DSP Implementation of an Audio Effects Processor on the Motorola DSP56303 Processor

Authors: Kevin Ziemer
Rob Wolfbrandt
Eric Veit
Raghav Mani
Jeff Brockmole

Introduction

The processing of musical signals is very important in both the production and composition of music. Audio effects have become commonplace since their advent in the late 1960's. As an emerging technology, engineers were forced to rely on circuit elements to get the desired effect. Although technically effective, this level of technology was time-consuming from a production perspective and yielded a product that was both bulky and expensive. The creation and refinement of digital signal processing coupled with other important advancements in the field of electrical engineering have moved audio effects into the mainstream. By its very nature, digital signal processing has enabled the development and production of very high quality effects that are far less bulky and vastly less expensive.

This paper describes the theory and usage of digital audio effects and how they can be implemented on the Motorola EVM56303 Digital Signal Processor.

The audio effects discussed are:

- Equalizer
- Delay
- Reverb
- Sampler
- Flange

1. Equalizer

1.1 - Introduction

Equalizers have always been an essential element to audio playback. It is the ubiquitous nature of the equalizer today and our appreciation of the engineering that made the equalizer possible that prompted us to include it as one of our audio effects in the project. What follows is a detailed overview on the research and design that went into the development of our equalizer

1.2 - Design Overview

The first design consideration that had to be addressed concerned the number of bands we would have in our equalizer and which frequencies those bands would contain. After investigation, we decided that the design that would best work with our overall project would be a three-band equalizer (with pass bands of 0-150 Hz for the lower band, 150-1250 Hz in the mid-band, and 1250-20,000 Hz for the upper band). Transfer functions of the filters that were implemented are shown below.

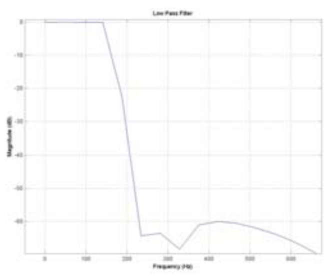
Figure 1: Low Pass Filter

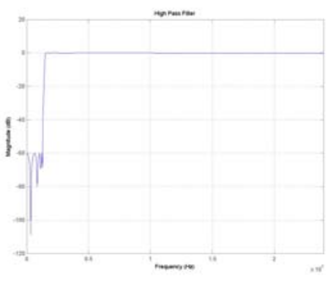
Figure 2: High Pass Filter

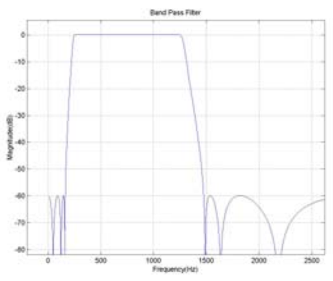
Figure 3: Band Pass Filter

1.2.1 Programming

The next step involved determining the type of filter to be used. Based on both our experience in lab and some additional research, we concluded that transposed direct form 2 bi-quad stages would best implement the appropriate elliptic IIR filters. We then set about determining the proper filter coefficients using MATLAB's FDATOOL. After designing the appropriate high and low pass filters for the treble and bass, it became apparent that MATLAB could not design a stable band pass filter. From that, we decided to achieve the mid-band section of the equalizer, we would cascade a low pass and high pass filter with the appropriate upper and lower cut-off frequencies. After the filters were designed in MATLAB and the proper form of the coefficients were extracted for the bi-quad stages, we started on the assembly coding.

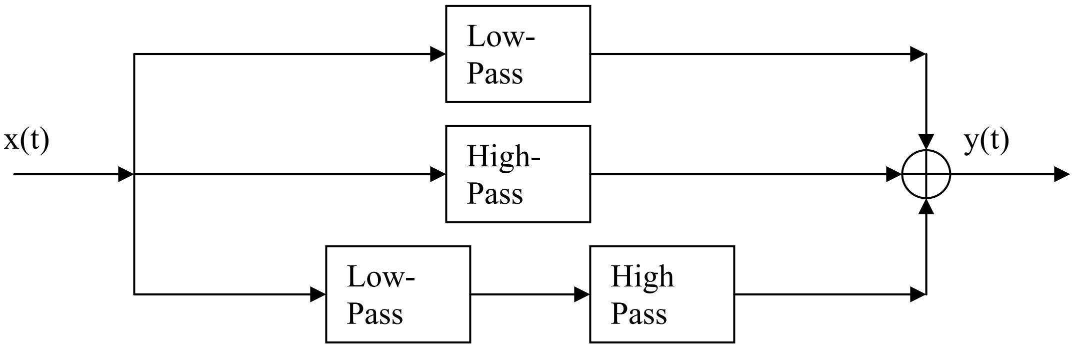

Figure 4: Signal Flow in Equalizer

The backbone of the equalizer design was the FFT program from lab. All FFT-relevant code was stripped out, leaving only the I/O support needed to implement the equalizer. Next we developed the IIR filter code. Using old labs and lectures as a guide, the code was developed and tested successfully. After testing the low-pass filter and completing all debug, the code was generalized to achieve all three bands of the equalizer. Once that was achieved, knob support was included to give the filters the added functionality of variable gain.

1.2.3 Problems

Although the development process went smoothly for the most part, it was not without problems. In early attempts to implement the equalizer we attempted to store 256 samples, and then process them all at once. That attempt did not go well. We realized that a better approach would be to process all the samples as they came in from the CODEC. That indeed proved a better method. Another problem that we ran into was in the final implementation when we ran out of room in the program memory. The way the memory was set up we were to have 1000 lines (0x400) for program code. In the final implementation of the equalizer, we had 0x4DC lines of code, which caused the processor to go off-chip, and caused the program to crash. After an extensive debug effort we (with significant help from Prof. Metzger) located the problem that was causing the program to crash, and we were able to determine the root cause. After we cleared those hurdles, the rest was clear sailing.

Then end result is a very marketable product, especially when bundled with our other effects. Future development for the equalizer aimed at enhancing its marketability would be to make more frequency bands for the user to set, and to give a wider range for gain swings in each band. Overall, the quality of the audio out of our equalizer was very high, and the effect had the desired response in its final implementation.

2. Delay

2.1 - Introduction

Delay, when considered as an acoustic effect, is the superposition of an original signal with one or more attenuated time-shifted versions of the original signal. Mathematically, the delayed output may be expressed as $y(t) = x(t) + A_1x(t-D_1) + A_2x(t-D_2) + A_3x(t-D_3)......$.
As a closed form summation the delayed output can thus be written as

$y(t) = x(t) + \sum_{n=1}^{m} A_n x(t-D_n)$ where the choise of m dictates the order of the delay stages, *A* is the attenuation coefficient for each stage and *D* is the delay amount for each stage.

2.2 - Delay as an audio effect

Delay is used heavily as an audio effect in the music industry to add a sense of depth and realism to specific audio voicings or tracks. Delay processors are tailor made for recording studios, guitar effects, synthesiser effects as well as DJ gear. As an effect it is usually found as part of a larger ‘multi-effects’ processor where there are numerous additional audio effects built into the main processor.
Most delay processors usually have several parameters that control the size of the attenuation coefficient, the amount of delay and, in certain circumstances, the number of delay stages.
The delay effect is a general model for another very common effect, echo. Echo is simply a multistage delay with integer multiple delays between each stage and exponential decay in attenuation coefficients between each delay stage. This models an original signal being reflected and attenuated at regular intervals. For its close resemblance to echo and its

inspiration for other effects such as chorus and reverberation delay is also an interesting starting point in the design of audio effects.

2.3 - Design considerations

Referring to the general delay equation we have that $y(t) = x(t) + \sum_{n=1}^{m} A_n x(t - D_n)$. For a given delay output we have a series of multipliplication and additions of various different delayed versions of the original input signal. This makes delay an idel effect to be implemented on a DSP processor.
Most professional delay processors on the market make use of an adjustable, multi-stage delay model. That is the delay output is modelled by $y(t) = x(t) + \sum_{n=1}^{m} A_n x(t - D_n)$ where m can be varied to select the number of stages. Even though it would be a relatively easy task (in DSP terms) to expand this delay output to two or more stages we decided that we would use a one stage delay model in our implementation. This is the most common model found in lower end consumer targeted products. It was found that this one stage model sounded had suitable depth nd quality to be used in the final implementation.
It was hence decided that our implementation of delay would use the one stage delay model $y[n] = x[n] + A_1 x[n - D_1]$. Our final implementation would ideally make use of knobs to control the attenuation and amount of delay of the first stage. The range and step size of attenuation and delay would be determined by the range of the knob parameters as well as the sampling rate and buffer size we would use in our code for delay.

2.4 - Delay implementation

To implement the delay code we decided to make use of already existing code for the Motorola EVM56303 DSP that computed FFT's. This code already had the required CODEC appendages to make use of the onboard switches, external knobs and to make use of the inputting and outputting of samples through the audio in/out of the development board.

2.4.1 - Basic implementation

To start off we decided it would be best to implement a delay model with constant attenuation coefficient and delay. The outputted delayed signal was to be of the form $y(t) = x(t) + 0.5x(t - 0.5)$. Below is the main portion of the code used to implement this function:

```
loop                ;
jsr    get_l_val    ; Get new sample from CODEC and store in accumulator
move a,x:(r4)       ; Store new sample in buffer
move x:(r4)-,x0     ; Put new sample in x0 decrement to delayed sample
move x:(r4),a       ; Move delayed sample into accumulator
asr #1,a,a          ; Shift delayed sample right by one bit (divide by 2), put in a
add x0,a            ; Add new sample with 0.5 * delayed sample, store in a
move    a,x0        ; Move delayed sequence from accumulator to x0
jsr    put_l_val    ; Output delayed sequence to CODEC
jmp loop            ; Start over from start of loop
```

The code starts off by getting a new sample from the CODEC and storing it in the accumulator, using the function get_l_val. The sample is then moved from the accumulator into the buffer. The size of the buffer has been specified earlier in the code and is set to the same size as the sampling rate. The new sample is then copied to the x0 register and the pointer is decremented to be at the position of the older, delayed sample. This delayed sample is then moved into the accumulator where it is halved. The new sample (in x0) is then added to the halved, delayed sample and stored in the accumulator. Once this has been done the entire delayed output is then stored in the x0 register where the put_l_val function takes this delayed sample and transfers it to the CODEC where it is outputted to the audio out of the development board.
From the above code snippet it easy to see how the attenuation coefficient is implemented into the delay stage. However, the way in which delay is introduced into the signal is a little more subtle. The amount of delay is intrinsically specified by the size of the offset used in the n (offset) register. When the pointer is decremented, the size of the decrement is specified by the offset register. If we wanted to grab an 1800 old sample, the offset register would be appropriately set to 1800. In the case of our code, we made use of a sample rate and modulo size of 27000 samples. Since we wanted a half second delay we set our offset register to 13500. In generality the delay time can be set by the relation $\text{delay time} = \frac{\text{offset size}}{\text{buffer size}}$.

2.4.2 - Final implementation

In the final implementation knob support was added to the code to change the amount of delay and the attenuation coefficient. The first knob was used to modify the size of the offset in the offset register which appropriately changed the amount of delay. Based on the use of a 27000 Hz sample rate, it was decided that the knob would change the amount of delay from 0 s to 1 s in 66.7ms increments. The second knob as used to modify the attenuation coefficient from 0 to 0.5 in 0.033 increments.

2.5 - Performance and improvement of delay effect

The final implementation of the delay effect proved to be a versatile high quality implementation. The amount of delay and attenuation had a fairly wide range. The delay ranged from 66.7ms to 1s. This gave a chorus sounding effect on one extreme to a long tap delay on the other extreme. The attenuation variation proved to be a useful parameter to incorporate as it adjusted the perceived level of the combined outputted level.
Even though we were pleased with the final implementation of the delay effect several improvements could be made to make this effect more marketable. Professional level delay processors make use of variable stage delay stages. This would be the most important and probably the most challenging feature to implement but would make the transition of our effect from consumer level to professional level. The second major addition to the design would be the addition of more memory so that the sampling rate could be increased upwards to 48000 Hz. This addition is completely hardware dependent and would not require much modification of our code.

3. Reverb

3.1 - Introduction

Reverberation is the combined effect of multiple sound reflections within a room. When a sound is played in a room, the sound wave does not only travel towards the listener's ears; it also spreads throughout the entire room. As the sound wave hits various obstacles in the room, most noticeably walls, it is reflected in other directions. During these reflections, the obstacles absorb some of the sound wave's energy based on the properties of their materials. This reduces the amplitude of the reflected sound waves. The time taken for these reflections increases the delay associated with the reflected sound waves. When the ear receives the superposition of all these delayed sound waves, it detects not only the original sound, but also the environment that the sound was played in. In general, this is why things sound different in a bathroom as opposed to a living room or a concert hall.

3.2 – Modeling Reverb

From the above description, it can be seen that there are two major components to the principle of reverberation: one is the size of the room, the other is the materials used in the room. The size of the room is important because as the size increases, the sound waves have to travel longer distances, and thus longer delays are created. The size of the room can be represented by it's volume. The materials used in the room can be represented by their absorption coefficients. As the absorption coefficients increase, the more the material will decrease the amplitude of the sound wave it reflects. The exact opposite of the absorption coefficient is the reflection coefficient.

A common measurement of this effect is through a room's reverberation time (T_{60}). The reverberation time is defined to be the time it takes a sound to reduce 60 dB in amplitude from the time its source is halted. The equation follows:

$$T_{60} = 0.163 \times \frac{V}{S_e} \quad \text{where} \quad S_e = \alpha_1 S_1 + \alpha_2 S_2 + \ldots + \alpha_n S_n$$

where V is the volume of the room, S_n is the surface area of an object, and α_n is the object's absorption coefficient.

3.3 – Brief History

Reverberation and other acoustic principles were known by the ancient Greeks. Their amphitheatres were designed so that sounds on the stage were projected while sounds in the crowd were muffled. Throughout times since, theaters evolved with the popular music of the time. It was found that a room could be acoustically matched to the music played in it. In fact, it was while fixing a problem with the acoustics at Harvard's Fogg theater in 1895, that a physicist named Sabine formed the equation given above for the reverberation time.

Although reverberation is still a factor in designing acoustic environments for the best possible received sound, it has taken on a new application in the making of music. When used for this purpose, reverberation is best known as an audio effect called reverb. By using reverb, an instrument or voice can be altered in its characteristic sound. Perhaps most commonly, reverb is added to make sounds have more weight and depth then normal.

Reverb is also becoming more popular as a playback option. Many stereos now have features where the user can select the musical environment—sometimes so specifically that, say, a particular opera house can be specified. There are also products that can calculate the reverberations in a room and then inversely compensate for them during playback.

3.4 – Implementation

There are many different ways to implement the reverb effect. Interestingly, analog reverb effects were mostly done by putting the signal on springs, vibrating metal plates, and even rippling water tanks. With digital reverb, algorithms on DSPs are used. All of these techniques are based on two ideas: delay and attenuation. As the number of delay stages increases, the effect becomes more believable and accurate. It is also better if the delay stages are random (hence the vibrating plate.)

In early digital designs, comb and all-pass filters were used to create the effect. It is now done with a large number of programmable delay stages, gains, filters, and feedback loops. This can all be implemented using DSP. The implementation used in this project is a very simplified version of this.

It was decided that the reverb effect for this project should be a relatively simple one since four other effects were also being created simultaneously. Therefore, the reverb at first took the very simple form of one delay stage with an associated attenuation. It is important to note that this equation has feedback—the delay stage is being fed with outputs rather than inputs. The equation follows:

$$y[n] = x[n] + A*y[n-D] \text{ where } A\text{=attenuation constant, } D\text{=delay time}$$

Although reverb can be much more complex, this equation was chosen due to its simplicity as well its ability to be easily upgraded with more delay stages. This was a good place to start programming.

3.5 – Writing the Code

First, the algorithm was implemented using MATLAB. The code was implemented using an array; although this was similar to a buffer in DSP, the addressing did not involve a pointer and was not circular. Also, a function called 'wavread' was used to input a complete sound file and the function 'sound' was used to play the result. All this coupled with a large 'for' loop led to long execution times. It was difficult to assess the quality of the effect because it was not in real time. Although the MATLAB coding was a good exercise, not much was determined. The code follows:

```
function [y]=reverb2(fn);
% fn-string containing filename of wav file
[x,fs]=wavread(fn);                         %read in wav file
xlen=length(x);                             %Calc. the number of samples in the file
%Initialize constants
a=0.3;delay=.2;
D=delay*fs;                                 %Calculate the number of samples in the delay
y=zeros(size(x));
%reverb
for i=2*D+1:1:xlen                          % D+1 is the index where the delay starts
  y(i)=x(i)+ a*y(i-D)+ a*.8*y(i-2*D);
end;
sound(y,fs);                                %play y
```

As is the case with the other effects in this project, the FFTLab code written by Professor Metzger for the lab exercises was used as the foundation of the reverb program. This existing code was used as a basis for input/output support between the A/D, DSP, and D/A. All relevance towards its previous FFT functions was removed or modified. The primary functions of the stripped code are setting up memory, initializing the CODEC, and providing interrupt request support.

It is clear that the equation above will need a circular buffer to keep track of delayed outputs. This was easy to implement using modulo addressing and a pointer. The maximum delay can be defined by setting the modulo of the buffer pointer to be the product of the delay in seconds and the sampling frequency in Hz. This means that the oldest value (position) in the buffer always corresponds to the maximum delay set above. In this implementation, a 48 kHz sampling rate and a 24,000 point buffer was used, at first, for a maximum delay of a half of a second. This was later found to be a rather short reverberation time.

The first generation code of the reverb loop follows:

```
reverb
        clr     a               ;clear accumulator to be safe
        jsr     get_l_val       ;input new sample from CODEC, store in accumulator
        move    a,x:(r4)-       ;store new sample to buffer, decrement to delayed pos.
        move    x:(r4)+,x0      ;put delayed sample into x0, increment to new pos.
        mac     #0.5,x0,a       ;scale x0, accumulate w/ new sample
        move    a,x:(r4)-       ;put reverbed sample into new, decrement
        move    a,x0            ;move value to x0 for output
        jsr     put_l_val       ;output reverbed sample to CODEC
        jmp     reverb          ;start over
```

This code is pretty straightforward. A new sample is brought in from the CODEC and stored into the first buffer position. This will later be replaced by the new output. This is done because the first time through the code, the buffer needs to be filled with preliminary values. The buffer pointer is then decremented and the oldest value is placed into x0. The buffer pointer is then incremented to point back to the position it started out at. The value in x0 is then scaled and accumulated with the new sample already in the accumulator. This value is then copied into the buffer and output to the CODEC. The buffer pointer is decremented and the whole process starts over.

3.6 – Testing, Updating, Adding Functionality

Listening to this version of the reverb, it sounded too much like delay when running it on music files. When using a guitar, the sound was not as desired. It was too obvious in its delay mechanism and not colored enough in its presentation. It was decided to add an additional delay stage to the code. This was done by using an appropriate offset register when addressing the buffer. This new delay value was stored in x1 and the code required an additional mac instruction.

It was decided that this effect would be used only with guitar, so the sampling rate was dropped to 16 kHz. Even 8 kHz would have been appropriate for guitar, but the extra room provided by the higher rate might be appreciated sometime. It was found the best sounding reverb occurred with the delay stages set at 1.5 and 0.25 seconds with coefficients of 0.1 and 0.3 respectively. The effect at this point was very likeable and sounded surprisingly similar to professional reverb systems.

Just because these parameters sounded best for one situation, it does not mean they will be ideal in all instances. Therefore, knob support was added to the code so that parameters could be changed on the fly. The code for the knobs was supplied by Professor Metzger. It was first modified for our use in the flanging effect. The major modification was the addition of limits to the rotation of the knobs to provide 16 possible knob positions.

Knob one was assigned to adjust the short delay time between zero and 1 second by adjusting the size of the offset register. The long delay time was kept at 1.5 seconds by the buffer size. Knob two was assigned to adjust the attenuation coefficient on the short delay stage between zero and 0.5. The coefficient of the 1.5 second delay stage was kept at 0.1. The knobs were set to initialize to the preferred settings mentioned above: 0.25 seconds and 0.3 attenuation.

Also, it was decided to provide an on/off switch for the effect. This was done through use of SW3 interrupt support. The code started off getting a value from the CODEC, outputting it to the CODEC, and checking to see if SW3 was pressed. When this event happened, the event bit was cleared and the code jumped into the reverb loop while still checking for an interrupt to send it back to its previous state.

These additions to the code made the effect sound much more professional and also added a useful interface to the major parameters of the effect. For use with a guitar, a preamplifier was also added to the input of the CODEC by setting an option. Virtually all effect processors on the market have the on/off feature and certainly the option to adjust the effect's parameters.

3.6 – Final Testing

Upon final iteration of the code with all the features mentioned above added, the effect sounded very good. The obvious delays from the first version were all but gone. The sound had a nice amount of echo and the character of the sound was very appropriate for the guitar. The effect was not able to mimic an environment like some other processors are capable of, but did succeed in supporting and strengthening the character of the sound as might be done when recording.

Not all combinations of knob settings sounded great. However, depending on what was being played, some settings sounded really good. It was nice to be able to play with the settings in real-time. The effect on/off switch was also a good idea since it provides the player with even more control over the sound.

3.7 - Conclusions

Although the reverb sounds good as it is, it has a lot of room for improvement. More delay stages could be added, the delay times could be randomized, filters could be used to separate out frequencies, more knobs could be used to control even more parameters of the function, etc. The list is endless

For what it is, the reverb effect could be marketed as a very simple, easy-to-use reverb processor. The SW3 function could be assigned to a pedal, and the knobs could be easily interfaced. With the addition of some of the ideas given above, a respectable product could be made provided it could sell for about $100 or so.

4. Sampler

4.1 Introduction

A sampler is a device used for recording and playing back sound, in which computer RAM is used as the storage medium.

The first sampler was invented in 1963 by Mellotron. Organ like in appearance, each key was connected to a head and tape loop. Samples were recorded onto each tape loop per customers request at the factory. One of the most famous uses of the Melotron can be heard as the flute sound in Led Zepplin's Stairway to Heaven. Over the years, samplers have evolved into smaller, digital rackmount units and footpedals. Samples are easily recorded by the user in an "on the fly" fashion.

4.2 Design Overview

Samplers read in samples at specified sample rate and store them in RAM. The amount of music a sampler can store is therefore dependent on the sampling rate and the amount of RAM available on the device. For example, a sampler sampling at a CD-quality rate of 44,100 Hz can store 1 second of music if 44,100 words of RAM are available. 44,100 Hz is the CD-quality standard, which ensures that frequencies up to 22,050 Hz (above the range of human hearing) are recorded. For our audio project we decided on a sample rate of 8 kHz to be used for sampling a guitar, which produced its highest fundamental at

1328 Hz. This enabled our sampler to store four seconds of music in the 32k words off-chip memory available on the Motorola EVM56303. Note that the sampling rate of 8 kHz sounds fine on our guitar application because the harmonics are not very strong on this particular instrument. This sampling rate would not however be appropriate for sampling instruments with lots of harmonic overtones (distorted guitar, saxophones, …).

4.2.1 Programming

The following sampler function allows an unprocessed guitar signal to pass through the EVM56303 through the entire execution of the program. When SW3 is pushed, the sampler is activated and samples are continuously written into memory until SW3 is pushed again or until the buffer is filled with 32,000 samples. Upon either of these two exit conditions, the contents of the buffer are continuously played in a looping manner until SW3 is pushed again. The process can then be repeated by the user indefinitely.

The main issue was creating a circular buffer with a size equivalent to the number of samples stored in it. This is ordinarily accomplished by loading a modulo register with the number of samples in the buffer minus one. Since the size of the buffer is dependant on when the user toggles the SW3, the modulo register needs to be set during runtime. To do this, the program starts out with the modulo set to linear addressing. Then a counter is used to count the number of samples loaded into the buffer. When the buffer is filled, the contents of the counter minus one are loaded into the modulo register.

4.2.2 Problems

The SW3 switch physically bounces for about 20 milliseconds after it has been pushed. This means that the state of the SW3 switch can not be reliably checked for at least 20 milliseconds after the switch has been pushed. Fortunately, this is not a problem for our application as we always sample for periods which exceed 20 milliseconds. It is however a slight nuisance to have to wait 20 milliseconds between turning off the sampler and restarting it (i.e. reloading the buffer with new samples). It is for this reason that the wait subroutines exist in the program.

4.2.3 Usage

Samplers are used extensively in the recording industry. Modern recording studios have digital sound banks filled with horn sounds, drum sounds, etc… Most of the sounds we hear in popular music are actually samples, which are played back in the required time and pitch.

Samplers are also used in live performance. Guitar virtuoso, Michael Hedges, often sampled acoustic guitar loops and would then weave countermelodies over top of the sampled melody.

Samplers are also valuable practicing aides. It is very helpful for beginning musicians to practice scales over a repetitive rhythm. This enforces good time keeping and musicality.

5.Flange

5.1 Introduction

The flange effect was invented by John Lennon using a reel-to-reel tape machine. The effect was performed by gently tapping the edges (flanges) of the reels, effectively slowing the reels down, and mixing the sound with the original sound. The resulting sound creates a varying delay, resulting in small changes in pitch.

5.2 Design Considerations

There are a few concerns when one begins to approach the implementation of the flange effect using digital means. One such concern is what measures need to be taken in order for the output to sound smooth. One problem with using a digital means to implement flange is that it can sometimes sound choppy because of the gap in time between samples. This can be countered using oversampling. This is commonly done using a linear fit to place samples between two consecutive sampled points. The drawback to using oversampling is that there is more processing time to interpolate the values, and a larger buffer is needed to store the same amount of audio time. The benefit is a smoother sounding effect.

In our first trials through Matlab, we used 5x oversampling, placing 4 samples using a linear fit between the original sampled points. The effect sounded very smooth; however, this operation took a long time for Matlab to process. When we took out the oversampling, the effect sounded identical. This led us to drop oversampling from our considerations.

5.2.1 Implementation of Varying Delay

The main task in the flange effect is implementing a varying delay. The approach we took was to use a table of values as indexing offset values. By using a periodic function with the values of one period stored in a circular buffer, you can cycle through the buffer, one sample at a time, and use that value as an offset for referencing data values. The first approach we took in Matlab was to use a triangle waveform as our offset values. This resulted in two distinct pitch shifts… one lower and one equally higher from the original sound. The solution to this was to use a sine wave as the basis of our offset values. This results in smooth transitions between lower and higher pitch shifts, and a more natural sounding effect.

5.2.2 Logistics

The first decision we had to make in our preparation for coding flange on the Motorola DSP was to decide what kind of sampling rate we wanted to use. Since the flange effect itself is not very computationally intensive, we decided to use a sampling frequency of 48,000 Hz. This sampling frequency makes the effect more versatile, allowing for the effect to be applied to music with higher tones than just guitar. After this was decided, we needed to figure out what size buffers would need to be set up in order for the flange to sound good.

In our Matlab simulations, we found that a good size for the audio buffer would be on the order of 200 – 2000 samples, depending on the frequency of the flange and how much of a pitch shift the user prefers. This is a pretty reasonable size buffer to implement on our evaluation board, and can be stored in either on-chip or off-chip memory.

The big problem with using a sinusoidal waveform as the flange envelope is the number of sine values you would need to implement the flange. Since the DSP has no sine function built in, it is the programmer's responsibility to generate the sine values. Traditional flange has a frequency of around 1Hz, which when sampling at 48 kHz, would require a table of 48,000 sine table values. It would be nice (and necessary, since the total amount of data memory on our evaluation board is less than 48,000 words) if the same frequency of flange could be generated using a smaller sine table.

5.2.3 Problem Solutions

The solution we decided to work towards was using the same idea of linear interpolation we were thinking about with the audio samples, only this time apply it to the sine table data points. Since the sine table's only effect on the flanged audio is a change in pitch, it is not necessary to be extremely precise with the roundness of it, as the human ear can not detect tiny variances in pitch. With this in mind, we changed our Matlab simulation to use a 256 point sine table and interpolate between the data points. When we ran the trial, the difference between having a small sine table in which interpolation was used and having a large sine table without interpolation was indistinguishable. The next step was to move on to translating our Matlab code into the assembly language that our DSP uses.

5.3 DSP Coding

The first DSP coding was done using interpolation of a 256 point sine table. The program was set up using a do loop, as it was written in our Matlab code. We assigned a variable that indicated the number of points between consecutive sine table points we wanted to interpolate for, and used that as the loop counter. We also stored the inverse of the loop counter ahead of time, in fractional format, in another memory location. The main portion of the loop was then based on the equation:

$$Offset = A*\left[\left(1-\frac{counter}{N}\right)(*\,p_1)+\left(\frac{counter}{N}\right)(*\,p_2)\right]$$

Where **A** is an integer value, $\mathbf{p_1}$ and $\mathbf{p_2}$ are pointers to two consecutive sine table values, **N** is the number of points you want to interpolate for, and **counter** is a counter for how many times you have gone through the do loop.

The coding of this implementation was very difficult to debug, as it was impossible to run through our program using the step function of the debugger due to interrupt support. Another problem we ran into with this code was that it required a lot of multiplications, and data movements. After further thought, and collaboration with Professor Metzger, we ended up dropping our do loop and the original equation for a similar equation that required fewer multiplications and data movements. The equation we developed took the form of:

$$Offset = A * \left[* p_1 + \left(\frac{B}{N} \right) (* p_2 - * p_1) \right]$$

Where **N** is the desired size of the sine table, and **B** is a number generated by a modulo register and is used as a counter. With this implementation, it was easy to set up a couple of if statements to test for boundaries, and the debugging was much easier. After we got this implementation to work, the next step was to figure out what kind of customization options we wanted to give the user.

5.3.1 Knob Support

With the four knobs we had at our disposal, we had some flexibility in making our flange algorithm adaptable. The typical varying delay for flange has a form of:

$$Delay = A * \sin(wt)$$

By changing **A**, you can change how far back and forth the flange goes, effectively changing the amount of pitch shift associated with the flange. To implement knob control for this factor, we set up the knobs to go from 0 to 15. Then, by multiplying by a constant gain factor, you can obtain a proper offset indexing value that will control how far back and forth the flange oscillates. We used knob one to control this. By changing **w**, you can change the frequency the flange oscillates. This was implemented by changing the number of points you interpolate between consecutive sine values. The tricky part we ran into in coding this part of the program was the division that was present in the equation. Division is very difficult to perform on the DSP. To solve this, we ended up creating a 16 value array to hold the possible values of 1/N. Then, buy using the value generated by the knob (0 – 15) as an offset, we were able to get the correct value of 1/N. Knob 2 controls the frequency of the flange. The third and fourth knobs were used to control the gain of the regular audio and the flanged audio respectively.

Conclusion

In conclusion, using the Motorola 56303 DSP, we were able to implement several different audio effects using the same hardware. The ability to use the same hardware for different tasks ends up bringing the overall cost down to a very affordable level. Our project, with a few modifications and improvements, could be marketed to the entry level audio industry. Adding more memory would allow us to load all audio effects at the same time, creating a true multi effects processor. By having all the effects on the chip at once, we could toggle effects using a knob or run multiple effects at once. This has been a good introductory design experience incorporating digital signal processing techniques with a marketable application.