them in SDRAM collecting 10 seconds of audio and then processing the samples collected. This would allow us to write and debug the basic encoder and then implement faster algorithms to make the encoding process faster than the sampling rate. If we could achieve this then we could create a circular buffer for the incoming audio samples and encode them in real-time. The basic process flow chart of our project is as follows, in Figure 1:

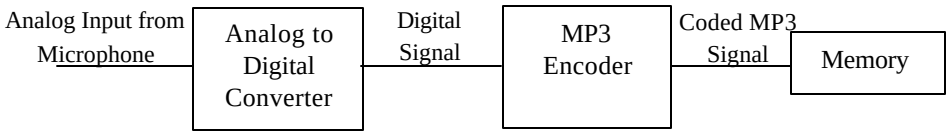


Figure 1: Basic Project Implementation

2 MP3 BITSTREAM

2.1 INTRODUCTION

Every MP3 file is composed of many parts, each of which is called a frame. Every frame in the file is an encoded chunk of 1152 time domain audio samples. These samples are then divided into two granules each, resulting in 576 samples. Every frame consists of a header, side info, audio data, and ancillary data. The header frame is 32 bits long. For an MPEG-1 mono audio file, the side frame is 17 bytes long. The audio data and ancillary data are variable length, but the average length of these areas can never exceed the length available specified by the encoded bit rate.

2.2 HEADER

The header block consists of 13 values corresponding to tables that give various data about the file. Figure 2 shows how the header frame is structured. The first 12 bits of the frame

Position	Purpose	Length (in Bits)
A	Frame sync	12
B	ID (1 for MPEG, 0 reserved)	1
C	MPEG layer (Layer I, II, III, etc.)	2
D	Protection (if on, then checksum follows header)	1
E	Bit rate index (lookup table used to specify bit rate for this MPEG version and layer)	4
F	Sampling rate frequency (44.1kHz, etc., determined by lookup table)	2
G	Padding bit (on or off, compensates for unfilled frames)	1
H	Private bit (on or off, allows for application-specific triggers)	1
I	Channel mode (stereo, joint stereo, dual channel, single channel)	2
J	Mode extension (used only with joint stereo, to conjoin channel data)	2
K	Copyright (on or off)	1
L	Original (off if copy of original, on if original)	1
M	Emphasis (respects emphasis bit in the original recording; now largely obsolete)	2

Figure 2: Header Frame Structure (from MP3 Tech website).

are required as sync bits, with all twelve set to '1'. The next bit, in the 'B' position will need to be set to '1', for MPEG format identification. The next two bits, in the 'C' position of the frame, will be set to layer 3 ('01'). The 'D' position bit determines whether a checksum is used to check for error in transmission or encoding. We will not use error checking so this bit should be set to '1'. The bit rate index, position 'E' should be set to 64 kb/s ('0101'). We will sample the audio at the sampling frequency of 32 kHz used on the AC '97 codec provided on the EZ-Lab board, so the sampling rate bits, position 'F' bits, should be set to '10'. Padding the bits is only necessary with a sampling frequency of 44.1 kHz so the padding bit, position 'G' should always be set to '0'. The position 'H' bit, the private bit, will be set to '0'. The channel mode, position 'I', should be set to mono ('11'). Since we are not encoding in joint stereo, the mode extension is not used so we can set the position 'J' bits to '00'. There is no copy write, so the 'K' position bit will be set to '0'. The original bit, position 'L' bit, should be set to '1'. Finally, the emphasis bit, 'M' bits, will be set to no emphasis ('00'). Thus our header blocks should remain constant over the whole file and take the following form:

Header bits: 1111 1111 1111 1011 0101 1000 1100 0100

2.3 SIDE INFO

For a mono encoded MP3 file, the side information block is 136 bits (17 bytes) long. This side block contains information about how the samples are encoded, including information about whether the channels share scale factors (stereo modes), the Huffman tables used for coding, the pointer to the end of the main data, and so forth. The side information is set up with the first bits used for information pertaining to both granules. Next there are two sections, with each containing bits pertaining to one of the granules.

The shared bits that start the side information block are main_data_end (9 bits), private bits (5 bits), and scale factor selection information (4 bits), totaling 18 bits. The remaining 118 bits are divided equally by the two granules. Main_data_end specifies the negative byte offset from the next frame header to the end of the current frame data. Private bits are bits used by Fraunhofer for private flags and are not needed (set to '00000'). The scale factor selection information (scfsi) is set for each scale factor band specified in a table. '0' is used for transferring scale factors for each granule and '1' is set if the scale factors for the first granule can be reused for the second granule.

Each granule's side information is coded in the following manner. The first 12 bits give the length of bits used for scale factors and Huffman encoded values. The next 9 bits give the number of pairs of spectral values exceeding an absolute value of 1. In other words, in the 576 values returned by the quantization, the nine bits give half the amount of bits at which all

remaining bits in the 576 point granule are less than the absolute value of 1. Next, 8 bits are used to set the global gain used for requantizing the values in the granule. The next four bits are used to select the number of bits used to transmit the scale factors. The following bit sets a flag to signal a non-normal windowing. The next 22 bits depend on how the block is split, and specify the Huffman tables used for each of the three regions in the "big_values" area of the granule. The final three bits for each granule specify additional amplification of higher frequencies, a scale factor for quantization, and which Huffman table to use for the "count_1" area of the granule.

After the above 136 bits in mono encoding, the main coded audio data will follow. The size of this data can be calculated because we know the negative offset from the next frame of the end of this data, and the total bits per frame is constant depending on the bit rate. Figure 3 shows the format of the granule partitioning.

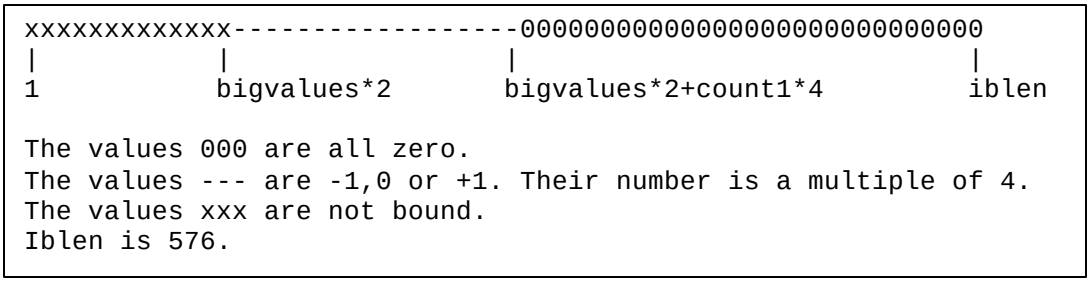


Figure 3: Granule Partitioning (from ISO/IEC IS11172-31723).

2.4 AUDIO DATA

The length of the audio block is variable depending on how many scale factors need to be coded and how much the data can be compressed using the Huffman tables. For each granule, the structure of the audio block is to code the scale factors and then code the Huffman bits that result from the encoding of samples. The main difference between layer 3 coding and layers 1 and 2 is that the audio data does not start immediately following the side information. In fact for a given frame, the audio data may start before the frame header is even received. This makes use of the "bit reservoir" available in the layer 3 specification. This reservoir lets a frame with little information be coded with less data and a following frame containing more information than could normally be coded within the frame at the given bit rate use the leftover bit space. This makes better use of the time dependence characteristics of an audio file to spectral resolution. Figure 4 shows the general bitstream organization, with all the sections discussed above.

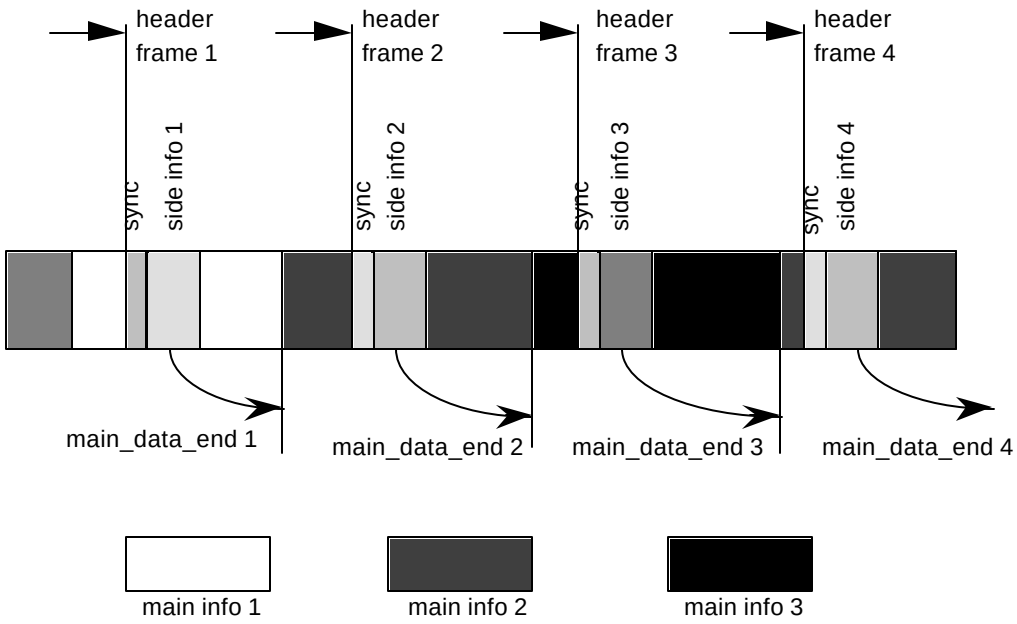


Figure 4: MPEG-1 Layer III Bitstream Organization (from ISO/IEC IS11172-31723).

3 ENCODING

3.1 INTRODUCTION

The encoding algorithm for MP3 is classified as perceptually lossless. This is because the encoding using the psychoacoustic model is lossy, but focuses losses outside of the human hearing ranges. The psychoacoustic model uses two acoustic properties to reduce data. Once the data has been reduced to only what is necessary, a lossless coding process is used to further compress the data. The coding is done using a set of Huffman tables specified in the MPEG-1 standard.

Figure 5 shows a flowchart of a simple MPEG/audio encoder. The digitized audio samples are sent through two different filters, the polyphase filterbank and a Fast Fourier Transform (FFT) for the psychoacoustic model. The reason for filtering the signal with two different filters is to get different time or frequency

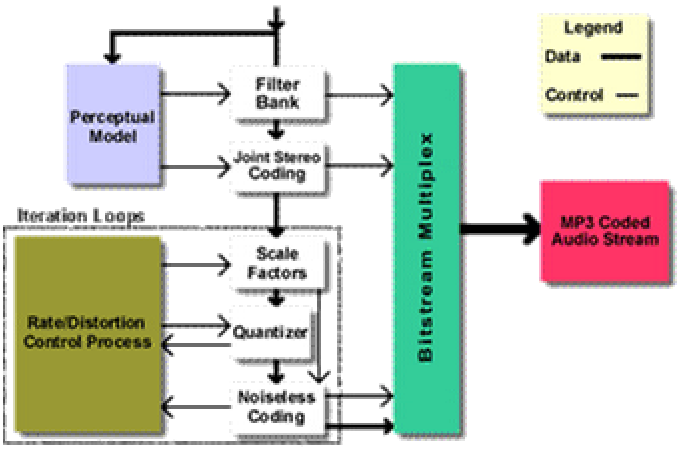


Figure 5: Encoder Flowchart (from Fraunhofer website).

resolutions for the different calculations. The outputs of these two filters are used to do noise allocation and quantization of samples. Finally the quantized values are encoded and the bit stream is formatted for output.

3.2 FILTERBANK

The polyphase filterbank transform uses a 512 point vector to do calculations and results in 32 output samples. In each subband, 36 consecutive outputs from the filterbank are collected and a modified discrete cosine transform is performed to give the frequency spectra of each subband. Each subband has a bandwidth equal to 1/32 of the Nyquist frequency. Figure 6 gives a sample of the calculations used to perform the filterbank transform. This is a very intensive calculation, but there are many algorithms that can be employed to reduce the number of multiplications and additions.

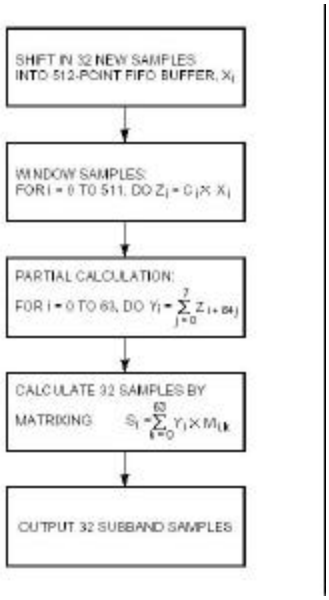


Figure 6: Typical Filterband Transform (from Pan, 1993).

3.3 PSYCHOACOUSTIC MODEL

For layer 3, the second psychoacoustic model given in the MPEG standard is generally used. However, any model can be employed for any encoder or the model can be avoided almost altogether. As a result, this allows encoders to be very different in implementation and behave with very different characteristics. The two main theories behind the model are as follows. The first is that our range of hearing is bounded within a certain bandwidth. Because of this, all signals outside of this bandwidth are discarded. Also, the threshold of intensity for a signal to be audible is frequency dependent. Therefore at very low and very high frequencies a signal needs to be stronger than at mid range frequencies for the human ear to pick it up. Another fact used when

encoding stereo but not for our project, is that low and high frequency sounds lose there spatial dependence. Therefore at these frequencies, a stereo signal can disregard many differences between the two channels. The second characteristic of human hearing that is used is something called signal masking. A strong signal will mask weaker signals that are close to it in frequency, but will not mask the same weak signal if there is a big enough frequency difference. One implementation of this model involves the calculation of two different FFTs. One FFT has a shift length of 576 samples and a window size of 1024, while the second FFT has a shift length of 192 samples and a window size of 256. The shorter block FFT is calculated three times for every one time the longer block FFT is calculated. The results from these FFTs can be used to calculate signal to mask ratios (SMR). These ratios, along with a table of quiet threshold values can be used to discard signals that are not clearly perceptible in the signal. Our encoder does not implement a psychoacoustic model at this time. Without the implementation of a psychoacoustic model the output from our encoder will likely have audible noise. An additional problem of the psychoacoustic models is that they are also very computationally intensive.

Figure 7: Thresholds of Sound Pressure

3.4 QUANTIZATION AND CODING

The quantization and coding of each of the granules is done using two nested iteration loops. The outer iteration loop is called the noise allocation loop and the inner iteration loop is called the quantization loop. The quantizer loop starts with step size of zero and increases the step size until the output vector can be coded within the given number of bits. When this is true the noise loop checks to make sure the distortion for each of the 32 subbands is below the allowed distortion. If it fails this test, the scale factor band is amplified and the inner loop is recalled. The Huffman tables used in the rate loop are chosen based on the largest quantization value in a given subregion. Various algorithms for choosing these tables are available.

Layer 3 also makes use of a bit reservoir. If a quantization uses more bits than the number of bits allowed in a frame, the frame can borrow bits from the bit reservoir. After the two loops have returned to the main process, the bit stream can be output to a buffer. If the bit reservoir becomes too large, or audio samples have stopped being collected, the remaining space not used by the bit reservoir should be padded with zeros, and all remaining frames in the output buffer should be written to whatever memory we choose to use in our hardware design.

4 HARDWARE

4.1 AUDIO SAMPLING

We will be using the provided audio sampling device on the EZ-Kit board. The device is the Analog Device's AD1819A SoundPort. A microphone can be attached to a port on the board and this device is configured to record audio samples at a rate of 32 kHz. The codec is connected to the SHARC processor via its second serial port. The processors control registers can then be set to have the audio samples transferred using DMA so the processor's bandwidth is not taxed as much as it would be if using a strict interrupt driven process. Thus multiple audio samples can be handled per interrupt driven by the AD1819A.

4.2 PROCESSOR

The SHARC DSP board used in this project is an Analog Devices ADSP-21065L EZ-KIT. The speed of the 32-bit DSP is 60 millions instructions per second (MIPS), which is not fast enough to allow real-time encoding in MP3 format using our slow algorithms.

4.3 MEMORY

Memory requirements for this project are very taxing to the EZ-Kit board. Provided on the board are 544 kb of internal user memory and 1 Mb of 32 bit memory spaces in external SDRAM. We will implement all of block 0 in internal memory to be 48 bit wide program memory. All of our code will need to fit in this memory. In block 1 we will implement 32 bit wide data memory storage for data memory. Block 1 will also contain the stack and the heap for our program. The audio samples, as well as all filter coefficients and data arrays will be stored in the external memory in SDRAM. Once we run out of program memory we will try to add 48 bit addresses in Block 1 and move any 32 bit addresses out of block 1 and into SDRAM as needed.

5 SOFTWARE

5.1 STRUCTURE

The general structure of the software implementation of our project is laid out in Figure 8. The input signal, changed from analog to digital format by the AD converter, is sent through two different paths. One stream goes through the filterbank, which discretizes the bit stream into 32 subbands. A modified discrete cosine transform (MDCT) is then performed on each of these subbands, slipping each band into 18 frequency lines, resulting in 576 output samples. In parallel, the other input signal is through a 1024 point Fourier Fast Transform, then to the psychoacoustic model. The model analyzes the audio signal and calculates the masking threshold and signal to mask ratios for each subband. The output from the two data filters is then sent to the iteration loops, which quantize the input vector to meet the demands and output the vector of quantized values, along with other relevant information. The values are then encoded using the Huffman algorithm, while the side information is coded in a parallel process. Finally, the results are formatted and packaged into the standard frame format and the coded signal is ready to be written to memory.

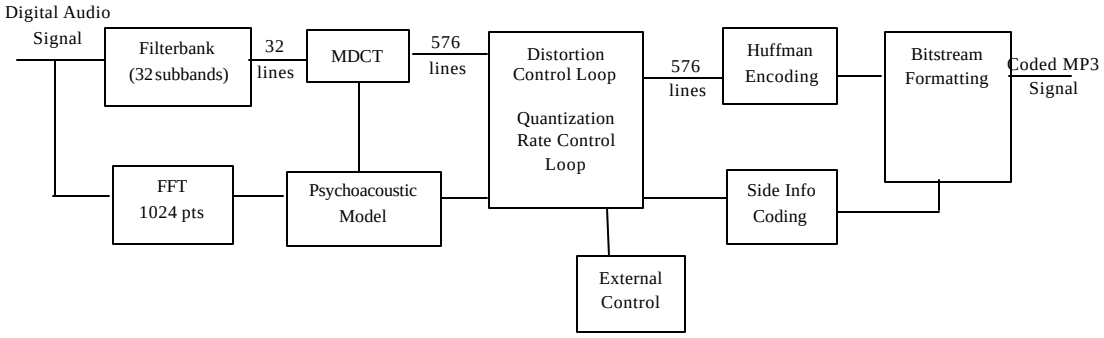


Figure 8: Software Structure

5.1.1 Filterbank

The basic process for the filterbank is shown above in Figure 6. The equation for the filterbank output is as follows:

$$s_t[i] = \sum_{k=0}^{63} \sum_{j=0}^{7} M[i][k](C[k + 64j]x[k + 64j])$$

$$M[i][k] = \cos\left(\frac{(2i + 1)(k + 16)\pi}{64}\right)$$

where i is the subband index (ranging from 0 to 31); $s_t[i]$ is the filter output for subband i at time t, where t is an integer multiple of 32 audio sample intervals; C[n] is one of 512 coefficients of the analysis window defined by the standard; x[n] is an audio input sample read from a 512-sample

buffer; and M[i][k] are known as the analysis matrix coefficients (Pan, 1995). Since C[n] are known coefficients, we can just store these values in memory and use a look-up table to reference the values when needed for calculation.

5.1.2 Psychoacoustic Model

As stated above, the psychoacoustic model reduces the data by determining the quiet threshold and corresponding signal to mask ratio. The basic steps in the second psychoacoustic model are as follows, as described in ISO/IEC IS11172-31723:

1. Reconstruct 1024 samples of input signal
2. Calculate the complex spectrum of the input signal
3. Calculate the unpredictability measure
4. Calculate the energy and unpredictability in the threshold calculation partitions
5. Convolve the partitioned energy and unpredictability with a spreading function
6. Calculate required SNR for each partition
7. Calculate power ratios and actual energy threshold
8. Spread the threshold energy over the FFT lines
9. Yield final energy and threshold of audibility
10. Calculate signal-to-mask ratios

Once the threshold calculation partitions are known, the corresponding scalefactor bands can be determined from a table.

Due to time restrictions for this project, we did not implement any of these for our project. We ran into trouble with the subband filterbank outputs and without clear inputs for this to work with, the psychoacoustic model would be useless. Once the filterbank works, a psychoacoustic model could be implemented which uses as many of the above steps as possible.

5.1.3 Iteration Loops and Encoding

The iteration loops are broken up into one main loop containing two smaller nested loops, an inner loop and outer loop. The inner loop is known as the quantization rate control loop and the outer loop is the distortion control loop. Figure 9 shows flow charts for all three loops.

For the general function, inputs, and outputs can be seen in the general iteration loop. A vector of spectral data is inputted, along with the coding demands, including the allowed distortion for the scalefactor bands, the number of scalefactor bands, and the bits available for Huffman and scalefactor coding. As long as the input vector contains at least one non-zero spectral value, the outer iteration loop is run. The outer loop calls the inner iteration loop, and checks the distortion of each scalefactor band. If the allowed distortion is exceeded, the scalefactor band is amplified and the inner loop is called again. This process continues until the constraints are met.

The quantization rate control loop, the inner loop, quantizes the input vector, and checks to see if the maximum of this quantization is within the allowable range. If not, the quantizer step size is increased, and the quantization is repeated. Once the quantization falls within the allowable range, the resulting values are manipulated and then coded using Huffman tables. There are 18 different tables that can be used and most data is encoded using two-dimensional tables. Again, these values will be stored in memory and referenced when needed. After the Huffman coding, the number of coded bits is counted and checked against the bits available for Huffman coding. If this value is exceeded, the quantizer step sized in increased, and the whole quantization process is repeated.

After the whole process is complete, all three iteration loops finished, the output is fed to the formatting and packaging block. The output data includes the vector of quantized values, the scalefactors, quantizer step size information, the number of unused bits available for later use, Huffman side information, the number of pairs of Huffman coded values, and the Huffman code table of regions.

The equation for quantizing values can be found in the ISO MPEG-I standard as well as the Huffman code tables. The algorithm for choosing a Huffman code table depends on the maximum value of quantized data. For a given maximum value of the pair of values, one, two or three tables are available each better for use with different signal characteristics. For our implementation we will sacrifice a little bit of sound quality by only utilizing one table for each maximum quantized value. By doing this we eliminate the need to code the data with multiple tables and then compare the length of coded data to determine the optimal table.

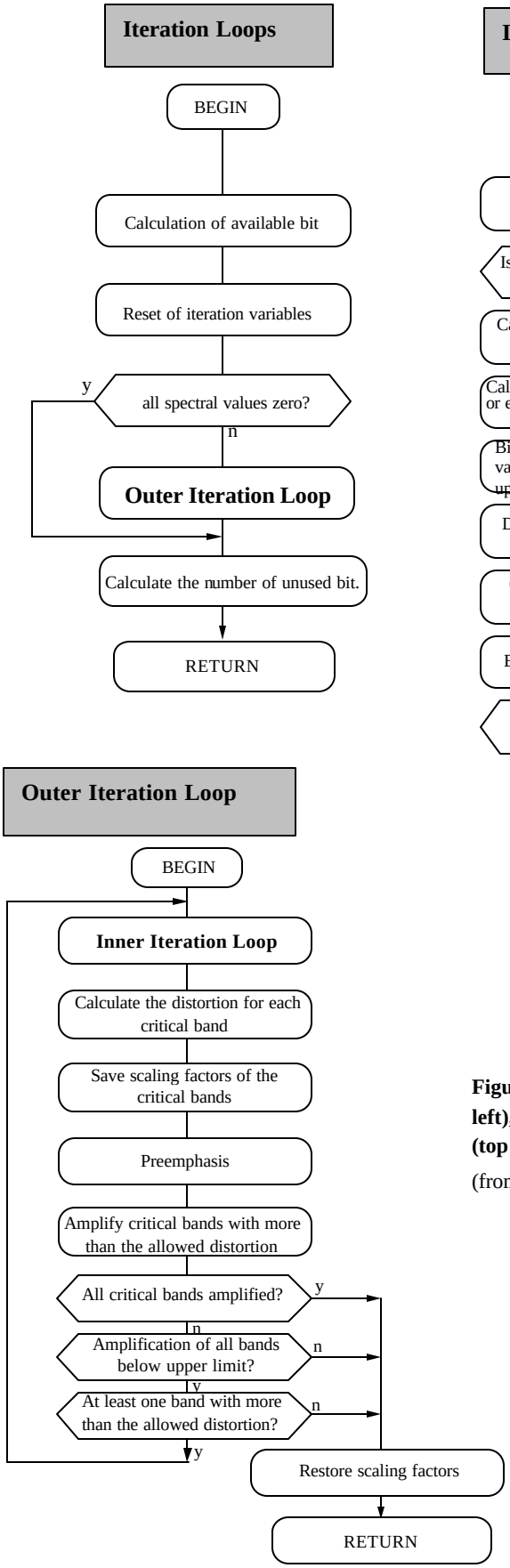


Figure 9: Flow chart for general iteration loops (top left), outer iteration loop (left), and inner iteration loop (top right).

(from ISO/IEC IS11172-31723).

5.1.4 Formatting and Packaging

Formatting and packaging of the data is the last step in the encoding process. The quantized and encoded data, along with the side information is put into the correct format. The organization of the bitstream is described in the audio data section and shown in Figure 4. The standard header frame, as described in MP3 bitstream section is included and the file is ready to be stored.

5.1.5 Loops

The main program will take care of all the initialization procedures and then loop and wait until interrupt requests are received to process the data. Data will be processed 576 audio samples at a time. Each block of 576 samples comprises one granule. Two granules are packaged in each frame.

5.2 TIME REQUIREMENTS

Due to time requirements of a real-time encoding system, we could not accomplish the many calculations necessary before the next samples were ready to be processed. By realizing this we switched to a non-real-time implementation that would allow for easier prototyping of our encoder. Because our encoder is not working properly and the full implementation was not completed we could not get accurate timing data for this report.

6 IMPLEMENTATION

6.1 Software

We began our project by realizing we could not use the serial monitor to initialize the SDRAM and CODEC because we would need the program space the monitor used up. The first part of our project involved getting the hardware initialized after a reset and begin an interrupt service routine for storing audio samples. We used the assembly language source code provided by Analog Devices to begin writing our program. We only used what we needed of their source code and interfaced it with our C program. Using these functions, we initialize the DSP, initialize the SDRAM, initialize the serial port, setup DMA transfers on the serial port and initialize registers in the AD1819 audio codec. These functions work properly and the DMA generates an interrupt every time there is a new audio sample.

The first decision we made was how to represent each of the samples. In order to make use of the math functions for floats, we had to cast the integers as floats. Bit shifting was performed, since the samples from the codec are 16 bits wide and the integers stored by the DMA

transfer are 32 bits wide. We shifted the bits to the most significant 16 bits in the integer and then divided by a scaling factor, so that our samples retained their sign and fell in the range of –32768 and 32767. With each sample we check if it is a valid sample and if it is store it in SDRAM. Once we had collected a predefined number of samples, we set a flag to begin processing the data.

When our infinite loop sees this flag is set, it begins encoding the data and writing the results over the same memory space. The first processing that is done is to put the samples through the polyphase filterbank. We take 512 samples of audio, window it by the coefficients C(n), perform a partial calculation, and then matrix that result by a cosine matrix. This gives us 32 subband samples that we store and concatenate with the previous 36 filter outputs. Once we have calculated the subbands, we transform the time domain samples into frequency domain samples using the modified discrete cosine transform (MDCT). This gives us the 576 spectral lines for the granule.

Once the spectral lines were calculated, we enter the iteration loops. The outer loop (quantize_data()) calls the inner loop (rate_loop()) to begin the iteration. The inner loop takes the frequency lines and scales them until the magnitude of every line is below 8207. Then the outer loop would check the number of bits needed to code these lines with the available bits for coding the frame. If it is within range, the encoder would save the frequency lines and the global gain used to scale the original samples. Once two granules have been encoded, the two sets of frequency lines are packed into an output bitstream and written over the original samples in SDRAM.

We believe that we could have implemented most of this had we not run into some major problems with the subband filterbank. The equations are given to us in the MPEG specification, so we did not think this would give us such problems, but it did. We used the slow algorithm explained in the ISO document and got results we could not resolve before the end of the semester. The filterbank works great when we sample a single tone sine wave at the codec and we get a spike in the frequency spectrum at the correct line in the quantized samples. However, for some reason we also get a spike at the DC term in the subband MDCT transform. Looking at the subband outputs before the MDCT is performed, we see that the signals appear to have a DC offset. If we run the program multiple times with the same input, we get different magnitudes of this offset. This is probably due to the phase difference at the start of recording of the sine wave in subsequent resets of our program. We could not figure out where this was coming from, so we wrote the same subband filterbank program in Matlab. We input the same audio samples that were stored in the SDRAM and the Matlab program results were as expected. There were no spikes in the DC term of the transform. We looked at the Matlab code and at our DSP C code and could not find any differences. Our conclusion is there might be a bug in the compiler involving the use floats in the manner we used them. One solution might be to change the calculations to doubles and see if this would fix anything. We cannot imagine why this would change anything,

but we found many bugs in the compiler where the correct C code did not work correctly on the DSP. One significant bug we found was the inability of the compiler to handle multiple calculations on one line of code. When we combined a statement such as

```
y[i] = c[i] * input[i] * cos_matrix[i][j];
```

the compiler would sometimes give an error when compiling that said it was not a valid statement and thought the * was a pointer reference. Other times the code would compile fine, but we would see weird results in the data. By splitting the statement into multiple lines some of these problems could be resolved.

We suspect there are little bugs like the char bug talked about in lab that we do not know about that are affecting our output. Without a working filterbank we could not get much further in testing our other parts of the program. We set up tables for the quantization and scaling and implemented functions to take our floating point frequency lines and turn them into integer values less than 8207. This appears to work right although there is no checking of distortion in the outer control loop to make our quantizer effective at combating audible noise. We also set up tables for the Huffman coding so that the quantized samples could be coded using less data, and we created functions to pack our quantized samples into the SDRAM as well as writing the header and side information to each frame.

The header is first written for the frame, at the beginning of frame, at the sync location. The side information follows. The encoded data, containing the scalefactors and Huffman encoded samples are then written to memory, starting at the location where the last frame ended, main_data_end from the previous frame, utilizing the bit reservoir described in section 2.4. The length of each Huffman encoded audio sample varies and the encoded data needs to be packed into a 32 bit package before it is written to memory, since each memory address in the SDRAM can store 32 bits. To pack the bits, a function was written which takes an input value along with its length in bits, and adds it to the existing bitstream after shifting it by the correct number of bits. After the length of the bitstream reaches 32 bits, it is written to memory and a new bitstream is created. Using this method, the variable length side information and encoded data are packaged into the correct MP3 format. At first glance these functions appear to work correctly, but they have not been fully debugged because there was less priority to finish these sections until we could get the filterbank working.

6.2 HARDWARE

The only hardware that was part of our project was a ten-dollar battery powered microphone with a monophonic mini plug that attached to the mic-in port on the EZ-Kit board. This worked fine and we could sample this at 32 kHz with no problems.

6.3 DATA

We tested our program on both the DSP and in Matlab. They gave us different results for the same input. We could not explain why it was doing this. Figure 10 shows the data we collected from the DSP. The input audio samples from the codec are correct.

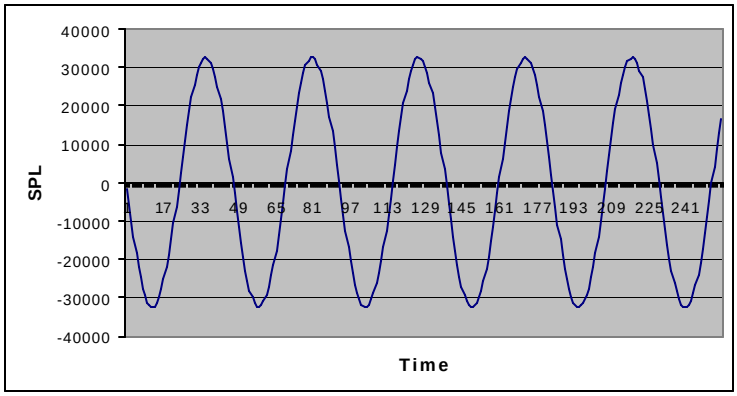


Figure 10: Input Audio Sample

Figure 11 shows the results from the DSP filterbank for a 700 kHz signal. The signal shows up as the negative spike at index 26 but there are also spikes around index 36. Index 36 is the frequency line for 1000 Hz and the DC coefficient of the 2nd subband transform. Figure 12 gives the filterbank results for the 700 kHz sample using Matlab. The absence of the extra frequency lines is obvious. Also notice the difference in magnitude between the two calculations. Figure 13 shos the point in our program where our data gets corrupted. The filtered and down sampled signals found in the subband array should not be offset in either direction from the center unless there is a dc response due to aliasing. An offset would be found in or signals if there were a signal at say 1 kHz where by aliasing effects of down sampling to 1 kHz we should see the presence of a dc term.

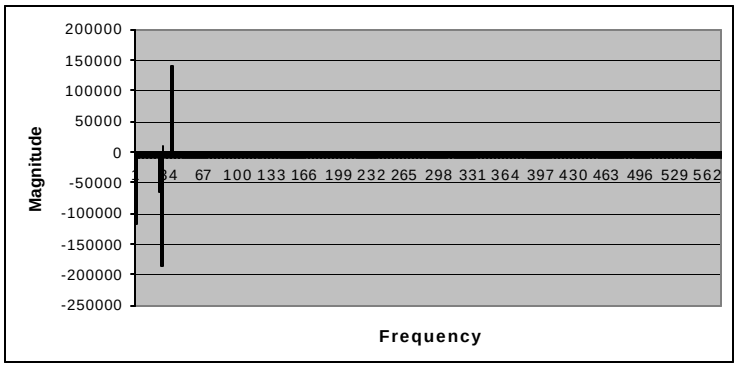
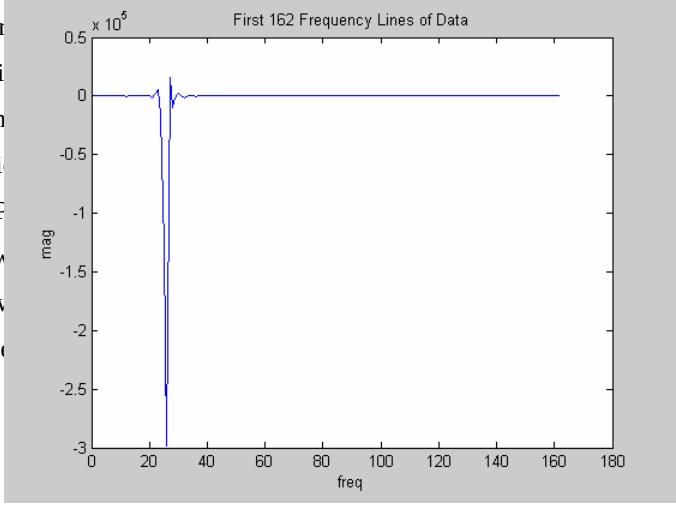


Figure 11: DSP Frequency Lines

Looking at the results of our analysis in Matlab and our program running on the DSP, we could not find the result to the pr se a second tone when the data is bei acy of the correct tone in the frequen uck on this single problem. We tri ods described in many papers on MP the signal to between –1 and .999 w to make sure that our calculations v nue with this project we would have as



doubles and see if the increased precision as well as different implementations of the compiler functions would make a difference in our results.

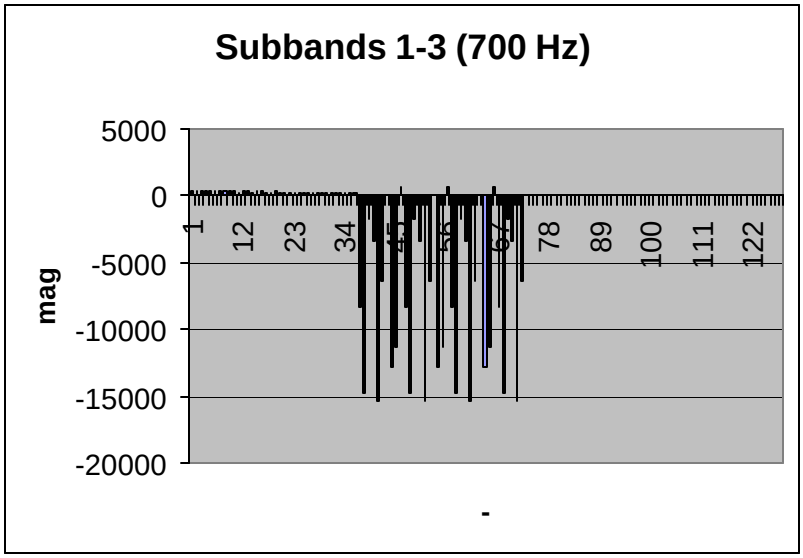


Figure 13: Subband Response DSP

7 APPENDIX A

7.1 Final.c

```
/*********************************************
**
**  Final Project  MP3 Encoder
**
**  final.c
**  Brian Loker, Robin Liu
**
**
**  This file contains the main function.
**  Initializes the DSP and AD1918 Codec and stores
**  the input audio samples.
**
*********************************************/
#include <def21065l.h>
#include <signal.h>
#include <stdio.h>
#include "c.h"

/* external assembly functions */
extern void Init_DSP();
extern void Init_65L_SDRAM_Controller();
extern void Program_SPORT1_Registers();
extern void Program_DMA_Controller();
extern void AD1819_Codec_Initialization();

/* External c functions */
extern void matrix_init();
extern void init_write();

/* Function Declarations */
void toggle_State();
void sample_Audio();

/* External variables */
extern int rx_buf[5];
extern int tx_buf[7];

/* Global variables */
int RECORD_STATE = 0;
int ENCODE_STATE = 0;
int *index;

void main()
{
        // initialize dsp and codec
        Init_DSP();
        Init_65L_SDRAM_Controller();
        Program_SPORT1_Registers();
        Program_DMA_Controller();
        AD1819_Codec_Initialization();

        /* setup interrupt service routines */
        asm("#include <def21065l.h>");
        interrupt(SIG_IRQ2, toggle_State);      // set IRQ2 interrupt routine
        interrupt(SIG_SPR1I, sample_Audio);  // set SPORT1 Receive interrupt
        asm("bit set IMASK IRQ2I | SPR1I;");
```

```
	matrix_init();
	init_write();

	while (1)
	{
		if(ENCODE_STATE) {
			process_Audio();
		}
	}
}

/*  change record state */
void toggle_State()
{
	if (RECORD_STATE != 1) {
		RECORD_STATE = 1;
		index = (int *) 0x03050000;
	} else {
		RECORD_STATE = 0;
	}
}

/* sample and store input audio */
void sample_Audio()
{
	int temp1;

	if (RECORD_STATE == 1) {
		/* Process Sample */
		/* convert to 32 bit fixed point 16 LSB = 0 */
		if (rx_buf[0] == 0xf800) {
			*index = 0;
			temp1 = rx_buf[3];
			temp1 = temp1 << 16;
			*index = temp1 / 65536;
			index++;
		}
		if (index > (int *) 0x03060000) {
			RECORD_STATE = 0;
			printf("Done Recording\n");
			printf("Processing.....\n");
			index = (int *) 0x030501ff;
			ENCODE_STATE = 1;

		}
	}
}
```

7.2 encode.c

```
/*********************************************
**
**  Final Project  MP3 Encoder
**
**  encode.c
**  Brian Loker, Robin Liu
**
**  This file processes and encodes the sample
**  audio.
**
*********************************************/
```

```
#include "scalefac_bands.h"
#include "huffman.h"
#include <math.h>
#include <stdio.h>

/*  Function declarations */
void process_Audio();
void quantize_data();
void rate_control();
void encode_data();
int linbits(int value);

/*  External functions */
extern void sub_filter();
extern void calc_mdct();
extern void init_frame();
extern void add_bits(int value, int length);
extern void write_number(int value, int length);

/*  Global arrays  */
section("seg_arr") float fifo[512];
section("seg_arr") float xr[576];
section("seg_arr") int ix[576];

/*  External variables */
extern volatile int *index;
extern volatile int sb_index;
extern volatile float freq[576];
extern volatile int main_data_end;

/*  Global variables  */
int qquant;
bool endofframe;
bool sideinfo, endofside;
bool writetotalbits, finalwrite;
int totalbits;
int granule = 1;


/*  Process 512 input audio samples  */
void process_Audio() {
        int i;

        if (granule == 1) init_frame();

        endofframe = false;
        for(i=0;i<512;i++) {
                fifo[i] = *index;
                index--;
        }
        index += 544;
        sub_filter();
        if(sb_index < 0) {
                calc_mdct();
                quantize_data();
                encode_data();
        }
        if (granule == 1) granule = 2;
        else granule = 1;
}

/*  Quantize audio sample subbands  */
```

```
void quantize_data() {
	int i, j;
	sb_bands *cur_band;

	for(i=0;i<21;i++) {
		*cur_band = SB_Band[i];
		for(j=cur_band->start_index;j<cur_band->end_index + 1;j++) {
			xr[j] = freq[j];
			xr[j] *= powf(2, scalefac[i]);
		}
	}
	for(i=550;i<576;i++) {
		xr[i] = 0;
	}
	qquant = 0;
	rate_control();
}

void rate_control() {
	int i;
	float temp;
	int max_val;

	max_val = 0;
	for(i=0;i<576;i++) {
		temp = fabsf(xr[i]);
		temp /= powf(2, qquant / 4.0);
		temp = powf(temp, .75);
		ix[i] = (int) floorf(temp + .4054);
		if(ix[i] > max_val) {
			max_val = ix[i];
		}
	}
	if(max_val > 8206) {
		qquant++;
		rate_control();
	}
}

/*  Encode data and write to bitstream  */
void encode_data()
{
	int i;
	int x,y,v,w;
	int linbitx, linbity, numlinbitx, numlinbity;

	totalbits = 0;

	// write side info
	endofside = false;
	sideinfo = true;
	if (granule == 1)
	{
		add_bits(main_data_end, 9);  // main_data_end
		add_bits(0,5);          // private bits, set to zero
		add_bits(15,4);  // scfsi, set to same for granule 1 and 2 for all 4
	}
	writetotalbits = true;
	add_bits(0, 12);  // number of huffman and scalefactor bits
	writetotalbits = false;
	add_bits(288, 9);  //  number of bigvalue bits
	add_bits(qquant+64, 8);  // quantizer step size info
```

```
add_bits(15,4);  // num bits used for transmission of scalefactors
add_bits(0,1);  // different windows, set to normal
for (i=0; i<3; i++)
{
        add_bits(24, 5);  // table select for regions 0 to 2
}
add_bits(200, 4);  // addr of border b/w regions 1 and 2
add_bits(400, 3);  // addr of border b/w regions 2 and 3
add_bits(0, 1); // no preflag
add_bits(0, 1); // scalefactors quant w/ step size 2
endofside = true;
add_bits(0, 1);         // select table A for count1table

// write scalefactors and Huffman encoded data
for (i=0; i<21; i++)
{
        add_bits(scalefac[i], 1);
        totalbits++;
}
i = 0;
while (i < 576)
{
        if ((i==574) && (granule == 2)) endofframe = true;
        x = ix[i];
        y = ix[i+1];
        if (x > 14)
        {
                numlinbitx = linbits(x);
                linbitx = x - 15;
                x = 15;
        } else {
                numlinbitx = 1;
                linbitx = 0;
        }
        if (y > 14)
        {
                numlinbity = linbits(y);
                linbity = y - 15;
                y = 15;
        } else {
                numlinbity = 1;
                linbity = 0;
        }
        add_bits(hc_24[abs(x)][abs(y)].hcode, hc_24[abs(x)][abs(y)].hlen);
        totalbits += hc_24[abs(x)][abs(y)].hlen;
        add_bits(linbitx, numlinbitx);
        totalbits++;
        if (x != 0)
        {
                add_bits(sign(x), 1);
                totalbits++;
        }
        add_bits(linbity, numlinbity);
        totalbits++;
        if (y != 0)
        {
                add_bits(sign(y), 1);
                totalbits++;
        }
        i = i + 2;
}
write_number(totalbits, 12);
```

```
}

/*  Returns log base 2 of value */
int linbits(int value)
{
        int i;

        i = 0;
        while (value > 1)
        {
                value /=2;
                i++;
        }
        return i;
}
```

7.3 subband.c

```
/*****************************************************
**
**  Final Project
**    subband.c
**
**  Contains implementations of the subband filter bank.
**
*****************************************************/
#include <math.h>

/*  Constants  */
#define coef1 0.0490873852123f; // pi / 64
#define coef2 0.0872664625997f; // pi / 36
#define coef3 0.0436332312998f; // pi / 72

/*  Global arrays  */
section("seg_arr") float Mat_f[32][64]; // Mat_f[i][k]
section("seg_arr") float Y[64];
section("seg_arr") float S[32];
section("seg_arr") float SB[32][36];
section("seg_arr") float freq[576];
section("seg_arr") float window[36];
section("seg_arr") float cos_matrix[18][36];
section("seg_arr") float cs[] = {0.857492926, 0.881741997, 0.949628649, 0.983314592, 0.995517816,
0.999160558, 0.999899195, 0.999993155};
section("seg_arr") float ca[] = {-0.514495755, -0.471731968, -0.313377454, -0.1819132, -0.094574193, -
0.040965583, -0.014198569, -0.003699975};

/*  Global variable  */
int sb_index = 17;

/*  External arrays  */
extern volatile float C[512];
extern volatile float fifo[512];

/*  Function declarations  */
void matrix_init();
void sub_filter();
void calc_mdct();
void alias_reduce();

/*  initialize variables  */
void matrix_init() {
```

```
	int i, k;
	int coef;
	float temp;

	// filterbank cosines
	for(i = 0;i<31;i++) {
		coef = (2*i) + 1;
		for(k = 0;k<64;k++) {
			Mat_f[i][k] = coef1;
			Mat_f[i][k] *= coef;
			Mat_f[i][k] *= (k - 16);
			Mat_f[i][k] = cosf(Mat_f[i][k]);
		}
	}

	// mdct window constants
	for (i=0; i<36; i++)
	{
		temp = coef2;
		temp *= (i + .5);
		window[i] = sinf(temp);
	}

	// mdct cosines
	for (i=0; i < 18; i++)
	{
		for (k=0; k<36; k++)
		{
			temp = coef3;
			temp *= ((2*k) + 19) * ((2*i) + 1);
			cos_matrix[i][k] = cosf(temp);
		}
	}
}

/*  filterbank (discretize into 32 subbands)  */
void sub_filter() {
	int i, j;
	int temp;
	float tempf, term;

	if(sb_index < 0) {
		sb_index = 17;
		for(i = 0;i<32;i++) {
			for(j=0;j<18;j++) {
				SB[i][(18 + j)] = SB[i][j];
			}
		}
	}

	for(i=0;i<512;i++) {
		tempf = fifo[i];
		tempf *= C[i];
		fifo[i] = tempf;
	}

	for(i=0;i<64;i++) {
		for(j=0;j<8;j++) {
			temp = (j * 64) + i;
			tempf = fifo[temp];
			Y[i] += tempf;
		}
```

```
		}

		for(i=0;i<32;i++) {
			tempf = 0;
			for(j=0;j<64;j++) {
				term = (Y[j] * Mat_f[i][j]);
				tempf += term;
			}
			SB[i][sb_index] = tempf;
		}

		sb_index--;
}


/*  MDCT (discretize into 576 frequency lines  */
void calc_mdct() {
		int i, j, k;
		int addr;
		float sum, temp;

		// mdct for long window (N = 36)
		for (i=0;i<32;i++) {
			for (j=0;j<18;j++) {
				sum = 0;
				temp = 0;
				for(k=0;k<36;k++) {
					temp = SB[i][k];
					temp *= window[k];
					temp *= cos_matrix[j][k];
					sum += temp;
				}
				if(i%2 > 0) {
					addr = 17 - j;
					addr += (i * 18);
				} else {
					addr = j;
					addr += (i * 18);
				}
				freq[addr] = sum;
			}
		}
		alias_reduce();
}

/*  Reduce aliasing (clean up signal)  */
void alias_reduce() {
		int i, j, addrl, addru;
		float templ, tempu;

		for(i=0;i<31;i++) {
			for(j=0;j<8;j++) {
				addrl = (18*i) + 17 - j;
				addru = (18*i) + 18 + j;
				templ = freq[addrl];
				tempu = freq[addru];
				freq[addrl] = templ * cs[j] + tempu * ca[j];
				freq[addru] = tempu * cs[j] - templ * ca[j];
			}
		}
}
```

7.4 writefile.c

```
/*********************************************
**
**  Final Project  MP3 Encoder
**
**  writefile.c
**  Brian Loker, Robin Liu
**
**
**  This file contains functions to write encoded
**  data to a bit stream and store data in memory
**   when bitstream is 32 bits (size of one SDRAM
**  address).
**
*********************************************/
#include <math.h>
#include <stdio.h>

/*  Constant  */
#define HEADER 0xfffb58c4;

/*  Function declaration */
void init_write();
void init_frame();
void write_number(int value, int length);
void add_bits(int value, int length);
void write_bytes();
void write_side();

/*  External variables  */
extern bool endofframe;
extern bool sideinfo;
extern bool endofside;
extern bool writetotalbits, finalwrite;

/*  Global variables  */
int *current_address;
int current_data;
int numbits;
int remaining_bytes;
int *begin_frame;
int remaining_side_info;
int remaining_bits;
int main_data_end;
int *curr_side;
int tempdata, tempposition;
int *tempaddress;
bool keepflag, finishside;


/*  Initialize pointers and variables for writing */
void init_write()
{
        begin_frame = (int *) 0x03070000;
        current_address = begin_frame + 5;
        numbits = 0;
        current_data = 0;
}

/*  Initialize frame, write header and initialize pointers */
void init_frame()
{
```

```
	int temp;
	temp = HEADER;
	*begin_frame = temp;
	curr_side = begin_frame + 1;
	finishside = false;
}

/*  Write final total bit count to side information  */
void write_number(int value, int length)
{
	value = value & 0xffff;
	value >> (length-1) & 0xff;
	tempdata = tempdata | (value << (32 - tempposition));
	*tempaddress = tempdata;
}

/*  Write value of bit length, length, to bitstream
 *	If bitstream = 32 bits, then write to memory  */
void add_bits(int value, int length)
{
	int temp;

	if (current_address == begin_frame)
	{
		current_address = begin_frame + 5; // beginning plus 160 header and sideinfo bits
		current_data = remaining_side_info;
		numbits = 8;
		finishside = true;
	}

	numbits += length;
	if (numbits > 32)
	{
		remaining_bits = numbits - 32;
		value = value & 0xffff;			// change value into binary
		value >> (length-1) & 0xff;
		temp = value << (32 - remaining_bits);
		current_data = current_data | (value >> remaining_bits);
		if (keepflag)
			tempdata = current_data;
		if (sideinfo)
			write_side();
		else
			write_bytes();
		numbits = remaining_bits;
		current_data = temp;
	} else
	{
		value = value & 0xffff;
		value >> (length-1) & 0xff;
		current_data = current_data | (value << (32 - numbits));
		if (writetotalbits)
		{
			keepflag = true;
			tempposition = numbits;
			tempaddress = current_address;
		}
		if (numbits == 32)
		{
			if (keepflag)
			{
				tempdata = current_data;
```

```
                        keepflag = false;
                }
                if (sideinfo)
                        write_side();
                else
                        write_bytes();
                // initialize pointers and variables for next frame
                if (endofframe)
                {
                        endofframe = false;
                        begin_frame = begin_frame + 72;  //(288 bytes in frame)
                        main_data_end = 4*(begin_frame - current_address);
                        // write remaining side info if not written yet
                        if (finishside == false)
                        {
                                current_data = remaining_side_info;
                                temp = 0x00;
                                current_data = current_data | (temp << (24));
                                finishside = true;
                        }
                        // set maximum bit reservior to 256 bytes
                        if ((main_data_end/4) > 65)
                        {
                                main_data_end = 65 * 4;
                                current_address = begin_frame - 65;
                        }
                }
                numbits = 0;
                current_data = 0;
        }
}
if (endofside)
{
        remaining_side_info = current_data;
        endofside = false;
        sideinfo = false;
}

// initialize pointers and variables for next frame
// write remaining data and side info, padded with zeros
if (endofframe)
{
        remaining_bits = 32 - numbits;
        temp = 0x00;
        current_data = current_data | (temp << (remaining_bits));
        write_bytes();
        if (finishside == false)
        {
                current_data = remaining_side_info;
                temp = 0x00;
                current_data = current_data | (temp << (24));
                finishside = true;
        }
        numbits = 0;
        current_data = 0;
        endofframe = false;
        begin_frame = begin_frame + 72;  //(288 bytes in frame)
        main_data_end = 4*(begin_frame - current_address);
        if ((main_data_end/4) > 65)
        {
                main_data_end = 65 * 4;
                current_address = begin_frame - 65;
```

```
                    }
                }
}

/* Write 32 bits of data to memory*/
void write_bytes()
{
                    *current_address = current_data;
                    current_address++;
}

/* Write 32 bits of side info to memory */
void write_side()
{
                    *curr_side = current_data;
                    curr_side++;
}
```

7.5 AD1819a_initialization.asm

```
/***   AD1819a_initialization.ASM   ********************************************
*                                                                              *
*      AD1819/ADSP-21065L SPORT1 Initialization Driver                         *
*      Developed using the ADSP-21065L EZ-LAB Evaluation Platform              *
*                                                                              *
*      This version sets up codec communication for Variable Sample Rate       *
*      Support.  After codec register programming, the ADCs and DACs are       *
*      powered down and back up again, so left/right valid bits and DAC        *
*      requests occur simultaneously in the same audio frame.                  *
*                                                                              *
*      For efficient handling of Tag bits/ADC valid/DAC requests, the codec     *
*      ISR is processes using the SPORT1 TX (vs RX) interrupt. The SPORT1 TX    *
*      interrupt is used to first program the AD1819A registers, with only     *
*      an RTI at that vector location. After codec initialization, the SPORT1   *
*      TX ISR jump label is installed, replacing an 'RTI' instruction, so that *
*      normal codec audio processing begins at that point.                     *
*                                                                              *
*                                  John Tomarakos              *
*                                  ADI DSP Applications Group          *
*                                  Revision 3.0                *
*                                  04/29/99                    *
*                                                                              *
********************************************************************************/

/* ADSP-21060 System Register bit definitions */
#include  <def21065l.h>
#include   <ezkit/1819regs.h>
#include   <asm_sprt.h>

.GLOBAL                 _Program_SPORT1_Registers;
.GLOBAL                 _Program_DMA_Controller;
.GLOBAL                 _AD1819_Codec_Initialization;
.GLOBAL                 _tx_buf;
.GLOBAL                 _rx_buf;

/* AD1819 Codec Register Address Definitions */
#define             REGS_RESET                          0x0000
#define             MASTER_VOLUME                       0x0200
#define             RESERVED_REG_1                      0x0400
#define             MASTER_VOLUME_MONO                  0x0600
#define             RESERVED_REG_2                      0x0800
```

```
#define         PC_BEEP_Volume              0x0A00
#define         PHONE_Volume                0x0C00
#define         MIC_Volume                  0x0E00
#define         LINE_IN_Volume              0x1000
#define         CD_Volume                   0x1200
#define         VIDEO_Volume                0x1400
#define         AUX_Volume                  0x1600
#define         PCM_OUT_Volume              0x1800
#define         RECORD_SELECT               0x1A00
#define         RECORD_GAIN                 0x1C00
#define         RESERVED_REG_3              0x1E00
#define         GENERAL_PURPOSE             0x2000
#define         THREE_D_CONTROL_REG         0x2200
#define         RESERVED_REG_4              0x2400
#define         POWERDOWN_CTRL_STAT         0x2600
#define         SERIAL_CONFIGURATION        0x7400
#define         MISC_CONTROL_BITS           0x7600
#define         SAMPLE_RATE_GENERATE_0      0x7800
#define         SAMPLE_RATE_GENERATE_1      0x7A00
#define         VENDOR_ID_1                 0x7C00
#define         VENDOR_ID_2                 0x7E00

/* Mask bit selections in Serial Configuration Register for
   accessing registers on any of the 3 codecs */
#define         MASTER_Reg_Mask             0x1000
#define         SLAVE1_Reg_Mask             0x2000
#define         SLAVE2_Reg_Mask             0x4000
#define         MASTER_SLAVE1               0x3000
#define         MASTER_SLAVE2               0x5000
#define         MASTER_SLAVE1_SLAVE2        0x7000

/* Macros for setting Bits 15, 14 and 13 in Slot 0 Tag Phase */
#define         ENABLE_VFbit_SLOT1_SLOT2    0xE000
#define         ENABLE_VFbit_SLOT1          0xC000

#define         AD1819_RESET_CYCLES   60
                        /* ad1819 RESETb spec = 1.0(uS) min */
                        /* 60(MIPs) = 16.67 (nS) cycle time, therefore >=    40 cycles */
#define         AD1819_WARMUP_CYCLES    60000
                        /* ad1819 warm-up = 1.0(mS) */
                        /* 60(MIPs) = 16.67 (nS) cycle time, therefore >= 40000 cycles */


/*-------------------------------------------------------------------------*/
.segment /dm    seg_dmda;
.var _rx_buf[5];      /* receive buffer */

                                        /* transmit buffer */
.var _tx_buf[7] = ENABLE_VFbit_SLOT1_SLOT2,     /* set valid bits for slot 0, 1, and 2 */
                  SERIAL_CONFIGURATION,                 /* serial configuration
register address */
                  0xFF80,                               /* initially set to 16-bit slot
mode for ADI SPORT compatibility*/
                  0x0000,                               /* stuff other slots with zeros
for now */
                  0x0000,
                  0x0000,
                  0x0000;
                  /* slots 5 and 6 are dummy slots, to allow enough time in the TX ISR to go     */
                  /* get rx slots 4 & 5 data in same audio frame as the ADC valid tag bits.      */
                  /* This is critical for slower sample rates, where you may not have valid data */
                  /* every rx audio frame. So you want to make sure there is valid right         */
```

```
                    /* channel data in the same rx DMA buffer fill as the detection of an ADC      */
                    /* valid right bit.  These extra slots are required ONLY for fs < 48 kHz.          */

.var rcv_tcb[8]  = 0, 0, 0, 0, 0, 5, 1, 0;      /* receive tcb */
.var xmit_tcb[8] = 0, 0, 0, 0, 0, 7, 1, 0;      /* transmit tcb */

/* Codec register initializations */
/* Refer to AD1819 Data Sheet for register bit assignments */
#define    Select_LINE_INPUTS        0x0404    /* LINE IN - 0X0404, Mic In - 0x0000 */
#define Select_MIC_INPUT  0x0000
#define Line_Level_Volume  0x0000    /* 0 dB for line inputs */
#define Mic_Level_Volume   0x0202
#define    Sample_Rate                    32000

.var Init_Codec_Registers[34] =
          MASTER_VOLUME,                0x0000,   /* Master Volume set for no attenuation */
          MASTER_VOLUME_MONO,           0x8000,         /* Master Mono volume is muted */
          PC_BEEP_Volume,               0x8000,         /* PC volume is muted */
          PHONE_Volume,                 0x8008,         /* Phone Volume is muted */
          MIC_Volume,                   0x8008,   /* MIC Input  analog loopback is muted */
          LINE_IN_Volume,               0x8808,   /* Line Input analog loopback is muted */
          CD_Volume,                    0x8808,         /* CD Volume is muted */
          VIDEO_Volume,                 0x8808,         /* Video Volume is muted */
          AUX_Volume,                   0x8808,         /* AUX Volume is muted */
          PCM_OUT_Volume,               0x0808,   /* PCM out from DACs is 0 db gain for both
channels */
          RECORD_SELECT,                Select_MIC_INPUT, /* Record Select on Line Inputs for
L/R channels */
          RECORD_GAIN,                  Mic_Level_Volume,         /* Record Gain set for 0 dB
on both L/R channels */
          GENERAL_PURPOSE,              0x0000,         /* 0x8000, goes through 3D circuitry */
          THREE_D_CONTROL_REG,          0x0000,         /* no phat stereo */
          MISC_CONTROL_BITS,            0x0000,   /* use SR0 for both Left and Right ADCs and
DACs, repeat sample */
          SAMPLE_RATE_GENERATE_0,       Sample_Rate,    /* user selectable sample rate */
          SAMPLE_RATE_GENERATE_1,       Sample_Rate;    /* Sample Rate Generator 1 not used in
this example */

.endseg;

.SEGMENT /pm     seg_pmco;

/* ---------------------------------------------------------------------------------------*/
/* Sport1 Control Register Programming
                              */
/* Multichannel Mode dma w/ chain, erly fs, act hi fs, fall edge, no pack, data=16/big/zero         */
/* ---------------------------------------------------------------------------------------*/

_Program_SPORT1_Registers:
          leaf_entry;
          /* sport1 receive control register */
          R0 = 0x0F8C40F0;          /* 16 chans, int rfs, ext rclk, slen = 15, sden & schen enabled */
          dm(SRCTL1) = R0;                        /* sport 0 receive control register */

          /* sport1 transmit control register */
          R0 = 0x001C00F0;          /* 1 cyc mfd, data depend, slen = 15, sden & schen enabled */
          dm(STCTL1) = R0;                        /* sport 0 transmit control register */

          /* sport1 receive frame sync divide register */
          R0 = 0x00FF0000;                  /* SCKfrq(12.288M)/RFSfrq(48.0K)-1 = 0x00FF */
          dm(RDIV1) = R0;
          dm(TDIV1) = R0;
```

```
        /* sport1 receive and transmit multichannel word enable registers */
        R0 = 0x0000001F;                    /* enable receive channels 0-4 */
        dm(MRCS1) = R0;
        R0 = 0x0000007F;                    /* enable transmit channels 0-6 */
        dm(MTCS1) = R0;

        /* sport1 transmit and receive multichannel companding enable registers */
        R0 = 0x00000000;                    /* no companding */
        dm(MRCCS1) = R0;                    /* no companding on receive */
        dm(MTCCS1) = R0;                    /* no companding on transmit */

        leaf_exit;

/*-------------------------------------------------------------------------------*/
/*                          DMA Controller Programming For SPORT1                */
/*-------------------------------------------------------------------------------*/

_Program_DMA_Controller:
        leaf_entry;
        r1 = 0x0001FFFF;                    /* cpx register mask */

        /* sport1 dma control tx chain pointer register */
        r0 = _tx_buf;
        dm(xmit_tcb + 7) = r0;      /* internal dma address used for chaining*/
        r0 = xmit_tcb + 7;  /* get DMA chaining internal mem pointer containing tx_buf address */
        r0 = r1 AND r0;                     /* mask the pointer */
        r0 = BSET r0 BY 17;                 /* set the pci bit */
        dm(xmit_tcb + 4) = r0;      /* write DMA transmit block chain pointer to TCB buffer */

        /* sport1 dma control rx chain pointer register */
        r0 = _rx_buf;
        dm(rcv_tcb + 7) = r0;       /* internal dma address used for chaining */
        r0 = rcv_tcb + 7;
        r0 = r1 AND r0;                     /* mask the pointer */
        r0 = BSET r0 BY 17;                 /* set the pci bit */
        dm(rcv_tcb + 4) = r0;       /* write DMA receive block chain pointer to TCB buffer*/

        r0 = dm(xmit_tcb + 4);              /* get address of the start IIT1A register */
        dm(CPT1A) = r0;                     /* write address to chain pointer register */
        r0 = dm(rcv_tcb + 4);               /* get address of start IIR1A register */
        dm(CPR1A) = r0;                     /* write address to chain pointer register  */

        leaf_exit;

/* ------------------------------------------------------------------------------- */
/*                      AD1819A Codec Initialization                               */
/* ------------------------------------------------------------------------------- */

_AD1819_Codec_Initialization:
        leaf_entry;
        bit set IMASK SPT1I;

Wait_Codec_Ready:                           /* Wait for CODEC Ready State */
        R0 = DM(_rx_buf + 0);               /* get bit 15 status bit from AD1819 tag phase
slot 0 */
        R1 = READY_BIT;                     /* mask out codec ready bit in tag phase */
        R0 = R0 AND R1;                     /* test the codec ready status flag bit */
        IF EQ JUMP Wait_Codec_Ready;    /* if flag is lo, continue to wait for a hi */

        idle;                               /* wait for a couple of TDM audio frames to pass */
        idle;
```

```
Initialize_1819_Registers:
        i4 = Init_Codec_Registers;      /* pointer to codec initialization commands */
        r15 = ENABLE_VFbit_SLOT1_SLOT2;         /* enable valid frame bit, and slots 1 and 2 valid
data bits */

        LCNTR = 17, DO Codec_Init UNTIL LCE;
                dm(_tx_buf + TAG) = r15;    /* set valid slot bits in tag phase for slots 0, 1 , 2 */
                r1 = dm(i4, 1);             /* fetch next codec register address */
                dm(_tx_buf + ADDR) = r1;    /* put fetched codec register address into tx slot 1 */
                r1 = dm(i4, 1);             /* fetch register data contents */
                dm(_tx_buf + DATA) = r1;    /* put fetched codec register data into tx slot 2 */
Codec_Init:     idle;                       /* wait until TDM frame is transmitted */

        /* For variable sample rate support, you must powerdown and powerback up the ADCs and DACs
        so that the incoming ADC data and DAC requests occur in left/right pairs */
PowerDown_DACs_ADCs:
        idle;
        r15 = ENABLE_VFbit_SLOT1_SLOT2;         /* enable valid frame bit, and slots 1 and 2 valid
data bits */
        dm(_tx_buf + TAG) = r15;                /* set valid slot bits in tag phase for slots 0, 1 , 2
*/
        r0=POWERDOWN_CTRL_STAT;
        dm(_tx_buf + ADDR) = r0;
        r0=0x0300;                              /* power down all DACs/ADCs */
        dm(_tx_buf + DATA) = r0;
        idle;
        idle;

        LCNTR = AD1819_RESET_CYCLES-2, DO reset_loop UNTIL LCE;
reset_loop:     NOP;                            /* wait for the min RESETb lo spec time */

        idle;
        r15 = ENABLE_VFbit_SLOT1_SLOT2;         /* enable valid frame bit, and slots 1 and 2 valid
data bits */
        dm(_tx_buf + TAG) = r15;            /* set valid slot bits in tag phase for slots 0, 1 , 2 */
        r0=POWERDOWN_CTRL_STAT;                 /* address to write to  */
        dm(_tx_buf + ADDR) = r0;
        r0=0;                               /* power up all DACs/ADCs */
        dm(_tx_buf + DATA) = r0;
        idle;
        idle;

        LCNTR = AD1819_WARMUP_CYCLES-2, DO warmup_loop2 UNTIL LCE;
warmup_loop2: NOP;                          /* wait for AD1819 warm-up */
        bit clr IMASK SPT1I;
        leaf_exit;                          /* End of AD1819A Initialization */

/* ---------------------------------------------------------------------------------- */
 .endseg;
```

7.6 Clear_SPT1_regs.asm

```
/* ////////////////////////////////////////////////////////////////////////////////
/                                                                                /
/  ROUTINE TO CLEAR AND RESET ALL SPORT1 REGISTERS                               /
/                                                                                /
/  This routine may be required for certaing AD1819A demos when using the 21065L EZ-LAB   /
/  RS232 Debug Monitor program.  The 21065L EZ-LAB boot EPROM Monitor kernel on power-up  /
/  executes a routine that programs the SPORT1 Control and DMA registers to              /
/  communicate with the AD1819A for the example supplied EZ-LAB demo programs.          /
/                                                                                /
```

```
/  When invoking the 65L VDSP RS232 Debugger, SPORT1 DMA is already active in          /
/  multichannel mode with DMA chaining.  If we wish to leave the SPORT TDM and DMA     /
/  channel configuration the same (i.e. 5 active channels and 5-word DMA buffers),     /
/  we are usually still able to reprogram the DMA controller to point to our own       /
/  own codec buffers with no side effects.  However, if we need to change any SPORT    /
/  control parameters such as the number of active TDM channels and DMA buffer sizes,  /
/  then the active EPROM Monitor SPORT TDM configuration on powerup of the EZ-LAB board /
/  will affect the re-programming of the SPORT within a downloaded DSP executable.     /
/                                                                                      /
/  Since the monitor program has already activated the SPORT1 registers after a board  /
/  reset, the side effect that occurs (when re-writing over the SPORT1 control         /
/  registers) is that MCM DMA data transfers are mistakingly restarted                 /
/  without the full programming of all the SPORT parameters.  Also, the TX and         /
/  RX buffers may be partially full or full, and can affect the DMA controller's       /
/  ability to correctly DMA in/out SPORT data to/from internal memory.  What results   /
/  is a failed link between the AD1819a and SPORT1 in user-modifed code, because       /
/  transmitted and recieved data is sent or received in different timeslots and misalign /
/  in the SPORT DMA buffers.                                                           /
/                                                                                      /
/  This routine simply clears all SPORT1 ctrl and DMA registers back to their          /
/  default states so that we can reconfigure it for our AD1819a application.          /
/                                                                                      /
/  John Tomarakos                                                                      /
/  ADI DSP Applications                                                                /
/  Rev 1.0                                                                             /
/  4/30/99                                                                             /
/                                                                                      /
////////////////////////////////////////////////////////////////////////////////////// */

/* ADSP-21060 System Register bit definitions */
#include  <def21065l.h>

.GLOBAL                    _Clear_All_SPT1_Regs;

.SEGMENT /pm    seg_pmco;

_Clear_All_SPT1_Regs:
        IRPTL = 0x00000000;       /* clear pending interrupts */
        bit clr imask SPT1I;

        R0 = 0x00000000;
        dm(SRCTL1) = R0;          /* sport1 receive control register */
        dm(RDIV1) = R0;           /* sport1 receive frame sync divide register */
        dm(STCTL1) = R0;          /* sport 0 transmit control register */
        dm(MRCS1) = R0;           /* sport1 receive multichannel word enable register */
        dm(MTCS1) = R0;           /* sport1 transmit multichannel word enable register */
        dm(MRCCS1) = R0;          /* sport1 receive multichannel companding enable register */
        dm(MTCCS1) = R0;          /* sport1 transmit multichannel companding enable register */

        /* reset SPORT1 DMA parameters back to the Reset Default State */
        R1 = 0x1FFFF;
        dm(IIR1A) = R1;
        dm(IIT1A) = R1;
        dm(GPR1A) = R1;
        dm(GPT1A) = R1;
        dm(CPR1A) = R1;
        dm(CPT1A) = R1;
        R1 = 0x0001;
        dm(IMR1A) = R1;
        dm(IMT1A) = R1;
        R1 = 0xFFFF;
        dm(CR1A)  = R1;
```

```
        dm(CT1A)  = R1;

        RTS;

.ENDSEG;
```

7.7 Init_065L_EZLAB.asm

```
/***  INIT_065L_EZLAB.ASM  ************************************************
*                                                                           *
*     ADSP-21065L EZ-LAB Initialization and Main Program Shell              *
*     Developed using the ADSP-21065L EZ-LAB Evaluation Platform            *
*                                                                           *
*                                                                           *
*                           John Tomarakos                                  *
*                           ADI DSP Applications Group                      *
*                           Revision 2.0                                    *
*                           4/28/99                                         *
*                                                                           *
****************************************************************************/

/* ADSP-21060 System Register bit definitions */
#include  "def21065l.h"
#include   "asm_sprt.h"

.GLOBAL                 _Init_DSP;

/*-------------------------------------------------------------------------*/

.segment /pm seg_pmco;

/*-------------------------------------------------------------------------*/
/* Note: This routine is first called at the Reset Vector in the Interrupt Vector Table */
/*-------------------------------------------------------------------------*/

_Init_DSP:
        leaf_entry;
        ustat1=0x3F;                    /* flags 4 thru 9 are outputs for LEDs */
        dm(IOCTL)=ustat1;
        bit set ustat1 0x3F;                                    /* toggle flag 4-9 LEDs */
        dm(IOSTAT)=ustat1;                                      /* turn on all LEDs */
        bit clr mode2 FLG3O | FLG2O | FLG1O | FLG0O;            /* flag 3, 2, 1 & 0 inputs  */
        bit set mode2 IRQ1E | IRQ2E | IRQ0E;            /* irq1 and irq2 edge sensitive         */
        IRPTL = 0x00000000;                             /* clear pending interrupts */
        bit set mode1 IRPTEN | NESTM;                   /* enable global interrupts & nesting */
        bit set imask 0x00000000;                       /* Don't allow interrupts during initialization */

        L0 = 0;
        L1 = 0;
        L2 = 0;
        L3 = 0;
        L4 = 0;
        L5 = 0;
        L6 = 0;
        L7 = 0;
        L8 = 0;
        L9 = 0;
        L10 = 0;
        L11 = 0;
        L12 = 0;
        L13 = 0;
```

```
        L14 = 0;
        L15 = 0;

        leaf_exit;

.endseg;
```

7.8 SDRAM_Init.asm

```
/*****************************************************************************
        ADSP-21065L EZ-LAB Evalutation Board
          - SDRAM Interface Initialization
          - Programmable I/O Flag LED Test
*****************************************************************************/
#include "def21065L.h"
#include "asm_sprt.h"

#define sdram_size 0xfffff

.GLOBAL         _Init_65L_SDRAM_Controller;

.SEGMENT/PM seg_pmco;

/*************************************************************************************************
        Setup the SDRAM
        Assumes SDRAM part# MT48LC1M16A1TG-12 S
                SDCLK=60MHz
                tCK=15ns min @ CL=2     -> SDCL=2
                tRAS=72ns min           -> SDTRAS=5
                tRP=36ns min            -> SDTRP=3
                tREF=64ms/4K rows       -> SDRDIV=(2(30MHz)-CL-tRP-4)64ms/4096=937cycles

        2 SDRAMs by 16 bits wide total=1Mbitx32
        Mapped to MS3 addresses 0x03000000-0x030fffff
*************************************************************************************************/

_Init_65L_SDRAM_Controller:
        leaf_entry;
                ustat1=dm(WAIT);
                bit clr ustat1 0x000f8000;      /*clear MS3 waitstate and mode*/
                dm(WAIT)=ustat1;

                ustat1=937;                     /*refresh rate*/
                dm(SDRDIV)=ustat1;

                ustat1=dm(IOCTL);               /*mask in SDRAM settings*/
                bit set ustat1 SDPSS|SDBN2|SDBS3|SDTRP3|SDTRAS5|SDCL2|SDPGS256|DSDCK1;
                dm(IOCTL)=ustat1;
          leaf_exit;

/*-------------------------------------------------------------------------------------*/

.ENDSEG;
```

7.8 NOT INCLUDED

c.h	:	contains definition of 512 point window
huffman.h	:	contains definitions of huffman tables
scalefac_bands.h	:	contains definitions of scalefactor bands and scalefactors

7.9 MATLAB FILE

```
function out = test(buffer)
%coeffiecients for window
c = [0,-0.000000477,-0.000000477,-0.000000477, … ,0.000000477,0.000000477,0.000000477];
%to setup matrix coefficients
for i = 1:32
   for k = 1:64
      j = (2 * i - 1) * (k - 17) * pi / 64;
      mat(i,k) = cos(j);
   end
end
for f = 1:1

xr = buffer;

for q = 1:36
   for i = 1:512
      y(513 - i) = xr((((q - 1) * 32) + i) + (35 * (f-1))) * c(i);
   end


   for i = 1:64
      z(i) = 0;
      for j = 1:8
         z(i) = z(i) + y(i + 64 * (j - 1));
      end
   end

   for i = 1:32
      sum(i) = 0;
      for j = 1:64
         sum(i) = sum(i) + z(j) * mat(i,j);
      end
   end

   for i = 1:32
      sb(i,q) = sum(i);
   end
end

SB1 = mdct36(sb(1,:));
SB2 = mdct36(sb(2,:));
SB3 = mdct36(sb(3,:));
SB4 = mdct36(sb(4,:));
SB5 = mdct36(sb(5,:));
SB6 = mdct36(sb(6,:));
SB7 = mdct36(sb(7,:));
SB8 = mdct36(sb(8,:));
SB9 = mdct36(sb(9,:));

%for aliasing correction
for w = 1:18
   temp1(w) = SB2(19-w);
   temp2(w) = SB4(19-w);
   temp3(w) = SB6(19-w);
   temp4(w) = SB8(19-w);
end

out = [SB1,temp1,SB3,temp2,SB5,temp3,SB7,temp4,SB9];

stem(out);
end
```

8 REFERENCES

Brandenburg, K. and G. Stoll, 1994. "ISO-MPEG-1 Audio: A Generic Standard for coding of High-Quality Digital Audio," *J. Audio Eng. Soc*.**, 42** (10), p. 780-792.

Fraunhofer website. http://www.iis.fhg.de/amm/dechinf/layer3/.

ISO/IEC IS11172-3, 1992. "Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbit/s - Part 3 Audio."

MP3 Tech website. http://mp3-tech.org.

Noll, P., 1997. "MPEG Digital Audio Coding," *IEEE Signal Processing Magazine*, p. 59-81.

Pan, D.Y., 1993. "Digital Audio Compression," *Digital Technical Journal*, **5** (2), p. 1-14.

Pan, D.Y., 1995. "A Tutorial on MPEG/Audio Compression," *IEEE Multimedia*, **2** (2), p. 60-74.

United States Patent 5,579,430, 1996. "Digital Encoding Process."