DAFX: Digital Audio Effects

EECS 452, Winter 2005
Final Report

Ross Penniman
Joseph Heremans
James Wang
Amanda Wallace
Jason Martin

Digital Audio Effects: Final Report

Motivation:

The main goal of this project was to design and implement a digital audio effects processor that could offer several different audio effects. Using the skills that our group learned in this course, along with other courses that we have taken, our team wanted to develop a project that has aesthetic and technical appeal to a wide range of audiences. Our team goal for our digital audio effects processor was to design something that is capable of operating in real time and still maintain good sound quality throughout the processing of an audio signal.

Commercial Value and Comparison:

With the high revenues the music industry has produced over the years, technology has played a major part in allowing music producers to continue to make innovative sounds that will be remembered for the ages. There are many digital audio effects processors on the market with a wide range of prices. Digital audio processors range in price from $200.00 with the Alesis MidiVerb 4, up to $8,100.00 with the TC Electronic M6000. In the development of our digital audio effects processor, we saw the need for a more cost effective processor that could be targeted for novice music producers and/or people who are looking for great quality at an economical price. Therefore, our team feels that the digital audio effects processor that we have designed would have great commercial value with the potential to be very successful.

User Interface:

The digital audio effects processor that our team has designed uses a knob box to interface between the user and the DSK. The user has the option of selecting from eight effects by pressing a key on the knob box keypad. With a selected audio effect, the user has the option to vary the different parameters of the audio effect by either incrementing or decrementing the parameters with a turn of one of the four knobs on the knob box.

Hardware Implementation:

The hardware implementation required a complex network of different devices working synchronously to receive the proper data. To input effect and parameter changes, a knob box was selected with 4 Panasonic Square Encoder Knobs and a 16 button keypad. The information would be passed from the knob box, through the complex programmable logic device (CPLD), into the 8051 microcontroller, then via the serial port to the RS-232 daughter board, which is connected to the DSK. Finally, the sound effect with the proper parameters will be output to the speakers (see Figure 1). Each device required trivial programming, but the overall integration of the hardware system proved to be quite difficult.

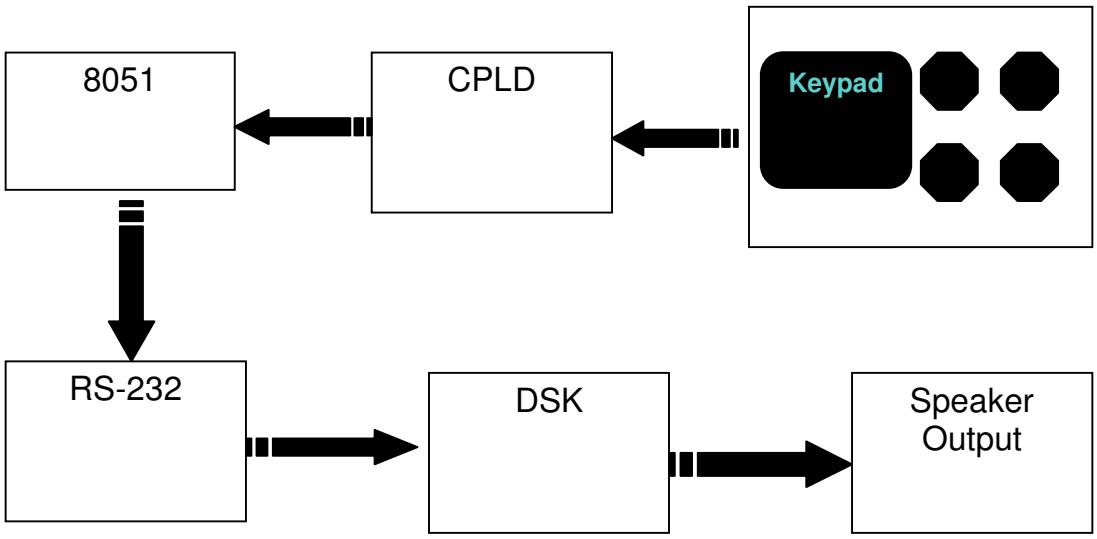


Figure 1: Hardware Overview

Knob Box:

The knob box was originally made by Dr. Metzger and had been slated to be added to the previous DSK but was never implemented. The box consisted of 4 square encoder knobs as well as one 16 button keypad.

Knobs:

Each square encoder knob has two signals associated with it, signal A and signal B. The relation between the two signals is dependent on the movement and direction of the knob. If the knob is turned in a counterclockwise direction signal A will lead signal B, which will later be decoded by the 8051 as a decrease or counterclockwise rotation (see Figure 2). In the case where signal B is leading signal A, the decoding program will signal that the knob has turned clockwise, or has increased in value. In the case where neither signal is leading the other, the knob has not changed position and no change needs to be made.

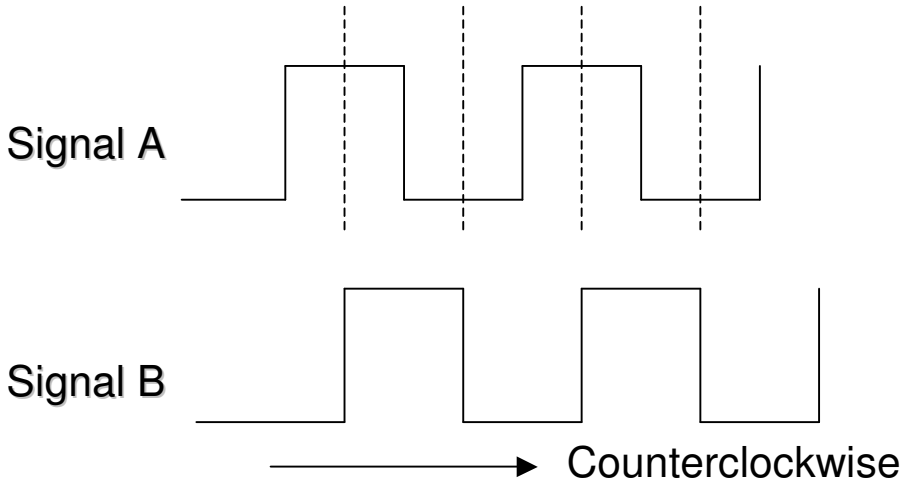


Figure 2: A counterclockwise rotation of an Encoder Knob

Keypad:

The keypad uses the idea of cross-pinning to form a connection. Essentially a 4-by-4 grid is made (comprised of the 16 keys) and a connection of two pins causes both pins to go high. The process is sequentially writing to each of the 4 columns and then reading the entire 8 bit number returned by the keypad (see Figure 3). The decode program, on the 8051 microcontroller, will then associate different 8-bit combinations with which key was being pressed.

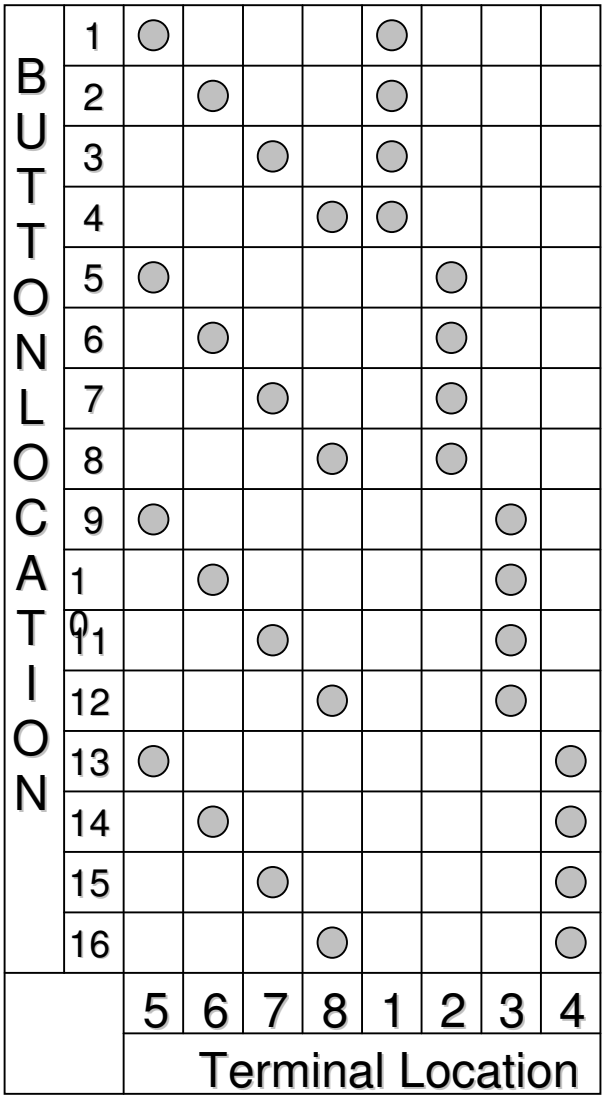


Figure 3: A button to terminal location translation diagram of the keypad

CPLD:

The CPLD did not really do any calculations or bit manipulations. Its sole purpose was to connect the knob box to the 8051 microcontroller. Therefore, through a bit of logical programming we did a simple input-output connection linking the knob box to the 8051 microcontroller.

8051:

The 8051 microcontroller is used to relay data from the CPLD to the DSK. For controlling the knobs, the 8051 simply reads in the data bits. For controlling the keypad, the 8051 has to read data bits from and write data bits to the CPLD. In our application, the 8051 is programmed to translate these data into the more workable ASCII codes and send them to the DSK. The 8051 is software-coded in C and physically programmed onto the chip using an EMP Device Programmer.

Serial Receiving Code:

The software in the DSK reads in three ASCII characters outputted by the 8051 microcontroller every time a knob is turned or keypad is pressed. The first character, denoted by 'K' or 'P', tells the DSK whether it is a knob or keypad that is being activated, respectively. The second character says which keypad or which knob is pressed or turned; it can be one of 16 keypads or one of 4 knobs. The last character for a knob turn, denoted by a 'U' for up and a 'D' for down, tells the DSK if the knob is being incremented or decremented. For the keypad, this character is an arbitrary 'X', which is discarded by the DSK in processing. After the different parameter information has been updated, the function for the audio effect that was selected will be called with new parameters.

Hardware Challenges:

Hardware challenges include keypad and knob sensitivity and switch bounce issues. Testing showed that keypads and knobs are too sensitive. They would sometimes transmit superfluous bits of information that the DSK could not handle all at once. To address this problem, a pre-scaler was implemented in the software. The pre-scaler discards the first three out of every four transmissions, reducing the resolution of the knobs and keypads but increasing their controllability.

Switch bounce also may affect the precision of the application. It may occur when a keypad is pressed and the switch doesn't make and break cleanly on the time scales of digital systems. Instead, a typical switch makes multiple transitions during the tens of milliseconds required to open or close, due to effects that include age, operating inertia, mechanical design, and the microscopic condition of the switch-contact surfaces. In our testing, this issue did not seem to have a noticeable effect on the applications and therefore is not explicitly addressed. However, it can be a project for students who may want to use the knob box in their applications in the future.

Software Implementations:

All effects in the DAFX project are implemented using C. The algorithms have been tested and verified using Matlab first, and then translated into C for real-time application. Most of the algorithms we used are based on those provided in the book *DAFX*, edited by Udo Zolzer. A few others such as flange and tremolo were found online. All of the block diagrams can be found in Appendix A. Matlab prototype source code is in Appendix B and the final working C source code is in Appendix C.

Flange/Chorus:

The flange and chorus effects are based on the same algorithm. The two differ only in the parameter values. The algorithm involves introducing a variable delay into the input signal. Two parameters need to be passed into the application: variation range and rate of variation. Variation range, usually from 0 to 8 milliseconds, essentially denotes the size of the delay buffer. Rate of variation, usually from 0.3 to 1 Hz, is simply the rate at which the delay varies. For a flange effect, the variation range will take on a lower input value; for a chorus effect, the buffer size is generally larger, so a larger variation range is required.

Panning/Ping Pong:

Panning or ping pong is called such because it causes the output to pan from one speaker to another and back again. This effect takes in only one input – "vary" which ranges from 0 to 32767. This parameter is added to a counter which causes the rate of the panning to speed up for large values and slow down for small values. To implement this, we have a panning variable that changes at a rate determined by "vary". This panning variable is in the range 0 to .5. We add this panning variable to .5 to get a resulting magnitude for the left output and we subtract this panning variable from .5 to get a resulting magnitude for the right output. We increment the panning variable by the counter described above until it reaches .5, then we decrement the panning variable by the counter until it reaches 0. This linear changing of output magnitudes is what creates this panning effect.

Phaser:

Phaser is where we take the input and add a delayed version of the input to it to get the resulting output. The input parameters for this are "delay" and "multFactor". "Delay" ranges from 0 to 100 ms and determines how large of a delay we want to use. "MultFactor" ranges from .5 to 1 and determines the loudness of the delayed input. This effect is called phaser because it involves a small delay time which means the input and delayed signal are heard as one sound but with a significant amount of phase cancellation. To implement the delay, we use a delay buffer of the length delay*Fs+1. We use this as a circular buffer where we have two pointers, one pointing to the sample at the desired delay and one pointing to the oldest value where we want the new sampled value to be written. Then we simply add the sample at the desired delay to the current input sample, and write the input sample to the buffer.

Echo:

Another effect we have is echo. It outputs a delayed version of the signal which is recycled and attenuated, producing the sound of many echoes. This is implemented almost the same way as the phaser except that instead of putting just the input into the delay buffer, it puts the entire resulting output into the delay buffer. Echo takes in four input parameters: "delay", "multFactor", "dMix", and "wMix". "Delay" and "multFactor" have the same ranges and work the same way as in the phaser. The parameter "dMix", or dry mix, ranges from 0 to 1 and determines the resulting magnitude of the input sample. The parameter "wMix", or wet mix, ranges from 0 to 1 and determines the resulting magnitude of the delayed signal. A delay buffer is set up the same way as in the phaser. After adding the modified input to the delayed input, the result is then stored back into the buffer. Since the sound in the feedback loop is reduced by multFactor, the sound will decay with each echo.

Ring Modulation:

Ring modulation takes the input and multiplies it by a cosine wave: $\cos(2*\pi*f_0/Fs*t)$. The value of f_0 is determined by the input parameter which ranges from 50 to 250 Hz. Multiplying by a cosine wave introduces a frequency component into the resulting output spectrum above and below the modulating frequency (see Figure 4). This causes the output to have a bell-like sound as it will make any input spectra inharmonic. The implementation of this algorithm is straight forward. We simply calculate a value for the cosine where t is a counter that increments for each new sample. When the f_0*t/Fs reaches 1, we reset t back to zero since this is one period of the cosine wave. This is to prevent distortion from occurring when the value of t overflows. Then we multiply the input by this resulting cosine value and output it to the DSK.

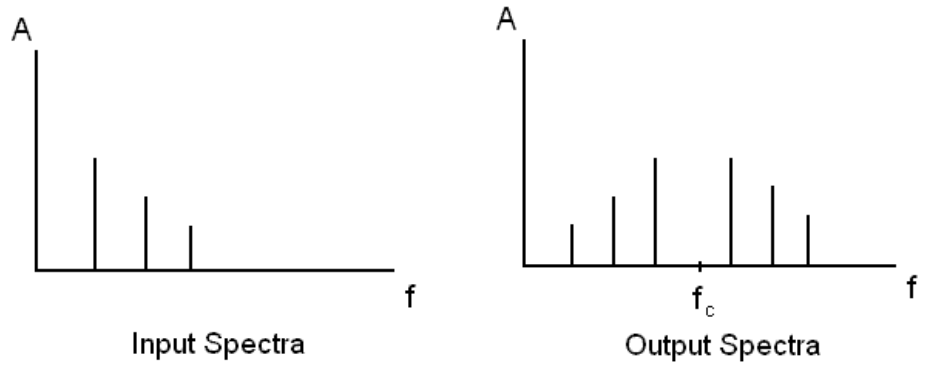


Figure 4: Ring modulation creates an output spectrum in which the input spectrum is mirrored around the modulating frequency (f_c)

Tremolo:

Tremolo varies the loudness of the output with a cosine wave. This sounds equivalent to turning the volume control up and down rapidly. This effect takes in two parameters called “depth” and “swells”. Depth ranges from 0 to 1 and determines the magnitude of the cosine wave. “swells” ranges from 1 to 5 Hz and determines how quickly the volume changes. To implement this, we determine a cosine value with the equation $\cos(2*\pi*swells*t/Fs)$. As in the ring modulation, t is a counter incremented with each sample and it is reset when $swells*t/Fs$ equals one. This value is then multiplied by the depth parameter and then added to one to get the resulting magnitude for the output.

Vibrato:

Vibrato varies the pitch of the output with a cosine wave. It operates on a similar principle as the flange effect in that it has a variable delay. The changes in delay are typically much larger than those for flange, so the size of the buffer required is also much larger. The effect takes in two parameters, one for the frequency of variation and one for the width of variation. As the buffer shortens, the program is effectively outputting samples faster than they are coming in by skipping

samples and interpolating. The result is that the sound speeds up and becomes higher pitched. As the buffer lengthens, the opposite happens. The program effectively outputs the samples slower than they are coming in by repeating samples and interpolating. Slowing down the sound makes it lower pitched. This is different from pitch shifting algorithms because the overall pitch of the sound does not change. This is why a simple buffer is all that is necessary.

Reverberation:

Reverberation is an effect which tries to simulate the effects of a real acoustic space on a sound. Depending on the settings, this could make the source sound as if it were located inside of a room, church, or concert hall. The way it does this is by layering many different echoes on top of each other. The primary way of characterizing reverberation is with its impulse response. The ideal impulse response would be very dense and be characteristic of Gaussian (white) noise. The simplest method, then, of creating reverberation would be to generate the desired impulse response and then convolve it with the input signal. This gives the best results, but is highly impractical because of the enormous amount of computation involved. All other methods are attempts to approximate the same result while greatly reducing the required computations.

The method that we implemented uses a bank of comb filters fed into a parallel pair of all-pass filters. A comb filter is simply a fixed delay buffer that feeds back into itself. The feedback loop has a gain reduction coefficient so that the sound will decay with each iteration through the buffer. The feedback loops also have a very gentle low-pass filter (-2 dB/octave) that reduces the high frequencies to give a more natural and euphonic sound. The term “comb filter” comes from the fact that the frequency response has many sharp dips in it that resemble a comb. Any frequency with a period n + 0.5 times the delay length (for any integer n) will be cancelled out. The all-pass filter is similar to the comb filter but does not have the low-pass filter and also has a feed-forward loop which allows the filter to have a smoother frequency response.

The lengths of the buffers were chosen using prime numbers (ranging from 6.5 to 96 ms) so that they would not have any common factors. This helped give a smooth frequency response. The all-pass filters were much shorter than any of the comb filters and were used primarily to add density to the impulse response.

Software Challenges:

The algorithms for each of the effects were developed in Matlab and then translated into C once they had been tested. Working in C on the DSK we ran into many difficulties in making the software operational. The first problem was that our program used many different floating point calculations. Because the C5510 does not do floating point math natively, these operation were significantly slowing down our program and preventing it from running in real time. The solution was to change all of our calculations to fixed point and utilize Q number formats to account for fractional numbers. We also used a lookup table in place of the cosine function and a polynomial approximation where the power function was used.

The next issue to tackle was that the program would crash if the buffers for the reverb effect were initialized. This was solved by moving all of the reverb buffers onto their own memory page. There

was also a large buffer shared by several effects (echo, phaser, vibrato, and flange) which required its own memory page as well.

Another issue that we ran into was that many of our internal calculations were overflowing at unlikely places. The problem turned out to be that if two 16 bit integers were being multiplied or added together, one of them had to be typecast as long (32 bits) in order to take advantage of 32 bit computation.

A final problem that required resolution was with effects such as vibrato and flange which required low frequency oscillation. The lookup table we were using (1024 point) was simply outputting the closest value it had to the real value required. This was then used to find the nearest array index of the next sample to be output. The truncation used in both steps was producing a significant amount of distortion. The first attempted solution was to implement interpolation so as to accommodate values that fell between table entries or array indices. This unfortunately only made a small improvement. The solution we found was to go back to using a floating point calculation for the cosine function. This gave significant improvements and still allowed the program to run in real time, as there was only one floating point calculation.

Accomplishments:

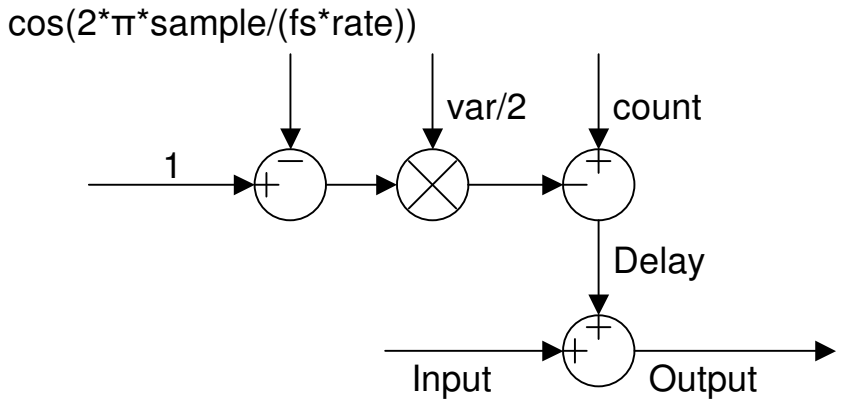
In completion of our project design, we were successfully able to implement all eight of the audio effects we planned on implementing in our pre-design and brainstorming phase. In previous semesters the knob box had been used, but not in the same way that we planned for our project. Our team decided to step into uncharted territory and try to implement the knob box into our project. We successfully incorporated the knob box and keypad into our design and opened up a new avenue for the usage of the knob box. One of the key successes of our project is that our processor operates in real-time while maintaining a good sound quality throughout the audio processing.

Extension into The Future:

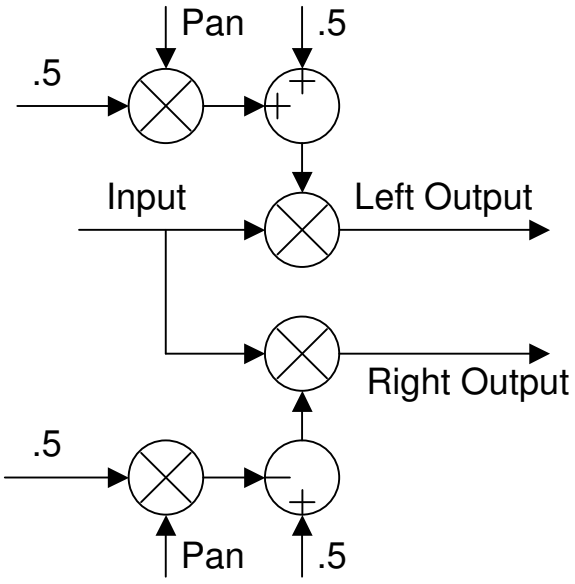
The design of our digital audio effects processor was a success, but at the same time, there is room to build upon the accomplishments we achieved this semester. For one, a future group has the option to implement eight more audio effects with the remaining eight keys on the keypad of the knob box. A drawback to our project was the limit of parameters due to the knob box only having four knobs. What makes some of the audio processors on the market so valuable is the number of parameters they are able to vary for different aspects of the audio effects. What a future group could possibly do is to design a knob box with more knobs or use two knob boxes in their design. Another extension that would make our project design more aesthetically pleasing would be a graphical display for the different effects and their corresponding parameters. Finally, to add to the aesthetic appeal of our design, a future group could create a level meter which would show the input and output levels of the audio signal. Such metering is key for connecting a processor to other devices.

Appendix A – Block Diagrams

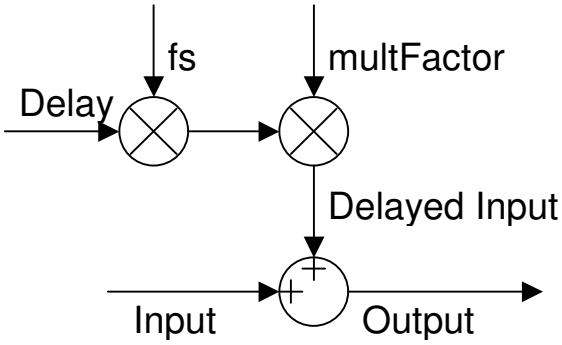
Flange/Chorus:



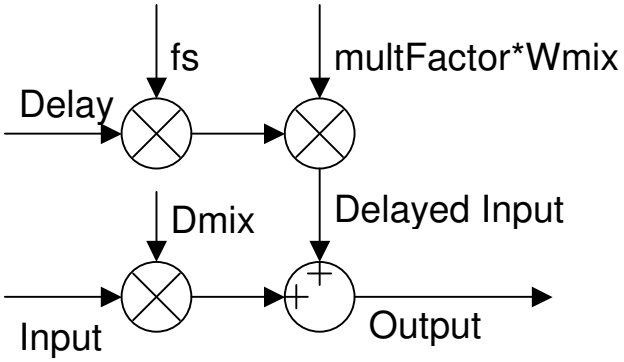
Panning/Ping Pong:



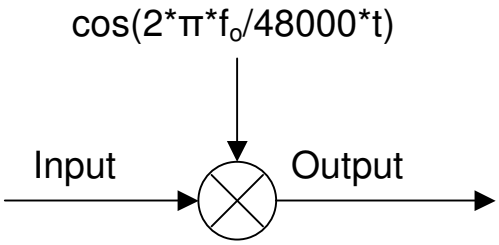
Phaser:



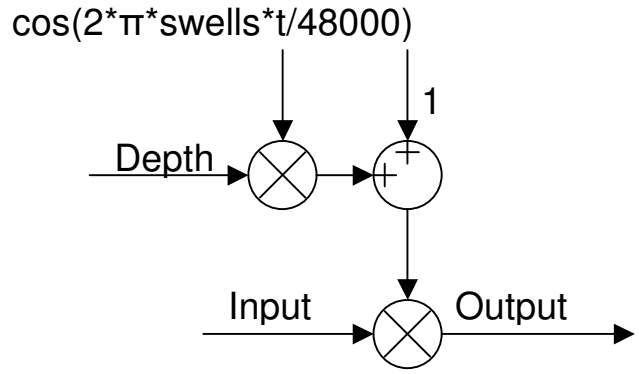
Echo:



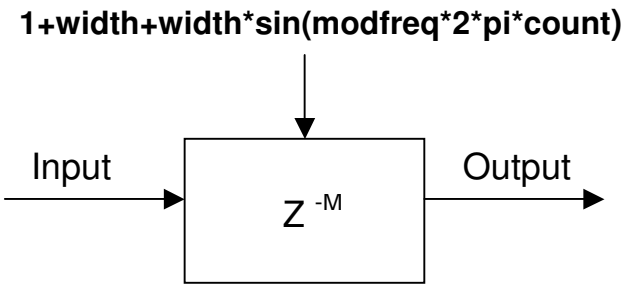
Ring Modulation:



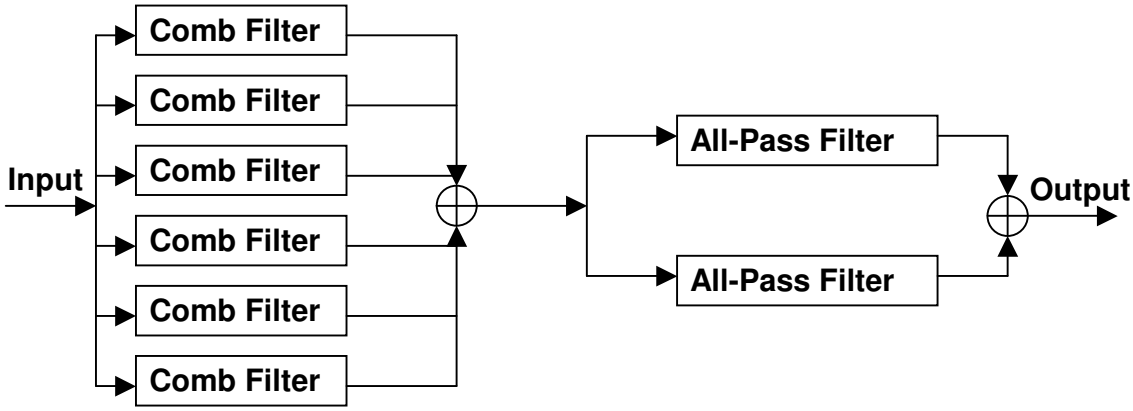
Tremolo:

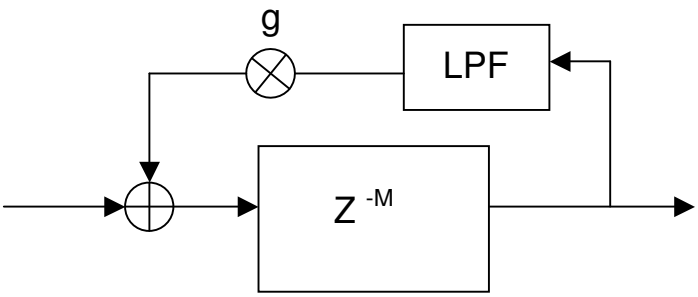


Vibrato:

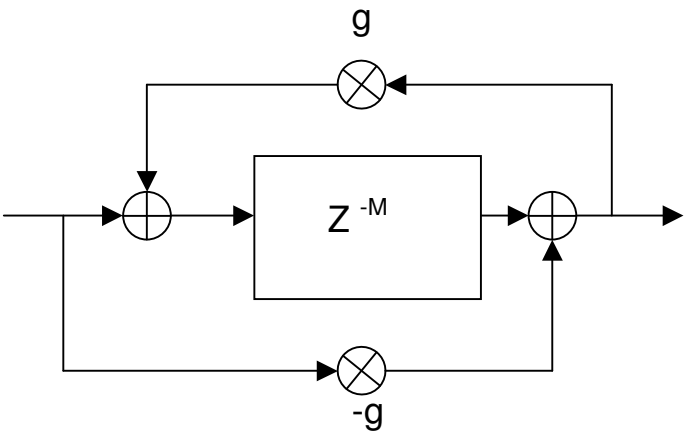


Reverberation:





Comb Filter



All-Pass Filter

Appendix B – Matlab Source Code

---

tremolo.m

```
function [y] = tremolo(x, fs, swells)
% y = tremolo(x, fs, swells)
% swells = swells per second
T=1/fs;
n=1:length(x);
g=1+0.8*cos(2*pi*swells.*n*T)';
y=x.*g;
```

---

phaser.m

```
%Phasing is achieved by adding a partially out of phase signal to the original
signal.
%To do this, a copy of the original signal is inverted and time-delayed,
%before being added to the original signal.

clear all

[y,fs,bits] = wavread('riff.wav');

delay = 10; % delay in ms
delay=floor((0.001*delay)*fs); % delay in samples
output = zeros(1,length(y)+1);

for(i=1:delay)
    output(i) = y(i);
end
for(i=(delay+1):length(y))
    output(i) = y(i) + .75 .* (y(i - delay));
end


wavwrite(output,fs,bits,'phasesax');

% Verification Plots
subplot(2,1,1);
plot(y(:,1));
axis([0,200000,-1,1]);
title('Input Signal');
subplot(2,1,2);
plot(output);
axis([0,200000,-1,1]);
title('Output Signal');
```

---

pan_algo.m

```
clear all;
[y,Fs,bits] = wavread('sax');
vol = 1;
% Float between -1 and 1
pan = -1;
% Good for values less than .000075
inc = .00002;
```

```
Output = zeros(2,length(y));
%Routput = zeros(1,length(y));
LVol = zeros(1,length(y));
RVol = zeros(1,length(y));
add = 1;

for(i=1:length(y))
    LVol(i) = vol + vol*pan;
    RVol(i) = vol - vol*pan;
    Output(1, i) = y(i)*LVol(i);
    Output(2, i) = y(i)*RVol(i);
    if(pan >= 1)
        add = 0;
    elseif(pan <= -1)
        add = 1;
    end
    if(add == 1)
        pan = pan + inc;
    else
        pan = pan - inc;
    end
end
wavwrite(transpose(Output),Fs,bits,'pansax');
```

---

echo1.m

```
% echo1
%
% Ross Penniman
% EECS 452 DAFX Project

clear all;
tic;
useWav=1;

if useWav;
    % Input Filename - MONO ONLY!!!
    [inSig, FS, NBITS]=wavread('riff.wav');
    inSig=transpose(inSig);
else
    FS=1000;
    % Use impulse as test signal
    inSig=zeros(1, 5000);
    inSig(1)=1;
end

dryMix=0.8;
wetMix=0.2;

% Delay Time, seconds
Td=0.3;

% Decay factor
g=0.3;

delaySamps=round(Td*FS);

delayBuf=zeros(1, delaySamps);

bufPtr=1;

for idx=1:(length(inSig)+2*FS);
```

```
    % Get next sample
    if(idx < length(inSig))
        inSamp=inSig(idx);
    end

    % Output oldest sample from buffer
    outSamp=delayBuf(bufPtr);

    % Replace oldest buffer with newest
    delayBuf(bufPtr)=delayBuf(bufPtr)*g + inSamp;

    % Update buffer pointer
    if (bufPtr < delaySamps)
        bufPtr=bufPtr+1;
    else
        bufPtr=1;
    end

    % Add outSamp to output signal
    outSig(idx)=outSamp;

end

maxpt=max(abs(outSig));
if (maxpt>1)
    outSig=outSig./maxpt;
end

lenDiff=length(outSig)-length(inSig);
inSig=[inSig, zeros(1, lenDiff)];

outSig=wetMix*outSig + dryMix*inSig;

if useWav;
    wavwrite(outSig,FS,NBITS,'echo1_out.wav');
else
    plot(outSig);
end

disp('Computation Time:');
toc;
```

---

flangechorus_for_c.m

```
% function [y]=flangechorus(fs, v, y, r)
%this version is for c-conversion
%may not necessarily be efficient for Matlab use
%
%   For a basic flanging effect.
%      fs = Sample rate
%      v = Variation.
%      x = Input audio signal. This should be a column
%          vector.
%      r = Rate.
%
%   Example:%
%      >>y = flange(fs,0.002,x,0.5);
%
%   For a basic chorus effect, Example:
%      >>y = flange(fs,0.2,x,0.5);
%
%   See also WAVREAD and SOUND
% For vox.wav: Elapsed time is 12211.937000 seconds.
```

```
tic;

[y,fs,bits] = wavread('vox.wav');

v=0.002; % variation delay time (seconds)
r=0.5; % rate (Hz)

md = ceil(v*fs); %(max delay time in samples)
y_length = length(y);
v=round(v*fs);

% pad beginning and end of input with zeros
for i=1:md
    z(i)=0;
end
for i=(md+1):(md+y_length)
    z(i)=y(i-md);
end
for i=(md+y_length):(2*md+y_length)
    z(i)=0;
end

m=max(abs(z));
rr=2*pi/round(fs*r);

for n=1:y_length+md
    b(n)=round((v/2)*(1-cos(rr*n)));
    z(n)=z(n+md)+z(n+md-b(n));
end

m=m/max(abs(z));
for i=1:length(z)
    y(i)=m*z(i);
end

wavwrite(y,fs,bits,'flangesax');

toc;
```

---

ringmod.m

```
[y,Fs,bits] = wavread('riff.wav');

output = zeros(1,length(y));
fo = 800;

for(i=1:length(y))
    output(i) = y(i)*cos(2*pi*fo/Fs*i);
end

wavwrite(output,Fs,bits,'ringsax');

% Verification Plots
subplot(2,1,1);
plot(y(:,1));
axis([0,200000,-1,1]);
title('Input Signal');
subplot(2,1,2);
plot(output);
axis([0,200000,-1,1]);
title('Output Signal');
```

---

vibrato_for_c.m

```
% function y=vibrato_for_c(x,SAMPLERATE,Modfreq,Width)
% Vibrato
% output = vibrato(x, SAMPLERATE, Modfreq, Width)
%x = name of sound wav
%SAMPLERATE =
clear all;

Modfreq =  3;
Width = 5e-3; % max of 20e-3
%[x,SAMPLERATE,bits] = wavread('sax');
% sine wave test, 1 kHz, 48 kHz FS
SAMPLERATE=48000;
bits=16;
t=0:((10*SAMPLERATE)-1);
x=sin(2*pi*t*1000/48000);

dptr=1; % pointer to most recent sample

WIDTH=round(Width*SAMPLERATE); % modulation width in # samples
DELAY=WIDTH; % basic delay in # samples

LEN=length(x);        % # of samples in WAV-file

MAX_BUFSIZE=2000; % should be (SAMPLERATE*0.02)+1 (1921 for 48 kHz, go 2000 to
be safe)
L=1+WIDTH*2;    % max array index for given width
%L=2+WIDTH*3;   % Textbook suggestion

DelayBuf=zeros(MAX_BUFSIZE,1); % memory allocation for delay
y=zeros(size(x)); % output array

for i=1:MAX_BUFSIZE;
    DelayBuf(i)=0;
end

for n=1:LEN % while effect==VIBRATO
   % load in most recent sample
   DelayBuf(dptr)=x(n);

   % index oscillator
   MOD=sin(2*pi*Modfreq*n/SAMPLERATE);
   % generate index values for linear interpolation
   ZEIGER=DELAY+WIDTH*MOD;
   i=floor(ZEIGER);
   frac=ZEIGER-i;

   % make i relative to dptr
   i=i+dptr;
   while (i > L)
       i=i-L;
   end

   if(i ~= L) i1=i+1;
   else i1=1;
   end

   % output proper buffer value with linear interpolation
   output=frac*DelayBuf(i1) + (1-frac)*DelayBuf(i);

   y(n)=output;

   % update dptr
   if(dptr>1) dptr=dptr-1;
```

```
   else dptr=L;
   end

end

wavwrite(y,SAMPLERATE,bits,'vibsine1.wav');
```

---

rossverb5.m

```
% rossverb5
% Comb and Allpass filter combination
%
% Ross Penniman
% EECS 452 DAFX Project
%

clear all;
tic;
useWav=0;


% Signal mixing
dryMix=0.7;
wetMix=0.3;

% Decay time, seconds
Td=2;

%Filter is designed for fs=48 kHz

if useWav;
    % Input Filename - MONO ONLY!!!
    [inSig, FS, NBITS]=wavread('narrate48.wav');
else
    FS=48000;
    t=linspace(0, Td, Td*FS+1);
    % Use impulse as test signal
    inSig=1;
end

% Add zeros onto input signal to make it long enough (Td sec worth)
inSig=[transpose(inSig), zeros(1, Td*FS)];

outSig=zeros(1, length(inSig));

% Delay time in milliseconds - prime numbers
% Comb Filters
CD1=1500450271/1e8;
CD2=2860486313/1e8;
CD3=4093082899/1e8;
CD4=5754853343/1e8;
CD5=5915587277/1e8;
CD6=9576890767/1e8;


% CD1=19;
% CD2=23;
% CD3=29;
% CD4=31;
% CD5=37;
% CD6=41;
% Allpass Filters
AD1=3267000013/5e8; % 6.534 ms
```

```
AD2=3628273133/5e8; % 7.257 ms


% AD1=5;
% AD2=7;

% Gain coefficients
% (set so that filters will decay 60 dB in Td seconds)
% Comb Filters
CG1=10^(-3*0.001*CD1/Td);
CG2=10^(-3*0.001*CD2/Td);
CG3=10^(-3*0.001*CD3/Td);
CG4=10^(-3*0.001*CD4/Td);
CG5=10^(-3*0.001*CD5/Td);
CG6=10^(-3*0.001*CD6/Td);
fprintf('CG1: %12.7d CG2: %12.7d CG3: %12.7d \n CG4: %12.7d CG5: %12.7d CG6:
%12.7d \n', CG1, CG2, CG3, CG4, CG5, CG6);
% Allpass Filters
% (do not change)
AG1=0.7;
AG2=0.607;


% Convert Delay time to samples
% Comb Filters
CD1=round(CD1*0.001*FS);
CD2=round(CD2*0.001*FS);
CD3=round(CD3*0.001*FS);
CD4=round(CD4*0.001*FS);
CD5=round(CD5*0.001*FS);
CD6=round(CD6*0.001*FS);
% Allpass Filters
AD1=round(AD1*0.001*FS);
AD2=round(AD2*0.001*FS);

fprintf('CD1: %12d CD2: %12d CD3: %12d \n CD4: %12d CD5: %12d CD6: %12d \n',
CD1, CD2, CD3, CD4, CD5, CD6);
fprintf('AD1: %12d AD2: %12d \n', AD1, AD2);

% Initialize the delay buffer with zeros
% Comb Filters
CDB1=zeros(1, CD1);
CDB2=zeros(1, CD2);
CDB3=zeros(1, CD3);
CDB4=zeros(1, CD4);
CDB5=zeros(1, CD5);
CDB6=zeros(1, CD6);
% Allpass Filters
ADB1=zeros(1, AD1);
ADB2=zeros(1, AD2);

% Pointers to oldest buffer sample
% Comb Filters
CP1=1;
CP2=1;
CP3=1;
CP4=1;
CP5=1;
CP6=1;
% Allpass Filters
AP1=1;
AP2=1;

% Lowpass filter setup - One pole lowpass at 1 kHz
% (in feedback loop of comb filters
A1 = 0.1333; % SOS(5)
```

```
B0 = 0.5666; % G(1)
B1 = 0.5666; % G(1)
% Previous input to filter, x(n-1)
C1IPre=0;
C2IPre=0;
C3IPre=0;
C4IPre=0;
C5IPre=0;
C6IPre=0;
% Previous output of filter, y(n-1)
C1OPre=0;
C2OPre=0;
C3OPre=0;
C4OPre=0;
C5OPre=0;
C6OPre=0;

% Loop samplewise over signal (simulates real-time operation)
for idx=1:length(inSig);

    inSamp=inSig(idx);
    % Compute intermediate sample (from comb filter bank)
    midSamp=CDB1(CP1)+CDB2(CP2)+CDB3(CP3)+CDB4(CP4)+CDB5(CP5)+CDB6(CP6);
    % Compute output sample (from allpass filters)
    AOut1=((-AG1)*midSamp)+ADB1(AP1);
    AOut2=((-AG2)*midSamp)+ADB2(AP2);
    outSamp=AOut1+AOut2;

    % Save old CDB values for later use
    Temp1=CDB1(CP1);
    Temp2=CDB2(CP2);
    Temp3=CDB3(CP3);
    Temp4=CDB4(CP4);
    Temp5=CDB5(CP5);
    Temp6=CDB6(CP6);

    % Replace oldest comb buffers with newest
    % (lowpass filter in feedback loop)
    CDB1(CP1)=CG1*(B0*CDB1(CP1) + B1*C1IPre - A1*C1OPre)+inSamp;
    CDB2(CP2)=CG2*(B0*CDB2(CP2) + B1*C2IPre - A1*C2OPre)+inSamp;
    CDB3(CP3)=CG3*(B0*CDB3(CP3) + B1*C3IPre - A1*C3OPre)+inSamp;
    CDB4(CP4)=CG4*(B0*CDB4(CP4) + B1*C4IPre - A1*C4OPre)+inSamp;
    CDB5(CP5)=CG5*(B0*CDB5(CP5) + B1*C5IPre - A1*C5OPre)+inSamp;
    CDB6(CP6)=CG6*(B0*CDB6(CP6) + B1*C6IPre - A1*C6OPre)+inSamp;

    % Y/X = (B0 + B1*z-1)/(1 + A1*z-1)
    % Y(1 + A1*z-1)=X(B0 + B1*z-1)
    % Y = X(B0 + B1*z-1) - Y*A1*z-1
    % Without lowpass filter
%     CDB1(CP1)=CG1*CDB1(CP1)+inSamp;
%     CDB2(CP2)=CG2*CDB2(CP2)+inSamp;
%     CDB3(CP3)=CG3*CDB3(CP3)+inSamp;
%     CDB4(CP4)=CG4*CDB4(CP4)+inSamp;
%     CDB5(CP5)=CG5*CDB5(CP5)+inSamp;
%     CDB6(CP6)=CG6*CDB6(CP6)+inSamp;

    % Update IPre and OPre values
    C1IPre=Temp1;
    C2IPre=Temp2;
    C3IPre=Temp3;
    C4IPre=Temp4;
    C5IPre=Temp5;
    C6IPre=Temp6;
    C1OPre=CDB1(CP1);
    C2OPre=CDB2(CP2);
    C3OPre=CDB3(CP3);
```

```
    C4OPre=CDB4(CP4);
    C5OPre=CDB5(CP5);
    C6OPre=CDB6(CP6);

    % Replace oldest allpass buffer value with newest
    ADB1(AP1)=(AG1*AOut1)+midSamp;
    ADB2(AP2)=(AG2*AOut2)+midSamp;

    % Update oldest buffer sample pointer
    % Comb Buffer 1
    if (CP1 >= CD1)
        CP1=1;
    else
        CP1=CP1+1;
    end
    % Comb Buffer 2
    if (CP2 >= CD2)
        CP2=1;
    else
        CP2=CP2+1;
    end
    % Comb Buffer 3
    if (CP3 >= CD3)
        CP3=1;
    else
        CP3=CP3+1;
    end
    % Comb Buffer 4
    if (CP4 >= CD4)
        CP4=1;
    else
        CP4=CP4+1;
    end
    % Comb Buffer 5
    if (CP5 >= CD5)
        CP5=1;
    else
        CP5=CP5+1;
    end
    % Comb Buffer 6
    if (CP6 >= CD6)
        CP6=1;
    else
        CP6=CP6+1;
    end
    % Allpass Buffer 1
    if (AP1 >= AD1)
        AP1=1;
    else
        AP1=AP1+1;
    end
    % Allpass Buffer 2
    if (AP2 >= AD2)
        AP2=1;
    else
        AP2=AP2+1;
    end
    % Add outSamp to output signal
    outSig(idx)=outSamp;
end

% Condition output signal
maxpt=max(abs(outSig));
fprintf('Max Output Value: %f \n',  maxpt);

if (maxpt > 1)
```

```
    outSig=outSig./maxpt;
end

if useWav;
    sizeDiff=length(outSig)-length(inSig);
    inSig=[inSig; zeros(sizeDiff, 1)];
    outSig=wetMix*outSig + dryMix*inSig;
    wavwrite(outSig,FS,NBITS,'rossverb5_out.wav');
else
    plot(t, outSig);
    title('Impulse Response, Rossverb5');
    xlabel('Time (seconds)');
    ylabel('Amplitude');
end

disp('Computation Time');
toc;
```

Appendix C – C Source Code

---

master_c.c

```
/*
      AUTHORS: Ross Penniman
               Jason Martin
               Amanda Wallace
      Project: Da FX Project

      EECS 452 Major Design Project
*/

#include "support/UART2support.h"
#include "support/asciiToInt.h"

//#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <extaddr.h>
// need to add setup_codec.c to project
#include "support\McBSP_452.h"
#include "sin_table.h"


#define max_setting 32767   /* maximum input */
#define min_setting 0    /* minimum input*/

#define FOREVER 1
#define size_of_inbuf 5
#define size_of_outbuf 5      //not using this
int inbuf_address[size_of_inbuf];
int outbuf_address[size_of_outbuf]; //not using this
int RateDivisor = 6;    //to give 38400 baud

// Parameter Initial Conditions
long current_param1=16384;
long current_param2=16384;
long current_param3=16384;
long current_param4=16384;
int effect_number = 1;

#define UART 0x500200
#define RBR (UART+0x00)
#define THR (UART+0x00)
#define DLL (UART+0x00)
#define DLM (UART+0x02)
#define IER (UART+0x02)
#define ISR (UART+0x04)
#define FCR (UART+0x04)
#define LCR (UART+0x06)
#define MCR (UART+0x08)
#define LSR (UART+0x0A)
#define MSR (UART+0x0C)
#define SPR (UART+0x0E)

// Define effect numbers
#define ECHO 1
#define TREMOLO 2
#define VIBRATO 3
```

```
#define REVERB 4
#define FLANGE 5
#define PHASER 6
#define PAN 7
#define RINGMOD 8

// Increment size for knobs
#define INC_SIZE 900;

// Effect function prototypes
void AIC23_IO(unsigned port, int LeftValue, int RightValue);
void flangechorus(int modfreq, int width, int dmix, int wmix);
void panning(int vary);
void phaser(int del, int multFactor);
void ringmod(int fo);
void tremolo(int swells, int depth);
void vibrato(int modfreq, int width);
void echo(int del, int multFactor, int dmix, int wmix);
void reverb(int rt60, int dmix, int wmix);

int satur8(long);

// IO variables
int LeftInput, RightInput, Left, Right; // sample values go into these

int effect=1;
long FS=48000;

const int SIN_TABLESIZE=1024;

const double pi = 3.1415927;

// Buffer for vibrato, flange, phasing, and echo

#pragma DATA_SECTION(buf, ".thebuf");
int buf[24001];
int i;

// Define reverb buffer sizes
int CD1=720;
int CD2=1373;
int CD3=1965;
int CD4=2762;
int CD5=2839;
int CD6=4597;
int AD1=314;
int AD2=348;

// Create memory space for reverb buffers
#pragma DATA_SECTION(CDB1, ".cdbuffer");
int CDB1[720];
#pragma DATA_SECTION(CDB2, ".cdbuffer");
int CDB2[1373];
#pragma DATA_SECTION(CDB3, ".cdbuffer");
int CDB3[1965];
#pragma DATA_SECTION(CDB4, ".cdbuffer");
int CDB4[2762];
#pragma DATA_SECTION(CDB5, ".cdbuffer");
int CDB5[2839];
#pragma DATA_SECTION(CDB6, ".cdbuffer");
int CDB6[4597];

#pragma DATA_SECTION(ADB1, ".cdbuffer");
int ADB1[314];
#pragma DATA_SECTION(ADB2, ".cdbuffer");
int ADB2[348];
```

```
void updateParam1(char arg5)
{
	/*	Do some work to calculate new parameter setting */

	/*If the fifth argument equals 0x55 or 0x75("U or u -> up"), increase
param controled by knob0*/
	if((arg5 == 0x55) || (arg5 == 0x75))
	{
		current_param1 = current_param1 + INC_SIZE;	//This will most
likely be different.
	}
	/*If the fifth argument equals 0x44 or 0x64("D or d -> down"), decrease
param controled by knob0*/
	else if((arg5 == 0x44) || (arg5 == 0x64))
	{
		current_param1 = current_param1 - INC_SIZE;	//This will most
likely be different.
	}

	/* Make sure new parameter is not outside of the limit*/
	if(current_param1 > max_setting)
	{
		current_param1 = max_setting;
	}
	else if(current_param1 < min_setting)
	{
		current_param1 = min_setting;
	}

}

void updateParam2(char arg5)
{

	/*	Do some work to calculate new parameter setting */

	/*If the fifth argument equals 0x55 or 0x75("U or u -> up"), increase
param controled by knob1*/
	if((arg5 == 0x55) || (arg5 == 0x75))
	{
		current_param2 = current_param2 + INC_SIZE;	//This will most
likely be different.
	}
	/*If the fifth argument equals 0x44 or 0x64("D or d -> down"), decrease
param controled by knob1*/
	else if((arg5 == 0x44) || (arg5 == 0x64))
	{
		current_param2 = current_param2 - INC_SIZE;	//This will most
likely be different.
	}

	/* Make sure new parameter is not outside of the limit*/
	if(current_param2 > max_setting)
	{
		current_param2 = max_setting;
	}
	else if(current_param2 < min_setting)
	{
		current_param2 = min_setting;
	}

}
```

```
void updateParam3(char arg5)
{
	/*	Do some work to calculate new parameter setting */

	/*If the fifth argument equals 0x55 or 0x75("U or u -> up"), increase
param controled by knob2*/
	if((arg5 == 0x55) || (arg5 == 0x75))
	{
		current_param3 = current_param3 + INC_SIZE;	//This will most
likely be different.
	}
	/*If the fifth argument equals 0x44 or 0x64("D or d -> down"), decrease
param controled by knob2*/
	else if((arg5 == 0x44) || (arg5 == 0x64))
	{
		current_param3 = current_param3 - INC_SIZE;	//This will most
likely be different.
	}

	/* Make sure new parameter is not outside of the limit*/
	if(current_param3 > max_setting)
	{
		current_param3 = max_setting;
	}
	else if(current_param3 < min_setting)
	{
		current_param3 = min_setting;
	}

}

void updateParam4(char arg5)
{
	/*	Do some work to calculate new parameter setting */

	/*If the fifth argument equals 0x55 or 0x75("U or u -> up"), increase
param controled by knob3*/
	if((arg5 == 0x55) || (arg5 == 0x75))
	{
		current_param4 = current_param4 + INC_SIZE;	//This will most
likely be different.
	}
	/*If the fifth argument equals 0x44 or 0x64("D or d -> down"), decrease
param controled by knob3*/
	else if((arg5 == 0x44) || (arg5 == 0x64))
	{
		current_param4 = current_param4 - INC_SIZE;	//This will most
likely be different.
	}

	/* Make sure new parameter is not outside of the limit*/
	if(current_param4 > max_setting)
	{
		current_param4 = max_setting;
	}
	else if(current_param4 < min_setting)
	{
		current_param4 = min_setting;
	}

}


int main()
{
	/*Input Protocol*/
```

```
      char arg1='p';          //(P or p, keypad) or a (K or k, knob)
      int arg2=1;       //Some number which selects an effect or knob depending
what 1st character was
      char arg3='u';          //(U or u) or (D or d), increment or decrement
parameter
      char arg_junk;    //just a random junk argument . probably character "x"

      //counter
      int i;


      Left = 0;
      Right = 0;

      setup_codec(); // need to add setup_codec.c to project

      // Initialize big buffer
      for(i=0; i<24001; i++) buf[i]=0;

      // Initialize Reverb Buffers
      for(i=0; i<CD1; i++) CDB1[i]=0;
      for(i=0; i<CD2; i++) CDB2[i]=0;
      for(i=0; i<CD3; i++) CDB3[i]=0;
      for(i=0; i<CD4; i++) CDB4[i]=0;
      for(i=0; i<CD5; i++) CDB5[i]=0;
      for(i=0; i<CD6; i++) CDB6[i]=0;
      for(i=0; i<AD1; i++) ADB1[i]=0;
      for(i=0; i<AD2; i++) ADB2[i]=0;

      //Source of possible errors? (possibly not optimized for three arguments)
      UART2setup(inbuf_address, size_of_inbuf, outbuf_address, size_of_outbuf,
RateDivisor);
      _enable_interrupts();

      /* Main simulation loop */


      while(FOREVER)
      {

            /*
            if (testForArg(1) == 0)
            {
                  arg1 = U2RxGet();//should be a (p or P) or a (k or K)
                  while(arg1==0x0a) arg1 = U2RxGet();
                  printf("arg1: %8d \n", asciiToInt(arg1));
            }
            */

            //if (testForArg(1) == 0) arg1 = U2RxGet();//should be a (p or P) or
a (k or K)
            arg1 = U2RxGet();


            if((arg1 == 0x50) || (arg1 == 0x70)) // set effect number
            //if((asciiToInt(arg1) > 0) && (asciiToInt(arg1) < 9)) // set effect
number
            {

                  arg2 = U2RxGet();//should be the effect number
                  //arg2=arg1;
                  arg_junk = U2RxGet();//I dont care what this value is because
I will not use it

                  //need to change the second and fourth argument
```

```
                  //into integers because I am using switch statements
                  arg2 = asciiToInt(arg2);

                  if((arg2 < 1) || (arg2 > 8))
                  {
                        //printf("ERROR: INVALID KEY NUMBER: %8d \n", arg2);
                  }
                  else
                  {
                        effect=arg2;
                        effect_number=arg2;
                  }
            }
            else if((arg1 == 0x4B) || (arg1 == 0x6B)) // set parameters
            {
                  //Continue to update the current effect parameters
                  arg2 = U2RxGet();//should be the knob number
                  arg3 = U2RxGet();//should be a (u or U) or a (d or D)

                  //need to change the second argument
                  //into integer because I am using switch statements
                  arg2 = asciiToInt(arg2);

                  /*Determing which knob was turned*/
                  switch(arg2)
                  {
                        /*Select one of the four knobs to get new parameter
value*/
                        case 1: /*knob 0*/
                              updateParam1(arg3);
                              break;
                        case 2: /*knob 1*/
                              updateParam2(arg3);
                              break;
                        case 3: /*knob 2*/
                              updateParam3(arg3);
                              break;
                        case 4: /*knob 3*/
                              updateParam4(arg3);
                              break;
                  }
                  // reset effect number to previous call
                  effect=effect_number;
            }
            else
            {
                  //printf("ERROR: INVALID COMMAND CHARACTER: %8c \n", arg1);
                  // Clean out instruction buffer
                  while(testForArg(1) == 0) arg_junk = U2RxGet();
            }

            switch(effect) //call appropriate effect
            {
                  case 1: // ECHO
                        //printf("Great: I got to the echo effect call \n");
                        //printf("P1: %8ld P2: %8ld P3: %8ld P4: %8ld \n",
current_param1, current_param2, current_param3, current_param4);
                        echo(current_param1, current_param2, current_param3,
current_param4);
                        break;
                  case 2: // TREMOLO
                        //printf("Great: I got to the tremolo effect call \n");
                        //printf("P1: %8ld P2: %8ld \n", current_param1,
current_param2);
                        tremolo(current_param1, current_param2);
                        break;
```

```
                  case 3: // VIBRATO
                        //printf("Great: I got to the vibrato effect call \n");
                        //printf("P1: %8ld P2: %8ld \n", current_param1,
current_param2);
                        vibrato(current_param1, current_param2);
                        break;
                  case 4: // REVERB
                        //printf("Great: I got to the Reverberation effect call
\n");
                        //printf("P1: %8ld P3: %8ld P4: %8ld \n",
current_param1, current_param3, current_param4);
                        reverb(current_param1, current_param3, current_param4);
                        break;
                  case 5: // FLANGE
                        //printf("Great: I got to the Chorus and Flanging effect
call \n");
                        //printf("P1: %8ld P2: %8ld P3: %8ld P4: %8ld \n",
current_param1, current_param2, current_param3, current_param4);
                        flangechorus(current_param1, current_param2,
current_param3, current_param4);
                        break;
                  case 6: // PHASE
                        //printf("Great: I got to the phase effect call 1 \n");
                        //printf("P1: %8ld P2: %8ld \n", current_param1,
current_param2);
                        phaser(current_param1, current_param2);
                        break;
                  case 7: // PANNING
                        //printf("Great: I got to the ping-pong effect call 1
\n");
                        //printf("P1: %8ld \n", current_param1);
                        panning(current_param1);
                        break;
                  case 8: // RING MOD
                        //printf("Great: I got to the ring modulation effect
call 1 \n");
                        //printf("P1: %8ld \n", current_param1);
                        ringmod(current_param1);
                        break;
                  default:
                        //printf("ERROR: ILLEGAL FUNCTION CALL: %8d \n",
effect);
                        break;
            }

      }//end of the main while loop

  //return(0);

}//end of main


void AIC23_IO(unsigned port, int LeftValue, int RightValue)
{
      McBSP_reg(port, McBSP_DXR2) = LeftValue;
      McBSP_reg(port, McBSP_DXR1) = RightValue;
      while((McBSP_reg(port, McBSP_SPCR1)&0x0002) == 0); // wait for sample
      LeftInput = McBSP_reg(2, McBSP_DRR2);
      RightInput = McBSP_reg(2, McBSP_DRR1);
}

int satur8(long x)
{
      int y;
      if (x > 32767) y = 32767;
      else if (x < -32767) y = -32767;
```

```
	else y = (int)x;

	return y;
}


void echo(int del, int multFactor, int dmix, int wmix)
{
	/* Input parameter delay ranges from 0 (No Effect) to 16384 (.5)s in Q15
	 * Input parameter multFactor ranges from 16384 (.5) to 32768 (1) in Q15
*/

	//int i;
	int readInd, writeInd;
	long output;
	int delay; //= (del*FS)>>15; // Q0
	long addVal;

	// Scaling
	// Input ranges from 0 to 32767 in Q15

	// Relates to 0 to 16384 in Q15 for del

	del=del>>1;

	// multFactor needs no scaling;

	delay = ((long)del*FS)>>15; // Q0

	for(i = 0; i <= delay; i++) buf[i] = 0;

	readInd = 0; // Q0
	writeInd = delay; // Q0

	while (effect == ECHO) {
		AIC23_IO(2, Left, Right);

		addVal = ((long)multFactor*(long)buf[readInd])>>15;

		output = (long)dmix*LeftInput + (long)wmix*buf[readInd]; //Q30

		Left = satur8(output>>15);
		Right = Left;

		buf[writeInd] = LeftInput + addVal; // Q15
		writeInd++;
		readInd++;
		if(writeInd > delay) writeInd = 0;
		if(readInd > delay) readInd = 0;

		effect=testForArg(effect);
	}
}

void vibrato(int modfreq, int width)
{

	// modfreq: Q11 0 to 16
	// width: Q15 0.02 or less (655 or less in Q15)

	int dptr=0; // pointer to most recent buffer sample
	int DELAY, WIDTH, L, MOD;
	//long lookup, lookup_0, lookup_1;
	long ZEIGER, FLOOR, FLOOR_1, frac, output, counter=0;
	//int i;
	long scale;
```

```
	// declare delay buffer
	//int buf[2000];

	// Scaling
	// Input ranges from 0 to 32767 in Q15

	// Relates to 0 to 32768 in Q11 for modfreq -> no scaling necessary
	modfreq=modfreq>>1;
	// Relates to 0 to 655 in Q15 for width
	scale = (long)width<<6;
	width = (int)((scale / (long)50)>>6);

	// initialize delay buffer


	// CHANGE PARAMETERS HERE!

	// calculate DELAY and WIDTH in samples
	WIDTH=((long)width*FS)>>15;
	DELAY=WIDTH;

	// calculate usable delay buffer
	L=2*WIDTH+1; //Q0

	while (effect == VIBRATO) {
		AIC23_IO(2, Left, Right);

		buf[dptr]=LeftInput;


		/*
		//printf("%8d, ", dptr);
		// oscillator = sin(2*pi*f0*n/fs);
		// (counter<<15)/FS = time in Q15

	lookup=((SIN_TABLESIZE*(((long)modfreq*((counter<<15)/FS))>>11))>>2);
//Q13
		//printf("%8d, ", lookup);

		// Interpolation
		lookup_0=((lookup>>13)<<13);
		frac=lookup-lookup_0; // frac is Q13
		lookup_0=(lookup_0>>13); // Q0

		if(lookup_0 < 1023) lookup_1=lookup_0+1;
		else lookup_1=0;

		MOD=((8192-
frac)*(long)SIN_TABLE[lookup_0]+frac*(long)SIN_TABLE[lookup_1])>>13; //
Q13*Q15=Q26>>13 = Q15
		*/



	MOD=(int)(32767*sin(2*pi*((double)modfreq/2048)*(double)counter/FS));

		ZEIGER=((long)DELAY<<15) + ((long)WIDTH*MOD); // Q15
		FLOOR=((ZEIGER>>15)<<15);
		frac=ZEIGER-FLOOR; //Q15

		FLOOR=(FLOOR>>15); // Q0
		//printf("%8ld \n", FLOOR);

		// make FLOOR relative to dptr
		FLOOR=FLOOR+dptr;
```

```
            while(FLOOR > L) FLOOR=FLOOR-L;

            if(FLOOR != L) FLOOR_1=FLOOR+1;
            else FLOOR_1=0;

            output=frac*(long)buf[FLOOR_1] + (32768-frac)*(long)buf[FLOOR];
//Q30

            Left=satur8(output>>15);
            Right=Left;

            // update dptr
            if(dptr > 0) dptr--;
            else dptr=L;

            // (32768<<11)/modfreq = period, Q15
            // (counter<<15)/FS = time in Q15
            if(((counter<<15)/FS) >= ((32768<<11)/(long)modfreq)) counter = 0;
            else counter++;

            effect=testForArg(effect);
      }
}

void flangechorus(int modfreq, int width, int dmix, int wmix)
{
      /* Input parameter width ranges from 0 (no effect) to 262 (.008)s in Q15
       * Input parameter modfreq ranges from 9830 (.3) to 32767 (<1) Hz  in Q15
*/


      int dptr=0; // pointer to most recent buffer sample
      int DELAY, WIDTH, L, MOD;
      //long lookup, lookup_0, lookup_1;
      long ZEIGER, FLOOR, FLOOR_1, frac, output, counter=0;
      //int i;
      long scale;

      // declare delay buffer

      // Scaling
      // Input ranges from 0 to 32767 in Q15

      // Relates to 0 to 32768 in Q15 for modfreq -> no scaling necessary
      // Relates to 0 to 262 in Q15 for width
      //printf("Flange Start \n");
      scale = (long)width<<6;
      width = (int)((scale / (long)125)>>6);

      // Relates to 9830 to 32767 in Q15 for modfreq
      // Take over the range 0 to 22937 and add 9830
      // Multiply by .7 (11 in Q4) to get 0 to 22937
      scale = ((long)modfreq*(long)11)>>4; // Q15*Q4=Q19
      scale = scale + (long)9830;
      modfreq = (int) scale;


      // CHANGE PARAMETERS HERE!

      // calculate DELAY and WIDTH in samples
      WIDTH=((long)width*FS)>>15;
      DELAY=WIDTH;

      // calculate usable delay buffer
      L=2*WIDTH+1; //Q0
```

```
      while (effect == FLANGE) {
            AIC23_IO(2, Left, Right);

            buf[dptr]=LeftInput;

            /*
            //printf("%8d, ", dptr);
            // oscillator = sin(2*pi*f0*n/fs);
            // (counter<<15)/FS = time in Q15

      lookup=((SIN_TABLESIZE*(((long)modfreq*((counter<<15)/FS))>>15))>>2);
//Q13
            //printf("%8d, ", lookup);

            // Interpolation
            lookup_0=((lookup>>13)<<13);
            frac=lookup-lookup_0; // frac is Q13
            lookup_0=(lookup_0>>13); // Q0

            if(lookup_0 < 1023) lookup_1=lookup_0+1;
            else lookup_1=0;

            MOD=((8192-
frac)*(long)SIN_TABLE[lookup_0]+frac*(long)SIN_TABLE[lookup_1])>>13; //
Q13*Q15=Q26>>13 = Q15
            */


      MOD=(int)(32767*sin(2*pi*((double)modfreq/32768)*(double)counter/FS));


            ZEIGER=((long)DELAY<<15) + ((long)WIDTH*MOD); // Q15
            FLOOR=((ZEIGER>>15)<<15);
            frac=ZEIGER-FLOOR; //Q15

            FLOOR=(FLOOR>>15); // Q0
            //printf("%8ld \n", FLOOR);

            // make FLOOR relative to dptr
            FLOOR=FLOOR+dptr;
            while(FLOOR > L) FLOOR=FLOOR-L;

            if(FLOOR != L) FLOOR_1=FLOOR+1;
            else FLOOR_1=0;

            output=(frac*(long)buf[FLOOR_1] + (32768-
frac)*(long)buf[FLOOR])>>15; //Q15

            output = (long)dmix*LeftInput + (long)wmix*output; //Q30

            Left=satur8(output>>15);
            Right=Left;

            // update dptr
            if(dptr > 0) dptr--;
            else dptr=L;

            // (32768<<11)/modfreq = period, Q15
            // (counter<<15)/FS = time in Q15
            if(((counter<<15)/FS) >= ((32768<<15)/(long)modfreq)) counter = 0;
            else counter++;

            effect=testForArg(effect);
      }
}
```

```
void panning(int vary)
{
      /* Input parameter vary ranges from 0 to 32768 */

      long vol;
      int add;
      long Lvol, Rvol;
      long output;
      long pan = 32768; // Q15

      // Scaling
      // Input ranges from 0 to 32767 in Q15

      vol = 16384; // Q15

      /* process values on both left and right */
      while (effect == PAN) {
            AIC23_IO(2, Left, Right);

            Lvol = vol + ((vol*(pan>>12))>>15); // Q15
            Rvol = vol - ((vol*(pan>>12))>>15); // Q15
            output = (Lvol*LeftInput)>>15; // Q15

            Left = satur8(output);

            output = (Rvol*LeftInput)>>15; // Q15

            Right = satur8(output);

            if((pan>>12) >= 32768) add = 0;
            else if((pan>>12) <= -32768) add = 1;
            if(add == 1) pan = pan + vary; // Q15
            else pan = pan - vary; // Q15

            effect=testForArg(effect);
      }
}

//void phaser(int del, int multFactor);

// change delay to milliseconds
void phaser(int del, int multFactor)
{
      /* Input parameter delay ranges from 0 (No Effect) to 3277 (.1)s in Q15
       * Input parameter multFactor ranges from 16384 (.5) to 32768 (1) in Q15
*/

      /* Phaser seems to be working as of April 4, 2005 */

      //int i;
      int readInd, writeInd;
      int output;
      //int buf[4801]; // Set to be the maximum possible delay plus one once
determined
      int delay = (del*FS)>>15; // Q0
      long addVal;
      long scale;

      // Scaling
      // Input ranges from 0 to 32767 in Q15

      // Relates to 0 to 3277 in Q15 for del
      scale = (long)del<<4;
      del = (int)((scale / (long)10)>>4);
```

```
	// Relates to 16384 to 32768 in Q15 for multFactor
	scale = (((long)multFactor<<1)/(long)2)>>1;
	scale = scale + (long)16384;
	multFactor = (int) scale;

	delay = (del*FS)>>15; // Q0

	for(i = 0; i < delay; i++) buf[i] = 0;

	readInd = 0 - delay; // Q0
	writeInd = 0; // Q0

	while (effect == PHASER) {
		AIC23_IO(2, Left, Right);

		addVal = ((long)multFactor*(long)buf[readInd])>>15;
		if(readInd < 0) output = LeftInput;
		else output = LeftInput + addVal;

		Left = satur8(output);
		Right = Left;

		buf[writeInd] = LeftInput; // Q15
		writeInd++;
		readInd++;
		if(writeInd > delay)
		{
			writeInd = 0;
		}
		if(readInd > delay)
		{
			readInd = 0;
		}

		effect=testForArg(effect);
	}
}


void reverb(int rt60, int dmix, int wmix)
{
	// rt60:  Q11  0.5 sec to 10 sec
	// dmix:  Q15  0 to 1
	// wmix:  Q15  0 to 1

	// rt60 converted to samples
	long RT60;

	//int i;

	// Input scale value - NEEDS TWEAKING!!!
	int inScale=4096; //Q15, 0.5
	int FXInput=0;


	// Feedback gain values, Q15
	int CG1, CG2, CG3, CG4, CG5, CG6;
	int AG1, AG2;
	// Buffer pointers
	int CP1, CP2, CP3, CP4, CP5, CP6;
	int AP1, AP2;
	// Filter coefficients
	int A1, B0, B1;   // Q15

	// Holders for lowpass filters
```

```
      // same data type as buffers
      int C1PreI, C2PreI, C3PreI, C4PreI, C5PreI, C6PreI;
      int C1PreO, C2PreO, C3PreO, C4PreO, C5PreO, C6PreO;

      // Intermediate signal values
      long midSamp, AOut1, AOut2, output;

      // Temp holders
      // same data type as buffers
      int Temp1, Temp2, Temp3, Temp4, Temp5, Temp6;

      // Gain calculation temporary exponent
      long exptemp;


      //printf("Entered Reverb \n");

      // Limit range of rt60 to 0.25 sec to 11 sec
      rt60=(((long)rt60*2)/3)+512;


      // Convert rt60 to samples
      RT60=(FS*(long)rt60)>>11;
      //printf("RT60: %8ld \n", RT60);

      // Gain coefficients
      // (set so that filters will decay 60 dB in rt60 seconds)
      // Comb Filters

      // Polynomial coefficients, Q15
      // 5881,    27591,    61530,    85665,    75337,    32765
      // CD1 and RT60 both in units of samples
      //CG1=exp10(-3*CD1/RT60);
      exptemp=(-3*32768*(long)CD1)/RT60; // Q15, range -1 to 0
      CG1=((((((((((((((5881*exptemp)>>15)+27591)*exptemp)>>15)+61530)*exptemp)
>>15)
                  +85665)*exptemp)>>15)+75337)*exptemp)>>15)+32765);
      exptemp=(-3*32768*(long)CD2)/RT60; // Q15, range -1 to 0
      CG2=((((((((((((((5881*exptemp)>>15)+27591)*exptemp)>>15)+61530)*exptemp)
>>15)
                  +85665)*exptemp)>>15)+75337)*exptemp)>>15)+32765);
      exptemp=(-3*32768*(long)CD3)/RT60; // Q15, range -1 to 0
      CG3=((((((((((((((5881*exptemp)>>15)+27591)*exptemp)>>15)+61530)*exptemp)
>>15)
                  +85665)*exptemp)>>15)+75337)*exptemp)>>15)+32765);
      exptemp=(-3*32768*(long)CD4)/RT60; // Q15, range -1 to 0
      CG4=((((((((((((((5881*exptemp)>>15)+27591)*exptemp)>>15)+61530)*exptemp)
>>15)
                  +85665)*exptemp)>>15)+75337)*exptemp)>>15)+32765);
      exptemp=(-3*32768*(long)CD5)/RT60; // Q15, range -1 to 0
      CG5=((((((((((((((5881*exptemp)>>15)+27591)*exptemp)>>15)+61530)*exptemp)
>>15)
                  +85665)*exptemp)>>15)+75337)*exptemp)>>15)+32765);
      exptemp=(-3*32768*(long)CD6)/RT60; // Q15, range -1 to 0
      CG6=((((((((((((((5881*exptemp)>>15)+27591)*exptemp)>>15)+61530)*exptemp)
>>15)
                  +85665)*exptemp)>>15)+75337)*exptemp)>>15)+32765);

      //printf("Gain Coefs: \n %8d, %8d, %8d, %8d, %8d, %8d \n", CG1, CG2, CG3,
CG4, CG5, CG6);

      //exit(0);

      // Allpass Filters, Q15
      //AG1=0.7;
      AG1=22938;
```

```
	//AG2=0.67293;
	AG2=22051;

	// Pointers to oldest buffer sample
	// Comb Filters
	CP1=0;
	CP2=0;
	CP3=0;
	CP4=0;
	CP5=0;
	CP6=0;
	// Allpass Filters
	AP1=0;
	AP2=0;

	// Lowpass filter setup - One pole lowpass
	// (in feedback loop of comb filters)
	// Q15
	//A1 = 0.1333;
	A1=4368;
	//B0 = 0.5666;
	B0=18566;
	//B1 = 0.5666;
	B1=18566;

	// Previous input to filter, x(n-1)
	C1PreI=0;
	C2PreI=0;
	C3PreI=0;
	C4PreI=0;
	C5PreI=0;
	C6PreI=0;
	// Previous output of filter, y(n-1)
	C1PreO=0;
	C2PreO=0;
	C3PreO=0;
	C4PreO=0;
	C5PreO=0;
	C6PreO=0;

	while (effect == REVERB) {

		//int LeftInput, RightInput, Left, Right
		AIC23_IO(2, Left, Right);

		FXInput=((LeftInput*(long)inScale)>>15);
		//printf("Input: %8d \n", LeftInput);

		// Compute intermediate sample (from comb filter bank)

	midSamp=(long)CDB1[CP1]+CDB2[CP2]+CDB3[CP3]+CDB4[CP4]+CDB5[CP5]+CDB6[CP6];
//Q15
		//printf("midSamp: %12ld \n", midSamp);

		// Compute output sample (from allpass filters)
	AOut1=(-1*AG1*midSamp)+((long)ADB1[AP1]<<15); // Q30
	AOut2=(-1*AG2*midSamp)+((long)ADB2[AP2]<<15); // Q30
	output=((((AOut1+AOut2)>>15)*wmix) + ((long)LeftInput*dmix))>>15;
	//printf("output: %12ld \n", output);

	// Saturate and assign
		Left=satur8(output);
		Right = Left;

	// Save old CDB values for later use
	Temp1=CDB1[CP1];
```

```
      Temp2=CDB2[CP2];
      Temp3=CDB3[CP3];
      Temp4=CDB4[CP4];
      Temp5=CDB5[CP5];
      Temp6=CDB6[CP6];

      // Replace oldest comb buffers with newest
      // (lowpass filter in feedback loop)
      CDB1[CP1]=satur8((((long)CG1*(((long)B0*CDB1[CP1] + (long)B1*C1PreI -
(long)A1*C1PreO)>>15))>>15) + FXInput);
      CDB2[CP2]=satur8((((long)CG2*(((long)B0*CDB2[CP2] + (long)B1*C2PreI -
(long)A1*C2PreO)>>15))>>15) + FXInput);
      CDB3[CP3]=satur8((((long)CG3*(((long)B0*CDB3[CP3] + (long)B1*C3PreI -
(long)A1*C3PreO)>>15))>>15) + FXInput);
      CDB4[CP4]=satur8((((long)CG4*(((long)B0*CDB4[CP4] + (long)B1*C4PreI -
(long)A1*C4PreO)>>15))>>15) + FXInput);
      CDB5[CP5]=satur8((((long)CG5*(((long)B0*CDB5[CP5] + (long)B1*C5PreI -
(long)A1*C5PreO)>>15))>>15) + FXInput);
      CDB6[CP6]=satur8((((long)CG6*(((long)B0*CDB6[CP6] + (long)B1*C6PreI -
(long)A1*C6PreO)>>15))>>15) + FXInput);

      // Update PreI and PreO values
      C1PreI=Temp1;
      C2PreI=Temp2;
      C3PreI=Temp3;
      C4PreI=Temp4;
      C5PreI=Temp5;
      C6PreI=Temp6;
      C1PreO=CDB1[CP1];
      C2PreO=CDB2[CP2];
      C3PreO=CDB3[CP3];
      C4PreO=CDB4[CP4];
      C5PreO=CDB5[CP5];
      C6PreO=CDB6[CP6];

      //printf("Feedback Loop Vals: \n");
      //printf("%8d, %8d, %8d, %8d, %8d, %8d \n", C1PreO, C2PreO, C3PreO,
C4PreO, C5PreO, C6PreO);

      // Replace oldest allpass buffer value with newest
      ADB1[AP1]=satur8((((long)AG1*(AOut1>>15))>>15)+midSamp);
      ADB2[AP2]=satur8((((long)AG2*(AOut2>>15))>>15)+midSamp);

      // Update oldest buffer sample pointer
      //Comb Buffer 1
      if (CP1 >= (CD1-1)) CP1=0;
      else CP1++;
      //Comb Buffer 2
      if (CP2 >= (CD2-1)) CP2=0;
      else CP2++;
      //Comb Buffer 3
      if (CP3 >= (CD3-1)) CP3=0;
      else CP3++;
      //Comb Buffer 4
      if (CP4 >= (CD4-1)) CP4=0;
      else CP4++;
      //Comb Buffer 5
      if (CP5 >= (CD5-1)) CP5=0;
      else CP5++;
      //Comb Buffer 6
      if (CP6 >= (CD6-1)) CP6=0;
      else CP6++;
      //All-pass Buffer 1
      if (AP1 >= (AD1-1)) AP1=0;
      else AP1++;
      //All-pass Buffer 2
```

```
      if (AP2 >= (AD2-1)) AP2=0;
      else AP2++;

      effect=testForArg(effect);
      }

}

void ringmod(int fo)
{

      long index=0;
      long lookup;
      long output;

      // scale input down from 0 to 32768
      // New range: 50 to 562 Hz
      fo=(fo>>7)+50;

      /* process values on both left and right */
      while (effect == RINGMOD) {
            AIC23_IO(2, Left, Right);

            // oscillator = sin(2*pi*f0*n/fs);
            // (index<<15)/FS = time in Q15
            lookup=((SIN_TABLESIZE*((long)fo*((index<<15)/FS)))>>15)%1024; //Q0
            //printf("%8ld \n", lookup);

            output=(long)LeftInput*SIN_TABLE[lookup]; //Q30

            Left=satur8(output>>15);
            Right = Left;


            // (32768)/fo = period, Q15
            // (index<<15)/FS = time in Q15
            if(((index<<15)/FS) >= (32768/(long)fo)) index = 0;
            else index++;

            effect=testForArg(effect);
      }
}


void tremolo(int swells, int depth)
{
      /* Input parameter swells ranges from 4096 (1) to 20480 (5)
                  swells per second in Q12
       * Input parameter depth ranges from 0 to 16384 (0.5) in Q15 */

      long gain = 0; //Q15
      long output;
      long counter = 0;
      long lookup;
      int offset=32768-depth; // Q15

      long scale;

      // Scaling
      // Input ranges from 0 to 32767 in Q15

      // Relates to 4096 to 20480 in Q12 for swells
      // Divide input by 2 and add 4096
      scale = (((long)swells<<1)/(long)2)>>1; // Q15*Q4=Q19
      scale = scale + (long)4096;
      swells = (int) scale;
```

```
      swells=20480;

      // Relates to 0 to 16384 in Q15 for depth
      depth = depth>>1;

      /* process values on both left and right */
      while (effect == TREMOLO) {

            // oscillator = sin(2*pi*f0*n/fs);
            // (counter<<15)/FS = time in Q15
            lookup=((SIN_TABLESIZE*((swells*((counter<<15)/FS))>>12))>>15)%1024;
//Q0

            // Display counter and lookup
            //printf("%8ld ", counter);
            //printf("%8ld ", lookup);

            gain=offset+(((long)depth*SIN_TABLE[lookup])>>15);

            // Display gain coefficients
            //printf("%8ld \n", gain);

            AIC23_IO(2, Left, Right);

            output = (LeftInput*gain)>>15; //Q15

            Left = satur8(output);
            Right = Left;

            // (32768<<12)/swells = period, Q15
            // (counter<<15)/FS = time in Q15
            if(((counter<<15)/FS) >= ((32768<<12)/(long)swells)) counter = 0;
            else counter++;

            effect=testForArg(effect);
      }
}
```

DAFX.cmd

```
/*****************************************************************
    LINKER command file for EECS 452 C5510DSK memory map.
    Small memory model --- Version 1.0  25Jul2003 KM
    Added large pages  --- Version 1.01 18Nov2003 KM

    Appears to work ok for large memory model as well.
    Linker represents addresses and allocations using 8-bit bytes!!!!!!

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

-stack    0x2000 /* Primary stack size   .. fills one 8KB block */
-sysstack 0x1000 /* Secondary stack size .. fills one half 8KB block */
-heap     0x2000 /* Heap area size       .. fills one 8KB block */

-c               /* Use C linking conventions: auto-init vars at runtime */
-u _Reset        /* Force load of reset interrupt handler */

MEMORY
{
  PAGE 0:      /* ---- Unified Program/Data Address Space ---- */
    MMR_RSVD      : origin = 0x000000, length = 0x0000BF /* 192 bytes MMR
reserved */
```

```
    VECT   (RWIX) : origin = 0x000100, length = 0x000100 /* 256 byte interrupt
vector */
    DARAM  (RWIX) : origin = 0x000200, length = 0x00FD00 /* almost 64KB of DARAM
*/
    SARAM0 (RWIX) : origin = 0x010000, length = 0x010000 /*   64KB of SARAM */
    SARAM1 (RWIX) : origin = 0x020000, length = 0x020000 /*  128KB of SARAM */
    SARAM2 (RWIX) : origin = 0x040000, length = 0x010000 /*   64KB of SARAM */
    /* SDRAM has 0xB0000 37776 KB of SDRAM .. notall  allocated here */
    SDRAM0 (RWIX) : origin = 0x050000, length = 0x010000 /*   64KB of SDRAM */
    SDRAM1 (RWIX) : origin = 0x060000, length = 0x020000 /*  128KB of SDRAM */
    SDRAM2 (RWIX) : origin = 0x080000, length = 0x020000 /*  128KB of SDRAM */
    SDRAM3 (RWIX) : origin = 0x0A0000, length = 0x020000 /*  128KB of SDRAM */
    SDRAM4 (RWIX) : origin = 0x0C0000, length = 0x020000 /*  128KB of SDRAM */
    SDRAM5 (RWIX) : origin = 0x0E0000, length = 0x020000 /*  128KB of SDRAM */
    SDRAM6 (RWIX) : origin = 0x100000, length = 0x020000 /*  128KB of SDRAM */
    SDRAM7 (RWIX) : origin = 0x120000, length = 0x020000 /*  128KB of SDRAM */
    SDRAM8 (RWIX) : origin = 0x140000, length = 0x020000 /*  128KB of SDRAM */
    SDRAM9 (RWIX) : origin = 0x160000, length = 0x020000 /*  128KB of SDRAM */
    SDRAM10(RWIX) : origin = 0x180000, length = 0x020000 /*  128KB of SDRAM */
    SDRAM11(RWIX) : origin = 0x1A0000, length = 0x020000 /*  128KB of SDRAM */
    FLASH         : origin = 0x400000, length = 0x80000
    VECS (RIX)    : origin = 0xffff00, length = 0x000100 /* 256-byte int
vector*/
}

SECTIONS
{
    .text       > SARAM0 PAGE 0 /* Code */

/* These sections must be on same physical memory page */
/* when small memory model is used */

    .data       > DARAM  PAGE 0 /* Initialized vars */
    .bss        > DARAM  PAGE 0 /* Global & static vars */
    .const      > DARAM  PAGE 0 /* Constant data */
    .sysmem     > DARAM  PAGE 0 /* Dynamic memory (malloc) */
    .stack      > DARAM  PAGE 0  ALIGN = 0x2000 /* Primary system stack */
    .sysstack   > DARAM  PAGE 0  ALIGN = 0x2000 /* Secondary system stack */
    .cio        > SARAM0 PAGE 0 /* C I/O buffers */
    .cdbuffer     > SARAM1 PAGE 0
    .thebuf       > SARAM2 PAGE 0

/* These sections may be on any physical memory page */
/* when small memory model is used */

    .switch     > SARAM0 PAGE 0 /* Switch statement tables */
    .cinit      > SARAM0 PAGE 0 /* Auto-initialization tables */
    .pinit      > SARAM0 PAGE 0 /* Initialization fn tables */

     vectors    > VECT   PAGE 0 /* Interrupt vectors */

/* Allocate pages in SARAM for when using large memory model */

     SARAMA     > SARAM0  PAGE 0
     SARAMB     > SARAM1  PAGE 0
     SARAMC     > SARAM2  PAGE 0

/* Allocate pages in SDRAM for when using large memory model */

     SDRAMA     > SDRAM0  PAGE 0   /* 32K word page */
     SDRAMB     > SDRAM1  PAGE 0   /* 64K word page */
     SDRAMC     > SDRAM2  PAGE 0   /* 64K word page */
     SDRAMD     > SDRAM3  PAGE 0   /* 64K word page */
     SDRAME     > SDRAM4  PAGE 0   /* 64K word page */
     SDRAMF     > SDRAM5  PAGE 0   /* 64K word page */
     SDRAMG     > SDRAM6  PAGE 0   /* 64K word page */
```

```
     SDRAMH     > SDRAM7  PAGE 0   /* 64K word page */
     SDRAMI     > SDRAM8  PAGE 0   /* 64K word page */
     SDRAMJ     > SDRAM9  PAGE 0   /* 64K word page */
     SDRAMK     > SDRAM10 PAGE 0   /* 64K word page */
     SDRAML     > SDRAM11 PAGE 0   /* 64K word page */
}
```

---

sin_table.h

```
int SIN_TABLE[1024]={
       0,      201,      402,      603,      804,     1005,     1206,     1407,
    1608,     1809,     2009,     2210,     2410,     2611,     2811,     3012,
    3212,     3412,     3612,     3811,     4011,     4210,     4410,     4609,
    4808,     5007,     5205,     5404,     5602,     5800,     5998,     6195,
    6393,     6590,     6786,     6983,     7179,     7375,     7571,     7767,
    7962,     8157,     8351,     8545,     8739,     8933,     9126,     9319,
    9512,     9704,     9896,    10087,    10278,    10469,    10659,    10849,
   11039,    11228,    11417,    11605,    11793,    11980,    12167,    12353,
   12539,    12725,    12910,    13094,    13279,    13462,    13645,    13828,
   14010,    14191,    14372,    14553,    14732,    14912,    15090,    15269,
   15446,    15623,    15800,    15976,    16151,    16325,    16499,    16673,
   16846,    17018,    17189,    17360,    17530,    17700,    17869,    18037,
   18204,    18371,    18537,    18703,    18868,    19032,    19195,    19357,
   19519,    19680,    19841,    20000,    20159,    20317,    20475,    20631,
   20787,    20942,    21096,    21250,    21403,    21554,    21705,    21856,
   22005,    22154,    22301,    22448,    22594,    22739,    22884,    23027,
   23170,    23311,    23452,    23592,    23731,    23870,    24007,    24143,
   24279,    24413,    24547,    24680,    24811,    24942,    25072,    25201,
   25329,    25456,    25582,    25708,    25832,    25955,    26077,    26198,
   26319,    26438,    26556,    26674,    26790,    26905,    27019,    27133,
   27245,    27356,    27466,    27575,    27683,    27790,    27896,    28001,
   28105,    28208,    28310,    28411,    28510,    28609,    28706,    28803,
   28898,    28992,    29085,    29177,    29268,    29358,    29447,    29534,
   29621,    29706,    29791,    29874,    29956,    30037,    30117,    30195,
   30273,    30349,    30424,    30498,    30571,    30643,    30714,    30783,
   30852,    30919,    30985,    31050,    31113,    31176,    31237,    31297,
   31356,    31414,    31470,    31526,    31580,    31633,    31685,    31736,
   31785,    31833,    31880,    31926,    31971,    32014,    32057,    32098,
   32137,    32176,    32213,    32250,    32285,    32318,    32351,    32382,
   32412,    32441,    32469,    32495,    32521,    32545,    32567,    32589,
   32609,    32628,    32646,    32663,    32678,    32692,    32705,    32717,
   32728,    32737,    32745,    32752,    32757,    32761,    32765,    32766,
   32767,    32766,    32765,    32761,    32757,    32752,    32745,    32737,
   32728,    32717,    32705,    32692,    32678,    32663,    32646,    32628,
   32609,    32589,    32567,    32545,    32521,    32495,    32469,    32441,
   32412,    32382,    32351,    32318,    32285,    32250,    32213,    32176,
   32137,    32098,    32057,    32014,    31971,    31926,    31880,    31833,
   31785,    31736,    31685,    31633,    31580,    31526,    31470,    31414,
   31356,    31297,    31237,    31176,    31113,    31050,    30985,    30919,
   30852,    30783,    30714,    30643,    30571,    30498,    30424,    30349,
   30273,    30195,    30117,    30037,    29956,    29874,    29791,    29706,
   29621,    29534,    29447,    29358,    29268,    29177,    29085,    28992,
   28898,    28803,    28706,    28609,    28510,    28411,    28310,    28208,
   28105,    28001,    27896,    27790,    27683,    27575,    27466,    27356,
   27245,    27133,    27019,    26905,    26790,    26674,    26556,    26438,
   26319,    26198,    26077,    25955,    25832,    25708,    25582,    25456,
   25329,    25201,    25072,    24942,    24811,    24680,    24547,    24413,
   24279,    24143,    24007,    23870,    23731,    23592,    23452,    23311,
   23170,    23027,    22884,    22739,    22594,    22448,    22301,    22154,
   22005,    21856,    21705,    21554,    21403,    21250,    21096,    20942,
   20787,    20631,    20475,    20317,    20159,    20000,    19841,    19680,
   19519,    19357,    19195,    19032,    18868,    18703,    18537,    18371,
```

```
 18204,   18037,   17869,   17700,   17530,   17360,   17189,   17018,
 16846,   16673,   16499,   16325,   16151,   15976,   15800,   15623,
 15446,   15269,   15090,   14912,   14732,   14553,   14372,   14191,
 14010,   13828,   13645,   13462,   13279,   13094,   12910,   12725,
 12539,   12353,   12167,   11980,   11793,   11605,   11417,   11228,
 11039,   10849,   10659,   10469,   10278,   10087,    9896,    9704,
  9512,    9319,    9126,    8933,    8739,    8545,    8351,    8157,
  7962,    7767,    7571,    7375,    7179,    6983,    6786,    6590,
  6393,    6195,    5998,    5800,    5602,    5404,    5205,    5007,
  4808,    4609,    4410,    4210,    4011,    3811,    3612,    3412,
  3212,    3012,    2811,    2611,    2410,    2210,    2009,    1809,
  1608,    1407,    1206,    1005,     804,     603,     402,     201,
     0,    -201,    -402,    -603,    -804,   -1005,   -1206,   -1407,
 -1608,   -1809,   -2009,   -2210,   -2410,   -2611,   -2811,   -3012,
 -3212,   -3412,   -3612,   -3811,   -4011,   -4210,   -4410,   -4609,
 -4808,   -5007,   -5205,   -5404,   -5602,   -5800,   -5998,   -6195,
 -6393,   -6590,   -6786,   -6983,   -7179,   -7375,   -7571,   -7767,
 -7962,   -8157,   -8351,   -8545,   -8739,   -8933,   -9126,   -9319,
 -9512,   -9704,   -9896,  -10087,  -10278,  -10469,  -10659,  -10849,
-11039,  -11228,  -11417,  -11605,  -11793,  -11980,  -12167,  -12353,
-12539,  -12725,  -12910,  -13094,  -13279,  -13462,  -13645,  -13828,
-14010,  -14191,  -14372,  -14553,  -14732,  -14912,  -15090,  -15269,
-15446,  -15623,  -15800,  -15976,  -16151,  -16325,  -16499,  -16673,
-16846,  -17018,  -17189,  -17360,  -17530,  -17700,  -17869,  -18037,
-18204,  -18371,  -18537,  -18703,  -18868,  -19032,  -19195,  -19357,
-19519,  -19680,  -19841,  -20000,  -20159,  -20317,  -20475,  -20631,
-20787,  -20942,  -21096,  -21250,  -21403,  -21554,  -21705,  -21856,
-22005,  -22154,  -22301,  -22448,  -22594,  -22739,  -22884,  -23027,
-23170,  -23311,  -23452,  -23592,  -23731,  -23870,  -24007,  -24143,
-24279,  -24413,  -24547,  -24680,  -24811,  -24942,  -25072,  -25201,
-25329,  -25456,  -25582,  -25708,  -25832,  -25955,  -26077,  -26198,
-26319,  -26438,  -26556,  -26674,  -26790,  -26905,  -27019,  -27133,
-27245,  -27356,  -27466,  -27575,  -27683,  -27790,  -27896,  -28001,
-28105,  -28208,  -28310,  -28411,  -28510,  -28609,  -28706,  -28803,
-28898,  -28992,  -29085,  -29177,  -29268,  -29358,  -29447,  -29534,
-29621,  -29706,  -29791,  -29874,  -29956,  -30037,  -30117,  -30195,
-30273,  -30349,  -30424,  -30498,  -30571,  -30643,  -30714,  -30783,
-30852,  -30919,  -30985,  -31050,  -31113,  -31176,  -31237,  -31297,
-31356,  -31414,  -31470,  -31526,  -31580,  -31633,  -31685,  -31736,
-31785,  -31833,  -31880,  -31926,  -31971,  -32014,  -32057,  -32098,
-32137,  -32176,  -32213,  -32250,  -32285,  -32318,  -32351,  -32382,
-32412,  -32441,  -32469,  -32495,  -32521,  -32545,  -32567,  -32589,
-32609,  -32628,  -32646,  -32663,  -32678,  -32692,  -32705,  -32717,
-32728,  -32737,  -32745,  -32752,  -32757,  -32761,  -32765,  -32766,
-32767,  -32766,  -32765,  -32761,  -32757,  -32752,  -32745,  -32737,
-32728,  -32717,  -32705,  -32692,  -32678,  -32663,  -32646,  -32628,
-32609,  -32589,  -32567,  -32545,  -32521,  -32495,  -32469,  -32441,
-32412,  -32382,  -32351,  -32318,  -32285,  -32250,  -32213,  -32176,
-32137,  -32098,  -32057,  -32014,  -31971,  -31926,  -31880,  -31833,
-31785,  -31736,  -31685,  -31633,  -31580,  -31526,  -31470,  -31414,
-31356,  -31297,  -31237,  -31176,  -31113,  -31050,  -30985,  -30919,
-30852,  -30783,  -30714,  -30643,  -30571,  -30498,  -30424,  -30349,
-30273,  -30195,  -30117,  -30037,  -29956,  -29874,  -29791,  -29706,
-29621,  -29534,  -29447,  -29358,  -29268,  -29177,  -29085,  -28992,
-28898,  -28803,  -28706,  -28609,  -28510,  -28411,  -28310,  -28208,
-28105,  -28001,  -27896,  -27790,  -27683,  -27575,  -27466,  -27356,
-27245,  -27133,  -27019,  -26905,  -26790,  -26674,  -26556,  -26438,
-26319,  -26198,  -26077,  -25955,  -25832,  -25708,  -25582,  -25456,
-25329,  -25201,  -25072,  -24942,  -24811,  -24680,  -24547,  -24413,
-24279,  -24143,  -24007,  -23870,  -23731,  -23592,  -23452,  -23311,
-23170,  -23027,  -22884,  -22739,  -22594,  -22448,  -22301,  -22154,
-22005,  -21856,  -21705,  -21554,  -21403,  -21250,  -21096,  -20942,
-20787,  -20631,  -20475,  -20317,  -20159,  -20000,  -19841,  -19680,
-19519,  -19357,  -19195,  -19032,  -18868,  -18703,  -18537,  -18371,
-18204,  -18037,  -17869,  -17700,  -17530,  -17360,  -17189,  -17018,
-16846,  -16673,  -16499,  -16325,  -16151,  -15976,  -15800,  -15623,
```

```
  -15446,   -15269,   -15090,   -14912,   -14732,   -14553,   -14372,   -14191,
  -14010,   -13828,   -13645,   -13462,   -13279,   -13094,   -12910,   -12725,
  -12539,   -12353,   -12167,   -11980,   -11793,   -11605,   -11417,   -11228,
  -11039,   -10849,   -10659,   -10469,   -10278,   -10087,    -9896,    -9704,
   -9512,    -9319,    -9126,    -8933,    -8739,    -8545,    -8351,    -8157,
   -7962,    -7767,    -7571,    -7375,    -7179,    -6983,    -6786,    -6590,
   -6393,    -6195,    -5998,    -5800,    -5602,    -5404,    -5205,    -5007,
   -4808,    -4609,    -4410,    -4210,    -4011,    -3811,    -3612,    -3412,
   -3212,    -3012,    -2811,    -2611,    -2410,    -2210,    -2009,    -1809,
   -1608,    -1407,    -1206,    -1005,     -804,     -603,     -402,     -201};
```

---

asciiToInt.c

```
/*****************************************************
*  EECS 452 -- DSP Lab                                *
*  DAFX Final Project                                 *
*                                                     *
*This function converts and ASCII charater into       *
*an interger                                          *
*                                                     *
*                                                     *
*                                                     *
*  Author: Jason Martin                               *
*  Created: 4/4/2005                                  *
*****************************************************/

int asciiToInt(char data)
{
    int value = 0;

    if(data == 0x31)
      {
            value = 1;
    }
      else if(data == 0x32)
      {
            value = 2;
      }
      else if(data == 0x33)
      {
            value = 3;
      }
      else if(data == 0x34)
      {
            value = 4;
      }
      else if(data == 0x35)
      {
            value = 5;
      }
      else if(data == 0x36)
      {
            value = 6;
      }
      else if(data == 0x37)
      {
            value = 7;
      }
      else if(data == 0x38)
      {
            value = 8;
      }
```

```
   return value;
}
```

---

asciiToInt.h

```
/*******************************************************
*  EECS 452 -- DSP Lab                                 *
*  DAFX Final Project                                  *
*                                                      *
*This function converts and ASCII charater into       *
*an interger                                          *
*                                                      *
*                                                      *
*                                                      *
*  Author: Jason Martin                                *
*  Created: 4/4/2005                                   *
*******************************************************/


//This function converts an ascii charater into an integer.
int asciiToInt(char data);
```