DIGITAL GUITAR EFFECTS UNIT AND AMPLIFIER

by

Kevin Salvador

Senior Project

ELECTRICAL ENGINEERING DEPARTMENT

California Polytechnic State University

San Luis Obispo

Table of Contents

List of Tables and Figures

Acknowledgements

I would like to thank my Senior Project advisor Dr. Pilkington for his help and enthusiasm throughout the process. His knowledge of music specifically guitars and the effects used with them helped tremendously. Without the knowledge acquired from his Digital Signal Processing course, which was taking concurrently with the first quarter of senior project, none of this would be possible. I’d also like to thank my lab partners from the DSP laboratory, David Smith and Eric Spinks whom help code the design functions used in this project. Also thank you to Dr. DePeiro for providing the template C code for implementing the digital filters of TI’s DSK boards.

Abstract

This report outlines the design and implementation of a digital guitar effects unit and amplifier. The main portion of this project consisted of the digital equalizer and effects. Several commercial equalizers were researched in order to decide the typical frequency bands and average amount of bands total. Eventually 8 bands were selected. A range of approximately 20Hz-3kHz was chosen based on test data of guitar signals. Popular effects that were incorporated in this project include Distortion, Echo, Reverb, Chorus and Flanger. The digital processor chosen was a Texas Instruments c6713 floating point processor. Designs for the various filters were done in MatLab and implementation on the processor was done through TI's Code Composer Studio.

I. Introduction

This DSP and amplifierunit is targeted towards guitarists. Electric guitars require an amplifier in order for the sounds to be audible. This project also provides them with signal processing “effects” to change the tones of their instrument. An alternative for these customers are analog effects filters such as simple passive filters. Since this project will make use of digital filters and a programmable processor, there is more flexibility and user control for desired output. The use of digital filters also reduces the size and number of components required for more complex processes. The DSP processor will also be combined with the amplifier in one unit for ease of transportation.

II. Background

Signal processing can be achieved in both the analog and digital domains. The difference between analog and digital is that analog waveforms are continuous in both time and amplitude while digital is discrete in both respects.

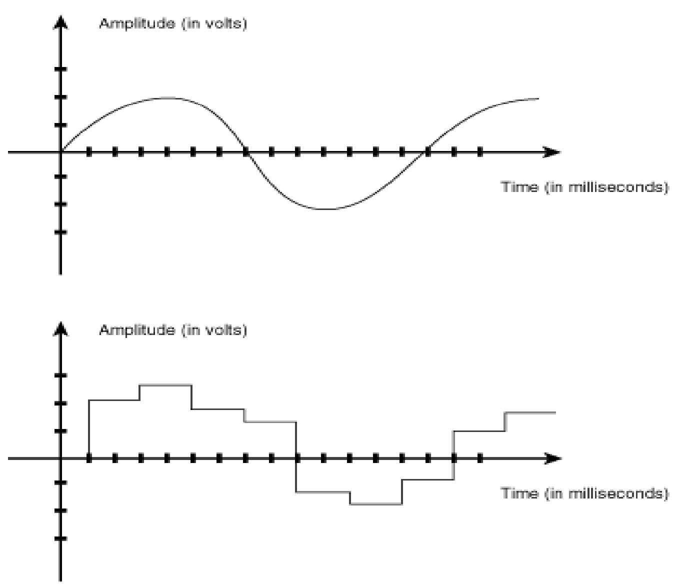


Figure 1: Analog vs Digital signal

In the sense of digital processing of musical instruments, the analog signal must be converted using an Analog-to-Digital converter (ADC). An ADC takes the analog signal and samples it in time and converts the amplitudes to binary values and in turn creating the discrete components of a digital signal. The rate at which analog signals are sampled is known as its sampling frequency and is either presented in Hz or samples/second. To power a transducer, a device that converters electrical energy to mechanical energy i.e a speaker, the digital signal must then be changed back using a Digital-to-Analog converter (DAC).

One of the most common forms of signal processing is filtering. In general the main purpose of filters is to reject and allow certain portions of the signal. In terms of audio signals certain frequencies are either attenuated or amplified. An equalizer in its most basic form is a combination of filters that controls the amplitude of the audio signal at defined frequencies. An example is a 2-band equalizer which uses a low pass filter and a high pass filter to control the gain of the lower frequencies, bass, and the higher frequencies, treble, independently. Another example of a commonly used audio filter is a comb filter which delays portions of the audio signal.

A majority of the effects used in this signal processor can be created using filters. The equalizer uses 1 low pass filter, 1 high pass filter and 6 band pass filters. The reverb, echo, chorus and flanger effects can all be achieved using various forms of the comb filter.

III. Requirements

Functional Requirements

-amplification of audio signal with controllable gain

-Digital signal processing to provide filtering and modulation of audio signal

-user controlled Equalizer (EQ)

-distortion

-presets (Reverb, Echo, Flanger, Chorus, and Auto-Wah)

Performance Requirements

-8Band EQ

-5 preset effects plus distortion

Critical System Parameter Selections / Settings

-drive 26W 8ohm speaker, @±15V, max current out ≈ 1.94-2.09A.

-SNR >60dB

-Nyquist Frequency >3.4kHz; Sampling Frequency > 6.8kHz.

-ADC input > 1Vpp

-ADC,DAC bit >8 bits

-120V AC 60Hz =>±15VDC dualsupply for gain stage and ±25VDC for output stage

IV. Design Approach Alternatives

The major design choice lies in the implementation of the digital equalizer. There are two main alternatives to consider when designing the equalizer. One method is Frequency Sampling (FIR) which makes controlling the magnitude response easy by directly

changing the |H(F)| values at certain frequency. The second method is using an individual band pass filter (IIR) for each frequency band. This method benefits in providing the designer more control over the actual range of each band by being able to select the appropriate cutoff frequencies.

The final design was chosen to be individual band pass filters. At the available sampling rates the method of Frequency Sampling required a filter of length X to achieve accurate positions for the desired bands. With individual band pass filters the design was based primarily on the exact center frequencies of each filter. This allowed for a much smaller order final filter of 16 consisting of 8 separate 2nd order band pass filter

V. Project Design

Blackbox Diagram

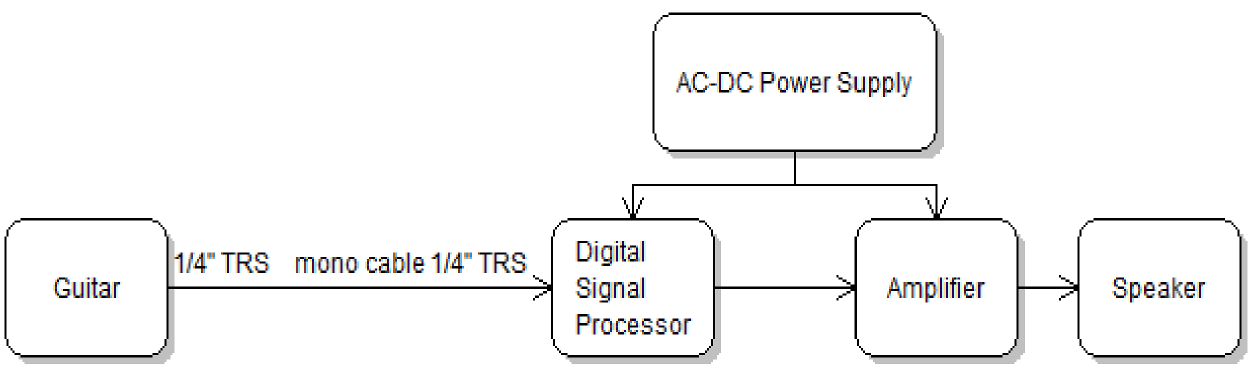


Figure 2: Blackbox Diagram

Definition of User Interfaces

This system will have an LCD interface that displays the current effect state of the DSP. A single button will be used to cycle through the various effect states. The LCD will also display the magnitude response of the EQ filter. Two buttons will be used to cycle through the various bands and two other buttons will be used to control the gain/attenuation. The volume of the amplifier will be controlled by the analog circuits via a potentiometer.

Due to issues outlined in the conclusion section of this report an LCD was not used and on board LEDs were used instead to display a binary value representing the current effect that was applied

Overall System Functional Diagram

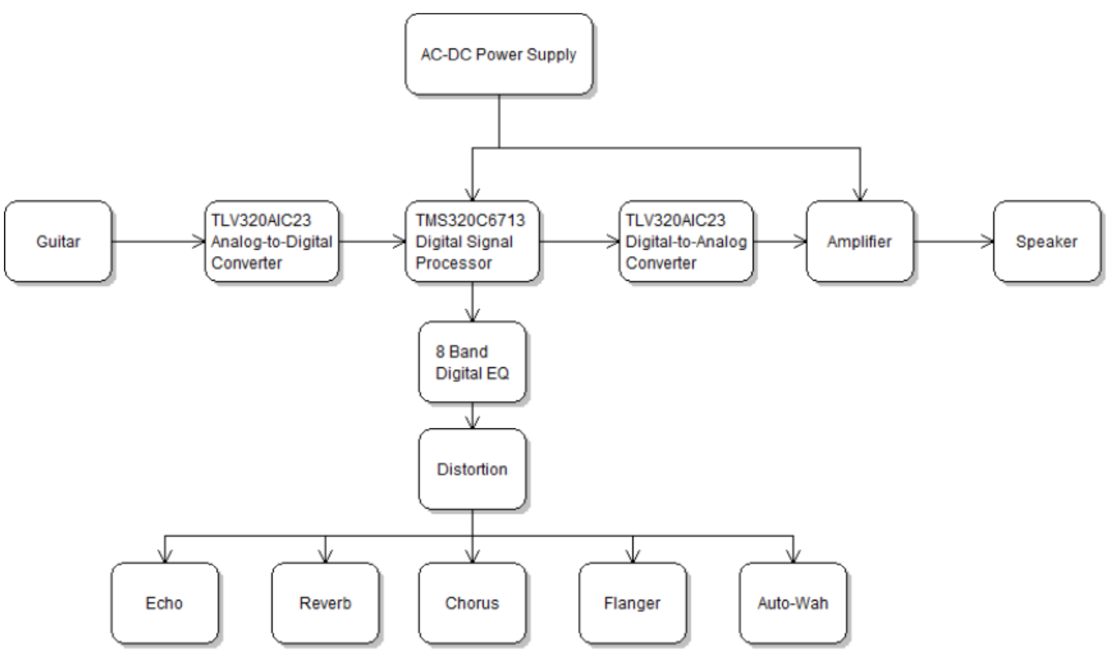


Figure 3: Functional Block Diagram

Analog-to-Digital Converter and Digital-to-Analog Converter

The ADC and DAC are found onboard the TI DSK6713 on the TLV320AIC23B Stereo Audio Codec. This chip has 24-bit sigma-delta converters. Both can achieve sampling rates from 8kHz – 96kHz. The ADC has an SNR of 90dB while the DAC has an SNR of 100dB. It also has an integrated headphone amplifier. The sampling frequency used for this project was 16kHz. This was chosen to provide accurate resolutions for the lower frequencies of the digital EQ. As the sampling frequency increases the lower band pass filters begin to overlap due to the minute differences between their digital frequencies. The relation between the analog frequency and digital frequency is:

$$F(digital\ frequency) = \frac{analog\ frequency}{sampling\ frequency}$$

Although an 8kHz sampling frequency would have provided a more accurate resolution, it created a 3.34ms delay between input. While not completely audible this delay was approximately a third of the maximum delay required for the flanger effect.

The sampling frequency also was required to be greater than 6.8kHz therefore having a Nyquist rate greater than 3.4kHz. This specification was obtained by running a guitar signal through an oscilloscope with Fast-Fourier Transform (FFT) capabilities. The highest fundamental frequency of a guitar was found to be approximately 1kHz. The 5^{th} and 7^{th}had an attenuation of XX dB and therefore were deemed not required for proper signal reproduction. A plot of the FFT is shown in Figure X.XX.

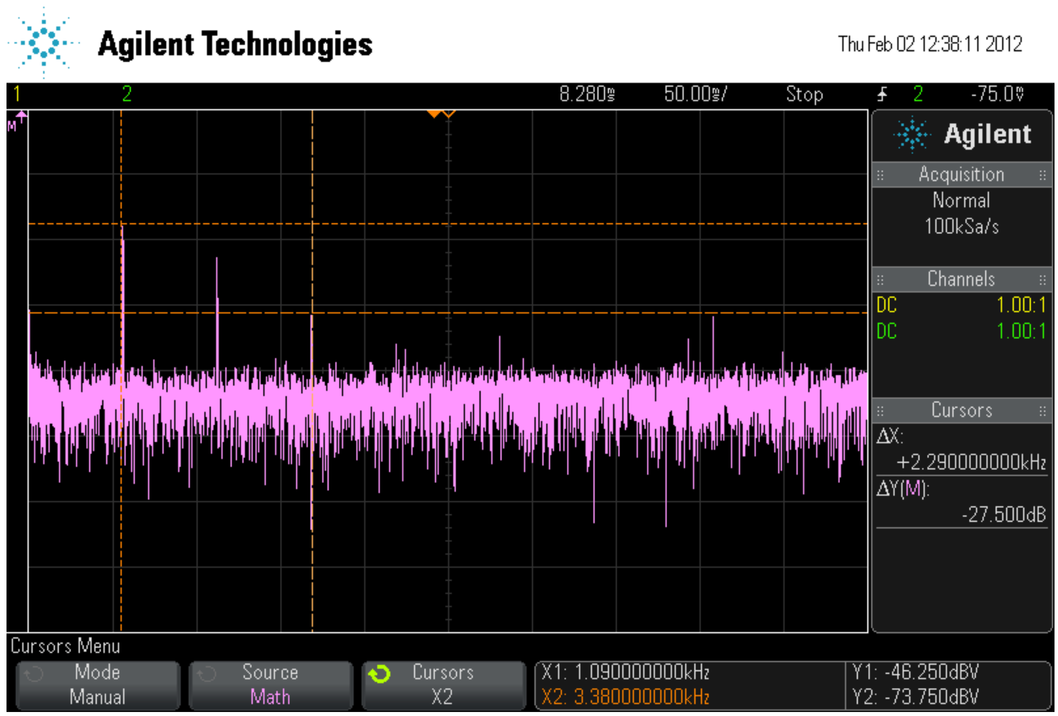


Figure 4: FFT of guitar signal at highest frequency (E string 24th fret)

Interfacing between the DSP and Stereo Audio Codec is done solely by the provided board support library (BSL) and chip support library (CSL).

Digital Signal Processor – Digital Equalizer and Effects

The main functionality of the DSP will be to equalize and filter the signal through a user controlled 8Band EQ. Each band will have a center or cutoff frequency of 128Hz, 256,Hz, 512Hz,800Hz, 1150Hz, 1500Hz, 1800Hz, 2200Hz. The gain of each band will range from ±12dB. The DSP will also have several preset effects that the user can choose to use. These 5 presets are Reverb, Distortion, Delay, Chorus, Flanger and Wah.

A TI/Spectrum digital DSK6713 was chosen as the development kid for this project. It uses a TI TMS320C6713B floating-point DSP which runs at 225MHz

and can operate approximately 1800 MIPS (million instructions per second) and 1350 MFLOPS (million floating point operations per second).

This development board was chosen primarily for its speed and my familiarity with its BSL and CSL. In our Digital Signal Processing course (EE 419) a TI DSK5416 was used and featured very similar function in their BSLs and CSLs. By using the similar libraries I was able to easily translate Dr. DePeiro's template code which was used in our DSP class.

Digital EQ

Prior to DSP implementation, the separate band pass filters of the EQ were designed in MatLab using the functions found in Appendix C. The required poles and zeroes for each 2nd order filter were selected based on center frequencyusing the relation:

$$e^{\pm j\theta} \; where \; \theta = 2\pi(F)$$

The poles were multiplied by a factor of .95 in order to pull the gain down since a pole on the unit circle would create a gain of infinity. The zeroes were initially multiplied by a factor of .9 to further reduce the gain. The gain factor K was then adjusted to normalize the gain to a value of 1 or 0 dB. The function then outputs the required coefficients to be used in the difference equations for DSP implementation.

The pole zero diagram and plot of the 512Hz centered band pass filter is shown below as an example:

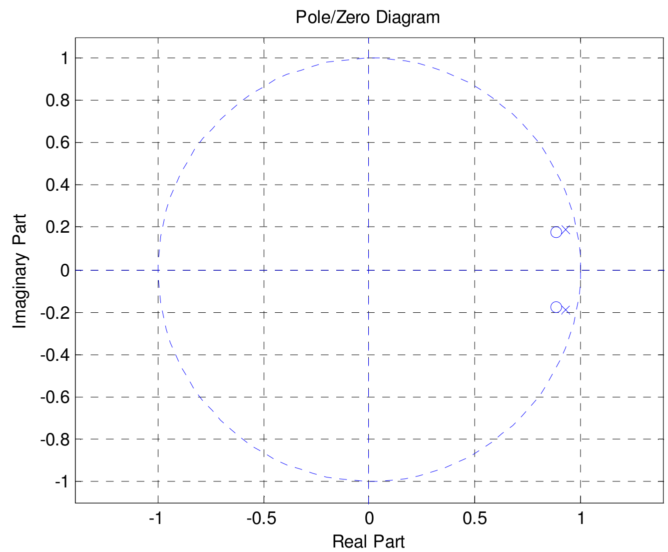


Figure 5: Pole-Zero diagram for band pass filter with Fc of 512Hz

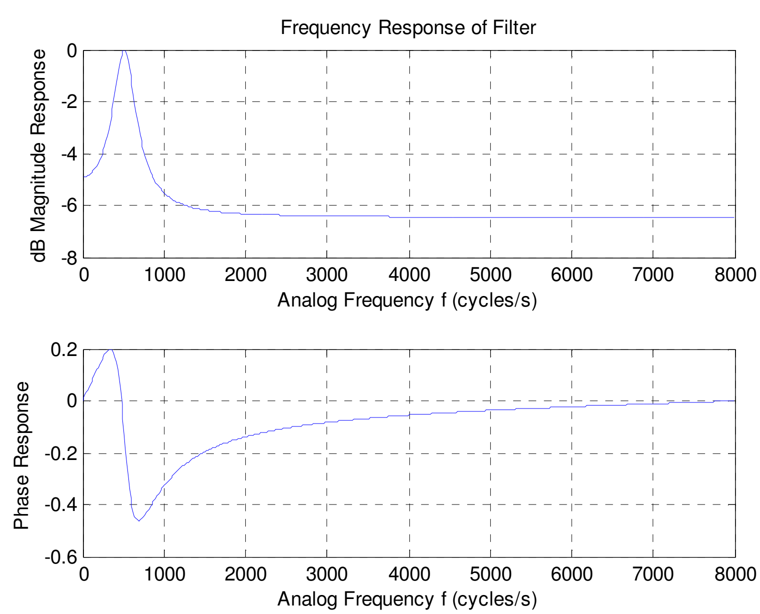


Figure 6: dB magnitude and phase response vs of bandpass filter with Fc = 512Hz

After each filter was simulated, the resulting coefficients were coded to create the filters on the DSP. The difference equation for each filter was coded in using the Direct Form II Transposed structure for 2nd order filters. This form of the difference equations cuts down the number of delays in half by combining the output and inputs that contain the same delayed index such as x[n-1] and y[n-1]. These are placed in an intermediate variable to used in the final difference equation. This speeds up the process by requiring the processor to update the input values half of the times that it normally would. The resulting implemented equations are shown below as well as a signal flow diagram of the direct form II transpose.

$$y[n] = B_0 x[n] + w1[n-1]$$

$$w1[n] = A_1 y[n] + B_1 x[n] + w2[n-1]$$

$$w2[n] = A_2 y[n] + B_2 x[n]$$

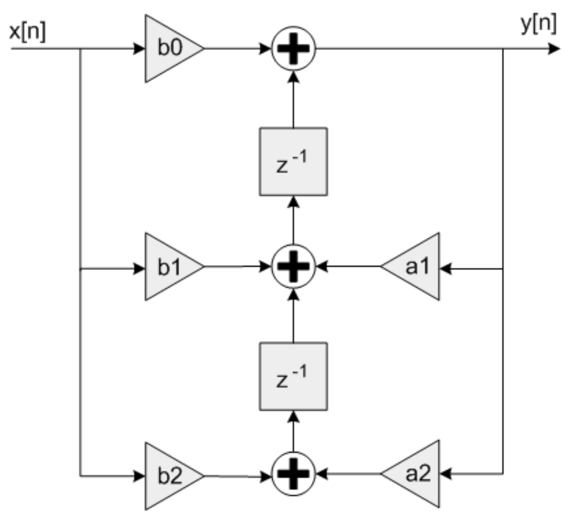


Figure 7: Signal Flow graph for the Direct Form II Transposed Difference Equation

The equalizer is applied at all times for every effect. A simple unity all pass filter is set at default by setting the gain factor K of each filter to 1.

Each gain factor is given its own variable which is placed in an array. The exact gain values are arranged in a look up table ranging from -12dB to 12 dB in 1dB increments. These dB values were converted to V/V gain values before being

implemented in the program code. Each gain factor is accessed by moving inside the array and the gains are changed by moving inside the gain look-up-table. To apply the gain a function is used to multiply the new values and the B coefficients of the input prior to being implemented in the difference equation.

Echo

The echo effect takes in an input signal and creates an attenuated delayed version of the input that is added to the original. The delay time is constant over time and simulates the stationary barrier that creates echoes in real life. The difference equation and block diagram for the echo effect is given by Ambardar as

$$y[n] = x[n] + \alpha x[n - Delay]$$

This specific design uses four separate delays to simulate a long decaying echo. The delay is created in code by using a delay buffer that iterates for a specified count length and takes in the input then outputs the delayed input when it reaches the count length. This specific echo effect was designed with the length of 2500 iterations.

Chorus

The chorus effect simulates the difference in timing and volume of each individual source as is typical of real life musicians. The main design of the chorus effect is very similar to the echo. Unlike the echo, the delay times for

chorus varies with time and the gain is normally also varied. Ambardar defines the difference equation as

$$y[n] = x[n] + \alpha x[n - variable\ Delay]$$

Although the delay varies, Ambardar states that the delays fall into a range of 20-30ms. To vary the delay a random number generate was used to output numbers in the range of 450-800 or 18-32ms. These random delay values were then placed in a look up table and changed every 1250 iterations or 50ms.

Flanger

The flanger effect is implemented similarly to the chorus. The main difference is that the range of the varied delay times is less than 10ms rather than 20-30ms. The chosen range for the random delay look-up-table is 100-250 iterations or 4-10ms. The same function that accessed the look up table in the chorus effect is used for this.

Reverb

The reverb effect simulates an oscillating echo that continues to vibrate and decay in volume. This is similar to echo but uses a feedback system therefore making this filter an IIR filter rather than a feed-forward FIR filter such as the three previous effects. Like the echo the delay is constant in time. Ambardar states that the reverb and echo are both part of sound reflections from a barrier but reverb consists of the reflections that occur later after the initial transmission. To achieve this feedback the delayed signal is of the output rather than the input. A decaying

gain is required for this filter or else the delay will create an infinite loop since an output will always be existent.

Auto-Wah

The wah-wah effect creates a human sounding tone from a guitar signal. The effect gets its name from the wah sounding vocalization. This uses a band pass filter with a varying center frequency. The change in frequency is normally triggered by an external input in the form of a pedal. In an auto-wah the frequency is varied internally by either an envelope detector or a low frequency oscillator (LFO).

This design uses an LFO to vary the frequency. Frequencies ranging from 100-1000 in increments of 10 were placed in a look-up-table. The LFO was created by having a counter oscillate between the bounds of the look-up-table.

Distortion

The simplest way of creating distortion is to clip the amplitude of the output signal. In the analog domain a clipping circuit consisting of resistors and diodes. In the digital domain arithmetic overflow is applied to the output. Overflow is achieved when the input signal is greater than the allowable value defined by the outputs data type. Normally overflow is avoided by design because distorted signals are unwanted. To apply distortion a threshold value is used to compare the input and if it’s greater than the threshold, then the threshold value is outputted.

Power Amplifier

The voltage gain stage of the amplifier comprises of a single LM741 op amp and uses a 10kΩ SPST potentiometer in series with a 4.7kkΩ resistor to vary the gain from 4.7V/V to 14.7V/V.

The output stage is created by a BJT output stage. Four BJTs, 2 NPN and 2 PNP are first connected together in a darlington pair configuration in order to increase the current amplification. The two darlington pairs are then combined in a push pull output. This allows the load to be driven by a positive as well as negative current. In doing so a small region, known as the “dead zone” appears where neither of the transistors are active and 0 current is produced. This is reduced by connecting diodes between the base junctions of the transistors and the output of the voltage gain stage.

These circuits were designed and tested using Linear Technologies LTspice IV. The simulated circuits and input and output waveforms are shown below.

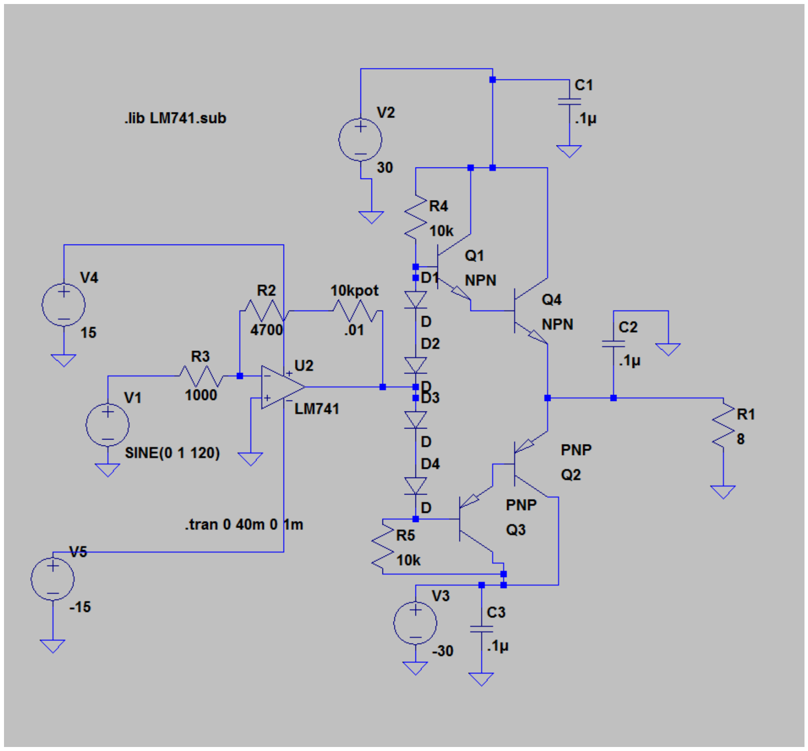


Figure 8: LTspice simulated circuit of voltage gain stage and power output stage

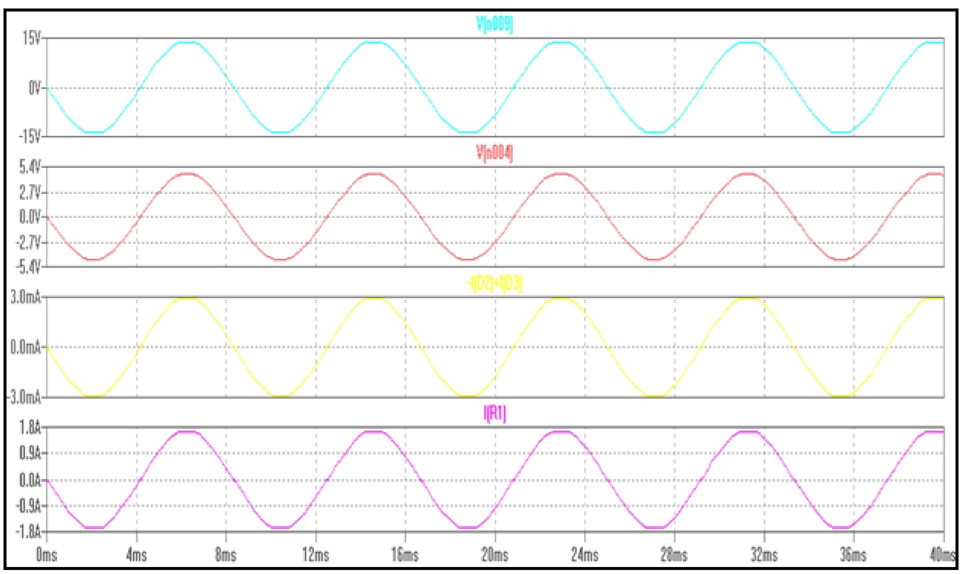


Figure 9: LTspice simulated waveforms of figure 7

The final design of the output stage uses a ±25V dual supply rather than ±30V.

Speaker

The speaker that was chosen for this project was salvaged from a broken Peavey Backstage amplifier. The speaker is rated for a maximum of 26W.

Power Supply

The center tapped transformer is rated to convert 120V AC on the primary side to 25.2V (12.6V-0V-12.6V) AC on the secondary side. A SPST switch and a 250V/3A fast-acting fuse is placed in series between the live wire of the transformer and live wire of the AC plug. This protects the transformer from drawing excessive current and also allows for the power supply to be turned on and off as opposed to being on when it is plugged to an outlet.

The secondary windings are connected to a full wave bridge rectifier to create an entirely positive voltage. Two 2200uF capacitors are used to reroute the voltage from the secondary windings to a second full wave bridge rectifier in order to achieve a dual power supply of ±25VDC. The voltage being supplied directly from the transformer is smoothed out by a 1000uF capacitor and the voltage from the capacitors is smoothed using a 2200uF. These smoothing capacitors create a flat voltage and therefore convert the rectified AC voltage to DC.

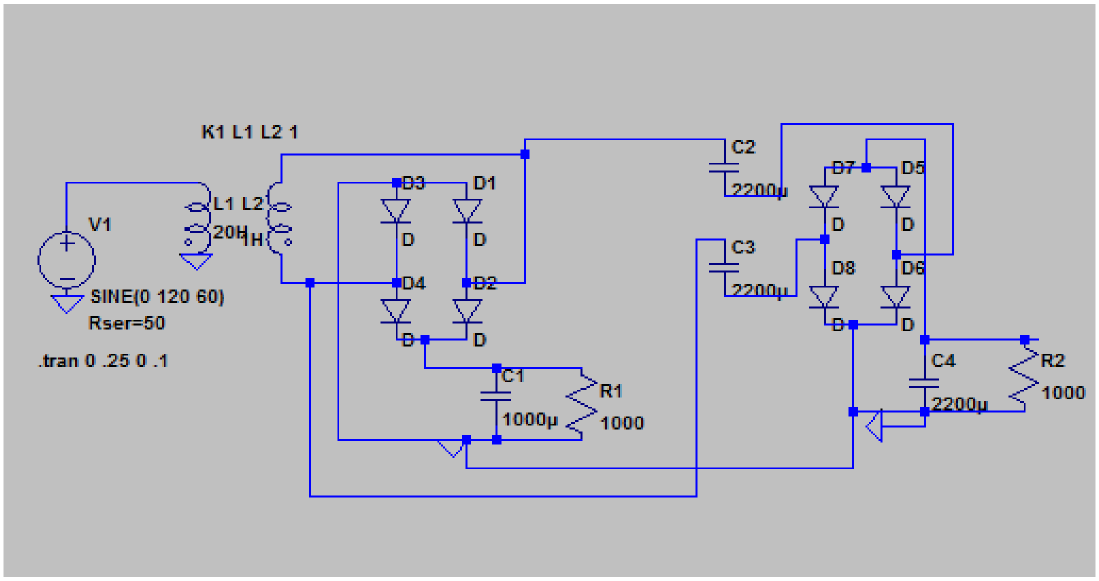


Figure 10: LTspice simulated circuit of dual power supply

The ±25VDC rails will be used to power the output transistors. The voltage will then be reduced via LM317T and LM337T linear voltage regulators. LM317T will provide a +15V rail and the LM337T will provide a -15V rail. This will be used to power the op amps used in the preamp stage and voltage gain stage.

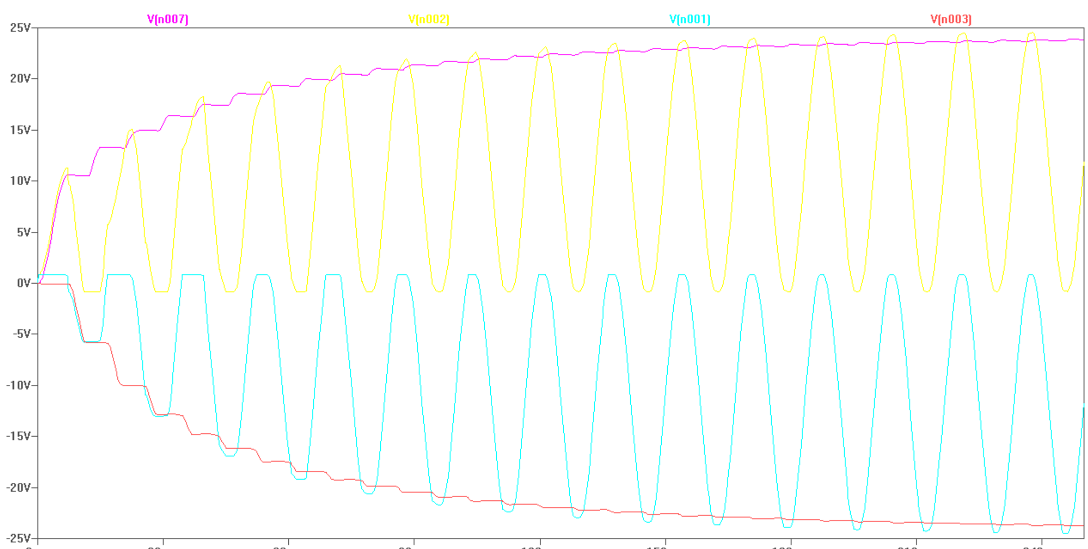


Figure 11: LTspice simulated dual rectified and smoothed waveforms

VI. Physical Construction and Implementation

It will be powered using an AC-DC power supply which is incorporated in the design project. The analog circuits were intended to be mountedon printed circuit boards. The DSP development board along with analog circuits will be packaged in a wooden crate speaker box.

Unfortunately the analog circuits were not packaged as initially proposed and instead were just tested and prototyped on bread boards. The amplifier circuit is shown in Figure 14. Final packaging was not achieved.

Figure 12: Setup used during the Senior Project Exhibition. Not included is board's power supply

Push buttons were wired to the GPIO pins of the DSP via bread board. To keep the board useable for future projects soldering and other permanent applications were avoided. The cluster of 4 buttons controls the digital equalizer. The two buttons that are

setup horizontal control the selection of the frequency band, while the two vertical buttons control the gain. The lone button to the right is used to cycle through the different effects. One of the onboard switches was used to control distortion.

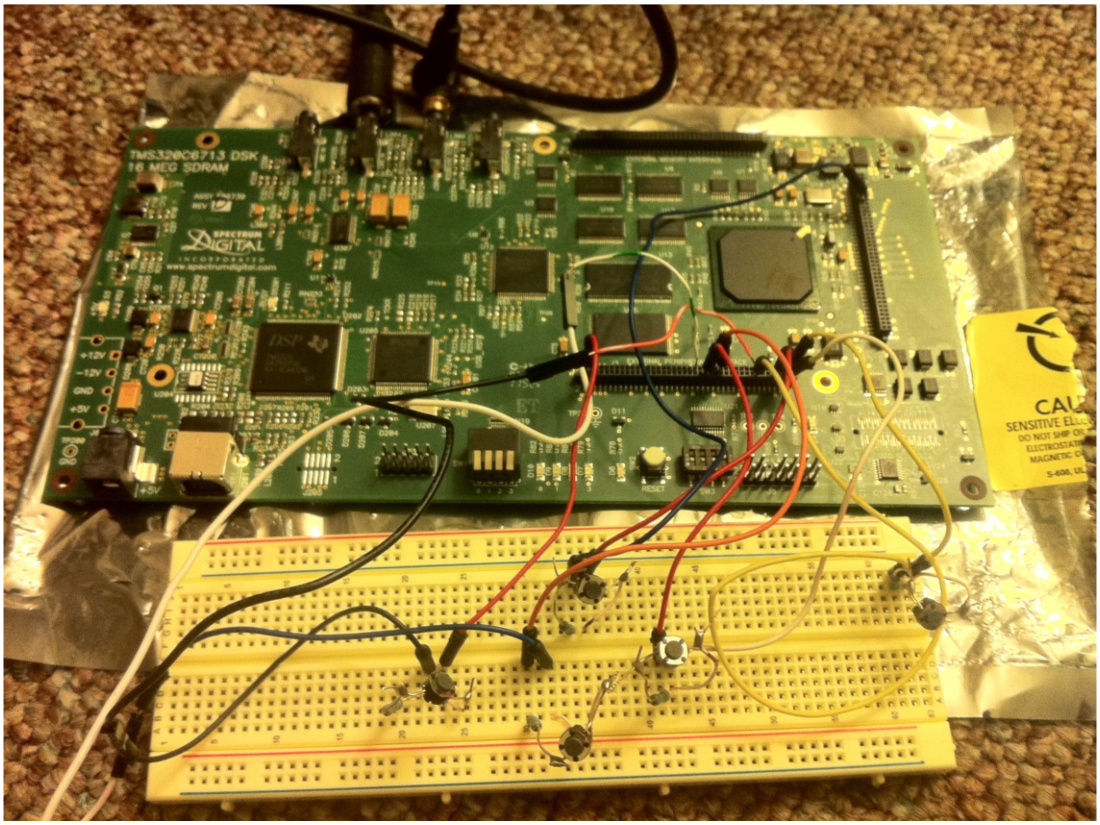


Figure 13: Close up of DSK6713board and setu of user buttons

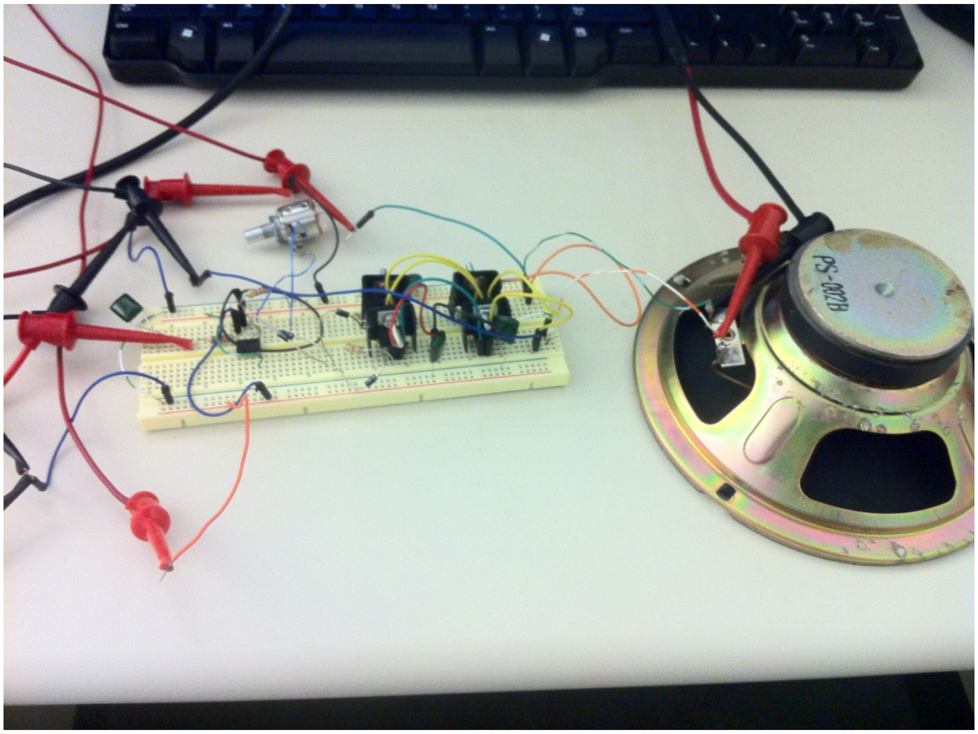


Figure 14: Power Amplifier and Speaker

VII. Integrated System Test Results

The first part of testing done was to check how long the iterative delay buffers were in real time. A pulse signal was run through the echo filter in order to translate the iterations to seconds. Figure 10 in the Section VI shows the tested delayed input of 1500 iterations being equivalent to approximately 60ms. There a rough estimate of 250 iterations = 10ms. This conversion value was tested along with chorus and flanger and the correct ranges were found to be 0-250 iterations for flanger and 500-750 iterations for chorus.

While simple to implement in code running the program in real time with a continuous audio input rather than a single pulse reveal some problems. The main problem was a problem with arithmetic overflow. Since the delays in flanger and chorus are at the most

only 30ms the output signal began to add up and eventually went out of range of the data type. To fix this issue a check for overflow was coded in the flanger and chorus functions. It checked whether or not the output data was greater than the data types max value. If it was greater it would simply set the value equal to the maximum.

A similar issue was found when implementing the reverb effect. Reverb as stated earlier uses a feedback system in which the output is fed back into itself. This automatically created an instance of overflow. Instead of using a overflow detector as was done previously, the delayed output was simply type cast to a lesser data type; in this case an Int16 instead of the output's normal double.

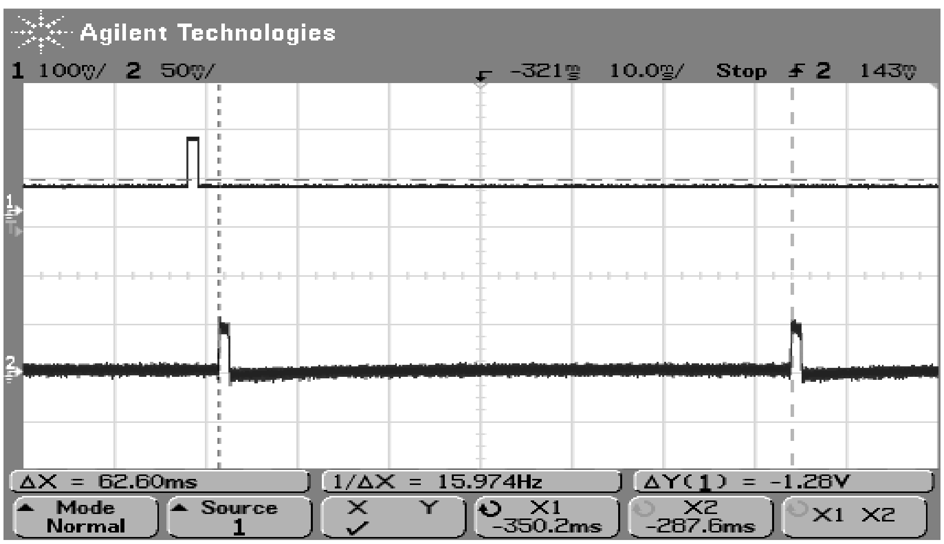


Figure 15: Input and Output waveforms displaying the delay time used for Echo effect

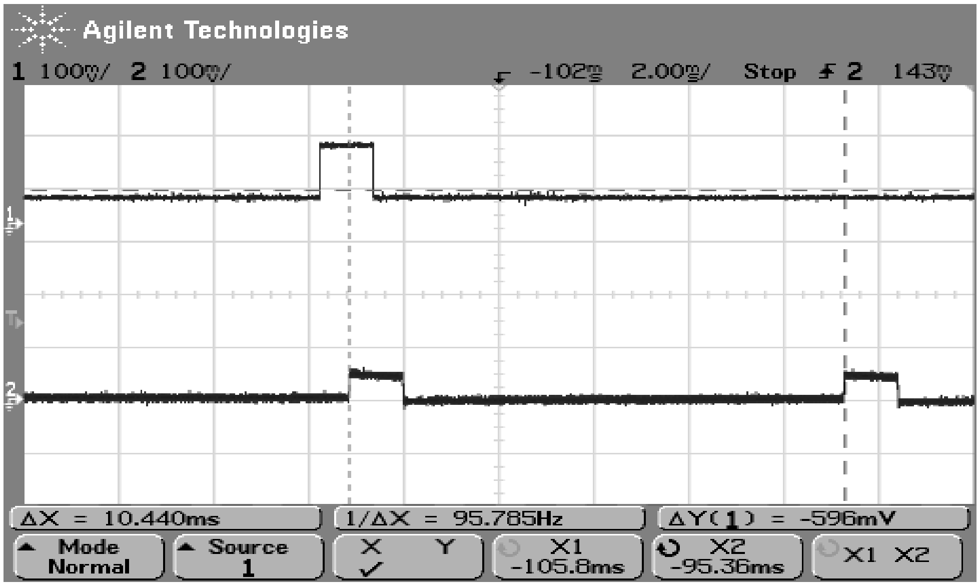


Figure 16: Input and Output waveforms displaying the delay time used for Echo effect

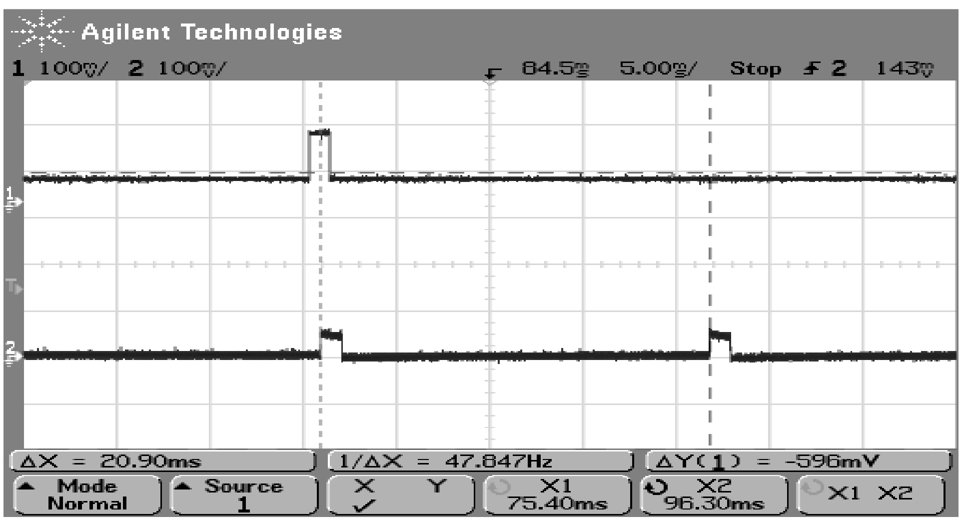


Figure 17: Input and Output waveforms displaying the delay time used for Echo effect

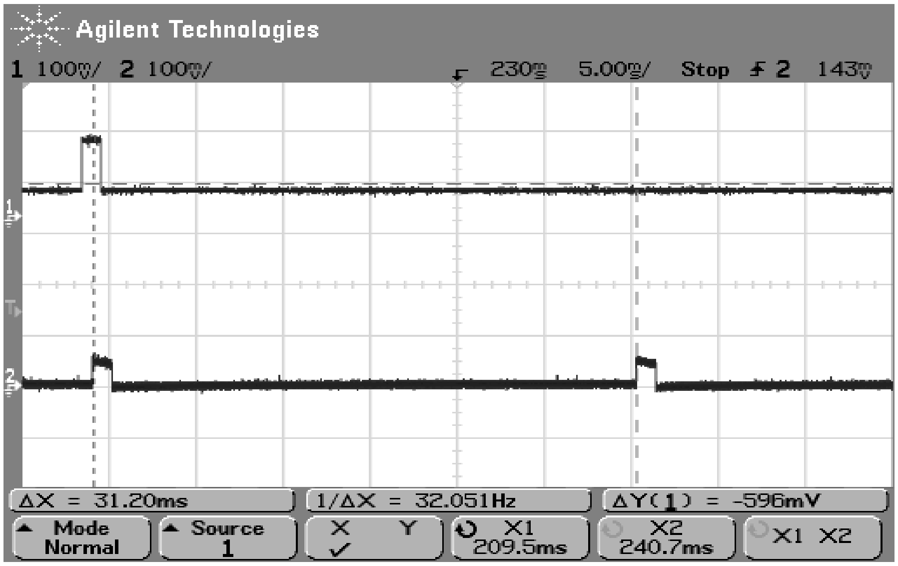


Figure 18: Input and Output waveforms displaying the delay time used for Echo effect

Secondly, each filter for the equalizer was tested individually. An oscilloscope and function generator was used to achieve the frequency response of each filter. The intervals of measured frequency were decreased as the response approached the center or cutoff frequency, in order to get a more accurate representation of the actual cutoff.

Issues arouse when certain filters began to output only noise signals. By viewing the memory map of the processor it was noticed that the intermediate values, w, of the difference equations were given values of Not A Number (NAN), or programming language's version of an undefined number. It was later found that the adjusted gain values weren't being initialized incorrectly in the code. Setting all the gain values to 1dB during initialization fixed this issue and all filters began to respond correctly.

1st Filter	Freq	30	40	50	60	70	80	90	100	110	120	130	140	150	200	300
	Vout	117	120	119	121	117	113	110	108	105	101	96	96	92	80	68
2nd Filter	Freq	120	150	200	220	250	280	300	330	400	500	600	800	1000		
	Vout	113	117	125	127	129	125	125	117	101	88	81	76	75		
3rd Filter	Freq	80	100	200	300	400	500	520	550	580	600	700	800	1000	1200	1500
	Vout	84	88	92	96	113	130	129	127	120	113	105	92	80	77	78
4th Filter	Freq	60	100	200	300	500	700	770	800	830	900	1000	1200	1500		
	Vout	84	85	84	90	100	120	131	130	131	120	100	90	80		
5th Filter	Freq	500	600	800	900	1000	1050	1100	1150	1200	1300	1500				
	Vout	72	76	88	90	96	103	115	121	115	101	80				
6th Filter	Freq	700	800	1000	1200	1300	1400	1450	1500	1550	1700	1900	2000			
	Vout	74	75	76	84	92	103	117	121	113	92	78	74			
7th Filter	Freq	1200	1300	1400	1500	1600	1700	1750	1800	1850	1900	2000	2100	2400		
	Vout	72	76	76	80	84	96	107	117	121	113	95	87	74		
8th Filter	Freq	1500	1800	1900	2000	2100	2200	2300	2400	2500	3000	3500	4000	5000	6000	7000
	Vout	46	70	82	96	101	113	125	127	127	123	111	103	101	94	90

Table 1: Filter Response Data for the individual filters (Center or cutoff frequency in yellow). Vout in mV

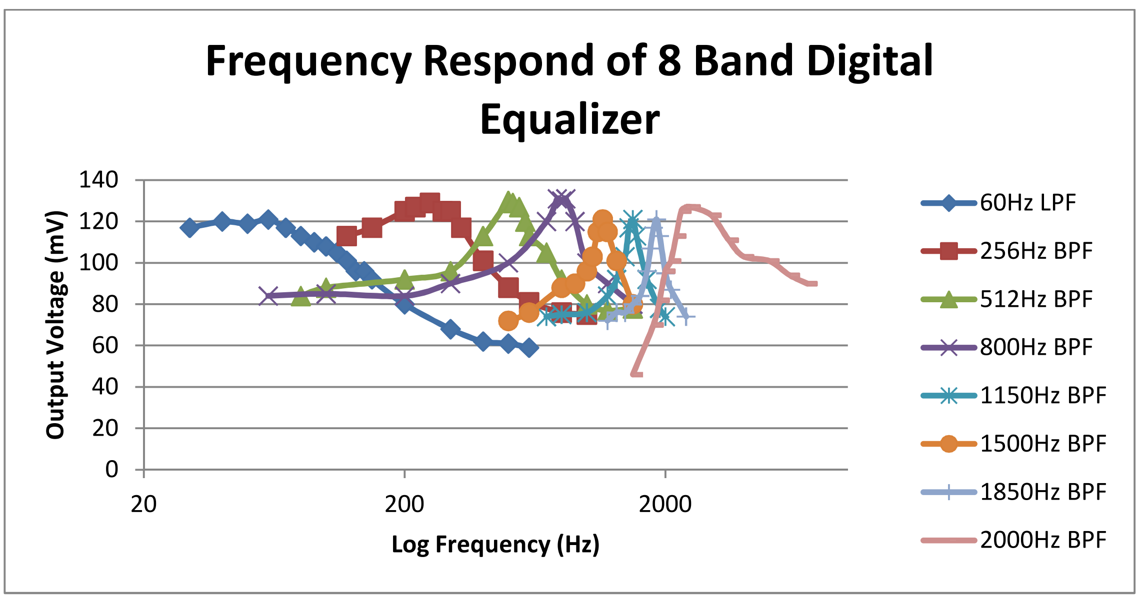


Figure 19: Frequency Response Plot of each individual filter

While all of the delay effects functioned correctly as well as the equalizer and distortion, the auto-wah did not. Depending on the frequency set for the low frequency oscillator it was possible to hear the sweeping of the band pass filter used in the design but not audible "wah" was created. Theoretically this varying frequency band pass filter should have created the wah-wah effect but for whatever reason, it did not.

Testing of the power supply was simply achieved by plugging it into an AC power outlet and observing the various voltages throughout the stages of the dual power supply. Testing and finalizing of the power supply however was not completed. The power supply was tested successfully through to the full bridge rectifier stage as is seen in Figure 16. However when measuring the voltages with smoothing capacitors, one of the capacitors blew. At this point, I did not continue handling the high voltages that were being used and produced by this supply.

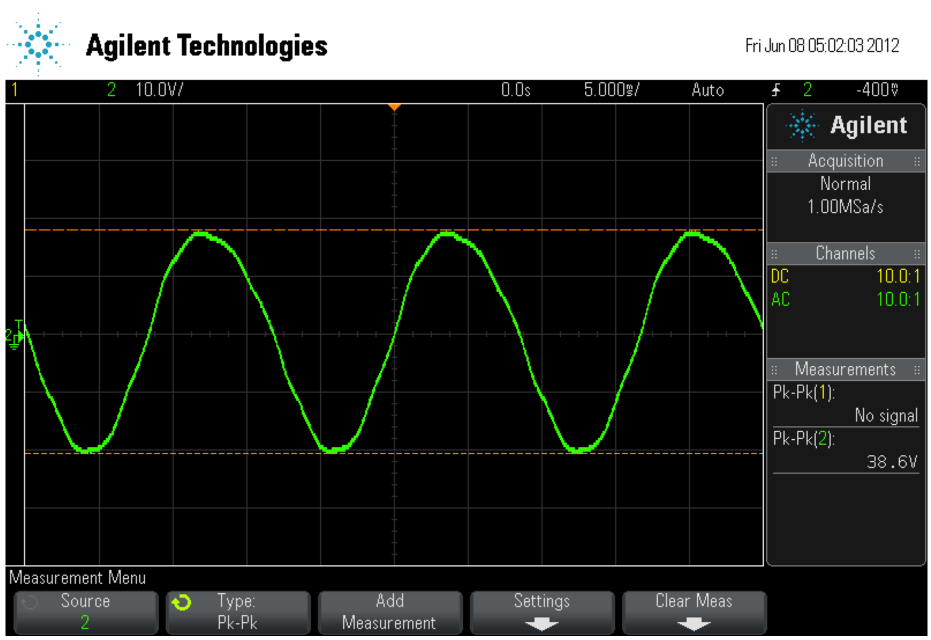


Figure 20: Secondary Winding voltage from transformer

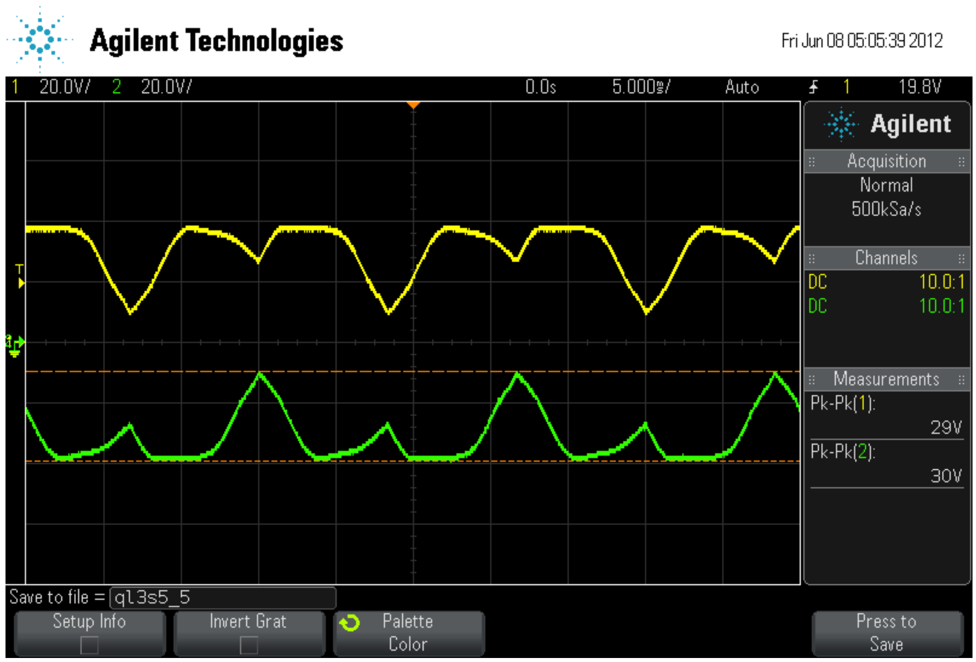


Figure 21: Positive and Negative rails of power supply

The output gain stage was tested and used a 10k linear potentiometer in series with a 4.7k resistor to give it a max gain of 14.7. In Figure 21 we see that our measured gain is approximately 13.33V/V. The potentiometer was measured using a multimeter and the maximum resistance value was 9.6k rather than 10k.

When first testing the power amplification the positive portion of the push pull wasn't outputting and resulted in negative current. A bad NPN transistor in the darlington pair was removed and a new one was replace. This fixed the problem and resulted in the waveform shown in Figure 22. The test setup is the same for both Figure 21 and Figure 22.

Low amplitudes and low frequencies were used for a majority of testing in order to not disturb other students working in the senior project labs.

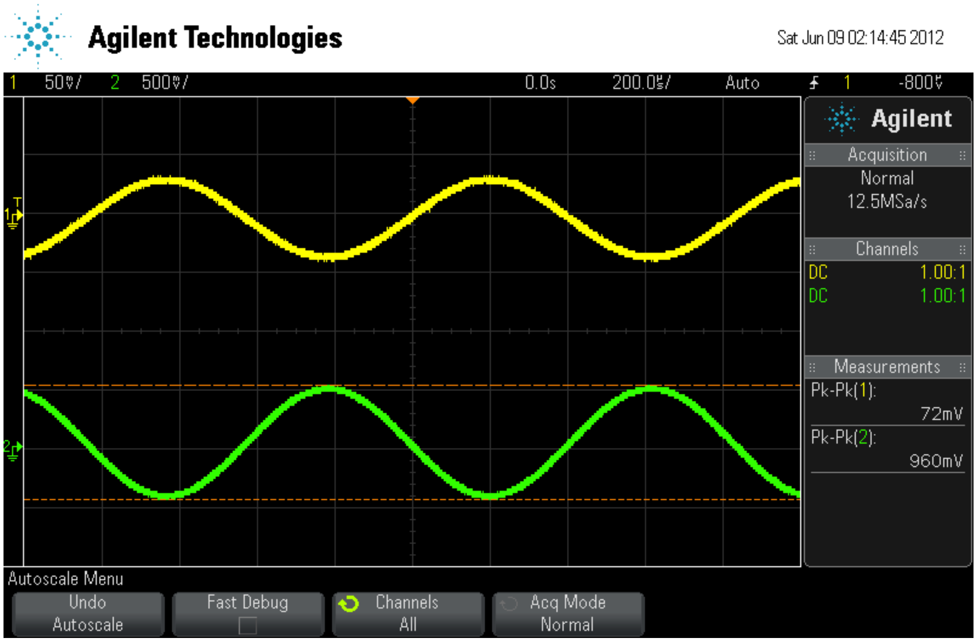


Figure 22: input voltage vs output voltage of op amp volume stage

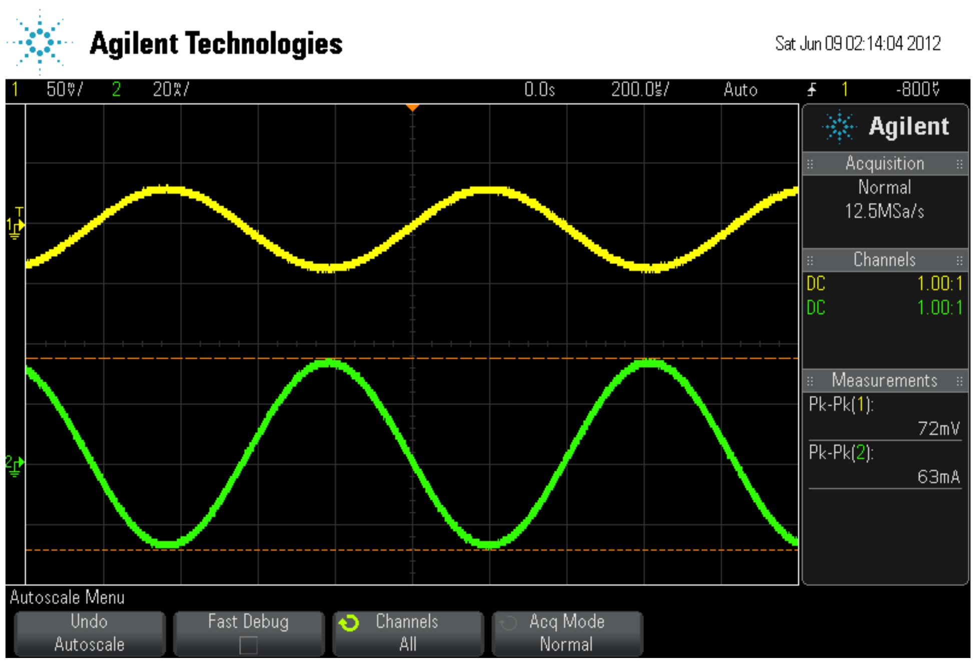


Figure 23: input voltage vs output current of power amplification stage

VIII. Conclusions

The main portions of the project were completed, such as the digital equalizer and effects and amplifier. However they weren't packaged together as intended. During the senior project exhibition a Roland Micro Cube amplifier was used to instead of the designed product. Also the DSK6713 was powered by its own power adapter. The DSK does have through holes where an external supply can be applied with voltages of 3.3V, 5V and 12V but I did not want to risk damaging the board.

Originally an LCD was going to be used to display the response of the equalizer as well as what effect was currently applied. Issues with processing speed caused the display to be excluded from the project. The data sheet gives a normal clock speed for the HD44780 as 270 kHz while the c6713 DSP ran at a speed of 225 MHz. In order for the LCD to process the data correctly large delays were required which interfered with the DSP's real time processing.

The c6713 DSP had more GPIO pins than were wired on the DSK6713. If access to all GPIO pins were available, I could have implemented more control over the guitar effects. One example would be the delay rates of the reverb and echo effects instead of having defined non-user variable rates. This feature of user-controlled sweep rates are incorporated in many of the currently manufactured effects units.

IX. Bibliography

Ambardar, Ashok. *Analog and Digital Signal Processing*. 2nd ed. Pacific Grove: Brooks/Cole Publishing Company, 1999. Print.

Pilkington, Wayne. Cal Poly Electrical Engineering 419: Digital Signal Processing. California Polytechnic State University. San Luis Obispo, CA. Winter 2012. Printed Notes and Class Lecture

APPENDIX

A. Specifications

225MHz DSP – 1800 MIPS and 1350 MFLOPS
8 Band EQ with ±12dB gain
5 effects – distortion, echo, reverb, chorus, flanger
90dB SNR 24 bit ADC
100dB SNR 24 bit DAC

B. Parts List and Costs

~$40 miscellaneous components (op amp, transistors, resistors, connectors etc.)
~$20 120V AC 2A to 12.6V-0-12.6V Transformer
~$30 HD44780 20x4 character display LCD
~$30 speaker and crate from old Peavey backstage practice amp
~$395 Texas Instruments/Spectrum Digital DSK6713 starter kit

Total ~$515

C.Timeline of Major Tasks and Milestones

See attached PERT chart

D. Program Listing

MatLab
Texas Instruements Code Composer v3.1 (DSK6713 edition)

E. MatLab and C code

MatLab design functions – credit Kevin Salvador, David Smith, Eric Spinks

```
function [Ak, Bk, HF, Fd, hn, n] = show_filter_responses_pz(poles, zeros, K ,
fsample, num_of_f_points, num_of_n_points, figure_num);
% Function that takes poles and zeroes and K and outputs Ak, Bk,


% where the arguments are:
% poles = a list of the Z plane locations (complex values) for all the
% POLES of the filter (a row vector)
% zeros = a list of the Z plane locations (complex values) for all the
% ZEROS of the filter (a row vector)
% K = Multiplier constant for the transfer function (which you should
% multiply H(z) by)
% fsample = sampling frequency (samples / second)
% num_of_f_points = the # of points for the freq. response plot
% num_of_n_points = the # of points for the unit sample response plot
% figure_num = number of the 1st figure to use for plots

% and the function returns the following information:
```

```
% Ak = a list of the Ak coefficients of the filter difference equation
% (coefficients of the "y" terms)
% Bk = a list of the Bk coefficients of the filter difference equation
% (coefficients of the "x" terms)
% HF = the DTFT frequency response values (linear scale)
% Fd = digital frequencies that match the freq response values
% hn - has the unit sample response sequence values
% n - has the corresponding sample indices (0 to [num_of_n_points - 1]);

Ak = poly(poles)  %p = poly(r) where r is a vector returns a row vector whose
elements are the coefficients of the polynomial whose roots are the elements
of r.
Bk = K*poly(zeros) % Bk is the polynomial coefficients of zeros times K the
scaling value given
[HF, W] = freqz(Bk, Ak, num_of_f_points); %[H,W] = FREQZ(B,A,N) returns the
N-point complex frequency response
%vector H and the N-point frequency vector W in radians/sample of the
%filter described by Bk and Ak

Fd = W./(2.*pi); % W = 2 Pi Fd; solves for Fd

figure(figure_num) % Creates the first figure
zplane(Bk, Ak); %zplane takes filter data Bk and Ak and plots the poles and
zeros in the z-plane
gridon
title ('Pole/Zero Diagram')

plot_freq_responses(Fd, HF, fsample, figure_num+1); % calls previous function
that plots HF vs. Fd and analog data using fsample as sampling rate

[hn, n] = unit_sample_response(Bk, Ak, num_of_n_points, figure_num+3); %
calculates and plots unit sample response at figure # 3 higher than one in
original input



[peak,max_index]=max(abs(HF))   % returns the idices of the max values of
abs(HF) in vector max_index
max_freq=Fd(max_index) % returns the frequency at which the max occurs

[minimum,min_index]=min(abs(HF)) % returns the idices of the min values of
abs(HF) in vector min_index
min_freq=Fd(min_index) % returns the frequency at which the min occurs

attenuation= 20.*log10(peak)-20.*log10(minimum) % returns the differnce in
the min and max values in dB

dB3=20.*log10(peak)-3  % returns the 3dB bandwidth magnitude in dB

above_index = find(20.*log10(abs(HF)) > dB3); % above_index is a vector with
indicies of all values of HF greater than the 3dB point
first_index = above_index(1); % returns the index of the first value that is
greater than the 3dB point
```

```
last_index= above_index(length(above_index)); %returns index of the last
value that is greater than the 3dB point

ifmax_index<last_index&&min_index>last_index%returns Low Pass if the max
value is at a lower frequency than the last point above the 3dB line and if
the lowest point is a higher frequency the last point above the 3dB line
disp('Filter Type = Low Pass')

elseifmax_index>first_index&&min_index<first_index&&last_index ==
num_of_f_points%returns High Pass if max value frequency is higher than the
first one above 3dB and
% the lowest value is at a lower frequency than the first value
% that is above the 3dB point and if last value above 3dB is the
% last value computed

disp('Filter Type = High Pass')

elseifmax_index>first_index&&max_index<last_index%returns Band pass if the
max value is at frequency between first and last frequency that is above 3dB
disp('Filter Type = Band Pass')
     BW3dB=(Fd(last_index)-Fd(first_index)).*fsample%returns the 3dB
bandwidth in frequency (Hz)

elseifmin_index>first_index&&min_index<last_index%returns Band Stop if min
value is at a frequency between the first and last frequencies above 3dB line
disp('Filter Type = Band Stop')

below_index = find(20.*log10(abs(HF)) < dB3); %finds values that are lower
than 3dB magnitude
first_index_low = below_index(1); % returns first index below 3dB point
last_index_low= below_index(length(below_index)); % returns last index below
3dB point

        BW3dB=(Fd(last_index_low)-Fd(first_index_low)).*fsample%returns 3dB
bandwidth in frequency (Hz)

else% to test if all went wrong
disp('Filter Type = Unknown')
end

///////////////////////////////////////////////////////////////////////////////////////////
% Function that takes a array of frequencies and plots them against an
% array of values

functionplot_freq_responses(Fd,HF, fsample, figure_num)

figure(figure_num)  % creates Figure 1

subplot(2,1,1) % makes Figure 1 have 2 subplots with 2 rows and 1 column

plot(Fd,abs(HF)) % plots frequency vs. absolute values of values in HF_values

gridon% turns the grid on
xlabel('Digital Frequency F (cycles/sample)') % labels the x axis
```

```
ylabel('Magnitude Response')  % labels the y axis
title('Frequency Response of Filter') % creates a title above the plot

subplot(2,1,2) % puts below plot in second row of Figure 1

plot(Fd,angle(HF)./pi)  %  plots  frequency  vs.  angles  of  the  values  in
HF_values

gridon% turns the grid on
xlabel('Digital Frequency F (cycles/sample)') % labels the x axis
ylabel('Phase Response') % labels the y axis


figure(figure_num + 1) % creates a new Figure

subplot(2,1,1) % makes Figure 2 have 2 subplots with 2 rows and 1 column

plot(Fd.*fsample,20.*log10(abs(HF)))   %   creats   a   stem   plot   graphing
frequencies vs. absolute values of values in HF_values with dots
gridon% turns the grid on
xlabel('Analog Frequency f (cycles/s)') % labels the x axis
ylabel('dB Magnitude Response') % labels the y axis
title('Frequency Response of Filter') % creates a title above the plot

subplot(2,1,2) % puts below plot in second row of Figure 1
plot(Fd.*fsample,angle(HF))  %  creats  a  stem  plot  graphing  frequencies  vs.
angle values of values in HF_values with dots
gridon% turns the grid on
xlabel('Analog Frequency f (cycles/s)') % labels the x axis
ylabel('Phase Response') % labels the y axis

///////////////////////////////////////
function   [hn,   n]   =   unit_sample_response(Bk,   Ak,   number_of_samples,
figure_number)

% Function that takes in filter data and creates the unit sample response
% to number_of_samples

% Where the values returned are:
% hn - has the unit sample response sequence values
% n - has the corresponding sample indices (starting at 0);
%
% Where the function parameters (arguments) are:
% Bk = a list of the Bk coefficients of the filter difference equation
% (coefficients of the “x” terms)
% Ak = a list of the Ak coefficients of the filter difference equation
% (coefficients of the “y” terms)
% number_of_samples = # of unit sample response sequence samples to find
% figure_number = # figure for the hn sequence plot

[dn, n] = unit_sample(number_of_samples);   % Calls function from last week
giving the input from this function call
```

```
hn = filter(Bk, Ak, dn); % uses MATLAB's filter function which outputs a filter's response given its characterisitcs and input
% with input set to unit_sample the output is the h[n]
figure(figure_number); % sets first figure to input figure_number
stem(n, hn, '.')  % creates a stem plot of h[n] vs. n with dots on top

gridon% turns grid on
xlabel ('Sample index') % makes the x label on stem plot
ylabel ('hn') % makes the y label on stem plot
title ('Unit Sample Response') % makes the title on stem plot

////////////////////////////////
% creates a function that takes a number of samples and returns the unit
% sample series as dn, and the index as n

function [dn,n]=unit_sample(number_of_samples) % name of the function and output and input format

dn=zeros(1,number_of_samples); %creates a 1 by number_of_samples array with all zeroes
dn(1)=1; % makes first index of array equal to 1
n=linspace(0,number_of_samples-1,number_of_samples); %creates n index starting at 0 and incrementing up to number_of_samples - 1
return

///////////////////////////////
% Uses show_filter_responses_pz

poles=[.95.*exp(i.*2.*pi.*1000./16000) .95.*exp(-i.*2.*pi.*1000./16000)];
zeros=[.9.*exp(i.*2.*pi.*1000./16000) .9.*exp(-i.*2.*pi.*1000./16000)];
K=1;
fsample=16000;
num_of_f_points=500;
num_of_n_points=50;
figure_num=1;


[Ak, Bk, HF, Fd, hn, n] = show_filter_responses_pz(poles, zeros, K , fsample, num_of_f_points, num_of_n_points, figure_num);
```

C-programming code

```
////////////////////////////////////
#include "tonecfg.h"
#include <dsk6713.h>
#include "dsk6713_aic23.h"
#include <csl_gpio.h>
#include <csl_gpiohal.h>
#include <csl.h>


#include "cpee_student_diffeq.h"
#include "dsk6713_dip.h"
#include "dsk6713_led.h"
#include <math.h>

#define maxdouble 1.7976931348623158e+308
#define Uint16 unsigned short

/* Codec configuration settings */
DSK6713_AIC23_Config config = {
   0x0017, // 0 DSK6713_AIC23_LEFTINVOL  Left line input channel volume
   0x0017, // 1 DSK6713_AIC23_RIGHTINVOL Right line input channel volume
   0x01F9, // 2 DSK6713_AIC23_LEFTHPVOL  Left channel headphone volume
   0x01F9, // 3 DSK6713_AIC23_RIGHTHPVOL Right channel headphone volume
   0x0011, // 4 DSK6713_AIC23_ANAPATH    Analog audio path control
   0x0000, // 5 DSK6713_AIC23_DIGPATH    Digital audio path control
   0x0000, // 6 DSK6713_AIC23_POWERDOWN  Power down control
   0x0043, // 7 DSK6713_AIC23_DIGIF      Digital audio interface format
   0x0001, // 8 DSK6713_AIC23_SAMPLERATE Sample rate control
0x0001  // 9 DSK6713_AIC23_DIGACT     Digital interface activation
};

//GPIO configuration settings
GPIO_Configgpio_config = {
        0x00000000, // gpgc = bypass interrupts and set gpio
   0x0000FFFF, // gpen = enable gpio pins 0-15
   0x0000FF07, // gdir = set gpio directions, 0=input, 1=output
   0x00000000, // gpval = save logic level of pins
   0x00000000, // gphm all interrupts disabled for io pins
   0x00000000, // gplm all interrupts to cpu or edma disabled
0x00000000  // gppol -- default state
};

long double output=0;
long double input;
long double delaylength = 0, delaylength2 = 0, delaylength3 = 0, delaylength4 = 0;
```

```
long double delay[10000], delay2[10000], delay3[10000], delay4[10000];

//random number delays for flanger effect, 250 ~10ms
doubleflangedelay[100]={
157,  132,  210,  202,  207,  182,  113,  200,  152,  206,
133,  227,  208,  180,  194,  175,  158,  168,  222,  205,
152,  164,  181,  121,  124,  125,  103,  220,  102,  106,
119,  234,  140,  230,  157,  112,  129,  184,  161,  237,
105,  191,  231,  179,  103,  136,  175,  190,  166,  197,
249,  196,  102,  140,  199,  177,  233,  139,  173,  170,
165,  121,  175,  106,  195,  222,  245,  213,  166,  227,
119,  150,  193,  220,  168,  119,  150,  205,  227,  173,
153,  114,  228,  234,  145,  205,  141,  135,  108,  241,
185,  107,  227,  135,  123,  109,  147,  151,  175,  129,
};

//random number delays for chorus effect, 250 ~10ms
doublechorusdelay[100]={
685,  797,  781,  450,  650,  752,  630,  691,  546,  686,
431,  555,  507,  568,  550,  495,  799,  771,  455,  771,
594,  495,  655,  467,  425,  770,  565,  562,  578,  722,
484,  637,  745,  728,  621,  657,  580,  589,  577,  489,
481,  459,  664,  527,  413,  684,  799,  426,  608,  704,
671,  436,  428,  477,  498,  445,  787,  527,  423,  787,
417,  671,  630,  732,  459,  636,  428,  731,  411,  445,
743,  499,  547,  676,  709,  644,  440,  504,  448,  792,
629,  562,  445,  751,  795,  616,  650,  741,  711,  742,
511,  589,  439,  784,  709,  617,  592,  760,  546,  773
};

//center frequencies of BPF for auto-wah
doublewahfreqsweep[50]={
500,  520,  540,  560,  580,  600,  620,  640,  660,  680,
700,  720,  740,  760,  780,  800,  820,  840,  860,  880,
900,  920,  940,  960,  980,  1000, 1020, 1040, 1060, 1080,
2000, 2020, 2040, 2060, 2080, 3000, 3020, 3040, 3060, 3080, 4000, 4020, 4040, 4060, 4080,
5000
};

//gain values in V/V ranging for -12db to 12db
double gain[25]={.25, .28, .32, .36, .4, .45, .5, .56,
.63, .707, .79, .89, 1, 1.12, 1.26, 1.41, 1.58, 1.79, 1.99,
 2.24, 2.51, 2.82, 3.16, 3.55, 3.98};

//array that holds gain for the 8 EQ bands
doubleBPfreq[8] = {1,1,1,1,1,1,1,1};
```

```
//input samples and output samples for left and right channels
Uint32 left_input,right_input,left_output,right_output;
Int16 afterEQ, distortint, predistort;
intbit_vals[8]={0};
intctrlbit_vals[3]={0};
inti = 0, ii=0, iii=0, iv=0, d=0, d2=0, freqcount = 0, freqcount2 = 1250, k = 13, EQ_freq =0,
control=0;
int state =0, hold = 0,hold1 = 0,hold2 = 0,hold3 = 0,hold4 = 0,hold5 = 0, curr_band = 0,
ka=13,kb=13,kc=13,kd=13,ke=13,kf=13,kg=13;
double yn1=0,yn2=0,xn1=0,xn2=0;
int x, y, on=1, c=128; // on/off variable and initial threshold c
unsigned char pattern1[8] = {0, 0, 0, 0, 0, 0, 0x1f, 0x1f}, pattern2[8] = {0, 0, 0, 0, 0, 0x1f, 0x1f,
0x1f}, pattern3[8] = {0, 0, 0, 0, 0x1f, 0x1f, 0x1f, 0x1f};
unsigned char pattern4[8] = {0, 0, 0, 0x1f, 0x1f, 0x1f, 0x1f, 0x1f}, pattern5[8] = {0, 0, 0x1f,
0x1f, 0x1f, 0x1f, 0x1f, 0x1f}, pattern6[8] = {0, 0x1f, 0x1f, 0x1f, 0x1f, 0x1f, 0x1f, 0x1f};
Int16 echo(Int16 channel,Uint32 delaylength);
Int16 flanger(Int16 channel);
Int16 chorus(Int16 channel);
Int16 reverb(Int16 channel,Uint32 delaylength);
Int16 wah_wah(Int16 input);
Int16 EQ(Int16 input, int k);
voidwriteData (intdata_array[], GPIO_Handleg_han);
voidwriteCtrl (intdata_array[], GPIO_Handleg_han);
voidstringToLCD(char str[], int line);
voidcharToLCD(char myChar);
voidclearLCD(GPIO_Handleg_han);
voidinitializeLCD(GPIO_Handleg_han);
intfreqsweep();
int sweep();
intdistortionthresh(int);
Int16 distortion(Int16 channel);
intfixdistortion(int x);
voidLCD_build(unsigned char location, unsigned char *ptr);

main()
{DSK6713_AIC23_CodecHandle hCodec;
GPIO_Handlegpio_handle;
int check;

        CSL_init();
  DSK6713_init();

  DSK6713_DIP_init();
  DSK6713_LED_init();
```

```
hCodec = DSK6713_AIC23_openCodec(0, &config);

gpio_handle = GPIO_open( GPIO_DEV0, GPIO_OPEN_RESET );
GPIO_config(gpio_handle,&gpio_config);

   DSK6713_AIC23_setFreq(hCodec,DSK6713_AIC23_FREQ_16KHZ);

   //initializeLCD(gpio_handle);
  /* for (EQ_freq = 0; EQ_freq<8; EQ_freq++)
          BPfreq[EQ_freq] = gain[k];*/
                  student_diffeq_initialize(BPfreq);



IRQ_globalDisable();




while(1)
   {
                  while (!DSK6713_AIC23_read(hCodec, &left_input));

                  while (!DSK6713_AIC23_read(hCodec, &right_input));

          while (!DSK6713_AIC23_write(hCodec, left_output));
                  while (!DSK6713_AIC23_write(hCodec, right_output));



                  //moves selected band from left to right
                  if(GPIO_pinRead (gpio_handle,GPIO_PIN5)==0){
                          DSK6713_waitusec(20000);
                          if(hold1==0){
                                  curr_band++;
                                  hold1 = 1;
                                  if (curr_band>7) curr_band = 7;
                          }
                  }
                  else hold1=0;

                  //moves selected band from right to left
                  if (GPIO_pinRead (gpio_handle,GPIO_PIN6)==0){
                          DSK6713_waitusec(20000);
                          if(hold4==0){
                                  curr_band--;
```

```
                    hold4 = 1;
                    if (curr_band<0) curr_band = 0;
          }
}
else hold4=0;

 //increase gain of current selected EQ band
if(GPIO_pinRead (gpio_handle,GPIO_PIN3)==0){
          DSK6713_waitusec(20000);
          if(hold2==0){
                    if (curr_band == 0){
                              k++;
                              if (k>24) k = 24;
                              BPfreq[0] = gain[k];
                    }
                    if (curr_band == 1){
                              ka++;
                              if (ka>24) ka = 24;
                              BPfreq[1] = gain[ka];
                    }
                    if (curr_band == 2){
                              kb++;
                              if (kb>24) kb = 24;
                              BPfreq[2] = gain[kb];
                    }
                    if (curr_band == 3){
                              kc++;
                              if (kc>24) kc = 24;
                              BPfreq[3] = gain[kc];
                    }
                    if (curr_band == 4){
                              kd++;
                              if (kd>24) kd = 24;
                              BPfreq[4] = gain[kd];
                    }
                    if (curr_band == 5){
                              ke++;
                              if (ke>24) ke = 24;
                              BPfreq[5] = gain[ke];
                    }
                    if (curr_band == 6){
                              kf++;
                              if (kf>24) kf = 24;
                              BPfreq[6] = gain[kf];
                    }
                    if (curr_band == 7){
```

```
                kg++;
                if (kg>24) kg = 24;
                BPfreq[7] = gain[kg];
            }
            //student_diffeq_initialize(BPfreq);
            hold2=1;
        }
}
else hold2=0;

//decreases gain of current selected EQ band
if (GPIO_pinRead (gpio_handle,GPIO_PIN4)==0){
        DSK6713_waitusec(20000);
        if(hold3==0){
            if (curr_band == 0){
                k--;
                if (k<0) k = 0;
                BPfreq[0] = gain[k];
            }
            if (curr_band == 1){
                ka--;
                if (ka<0) ka = 0;
                BPfreq[1] = gain[ka];
            }
            if (curr_band == 2){
                kb--;
                if (kb<0) kb = 0;
                BPfreq[2] = gain[kb];
            }
            if (curr_band == 3){
                kc--;
                if (kc<0) kc = 0;
                BPfreq[3] = gain[kc];
            }
            if (curr_band == 4){
                kd--;
                if (kd<0) kd = 0;
                BPfreq[4] = gain[kd];
            }
            if (curr_band == 5){
                ke--;
                if (ke<0) ke = 0;
                BPfreq[5] = gain[ke];
            }
            if (curr_band == 6){
                kf--;
```

```
					if (kf<0) kf = 0;
					BPfreq[6] = gain[kf];
				}
				if (curr_band == 7){
					kg--;
					if (kg<0) kg = 0;
					BPfreq[7] = gain[kg];
				}
				//student_diffeq_initialize(BPfreq);
				hold3=1;
			}
		}
		else hold3=0;

		//cycle through the different effects
	if (GPIO_pinRead (gpio_handle,GPIO_PIN7)==0){
			DSK6713_waitusec(20000);//prevent debouncing
			if(hold==0){//check if button is being held, only increments once per press
				state++;
				hold = 1;
			}
		}
		else hold=0;


		if (state==0){//echo effect state
			afterEQ = EQ(left_input, k);
			if (DSK6713_DIP_get(0)==0){
			predistort = echo(afterEQ, 2500);
			distortint = distortion(predistort);
			left_output = distortint;}
		elseleft_output = echo(afterEQ, 2500);
			DSK6713_LED_on(0);
			DSK6713_LED_off(1);
			DSK6713_LED_off(2);
			DSK6713_LED_off(3);}
		else if (state==1){//chorus effect state
			afterEQ = EQ(left_input, k);
			if (DSK6713_DIP_get(0)==0){
			predistort = chorus(afterEQ);
			distortint = distortion(predistort);
			left_output = distortint;}
			elseleft_output = chorus(afterEQ);
			DSK6713_LED_off(0);
			DSK6713_LED_on(1);
			DSK6713_LED_off(2);
```

```
				DSK6713_LED_off(3);}
			else if (state==2){//flanger effect state
				afterEQ = EQ(left_input, k);
				if (DSK6713_DIP_get(0)==0){
				predistort = flanger(afterEQ);
				distortint = distortion(predistort);
				left_output = distortint;}
			elseleft_output = flanger(afterEQ);
				DSK6713_LED_off(0);
				DSK6713_LED_off(1);
				DSK6713_LED_on(2);
				DSK6713_LED_off(3);}
		else if (state==3){//reverb effect state
			afterEQ = EQ(left_input, k);
				if (DSK6713_DIP_get(0)==0){
				predistort = reverb(afterEQ, 2500);
				distortint = distortion(predistort);
				left_output = distortint;}
			elseleft_output = reverb(afterEQ, 2500);
				DSK6713_LED_on(0);
				DSK6713_LED_on(1);
				DSK6713_LED_off(2);
				DSK6713_LED_off(3);}
		else if (state==4){
			afterEQ = EQ(left_input, k);
				if (DSK6713_DIP_get(0)==0){
				predistort = wah_wah(afterEQ);
				distortint = distortion(predistort);
				left_output = distortint;}
			elseleft_output = wah_wah(left_input);
				DSK6713_LED_off(0);
				DSK6713_LED_off(1);
				DSK6713_LED_off(2);
				DSK6713_LED_on(3);}
			else if (state ==5) state =0;

			//since guitar single is mono both output signals are the same in case of stereo out
			right_output=left_output;
		};

}

Int16 echo(Int16 channel,Uint32 delaylength)
{

		output = 0;
```

```
        input = channel;

        output = input + .3*delay[i] + .5*delay2[ii] + .7*delay3[iii]+ .9*delay4[iv];
        delay[i] = input;
        delay2[ii] = input;
        delay3[iii] = input;
        delay4[iv] = input;
        i++;
        ii++;
        iii++;
        iv++;
        if (i>=delaylength) i = 0;
        if (ii>=delaylength*.9) ii=0;
        if (iii>=delaylength*.7) iii=0;
        if (iv>=delaylength*.5) iv=0;

        return (Int16) output;
}

Int16 reverb(Int16 channel,Uint32 delaylength)
{
        output = 0;

        input = channel;
        output = input + .7*delay[i];// + delay2[ii] + delay3[iii] + delay4[iv];
        delay[i] = (Int16) output;
        delay2[ii] = output;
        delay3[iii] = output;
        delay4[iv] = output;
        i++;
        ii++;
        iii++;
        iv++;
        if (i>=delaylength) i = 0;
        if (ii>=delaylength*.9) ii=0;
        if (iii>=delaylength*.5) iii=0;
        if (iv>=delaylength*.2) iv=0;

        return(Int16) output;
}

Int16 flanger(Int16 channel)
{
        input=channel;
        d2++;
        if (d2>=1250){
```

```
			d++;
			d2=0;
		}
		if(d>=100) d=0;
		output = (Int16)(input) + (Int16)(.25*delay[i]);
		delay[i] = input;
		i++;
		if (i>=flangedelay[d]) i = 0;
		if (output>maxdouble) output = .7*maxdouble;
		if (output<-maxdouble) output = -.7*maxdouble;
		return (Int16) (output);
}

Int16 chorus(Int16 channel)
{
		input = channel;
		d2++;
		if (d2>=1250){
			d++;
			d2=0;
			}
		if(d>=100) d=0;
		output = input + (Int16) (.5*delay[i]);
		delay[i] = input;
		i++;
		if (i>=chorusdelay[d]) i = 0;
		//minimize distortion from addition of delayed signals
		if (output>maxdouble) output = .7*maxdouble;
		if (output<-maxdouble) output = -.7*maxdouble;
		return (Int16) output;
		}

Int16 wah_wah(Int16 input)
{
		Int16 freq;
		double y;

		freq = freqsweep();


		y = .7*input - 1.164*cos(2*PI*(freq/16000))*xn1 + .567*xn2 + 
1.9*cos(2*PI*(freq/16000))*yn1 - .9025*yn2;
		xn2 = xn1;
		xn1 = input;
		yn2 = yn1;
		yn1 = y;
```

```
return (Int16) y;
}

//sweeps center frequency of band pass filter for auto-wah
intfreqsweep(){
Int16 freq;
	if(!--freqcount2){
		if (!control){
				freq = wahfreqsweep[freqcount++];
				if (freqcount>50) control = 1;
		}
		else if (control){
				freq = wahfreqsweep[freqcount--];
				if (freqcount==0) control = 0;
		}
		freqcount2 = 1250;
	}


	return freq;
}

Int16 EQ(Int16 input,int k){
	output = student_diffeq_evaluate(input);
	return output;
}

//initializes the LCD screen
voidinitializeLCD(GPIO_Handleg_han){
	DSK6713_waitusec(2000000);
	ctrlbit_vals[0] = 0;
	ctrlbit_vals[1] = 0;
	ctrlbit_vals[2] = 0;
	writeCtrl(ctrlbit_vals, g_han);
	ctrlbit_vals[0] = 0;
	ctrlbit_vals[1] = 0;
	ctrlbit_vals[2] = 1;
	writeCtrl(ctrlbit_vals, g_han);
	bit_vals[0] = 0;
	bit_vals[1] = 0;
	bit_vals[2] = 0;
	bit_vals[3] = 1;
	bit_vals[4] = 1;
	bit_vals[5] = 1;
```

```
            bit_vals[6] = 0;
            bit_vals[7] = 0;
            writeData(bit_vals, g_han);
            ctrlbit_vals[0] = 0;
            ctrlbit_vals[1] = 0;
            ctrlbit_vals[2] = 0;
            writeCtrl(ctrlbit_vals, g_han);
            DSK6713_waitusec(500000);

ctrlbit_vals[0] = 0;
            ctrlbit_vals[1] = 0;
            ctrlbit_vals[2] = 1;
            writeCtrl(ctrlbit_vals, g_han);
            bit_vals[0] = 1;
            bit_vals[1] = 1;
            bit_vals[2] = 1;
            bit_vals[3] = 1;
            bit_vals[4] = 0;
            bit_vals[5] = 0;
            bit_vals[6] = 0;
            bit_vals[7] = 0;
            writeData(bit_vals, g_han);
            ctrlbit_vals[0] = 0;
            ctrlbit_vals[1] = 0;
            ctrlbit_vals[2] = 0;
            writeCtrl(ctrlbit_vals, g_han);
            DSK6713_waitusec(500000);

            clearLCD(g_han);
}



intdistortionthresh(int x){
            float y, xc = x/c;
            y = x * (1 - b * xc * xc);
            if (x>c) y = a*c; // force the threshold values
            if (x<-c) y = -a*c;

            return ((int) y);
}

Int16 distortion(Int16 channel){
            output = (Int16) distortionthresh((int) channel);
            return ((Int16) (4*output));
}
```

```
//write array to data pins of LCD
voidwriteData (intdata_array[], GPIO_Handleg_han){
        GPIO_pinWrite (g_han,GPIO_PIN8, data_array[0]);
        GPIO_pinWrite (g_han,GPIO_PIN9,data_array[1]);
        GPIO_pinWrite (g_han,GPIO_PIN10,data_array[2]);
        GPIO_pinWrite (g_han,GPIO_PIN11,data_array[3]);
        GPIO_pinWrite (g_han,GPIO_PIN12,data_array[4]);
        GPIO_pinWrite (g_han,GPIO_PIN13,data_array[5]);
        GPIO_pinWrite (g_han,GPIO_PIN14,data_array[6]);
        GPIO_pinWrite (g_han,GPIO_PIN15,data_array[7]);
}

//write array to control pins of LCD
voidwriteCtrl (intdata_array[], GPIO_Handleg_han){
        GPIO_pinWrite (g_han,GPIO_PIN1,data_array[0]);
        GPIO_pinWrite (g_han,GPIO_PIN0,data_array[1]);
        GPIO_pinWrite (g_han,GPIO_PIN2,data_array[2]);
}

//clears LCD screen
voidclearLCD(GPIO_Handleg_han){
        DSK6713_waitusec(2000);
ctrlbit_vals[0] = 0;
        ctrlbit_vals[1] = 0;
        ctrlbit_vals[2] = 0;
        writeCtrl(ctrlbit_vals, g_han);//PORTJ = 0x00;

        ctrlbit_vals[0] = 0;
        ctrlbit_vals[1] = 0;
        ctrlbit_vals[2] = 1;
        writeCtrl(ctrlbit_vals, g_han);//PORTJ = 0x04;

        bit_vals[0] = 1;
        bit_vals[1] = 0;
        bit_vals[2] = 0;
        bit_vals[3] = 0;
        bit_vals[4] = 0;
        bit_vals[5] = 0;
        bit_vals[6] = 0;
        bit_vals[7] = 0;
        writeData(bit_vals, g_han);//PORTB = 0x01;
        ctrlbit_vals[0] = 0;
        ctrlbit_vals[1] = 0;
        ctrlbit_vals[2] = 0;
        writeCtrl(ctrlbit_vals, g_han);//PORTJ = 0x00;
```

```
}

//Input:
//     location: location where you want to store
//               0,1,2,....7
//     ptr: Pointer to pattern data
//
//Usage:
//     pattern[8]={0x04,0x0E,0x0E,0x0E,0x1F,0x00,0x04,0x00};
//     LCD_build(1,pattern);
//
//LCD Ports are same as discussed in previous sections

/*void LCD_build(unsigned char location, unsigned char *ptr){
unsigned char i;
if(location<8){
LCD_command(0x40+(location*8));
for(i=0;i<8;i++)
LCD_senddata(ptr[ i ]);
    }
}*/

/*Kevin Inong Salvador
Senior Project
Advisor: Dr. Wayne Pilkington
This portion of code was derived from Dr. DePeiro's template codes used in EE 419 at Cal Poly
*/

#include "dsk6713.h"
#include "cpee_student_diffeq.h"
#include "math.h"

//coefficients for 2nd order LPF fc = 128Hz
doubleAk[3]={  1.0000,   -1.8996,    0.9025}, Bk[3]={    0.2770,   -0.4985,    0.2244};
double w1a=0,w2a=0,w11a=0,w22a=0;

//coefficients for 2nd order BPF fc = 256Hz
double Ak2[3]={ 1.0000 ,  -1.8904 ,   0.9025}, Bk2[3]={0.4695,   -0.8408,    0.3803};
double w1b=0,w2b=0,w11b=0,w22b=0;

//coefficients for 2nd order BPF fc = 512Hz
double Ak3[3]={    1.0000 ,  -1.8617,    0.9025}, Bk3[3]={0.5000,   -0.8819,    0.4050};
double w1c=0,w2c=0,w11c=0,w22c=0;

//coefficients for 2nd order BPF fc = 512Hz
```

```
double Ak4[3]= {     1.0000,   -1.8070,    0.9025}, Bk4[3] = {0.5081,   -0.8699,    0.4116};
double w1d=0,w2d=0,w11d=0,w22d=0;

//coefficients for 2nd order BPF fc = 512Hz
double Ak5[3]= {1.0000,   -1.7095,    0.9025}, Bk5[3] = {0.5081,   -0.8229,    0.4116};
double w1e=0,w2e=0,w11e=0,w22e=0;

//coefficients for 2nd order BPF fc = 512Hz
double Ak6[3]= {1.0000,   -1.5798,    0.9025}, Bk6[3] = {0.5081,   -0.7605,    0.411};
double w1f=0,w2f=0,w11f=0,w22f=0;

//coefficients for 2nd order BPF fc = 512Hz
double Ak7[3]= {     1.0000,   -1.4203,    0.9025}, Bk7[3] = {0.5081,   -0.6837,    0.4116};
double w1g=0,w2g=0,w11g=0,w22g=0;

//coefficients for 2nd order BPF fc = 512Hz
double Ak8[3]= {1.0000,   -1.0150,    0.6400}, Bk8[3] = {0.5081,   -0.9229,    0.5081};
double w1h=0,w2h=0,w11h=0,w22h=0;

//coefficients for 2nd order BPF fc = 512Hz
double Ak9[3]= {1.0000,   -0.0000,    0.6400}, Bk9[3] = {0.5081,    0.3517,    0.5081};
double w1i=0,w2i=0,w11i=0,w22i=0;

//gain multipliers
double Bk1_gain[3]={0}, Bk2_gain[3]={0}, Bk3_gain[3]={0}, Bk4_gain[3]={0},
Bk5_gain[3]={0}, Bk6_gain[3]={0}, Bk7_gain[3]={0}, Bk8_gain[3]={0};
int k1 = 0;
doubleyouta=0,youtb=0, youtc=0, youtd=0, youte=0, youtf=0, youtg=0, youth=0, yout=0;

////////////////////////////////////////////////////////////////
voidstudent_diffeq_initialize(double Karray[])
{
	//set gain values
	for (k1=0;k1<3;k1++){
		Bk1_gain[k1] = Karray[0]*Bk[k1];
		Bk2_gain[k1] = Karray[1]*Bk2[k1];
		Bk3_gain[k1] = Karray[2]*Bk3[k1];
		Bk4_gain[k1] = Karray[3]*Bk4[k1];
		Bk5_gain[k1] = Karray[4]*Bk5[k1];
		Bk6_gain[k1] = Karray[5]*Bk6[k1];
		Bk7_gain[k1] = Karray[6]*Bk7[k1];
		Bk8_gain[k1] = Karray[7]*Bk8[k1];
	}

	return;
}
```

```
Int16 student_diffeq_evaluate(Int16 xin)
{

        //direct form II transpose diffeq for 2nd order LPF fc = 128Hz
        youta = (Bk1_gain[0]*xin) + w11a;
        w1a = -Ak[1]*youta+Bk1_gain[1]*xin+w22a;
        w2a = -Ak[2]*youta+Bk1_gain[2]*xin;
        w22a = w2a;
        w11a = w1a;

  //direct form II transpose diffeq for 2nd order BPF fc = 128Hz
        youtb = (Bk2_gain[0]*xin)+w11b;
        w1b = -Ak2[1]*youtb+Bk2_gain[1]*xin+w22b;
        w2b = -Ak2[2]*youtb+Bk2_gain[2]*xin;
        w22b = w2b;
        w11b = w1b;

  //direct form II transpose diffeq for 2nd order BPF fc = 128Hz
        youtc = Bk3_gain[0]*xin+w11c;
        w1c = -Ak3[1]*youtc+Bk3_gain[1]*xin+w22c;
        w2c = -Ak3[2]*youtc+Bk3_gain[2]*xin;
        w22c = w2c;
        w11c = w1c;

        //direct form II transpose diffeq for 2nd order BPF fc = 128Hz
        youtd = Bk4_gain[0]*xin+w11d;
        w1d = -Ak4[1]*youtd+Bk4_gain[1]*xin+w22d;
        w2d = -Ak4[2]*youtd+Bk4_gain[2]*xin;
        w22d = w2d;
        w11d = w1d;

  //direct form II transpose diffeq for 2nd order BPF fc = 128Hz
        youte = Bk5_gain[0]*xin+w11e;
        w1e = -Ak5[1]*youte+Bk5_gain[1]*xin+w22e;
        w2e = -Ak5[2]*youte+Bk5_gain[2]*xin;
        w22e = w2e;
        w11e = w1e;

        //direct form II transpose diffeq for 2nd order BPF fc = 128Hz
        youtf = Bk6_gain[0]*xin+w11f;
        w1f = -Ak6[1]*youtf+Bk6_gain[1]*xin+w22f;
        w2f = -Ak6[2]*youtf+Bk6_gain[2]*xin;
        w22f = w2f;
        w11f = w1f;
```

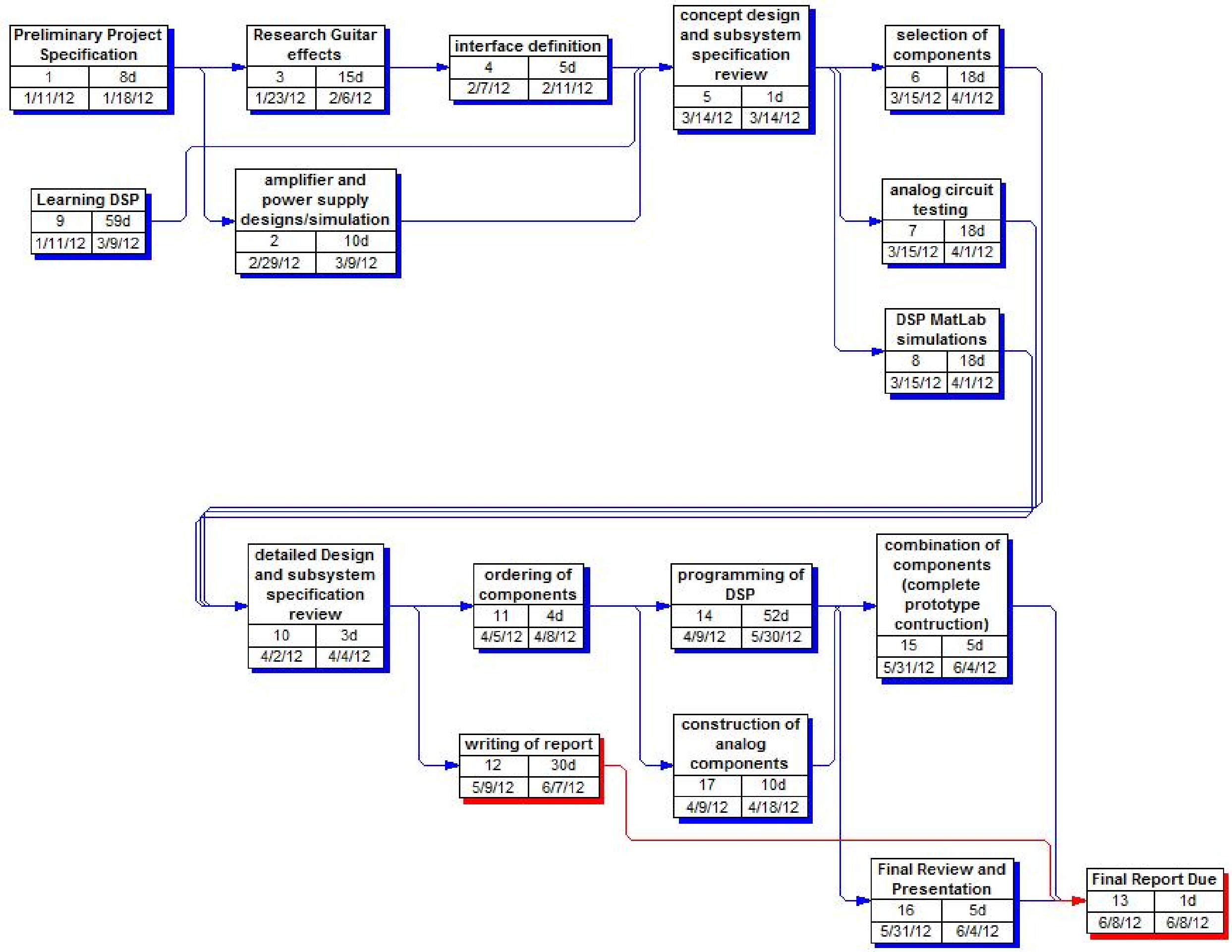
```
	//direct form II transpose diffeq for 2nd order BPF fc = 128Hz
	youtg = Bk7_gain[0]*xin+w11g;
	w1g = -Ak7[1]*youtg+Bk7_gain[1]*xin+w22g;
	w2g = -Ak7[2]*youtg+Bk7_gain[2]*xin;
	w22g = w2g;
	w11g = w1g;

	//direct form II transpose diffeq for 2nd order BPF fc = 128Hz
youth = Bk8_gain[0]*xin+w11h;
	w1h = -Ak8[1]*youth+Bk8_gain[1]*xin+w22h;
	w2h = -Ak8[2]*youth+Bk8_gain[2]*xin;
	w22h = w2h;
	w11h = w1h;

	//direct form II transpose diffeq for 2nd order HPF fc = 128Hz
/*	youti = Bk9[0]*x+w11i;
	w1i = -Ak9[1]*youti+Bk9[1]*x+w22i;
	w2i = -Ak9[2]*youti+Bk9[2]*x;
	w22i = w2i;
	w11i = w1i;*/

	//final 18th order filter diffeq, parallel combination of all previous filters
	yout = .7*xin + (Int16) youta + (Int16) youtb + (Int16) youtc + (Int16) youtd + (Int16)
youte + (Int16) youtf + (Int16) youtg + (Int16) youth;//
(Int16) youti;

	return (Int16) (yout);
```



Preliminary Project Specification
1
8d
1/11/12
1/18/12
Research Guitar effects
3
15d
1/23/12
2/6/12
interface definition
4
5d
2/7/12
2/11/12
concept design and subsystem specification review
5
1d
3/14/12
3/14/12
selection of components
6
18d
3/15/12
4/1/12
Learning DSP
9
59d
1/11/12
3/9/12
amplifier and power supply designs/simulation
2
10d
2/29/12
3/9/12
analog circuit testing
7
18d
3/15/12
4/1/12
DSP MatLab simulations
8
18d
3/15/12
4/1/12
detailed Design and subsystem specification review
10
3d
4/2/12
4/4/12
ordering of components
11
4d
4/5/12
4/8/12
programming of DSP
14
52d
4/9/12
5/30/12
combination of components (complete prototype contruction)
15
5d
5/31/12
6/4/12
writing of report
12
30d
5/9/12
6/7/12
construction of analog components
17
10d
4/9/12
4/18/12
Final Review and Presentation
16
5d
5/31/12
6/4/12
Final Report Due
13
1d
6/8/12
6/8/12