Problem Statement

The Spectrum Digital OMAP-L137 Evaluation Module has been used this semester to investigate the methods of digital signal processing, specifically for audio processing. The lectures provided background and theory while the labs applied and reinforced applied coding with hands on examples. To illustrate what has been learned this semester, I will use this development board to create some delay based modulation effects used by many composers and audio engineers today. As well as being able to record audio to the development board, playback the recorded audio as well as reverse playback, and being able to change the pitch of the recorded audio.

Theory and Implementation

The effects are echoing, flanging, chorusing, and vibrato. They all use the delay line modulation structure as seen in figure one. This works by first taking the input and storing it into the delay buffer (Z), then it takes the sample (Z^{-N}) and applies an attenuation to the retrieved delayed sample and combines it with the original input. The final result is stored into the results array for outputting by the development board.

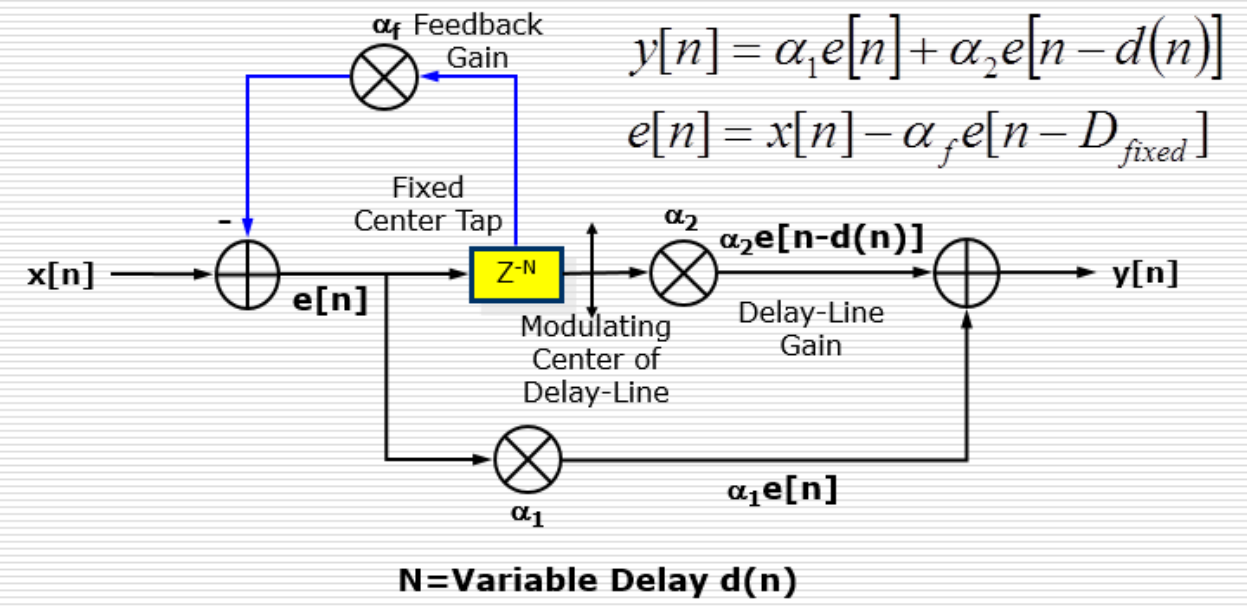

Figure 1 - Delay Line Modulation from Dr. Kepuska

In the delay line modulation structure used above, the delay is fixed. By adding a simple concept to change the center of the delay line will help be the back bone to many of the effects for this project. To achieve this, a circular buffer is used to allow the modulation of the delay line to change where the

sample is being retrieved from in the delay buffer. This modulation of the where the sample is being pulled from brings up another observation, when the delayed signal and the mixed signal are mixing this variable delay creates some interesting harmonics or wave cancelations depending on how long the delay is for.

The following code shows a simple implementation of a single echo using the delay line modulation structure in figure 1. It begins by copying the current input sample into the delay buffer, and then calculates the where in the buffer the needed delayed samples is located. Once found, it then adds the attenuated delayed samples to the current input samples and pass the result to the output buffer. The counters that keep track of the read and write positions of the delay buffer are clipped to the size of the

```
//A single echo of the input buffer based on fixed dealy
void SimpleEcho (short *pIn, short *pOut)
{
    int i;
    short delay = BUFLEN * 2 *2;
    static int echo_simple_sample_counter = 0;

    for(i = 0; i < BUFLEN * 2; i++)
    {
        //Copy the current input sample into the delay buffer
        EchoBuf[echo_simple_sample_counter] = pIn[i];

        //Calculate the read postion in the delay buffer
        int cur_index = echo_simple_sample_counter - delay;

        //Clipping read position on the buffer
        if(cur_index < 0)
            cur_index += SDRAM_BUF_SIZE;

        //Add the current dealyed sample to the current input sample
        pOut[i] = (short)(pIn[i] + (0.5 * EchoBuf[cur_index]));

        //Incremnt the read position and clip on the dealy buffer size
        echo_simple_sample_counter++;
        if(echo_simple_sample_counter >= SDRAM_BUF_SIZE)
        {
            echo_simple_sample_counter = 0;
        }
    }
}
```

delay buffer to keep it from finding or writing corrupt data out of bounds.

Flanging is created by varying the input signal with variable time delayed. The frequency of the time delay needs to be low with a about a 25 to 250 millisecond delay on the input. The swooshing effect is created by the sweeping up and down the frequencies that are close to the input sample, which in turn is due to the implementation of a circular buffer that will lower and raise the pitch as it rotates around its center. The rotation is due to the use of the circular buffer along with a low frequency oscillator that will move the delay sample value between within a fixed window bounded by the given sample delay length for the circular buffer. This structure of the flange effect can be visualized as seen below.

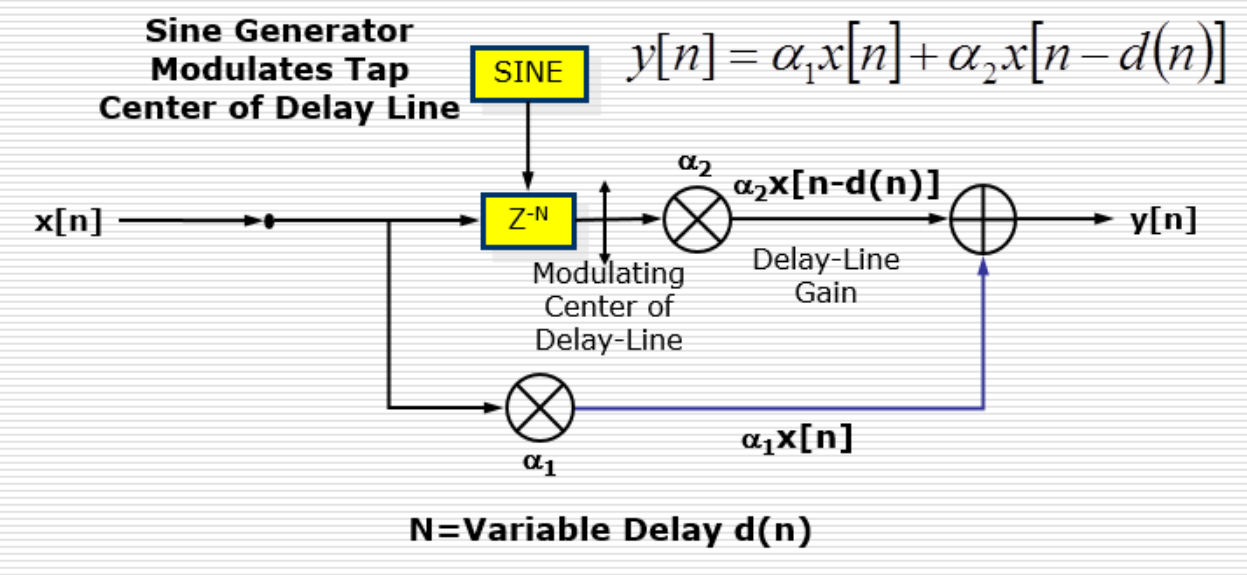

Figure 2 - Flange structure by Dr. Kepuska

The code implementation of the flanger effect only has one difference from the basic fixed echo delay function, that the delay itself is controlled by a low frequency oscillator comprised of computing cosine that has period that is normalized for the sample rate and desired frequency over this sample rate.

```
//Creates the flanger effect by attenuating the pitch of input samples by reading the delay
//buffer at diffent rates
void Flanger(short *pIn, short *pOut)
{
    static int flang_n = 0;
    static int flang_sample_counter = 0;

    int i;
    float PI = 3.14f;
    float FSAMPLE = 44100.0f;
    float flanger_freq = 0.4f;
    float D = 2205.0f; // Sample Delay

    float f_cycle = (flanger_freq/FSAMPLE);

    for(i = 0; i < BUFLEN*2; i++)
    {
        //Create variable delay LFO (based of cosine) (cicular buffer)
        short delay = (short)((D/2.0) * (1.0 - cosf(2.0 * PI * (float)flang_n *  f_cycle)));

        //Copy current input sample into FlangBuf
        FlangBuf[flang_sample_counter] = pIn[i];

        //Calculate the index to read from
        int cur_index = flang_sample_counter - delay;

        //Clipping on the buffer
        if(cur_index < 0)
            cur_index += SDRAM_BUF_SIZE;

        //Output the current input plus the sample from the FlangBuf
        pOut[i] = (short) ((pIn[i]) +( 0.5 * FlangBuf[cur_index]));

        //Increment the counter for the LFO
        flang_n++;
        //Clip the on the period of the LFO
        if(flang_n > (1/f_cycle))
            flang_n = 0;

        flang_sample_counter++;
        if(flang_sample_counter >= SDRAM_BUF_SIZE)
        {
            flang_sample_counter = 0;
        }
    }
}
```

The chorus effect helps to create richer and fuller sounds. Typically, to enhance the sound of a single instrument or singer to make it sound like many musicians or singers are playing simultaneously. Like many musicians playing at once, the effect needs to have slight differences in the timing, volume and

pitch of the notes being played or sung. These can be recreated by adding multiple time varying delays to the input while slightly changing the pitch of each recreated note. It is similar to the flange effect, but the delay is longer which will help to achieve lower harmonics as in figure four. Again the structure, see figure three, is similar to a flange, but this has two delayed samples to add to the original input. Each delayed sample has its own attenuation, low frequency oscillator, and delay length to make them different, but similar enough to make some interesting harmonics.

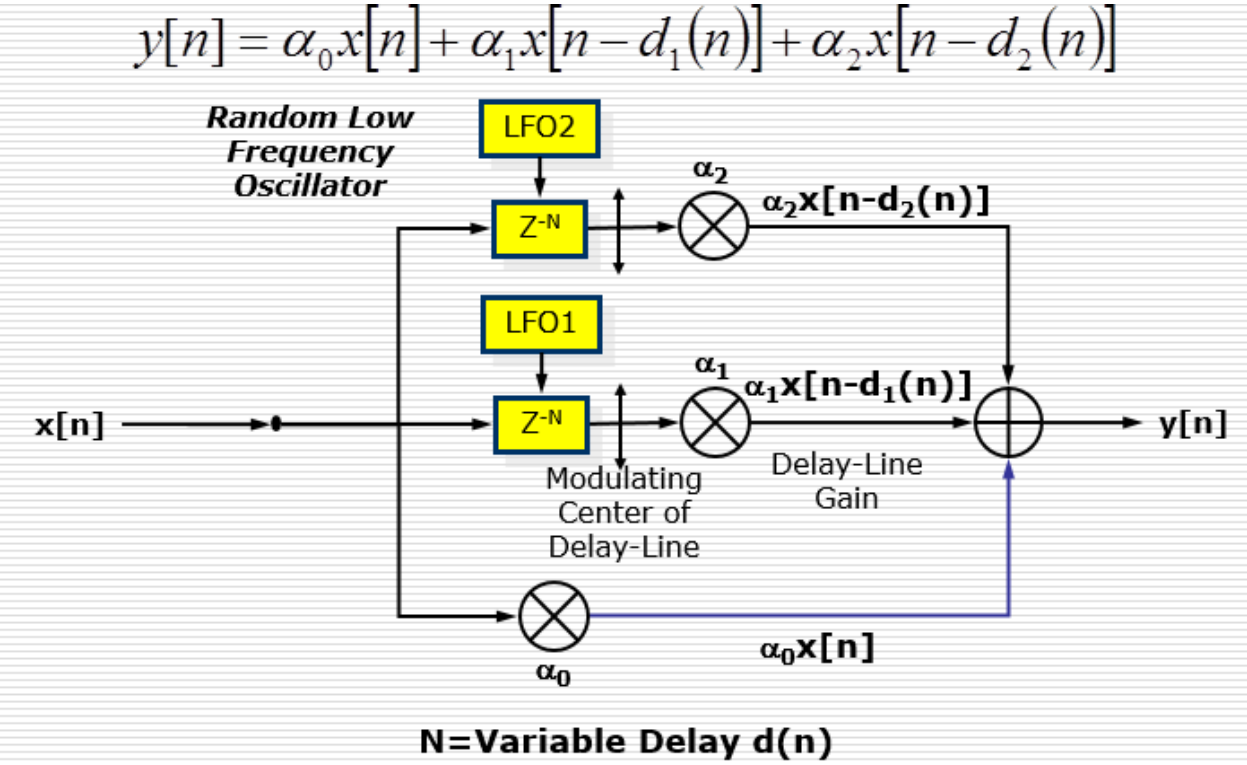

Figure 3 - Chorus Structure from Dr. Kepuska

The implementation of chorusing is similar to the flanger function, but has two delay samples that have a higher frequency for their low frequency oscillator and a longer delay time. The needed delayed samples are found and added to the current input sample, creating the multiple instrument effect of the chorusing. All the need counters are being tacked and are clipped to the size of the buffer when need to provide continuity of the delayed samples.

```
//Provides a two attenuating pitches of the input samples to create the effect of two extra
//insturments playing
void Chorusing (short *pIn, short *pOut)
{
    static int chorus_sample_counter = 0;
    static int chorus_n1 = 0;
    static int chorus_n2 = 0;
    int i;
    float PI = 3.14f;
    float FSAMPLE = 44100.0f;
    float chorus_freq1 = 0.7f;
    float chorus_freq2 = 0.9f;
    float D1 = 661.0f;
    float D2 = 1323.0f;
    float f_cycle1 = (chorus_freq1/FSAMPLE);
    float f_cycle2 = (chorus_freq2/FSAMPLE);

    for(i = 0; i < BUFLEN * 2; i++)
    {
        //Calculate delays based off LFO
        short delay1 = (short)(short)((D1/2.0) * (1.0 - cosf(2.0 * PI * chorus_n1 *  f_cycle1)));
        short delay2 = (short)(short)((D2/2.0) * (1.0 - cosf(2.0 * PI * chorus_n2 *  f_cycle2)));

        //Calculate the current read index
        short cur_index1 = chorus_sample_counter - delay1;
        short cur_index2 = chorus_sample_counter - delay2;

        ChorusBuf[chorus_sample_counter] = pIn[i];

        //Clipping on the buffer
        if(cur_index1 < 0)
            cur_index1 += SDRAM_BUF_SIZE;

        if(cur_index2 < 0)
            cur_index2 += SDRAM_BUF_SIZE;

        //Construnc the new output based off dealyed and current samples
        pOut[i] = (short) (pIn[i] + (0.5 * ChorusBuf[cur_index1]) + ( 0.3 * ChorusBuf[cur_index2]

        //Incerment all counters
        chorus_n1++;
        if(chorus_n1 > (1/f_cycle1))
            chorus_n1=0;

        chorus_n2++;
        if(chorus_n2 > (1/f_cycle2))
            chorus_n2=0;


        chorus_sample_counter ++;
        if(chorus_sample_counter > SDRAM_BUF_SIZE)
        {
            chorus_sample_counter = 0;
        }
    }
}
```

The vibrato effect is used to help sustain or oscillate a note directly after it has been played. This is less of the delay line modulation effect and more of a pitch modulation effect. By taking one of the chorus effects, and increasing the frequency of the low frequency oscillator, will create a more noticeable pitch variation of the echoed note. This will play the notes back quicker or slower depending upon the position of the circular buffer in the delay buffer. The structure can be seen below in figure 4.

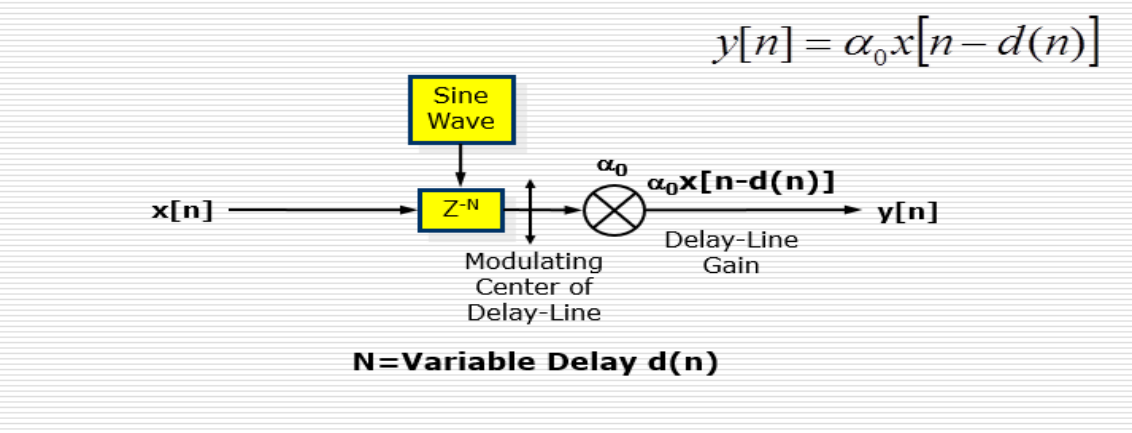

Figure 4 - Vibrato structure from Dr. Kepuska

Again, the code implementation is modified slightly from the rest. It copies the current input sample into the delay buffer. Calculates the delay sample need based off the low frequency oscillator, and then outputs then delayed sample. Does not add in the current input because the delay buffer will eventually produce the desired input sample not soon after due to the short delay time.

```
//quickly echos the sample input back with a faster chaning pitch
void Vibrato(short *pIn, short *pOut)
{
    static int vib_sample_counter = 0;
    static int vib_n = 0;
    int i;
    float PI = 3.14;
    float FSAMPLE = 44100.0;
    float vib_freq = 1.5f;
    int D = 100;

    float f_cycle = (vib_freq/FSAMPLE);

    for(i = 0; i < BUFLEN * 2; i++)
    {
        //Calculate the dealy based off the LFO
        short delay = (short)((D/2) * (sinf(2 * PI * (float)vib_n *  f_cycle)));

        //Store the current input sample into the delay buffer
        VibBuf[vib_sample_counter] = pIn[i];

        //Calculate the read position for the delayed sample
        int cur_index = vib_sample_counter - delay;
        //Clipping on the buffer
        if(cur_index < 0)
            cur_index += SDRAM_BUF_SIZE;

        //Output the dealyed sample
        pOut[i] = (short) (1.0 * VibBuf[cur_index]);

        vib_n++;
        if(vib_n > 1/f_cycle)
            vib_n=0;

        vib_sample_counter++;
        if(vib_sample_counter >= SDRAM_BUF_SIZE)
        {
            vib_sample_counter = 0;
        }
    }
}
```

Distortion is simply achieved by deformation of the original waveform. From what I have read, there is no one way to distort a waveform, so I simply took hyperbolic tangent of two times the input sample, attenuated it, then outputted the result. The code is seen below.

```
//Distortion of the input samples by using the hyperbolic tangent function
void Distortion(short *pIn, short *pOut)
{
    int     i;
    float   amplitude = 1500;

    for(i = 0; i < BUFLEN; i++)
    {
        pOut[i*2] = (short)(amplitude * (tanh( 2 * pIn[i*2])));
        pOut[i*2+1] = (short)(amplitude * (tanh( 2 * pIn[i*2+1])));
    }
}
```

The recording of input into the development board is also quite simple. Due to the small amount of internal random access memory on the board, the SD random access memory, external, must be utilized. By linking the SDRAM to our program, we can use a preprocessor to allocate the array created in code to memory on the SDRAM. All thought the SDRAM is slower to access than the internal RAM, it provides 64MB of space of use to the programmer. As with any C derived programming language, the exact size of the array is need at compile time so that the proper space can be allocated by the complier. I have allocated enough space to record and playback 15 seconds worth of sound. The length of the array can be found by first, multiplying the sample rate (44.1 kHz) by the number of seconds desired (15) to get the total number of samples need for the whole 15 seconds. Recording is done by copying the contents of the input buffer to the recording buffer on the SDRAM.

```
//Copy the contents of the input buffer to the recording buffer
void Record(short *in)
{
    memcpy(&record_buffer[record_counter], &in[0], BUFSIZE);

    record_counter += BUFLEN *2;

    if(record_counter >= REC_BUF_LEN)
    {
        record_counter = 0;
        //Play forwards
        rec_mode = 2;
    }
}
```

The position of where the samples are being copied is tracked and used to end the copying portion of the code. Playback is recording in reveres to the output buffer of the development board. It also keeps

track of the samples being played out to the development board and will stop copy from the recording

```
//Plays the samples recored in the recording buffer back to the output buffer
void Forward(short *out)
{

    memcpy(&out[0], &record_buffer[record_counter], BUFSIZE);

    record_counter += BUFLEN *2;

    if(record_counter >= REC_BUF_LEN)
    {
        record_counter = REC_BUF_LEN;
        //Play backward
        rec_mode = 3;
    }
}
```

buffer once all the data has been copied out.

Playing back the recorded samples in revers is a little bit different of an operation than the previous two. The data cannon be simply copied using the optimized memory copy function, but has be copied over sample by sample, but in reverse order from the recording buffer. This again is tracked to prevent the

```
//Plays the samples recored in the recording buffer in revers order to the output buffer
void Back(short *out)
{
    int i;

    for(i=0;i < BUFLEN * 2; i++)
    {
        out[i] = record_buffer[record_counter-i];
    }

    record_counter -= BUFLEN *2;

    if(record_counter <= 0)
    {
        record_counter = 0;
        //Play forwards
        rec_mode = 0;
    }
}
```

function from reading out of bounds on the recording array.

Results

The results of my project are working functions for the delay modulation based audio effects echo, flanger, chorus, and vibrato as well as the ability to record and playback the recording forward and backwards. Also, the code was setup in a way where the audio effects could be stacked either on top of each other; a noise gate or compressor is recommended when mixing audio effects due to the possibility of having a wider range of dynamics. The recorded audio could even be the source for the effect functions. The understating, research, theory, and code development/testing has given me a great comprehension of delay based modulation effects. It took a combination of digital signal processing theory along with an understating of the OMAP-L137 development board code development process to create the audio effects.

Additional Notes

It would have beneficial to have had some labs on the use and code implementation of the Fast Fourier Transforms. I performed a lot or research and time trying to implement a pitch detection and pitch correction of audio that utilized the Fast Fourier Transforms to no avail and had to refocus my efforts for this project. The audio effects that I created were very enlightening in to the thought process behind digital signal processing and how to implement them on a microcomputer architecture.

References

http://www.donreiman.com/Introductory_Home_Page.htm

http://www.csounds.com/ezine/winter1999/processing/