18-551 Spring 2000

FINAL REPORT

# AUTOMOTIVE AUDIO PROCESSOR

## GROUP 12


Joel Tan (jtan+)
Kevin Perry (kperry+)
Sean Compton (sc7h+)

## ABSTRACT

An audio processor for enhancing sound reproduction in the automotive environment using an automatically calibrating 1/3 octave equalizer, speaker time alignment, and addition of reverb. Spectrum analysis of speaker output in the listening environment, captured through a calibrated microphone, allows accurate representation of frequency response. Accurate adjustment of individual frequency bands using digital filters eliminates peaks and dips to produce ideal frequency response. Speaker time alignment compensates for limited speaker placement locations. Reverb allows the small cabin space to take on larger sonic dimensions.

## PROBLEM

The automotive environment is one of the least ideal listening environments because it was designed as a means of transportation first, and as a listening environment last.. Yet, the car audio industry is a booming marketplace. From new car owners adding "premium sound" options, to car audio fanatics who compete in sound quality competitions, they are all driven by the same hunger for better sound when they are driving. Our project is targeted at the serious audiophile who seeks quality sound in an automobile for competition purposes. Car audio competitions are sanctioned by several different organizations, most notably International Auto Sound Challenge Association (IASCA) and United States Autosound Competition (USAC). The various criteria for judging include spectral balance, imaging, soundstage height, width and depth. Scoring is based on subjective listening tests by the judges. Real Time Analysers (RTAs) are also used to measure the sound pressure level (SPL) and frequency response of the competitors' systems. A scoring algorithm is programmed into the judges' RTA so that a perfect score in the RTA section can only achieved by a flat frequency response with a tolerance of +/- 3dB. In the Expert class competitions, top competitors are typically separated by only fractions of a point. Tweaking of a system to perfection is the key to winning. Currently, competitors use a multitude of processors and equipment costing around $10,000. We designed our project with features to

allow competitors maximum versatility and control over every aspect of their sound in one package.

There are inherent flaws in the sonic character of the passenger cabin. The space is much smaller than a regular room, resulting in a very enclosed sound when compared to a spacious sounding concert hall. The noise floor is quite high when the vehicle is in motion, but for our purposes of competition, the vehicles are generally parked and the power is drawn from the battery.

The surfaces in the cabin have poor sound reflection/dispersion properties (eg. Glass, flexible interior panels). The placement of these reflecting surfaces and absorptive surfaces is haphazard from an audio standpoint. The choice of mounting locations for loudspeakers is somewhat limited, as is the amount of available space. Phase interference between tweeters and midrange drivers that are mounted off-axis relatively causes peaks and dips in the frequency response that detract from the smoothness of a system. Peaks in the overall response also occur when a reflecting surface is present at odd fractions of a wavelength. Diaphragmatic resonance in the flexible interior panels causes a frequency response dip at the resonant frequency of the panel. Standing waves are also present, as in any room, and serve only to make frequency response uneven. A peak also occurs at the resonance of the interior cabin.

Speakers generally have a flat response only with a tolerance of +/-5dB for the expensive ones. The distance between each speaker of the stereo pair and the listener is also unequal, so sound from the farther speaker arrives later and sounds softer than the near speaker. As a result, sound reproduction in a passenger cabin is often much less than ideal. Similar problems can also occur in performance halls and listening rooms of all sizes, so the proposed audio processor is not limited in use to automobiles. The audio signal will be sampled at 48kHZ and has a bandwidth of 20-20kHz.

## SIMILAR PREVIOUS 18-551 WORK

- Adaptive Stereo Equalizer for Room Acoustics and Loudspeaker Cross talk Compensation, Spring 1995

- Real Time Parametric Equalizer, Spring 1997

Both of these past projects encountered large complexity issues. The Adaptive Stereo Equalizer was in the end unable to run at 44.1 kHz and had many problems with available memory. The Real Time Parametric Equalizer group was unable to implement the process real time on the DSP board. They were forced to run completely on the PC side. Even with a great deal of memory used, the project was only able to run at 8 bit Mono, 22.05 kHz, in addition to only being able to run 4 filters. Though the projects differ in specific goal, they are very similar in terms of general approach, filter implementation, memory and speed requirements, and so forth.

## SOLUTION

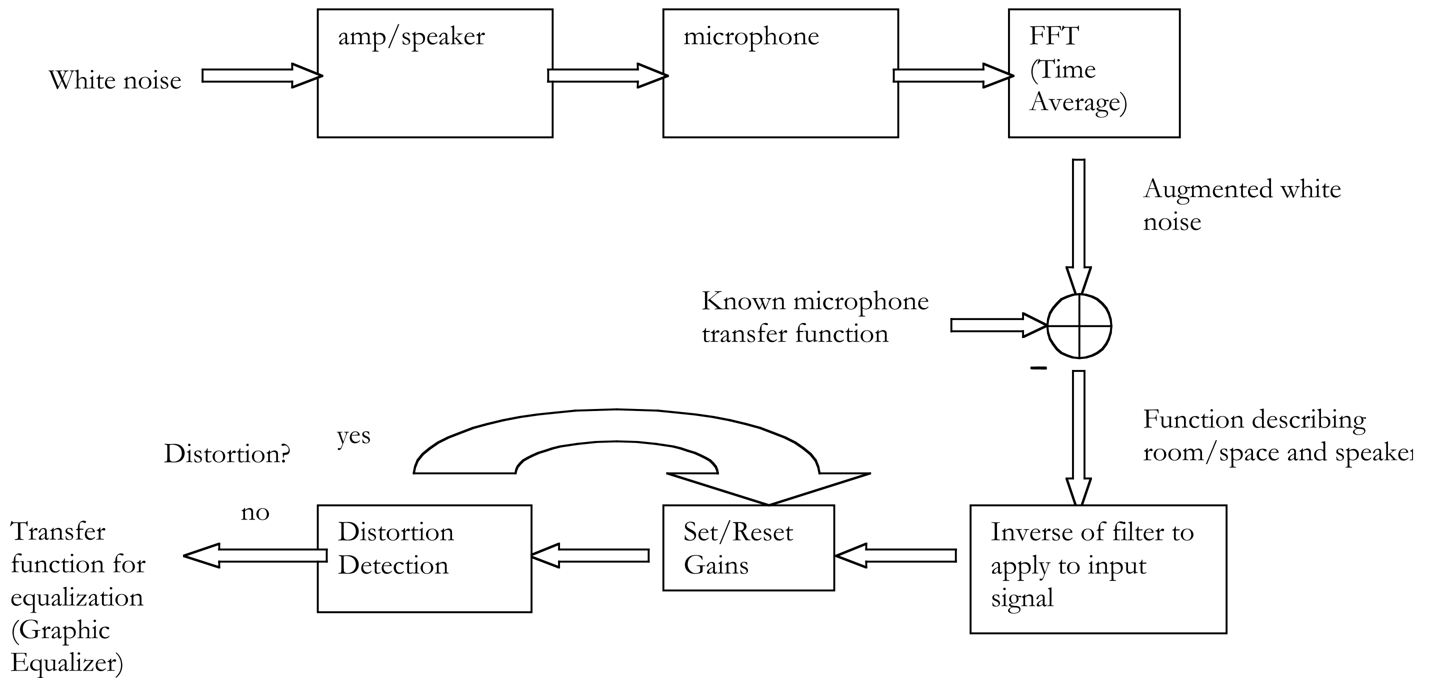
We propose to improve and correct some of these issues using the processing power of the C67 EVM. To correct the frequency response, we will implement the automatically calibrating 1/3 octave equalizer. The processor will also be able to add reverb to the signal so that the small cabin can sound more spacious. The time alignment will compensate for the poor speaker locations.

The beginning steps in the calibration simply involve the playback of white noise through the equipment that will be used in the sound system. White noise is used due to its property of having the same magnitude response at all frequencies. This will make it easier to determine which frequencies are affected by the current space and equipment. The room and speaker have their own transfer function which will affect the signal. The microphone also has a transfer function associated with it; however, its frequency response is known prior to calibration. Therefore the only unknown before reaching the time averaged FFT

would be the transfer function of the room and speaker which is one of the major factors which the graphic equalizer will correct.

**EQUALIZER**

CALIBRATION



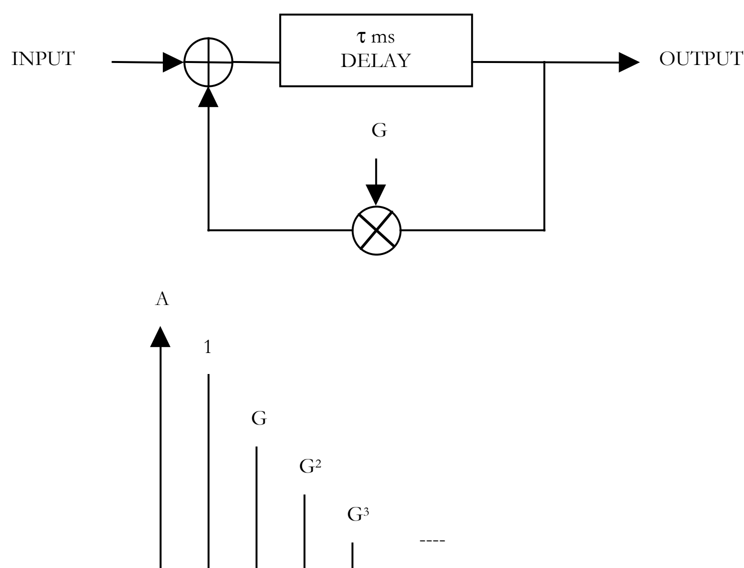

After input to the microphone, the time average FFT will display the frequency response of the signal with all of the augmentations of the equipment that it has passed through. The known microphone transfer function is then used in order to remove the transfer function of the microphone from the signal. The result is a transfer function describing the exact frequency response of the room with the amp and speaker used. We want to apply the inverse of this function to the equalizer output so that a flat overall response can be achieved. A completely flat response would not sound good since our ears have a transfer function which peaks around 1kHz. Instead, the signal is equalized to match a user-defined curve with a high frequency roll off and some bass emphasis. Bandpass filters, spaced 1/3 octave apart, will be used in parallel to attenuate or amplify each frequency band to match the user input curve. Once the gains have been set, a sine wave sweep is performed for each

frequency center of the equalizer. The FFT of this output is performed to ensure no harmonic distortion has been introduced due to clipping. Clipping may be a result of filter gains being set too high or a result of the amplifier or speakers distorting when they run out of headroom. Diaphragmatic resonance in the interior panels absorbs power at their resonant frequency. No change in perceived loudness at that frequency can be achieved no matter how high the gain at that frequency band is set. In order to conserve power, these "holes" need to be detected and the gains on these bands need to be limited. During playback, the signal is routed through the equalizer before reaching the amplifier.

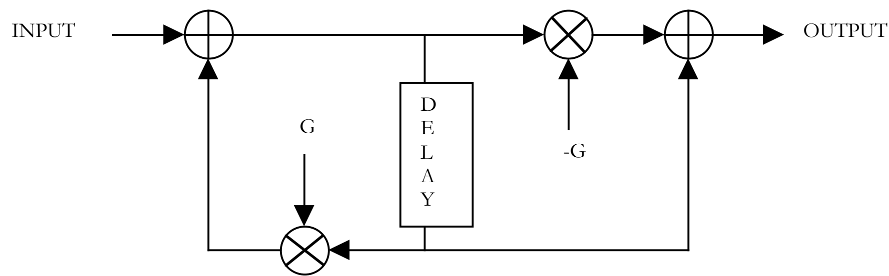
**REVERB & DELAY**

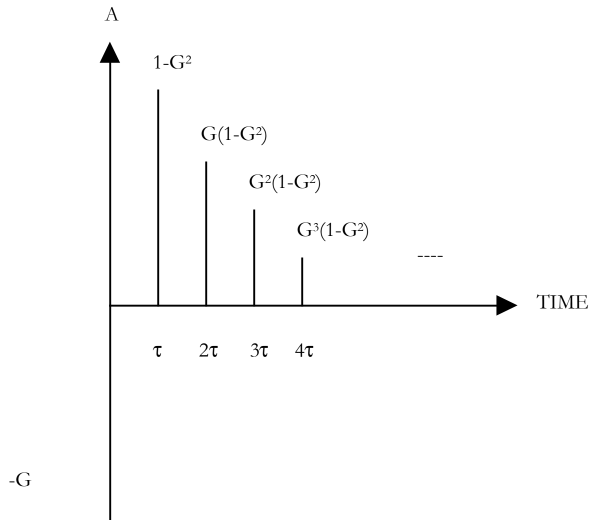
The addition of digital reverberation to our unit will add depth to the sound output, coloring it and changing its tone quality. This in effect will simulate a perceived acoustic quality of the listening space, making an enclosed area sound more spacious for example. This is done by producing delayed images of the original signal that decay exponentially with time. Digital reverberation may be modeled as a filter with an impulse response that reproduces as closely as possible the impulse response of the desired listening space, such as a cathedral or concert hall.

We propose to use the Schroeder Reverberation Model for our reverberator unit for it's ability to handle long reverberation times by using a minimal amount of calculations as well as short untapped delay lines. Schroeder Reverberators consist of a combination of comb filters and all-pass networks. A model of our comb filter is seen below. The comb filter simply sends the input signal into a delay line and then multiplies a feedback line from the output with a decay factor G and sums this with the input signal.



The impulse response of the filter is a pulse train of decaying exponentials. Passing a signal



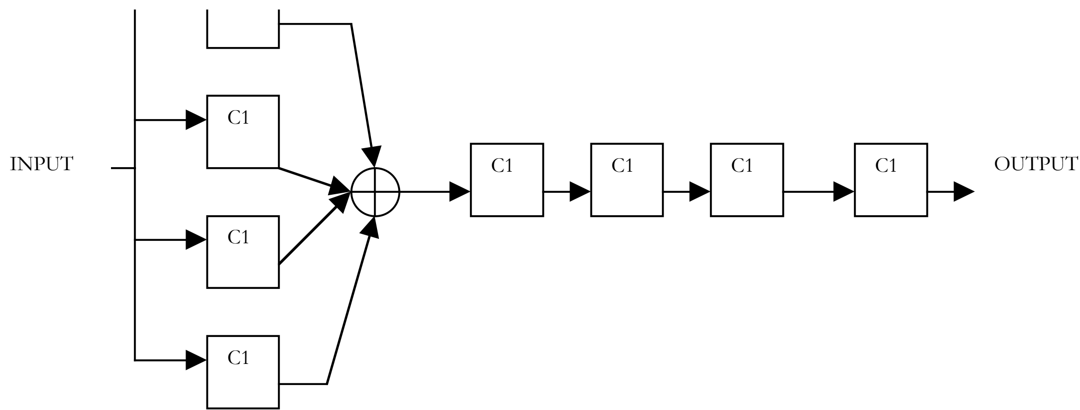




through this filter will cause the output to "ring" at a frequency $1/\tau$. The second unit in our reverberator is the all-pass network.

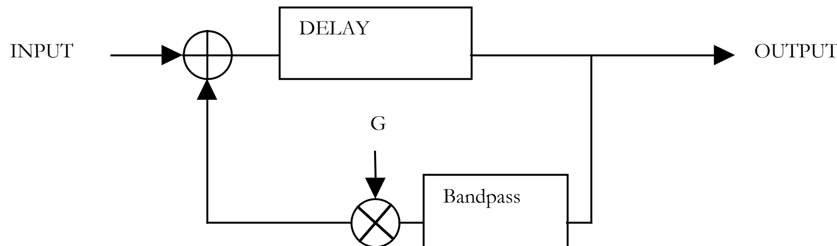
The all-pass network passes equally all signals in the steady state, but significantly effects the phase of individual components. There is no delay between the input and output of the signal.

When reverberator modules are connected in parallel, their collective impulse response is the sum of the individual responses, and the number or pulses generated will be the sum of the number pulses from each unit. Likewise, when placed in cascade each unit's response will in turn trigger the next one, and the number of pulses generated will be the product of the number of pulses from each unit. The Schroeder model calls for a combination of comb filters connected in parallel and cascaded all-pass networks in order to minimize distortion in

the frequency domain. An example of such a system with four comb filters and all-pass networks is shown below.




In addition to the required combination of filters needed for this reverberator unit, we must address the issue that in real auditoriums and concert halls, low frequency sounds have longer reverberation times. In order to model this, we are using a variation of the Schroeder model which incorporates a bandpass filter on the comb filter feedback path.




The passband on the filter will be chosen in order to model different reverberation times at different frequencies. The actual values will depend upon the response of the listening environment we wish to simulate.

Finally, we must consider the ratio of the amount of the input signal we wish to reverberate. By passing a percentage of the signal through to the output without filtering it through the Schroeder reverberator, we give the output better definition and presence. Otherwise, an output signal with 100% reverberation would sound very hollow, without a sensation of the initial sound and it's following delays. The desired simulated distance of the listener to the signal source determines this reverberation ratio. For multichannel output separate reverberators will be implemented on each channel to create spatial depth.

We have a rough estimate of the delay period for the comb filters in our model of 50ms. We know also the ratio of the longest to shortest delays is in the range of 1.7:1. This should

provide approximate concert hall characteristics. (ref Computer Music, Charles Dodge, Schirmer Books, 1997)

## DEMO

Microphone will be connected to the EVM and the audio source goes into the EVM and the output of the EVM is played through speakers. First we play a CD with recorded white noise to perform the calibration. Reverb and then time alignment will be turned on sequentially so that the audience can appreciate the improvements made to the sound quality.

## IMPLEMENTATION

### 1/3 OCTAVE SPECTRUM ANALYZER

Plan:

- Sample 16384 points of audio input
- Perform radix-4 FFT on 16384 points
- Average the values in each band
- During initial run of the program, a lookup table is generated to relate the bounds of each frequency band to the nearest discrete frequency in the FFT.
  - next center frequency = current center frequency*(2^(1/3))
  - for each center frequency, frequency bounds are
    i. Lower bound = center frequency*(2^(-1/6))
    ii. Upper bound = center frequency*(2^(1/6))

Actual:

- 16384 pt FFT required 32768 floats and 131 kbytes
    - had to run off chip
    - poor resolution at low frequencies
    - too time consuming
- had to use 4096 pt FFT
    - total resolution severely reduced (only 1-2 samples) for each center below 60 Hz
    - ran quickly on chip
    - averaging did not help low frequencies

Solution:

- 4096 pt FFT with Averaging
    - as noted, had very poor resolution at low frequencies
- low pass filter, and oversampling
    - take 16384 samples and low pass at 1kHz
    - take every $4^{th}$ sample and run 4096 pt FFT
    - unresolved, no comprehensible results

**FILTERS**

Plan:

Many different Algorithms were tried out for Equalizer and filter implementations. The following solutions were experimented with:

- Motorola Solutions
- TI Bandpass Solutions
- Cascade Biquad Solutions
- others found at various DSP websites, IIR vs FIR, etc

None of the solutions performed well.

- Filter orders were severely reduced
- Q factor for pass bands unacceptable
- Large problems generating correct coefficients

Most of these algorithms we found later utilized SAA approximation in determining filter coefficients. We discovered that SAA approximation works fine for low frequencies, but produces large error for frequencies greater than 5 kHz. This was a large problem since our resolution of lower frequencies was also severely reduced in the FFT.

Solution:

We decided on writing two of our own solutions and comparing the results. We tried implementing cascaded biquads/Butterworth filters for IIR filter implementation. We generated coefficients and compared the results to a MATLAB simulation, the results were very similar.

Problems:

The implementation was too slow for real time. 26 $2^{nd}$ order Butterworth filters incurred a huge delay when running input from the spectrum analyzer. (+300 cycles, ea)

### REVERB/TIME DELAY

Implementation:

- Modified to fit on chip and incur less timing delay.
- Used allpass filters with circular delay buffer implementation
- Delay times kept small, but ideal for automotive environment
- Speed of sound in dry air at 68F is 344m/sec.
- 344m/sec ^-1 * Sampling Rate * left to right ear speaker distance difference is equal to number delay samples
- Never reached stage to implement with equalizer

---

## PROBLEMS

---

### MEMORY

Had to spend a great deal of time on this initially. Memory availability seems to be the best we can do for now, not fully optimized yet though. This is due to overall program bugs which are preventing us from proceeding to this stage.

Onchip:

Buffer 4096*2 floats * 4 bytes = 32768 bytes

W (twiddle factors) _*4096*2 floats * 4 bytes = 24576 bytes

Filter coefficients 3*30*3 ints * 4 bytes + 1*30*3 floats * 4 bytes = 1440 bytes

Flags, counter vars, other … 56 bytes

Total = 58840 bytes used on chip

Offchip

X (fft average) 4096*2 floats * 4 bytes = 32768 bytes

Revtable (digit reversal) 4096 ints * 4 bytes = 16384 bytes

Total = 49152

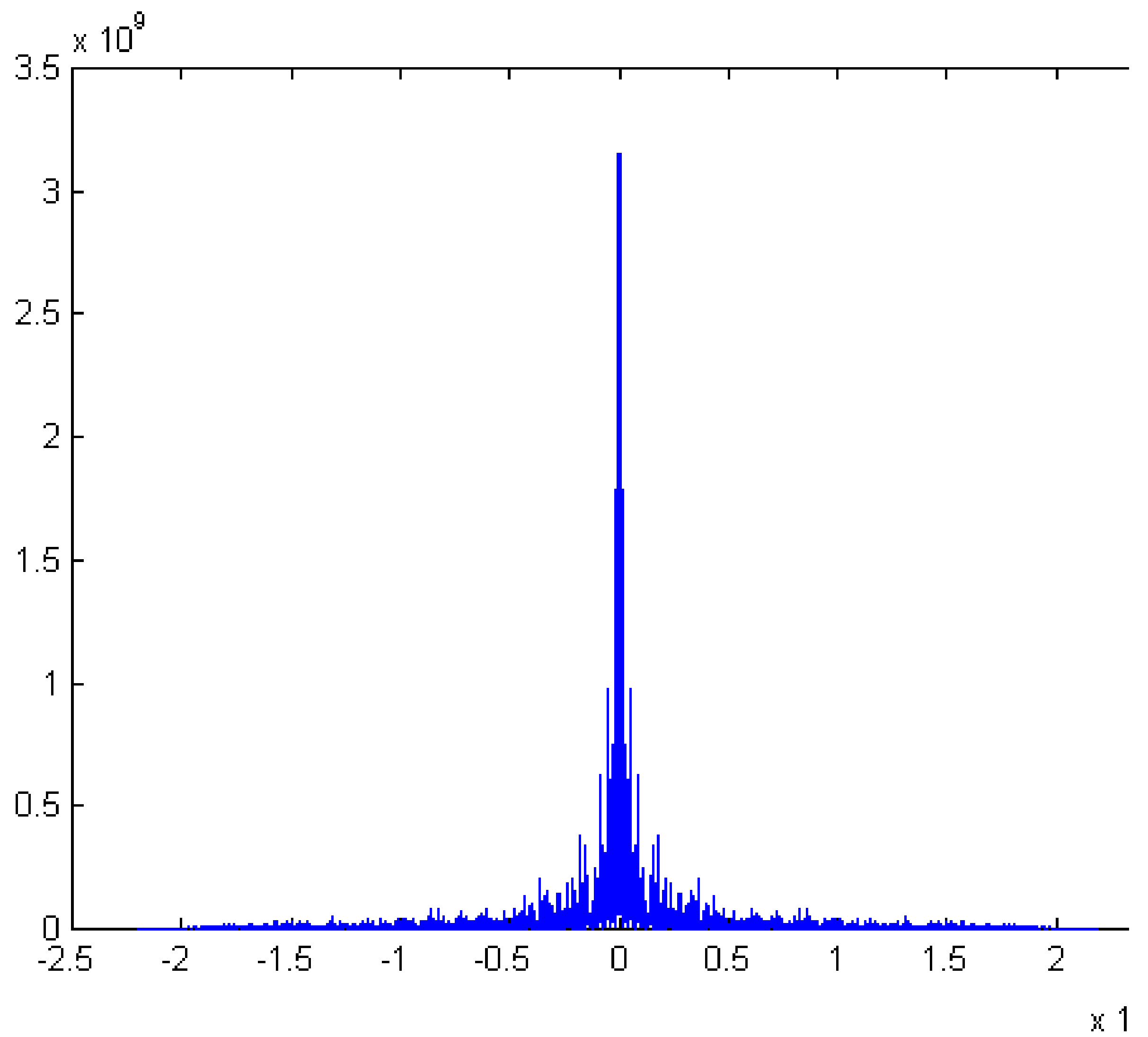
For optimization of the fft averaging, we were going to use dma transfer to copy the x array into the w array in order to keep all calculations on-chip; however, failure to get the project working precluded us from doing this. Some for loops were unraveled in order to make use of the parrallel functions of the processor, but once again we prefered to try to get a working product before trying to optimize the project too much. All libraries and function calls were placed off-chip in order to alleviate the problem of onchip-program overflow, however this seemed to have little affect on the speed of the function calls. Our fft averaging portion, averaged around 3 million cycles per fft.

### OTHER

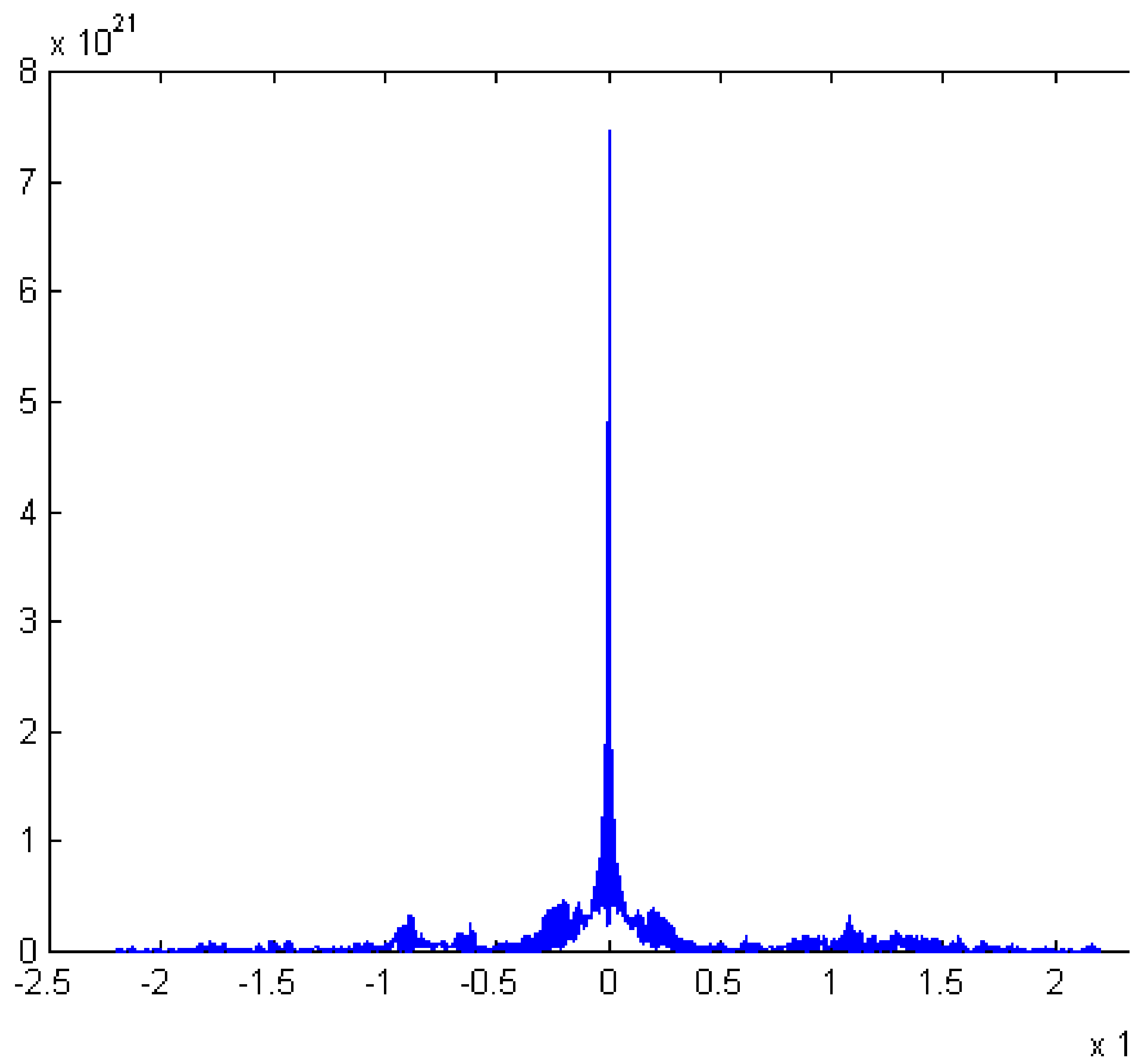
Currently still unable to produce usable output from the EVM board. We attempted solutions from previous lab assignments, and handwritten code. It is still not working however and the problem remains unknown. Currently produces sporadic bursts of output from the speakers.

## RESULTS

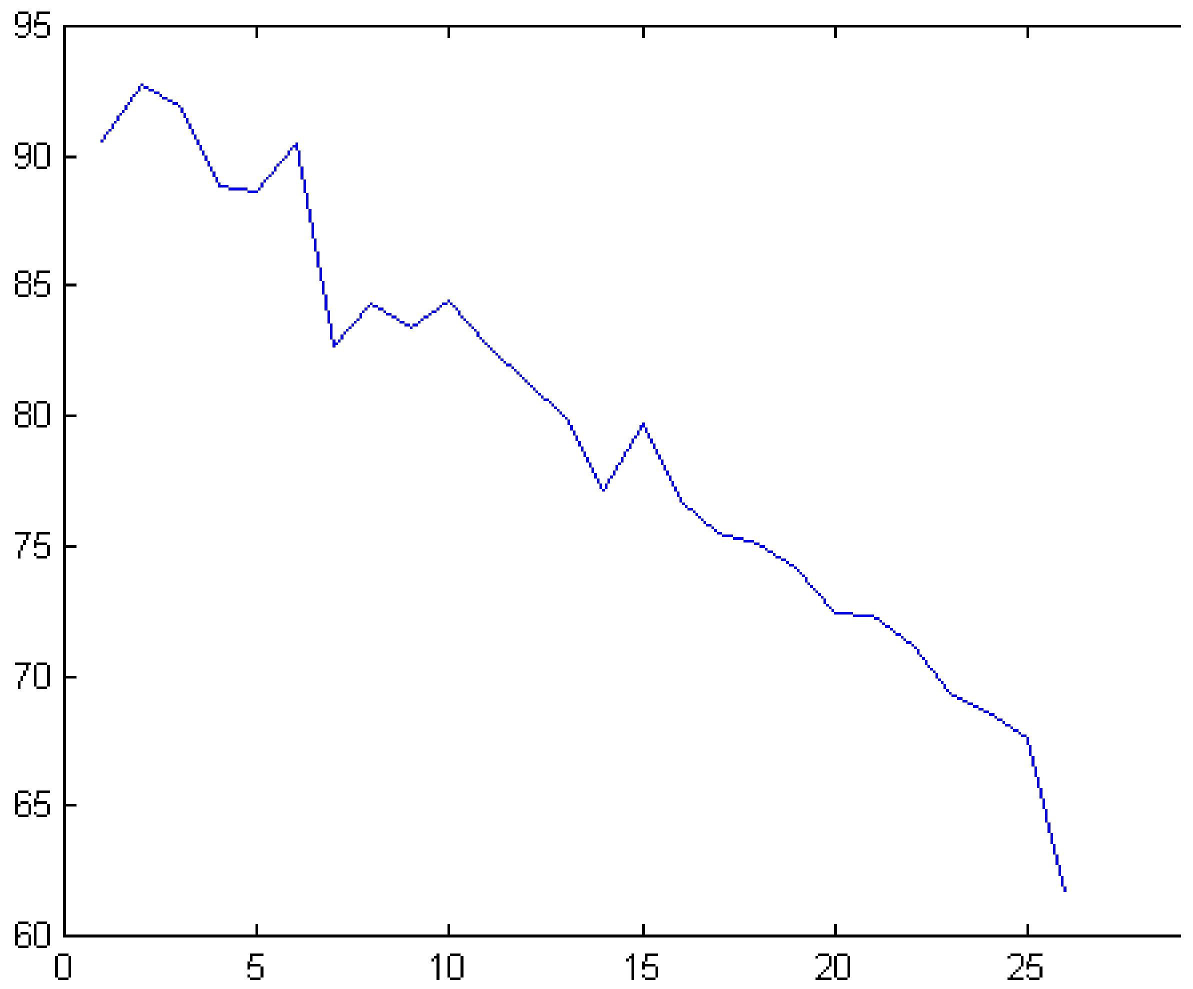
- FFT Signal




• Post Equalization, poor resolution, especially in low bands


x 10^21
8
7
6
5
4
3
2
1
0
-2.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
x 1

- Power Spectrum, after averaging, +/- 5db error



## HARDWARE PURCHASES

LinearX M31 General Purpose, Low Cost Measurement Microphone

$150 + shipping + handling + tax

LM 317 T Adjustable voltage Regulator, $4 + tax

XLR/8th in" adaptor, $11 + tax

## CODE

```
/***************************************************************/
/* 18-551 Project Group12                                      */
/*                                                             */
/* fft.c:              */
/*                                                             */
/***************************************************************/

/*-------------------------------------------------------------*/
/* INCLUDES AND LOCAL DEFINES                                  */
/*-------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>

#include <common.h>
#include <mcbsp.h>            /* mcbsp devlib */
#include <mcbspdrv.h>          /* mcbsp driver */
#include <codec.h>             /* codec library */
#include <board.h>            /* EVM library */
#include <mathf.h>            /* Math library */
#include <intr.h>              /* interrupt library */
#include <pci.h>
#include <dma.h>

#define BUFFER_LEN 8192        /* buffer length 8192 to allow real and
imaginary nums for FFT*/
#define NUM_FFTS 300       /* Number of FFTs to average*/
#define NUM_BANDS 26
#define MAX_FLUSH 10
```

```
/*----------------------------------------------------------------*/
/* GLOBAL VARIABLES                                                */
/*----------------------------------------------------------------*/


float buffer[BUFFER_LEN];        /* Input buffer and FFT buffer */
float w[BUFFER_LEN*3/4];              /* FFT twiddle factors (complex
interleaved) */
float g[NUM_BANDS];
int xindex = 0;                  /* transmit iterator */
int rindex = 0;                  /* receive iterator */
short buffer_full = FALSE;
short play = FALSE;
short buffer_flush = 0;
int bands[NUM_BANDS] =
{4,2,2,3,3,4,6,6,9,11,13,17,22,27,34,43,54,68,87,108,136,173,216,273,34
5,433};

int xq[NUM_BANDS][3];    /* x[n],x[n-1],x[n-2] */
int yq[NUM_BANDS][3];    /* y[n],y[n-1],y[n-2] */
int fi[NUM_BANDS][3];    /* i, j, k */
float fc[NUM_BANDS][5];  /* 0=a, 1=b, 2=c */
float diff[NUM_BANDS];

float s, out, w2, w1, w0, w0buffer;

float *longbuffer;
unsigned int lbi = 0;

float c[8]={-0.5928,0.8231,2,1,-0.9115,0.8364,-2,1};
float d[2]={0.0,0.0};

float gain[NUM_BANDS];
float preset[NUM_BANDS] = {102.5000, 101.5000, 100.5000, 99.5000,
98.5000, 97.5000,
      96.5000, 95.0000, 94.0000, 92.5000, 91.0000, 90.0000, 89.0000,
88.0000, 86.5000,
      85.0000, 84.0000, 83.0000, 82.0000, 81.0000, 79.5000, 78.0000,
77.0000, 76.0000,
      75.0000, 70.0000};

/* Function prototypes */
int request_transfer(void *buf, int size, int command);
int wait_transfer();
void cfftr4_dif(float* x, float* w, short n); /* Prototype for FFT
routine */

/*----------------------------------------------------------------*/
/* FUNCTIONS                                                       */
/*----------------------------------------------------------------*/

/* Interrupt vector to be called whenever a single sample of data is
   ready to be read.  For each sample, we simply store it in a buffer
   and increment the index into the buffer. */
void set_coeff(void)
{
      /* 31 hz center*/
```

```
  /* fc[0][0] = -0.9952;
  fc[0][1] = 1.9952;
  fc[0][2] = -0.0024;
  fc[0][3] = 0.0;
  fc[0][4] = .0024;
*/
/* 20 - 62 hz bandpass*/
  fc[0][0] = -0.9932;
  fc[0][1] = 1.9931;
  fc[0][2] = -0.0034;
  fc[0][3] = 0.0;
  fc[0][4] = 0.0034;

/* 78.7 hz center*/
  fc[1][0] = -0.9974;
  fc[1][1] = 1.9973;
  fc[1][2] = -0.0013;
  fc[1][3] = 0.0;
  fc[1][4] = 0.0013;

  /* 99.2 hz center*/
  fc[2][0] = -0.9967;
  fc[2][1] = 1.9965;
  fc[2][2] = -0.0016;
  fc[2][3] = 0.0;
  fc[2][4] = 0.0016;

  /* 125 hz center  */
  fc[3][0] = -0.9959;
  fc[3][1] = 1.9956;
  fc[3][2] = -0.0021;
  fc[3][3] = 0.0;
  fc[3][4] = 0.0021;

  /*158 hz center */
  fc[4][0] = -0.9948;
  fc[4][1] = 1.9943;
  fc[4][2] = -0.0026;
  fc[4][3] = 0.0;
  fc[4][4] = 0.0026;

  /*198 hz center */
  fc[5][0] = -0.9935;
  fc[5][1] = 1.9927;
  fc[5][2] = -0.0033;
  fc[5][3] = 0.0;
  fc[5][4] = 0.0033;

  /* 250 hz center*/
  fc[6][0] = -0.9918;
  fc[6][1] = 1.9905;
  fc[6][2] = -0.0041;
  fc[6][3] = 0.0;
  fc[6][4] = 0.0041;

  /*315 hz*/
  fc[7][0] = -0.9897;
```

```
fc[7][1] = 1.9877;
fc[7][2] = -0.0052;
fc[7][3] = 0.0;
fc[7][4] = 0.0052;

/*397 hz*/
fc[8][0] = -0.9870;
fc[8][1] = 1.9838;
fc[8][2] = -0.0065;
fc[8][3] = 0.0;
fc[8][4] = 0.0065;

/* 500 center*/
fc[9][0] = -0.9836;
fc[9][1] = 1.9786;
fc[9][2] = -0.0082;
fc[9][3] = 0.0;
fc[9][4] = 0.0082;

/*630 hz*/
fc[10][0] = -0.9794;
fc[10][1] = 1.9715;
fc[10][2] = -0.0103;
fc[10][3] = 0.0;
fc[10][4] = 0.0103;

/*794 hz */
fc[11][0] = -0.9742;
fc[11][1] = 1.9615;
fc[11][2] = -0.0129;
fc[11][3] = 0.0;
fc[11][4] = 0.0129;

/* 1 khz center*/
fc[12][0] = -0.9675;
fc[12][1] = 1.9476;
fc[12][2] = -0.0162;
fc[12][3] = 0.0;
fc[12][4] = 0.0162;

/*1.26 khz */
fc[13][0] = -0.9593;
fc[13][1] = 1.9278;
fc[13][2] = -0.0204;
fc[13][3] = 0.0;
fc[13][4] = 0.0204;

/*1.587 khz  */
fc[14][0] = -0.9490;
fc[14][1] = 1.8993;
fc[14][2] = -0.0255;
fc[14][3] = 0.0;
fc[14][4] = 0.0255;

/* 2 khz center*/
fc[15][0] = -0.9361;
fc[15][1] = 1.8580;
```

```
fc[15][2] = -0.0319;
fc[15][3] = 0.0;
fc[15][4] = 0.0319;

/* 2.52 khz   */
fc[16][0] = -0.9201;
fc[16][1] = 1.7976;
fc[16][2] = -0.0399;
fc[16][3] = 0.0;
fc[16][4] = 0.0399;

/*3.175 khz  */
fc[17][0] = -0.9004;
fc[17][1] = 1.7091;
fc[17][2] = -0.0498;
fc[17][3] = 0.0;
fc[17][4] = 0.0498;

/* 4 khz center*/
fc[18][0] = -0.8760;
fc[18][1] = 1.5791;
fc[18][2] = -0.0620;
fc[18][3] = 0.0;
fc[18][4] = 0.0620;

/*5.04 khz   */
fc[19][0] = -0.8462;
fc[19][1] = 1.3893;
fc[19][2] = -0.0769;
fc[19][3] = 0.0;
fc[19][4] = 0.0769;

/*6.350 khz  */
fc[20][0] = -0.8097;
fc[20][1] = 1.1158;
fc[20][2] = -0.0951;
fc[20][3] = 0.0;
fc[20][4] = 0.0951;

/* 8 khz center*/
fc[21][0] = -0.7656;
fc[21][1] = 0.7318;
fc[21][2] = -0.1172;
fc[21][3] = 0.0;
fc[21][4] = 0.1172;

/*  10.079 khz  */
fc[22][0] = -0.7126;
fc[22][1] = 0.2167;
fc[22][2] = -0.1437;
fc[22][3] = 0.0;
fc[22][4] = 0.1437;

/* 12.6 khz center*/
fc[23][0] = -0.6493;
fc[23][1] = -0.4182;
fc[23][2] = -0.1753;
```

```
      fc[23][3] = 0;
      fc[23][4] = 0.1753;

      /* 16 khz center*/
      fc[24][0] = -0.5745;
      fc[24][1] = -1.0804;
      fc[24][2] = -0.2127;
      fc[24][3] = 0;
      fc[24][4] = 0.2127;

      /* 20.15 khz center*/
      fc[25][0] = -0.5385;
      fc[25][1] = -1.5385;
      fc[25][2] = -0.2307;
      fc[25][3] = 0;
      fc[25][4] = 0.2307;

}

float filter(float in, int band)
{
  float out;

  xq[band][fi[band][0]] = in;
  fi[band][1] = fi[band][0] - 2;
  if( fi[band][1] < 0 ) fi[band][1] += 3;
  fi[band][2] = fi[band][0] - 1;
  if( fi[band][2] < 0 ) fi[band][2] += 3;
  yq[band][fi[band][0]] = 2 * (fc[band][0] * (xq[band][fi[band][0]] -
xq[band][fi[band][1]])
            + fc[band][2] * yq[band][fi[band][2]] - fc[band][1] *
yq[band][fi[band][1]]);
  out = yq[band][fi[band][0]];
  fi[band][0] ++;
  if (fi[band][0] > 2) fi[band][0] = 0;
  return out;
}

float butter(float in, int band)
{
      out = 0;
      w0buffer = 0;

      w2 = w1;
      w0buffer += fc[band][0] * w2;
      out += fc[band][2] * w2;

      w1 = w0;
      w0buffer += fc[band][1] * w1;
      out += fc[band][3] * w1;

      w0 = w0buffer + in;
      out += fc[band][4] * w0;


      return out;
}
```

```
interrupt void rcvISR(void)
{
  /* int data; */
  /* MCBSP0_DRR is the hardware register where samples are stored */
  /*data = MCBSP0_DRR;

  data = data ;*/
  float adc;
  adc = (float) ((signed short int)((MCBSP0_DRR & 0xFFFF0000) >> 16));
  s=adc;
  /*INTR_DISABLE(CPU_INT15);
  adc = biquad(2, c, d, adc);

  INTR_ENABLE(CPU_INT15);
  */
  buffer[rindex++] = adc;
  buffer[rindex++] = 0;

  if(buffer_flush < MAX_FLUSH)
  {
      if(rindex >= BUFFER_LEN)
      {
            rindex = 0;
            buffer_flush ++;
      }
  }
  else
  {
      /*if(rindex == BUFFER_LEN/2)
      {
            buffer_full = TRUE;
            INTR_DISABLE(CPU_INT15);
      }
      else*/
  if (rindex >= BUFFER_LEN)
      {
            rindex = 0;
            buffer_full = TRUE;
            if (play == FALSE) INTR_DISABLE(CPU_INT15);
      }
  }

/*  if ((rindex >= BUFFER_LEN) && (buffer_flush >= MAX_FLUSH)){
    rindex = 0;
    buffer_full = TRUE;
    INTR_DISABLE(CPU_INT15);
  }
  else if ((rindex >= (BUFFER_LEN/2)) && (buffer_flush >= MAX_FLUSH)){
      buffer_full = TRUE;
      INTR_DISABLE(CPU_INT15);
  }
  else if ((rindex >= BUFFER_LEN) && (buffer_flush < MAX_FLUSH)){
    rindex = 0;
    buffer_flush++;
  }     */
}
```

```
interrupt void xmitISR(void)
{

  /* MCBSP0_DXR is the hardware register for outgoing samples */

  float dac;
  int i;
  xindex +=2;
  dac = 0;

  if (xindex >= BUFFER_LEN)
      xindex = 0;

  for(i=0; i< NUM_BANDS; i++)
      {
            dac += pow(2, (gain[i]/3)) * butter(buffer[xindex], i);
      }
  MCBSP0_DXR=((((int)dac)&0x0000ffff) << 16) + ((int)dac)&0x0000FFFF;

}

/* This function creates the lookup table for digit reversing.  After
   it is run, revtable[n] equals the pairwise digit-reversal of n.
   n is the size of the FFT this table will be used for.*/
void mkrevtable(int n, int *revtable) {
  int bits, i, j, r, o;

  bits= (31 - _lmbd(1, n))/2;  /* _lmbd(1,n) finds leftmost 1 bit in n
*/
  for(i=0; i<n; i++) {
    r=0; o=i;
    _nassert(bits>=3);
    for(j=0; j<bits; j++) {
      r <<= 2;
      r |= o & 0x03;
      o >>= 2;
    }
    revtable[i] = r;
  }
}

/* Function for filling the table of FFT twiddle factors.  n is the
   size of the FFT to be used */
void fillwtable(int n) {
    int k;
/* Fill in the function to generate twiddle factors */
      for (k=0; k<(.75*n); k++){
            w[2*k] = cosf(2*PI*k/n);
            w[2*k+1] = sinf(2*PI*k/n);
      }
}



float lpf2k( float x )
{
```

```
float y,m;
static float history[10] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};

static float xcoeffs[10] = {
 1.4778561177e-09, 1.4778561177e-08, 6.6503525298e-08, 1.7734273413e-
07,
 3.1034978472e-07, 3.7241974167e-07, 3.1034978472e-07, 1.7734273413e-
07,
 6.6503525298e-08, 1.4778561177e-08
};
static float ycoeffs[10] = {
 1.6046712177e-01,-1.8969030586e+00, 1.0121269133e+01,-
3.2104063578e+01,
 6.7050122622e+01,-9.6360744485e+01, 9.6524146934e+01,-
6.6558160358e+01,
 3.0242734993e+01,-8.1788678098e+00
};
static int put = 0, serve = 1;
int i, s;

    m = x;
    y = 0.0;

    s = serve;
    for( i=0; i<10; i++)
    {
        m -= history[s] * ycoeffs[i];
        y += history[s++] * xcoeffs[i];
        if( s >= 10 ) s = 0;
    }

    y += m * xcoeffs[0];

    history[put++] = m; serve++;
    if( put >= 10 ) put = 0;
    if( serve >= 10) serve = 0;

    return (float) y;
}

void makelong(void)
{
      int i;
      /* use DMA speedup ? */
      for(i = 0; i < BUFFER_LEN; i++)
            longbuffer[lbi+i] = buffer[i];

      lbi += BUFFER_LEN;
      if(lbi == 4*BUFFER_LEN) lbi = 0;
}

void decimate(void)
{
      int i;
      for(i=0; i < BUFFER_LEN; i+=2)
      {
            buffer[i] = longbuffer[i*4];
```

```
            buffer[i+1] = 0;
      }
}

/******* MAIN ********/
int main(void)
{
  long int i, j, offset;   /* loop vars */
  int rev_i;               /* digit reversal var */
  float *x;                /* pointer to fft average */
  int *revtable;           /* pointer to revtable */
  float dac;
  float avg;

  Mcbsp_dev dev;
  evm_init();               /* Initialize the board */
  mcbsp_drv_init();         /* Call this before using McBSP functions
*/
  pci_driver_init();        /* Call before using any PCI code */

  DMA_AUXCR = 0x00000010;      /* Set priority of HPI over CPU to avoid
crashing */

  w2 = 0;
  w1 = 0;
  w0 = 0;

  revtable = malloc((sizeof(int))*(BUFFER_LEN/2));  /* Look-up table
for digit-reversing */
  x = malloc((sizeof(float))*BUFFER_LEN);              /* FFT output
average (complex interleaved) */
  longbuffer = malloc(sizeof(float)*4*BUFFER_LEN);

  for (i=0; i < BUFFER_LEN; i++) x[i] = 0;

  /*************** Initialise the filter variables *************/
  for (i=0; i < NUM_BANDS; i++){
      for (j = 0; j < 3; j++)
      {
        xq[i][j] = 0;
      yq[i][j] = 0;
      fi[i][j] = 0;
    }
  }
  set_coeff();


  /****************** McBSP setup *************************/
  dev= mcbsp_open(0);          /* Open serial port */
  if (dev == NULL){              /* Error checking */
    printf("Error opening MCBSP 0\n");
    return(ERROR);
  }

  /* Configure McBSP */
  mcbsp_setup(dev);   /* See bottom of this file */
  printf("Serial port setup complete\n");
```

```
  /******************* configure CODEC *********************/
  /* EXIT_ERROR is a macro which jumps to exit_err if the function
     returns an ERROR */
  EXIT_ERROR(codec_init());

  codec_change_sample_rate(44100, TRUE);
  /*codec_audio_control_format(LINEAR_16BIT_SIGNED_LE,TRUE,BOTH);    */
  codec_interrupt_enable();
  codec_playback_enable();

  /* A/D 0.0 dB gain, turn off 20dB mic gain, sel (L/R)LINE input */
  EXIT_ERROR(codec_adc_control(LEFT,0.0,FALSE,LINE_SEL));
  EXIT_ERROR(codec_adc_control(RIGHT,0.0,FALSE,LINE_SEL));

  /* mute (L/R)LINE input to mixer */
  EXIT_ERROR(codec_line_in_control(LEFT,0.0,TRUE));
  EXIT_ERROR(codec_line_in_control(RIGHT,0.0,TRUE));

  /* D/A 0.0 dB atten, do not mute DAC outputs */
  EXIT_ERROR(codec_dac_control(LEFT, 0.0, FALSE));
  EXIT_ERROR(codec_dac_control(RIGHT, 0.0, FALSE));

  printf("Codec setup complete\n");


  /**************** setup interrupt routines ******************/
  intr_init();

  /* Hook up serial transmit interrupt to CPU Interrupt 14 */
  intr_map(CPU_INT14,ISN_XINT0);

  INTR_CLR_FLAG(CPU_INT14);     /* Clear any old interrupts */
  intr_hook(xmitISR,CPU_INT14); /* Hook our own xmitISR into chain for
14 */

  /* Repeat the same process for the receive interrupt */
  intr_map(CPU_INT15,ISN_RINT0);
  INTR_CLR_FLAG(CPU_INT15);
  intr_hook(rcvISR,CPU_INT15);

  /* Enable all necessary interrupts */
  INTR_ENABLE(CPU_INT_NMI);     /* Non-maskable interrupt */
  /*INTR_ENABLE(CPU_INT15);*/   /* Do not enable this interupt yet? */
  INTR_GLOBAL_ENABLE();         /* Controls whether ANY interrupts
function */

  printf("Interrupt setup complete\n");



  mkrevtable(BUFFER_LEN/2, revtable); /* Make the digit-reversal look-
up table */
  fillwtable(BUFFER_LEN/2);           /* Make the table of FFT twiddle
factors */
```

```
  MCBSP_ENABLE(dev->port,MCBSP_RX|MCBSP_TX);


  /********************* Time Averaged FFT ******************/

  for (i=0; i< NUM_FFTS; i++){
      INTR_ENABLE(CPU_INT15);                /* start receive interupt
handler */
      buffer_full = FALSE;
      while (buffer_full == FALSE){}

      /*if (i == 15){
            request_transfer(buffer, (BUFFER_LEN/2)*4, 0x02);
            wait_transfer();
      }           */


      /****** start decimation FFT code ******/

      /*makelong();    */

      /****** end decimation FFT code ******/

           /*
      if((i > 0) && (i % 4 == 0))    {

      for(j=0; j<4*BUFFER_LEN; j+=2)
            longbuffer[j] = lpf2k(longbuffer[j]);

      decimate();
      cfftr4_dif(buffer, w, BUFFER_LEN/2);
      }    */

      /*for(j=0; j < BUFFER_LEN; j+=2)
            buffer[j] = lpf2k(buffer[j]);  */

      cfftr4_dif(buffer, w, BUFFER_LEN/2);
      for (j=0; j<BUFFER_LEN; j++){
        x[j] = buffer[j] + x[j];          /* average with old fft's and
store offchip */
      }


  }

  for (i=0; i < BUFFER_LEN; i++){
      x[i] = x[i] / NUM_FFTS;
  }
  printf("averaging complete\n");

  /* Example of digit-reversal.  Remember you want the magnitude
squared. */
  for(i=0; i< (BUFFER_LEN/2); i++) {
    rev_i = revtable[i];      /* rev_i is now the digit-reversal of i
*/
```

```
      buffer[i] = x[2*rev_i]*x[2*rev_i] + x[2*rev_i + 1]*x[2*rev_i +
1];
      /* real part is  x[2*rev_i] */
      /* Imaginary part is x[2*rev_i + 1] */
  }

 request_transfer(buffer, (BUFFER_LEN/2)*4, 0x02);
 wait_transfer();

  offset = 1;
  for (i=0; i<NUM_BANDS; i++)
  {
      g[i]=0;
      for(j=0; j < bands[i]; j++)
      {
            g[i]+= buffer[offset + j];
      }
      g[i] = g[i] / bands[i];
      offset += bands[i];

      g[i] = 10*log10f(g[i]);
      diff[i] = g[i];
      g[i] = preset[i] - g[i];/* move this to send back 30band
response*/
  }
  request_transfer(diff, (BUFFER_LEN/2)*4, 0x02);
  wait_transfer();

  /* normalize gains */
  avg = 0;
  for (i=0; i<NUM_BANDS; i++)
  {
      avg += g[i];
  }
  avg = avg / NUM_BANDS;

  for (i=0; i<NUM_BANDS; i++)
  {
      g[i] = g[i] - avg;
  }


  /* 3db tolerance */
  for (i=0; i<NUM_BANDS; i++)
  {
      if ((g[i] < 3) && (g[i] > -3)) gain[i] = 0;
      else if (g[i] < 12) gain[i] = g[i] - 2;
      else if (g[i] > -12) gain[i] = g[i] + 2;
      else if (g[i] > 12) gain[i] = 12;
      else if (g[i] < -12) gain[i] = -12;
  }


  request_transfer(g, (BUFFER_LEN/2)*4, 0x02);
  wait_transfer();
  request_transfer(gain, (BUFFER_LEN/2)*4, 0x02);
  wait_transfer();
```

```
  rindex = 0;
  INTR_ENABLE(CPU_INT15);
  buffer_full = FALSE;
  while(buffer_full == FALSE);
  for (j = 0; j < BUFFER_LEN; j+=2){
      x[j] = 0.2512 * buffer[j];
      for(i=0; i< NUM_BANDS; i++)
      {
            if(gain[i] != 0)
            {
            /*    if(gain[i] > 0) x[j] += (exp10f(gain[i]/20)/4) *
butter(buffer[j], i);
                  else x[j] -= exp10f(gain[i]/20) * butter(buffer[j],
i);*/
            }
      }
      x[j+1] = 0;
  }

  cfftr4_dif(x, w, BUFFER_LEN/2);

  for(i=0; i< (BUFFER_LEN/2); i++) {
    rev_i = revtable[i];      /* rev_i is now the digit-reversal of i
*/
      buffer[i] = x[2*rev_i]*x[2*rev_i] + x[2*rev_i + 1]*x[2*rev_i +
1];
      /* real part is  x[2*rev_i] */
      /* Imaginary part is x[2*rev_i + 1] */
  }
  request_transfer(buffer, (BUFFER_LEN/2)*4, 0x02);
  wait_transfer();

  offset = 1;
  for (i=0; i<NUM_BANDS; i++)
  {
      g[i]=0;
      for(j=0; j < bands[i]; j++)
      {
            g[i]+= buffer[offset + j];
      }
      g[i] = g[i] / bands[i];
      offset += bands[i];

      g[i] = 10*log10f(g[i]);
  }

  rindex = 0;
  play = TRUE;
  INTR_ENABLE(CPU_INT15);
  INTR_ENABLE(CPU_INT14);



 /*request_transfer(w, (BUFFER_LEN/2)*4, 0x02);
 wait_transfer();*/
```

```
 free(revtable);
 free(x);

 /*INTR_ENABLE(CPU_INT14);
 INTR_ENABLE(CPU_INT15);*/


 request_transfer(NULL, 0, 0x04);
 wait_transfer();
 printf("End Task\n");

 /*return 0;*/

 exit_err:
 return 1;
}


int mcbsp_setup(Mcbsp_dev dev)
{
  /* Structure with all configuration parameters for serial port */
  Mcbsp_config mcbspConfig;

  memset(&mcbspConfig,0,sizeof(mcbspConfig)); /* Initialize everything
to 0 */

  mcbspConfig.loopback              = FALSE;

  mcbspConfig.tx.update             = TRUE;
  mcbspConfig.tx.clock_polarity     = CLKX_POL_RISING;
  mcbspConfig.tx.frame_sync_polarity= FSYNC_POL_HIGH;
  mcbspConfig.tx.clock_mode         = CLK_MODE_EXT;
  mcbspConfig.tx.frame_sync_mode    = FSYNC_MODE_EXT;
  mcbspConfig.tx.phase_mode         = SINGLE_PHASE;
  mcbspConfig.tx.frame_length1      = 0;
  mcbspConfig.tx.word_length1       = WORD_LENGTH_32;
  mcbspConfig.tx.frame_ignore       = FRAME_IGNORE;
  mcbspConfig.tx.data_delay         = DATA_DELAY0;

  mcbspConfig.rx.update             = TRUE;
  mcbspConfig.rx.clock_polarity     = CLKR_POL_FALLING;
  mcbspConfig.rx.frame_sync_polarity= FSYNC_POL_HIGH;
  mcbspConfig.rx.clock_mode         = CLK_MODE_EXT;
  mcbspConfig.rx.frame_sync_mode    = FSYNC_MODE_EXT;
  mcbspConfig.rx.phase_mode         = SINGLE_PHASE;
  mcbspConfig.rx.frame_length1      = 0;
  mcbspConfig.rx.word_length1       = WORD_LENGTH_32;
  mcbspConfig.rx.frame_ignore       = FRAME_IGNORE;
  mcbspConfig.rx.data_delay         = DATA_DELAY0;

  /* Pass entire structure to mcbsp_config, a library function which
     sets registers according to the contents of the structure */
  if(mcbsp_config(dev,&mcbspConfig) != OK) {
        printf("Couldn't configure McBSP device %i\n", dev);
        return(ERROR);
  }
```

```
  return(OK);
}

/* Use mailbox 1 for address, 2 for size, and 3 for command */
int request_transfer(void *buf, int size, int command)
{
    amcc_mailbox_write(2, size);
    amcc_mailbox_write(3, command);
      pci_message_sync_send((unsigned int)buf, FALSE);
      return(0);
}

/* The PC will send a message when the transfer is complete.  Wait
      for that to happen */
int wait_transfer()
{
      unsigned int value;

      pci_message_sync_retrieve(&value);
      return(value);
}
/**********************************************/
/*          (PC side)                         */
/*                                            */
/*  Last Modified: '5/1/00 10:51:46 AM'       */
/**********************************************/

#include <stdio.h>
#include <windows.h>
#include "evm6xdll.h"

/* Buffer size */
#define F_SIZE 8192


#undef LOAD_FILE
//#define LOAD_FILE    /* Uncomment to automatically load program */

/* Define these to match your setup */
#define INFILE "c:\\551\\group12\\zeros.raw"
#define OUTFILE "c:\\551\\group12\\results"
#define EVM_FILE "c:\\551\\group12\\echo.out"

/* Structure for holding transfer information */
struct transfer_s {
      void *buffer;
      unsigned long size;
      unsigned long command;
};

/* Function prototypes */
#ifdef LOAD_FILE
int load_file(HANDLE hBd, LPVOID hHpi);
#endif
int wait_request(struct transfer_s *ts);
int send_data(struct transfer_s *ts, void *local_buf);
int get_data(struct transfer_s *ts, void *local_buf);
```

```
void hpi_write_word(ULONG addr, ULONG data);

/* Global vars */
float result[F_SIZE];                     /* Output */

HANDLE hBd    = NULL;
LPVOID hHpi = NULL;
HANDLE h_event;


/* Writes 'result' to a file.  number is the integer to append to the
      end of the filename */
void data_to_file(int number)
{
      FILE *outfile;
      char fname[255];

      sprintf(fname, "%s%i.raw", OUTFILE, number);
      outfile=fopen(fname, "wb");
      fwrite(result, F_SIZE, sizeof(short int), outfile);
      fclose(outfile);
}

int main(int argc, char **argv) {
      int program_exit=0, i=0;
      struct transfer_s ts;

      /* Open the board */
      hBd = evm6x_open(0, 1);
      if ( hBd == INVALID_HANDLE_VALUE ) {
            fprintf(stderr, "Couldn't open board\n");
            exit(1);
      }
      fprintf(stderr, "Opened connection to board...\n");

      /* Also open connection to HPI */
    hHpi = evm6x_hpi_open(hBd);
    if ( hHpi == NULL ) exit(4);

#ifdef LOAD_FILE
      load_file(hBd, hHpi);
      fprintf(stderr, "Loaded program...\n");
#else
      /* Set DMA AUX priority greater than CPU priority, so we
            don't lock the PCI bus */
    hpi_write_word(0x01840070 /*Addr*/, 0x00000010 /*Data*/);

      /* Reset the mailboxes and FIFO */
      hpi_write_word(0x0170003C, 0x0e000000);
#endif

      /* Setup a windows event so we don't have to poll for incoming
            messages */
      evm6x_clear_message_event(hBd);
      sprintf( s_buffer, "%s%d",EVM6X_GLOBAL_MESSAGE_EVENT_BASE_NAME,
0);
      h_event = OpenEvent( SYNCHRONIZE, FALSE, s_buffer );
```

```
	/* Main loop... Waits for messages from the EVM and does a
transfer
		depending on the value of the message received */
	while(!program_exit) {
		wait_request(&ts);
		fprintf(stderr, "Transfer request: CMD %x, SIZE %i, ADDRESS
%x\n", ts.command,
			ts.size, ts.buffer);

		/* Now based on the command that was sent, do something */
		switch(ts.command) {
		case(0x01): /* Send frame */
			if(ts.size != F_SIZE) fprintf(stderr, "Wrong
size!!!\n");
			send_data(&ts, frame);
			break;
		case(0x02): /* Get frame */
			if(ts.size != F_SIZE*sizeof(short int))
				fprintf(stderr, "Wrong size\n");
			get_data(&ts, result);
			data_to_file(++i);	/* Output frame to a new file
*/
			break;
		case(0x03): /* Print string */
			get_data(&ts, temp);
			printf("%s\n", temp);
			break;
		case(0x04): /* Exit program */
			program_exit = 1;
			break;
		default:
			fprintf(stderr, "Unknown command\n");
			break;
		}
	}

	/* Clean up and exit */
	if (!evm6x_hpi_close(hHpi)) exit(9);
	if (!evm6x_close(hBd)) exit(16);
	return(0);
}

/* Waits for windows event signalling incoming message.  Then reads
	address from mailbox 1, size from mailbox 2, and command from
	mailbox 3 */
int wait_request(struct transfer_s *ts)
{
	/* wait for event signaling a message from the DSP */
	WaitForSingleObject( h_event, INFINITE );

	if(!evm6x_retrieve_message(hBd, (unsigned long *)&ts->buffer))
		fprintf(stderr, "Error retrieving 1...\n");
	if(!evm6x_mailbox_read(hBd, 2, (unsigned long *)&ts->size))
		fprintf(stderr, "Error retrieving 2...\n");
	if(!evm6x_mailbox_read(hBd, 3, (unsigned long *)&ts->command))
		fprintf(stderr, "Error retrieving 3...\n");
```

```
	return(0);
}

/* Size and address are given in struct ts.  local_buf is the address
of
	the PC buffer to send */
int send_data(struct transfer_s *ts, void *local_buf)
{
	unsigned long int ulLength = ts->size;

	/* Write the data to EVM memory*/
	if (!evm6x_hpi_write(hHpi, (PULONG)local_buf, (PULONG)&ulLength,
(ULONG)ts->buffer)) {
		fprintf(stderr, "HPI write error\n");
		exit(1);
	}
	if (ulLength != ts->size) {
		fprintf(stderr, "HPI only wrote %i bytes\n");
		exit(1);
	}

	/* Use a message to signal the EVM that the transfer is done */
	if (!evm6x_send_message(hBd, (PULONG)&ts->command)) {
		fprintf(stderr, "Send message error!\n");
		exit(1);
	}
	return(0);
}

/* Same format as send_data */
int get_data(struct transfer_s *ts, void *local_buf)
{
	unsigned long int ulLength = ts->size;

	/* Read data from EVM memory */
    if (!evm6x_hpi_read(hHpi, (PULONG)local_buf, (PULONG)&ulLength,
(ULONG)ts->buffer)) {
		fprintf(stderr, "HPI write error\n");
		exit(1);
	}
	if (ulLength != ts->size) {
		fprintf(stderr, "HPI only wrote %i bytes\n");
		exit(1);
	}

	/* Signal EVM that transfer is done */
	if (!evm6x_send_message(hBd, (PULONG)&ts->command)) {
		fprintf(stderr, "Send message error!\n");
		exit(1);
	}
	return(0);
}

/* Write a single 32-bit word to EVM memory */
void hpi_write_word(ULONG addr, ULONG data)
{
```

```
        ULONG ulLength = 4;

        if (!evm6x_hpi_write(hHpi, &data, &ulLength, addr)) {
              fprintf(stderr, "Error writing word via HPI\n");
              exit(1);
        }
        if (ulLength != 4 ) {
              fprintf(stderr, "Error writing word via HPI\n");
              exit(1);
        }
}

#ifdef LOAD_FILE
/* Load the .out file and run the program.  Code Composer must not
        be running */
int load_file(HANDLE hBd, LPVOID hHpi)
{
  if ( !evm6x_reset_board(hBd) ) exit(2);
  if ( !evm6x_reset_dsp(hBd,HPI_BOOT) ) exit(3);
  if ( !evm6x_init_emif(hBd, hHpi) ) exit(5);

  /* Load program */
  if (!evm6x_coff_load(hBd,hHpi,EVM_FILE,0,0,0)) exit(8);

  /* Set HPI priority and reset mailboxes */
  hpi_write_word(0x01840070 /*Addr*/, 0x00000010 /*Data*/ );
  hpi_write_word(0x0170003C, 0x0e000000);

  if (!evm6x_unreset_dsp(hBd)) exit(10);
  if (!evm6x_set_timeout(hBd,1000)) exit(11);
        return(0);
}
#endif
```