```
.var    sweep_rate = 1;                       /* controls M register, signal frequency fc = c*Mreg*ftimer
*/
.var    sweep_width = 0;                      /* controls width of sweep, ranges from 0 to - 15 */
.var    w[D + 1];                             /* delay-line buffer, max delay = D */
.var    sine_value;                           /* wavetable update via timer for delay calcuation */
.var    sine[WaveSize] = "sinetbl.dat";       /* min frequency f1 = fs/Ds = 8/4 = 2 Hz */
                                              /* load one period of the wavetable */
.endseg;


/* -------------PROGRAM MEMORY CODE---------------------------------*/
.segment /pm pm_code;

Init_Flange_Buffers:
        B2 = w;  L2 = @w;                     /* delay-line buffer pointer and length */
        m2 = 1;

        LCNTR = L2;                           /* clear delay line buffer to zero */
         DO clrDline UNTIL LCE;
clrDline:          dm(i2, m2) = 0;

        B6 = sine;                            /* pointer for signal generator */
        L6 = @sine;                           /* get size of sine table lookup */

        RTS;

/*--------------------------------------------------------------------*/

/* Set up timer for the Chorus Effects wavetable generator */

Timer0_Initialization:
        bit clr mode2 TIMEN0;                 /* timer off initially */
        bit set mode2 PWMOUT0 | PERIOD_CNT0 | INT_HI0;       /* latch timer0 to high priority timer int */

        r0 = modulation_rate;
        DM(TPERIOD0) = r0;
         DM(TCOUNT0) = r0;                    /* assuming 16.7 nSec cycle @ 60 MIPs */
        r0 = 10;
        DM(TPWIDTH0) =r0;

        bit set imask TMZHI;                  /* timer high priority  */
        bit set mode2 TIMEN0;                 /* timer on */

        rts;

/* ------------------------------------------------------------------------------------------ */
/*                                                                                            */
/*                      Wavetable Generator used for Flange Delay Line Modulation             */
/*                                                                                            */
/* ------------------------------------------------------------------------------------------ */
/* High Priority Timer Interrupt Service Routine for Delay Line Modulation of Flange Buffer */
/* This routine is a wavetable generator, where r3 = where r3 = D2 * sin(2*pi*fc*t)   */
/* and it modulates the delay line around rotating tap center */

wavetable_gen:
        bit set mode1 SRRFL;          /* enable secondary registers r0 - r8 */
        nop;                          /* 1 cycle latency writing to Mode1 register */

        m6 = dm(sweep_rate);          /* desired increment c - frequency f = c x fs / WaveSize */
        r1 = D2;                      /* Nominal Center Tap Delay Time */
        r2 = dm(i6, m6);              /* get next value in wavetable */
        r4 = dm(sweep_width);         /* store to memory for chorus routine */
        r2 = ashift r2 by r4;         /* control amount of variable delay via sweep_width */
        r3 = r1 * r2 (SSFR);          /* scale Nominal Delay Time by a fractional value */
        dm(sine_value) = r3;          /* save for flange routine */
        rti(db);
        bit clr mode1 SRRFL;          /* disable secondary register set */
        nop;                          /* 1 cycle latency to write to mode1 register */

/* ----------------------------------------------------------------------------------------- */
/*                                                                                           */
/*                      Digital Flanger - process right channel only                         */
/*                                                                                           */
/* ----------------------------------------------------------------------------------------- */
```

```
Flanger_Effect:
        r15 = DM(Right_Channel);      /* get x-input, right channel */

        r3 = dm(sine_value);          /* calculate time-varing delay deviation */
        r14 = D2;                     /* nominal tap center delay */
        r4 = r14 - r3;                /* r4 = d = D2 - D2 * sin(2*pi*fc*t) */

        /* r4 now will be used to set m2 register to fetch time varying delayed sample */
        m2 = r4;                      /* tap outputs of circular delay line */
        modify(i2, m2);               /* go to delayed sample */
        r4 = -r4;                     /* negate to get back to where we were */
        m2 = r4;                      /* used to post modify back to current sample */
        r6 = dm(i2, m2);              /* get time-varying delayed sample */

        /* r8 will be used to get Nominal Tap Center delayed sample for flange feedback */
        m2 = D2;                      /* tap outputs of circular delay line */
        modify(i2, m2);               /* go to delayed sample */
        m2 = -D2;                     /* negate to get back to where we were */
        r8 = dm(i2, m2);              /* get nominal delayed sample, postmodify back to current sample */

        /* crank out difference equation */
        r5 = a0;                      /* input gain */
        mrf = r15 * r5 (SSF);         /*   mrf = a0 * x    */
        r9 = dm(feedback_gain);       /* gain for feedback of nominal tap center*/
        mrf = mrf + r8 * r9 (SSF);    /*   mrf = a0 * x - af * sNominal */
        r12 = mr1f;                   /* save for input to flanger delay line */

        r7 = a1;                      /* delay line gain */
        mrf = mrf + r7 * r6 (SSFR);   /*   mrf = a0 * x + a * s1 - af * sNominal  */
        mrf = SAT mrf;                /* saturate if necessary */
        r10 = mr1f;                   /* flanged result in r10 */

        /* put 'input minus feedback' sample from r12 into tap-0 of delay line */
        /* and backshift circular delay-line buffer pointer */
        dm(i2, -1) = r12;

        /* send flanged result to both left and right output channels */
        DM(Left_Channel)=r10;
        DM(Right_Channel)=r10;

        rts;

/* ------------------------------------------------------------------------------- */
/*                                                                                 */
/*                   IRQ1 Pushbutton Interrupt Service Routine                     */
/*                                                                                 */
/*      This routine allows the user to modify flanger width and rate presets.     */
/*                                                                                 */
/*      Default before 1st IRQ push:                                               */
/*      1st Pushbutton Press:                                                      */
/*      2nd Pushbutton Press:                                                      */
/*      3rd Pushbutton Press:                                                      */
/*      4th Pushbutton Press:                                                      */
/*      5th Pushbutton Press:                                                      */
/*      6th Pushbutton Press:                                                      */
/*      7th Pushbutton Press:  Reverts back to 1st Pushbutton Press                */
/*                                                                                 */
/*      The pushbutton setting is shown by the active LED setting, all others are  */
/* --------------------------------------------------------------------------------*/

change_depth_rate_width:
        bit set mode1 SRRFH;                 /* enable background register file */
        NOP;                                 /* 1 CYCLE LATENCY FOR WRITING TO MODE1 REGISER!!   */

        r13 = 6;                             /* number of presets */
        r15 = DM(IRQ1_counter);              /* get preset count */
        r15 = r15 + 1;                       /* increment preset */
        comp (r15, r13);
        if ge r15 = r15 - r15;               /* reset to zero */
        DM(IRQ1_counter) = r15;              /* save preset count */

        r10 = pass r15;                      /* get preset mode */
        if eq jump delay_settings_2;         /* check for count == 0 */
        r10 = r10 - 1;
        if eq jump delay_settings_3;         /* check for count == 1 */
```

```
       r10 = r10 - 1;
       if eq jump delay_settings_4;          /* check for count == 3 */
       r10 = r10 - 1;
       if eq jump delay_settings_5;          /* check for count == 4 */
       r10 = r10 - 1;
       if eq jump delay_settings_6;          /* check for count == 5 */

delay_settings_1:                            /* count therefore, is == 6 if you are here */
       r14 = 1;              DM(sweep_rate) = r14;
       r14 = 0;              DM(sweep_width) = r14;
       bit set ustat1 0x3E;                         /* turn on Flag4 LED */
       bit clr ustat1 0x01;
       dm(IOSTAT)=ustat1;
       jump done;

delay_settings_2:
       r14 = 2;              DM(sweep_rate) = r14;
       r14 = 0;              DM(sweep_width) = r14;
       bit set ustat1 0x3D;                         /* turn on Flag5 LED */
       bit clr ustat1 0x02;
       dm(IOSTAT)=ustat1;
       jump done;

delay_settings_3:
       r14 = 3;              DM(sweep_rate) = r14;
       r14 = -1;             DM(sweep_width) = r14;
       bit set ustat1 0x3B;                         /* turn on Flag6 LED */
       bit clr ustat1 0x04;
       dm(IOSTAT)=ustat1;
       jump done;

delay_settings_4:
       r14 = 4;              DM(sweep_rate) = r14;
       r14 = -1;             DM(sweep_width) = r14;
       bit set ustat1 0x37;                         /* turn on Flag7 LED */
       bit clr ustat1 0x08;
       dm(IOSTAT)=ustat1;
       jump done;

delay_settings_5:
       r14 = 5;              DM(sweep_rate) = r14;
       r14 = -2;             DM(sweep_width) = r14;
       bit set ustat1 0x2F;                         /* turn on Flag8 LED */
       bit clr ustat1 0x10;
       dm(IOSTAT)=ustat1;
       jump done;

delay_settings_6:
       r14 = 6;              DM(sweep_rate) = r14;
       r14 = 0;              DM(sweep_width) = r14;
       bit set ustat1 0x1F;                         /* turn on Flag9 LED */
       bit clr ustat1 0x20;
       dm(IOSTAT)=ustat1;

done:
       rti(db);
       bit clr mode1 SRRFH;                         /* switch back to primary register set */
       nop;

/* ------------------------------------------------------------------------------------ */
/*                      IRQ2 Pushbutton Interrupt Service Routine                       */
/*                                                                                      */
/*     Intensifies the effect of the flanged sound to sound more metallic.              */
/*     Negative Feedback subtracts (inverts) the output of the fixed tap center output  */
/*     Positive Feedback adds the fixed tap center output of the flange delay line      */
/*                                                                                      */
/*     Default before 1st IRQ push:  Delay = 20.83 msec                                 */
/*     1st Pushbutton Press:  Feedback Setting #1 -                                     */
/*     2nd Pushbutton Press:  Feedback Setting #2 -                                     */
/*     3rd Pushbutton Press:  Feedback Setting #3 -                                     */
/*     4th Pushbutton Press:  Feedback Setting #4 -                                     */
/*     5th Pushbutton Press:  Feedback Setting #4 -                                     */
/*     6th Pushbutton Press:  Feedback Setting #4 -                                     */
/*     7th Pushbutton Press:  Reverts back to 1st Pushbutton Press                      */
/*                                                                                      */
/*      The pushbutton setting is shown by the inactive LED setting, all others are on     */
/*      (reverse of IRQ1 LED settings, which show lit LED for  setting #)                  */
/* ------------------------------------------------------------------------------------ */

select_flange_feedback_gain:
        bit set mode1 SRRFH;                /* enable background register file */
        NOP;                                /* 1 CYCLE LATENCY FOR WRITING TO MODE1 REGISER!!   */

        r13 = 6;                            /* number of presets */
        r15 = DM(IRQ2_counter);             /* get preset count */
        r15 = r15 + 1;                      /* increment preset */
        comp (r15, r13);
        if ge r15 = r15 - r15;              /* reset to zero */
        DM(IRQ2_counter) = r15;             /* save preset count */

        r10 = pass r15;                     /* get preset mode */
        if eq jump feedback_settings_2;     /* check for count == 0 */
        r10 = r10 - 1;
        if eq jump feedback_settings_3;     /* check for count == 1 */
        r10 = r10 - 1;
        if eq jump feedback_settings_4;     /* check for count == 3 */
        r10 = r10 - 1;
        if eq jump feedback_settings_5;     /* check for count == 4 */
        r10 = r10 - 1;
        if eq jump feedback_settings_6;     /* check for count == 4 */

feedback_settings_1:                        /* count therefore, is == 5 if you are here */
        /* no feedback */
        r14 = 0x00000000;      DM(feedback_gain) = r14;
        bit clr ustat1 0x3E;
        bit set ustat1 0x01;                /* turn off Flag4 LED */
        dm(IOSTAT)=ustat1;
        jump exit;

feedback_settings_2:
        /* add some small positive feedback */
        r14 = 0x20000000;      DM(feedback_gain) = r14;
        bit clr ustat1 0x3D;
        bit set ustat1 0x02;                /* turn off Flag5 LED */
        dm(IOSTAT)=ustat1;
        jump exit;

feedback_settings_3:
        /* add some small negative feedback */
        r14 = 0x5A82799A;      DM(feedback_gain) = r14;
        bit clr ustat1 0x3B;
        bit set ustat1 0x04;                /* turn off Flag6 LED */
        dm(IOSTAT)=ustat1;
        jump exit;

feedback_settings_4:
        /* add a medium amount of positive feedback */
        r14 = 0xA57D8666;      DM(feedback_gain) = r14;
        bit clr ustat1 0x37;
        bit set ustat1 0x08;                /* turn off Flag7 LED */
        dm(IOSTAT)=ustat1;
        jump exit;

feedback_settings_5:
        r14 = 0x67FFFFFF;      DM(feedback_gain) = r14;
        bit clr ustat1 0x2F;
        bit set ustat1 0x10;                /* turn off Flag8 LED */
        dm(IOSTAT)=ustat1;
        jump exit;

feedback_settings_6:
        r14 = 0x90000000;      DM(feedback_gain) = r14;
        bit clr ustat1 0x1F;
        bit set ustat1 0x20;
        dm(IOSTAT)=ustat1;                  /* turn off Flag9 LED */
exit:
        rti(db);
        bit clr mode1 SRRFH;                /* switch back to primary register set */
        nop;
.endseg;
```

3.3.2.2 Chorus Effect

Chorusing is used to "thicken" sounds. This time delay algorithm (between 15 and 35 milliseconds) is designed to duplicate the effect that occurs when many musicians play the same instrument and same music part simultaneously. Musicians are usually synchronized with one another, but there are always slight differences in timing, volume, and pitch between each instrument playing the same musical notes. This chorus effect can be re-created digitally with a variable delay line rotating around the tap center, adding the time-varying delayed result together with the input signal.

Using this digitally recreated effect, a 6-string guitar can also be 'chorused' to sound more like a 12-string guitar. Vocals can be thickened to sound like more than one musician is singing.

The chorus algorithm is similar to flanging, using the same difference equation, except the delay time is longer. With a longer delay-line, the comb filtering is brought down to the fundamental frequency and lower order harmonics (Figure 70). Figure 67 shows the structure of a chorus effect simulating 2 instruments [2, 6].

Figure 67. Implementation of a Chorus Effect Simulating 2 Instruments

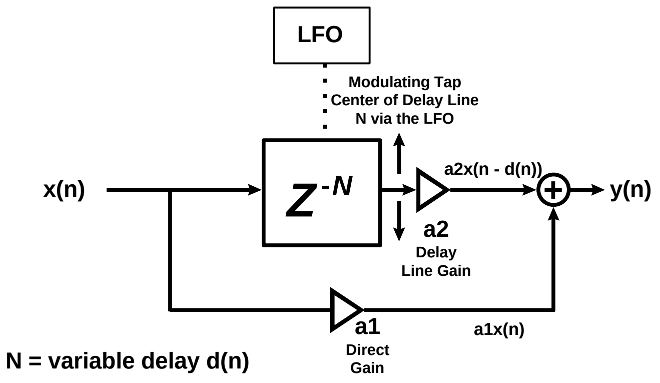
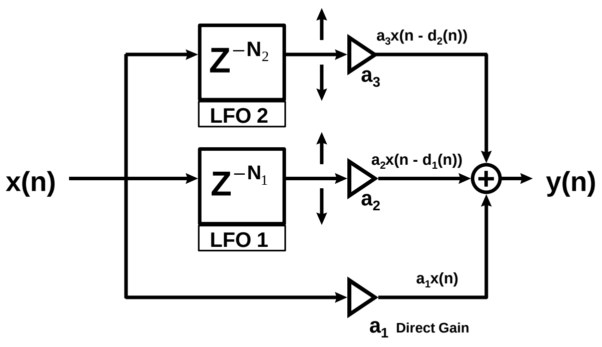


Figure 68. Chorus Effect Simulating 3 Instruments



Can use same LFO table with pointers 90 degrees apart.

The difference equation is for Figure 67 is:

$$y(n) = a_1 x(n) + a_2 x(n - d(n))$$

Some example difference equations for simulating 2 or 3 musicians are shown below.

An example fixed point fractional implementation is:

$$\boxed{y(n) = \frac{1}{2}x(n) + \frac{1}{2}x(n - d(n))}$$

Scaling each signal by ½ will equally mix both signal to around the same volume while ensuring no overflow when the signals are added.

To implement a chorus of 3 instruments, 2 variable delay lines can be used (Figure 68). Use a scaling factor of 1/3 to prevent overflow with fixed point math while mixing all three signals with equivalent gain.

$$\boxed{y(n) = \frac{1}{3}x(n) + \frac{1}{3}x(n - d_1(n)) + \frac{1}{3}x(n - d_2(n))}$$

Another Implementation Example as described by Orfanidis[2] is:

$$y(n) = \frac{1}{3}\left[\, x(n) + a_1(n)x(n - d1(n)) + a_2(n)x(n - d2(n)) \right]$$

where the gain coefficients a1(n) and a2(n) can be a low-frequency random number with unity mean.

The small variations in the delay time can be introduced by a *random LFO* at around 3 Hz. A low frequency random LFO lookup table (for example, Figure 69) can be used to recreate the random variations of the musicians, although the circular buffer will still be periodic.

Figure 69.
Example 4K Random LFO Wavetable Storage

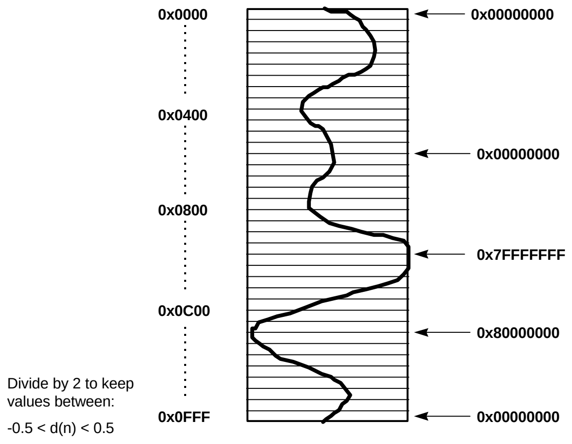
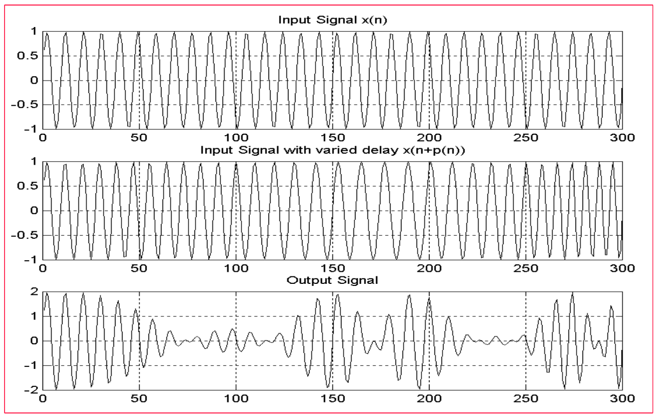


Figure 70.
Chorus Result of Adding a Variable Delayed Signal To It's Original

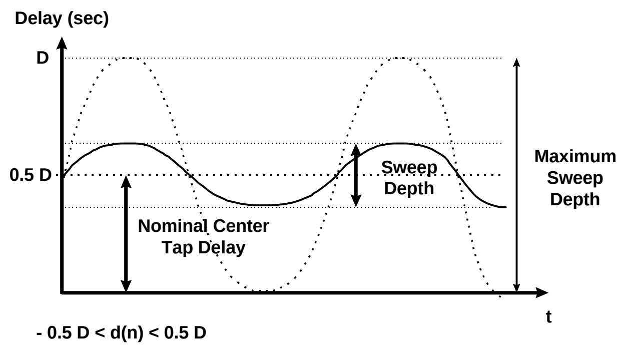


A result of an increasing slope in the LFO will cause the pitch to be lower. A negative slop will result in a pitch increase. The LFO value in the table can be updated on a sample basis via the chorus processing routine, or the wavetable look-up can be modified using the DSP's on-chip programmable timer. The varying delay d(n) will be updated using the following equation:

$$\mathbf{d(n) = D(0.5 + LFO(n))}, \quad \text{or} \quad \mathbf{d(n) = D(0.5 + v(n))}$$

The signal v(n) is described by Orfanidis [2] as a zero-mean low-frequency random signal varying between [-0.5,0.5]. An easy technique to ensure fixed point signals stay within this range would be to take a lookup table with fractional numbers ranging from -1 to 0.9999 and dividing each lookup value by 2.

Figure 71.
Chorus Delay Parameters



Maximum Sweep Depth < +/- 0.5D, or between 0 < d(n) < Dc

Sweep Rate - speed of wavetable playback, infuences pitch change

Chorus Parameters

Like the flanger, most units offer "*Modulation*" *(or Rate)* and "*Depth*" controls.

- *Depth (or Delay)*

controls the length of the delay line, allowing a user to change the length on-the-fly.

- *Sweep Depth*

Determines how much the time offset changes during an LFO cycle. It combined with the delay line value for a total delay used to process the signal.

- *Modulation*

The variations in delay time will be introduced by a **low-frequency oscillator (LFO)**. This frequency can usually be controlled with the "*Sweep Rate*" parameter. Usually, the LFO consists of a low frequency random signal. When the waveform is at the largest value, variable delay that results will be the maximum delay possible. A result of an increasing slope in the LFO will cause the pitch to be lower. A negative slop will result in a pitch increase.

Sine and Triangle waves can be used to vary the delay time. One easy method for generating the modulation value is through a wavetable lookup. The value in the table can be modified on a sample basis via the chorus routine, or the lookup can be determined using the DSP's on-chip programmable timer. When the timer count expired and the DSP vectors off to the Timer Interrupt Service Routine, the modulation value can then be updated with the next value in the waveform buffer. The LFO can be repeated continuously by making the wavetable a circular buffer. Using a cosine wavetable, the varying delay d(n) will be updated using the following equation:

.

$$d(n) = D(0.5 + LFO(2p\, n f_{Delay}))$$

where D = Delay Line Length
Fdelay = Frequency of the LFO with a period of 2 Pi of the LFO
n = the nth location in the wavetable lookup

The small variations in the time delays and amplitudes can also be simulated by varying them **randomly** at a very low frequency around 3 Hz.

$$d(n) = D(0.5 + v(n))$$

where,
v(n)= current variable delay value from the random LFO generator

or,

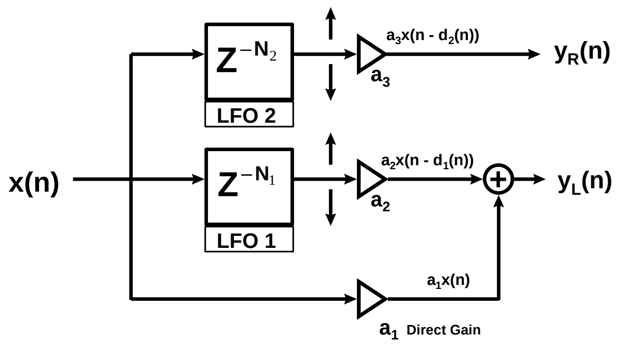
d(n) = D(0.5 + random_LFO_number(n))

The signal v(n) as described by Orfanidis [2] is a zero-mean low-frequency random signal varying between [-0.5,0.5]. An easy technique to ensure fixed point signals stay within this range would be to take a lookup table with fractional numbers ranging from -1 to 0.9999 and dividing each lookup value by 2. This can easily be done with an arithmetic shift instruction shifting the input to the right by 1 binary place to divide the lookup value by 2.

Stereo Chorus

This effect is achieved by panning the chorus result on back and forth on both stereo channels, creating the impression of movement of the sound in space. The effect can also be created by sending the unaltered input signal on one output stereo channel and the chorused result to the opposite channel.

Example Stereo Chorus Effect



Can use same LFO table with pointers 90 degrees apart.

Figure 72.

Flanging/Chorusing Similarities and Differences

Both Flanging and Chorusing use variable buffers to change the time delay on the fly. Both effects achieve these variations in delay time by using a low frequency oscillator (LFO). This parameter is available on commercial units as the “sweep rate”. The “sweep-depth” parameter is what determines the amount of delay in the sweep period. The greater the depth, the farther the peaks and dips of the phase cancellation.

The key difference between the two effects is the flanger found in many commercial units changes the delay using a low frequency sine-wave generator, where the chorus usually changes the delay using a low-frequency random noise generator. In addition, the flanger modulates the length of the delay from 0 to D, while the chorus modulates the delay from ???? (expand further)

Example Stereo Chorus Implementation of 3 instruments Using the ADSP-21065L

```
/*********************************************************************************************
 STEREO_CHORUS_FEEDBACK.ASM - stereo chorusing effect simulating 3 voices/musical instruments
       (SPORT1 Rx ISR count & update method - delay calculation determined by a counter
       incremented in the serial port's audio processing routine)

       Chorus Effect as Described by:
       1. Jon Dattorro in "Effect Design Part 2 - Delay-Line Modulation and Chorus,"
          J. Audio Eng. Socienty, Vol. 45, No. 10, October 1997

       2. Eq(8.2.20) of Introduction to Signal Processing.
          By Sophocles J. Orfanidis - 1996, Prentice-Hall
          ISBN 0-13-209172-0

       This version uses Linear Interpolation (versus Allpass Interpolation) along with
       integer (not fractional) sample delay.  Since the sample rate is 48K, for most
       lower bandwidth signals, Linear Interpolation is probably adequate for most
       instruments.

        I/O equations:
              yL(n) = 1.0 * x(n) + 0.7071*x(n - d1(n)) - 0.7071*x(n - D1/2)
                yR(n) = 1.0 * x(n) + 0.7071*x(n - d2(n)) - 0.7071*x(n - D2/2)

        x(n) -----O------------------------------------------|>-------->O--------> yL(n)
                  ^ -      |                        ^       0.7071      ^
                  |        |                       /                    |
                  |        |             _________/_____                |
                  |        |            |               |  Rotating Tap |
                  |        |            |   Z^(-D1)     |   Center      |
                  |        |----------->|               |-----------|>--|
                  |                     |     d1(n)     |         0.7071
                  |                     |               |
                  |                     |_______________|
                  |                          /    |
                  |                         /     |  Fixed Tap
                  |                        /      |   Center
                  |           feedback1           |   D1 / 2
                  |-------------<|---------------|
                  |
                  O------------------------------------------|>-------->O--------> yR(n)
                  | -      |                        ^        0.7071     ^
                  |        |                       /                    |
                  |        |             _________/_____                |
                  |        |            |               |  Rotating Tap |
                  |        |            |   Z^(-D2)     |   Center      |
                  |        |----------->|               |-----------|>--|
                  |                     |     d2(n)     |         0.7071
                  |                     |               |
                  |                     |_______________|
                  |                          /    |
                  |                         /     |  Fixed Tap
                  |                        /      |   Center
                  |           feedback2           |   D2 / 2
                  |-------------<|---------------|



       What the Chorusing Effect does?
       Chorusing simulates the effect of multiple instruments/voices playing the same musical
       arrangement at the same time.  In actual concert situations, musicians are usually
       synchronized together, except for small variations in amplitude and timing.
       This effect is achieved by allowing the time delay (and also amplitude) to vary
       randomly or sinusoudally in time by using a random number generation routine or sine
       wavetable.


        For each input sample, the sample processing algorithm does the following:
                       store input sample s0 to 2 delay lines
                       modify wavetable(when necessary)
                       generated variable delay, d = D * (0.5 + randnum(fc*t))
                       s1 = sample1 = tap(D, w1, p1, d)
                       s2 = sample2 = tap(D/2)
                       y = a0 * s0 + a1 * s1 - af * s2
       Developed for the 21065L EZ-LAB Evaluation Board
   ********************************************************************************************/

/* ADSP-21060 System Register bit definitions */
#include        "def21065l.h"
#include        "new65Ldefs.h"

.GLOBAL         chorus_effect;
.GLOBAL         Init_Chorus_Buffers;
.GLOBAL         change_depth_rate_width;
.GLOBAL         select_feedback_gain;

.EXTERN         Left_Channel;
.EXTERN         Right_Channel;

/* Chorus Control Parameters */
#define a0L             0x7FFFFFFF              /* a0 = 0.99999999, left input gain */
#define a1L             0x5A82799A              /* a1 = 0.707106781, left output gain of tapped delay
                                                   line */
#define a0R             0x7FFFFFFF              /* a0 = 0.99999999, right input gain */
#define a1R             0x5A82799A              /* a1 = 0.707106781, left output gain of tapped delay
                                                   line */
#define afL             0x5A82799A              /* a0 = 0.707106781, negative feedback gain */
#define afR             0x5A82799A              /* a0 = 0.707106781, negative feedback gain */
                                                /* = 0xA57D8666 for positive feedback, - 0.707106781 */
#define D1              870                     /* Depth, or TD = D1/fs = 870/48000 = 18 msec */
#define D2              1120                    /* Depth, or TD = D2/fs = 1120/48000 = 23 msec */
#define DepthL          D1                      /* DepthL is equivalent to time delay of signal, or
                                                   d-line 1 size */
#define DepthR          D2                      /* DepthR is equivalent to time delay of signal, or
                                                   d-line 2 size */
#define L_Del_TapCenter       DepthL/2          /* D1/2, used for add & subtracting delay from
                                                   nominal tap center */
#define R_Del_TapCenter       DepthR/2          /* D2/2, used for add & subtracting delay from
                                                   nominal tap center */
#define WaveSize        4000                    /* semi-random/sinusoidal wavetable size*/

/* chorus control parameters */
#define c               1       /* wavetable signal freq fc1 = c1*(update time)*WaveSize = 4 Hz */
#define chorus_width  -2        /* Enter # from 1 to 31.  Do not enter 0, to keep #'s +/- 0.5 */
#define modulation_rate 80      /* Update wavetable pointer for delay calc every 40 interrupts */
                                /* sine wavetable has a rate of modulation of about 0.15 Hz */
/* playing with the width, modulation rate, depth size  and feedback affects
   the intensity of the chorus'ed sound */


.segment /dm     dmchorus;

.var    IRQ1_counter = 0x00000004;
.var    IRQ2_counter = 0x00000004;

.var    feedback_gainL = afL;
.var    feedback_gainR = afR;
.var    sweep_rate = modulation_rate;
.var    sweep_widthL = chorus_width;            /* controls width of left sweep, ranges from -1 to - 15 */
.var    sweep_widthR = chorus_width;            /* controls width of right sweep, ranges from -1 to - 15 */

.var    w1[DepthL + 1];                         /* delay-line buffer 1, max delay = D1 */
.var    w2[DepthR + 1];                         /* delay-line buffer 2, max delay = D2 */
.var    excursion_valueL;                       /* wavetable offset value for delay calculation */
.var    excursion_valueR;                       /* wavetable offset value for delay calculation */
.var  random[WaveSize]="sinetbl.dat";           /* store one period of the wavetable */
                                                /* minimum frequency of table =
                                                   (freq of delay update)/WaveSize = 8/4 = 2 Hz */
.var    wavetbl_counter = 0x00000000;

.endseg;


/* -------------PROGRAM MEMORY CODE---------------------------------*/
.segment /pm pm_code;

Init_Chorus_Buffers:
       B2 = w1;  L2 = @w1;           /* left delay-line buffer pointer and length */
       m2 = 1;

       LCNTR = L2;                   /* clear left delay line buffer to zero */
        DO clrDlineL UNTIL LCE;
clrDlineL:         dm(i2, m2) = 0;

       B3 = w2;  L3 = @w2;           /* right delay-line buffer pointer and length */
       m3 = 1;

       LCNTR = L3;                   /* clear right delay line buffer to zero */
        DO clrDlineR UNTIL LCE;
clrDlineR:         dm(i3, m3) = 0;


       B6 = random;                  /* left channel pointer for signal generator */
       L6 = @random;                 /* get size of table lookup */
       B7 = random;                  /* right channel pointer for signal generator */
       I7 = random + 1000;           /* offset 90 degrees so modulators in quadradure phase */
       L7 = @random;                 /* get size of table lookup */

       RTS;

/* -------------------------------------------------------------------------------*/
/*                                                                               */
/*                    Digital Chorus Routine - process both channels together     */
/*                                                                               */
/* ------------------------------------------------------------------------------ */

chorus_effect:
       /* combine both left and right input samples together into 1 signal */
       r0 = dm(Left_Channel);        /* left input sample */
       r1 = dm(Right_Channel);       /* right input sample */
       r0 = ashift r0 by -1;         /* scale signal by 1/2 for equal mix */
       r1 = ashift r1 by -1;         /* scale signal by 1/2 for equal mix */
       r15 = r0 + r1;                /* 1/2xL(n) + 1/2 xR(n) */

test_wav_update:
       /* update sine value from lookup table? Update every 80 SPORT rx interrupts */
       /* sweep frequency = 80 * c * 4000 / fs = 96000 /48k = .15 sec */
       r11 = DM(sweep_rate);         /* count up to 80 interrupts (default) */
       r10 = DM(wavetbl_counter);    /* get last count from memory */
       r10 = r10 + 1;                /* increment preset */
       comp (r10, r11);              /* compare current count to max count */
       if ge r10 = r10 - r10;        /* if count equals max, reset to zero and start over */
       DM(wavetbl_counter) = r10;    /* save updated count */

       r12 = pass r10;               /* test for wave count 0? */
       if eq jump update_wavetbl_ptrs;

       /* if you are here, reuse same random values for now */
       jump do_stereo_chorus;

       /* if necessary, calculate updated pointer to wavetables */
update_wavetbl_ptrs:
       m6 = c;                       /* desired increment c - frequency f = c x fs / D */
       r1 = DepthL;                  /* Total Delay Time */
       r2 = dm(i6, m6);              /* get next value in wavetable */
       r4 = dm(sweep_widthL);
       r2 = ashift r2 by r4;         /* divide by at least 2 to keep 1.31 #s between 0.5 and -0.5 */
       r3 = r1 * r2 (SSFR);          /* multiply Delay 1 by a fractional value from 0 to 0.5 */
       dm(excursion_valueL) = r3;

       m7 = c;                       /* desired increment c - frequency f = c x fs / D */
       r1 = DepthR;                  /* Total Delay Time */
       r2 = dm(i6, m6);              /* get next value in wavetable */
       r4 = dm(sweep_widthR);
       r2 = ashift r2 by r4;         /* divide by at least 2 to keep 1.31 #s between 0.5 and -0.5 */
       r3 = r1 * r2 (SSFR);          /* multiply Delay 1 by a fractional value from 0 to 0.5 */
       dm(excursion_valueR) = r3;

do_stereo_chorus:
       r3 = dm(excursion_valueL);    /* calculate time-varing delay for 2nd voice */
       r1 = L_Del_TapCenter;         /* center tap for delay line */
       r4 = r1 + r3;                 /*   r4 = d(n) = D1/2 + D1 * random(fc*t)   */
```

```
process_left_ch:
       /* r4 now will be used to set m2 register to fetch time varying delayed sample */
       m2 = r4;                            /* tap outputs of circular delay line */
       modify(i2, m2);                     /* go to delayed sample */
       r4 = -r4;                           /* negate to get back to where we were */
       m2 = r4;                            /* used to post modify back to current sample */
       r9 = dm(i2, m2);                    /* get time-varying delayed sample 1 */

       /* r8 will be used to get Nominal Tap Center delayed sample for flange feedback */
       m2 = L_Del_TapCenter;               /* tap outputs of circular delay line */
       modify(i2, m2);                     /* go to delayed sample */
       m2 = -L_Del_TapCenter;              /* negate to get back to where we were */
       r7 = dm(i2, m2);                    /* get delayed sample, postmodify back to current sample */

       /* crank out difference equation */
       r8 = a0L;                           /* left input gain */
       mrf = r8 * r15 (SSF);               /*   mrf = a0L * x-input   */
       r8 = dm(feedback_gainL);            /* left gain for feedback of nominal tap center*/
       mrf = mrf - r8 * r7 (SSF);          /*   mrf = a0L * x - afL * sNominalL   */
       r12 = mr1f;                         /* save for input to chorus left delay line */

       r8 = a1L;                           /* left delay line output gain */
       mrf = mrf + r8 * r9 (SSFR);         /*  mrf = a0L * xL + a1L * s1L - afL * sNominalL   */
       mrf = SAT mrf;                      /* saturate result if necessary */
       r10 = mr1f;                         /* chorus result in r10 */

       /* put 'input minus feedback' sample from r12 into tap-0 of delay line */
       /* and backshift circular delay-line buffer pointer */
       dm(i2, -1) = r12;

       /* write chorus'ed output sample to left output channel */
       dm(Left_Channel)  = r10;            /* left output sample */

process_right_ch:
       r3 = dm(excursion_valueR);          /* calculate time-varing delay for 2nd voice */
       r1 = R_Del_TapCenter;               /* center tap for delay line */
       r4 = r1 + r3;                       /*   r4 = d(n) = D2/2 + D2 * random(fc*t)   */

       /* r4 now will be used to set m3 register to fetch time varying delayed sample */
       m3 = r4;                            /* tap outputs of circular delay line */
       modify(i3, m3);                     /* go to delayed sample */
       r4 = -r4;                           /* negate to get back to where we were */
       m3 = r4;                            /* used to post modify back to current sample */
       r9 = dm(i3, m3);                    /* get time-varying delayed sample 1 */

       /* r8 will be used to get Nominal Tap Center delayed sample for chorus feedback */
       m3 = R_Del_TapCenter;               /* tap outputs of circular delay line */
       modify(i3, m3);                     /* go to delayed sample */
       m3 = -R_Del_TapCenter;              /* negate to get back to where we were */
       r7 = dm(i3, m3);                    /* get delayed sample, postmodify back to current sample */

       /* crank out difference equation */
       r8 = a0R;                           /* left input gain */
       mrf = r8 * r15 (SSF);               /*   mrf = a0R * x-input   */
       r8 = dm(feedback_gainR);            /* gain for feedback of nominal tap center*/
       mrf = mrf - r8 * r7 (SSF);          /*   mrf = a0R * xR - afR * sNominalR   */
       r12 = mr1f;                         /* save for input to chorus right delay line */

       r8 = a1R;                           /* right delay line output gain */
       mrf = mrf + r8 * r9 (SSFR);         /*   mrf = a0R * x + a1R * s1R - af * sNominalR   */
       mrf = SAT mrf;                      /* saturate result if necessary */
       r10 = mr1f;                         /* chorus result in r10 */

       /* put 'input minus feedback' sample from r12 into tap-0 of delay line */
       /* and backshift circular delay-line buffer pointer */
       dm(i3, -1) = r12;

       /* write chorus'ed output sample to right output channel */
       dm(Right_Channel) = r10;            /* right output sample */

       rts;
```

3.3.2.3 Vibrato

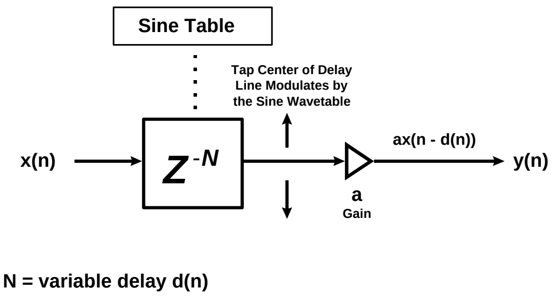
The **vibrato** effect that duplicates 'vibrato' in a singer's voice while sustaining a note, a musician bending a stringed instrument, or a guitarist using the guitars 'whammy' bar. This effect is achieved by evenly modulating the pitch of the signal. The sound that is produced can vary from a slight enhancement to a more extreme variation. It is similar to a guitarist moving the 'whammy' bar, or a violinist creating vibrato with cyclical movement of the playing hand. Some effects units offered vibrato as well as a tremolo. However, the effect is more often seen on chorus effects units.

The slight change in pitch can be achieved (with a modified version of the chorus effect) by varying the depth with enough modulation to produce a pitch oscillation. This is accomplished by changing the modify value of the delay-line pointer on-the-fly, and the value chosen is determined by a lookup table. This results in the interpolation/decimation of the stored samples via rotating the center tap of the delay line. The stored 'history' of samples are thus played back at a slower, or faster rate, causing a slight change in pitch.

To obtain an even variation in the pitch modulation, the delay line is modified using a sine wavetable. Note that this a stripped down of the chorus effect, in that the direct signal is not mixed with the delay-line output.

This effect is often confused with 'tremolo', where the *amplitude* is varied by a LFO waveform. The tremolo and vibrato can both be combined together with a time-varying LPF to produce the effect produced by a rotating speaker (commonly referred to a 'Leslie' Rotating Speaker Emulation). An example rotating speaker emulation effect is also shown in this section.

Figure 73.
Implementation of the Vibrato Effect



Can use chorus or flanger algorithm, but no mix of input signal is required

Example Vibrato Implementation Using The ADSP-21065L

```
/* ------------------------------------------------------------------------------------------ */
/*                                                                                            */
/*                      Digital Vibrato Routine - process both channels together             */
/*                                                                                            */
/* ------------------------------------------------------------------------------------------ */

vibrato_effect:
       /* combine both left and right input samples together into 1 signal */
       r0 = dm(Left_Channel);              /* left input sample */
       r1 = dm(Right_Channel);             /* right input sample */
       r0 = ashift r0 by -1;               /* scale signal by 1/2 for equal mix */
       r1 = ashift r1 by -1;               /* scale signal by 1/2 for equal mix */
       r2 = r0 + r1;                       /* 1/2xL(n) + 1/2 xR(n) */

test_sine_update:
       /* update sine value from lookup table? Update every 12 SPORT rx interrupts */
       /* sweep rate = 12 * sin_inc * 4000 / fs = 48000 /48k = 1 sec */
       r11 = DM(modulation_rate);          /* count up to 24 interrupts */
       r10 = DM(wavetbl_counter);          /* get last count from memory */
       r10 = r10 + 1;                      /* increment preset */
       comp (r10, r11);                    /* compare current count to max count */
       if ge r10 = r10 - r10;              /* if count equals max, reset to zero and start over */
       DM(wavetbl_counter) = r10;          /* save updated count */
```

```
        r12 = pass r10;                      /* test for wave count 0? */
        if eq jump update_sinetbl_ptr;

        jump do_vibrato;                     /* if you are here, reuse same sine value for now */
                                             /* dm(sine_value) remains unchanged */

        /* if necessary, calculate updated pointer to sine wavetable */
update_sinetbl_ptr:
        m6 = dm(sin_inc);                    /* desired increment sin_inc - f = sin_inc x fs / D */
        r6 = D;
        r7 = dm(i6, m6);
        r4 = dm(pitch_bend);
        r7 = ashift r7 by r4;                /* divide by 2 to keep #s between 0.5 and -0.5 */
        r8 = r6 * r7 (SSFR)                  /* delay multiplication factor */
        dm(sine_value) = r8;                 /* save to current sine value to be reused above */

do_vibrato:
        r3 = dm(sine_value);                 /* get previous or newly updated sine value */
        r1 = D2;                             /* get nominal tap center delay */
        r4 = r1 + r3;                        /* r4 = d(n) = D/2 + D * sin(fc*t) */

        /* r4 now will be used to set m2 register to fetch time varying delayed sample */
        m2 = r4;                             /* set tap valud for output of circular delay line */
        modify(i2, m2);                      /* go to delayed sample address */
        r4 = -r4;                            /* negate to get back to where we were */
        m2 = r4;                            /* set up to go back to current mem location after access */
        r10 = dm(i2, m2);                    /* get delayed sample */

        /* put input sample from r2 into tap-0 of delay lines */
        /* and backshift pointer & update circular delay-line buffer*/
        dm(i2, -1) = r2;

        /* write vibrato output sample to AD1819A DAC channels */
        dm(Left_Channel)  = r10;             /* left output sample */
        dm(Right_Channel) = r10;             /* right output sample */

        rts;
```

21065L Rotating Speaker Emulation Implementation (Vibrato & Tremolo Combo)

```
/***   ROTATING_SPEAKER.ASM   *************************************************************
*                                                                                         *
*       AD1819A/ADSP-21065L EZLAB Rotating Speaker Emulation Effect Program               *
*       Developed using ADSP-21065L EZ-LAB Evaluation Platform                            *
*                                                                                         *
*        I/O Equation:  y(n) = [x(n)*sin(2*pi*fc*t)] convolve [ x(n-d(n)) ]               *
*                                                                                         *
*                                                                                         *
*                                               ^                 ^                       *
*              sin(2*PI*fc*t)                  /                 /                        *
*                    |             _________/_____         ___/____                       *
*                    |            |               |       |        |                      *
*                    v            |   Z^(-D)      |       |  LPF   |                      *
*      x(n) -------->O------------>|               |-------|  H(z)  |---> y(n)            *
*                    X            |     d(n)      |       |        |                      *
*                                 |               |       |________|                      *
*                                 |_______________|      /  Variable                      *
*                                      / Rotating Tap   /   Cutoff                        *
*                                     /   Center            Freq                          *
*                                    /                                                    *
*                                                                                         *
*                                                                                         *
*                                                                                         *
*       What the rotating speaker effect does?                                            *
*       -------------------------------------                                             *
*       A popular effect used on keyboards and Hammond organs. This effect                *
*       consists of a combination of the tremolo and vibrato effects, combined            *
*       a low-pass filter with a variable cutoff frequency.  This combination             *
*       simulates what was done by audio engineers who would actually produce             *
*       this effect by rotating a speaker cabinet.  The resulting sound heard             *
*        by the ears is a cyclical increase/decrease in volume, a doppler effect          *
*        from the speaker rotating, and a cyclical drop in high frequencies               *
*       whenever the speaker is facing away from the listener.                         *
*                                                                                      *
****************************************************************************************/

/* ADSP-21065L System Register bit definitions */
#include       "def21065l.h"
#include       "new65Ldefs.h"

.EXTERN        Left_Channel;
.EXTERN        Right_Channel;

.GLOBAL        Init_Tremolo_Vibrato_Buffers;
.GLOBAL        Rotating_Speaker_Effect;
.GLOBAL        change_speaker_rotate_rate;
.GLOBAL        select_tremolo_effect;

.segment /dm    rotspeak;

#define WaveTableSize  4000           /* sinusoidal wavetable, 1 period in table of 4000 sine wave
elements  */
#define D             2000           /* Depth, or TD = D/fs = 1000/48000 = 20.83 msec */
                                     /* increasing depth size D increases pitch variation */
#define Depth         D          /* Depth is equivalent to time delay of a signal, or delay-line size */
#define D2            Depth/2        /* D/2, used for addig & subtracting delay from tap center */


.var   IRQ1_counter = 0x00000003;
.var   IRQ2_counter = 0x00000000;
.var   wavetbl_counter = 0x00000000;
.var   Effect_Ctrl = 0x00000001;     /* memory flag that determines which tremolo routine executes */
.var   pitch_bend = -6;              /* Enter # from -1 to -15.  Do not enter 0.*/
                                     /* -1 to -5  - large pitch bend*/
                                     /* -6 to -10 - medium pitch bend */
                                     /* -11 to -15 - small pitch bend */
.var   sin_inc = 2;                  /* wavetable signal frequency ftable = sin_inc * fs = ? Hz */
.var   modulation_rate = 3;          /* controls how fast the wavetable is updated in the SPORT1 rx ISR
*/
.var   sine_value;                   /* used for tremolo control */
.var   excursion_value;              /* used for vibrato delay offset calculation */
.var    sine[WaveTableSize] = "sinetbl.dat";
.var   w[D + 1];                     /* delay-line buffer, max delay = D */

.endseg;

/*------------------------------------------------------------------------------*/

.segment /pm pm_code;

/*-------------------------------------------------------------------------------------*/
Init_Tremolo_Vibrato_Buffers:
       B2 = w;  L2 = @w;              /* delay-line buffer pointer and length */
       m2 = 1;

       LCNTR = L2;                    /* clear delay line buffer to zero */
        DO clrDline UNTIL LCE;
clrDline:         dm(i2, m2) = 0;

       B6 = sine;                    /* pointer for signal generator */
       L6 = @sine;                   /* get size of sine table lookup */

       RTS;

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

                      ROTATING SPEAKER (Vibrato & Tremolo Combo) AUDIO EFFECT

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

Rotating_Speaker_Effect:
       r1 = DM(Left_Channel);
       r1 = ashift r1 by -1;
       r2 = DM(Right_Channel);
       r2 = ashift r2 by -1;
       r3 = r2 + r1;
```

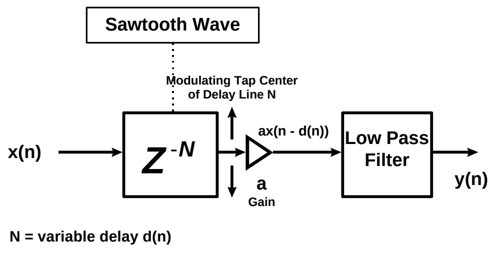
```
        /* generate sine value from wavetable generator if necessary, where r4 = sin(2*pi*fc*t)   */
test_wavtbl_update:
        /* update sine value from lookup table? Update every 80 SPORT rx interrupts */
        /* sweep frequency = 80 * c * 4000 / fs = 96000 /48k = .15 sec */
        r11 = DM(modulation_rate);           /* count up to 80 interrupts */
        r10 = DM(wavetbl_counter);           /* get last count from memory */
        r10 = r10 + 1;                       /* increment preset */
        comp (r10, r11);                     /* compare current count to max count */
        if ge r10 = r10 - r10;               /* if count equals max, reset to zero and start over */
        DM(wavetbl_counter) = r10;           /* save updated count */

        r12 = pass r10;                              /* test for wave count 0? */
        if eq jump update_wavetbl_ptr;

        /* if you are here, reuse same sine value for now.. dm(sine_value) remains unchanged */
        jump do_vibrato;

        /* if necessary, calculate updated pointer to wavetable */
update_wavetbl_ptr:
        m6 = dm(sin_inc);                    /* desired increment sin_inc - frequency f = sin_inc x fs /
D */
        r7 = dm(i6, m6);                     /* get next value from sine lookup table */
        dm(sine_value) = r7;                 /* use for tremolo_effect amplitude scaling factor */
        r4 = dm(pitch_bend);                 /* controls scaling of center tap delay offset */
        r7 = ashift r7 by r4;                /* divide by at least 2 to keep #s between 0.5 and -0.5 */
        r6 = D;                              /* Total Delay Time D = amplitude of sine wave lookup */
        r8 = r6 * r7 (SSFR);                 /* delay multiplication factor */
        dm(excursion_value) = r8;            /* save to current sine value to be reused above */

do_vibrato:
        r2 = dm(excursion_value);            /* get previous or newly updated scaled sine value */
        r1 = D2;                             /* get nominal tap center delay */
        r4 = r1 + r2;                        /* r4 = d(n) = D/2 + D * sin(fc*t) */

        /* r4 now will be used to set m2 register to fetch time varying delayed sample */
        m2 = r4;                             /* set tap valud for output of circular delay line */
        modify(i2, m2);                              /* go to delayed sample address */
        r4 = -r4;                            /* negate to get back to where we were */
        m2 = r4;                             /* set up to go back to current location after memory access
*/
        r10 = dm(i2, m2);                    /* get delayed sample */

        /* write vibrato output sample to r0 for tremolo routine */
        r0 = r10;

        /* put input sample from r2 into tap-0 of delay lines */
        /* and backshift pointer & update circular delay-line buffer*/
        dm(i2, -1) = r3;

which_tremolo_routine:
        r4 = DM(Effect_Ctrl);
        r4 = pass r4;
        if eq jump mono_tremolo_effect;              /* if == 1, execute mono tremolo routine */
                                             /* otherwise, execute stereo tremolo routine */
stereo_tremolo_effect:
        /* get generated sine value from wavetable generator, where r4 = sin(2*pi*fc*t)   */

        r4 = dm(sine_value);
        r5 = r0 * r4 (SSFR);
        /* test current sine value to pan left or right, if + then pan left, if - then pan right */
        r4 = pass r4;
        IF LE JUMP (pc, pan_right_channel);

pan_left_channel:
        /* write tremolo result sample to left/right output channels */
        DM(Left_Channel) = r5;
        r6 = 0x00000000;
        DM(Right_Channel) = r6;
        JUMP (pc, tremolo_done);

pan_right_channel:
        /* write tremolo result sample to left/right output channels */
        r6 = 0x00000000;
        DM(Left_Channel) = r6;
        DM(Right_Channel) = r5;
```

```
tremolo_done:
       rts;

/* ------------------------------------------------------------------------------------ */

mono_tremolo_effect:
       /* get generated sine value from wavetable generator, where r4 = sin(2*pi*fc*t)  */
       r4 = dm(sine_value);
       r5 = r0 * r4 (SSFR);

       /* write tremolo result sample to left/right output channels */
       dm(Left_Channel)  = r5;                    /* left output sample */
       dm(Right_Channel) = r5;                    /* right output sample */

       RTS;

/* ------------------------------------------------------------------------------------ */
```

3.3.2.4 Pitch Shifter

An interesting and commonly used effect is changing the pitch of an instrument or voice. The algorithm that can be used to implement a pitch shifter is the chorus or vibrato effect. The chorus routine is used if the user wishes to include the original and pitch shifted signals together. The vibrato routine can be used if the desired result is only to have pitch shifted samples, which is often used by TV interviewers to make an anonymous persons voice unrecognizable. The only difference from these other effects is the waveform used for delay line modulation. The pitch shifter requires using a sawtooth/ramp wavetable to achieve a 'linear' process of dropping and adding samples in playback from an input buffer. The slope of the sawtooth wave as well as the delay line size determines the amount of pitch shifting that is performed on the input signal.

Figure 74.
Implementation of a Generic Pitch Shifter



The audible side effect of using the 2 instrument chorus algorithm (with one delay line) is the ‘clicks’ that are produced whenever the delay pointer passes the input signal pointer when samples are added or dropped. This is because output pointer is moving through the buffer at a faster/slower rate than the input pointer, thus eventually causing an overlap. To reduce or eliminate this undesired artifact cross-fading techniques can be used between two alternating delay line buffers with a windowing function, so when one of delay line output pointers are close to the input, a zero crossing will occur at the overlay to avoid the 'pop' that is produced. For higher pitch shifted values, there is a noticable 'warble' audio modulation produced as a result of the outputs of the delay lines being out of phase, which causes periodic cancellation of frequencies to occur. Methods to control the delay on-the-fly to prevent the phase cancellations have been proposed, but are not implemented in our reference examples. A basic 21065L assembly example of the pitch shifter used as a Detune Effect is shown in the next section.

Again, the DSP timer can update the delay-line retrieval address value of a previous sample which results in a linear adding and dropping of samples. Using a positive or negative slope determines if the pitch of an audio signal will be shifted up or down (see diagram below). The same look-up table can be used to pitch shift up or down. The address generation unit only needs to use an increment or decrement modify register that will move forward or backwards in the table. Multiple harmonies can be created by having multiple pointers with positive and negative address modify values circling through the table.

Figure 75.
Example Two-Voice Pitch Shifter Implementation

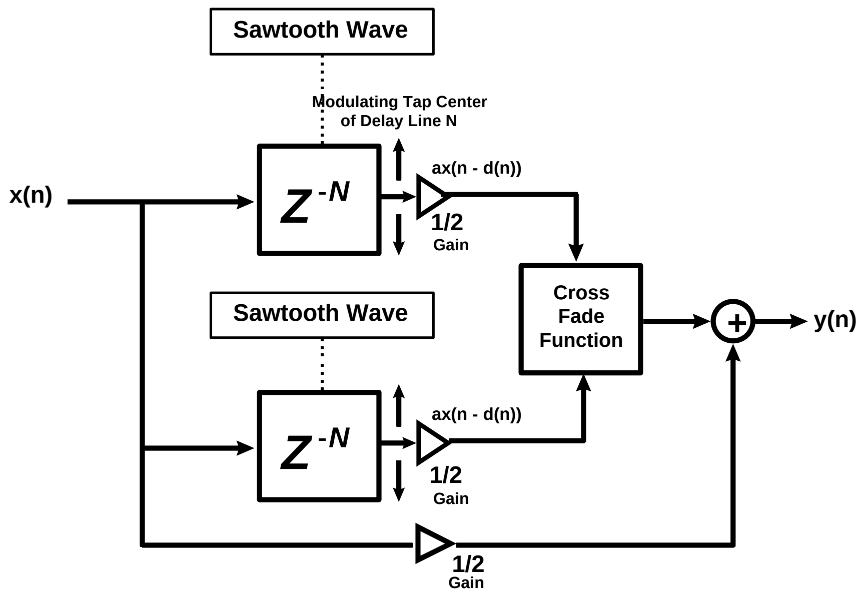
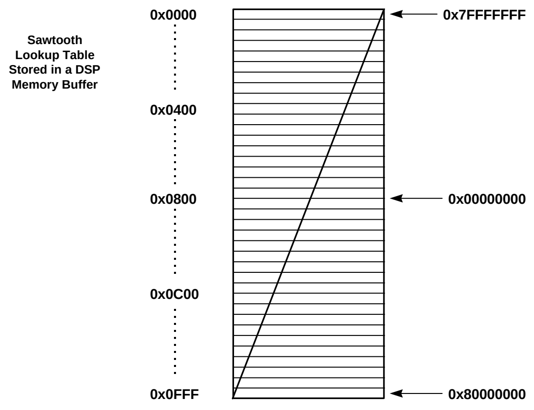
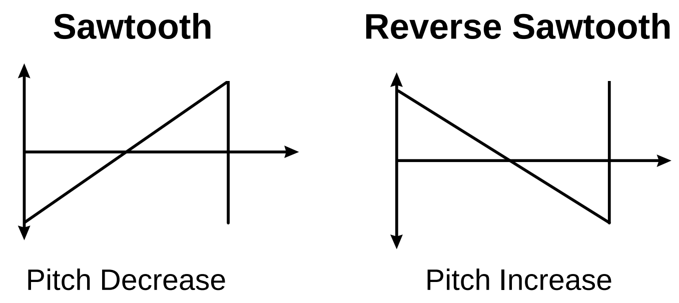


Figure 76.

Example 4K Sawtooth Wavetable





3.3.2.5 Detune Effect

The Detune Effect is actually a version of the Pitch Shifter. The pitch shifted result is set to vary from the input by about +/- 1% of the input frequency. This is done by setting the Pitch Shift factor to 0.99 or 1.01 The effect's result is to increase or decrease the output and combine the pitch shift with the input to vary a few Hz, resulting in an ‘out of tune effect’. (The algorithm actually uses a version of the chorus effect with a sawtooth to modulate the delay-line). Small pitch scaling values produce a ‘chorus like’ effect and imitates two instruments slightly out of tune. This effect is useful on vocal tracks to give impression of 2 musicians singing the same part using 1 person's voice. The pitch shifting result is to small for the formant frequencies of the vocal track to be affected, so the shifted voice still sounds realistic.

For a strong Detune Effect, vary the pitch by 5-10 Hz
For a weak Detune Effect ('Sawtooth Chorus' sound), vary the pitch by 2-3 Hz

21065L Example Detune Effect - Slight Pitch Shift Variation of an Input Signal

```
/* ***********************************************************************************************
       DETUNE_EFFECT.ASM - DETUNE.ASM - Pitch Shift Effect Simulating 2 Voices Slightly Out Of Pitch
       (Timer0 Update Method - delay calculation determined by the on-chip programmable timer)

   x(n) ----------------------------------------|>------------------------------->O----> y(n)
                |                         ^       1/2                           + ^
                |                        /                                        |
                |              _________/______                                   |
                |             |              |                                    |
                |             |     Z^(-D)   |                                    |
                |------------>|              |-----------|>-------- |             |
                |             |     d1(n)    |          1/2         |             |
                |             |              |                      |             |
                |             |______________|                      |             |
                |                  /                             +  v             |
                |                 /                         |-------------|       |
                |                / Linear Incrementing Tap  |  Cross Fade |       |
                |                  via Sawtooth Wavetable   |  Windowing  |------- |
                |                                           |  Functions  |
                |                                           |-------------|
                |                                                +  ^
                |                          ^                        |
                |                         /                         |
                |               _________/____                      |
                |              |              |                     |
                |              |     Z^(-D)   |                     |
                |------------->|              |-----------|>------- |
                               |     d2(n)    |           1/2
                               |              |
                               |______________|
                                   /
                                  /
                                 / Linear Incrementing Tap
                                   via Sawtooth Wavetable


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

/* ADSP-21060 System Register bit definitions */
#include       "def21065l.h"
#include       "new65Ldefs.h"

.GLOBAL        pitch_shifter;
.GLOBAL        Init_Pitch_Buffers;
.GLOBAL        Timer0_Initialization;
.GLOBAL        wavetable_gen;
.EXTERN        Left_Channel;
.EXTERN        Right_Channel;

/* pitch shift filter coefficients */
#define a0            0x40000000            /* One Half, or 0.5 in 1.31 format */
#define a1            0x40000000            /* One Half, or 0.5 in 1.31 format */
#define a2            0x40000000            /* One Half, or 0.5 in 1.31 format */
/* delay buffer and wavetable definitions */
/* increasing depth size D increases pitch variations */
#define D             2000           /* Depth, or TD = D/fs = 1500/44100 = 35 msec */
#define D2            D/2
#define WaveSize      4000           /* triangular wavetable size*/
#define WinSize       4000           /* window function for cross-fading delay-lines */

/* pitch shift control parameters */
#define c             2              /* signal frequency fc1 = c1 * freqtimer = 4 Hz */
#define modulation_rate 50000        /* # of DSP cycles between Timer Expire ISR update */
#define pitch_depth   1              /* Enter # from 1 to 10.  DO NOT ENTER '0'!, to keep #'s
                                        between +/- 0.5 in fractional format */
                                     /* The lower the number, the greater the pitch bend */

/* playing with the pitch depth, modulation rate, and pitch-depth size affects
   the intensity of the pitch shifted sound,  the pitch-depth setting is strongest
   at lower numbers 1 to 3, it is less intense for 4 to 7.*/

{sawtooth chorus - modrate=50000, pitchdepth=1, c=1,  D=1000)
{detune effect  - modrate=50000, pitchdepth=1, c=2,  D=1000}
```

```
.segment /dm   delaylin;

.var         w1[D + 1];              /* delay-line buffer, max delay = D */
.var         w2[D + 1];              /* delay-line buffer, max delay = D */

.endseg;

.segment /dm    dmpitch;

.var        saw_value1;              /* wavetable update via timer for delay calculation */
.var        saw_value2;
.var        volume_dline1;
.var        volume_dline2;
.var        window_fnc[WinSize] = "triang_window.dat";  /* load window function for crossfade control */
.var        sawtooth_wav[WaveSize] = "sawtooth.dat";        /* load one period of the wavetable */
                                            /* min frequency f1 = fs/Ds = 8/4 = 2 Hz */
.endseg;


/* -------------PROGRAM MEMORY CODE---------------------------------*/
.segment /pm pm_code;

Init_Pitch_Buffers:
       B2 = w1;  L2 = @w1;                  /* delay-line 1 buffer pointer and length */
       m2 = 1;

       LCNTR = L2;                          /* clear delay line buffer to zero */
        DO clrDline1 UNTIL LCE;
clrDline1:         dm(i2, m2) = 0;

       B3 = w2;  L3 = @w2;                  /* delay-line 2 buffer pointer and length */
       m3 = 1;

       LCNTR = L3;                          /* clear delay line buffer to zero */
        DO clrDline2 UNTIL LCE;
clrDline2:         dm(i3, m3) = 0;

       B6 = sawtooth_wav;                   /* pointer 1 for sawtooth signal generator */
       L6 = @sawtooth_wav;                  /* get size from sawtooth number table lookup */
       B7 = sawtooth_wav;                   /* pointer 2 for sawtooth signal generator */
       I7 = sawtooth_wav + 2000;            /* start in middle of table */
       L7 = @sawtooth_wav;                  /* get size from sawtooth number table lookup */

       B4 = window_fnc;                     /* pointer for crossfade window for delay-line 1 */
       L4 = @window_fnc;                    /* get length of window buffer */
       B5 = window_fnc;                     /* pointer for crossfade window for delay-line 2 */
       I5 = window_fnc + 2000;                      /* start in middle of table */
       L5 = @window_fnc;                    /* get length of window buffer */

       RTS;

/*-----------------------------------------------------------------*/

/* Set up timer for the Chorus Effects wavetable generator */

Timer0_Initialization:
       bit clr mode2 TIMEN0;                /* timer off initially */
       bit set mode2 PWMOUT0 | PERIOD_CNT0 | INT_HI0;      /* latch timer0 to high priority timer int */

       r0 = modulation_rate;
       DM(TPERIOD0) = r0;
        DM(TCOUNT0) = r0;                   /* assuming 16.7 nSec cycle @ 60 MIPs */
       r0 = 10;
       DM(TPWIDTH0) =r0;

       bit set imask TMZHI;                 /* timer high priority  */
       bit set mode2 TIMEN0;                /* timer on */

       rts;


/* --------------------------------------------------------------------------------------------- */
/*                                                                                               */
/*                   Wavetable Generator used for Pitch Shift Delay Line Modulation              */
/*                                                                                          */
/* ---------------------------------------------------------------------------------------  */
/* High Priority Timer Interrupt Service Routine for Delay Line Modulation of Chorus Buffers */
/* This routine is a wavetable generator, where r3 = D2 * sin(2*pi*fc*t)   */
/* and it modulates the chorus delay line around rotating tap center */

wavetable_gen:
       bit set mode1 SRRFL;
       nop;                               /* 1 cycle latency writing to Mode1 register */

       /* c = desired wavetable increment (DAG modifier), where frequency f = c x fs / D */

sawtooth1:
       r1 = D;                            /* Total Delay Time */
       r2 = dm(i6, c);                    /* get next value in wavetable */
       r2 = ashift r2 by -pitch_depth;    /* divide by at least 2 to keep 1.31 #s between 0.5/-0.5 */
       r3= r1 * r2 (SSFR);                /* multiply Delay by a fractional value from 0 to 0.5 */
       dm(saw_value1) = r3;               /* store to memory for chorus routine */

sawtooth2:
       r1 = D;                            /* Total Delay Time */
       r2 = dm(i7, c);                    /* get next value in wavetable */
       r2 = ashift r2 by -pitch_depth;    /* divide by at least 2 to keep 1.31 #s between 0.5/-0.5 */
       r3 = r1 * r2 (SSFR);               /* multiply Delay by a fractional value from 0 to 0.5 */
       dm(saw_value2) = r3;               /* store to memory for pitch shift routine */

/* determine cross-fade gain control values for both delay-line buffers used in pitch shift routine */
/* scaling factor will be 0x00000000 whenever the center tap crosses input of the delay line buffer */

triangl_window_value1:
       r1 = dm(i4, c);                            /* corresponds with sawtooth 1 */
       dm(volume_dline1) = r1;

triangl_window_value2:
       r1 = dm(i5, c);                            /* corresponds with sawtooth 2 */
       dm(volume_dline2) = r1;

       bit clr mode1 SRRFL;
       rti;

/* ---------------------------------------------------------------------------------------- */
/*                                                                                          */
/*                   Digital Pitch Shifter Routine - process right channel only             */
/*                                                                                          */
/* ---------------------------------------------------------------------------------------- */

pitch_shifter:
       r15 = DM(Right_Channel);           /* get x-input, right channel */

       r3 = dm(saw_value1);               /* calculate time-varing delay for 2nd voice */
       r1 = D2;                           /* center tap for delay line 1*/
       r4 = r1 + r3;                      /*   r4 = d(n) = D/2 + D * random(fc*t)  */

       r5 = dm(saw_value2);               /* calculate time-varing delay for 3nd voice */
       r2 = D2;                           /* center tap for delay line 2 */
       r6 = r2 + r5;                      /*   r6 = d(n) = D/2 + D * random(fc*t)  */

       r8 = a0;                           /* input gain */
       mrf = r8 * r15 (SSF);              /*   mrf = a0 * x-input  */

       m2 = r4;                           /* tap outputs of circular delay line */
       modify(i2, m2);                    /* go to delayed sample */
       r4 = -r4;                          /* negate to get back to where we were */
       m2 = r4;                           /* used to post modify back to current sample */
       r9 = dm(i2, m2);                   /* get time-varying delayed sample 1 */
       r11 = dm(volume_dline1);           /* get scaling factor */
       r9 = r9 * r11 (SSF);               /* multiply by delay line output */

       r8 = a1;                           /* delay-line 1 gain */
       mrf = mrf + r8 * r9 (SSF);         /*   mrf = a0 * x + a1 * s1  */

       m3 = r6;                           /* tap outputs of circular delay line */
       modify(i3, m3);                    /* go to delayed sample */
       r6 = -r6;                          /* negate to get back to where we were */
       m3 = r6;                           /* used to post modify back to current sample */
```

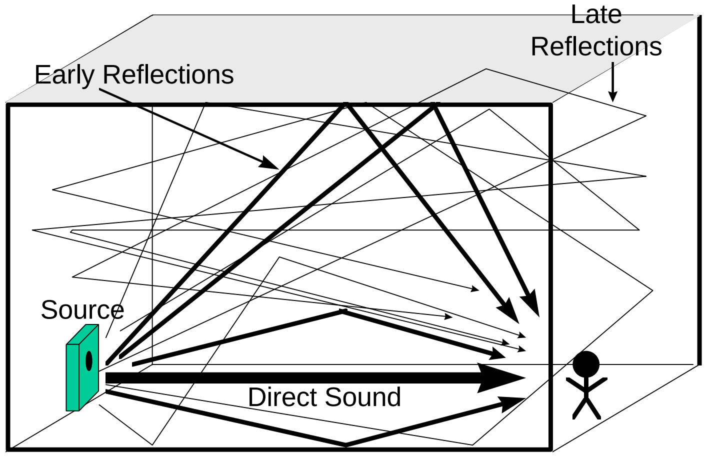
```
        r10 = dm(i3, m3);                    /* get time-varying delayed sample 2 */
        r11 = dm(volume_dline2);
        r10 = r10 * r11 (SSF);

        r9 = a2;                             /* delay-line 2 gain */
        mrf = mrf + r9 * r10 (SSFR);         /*   mrf = a0 * x + a1 * s1 + a2 + s2  */
        mrf = SAT mrf;                       /* saturate result if necessary */
        r10 = mr1f;                          /* pitch shifted result in r10 */


        /* put input sample from r15 into tap-0 of delay lines */
        dm(i2, 0) = r15;
         dm(i3, 0) = r15;

        /* backshift pointers & update circular delay-line buffers */
        modify(i2, -1);
        modify(i3, -1);

        /* write pitch shifted output sample to left/right output channels */
        DM(Left_Channel)=r10;
        DM(Right_Channel)=r10;

        rts;

.endseg;
```

3.3.3 Digital Reverberation Algorithms for Simulation of Large Acoustic Spaces

Reverberation is another time-based effect. More complex processing than echoing, chorusing or flanging, reverberation is often mistaken with delay or echo effects. Most multi-effects processing units provide a variation of both effects.

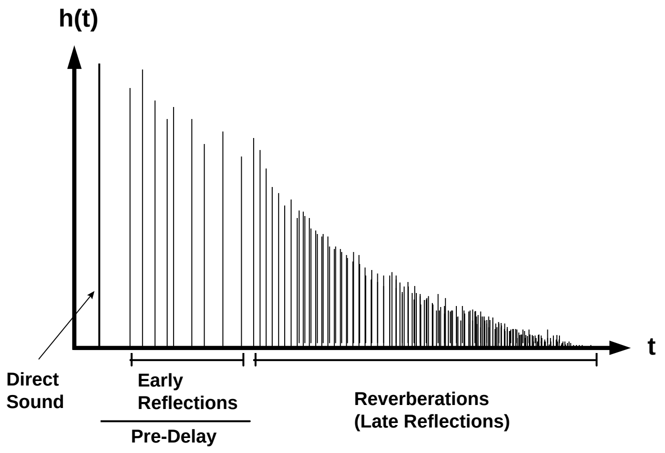
The first simulation reverb units in the 60's and 70's consisted of using a mechanical spring or plate attached to a transducer and passing the electrical signal through. Another transducer at the other end converted the mechanical reflections back to the output transducer. However, this did not produce realistic reverberation. M.A Schroeder and James A. Moorer developed algorithms for producing realistic reverb using a DSP.

Figure 78.
Reverberation of Large Acoustic Spaces



The reverb effect simulates the effect of sound reflections in a large concert hall or room (Figure 78). Instead of a few discrete repetitions of a sound like a multi-tap delay effect, the reverb effect implements many delayed repetitions so close together in time that the ear cannot distinguish the differences between the delays. The repetitions are blended together to sound continuous. The sound source goes out in every direction from the source, bounces off the walls and ceilings and returns from many angles with different delays. Reverberation is almost always present in indoor environments, and the reflections are greater for hard surfaces. As Figure 12 below shows, Reverberated Sound is classified as three components: Direct Sound, Early reflections and the Closely Blended Echos (Reverberations) [11,12,14].

Figure 79.
Impulse Response For Large Auditorium Reverberations



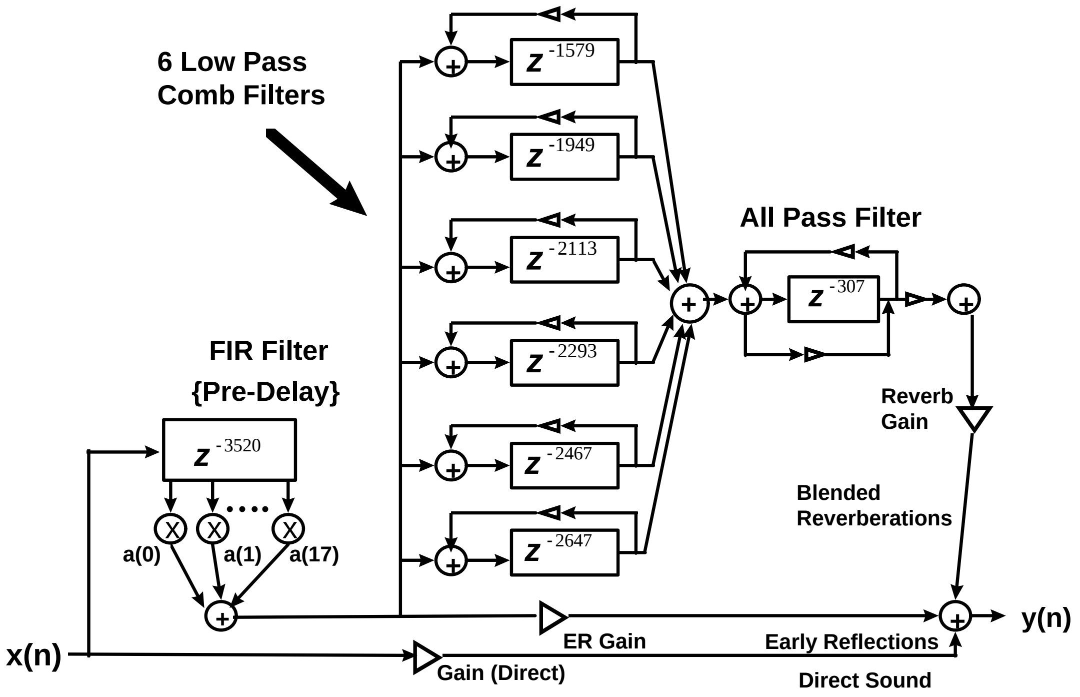
- **Direct Sound** - directly reaches the listener from the sound source.
- **Early reflections -** early echos which arrive within 10 ms to 100 ms by the early reflections of surfaces after the direct sound.
- **Closely Blended Echos** - is produced after 100 ms early reflections.

Figure 79 shows an impulse response of a large acoustic space, such as an auditorium or gymnasium. In a typical large auditorium, the first distinct delay responses that the user will hear are termed 'early reflections'. These early reflections are a few relatively close echos that actually occur in as reverberation in large spaces. The early reflections are the result of the first bounce back of the source by surfaces that are nearby. Next come echos which follow one another at such small intervals that the later reflections are no longer distinguishable to the human ear. A Digital Reverb typically will process the input through multiple delayed filters and add the result together with early reflection computations. Various parameters to consider in the algorithm would be the decay time (time it takes for reverb to decay) , presense (dry signal output vs. reverberations), and tone control (bass or treble) of the output reverberations.

M.A. Schroeder suggested 2 ways for producing a more realistic sounding reverb. The first approach was to implement 5 allpass filters cascaded together. The second way was to use 4 comb filters in parallel, summing their outputs, then passing the result through 2 allpass filters in cascade.

James A. Moorer expanded on Schroeder's research. One drawback to the Schroeder Reverb is that the high frequencies tend to reverberate longer than the lower frequencies. Moorer proposed using a *low pass comb filter* for each reverb stage to enlarge the density of the response. He demonstrated a technique involving 6 parallel comb filters with a low pass output, summing their outputs and then sending the summed result to an allpass filter before producing the final result. Moorer also recommended including the simulation of the early reflections common in concert halls using a tapped-delay FIR filter structure, along with the reverb filters for a more realistic response. Some initial delays can be added to the input signal by using an FIR filter ranging from 0 to 80 milliseconds. Moorer chose appropriate filter coefficients to produce 19 early reflections. Moorer's reverberator produced a more realistic reverb sound than Schroeder's, but still produces a rough sound for impulse signals such as drums.

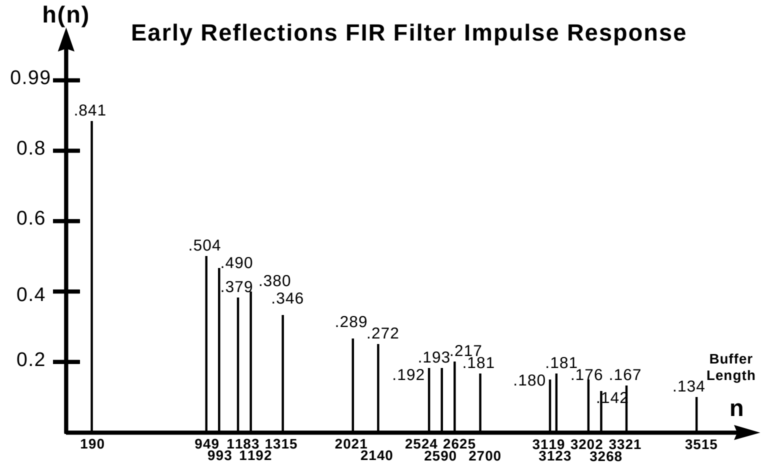
Figure 80.
James A. Moorer's Digital Reverberation Structure



The figure above shows the structure for J.A Moorer’s Digital Reverb [11] algorithm for a large auditorium response assuming a 44.1 kHz sampling rate. Moorer demonstrated a technique involving 6 parallel comb filters with a low pass output, summing their outputs and then sending the summed result to an all-pass filter before producing the final result. For realistic sounding reverberation, the DSP requires the use of large delay lines for both the comb filter and early reflection buffers. Each comb filter incorporates a different length delay-line. The reverberation delay time depends on the length the delay-line buffer sizes and the sampling rate. Fine tuning of input and feedback for each comb filter gain and delay-line values vary the reverberation effect to provide a different room size characteristic. Since all are programmable, the decay response can be modified on the fly to change the amount of the reverb effect.

Figure 82.

Early Reflections FIR Filter Impulse Response



Example Reverb Specifications for a Large Auditorium response at 44.1 kHz Sampling Rate

Delay Line	*Buffer Length*	*Time Delay*
Comb 1	*1759*	*40 ms*
Comb 2	*1949*	*44 ms*
Comb 3	*2113*	*48 ms*
Comb 4	*2293*	*52 ms*
Comb 5	*2467*	*56 ms*
Comb 6	*2647*	*60 ms*
Early Reflections	*3520*	*80 ms*
All-Pass Filter	*307*	*7 ms*

Table 6.

Early Reflection Delay Tap Lengths	Early Reflection Tap Gain Parameters Fractional 1.15 Representation
-190	0.841 = 0x6BA6
-759	0.504 = 0x4083
-44	0.490 = 0x3ED9
-190	0.379 = 0x3083
-9	0.380 = 0x30A4
-123	0.346 = 0x2C4A
-706	0.289 = 0x24FE
-119	0.272 = 0x22D1
-384	0.192 = 0x1893
-66	0.193 = 0x18B4
-35	0.217 = 0x1BC7
-75	0.181 = 0x172B
-419	0.180 = 0X170A
-4	0.181 = 0x172B
-79	0.176 = 0x1687
-66	0.142 = 0x122D
-53	0.167 = 0x1560
-194	0.134 = 0x1127

Reverb Building Blocks Low Pass Comb Filter and All Pass Filter Structures

For realistic sounding reverberation, the DSP requires the use of large delay lines for both the comb filter and early reflection buffers. The comb filter is used to increase echo density and give the impression of a sizable acoustic space. Each comb filter incorporates a different length delay line. Each delay line can be tuned to a different value to provide a different room size characteristic. Fine tuning of input and feedback gains for each comb filter gain and comb filter delay-line sizes will vary the reverberation response. Since these parameters are programmable, the decay response can be modified on the fly to change the amount of the reverb effect for simulation of a large hall or small room. The total reverberation delay time depends on the size the comb filter/early reflections buffers and the sampling rate.

Low pass filtering in each comb filter stage reduces the metallic sound and shortens the reverberation time of the high frequencies, just as a real auditorium response does. The allpass filter is used along with the comb filters to add some color to the 'colorless/flat' sound by varying the phase, thus helping to emulate the sound characteristics of a real auditorium.

ADSP-21061 Low Pass Comb Filter Subroutine Example

```
/*     Low Pass IIR Comb Filter Structure:

        x(n) --------->O--------------------------------------------------------> y(n)
                       ^ +                                                |
                       |                                              ____|____
                       |                                             |         |
                       |                                             | z^(-D)  |
                  u(n) |                                             |_________|
                       |                                                  |
                       |                 b0                               |
                       |-----------O-----<|--------------------O----------|
                                 + |             | v0(n)       | +
                                   |             |             |
                                   |          ___|___          |
                                   |         |       |         |
                                   |         |  z^-1 |         |
                                   |         |_______|         |
                                   |             | v1(n)       |
                                   |    b1       |      a1     |
                                   |-----<|------|------|>-----|

*/

Low_Pass_Comb_Filter:
       L3 = @comb_output;
       B3 = comb_output;
       R0 = 0x50710000;                 /* gf = Gf/(1+gl) -- comb feedback gain */
       R1 = 0x26660000;                 /* gl -- low pass filter gain */

       L9 = COMB_LENGTH;
       R4 = PM(I9,0);                   /* read comb buffer -> output */

       R5 = DM(comb_lpf_state);         /* read previous low pass filter state */
       MRF = R4*R0 (SSF), DM(I3,M1) = R4;  /* save output, feedback comb output */
       MRF = MRF + R5*R1 (SSFR);        /* add low pass filter state     */
       R10 = MR1F;
       DM(comb_lpf_state) = R10;        /* replace with new filter state */

       MR1F = R5;                       /* get old low pass filter output */
       R7 = 0x46760000;                 /* gi = 1/(1 + gf + gf*gl) */
       MRF = MRF + R6*R7 (SSFR);        /* mac lpf output with input*/
       R10 = MR1F;
       PM(I9,-1) = R10;                 /* write sample to buffer */
       RTS;
```

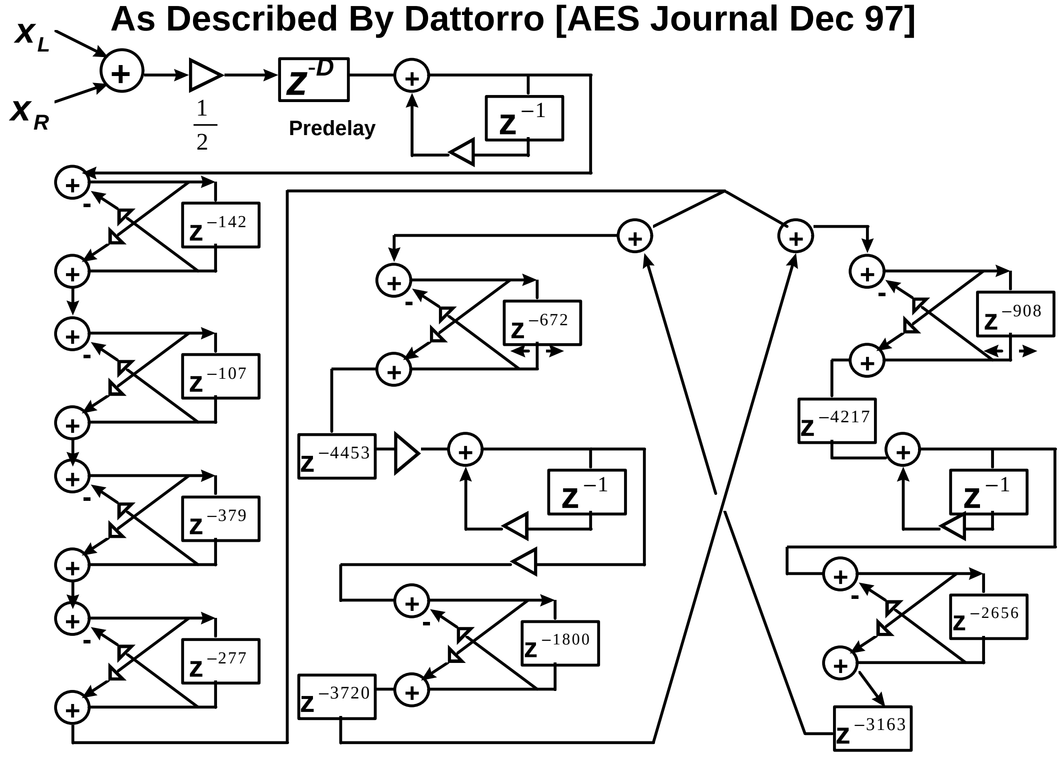
ADSP-21065L Example All-Pass Filter Implementation

```
/*     1st Order Allpass Transfer function and I/O difference equations:
                -a + z(-D)
        H(z) = -----------          y(n) = ay(n - D) - ax(n) + x(n - D)
              1 - az^(-D)



       Allpass Filter Structure:
                              a
                           |-------------|>------------|
                           |                           |
                           |      _________            |
                           |     |         |         + v
        x(n) --------->O---------| z^(-D)  |-----------O--------> y(n)
                       ^ +       |_________|       |
                       |                           |
                       |          -a               |
                       |---------<|----------------|



       Using the Allpass Filter as a reverb building block
       ---------------------------------------------------
       IIR comb filters tend to magnify input signal frequencies near comb filter peak
       frequencies.  Allpass filters can be used to prevent this 'coloration' of the input
       since it has a relatively flat magnitude response for all frequencies.
*/

all_pass_filter:
       L0 = @all_pass;
       R1 = 0x599A0000;                 /* feedback gain    */

       R10 = DM(I0,0);                  /* load output of buffer */
       MR1F = R10;
       MRF = MRF + R1*R0 (SSFR);        /* add to (feedback gain)*(input) */
       MRF = SAT MRF;
       R3 = MR1F;                       /* output of all pass in R3 */

       MR1F = R0;                       /* put input of all pass in MR1F */
       MRF = MRF - R1*R3 (SSFR);        /* input - (feedback gain)*(output) */
       MRF = SAT MRF;
       R10 = MR1F;
       DM(I0,M7) = R10;                 /* save to input of buffer */
       RTS;
```

Figure 83 is an example implementation of a Plate Reverb topology that is described by Dattorro[30], and will not be discussed in too much detail here. The reader should refer to Dattorro's description on this class of reverb algorithms, although it is not explored in much theoretical detail, as he suggests that it is best explored through tinkering with the gains delay line values and output tap points. This suggested implementation yields very high quality reverberation very efficiently. Notice that is has similar building blocks using comb filters. In addition, it uses allpass filter diffusers based on lattice structure topologies.

Figure 83.

Griesinger's Plate Class Reverberation Structure As Described By Dattorro [AES Journal Dec 97]



```
/* ***************************************************************************************************

       Digital Plate-Class Reverberation Audio Effect - Griesinger's Model
       Described by Jon Dattorro in "Effect Design Part 1: Reverberator and
       Other Filters, " Journal of the Audio Engineering Society," Vol. 45,
       No. 9, pp. 660-684, September 1997.

       Created for the 21065L EZ-LAB Evaluation Platform

       Includes on-the-fly selection of reverb comb & allpass filter gains/lengths,
       predelay gain/length presets, and left/right panning via IRQ1 and IRQ2 pushbutton control

       ** Works well for single instruments such as a guitar, keyboard, violin....

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

/* ADSP-21060 System Register bit definitions */
#include       "def21065l.h"
#include       "new65Ldefs.h"

.GLOBAL        Digital_Reverberator;
.GLOBAL        Init_Reverb_Buffers;
.GLOBAL        change_reverb_settings;
.GLOBAL        modify_reverb_mix;
.EXTERN        Left_Channel;
.EXTERN        Right_Channel;

/* Default Reverberation Parameters - Reverb DM Variables and pointers */
.segment /dm    rev_vars;

#define Fs1    29761          /* Sample Rate is 29761 Hz, default recommended by Dattorro */
                              /* reducing Fs will reduce delay line memory requirements */
                              /* Reverberation can be convincing at sample rates as low as 20-24 kHz */
#define Fs2    48000          /* Sample Rate is 48000 Hz, delay lines increased for 48 kHz */

/* reverb left channel output taps at 48 kHz fs */
#define D1_L                  429            /* = 266 @ 29761 Hz Fs */
#define D2_L                  4797           /* = 2974 @ 29761 Hz Fs*/
#define D3_L                  3085           /* = 1913 @ 29761 Hz Fs */
#define D4_L                  3219           /* = 1996 @ 29761 Hz Fs*/
#define D5_L                  3210           /* = 1990 @ 29761 Hz Fs */
#define D6_L                  302            /* = 187 @ 29761 Hz Fs */
#define D7_L                  1719           /* = 1066 @ 29761 Hz Fs */

/* reverb right channel output taps at 48 kHz fs*/
#define D1_R                  569            /* = 353 @ 29761 Hz Fs */
#define D2_R                  3627           /* = 3627 @ 29761 Hz Fs */
#define D3_R                  5850           /* = 1228 @ 29761 Hz Fs */
#define D4_R                  2673           /* = 2673 @ 29761 Hz Fs */
#define D5_R                  4311           /* = 2111 @ 29761 Hz Fs */
#define D6_R                  540            /* = 335 @ 29761 Hz Fs */
#define D7_R                  195            /* = 121 @ 29761 Hz Fs */

/* pointers for input and decay diffusers */
.VAR   all_pass_ptr1;
.VAR   all_pass_ptr2;
.VAR   all_pass_ptr3;
.VAR   all_pass_ptr4;
.VAR   all_pass_ptr5;
.VAR   all_pass_ptr6;
.VAR   all_pass_ptr7;
.VAR   all_pass_ptr8;

/* Single Length Comb Filter State Variables */
.VAR   comb1_feedback_state;
.VAR   comb2_feedback_state;
.VAR   comb3_feedback_state;

.VAR   recirculate1_feedback;
.VAR   recirculate2_feedback;
.VAR   diffusion_result;

.VAR   predelay_output;
.VAR   reverb_left;
.VAR   reverb_right;
.VAR   Lrev_output_taps[6];
.VAR   Rrev_output_taps[6];

.endseg;


/* -------------DATA MEMORY DELAY LINE & ALLPASS FILTER BUFFERS  ------------------------*/
.segment /dm   dm_revrb;

/* Allpass and Delay Line Filter Length Definitions */
#define allpass_Dline1        2168                  /* = 1344 @ 29761 Hz Fs */
#define allpass_Dline2        2930                  /* = 1816 @ 29761 Hz Fs */
#define ALLPASS1_LENGTH       229                   /* = 142 @ 29761 Hz Fs */
#define ALLPASS2_LENGTH       172                   /* = 107 @ 29761 Hz Fs */
#define ALLPASS3_LENGTH       611                   /* = 379 @ 29761 Hz Fs */
#define ALLPASS4_LENGTH       447                   /* = 277 @ 29761 Hz Fs */
#define ALLPASS5_LENGTH       allpass_Dline1/2      /* 2168/2 variable delay rotating around tap center
*/
#define ALLPASS6_LENGTH       allpass_Dline2/2      /* 2930/2 variable delay rotating around tap center
*/
#define ALLPASS7_LENGTH       2903                  /* = 1800 @ 29761 Hz Fs */
#define ALLPASS8_LENGTH       4284                  /* = 2656 @ 29761 Hz Fs */
#define D_Line1               7182                  /* = 4453 @ 29761 Hz Fs */
#define D_Line2               6000                  /* = 3720 @ 29761 Hz Fs */
#define D_Line3               6801                  /* = 4217 @ 29761 Hz Fs */
#define D_Line4               5101                  /* = 3163 @ 29761 Hz Fs */

/* Audio delay lines */
.VAR   predelay[6321];
.VAR   w1[D_Line1];
.VAR   w2[D_Line2];
.VAR   w3[D_Line3];
```

```
.VAR    w4[D_Line4];

/* input diffusers using allpass filter structures */
.VAR    diffuser_1[ALLPASS1_LENGTH];
.VAR    diffuser_2[ALLPASS2_LENGTH];
.VAR    diffuser_3[ALLPASS3_LENGTH];
.VAR    diffuser_4[ALLPASS4_LENGTH];
.VAR    decay_diffuser_A1[ALLPASS5_LENGTH];  /* allpass diffuser with variable delay */
.VAR    decay_diffuser_B1[ALLPASS6_LENGTH];  /* allpass diffuser with variable delay */
.VAR    decay_diffuser_A2[ALLPASS7_LENGTH];
.VAR    decay_diffuser_B2[ALLPASS8_LENGTH];

.endseg;


/* ----------- INTERRUPT/FLAG REVERB FX DEMO CONTROL PARAMETERS ---------*/
.segment /dm   IRQ_ctl;

/* Reverb Control Parameters, these control 'knobs' are used to change the response on-the-fly */
.VAR    decay = 0x40000000;           /* Rate of decay - 0.05 */
.VAR    bandwidth = 0x7f5c28f6;       /*  = 0.9995, High-frequency attenuation on input */
                                      /* full bandwidth = 0x9999999 */
.VAR    damping = 0x0010624d;         /*  = 0.0005, High-frequency damping; no damping = 0.0 */
.VAR    predelay_time = 200;          /* controls length (L6 register)of predelay buffer */
                                      /* length value always << max buffer length! */

.VAR    decay_diffusion_1 = 0x5999999a;       /*  = 0.70, Controls density of tail */
.VAR    decay_diffusion_2 = 0x40000000;       /*  = 0.50, Decorrelates tank signals */
                                      /* decay diffusion 2 = decay +0.15, floor = 0.25, ceiling - 0.50 */
.VAR    input_diffusion_1 = 0x60000000;       /*  = 0.75, Decorrelates incoming signal */
.VAR    input_diffusion_2 = 0x50000000;       /*  = 0.625, Decorrelates incoming signal */

.VAR    DRY_GAIN_LEFT = 0x7FFFFFFF;   /* Gain Control for left channel */
                                      /* scale between 0x00000000 and 0x7FFFFFFF */
.VAR    DRY_GAIN_RIGHT = 0x7FFFFFFF;  /* Gain Control for right channel */
                                      /* scale between 0x00000000 and 0x7FFFFFFF */
.VAR    PREDEL_GAIN_LEFT = 0x00000000;/* Gain Control for predelay output */
                                      /* scale between 0x00000000 and 0x7FFFFFFF */
.VAR    PREDEL_GAIN_RIGHT = 0x00000000; /* Gain Control for predelay output */
                                      /* scale between 0x00000000 and 0x7FFFFFFF */
.VAR    WET_GAIN_LEFT = 0x7FFFFFFF;   /* Gain for reverb result */
                                      /* scale between 0x00000000 and 0x7FFFFFFF */
.VAR    WET_GAIN_RIGHT = 0x7FFFFFFF;  /* Gain for reverb result */
                                      /* scale between 0x00000000 and 0x7FFFFFFF */

.VAR    IRQ1_counter = 0x00000004;    /* selects preset 1 on first IRQ1 assertion */
.VAR    IRQ2_counter = 0x00000004;    /* selects preset 1 on first IRQ2 assertion */

.endseg;

/* -------------PROGRAM MEMORY CODE---------------------------------*/
.segment /pm pm_code;

Init_Reverb_Buffers:
        /* initialize all-pass filter pointers to top of respective buffers */
        B7 = diffuser_1;
        DM(all_pass_ptr1) = B7;
        B7 = diffuser_2;
        DM(all_pass_ptr2) = B7;
        B7 = diffuser_3;
        DM(all_pass_ptr3) = B7;
        B7 = diffuser_4;
        DM(all_pass_ptr4) = B7;
        B7 = decay_diffuser_A1;
        DM(all_pass_ptr5) = B7;
        B7 = decay_diffuser_B1;
        DM(all_pass_ptr6) = B7;
        B7 = decay_diffuser_A2;
        DM(all_pass_ptr7) = B7;
        B7 = decay_diffuser_B2;
        DM(all_pass_ptr8) = B7;

        /* Initialize Audio Delay Lines */
        B2 = w1;  L2 = @w1;           /* delay-line buffer 1 pointer and length */
        B3 = w2;  L3 = @w2;           /* delay-line buffer 1 pointer and length */
        B4 = w3;  L4 = @w3;           /* delay-line buffer 1 pointer and length */
        B5 = w4;  L5 = @w4;           /* delay-line buffer 1 pointer and length */
        B6 = predelay; L6 = @predelay;

        /* clear all audio delay line buffers to zero */
        m2 = 1; m3 = 1; m4 = 1; m5 = 1; m6 = 1;

        LCNTR = L2;
         DO clrDline_1 UNTIL LCE;
clrDline_1:    dm(i2, m2) = 0;

        LCNTR = L3;
         DO clrDline_2 UNTIL LCE;
clrDline_2:    dm(i3, m3) = 0;

        LCNTR = L4;
         DO clrDline_3 UNTIL LCE;
clrDline_3:    dm(i4, m4) = 0;

        LCNTR = L5;
         DO clrDline_4 UNTIL LCE;
clrDline_4:    dm(i5, m5) = 0;

        LCNTR = L6;
         DO clrDline_5 UNTIL LCE;
clrDline_5:    dm(i6, m6) = 0;

        RTS;

/* ----------------------------------------------------------------------------------------------  */
/*                                                                                                 */
/*                           Digital Reverb Filter Routines                                        */
/*                                                                                                 */
/* ------------------------------------------------------------------------------------------------*/

Digital_Reverberator:
        /* combine both left and right input samples together into 1 signal */
        r0 = dm(Left_Channel);                /* left input sample */
        r1 = dm(Right_Channel);               /* right input sample */
        r0 = ashift r0 by -1;                 /* scale signal by 1/2 for equal mix */
        r1 = ashift r1 by -1;                 /* scale signal by 1/2 for equal mix */
        r2 = r0 + r1;                         /* 1/2xLeft(n) + 1/2 xRight(n) = sum of input samples */

compute_predelay:
        L6 = dm(predelay_time);               /* get predelay time setting, default is L6=@predelay */
        r3 = dm(i6, 0);                       /* get oldest sample, time delay = D_Line1 x fs*/
        dm(i6, -1) = r2;                      /* write input sample to buffer */
        dm(predelay_output) = r3;             /* used for final mix */
        r2 = r3;                              /* r2 is input to reverb routines */

        call (PC, hi_freq_input_atten);        /* attenuate high-freqencies on input */

        r0 = r2;
        call (PC, input_diffusers);           /* call all-pass filters */

        call (PC, reverberation_tank);        /* controls the rate of reverberation decay */

        call reverb_mixer;

/*
bypass_mixer:
        r0 = dm(reverb_left);
        dm(Left_Channel) = r0;
        r0 = dm(reverb_right);
        dm(Right_Channel) = r0;
*/
        rts;

/* ----------------------------------------------------------------------------------------------  */
/*                                                                                                 */
/*                           High Frequency Input Attenuator                                       */
/*                                                                                                 */
/*      This routine is a simple comb filter, used to attenuate high frequencies on the input.     */
/*      Higher frequencies tend to diminish faster than lower frequecies in reverberant            */
/*      acoustic spaces                                                                            */
/*      DM(bandwidth) = full bandwidth gain when set to 0.9999999 (0x7FFFFFFF in 1.31 format)     */
/* -------------------------------------------------------------------------------------------  */

hi_freq_input_atten:
        /* comb filter 1 - attenuate high frequencies on the input */
        /* full bandwidth gain is 0.9999999 */
        r4 = r2;                             /* r4 = comb filter input */
        r6 = dm(bandwidth);                  /* high frequency bandwidth gain */
        r7 = 0x7FFFFFFF;                     /* 0.99999 or approximately = 1 */
        r8 = r7 - r6;                        /* r8 = 1 - bandwidth */
        r5 = DM(comb1_feedback_state);       /* read previous low pass filter output state */
        mrf = r4*r6 (SSF);                   /* scale comb filter input */
        mrf = mrf + r5*r8 (SSFR);            /* add previous filter state */
        r2 = mr1f;                           /* r10 = comb filter output */
        dm(comb1_feedback_state) = r10;      /* replace with new filter state */
        rts;                                 /* return from subroutine */

/* ------------------------------------------------------------------------------------------   */

/* --------------- All Pass Filter Routines ------------------------------ */
/*                                                                         */
/*      Each all-pass filter diffusers are implemented in the topology     */
/*      of a two-multiplier lattice structure.                             */
/*                                                                         */
/*              input   ->      R0                                         */
/*              output  ->     R3                                          */
/*                                                  `                      */
/*      Also, it is not necessary to save and restore all comb filter      */
/*      FIR or all-pass filter index and length registers, if only         */
/*      doing this reverb demo.  These extra instructions are included     */
/*      so that this example can easily be combined with other audio       */
/*      effects that require the use of multiple buffers                   */
/*                                                                         */
/*      1 index register I7 is used for all 4 allpass filters              */
/* ----------------------------------------------------------------------- */

input_diffusers:

        B7 = diffuser_1;                     /* set base address to buffer */
        I7 = DM(all_pass_ptr1);              /* get previous allpass 1 pointer address */
        L7 = @diffuser_1;                    /* set length of circular buffer */
        R1 = DM(input_diffusion_1);          /* feedback gain for allpass */

allpass_filt1:
        R10 = DM(I7,0);                      /* load output of buffer */
        MR1F = R10;                          /* put in forground MAC register */
        MRF = MRF + R1*R0 (SSFR);            /* add to (feedback gain)*(input) */
        MRF = SAT MRF;                       /* saturate if necessary */
        R3 = MR1F;                           /* output of all pass in R3 */

        MR1F = R0;                           /* put input of all pass in MR1F */
        MRF = MRF - R1*R3 (SSFR);            /* input - (feedback gain)*(output) */
        MRF = SAT MRF;                       /* saturate if necessary */
        R10 = MR1F;                          /* put MAC result in register file */
        DM(I7,1) = R10;                      /* save to input of buffer, update pointer */
        DM(all_pass_ptr1) = I7;              /* save current allpass 1 pointer address for next time */


allpass_filt2:
        B7 = diffuser_2;                     /* set base address to buffer */
        I7 = DM(all_pass_ptr2);                      /* get previous allpass 2 pointer address */
        L7 = @diffuser_2;                    /* set length of circular buffer */
        R1 = DM(input_diffusion_1);          /* feedback gain for allpass 1*/
        R0 = R3;                             /* output of allpass 1 = input of allpass 2 */

        R10 = DM(I7,0);                      /* load output of buffer */
        MR1F = R10;                          /* put in forground MAC register */
        MRF = MRF + R1*R0 (SSFR);            /* add to (feedback gain)*(input) */
        MRF = SAT MRF;                       /* saturate if necessary */
        R3 = MR1F;                           /* output of all pass in R3 */

        MR1F = R0;                           /* put input of all pass in MR1F */
        MRF = MRF - R1*R3 (SSFR);            /* input - (feedback gain)*(output) */
        MRF = SAT MRF;                       /* saturate if necessary */
        R10 = MR1F;                          /* put MAC result in register file */
```

```
       DM(I7,1) = R10;                    /* save to input of buffer, update pointer */
       DM(all_pass_ptr2) = I7;            /* save current allpass 2 pointer address for next time */

allpass_filt3:
       B7 = diffuser_3;                   /* set base address to buffer */
       I7 = DM(all_pass_ptr3);            /* get previous allpass 3 pointer address */
       L7 = @diffuser_3;                  /* set length of circular buffer */
       R1 = DM(input_diffusion_2);        /* feedback gain for allpass 2*/
       R0 = R3;                           /* output of allpass 2 = input of allpass 3 */

       R10 = DM(I7,0);                    /* load output of buffer */
       MR1F = R10;                        /* put in forground MAC register */
       MRF = MRF + R1*R0 (SSFR);          /* add to (feedback gain)*(input) */
       MRF = SAT MRF;                     /* saturate if necessary */
       R3 = MR1F;                         /* output of all pass in R3 */

       MR1F = R0;                         /* put input of all pass in MR1F */
       MRF = MRF - R1*R3 (SSFR);          /* input - (feedback gain)*(output) */
       MRF = SAT MRF;                     /* saturate if necessary */
       R10 = MR1F;                        /* put MAC result in register file */
       DM(I7,1) = R10;                    /* save to input of buffer, update pointer */
       DM(all_pass_ptr3) = I7;            /* save current allpass 3 pointer address for next time */

allpass_filt4:
       B7 = diffuser_4;                   /* set base address to buffer */
       I7 = DM(all_pass_ptr4);            /* get previous allpass 4 pointer address */
       L7 = @diffuser_4;                  /* set length of circular buffer */
       R1 = DM(input_diffusion_2);        /* feedback gain for allpass 3 */
       R0 = R3;                           /* output of allpass 3 = input of allpass 4 */

       R10 = DM(I7,0);                    /* load output of buffer */
       MR1F = R10;                        /* put in forground MAC register */
       MRF = MRF + R1*R0 (SSFR);          /* add to (feedback gain)*(input) */
       MRF = SAT MRF;                     /* saturate if necessary */
       R3 = MR1F;                         /* output of all pass in R3 */
       DM(diffusion_result) = R3;         /* save for holding tank routines */

       MR1F = R0;                         /* put input of all pass in MR1F */
       MRF = MRF - R1*R3 (SSFR);          /* input - (feedback gain)*(output) */
       MRF = SAT MRF;                     /* saturate if necessary */
       R10 = MR1F;                        /* put MAC result in register file */
       DM(I7,1) = R10;                    /* save to input of buffer, update pointer */
       DM(all_pass_ptr4) = I7;            /* save current allpass 4 pointer address for next time */

       RTS;                               /* return from subroutine */


/* ----------------- Reverb Holding Tank Routines -------------------------------  */
/*                                                                                 */
/*                    R3      <-      input                                        */
/*                                                                                 */
/*                    output  ->      DM(reverb_left)                              */
/*                                    DM(reverb_right)                             */
/*                                                                                 */
/*     The reverberation tank recirculates 4 diffusers.  It's purpose              */
/*     is to 'trap' the input and make it recirculate in a 'figure 8'              */
/*     structure, thus altering the tail of the decaying reverb response.          */
/*     The decay coefficients determine the rate of decay.                         */
/*     It is recommended to set the coefficents by ear.                            */
/*                                                                                 */
/*     Also note:  Diffuser Lattices A1 and B1 have negative coefficients.         */
/*     The allpass structures have the MAC adds and subtract instructions          */
/*     reversed.  The impulse response changes, but it is still an allpass        */
/*     filter. This is recommended by Dattorro to further enhance the delay        */
/*     diffusion between both sides of the holding tank                            */
/*                                                                                 */
/* ------------------------------------------------------------------------------  */

reverberation_tank:
       /* initialize left & right output tap buffer pointers */
       B0 = Lrev_output_taps;
       L0 = @Lrev_output_taps;
       B1 = Rrev_output_taps;
       L1 = @Rrev_output_taps;
```

```
        r1 = DM(recirculate1_feedback);       /* get previously trapped incoming audio signal */
        r2 = DM(decay);                       /* Rate of decay on previous trapped signal */
        r4 = r1*r2 (SSF);                     /* scale prior signal state */
        r3 = DM(diffusion_result);            /* get output from input diffusion section */
        r14 = r3 + r4;                        /* add to current input diffuser result */

        r1 = DM(recirculate2_feedback);       /* get previously trapped incoming audio signal */
        r2 = DM(decay);                       /* Rate of decay on previous trapped signal */
        r4 = r1*r2 (SSF);                     /* scale prior signal state */
        r3 = DM(diffusion_result);            /* get output from input diffusion section */
        r15 = r3 + r4;                        /* add to current input diffuser result */
        /* r14 and r15 are inputs to holding tank */

diffusor_A1:                                  /* allpass filter with variable delay */
        B7 = decay_diffuser_A1;               /* set base address to buffer */
        I7 = DM(all_pass_ptr5);               /* get previous allpass 2 pointer address */
        L7 = @decay_diffuser_A1;              /* set length of circular buffer */
        r1 = DM(decay_diffusion_1);           /* feedback gain for allpass 1*/
        r0 = r15;                             /* tank input 1 */

        r10 = DM(I7,0);                       /* load output of buffer */
        mr1f = r10;                           /* put in forground MAC register */
        mrf = mrf - r1*r0 (SSFR);             /* add to -(feedback gain)*(input) */
        mrf = SAT mrf;                        /* saturate if necessary */
        r3 = mr1f;                            /* output of all pass in R3 */

        mr1f = r0;                            /* put input of all pass in MR1F */
        mrf = mrf + r1*r3 (SSFR);             /* input + (feedback gain)*(output) */
        mrf = SAT mrf;                        /* saturate if necessary */
        r10 = mr1f;                           /* put MAC result in register file */
        DM(I7,-1) = r10;                      /* save to input of buffer, update pointer */
        DM(all_pass_ptr5) = I7;               /* save current allpass 2 pointer address for next time */

audio_delay1:
        r5 = dm(i2, 0);                       /* get oldest sample, time delay = D_Line1 x fs*/

        /* tap inside of circular delay line 1, r0 = sampleD5_L = D5_L-th tap */
        m4 =  D5_L;  modify(i2, m2);          /* point to d-th tap */
        m4 = -D5_L;  r0 = dm(i2, m2);         /* put d-th tap in data register */
        DM(Lrev_output_taps + 4)= r0;         /* write to 4th location in left reverb output buffer */

        /* tap inside of circular delay line 1, r0 = sampleD1_R = D1_R-th tap */
        m4 =  D1_R;  modify(i2, m2);          /* point to d-th tap */
        m4 = -D1_R;  r0 = dm(i2, m2);         /* put d-th tap in data register */
        DM(Rrev_output_taps + 0)= r0;         /* write to 0'th location in left reverb output buffer */

        /* tap inside of circular delay line 1, r0 = sampleD2_R = D2_R-th tap */
        m4 =  D2_R;  modify(i2, m2);          /* point to d-th tap */
        m4 = -D2_R;  r0 = dm(i2, m2);         /* put d-th tap in data register */
        DM(Rrev_output_taps + 1)= r0;         /* write to 1st location in left reverb output buffer */

        dm(i2, -1) = r3;                      /* write input sample to buffer */


comb_filter_2:
        /* comb filter 2 */
        r4 = r5;                              /* r4 = comb filter input */
        r6 = dm(damping);                     /* high frequency dampling gain */
        r7 = 0x7FFFFFFF;                      /* 0.99999 or approximately = 1 */
        r8 = r7 - r6;                         /* r8 = 1 - damping */
        r5 = DM(comb2_feedback_state);        /* read previous low pass filter output state */
        mrf = r4*r8 (SSF);                    /* scale comb filter input */
        mrf = mrf + r5*r6 (SSFR);             /* add previous filter state  */
        r10 = mr1f;                           /* r10 = comb filter output */
        dm(comb2_feedback_state) = r10;       /* replace with new filter state */

        r11 = DM(decay);
        r0 = r10*r11 (SSFR);

diffusor_A2:                                  /* allpass filter with variable delay */
        B7 = decay_diffuser_A2;               /* set base address to buffer */
        I7 = DM(all_pass_ptr6);               /* get previous allpass 2 pointer address */
        L7 = @decay_diffuser_A2;              /* set length of circular buffer */
        R1 = DM(decay_diffusion_2);           /* feedback gain for allpass 1*/
        R0 = R3;                              /* output of allpass 1 = input of allpass 2 */
```

```
       r10 = DM(I7,0);                      /* load output of buffer */
       mr1f = r10;                          /* put in forground MAC register */
       mrf = mrf + r1*r0 (SSFR);            /* add to (feedback gain)*(input) */
       mrf = SAT mrf;                       /* saturate if necessary */
       r3 = mr1f;                           /* output of all pass in R3 */

       mr1f = r0;                           /* put input of all pass in MR1F */
       mrf = mrf - r1*r3 (SSFR);            /* input - (feedback gain)*(output) */
       mrf = SAT mrf;                       /* saturate if necessary */
       r10 = mr1f;                          /* put MAC result in register file */

       /* tap inside of decay diffuser A2 filter delay line, r0 = sampleD6_L = D6_L-th tap */
       m7 =  D6_L;  modify(i7, m7);         /* point to d-th tap */
       m7 = -D6_L;  r0 = dm(i7, m7);        /* put d-th tap in data register */
       DM(Lrev_output_taps + 5)= r0;        /* write to 5th location in left reverb output buffer */

       /* tap inside of decay diffuser A2 filter delay line, r0 = sampleD3_R = D3_R-th tap */
       m7 =  D3_R;  modify(i7, m7);         /* point to d-th tap */
       m7 = -D3_R;  r0 = dm(i7, m7);        /* put d-th tap in data register */
       DM(Rrev_output_taps + 2)= r0;        /* write to 2nd location in left reverb output buffer */

       DM(I7,-1) = r10;                     /* save to input of buffer, update pointer */
       DM(all_pass_ptr6) = I7;              /* save current allpass 2 pointer address for next time */

audio_delay2:
       r5 = dm(i3, 0);                      /* get oldest sample, time delay = D_Line2 x fs*/

       /* tap inside of circular delay line 2, r0 = sampleD7_L = D7_L-th tap */
       m4 =  D7_L;  modify(i3, m3);         /* point to d-th tap */
       m4 = -D7_L;  r0 = dm(i3, m3);        /* put d-th tap in data register */
       DM(Lrev_output_taps + 6)= r0;        /* write to 6th location in left reverb output buffer */

       /* tap inside of circular delay line 2, r0 = sampleD4_R = D4_R-th tap */
       m4 =  D4_R;  modify(i3, m3);         /* point to d-th tap */
       m4 = -D4_R;  r0 = dm(i3, m3);        /* put d-th tap in data register */
       DM(Rrev_output_taps + 3)= r0;        /* write to 3rd location in left reverb output buffer */

       dm(i3, -1) = r3;                     /* write input sample to buffer */

       /* feed output back to top of tank */
       r5 = DM(recirculate2_feedback);      /* feed to other side of tank 8 */

diffusor_B1:                                /* allpass filter with variable delay */
       B7 = decay_diffuser_B1;              /* set base address to buffer */
       I7 = DM(all_pass_ptr7);              /* get previous allpass 2 pointer address */
       L7 = @decay_diffuser_B1;             /* set length of circular buffer */
       r1 = DM(decay_diffusion_1);          /* feedback gain for allpass 1*/
       r0 = r15;                            /* tank input 1 */

       r10 = DM(I7,0);                      /* load output of buffer */
       mr1f = r10;                          /* put in forground MAC register */
       mrf = mrf - r1*r0 (SSFR);            /* add to -(feedback gain)*(input) */
       mrf = SAT mrf;                       /* saturate if necessary */
       r3 = mr1f;                           /* output of all pass in R3 */

       mr1f = r0;                           /* put input of all pass in MR1F */
       mrf = mrf + r1*r3 (SSFR);            /* input + (feedback gain)*(output) */
       mrf = SAT mrf;                       /* saturate if necessary */
       r10 = mr1f;                          /* put MAC result in register file */
       DM(I7,-1) = r10;                     /* save to input of buffer, update pointer */
       DM(all_pass_ptr7) = I7;              /* save current allpass 2 pointer address for next time */

audio_delay3:
       r5 = dm(i4, 0);                      /* get oldest sample, time delay = D_Line3 x fs */

       /* tap inside of circular delay line 3, r0 = sampleD1_L = D1_L-th tap */
       m4 =  D1_L;  modify(i4, m4);         /* point to d-th tap */
       m4 = -D1_L;  r0 = dm(i4, m4);        /* put d-th tap in data register */
       DM(Lrev_output_taps)= r0;            /* write to 0'th location in left reverb output buffer */

       /* tap inside of circular delay line 3, r0 = sampleD2_L = D2_L-th tap */
       m4 =  D2_L;  modify(i4, m4);         /* point to d-th tap */
       m4 = -D2_L;  r0 = dm(i4, m4);        /* put d-th tap in data register */
       DM(Lrev_output_taps + 1)= r0;        /* write to 1st location in left reverb output buffer */
       /* tap inside of circular delay line 3,  r0 = sampleD5_R = D5_R-th tap */
       m4 =  D5_R;  modify(i4, m4);          /* point to d-th tap */
       m4 = -D5_R;  r0 = dm(i4, m4);         /* put d-th tap in data register */
       DM(Rrev_output_taps + 4)= r0;         /* write to 4th location in left reverb output buffer */

       dm(i4, -1) = r3;                      /* store current input into delay line */

comb_filter_3:
       /* comb filter 3 */
       r4 = r5;                              /* r4 = comb filter input */
       r6 = dm(damping);                     /* high frequency dampling gain */
       r7 = 0x7FFFFFFF;                      /* 0.99999 or approximately = 1 */
       r8 = r7 - r6;                         /* r8 = 1 - damping */
       r5 = DM(comb3_feedback_state);        /* read previous low pass filter output state */
       mrf = r4*r8 (SSF);                    /* scale comb filter input */
       mrf = mrf + r5*r6 (SSFR);             /* add previous filter state  */
       r10 = mr1f;                           /* r10 = comb filter output */
       dm(comb3_feedback_state) = r10;       /* replace with new filter state */

       r11 = DM(decay);
       r0 = r10*r11 (SSFR);

diffusor_B2:                                 /* allpass filter with variable delay */
       B7 = decay_diffuser_B2;               /* set base address to buffer */
       I7 = DM(all_pass_ptr8);               /* get previous allpass 2 pointer address */
       L7 = @decay_diffuser_B2;              /* set length of circular buffer */
       R1 = DM(decay_diffusion_2);           /* feedback gain for allpass 1*/
       R0 = R3;                              /* output of allpass 1 = input of allpass 2 */

       r10 = DM(I7,0);                       /* load output of buffer */
       mr1f = r10;                           /* put in forground MAC register */
       mrf = mrf + r1*r0 (SSFR);             /* add to (feedback gain)*(input) */
       mrf = SAT mrf;                        /* saturate if necessary */
       r3 = mr1f;                            /* output of all pass in R3 */

       mr1f = r0;                            /* put input of all pass in MR1F */
       mrf = mrf - r1*r3 (SSFR);             /* input - (feedback gain)*(output) */
       mrf = SAT mrf;                        /* saturate if necessary */
       r10 = mr1f;                           /* put MAC result in register file */

       /* tap inside of decay diffuser filter delay line, r0 = sampleD3_L = D3_L-th tap */
       m7 =  D3_L;  modify(i7, m7);          /* point to d-th tap */
       m7 = -D3_L;  r0 = dm(i7, m7);         /* put d-th tap in data register */
       DM(Lrev_output_taps + 2)= r0;         /* write to 2nd location in left reverb output buffer */

       /* tap inside of decay diffuser A2 filter delay line, r0 = sampleD6_R = D6_R-th tap */
       m7 =  D6_R;  modify(i7, m7);          /* point to d-th tap */
       m7 = -D6_R;  r0 = dm(i7, m7);         /* put d-th tap in data register */
       DM(Rrev_output_taps + 5)= r0;         /* write to 5th location in left reverb output buffer */

       DM(I7,-1) = r10;                      /* save to input of buffer, update pointer */
       DM(all_pass_ptr8) = I7;               /* save current allpass 2 pointer address for next time */

audio_delay4:
       r5 = dm(i5, 0);                       /* get oldest sample, time delay = D_Line4 x fs */

       /* tap inside of circular delay line 4, r0 = sampleD4_L = D4_L-th tap */
       m4 =  D4_L;  modify(i5, m5);          /* point to d-th tap */
       m4 = -D4_L;  r0 = dm(i5, m5);         /* put d-th tap in data register */
       DM(Lrev_output_taps + 3)= r0;         /* write to 3rd location in left reverb output buffer */

       /* tap inside of circular delay line 4, r0 = sampleD7_R = D7_R-th tap */
       m4 =  D7_R;  modify(i5, m5);          /* point to d-th tap */
       m4 = -D7_R;  r0 = dm(i5, m5);         /* put d-th tap in data register */
       DM(Rrev_output_taps + 6)= r0;         /* write to 6th location in left reverb output buffer */

       dm(i5, -1) = r3;                      /* save delay line input to buffer */

       /* feed output back to top of tank */
       r5 = DM(recirculate1_feedback);       /* feed to other side of tank 8 */

reverb_output_taps:
/* combine all of the left & right reverb output taps taken at different point in the holding tank */
```

```
        /* left output, all wet */
        B0 = Lrev_output_taps;                /* sum of left reverb output taps */
        R1 = 0x4CCCCCCD;                      /* scale tap outputs by 0.60  */
        R0 = DM(I0,1);                        /* load first output tap      */
        MRF = R0*R1 (SSF), R0 = DM(I0,1);            /* compute product, load next output*/
        MRF = MRF + R0*R1 (SSF), R0 = DM(I0,1);      /* compute sum of products    */
        MRF = MRF - R0*R1 (SSF), R0 = DM(I0,1);
        MRF = MRF + R0*R1 (SSF), R0 = DM(I0,1);      /* and so on ...              */
        MRF = MRF - R0*R1 (SSF), R0 = DM(I0,1);
        MRF = MRF - R0*R1 (SSF), R0 = DM(I0,1);
        MRF = MRF - R0*R1 (SSFR);
        R2 = MR1F;
        DM(reverb_left) = R2;                        /* save left reverb result */

        /* right output, all wet */
        B1 = Rrev_output_taps;                       /* sum of right reverb output taps */
        R1 = 0x4CCCCCCD;                             /* scale tap outputs by 0.60  */
        R0 = DM(I1,1);                               /* load first output tap      */
        MRF = R0*R1 (SSF), R0 = DM(I1,1);            /* compute product, load next output*/
        MRF = MRF + R0*R1 (SSF), R0 = DM(I1,1);      /* compute sum of products    */
        MRF = MRF - R0*R1 (SSF), R0 = DM(I1,1);      /* subtract product           */
        MRF = MRF + R0*R1 (SSF), R0 = DM(I1,1);      /* and so on ...              */
        MRF = MRF - R0*R1 (SSF), R0 = DM(I1,1);
        MRF = MRF - R0*R1 (SSF), R0 = DM(I1,1);
        MRF = MRF - R0*R1 (SSFR);
        R2 = MR1F;
        DM(reverb_right) = R2;                       /* save right reverb result */
        RTS;
/* ------------------------------------------------------------------------------------- */

reverb_mixer:
        r2 = 0x2AAA0000;                      /* set up scaling factor for result */
                                              /* mix input with eref & reverb result by 1/3 */
        /* mix left channel */
        r10 = DM(Left_Channel);               /* get current left input sample    */
        r11 = DM(DRY_GAIN_LEFT);              /* scale between 0x0 and 0x7FFFFFFF */
        r10 = r10 * r11(ssf);                 /* x(n) *(G_left)                  */
        mrf = r2 * r10(ssf);                  /* 0.33 * (G_left) * x(n)          */

        r1 = dm(reverb_left);                 /* get reverb result                */
        r11 = DM(WET_GAIN_LEFT);              /* scale reverb between 0x0 and 0x7FFFFFFF */
        r1 = r1 * r11 (ssf);                  /* x_reverb(n) * RG_left           */
        mrf = mrf + r1*r2 (ssf);              /* add reverb to input sample      */
        r10 = mr1f;                           /* 0.33*x(n) +0.33*x(rev_result)    */

        r1 = dm(predelay_output);
        r4 = DM(PREDEL_GAIN_LEFT);            /* scale between 0x0 and 0x7FFFFFFF */
        r3 = r1 * r4 (ssf);                   /* x_er(n) * (ER_G_left)           */
        mrf = mrf + r3*r2 (ssfr);             /* yL(n)=0.33*x(n) +0.33*x(rev_result) + 0.33*x(ear_ref) */
        r10 = mr1f;
        dm(Left_Channel) = r10;               /* output left result */

        /* mix right channel */
        r10 = DM(Right_Channel);              /* get current right input sample    */
        r11 = DM(DRY_GAIN_RIGHT);             /* scale between 0x0 and 0x7FFFFFFF */
        r10 = r10 * r11(ssf);                 /* x(n) *(G_right)                 */
        mrf = r2 * r10(ssf);                  /* 0.33 * (G_right) * x(n)         */

        r1 = dm(reverb_right);                /* get reverb result                */
        r11 = DM(WET_GAIN_RIGHT);             /* scale reverb between 0x0 and 0x7FFFFFFF */
        r1 = r1 * r11 (ssf);                  /* x_reverb(n) * RG_right          */
        mrf = mrf + r1*r2 (ssf);              /* add reverb to input sample      */
        r10 = mr1f;                           /* 0.33*x(n) +0.33*x(rev_result)    */

        r1 = dm(predelay_output);
        r4 = DM(PREDEL_GAIN_RIGHT);           /* scale between 0x0 and 0x7FFFFFFF */
        r3 = r1 * r4 (ssf);                   /* x_er(n) * (ER_G_right)          */
        mrf = mrf + r3*r2 (ssfr);             /* yR(n)=0.33*x(n) +0.33*x(rev_result) + 0.33*x(ear_ref) */
        r10 = mr1f;
        dm(Right_Channel) = r10;              /* output right result  */
        rts;
```

*3.4 **Amplitude-Based Audio Effects***

Amplitude-Based audio effects simply involve the manipulation of the amplitude level of the audio signal, from simply attenuating or increasing the volume to more sophisticated effects such as dynamic range compression/expansion. Below is a list of effects that can fall under this category:

Volume Control
Amplitude Panning (Trigonometric / Vector-Based)
Tremolo (Auto Tremolo)
"Ping-Pong" Panning (Stereo Tremolo)
Dynamic Range Control
- ***Compression***
- ***Expansion***
- ***Limiting***

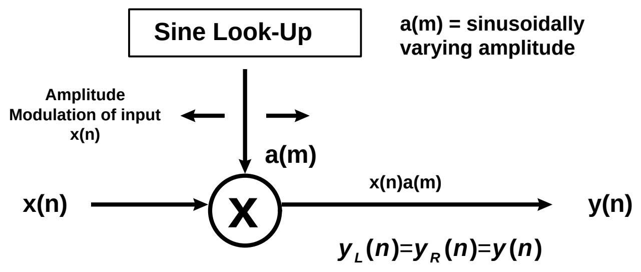
Noise Gating

3.4.1 **Tremolo - Digital Stereo Panning Effect**

Tremolo consists of *panning* the output result between the left and right output stereo channels at a slow periodic rate. This is achieved by allowing the output panning to vary in time periodically with a low frequency sinusoid. This example pans the output to the left speaker for positive sine values and pans the output to the right speaker for negative sine values (Figure 85). The analog version of this effect was used frequently on guitar and keyboard amplifiers manufactured in the '70s. A mono version of this effect (Figure 84) can be done easily by modifying the code to place the tremolo result to both speakers instead of periodically panning the result. The I/O difference equation is as follows:

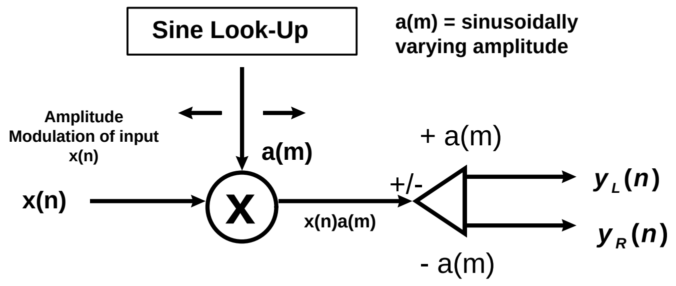
$$\boxed{y(n) = x(n) * \sin(2\pi f_{cycle} t)}$$, Mono Tremolo

Figure 84.
Mono Implementation of the Tremolo Effect



Sine Look-Up
a(m) = sinusoidally varying amplitude
Amplitude Modulation of input x(n)
a(m)
x(n)
X
x(n)a(m)
y(n)
$y_L(n)=y_R(n)=y(n)$

Figure 85.
Stereo Implementation of the Tremolo Effect



Sine Look-Up
a(m) = sinusoidally varying amplitude
Amplitude Modulation of input x(n)
a(m)
x(n)
X
x(n)a(m)
+/-
+ a(m)
- a(m)
$y_L(n)$
$y_R(n)$

For Stereo Tremolo:
when a(m) > 0
x(n)a(m) panned to left channel
right channel is zero
when a(m) < 0
x(n)a(m) panned to right channel
left channel is zero

Example Stereo Tremolo Implementation on the ADSP-21065L

```
tremolo_effect:
      r1 = DM(left_input);
      r1 = ashift r1 by -1;
      r2 = DM(right_input);
      r2 = ashift r2 by -1;
      r3 = r2 + r1;

/* generated sine value from wavetable generator, where r4 = sin(2*pi*fc*t)   */
      r4 = dm(sine_value);
      mrf = r3 * r4 (SSFR);
      r5 = mr1f;
      r4 = pass r4;           /* read current sine value to pan left or right */
                              /* if + then pan left, if - then pan right */
      IF LE JUMP pan_right_channel;

pan_left_channel:
      DM(left_output) = r5;
      r6 = 0x00000000;
      DM(right_output) = r6;
      JUMP done;

pan_right_channel:
      r6 = 0x00000000;
      DM(left_output) = r6;
      DM(right_output) = r5;

done: rts;
```

3.4.2 **Signal Level Measurement**

There are many ways to measure the amplitude of a signal. The technique described below uses a simple signal averaging algorithm to determine the signal level. It rectifies the incoming signal and averages it with the 63 previous rectified samples. Notice, however, that it only requires 6 instructions to average 64 values. This is because we are not recalculating the summation of 64 values and dividing this sum by 64 but rather updating a running average. This is how it works:

$$x_{averageold} = \frac{x[n-64] + x[n-63] + x[n-62] + \ldots + x[n-1]}{64}$$

$$x_{averageold} = \frac{x[n-64]}{64} + \frac{x[n-63]}{64} + \frac{x[n-62]}{64} + \ldots + \frac{x[n-1]}{64}$$

$$x_{averagenew} = \frac{x[n-63] + x[n-62] + x[n-61] + \ldots + x[n]}{64}$$

$$x_{averagenew} = \frac{x[n-63]}{64} + \frac{x[n-63]}{64} + \frac{x[n-61]}{64} + \ldots + \frac{x[n]}{64}$$

$$x_{averagenew} - x_{averageold} = \frac{x[n]}{64} - \frac{x[n-64]}{64}$$

$$x_{averagenew} = x_{averageold} + \frac{x[n]}{64} - \frac{x[n-64]}{64}$$

This algorithm needs to be run 64 times before $x_{averageold}$ contains a valid signal average value.

```
/* set up variables */

.segment /dm dm_variables;
.var average_line[64];
.endseg;


/*========================================================================
      Amplitude Measurement

      f15 = 1/64
      f0 = current sample
      f14 = current amplitude
      i7 = pointer to average_line
========================================================================*/

Amplitude:
      f0 = abs f0;      /* take absolute value of incoming sample */
      f0 = f0 * f15;    /* divide incoming sample by length of average line */
      f1 = dm(i7,0);    /* fetch last value in average line */
      f14 = f14 + f0;   /* add it to the running average value */
      rts(db);          /* delayed return from subroutine */
      f14 = f14 - f1;   /* subtract new sample from running average */
      dm(i7,1) = f0;    /* store new sample over old sample in average line */
```

3.4.3 **Dynamics Processing**

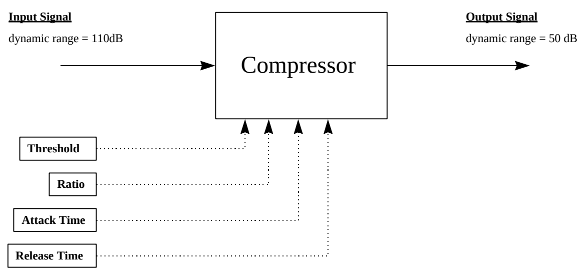
Dynamic processing algorithms are used to change the dynamic range of a signal. This means altering the distance in volume between the softest sound and the loudest sound in a signal. There are two types of dynamic processing algorithms : compressors/limiters and expanders.

3.4.3.1 **Compressors and Limiters**

The function of a compressor and limiter is to keep the level of a signal within a specific dynamic range. This is done using a technique called *gain reduction*. A gain reduction circuit reduces the amount of additional gain above a threshold setting by a certain ratio (Stark, 1996). The ultimate objective is to keep the signal from going past a specific level.

Compressors and limiters have many applications. They are used to limit the dynamic range of a signal so it can be transmitted through a medium with a limited dynamic range. An expander (covered later in this section) can then be used on the other side to expand the dynamic range back to its original levels. Compressors are also widely used in the recording industry to prevent signals from distorting as a result of overdriven mixer circuitry.

Compressors



Compressors are used to 'compress' the dynamic range of a signal

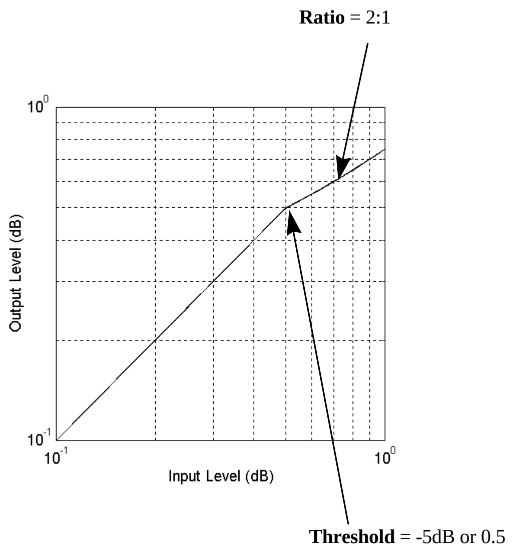
Parameters

Threshold : the level at which the dynamics processor begins adjusting the volume of the signal

Compression Ratio : level comparison of the input and output signals of the dynamics processor past the threshold

Attack Time : The amount of time it takes once the input signal has passed the threshold for the dynamics processsor to begin attenuating the signal

Release Time : The amount of time it takes once the input signal has passed below the threshold for the dynamics processor to stop attenuating the signal



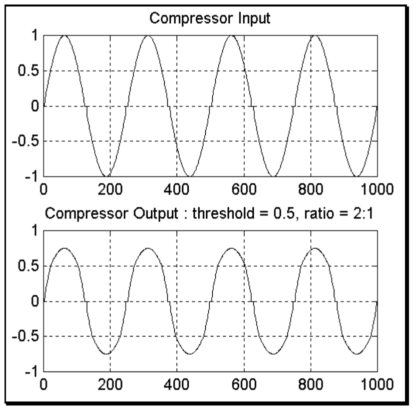
There are two primary parameters for a compressor : threshold and ratio. The threshold is the signal level at which the gain reduction begins and the ratio is the amount of gain reduction that takes place past the threshold. A ratio of 2:1, for example, would reduce the signal by a factor of two when it passed the threshold level as seen in the first compressor example below.

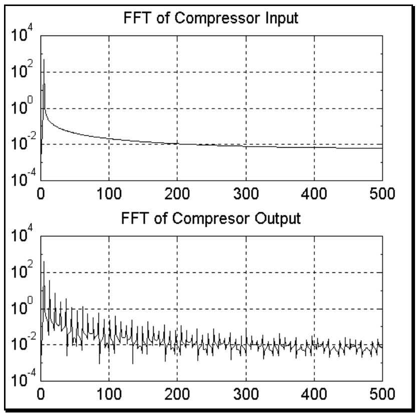
Two other parameters commonly found in compressors are attack time and release time. The attack is the amount of time it takes the compressor to begin compressing a signal once it has crossed the threshold. This helps preserve the natural dynamics of a signal. The release time, on the other hand, is the amount of time it takes the compressor to stop attenuating the signal once its level has passed below the threshold.

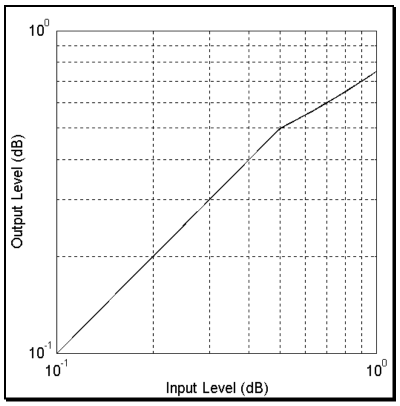
A compressor with a ratio greater than about 10:1 is considered a limiter (Stark, 1996). The effect of a limiter is more like a clipping effect than a dampening effect of a low-ratio compressor. This clipping effect can add many gross harmonics to a signal as seen in the examples below. These harmonics increase in number and amplitude as the threshold level is lowered.

The figures on the following page show the example input and output waveforms, FFTs and gain ratios of some different compressor configurations.

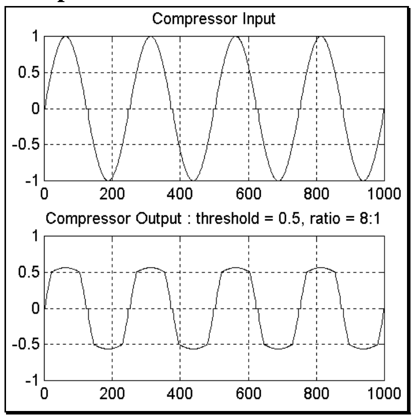
Compressor Characteristics : threshold = 0.5, ratio = 2:1

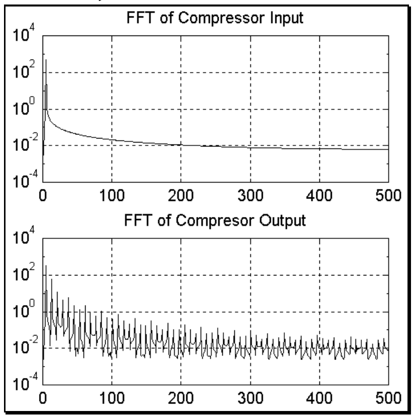


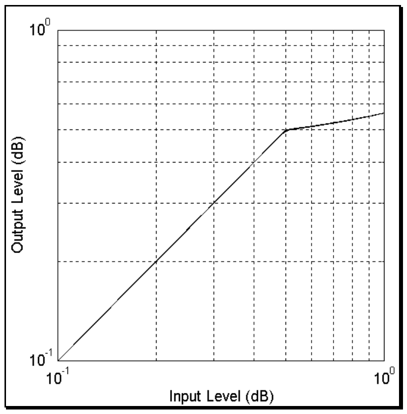




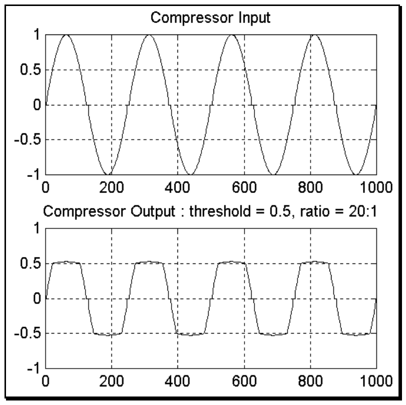
Compressor Characteristics : threshold = 0.5, ratio = 8:1

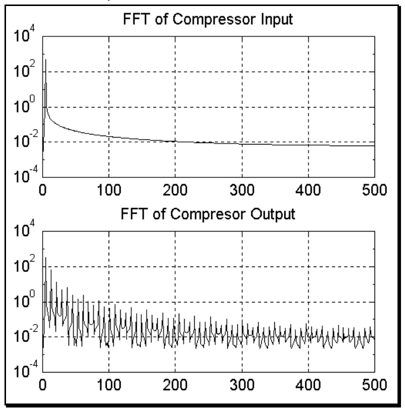


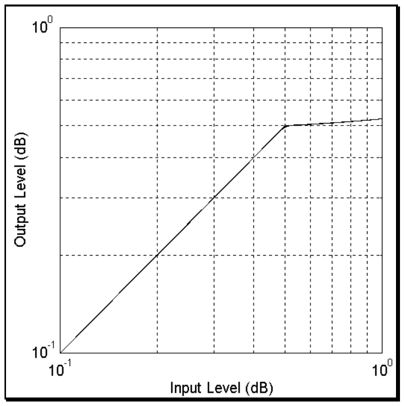




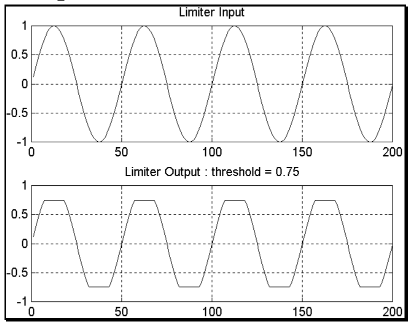
Compressor Characteristics : threshold = 0.5, ratio = 20:1

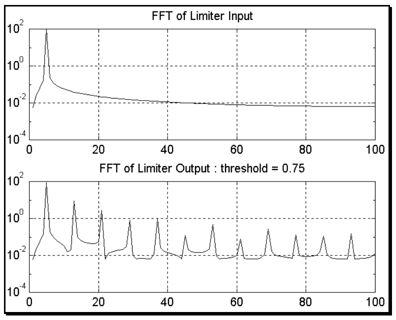


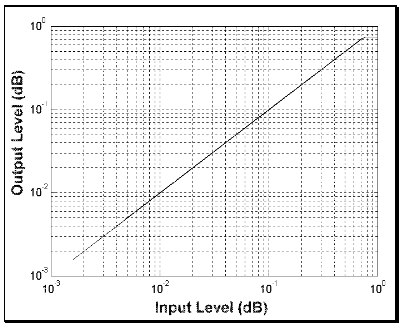




Compressor Characteristics : threshold = 0.75, ratio = ∞:1







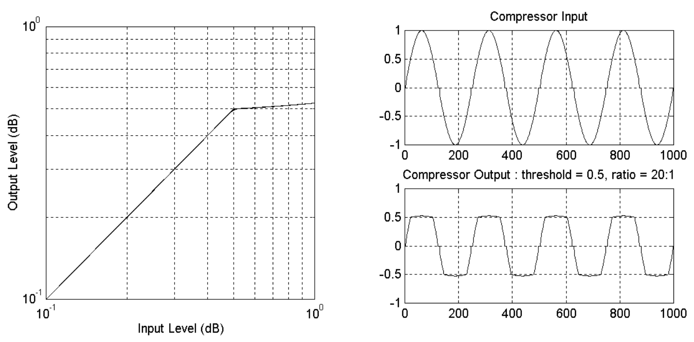
The code example below is a simple stereo compressor. This implementation does not use the attack and release parameters.

Stereo Compressor Implementation on an Analog Devices' ADSP21065L

```
/*    Stereo Compressor

      inputs:
      f2 = left channel data
      f3 = right channel data

      outputs:
      f2 = compressed left channel data
      f3 = compressed right channel data
*/

Compressor:
      f0 = 0.05;              /* f0 = ratio = 1/20 */
      f1 = 0.5;               /* f1 = threshold = 0.5 */
      f4 = abs f2;
      comp(f4,f1);            /* Is left channel past threshold? */
      if LT jump CheckRight;  /* If not, check the right channel */
      f4 = f4 - f1;           /* signal = signal - threshold */
      f4 = f4 * f0;           /* signal = signal * ratio */
      f4 = f4 + f1;           /* signal = signal + threshold *
      f2 = f4 copysign f2;    /* f2 now contains compressed left channel*/
CheckRight:
      f4 = abs f3;
      comp(f4,f1);            /* Is right channel past threshold? */
      if LT rts;              /* if not, return from subroutine */
      f4 = f4 - f1;           /* signal = signal - threshold */
      f4 = f4 * f0;           /* signal = signal * ratio */
      rts (db);               /* delayed return from subroutine */
      f4 = f4 + f1;           /* signal = signal + threshold *
      f3 = f4 copysign f3;    /* f2 now contains compressed right channel*/
```

Limiters

A limiter is a compressor with a compression ratio greater than about 10:1



The following code example is a stereo limiter with a ratio of 1:∞ - in other words, it clips the signal at a certain threshold. As seen below, this is extremely simple to do using the **clip** function.

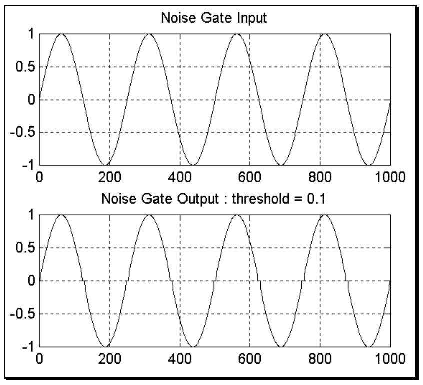
Stereo Limiter Implementation on an Analog Devices' ADSP21065L

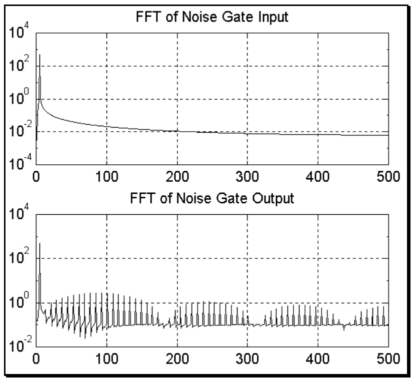
```
/*    Stereo Limiter

      Inputs:
      f2 = left channel data
      f3 = right channel data

      Outputs:
      f2 = limited left channel data
      f3 = limited right channel data
 */

Limit:
      f1 = 0.75;                  /* Threshold = .75 */
      rts (db);                   /* delayed return from subroutine */
      f2 = clip f2 by f1;         /* Limit left channel */
      f3 = clip f3 by f1;         /* Limit right channel */
```

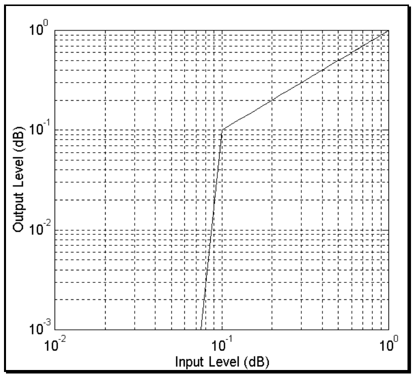
3.4.3.2 Noise Gate/Downward Expander

A noise gate or downward expander is used to reduce the gain of a signal below a certain threshold. This is useful for reducing and eliminating noise on a line when no signal is present. The difference between a noise gate and a downward expander is similar to the difference between a limiter and a compressor. A noise gate cuts signals that fall below a certain threshold while a downward expander has a ratio at which it dampens signals below a threshold.

Noise Gate Characteristics : Threshold = 0.1

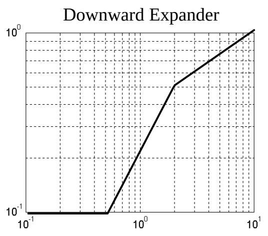


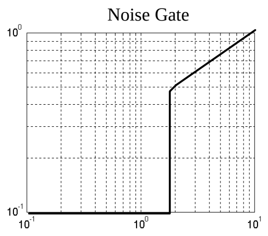




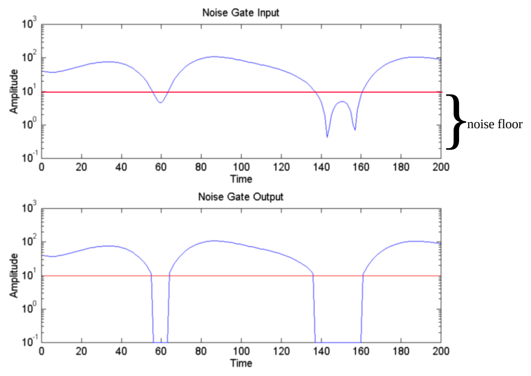
Noise Gates

Compressor : Limiter :: Downward Expander : Noise Gate





Noise Gate Example



The following code example is a stereo noise gate. This algorithm turns off the signal if its amplitude is below a certain level using RMS level detection of a signal to determine if the signal itself should be turned on or not.

Stereo Noise Gate Implementation on an Analog Devices' ADSP21065L

```
/***   NOISE_GATE.ASM   *****************************************************************
*                                                                                      *
*       ADSP-21065L EZLAB  Noise Gate Effect Program                                   *
*       Developed using ADSP-21065L EZ-LAB Evaluation Platform                         *
*                                                                                      *
*                                                                                      *
*       What the noise gate does?                                                      *
*       -------------------------                                                      *
*       Reduces the amount of gain below a certain threshold to reduce or eliminate    *
*       noise produced when no audio signal is present, while still allowing the       *
*       signal to pass thru.  This is useful after processing multiple audio effects   *
*       that can introduce noise above the noise floor of the AD1819a DACs.            *
*                                                                                      *
*       Parameters:                                                                    *
*       ----------                                                                     *
*       Threshold:  The level at which the noise gate processor begins decreasing the  *
*       volume of the signal.                                                          *
*       NOTE: Threshold values are in RMS.  This routine calculates the RMS of the     *
*       audio signal in determining if low level noise should be removed.  A running   *
*       average is not sufficent otherwise the audio signal will be severely distorted *
*                                                                                      *
*       Future Parameters that will be added in Rev 2.0:                               *
*       ---------------------------------------------                                  *
*       Attack Time:  The amount of time it takes once the input signal has passed the *
*       threshold for the dynamics processor to begin attenuating the signal.          *
*       Release Time:  The amount of time it takes once the input signal has passed    *
*       below the threshold for the dynamics processor to stop attenuating the signal      *
*                                                                                          *
*       The audio data is sent out to the AD1819A Line Outputs                             *
*                                                                                          *
*       ++ TABs set to 8 characters in VDSP IDE under TOOLS->>OPTIONS Menu                 *
*                                                                                          *
*                                            Dan Ledger & John Tomarakos                   *
*                                            ADI DSP Applications Group                    *
*                                            Revision 1.0                                  *
*                                            11/19/98                                      *
*                                                                                          *
*******************************************************************************************/

/* ADSP-21065L System Register bit definitions */
#include        "def21065l.h"
#include        "new65Ldefs.h"

.EXTERN         Left_Channel;
.EXTERN         Right_Channel;

.GLOBAL         Noise_Gate;
.GLOBAL         select_threshold;
.GLOBAL         init_averaging_buffers;


.segment /dm    noisegt;

.var    IRQ1_counter = 0x00000003;
.var    threshold = 0.04;
.var    Left_RMS_Result;
.var    Right_RMS_Result;
.var    left_float;
.var    right_float;
.var    left_RMS_squared = 0.0;
.var    right_RMS_squared = 0.0;
.var    left_RMS_line[500];             /* used to detect the RMS value of the left channel audio signal */
.var    right_RMS_line[500];            /* used to detect the RMS value of the right channel audio signal
*/

.endseg;


.segment /pm pm_code;

init_averaging_buffers:
        B6 = left_RMS_line;
        L6 = @left_RMS_line;            /* delay-line buffer pointer and length */
        m6 = 1;

        LCNTR = L6;                     /* clear delay line buffer to zero */
         DO clrDlineL UNTIL LCE;
clrDlineL:          dm(i6, m6) = 0;

        B7 = right_RMS_line;
        L7 = @right_RMS_line;           /* delay-line buffer pointer and length */
        m7 = 1;

        LCNTR = L7;                     /* clear delay line buffer to zero */
         DO clrDlineR UNTIL LCE;
clrDlineR:          dm(i7, m7) = 0;

        RTS;

/*********************************************************************************
 *                                                                               *
 *                              STEREO NOISE GATE ROUTINE                        *
 *                                                                               *
 *              inputs:                                                          *
 *              f2 = left channel data                                           *
 *              f3 = right channel data                                          *
 *                                                                               *
 *              outputs:                                                         *
 *              f2 = compressed left channel data                                *
 *              f3 = compressed right channel data                               *
 *                                                                               *
 ***********************************************************************************/

Noise_Gate:
       r2 = DM(Left_Channel);        /* left input sample */
       r3 = DM(Right_Channel);       /* right input sample */

       r1 = -31;                     /* scale the sample to the range of +/-1.0 */

       f2 = float r2 by r1;          /* convert left fixed point sample to floating point */

       f3 = float r3 by r1;          /* convert right fixed point sample to floating point */

       DM(left_float) = f2;          /* save floating point samples temporarily */
       DM(right_float) = f3;

       f15 = 0.002;                  /* 1/500 = 0.002*/
       f5 = DM(threshold);           /* f1 = Threshold = 0.1 */

RMS_left_value:
       f0 = abs f2;                  /* take absolute value of incoming left sample */
       f1 = f0;                      /* get ready to square the input */
       f0 = f0 * f1;                 /* f0 = square(abs(x)) */
       f0 = f0 * f15;                /* divide incoming squared sample by length of RMS line */
       f1 = dm(i6,0);                /* fetch oldest value in RMS line */
       f10 = DM(left_RMS_squared);   /* get previous running average of the squares of the input */
       f10 = f10 + f0;               /* add scaled squared input to the running average value */
       f10 = f10 - f1;               /* subtract oldest squared sample from running average */
       DM(left_RMS_squared) = f10;   /* save new running average of the square of the inputs samples */
       dm(i6,1) = f0;                /* store new scaled squared sample over old sample in RMS line */

       /* calculate square root of new average in f10 based on the Newton-Raphson iteration algorithm */
       f8 = 3.0;
       f2 = 0.5;
       f4 = RSQRTS f10;              /* Fetch seed */
       f1 = f4;
       f12 = f4 * f1;                /* F12=X0^2 */
       f12 = f12 * f0;               /* F12=C*X0^2 */
       f4 = f2 * f4, f12 = f8 - f12; /* F4=.5*X0, F10=3-C*X0^2 */
       f4 = f4 * f12;                /* F4=X1=.5*X0(3-C*X0^2) */
       f1 = f4;
       f12 = f4 * f1;                /* F12=X1^2 */
       f12 = f12 * f0;               /* F12=C*X1^2 */
       f4 = f2 * f4, f12 = f8 - f12; /* F4=.5*X1, F10=3-C*X1^2 */
       f4 = f4 * f12;                /* F4=X2=.5*X1(3-C*X1^2) */
       f1 = f4;
       f12 = f4 * f1;                /* F12=X2^2 */
       f12 = f12 * f0;               /* F12=C*X2^2 */
       f4 = f2 * f4, f12 = f8 - f12; /* F4=.5*X2, F10=3-C*X2^2 */
       f4 = f4 * f12;                /* F4=X3=.5*X2(3-C*X2^2) */
       f10 = f4 * f10;               /* X=sqrt(Y)=Y/sqrt(Y) */
       DM(Left_RMS_Result) = f10;

gate_left:
       f2 = DM(left_float);
       f10 = abs f10;                /* get absolute value of running average */
       comp(f10,f5);                 /* compare to desired threshold */
       if LT f2 = f2 - f2;           /* if left channel < threshold, left channel = 0.0 */

       /* send gated results to left DAC channel */
       r1 = 31;                      /* scale the result back up to MSBs */
       r2 = fix f2 by r1;            /* convert back to fixed point number */
       DM(Left_Channel) = r2;

RMS_right_value:
       f0 = abs f3;                  /* take absolute value of incoming right sample */
       f1 = f0;                      /* get ready to square the input */
       f0 = f0 * f1;                 /* f0 = square(abs(x)) */
       f0 = f0 * f15;                /* divide incoming squared sample by length of RMS line */
       f1 = dm(i7,0);                /* fetch oldest value in RMS line */
       f10 = DM(right_RMS_squared);  /* get previous running average of the squares of the input */
       f10 = f10 + f0;               /* add scaled squared input to the running average value */
       f10 = f10 - f1;               /* subtract oldest squared sample from running average */
       DM(right_RMS_squared) = f10;  /* save new running average of the square of the inputs samples */
       dm(i7,1) = f0;                /* store new scaled squared sample over old sample in RMS line */
```

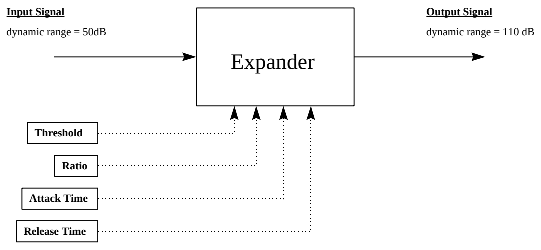
```
        /* caclulate square root of new average in f10 based on the Newton-Raphson iteration algorithm */
        f8 = 3.0;
        f2 = 0.5;
        f4 = RSQRTS f10;               /* Fetch seed */
        f1 = f4;
        f12 = f4 * f1;                 /* F12=X0^2 */
        f12 = f12 * f0;                /* F12=C*X0^2 */
        f4 = f2 * f4, f12 = f8 - f12;  /* F4=.5*X0, F10=3-C*X0^2 */
        f4 = f4 * f12;                 /* F4=X1=.5*X0(3-C*X0^2) */
        f1 = f4;
        f12 = f4 * f1;                 /* F12=X1^2 */
        f12 = f12 * f0;                /* F12=C*X1^2 */
        f4 = f2 * f4, f12 = f8 - f12;  /* F4=.5*X1, F10=3-C*X1^2 */
        f4 = f4 * f12;                 /* F4=X2=.5*X1(3-C*X1^2) */
        f1 = f4;
        f12 = f4 * f1;                 /* F12=X2^2 */
        f12 = f12 * f0;                /* F12=C*X2^2 */
        f4 = f2 * f4, f12 = f8 - f12;  /* F4=.5*X2, F10=3-C*X2^2 */
        f4 = f4 * f12;                 /* F4=X3=.5*X2(3-C*X2^2) */
        f10 = f4 * f10;                /* X=sqrt(Y)=Y/sqrt(Y) */
        DM(Right_RMS_Result) = f10;

gate_right:
        f3 = DM(right_float);
        f10 = abs f10;
        comp(f10,f5);
        if LT f3 = f3 - f3;            /* if right channel < threshold, right channel = 0 */

        /* send gated results to right DAC channel */
        r1 = 31;                       /* scale the result back up to MSBs */
        r3 = fix f3 by r1;             /* convert back to fixed point number */
        DM(Right_Channel) = r3;
        rts;
```

3.4.3.3 Expanders

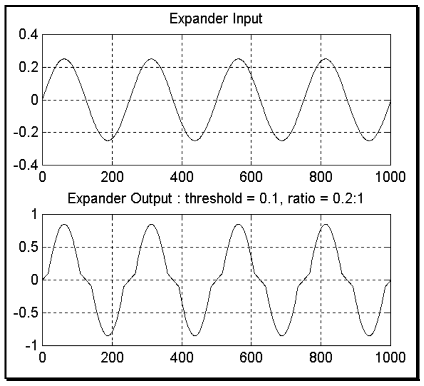
An expander is a device used to increase the dynamic range of a signal and complement compressors. For example, a signal with a dynamic range of 70 dB might pass through an expander and exit with a new dynamic range of 100 dB. These can be used to restore a signal that was altered by a compressor.

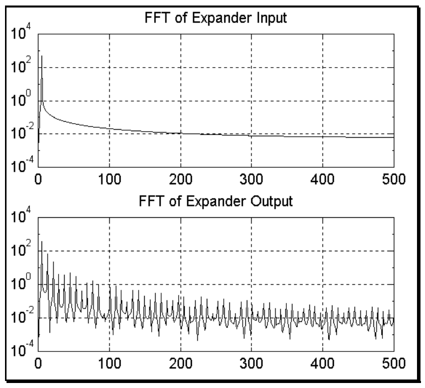
Below are the properties of an expander.

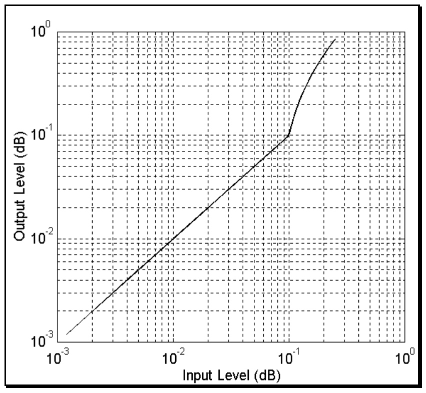
Expanders



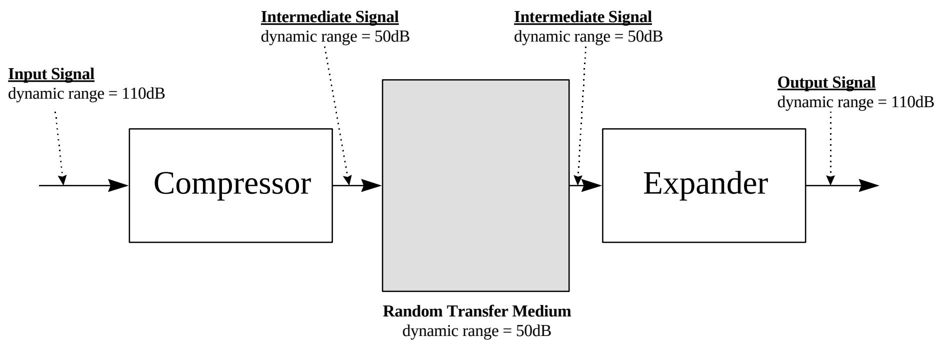
Expanders are used to ‘expand’ the dynamic range of a signal







Compressor-Expander Application Example



```
/***   COMPANDER.ASM   *********************************************************************
*                                                                                          *
*      ADSP-21065L EZ-LAB Companding Effect Program                                        *
*      Developed using ADSP-21065L EZ-LAB Evaluation Platform                              *
*                                                                                          *
*                                                                                          *
*      What the compander does?                                                            *
*      -------------------------                                                           *
*      Combination of a compressor and expander.  The peaks signal levels above the        *
*      upper threshold are compressed, while lower level signals above the lower           *
*      threshold are expanded.                                                             *
*                                                                                          *
*      Parameters:                                                                         *
*      ----------                                                                          *
*      Threshold:  The level at which the dynamics processor begins adjusting the          *
*      volume of the signal.                                                               *
*      Compression Ratio:  Level comparison of the input and output signals of the         *
*       dynamics processor past the threshold.                                             *
*                                                                                          *
*      Future Parameters that will be added in Rev 2.0:                                    *
*      ---------------------------------------------                                       *
*      Attack Time:  The amount of time it takes once the input signal has passed the      *
*      threshold for the dynamics processor to begin attenuating the signal.               *
*      Release Time:  The amount of time it takes once the input signal has passed         *
*       below the threshold for the dynamics processor to stop attenuating the signal      *
*********************************************************************************************/

/* ADSP-21065L System Register bit definitions */
#include        "def21065l.h"
#include        "new65Ldefs.h"

.EXTERN         Left_Channel;
.EXTERN         Right_Channel;

.GLOBAL         Stereo_Compander;
.GLOBAL         select_compander_ratios;
.GLOBAL         select_compander_thresholds;


.segment /dm    compand;

.var    IRQ1_counter = 0x00000003;
.var    IRQ2_counter = 0x00000003;

.var    comp_ratio = 0.05;
.var    comp_threshold = 0.5;
.var    expan_ratio = 1.5;
.var    expan_threshold = 0.2;

.endseg;


.segment /pm pm_code;

/*************************************************************************************
 *                                                                                   *
 *                        STEREO COMPANDER ROUTINE                                   *
 *                                                                                   *
 *************************************************************************************/

Stereo_Compander:
        r2 = DM(Left_Channel);          /* left input sample */
        r3 = DM(Right_Channel);         /* right input sample */

        r1 = -31;                       /* scale the sample to the range of +/-1.0 */

        f2 = float r2 by r1;            /* convert fixed point sample to floating point */

        f3 = float r3 by r1;            /* convert fixed point sample to floating point */

        f0 = DM(comp_ratio);            /* f0 = ratio = 1/20 */
        f1 = DM(comp_threshold);        /* f1 = threshold = 0.5 */

compress_left:
        f4 = abs f2;
        comp(f4,f1);                    /* Is left channel past threshold? */
        if LT jump compress_right;      /* If not, check the right channel */
        f4 = f4 - f1;                   /* signal = signal - threshold */
        f4 = f4 * f0;                   /* signal = signal * ratio */
        f4 = f4 + f1;                   /* signal = signal + threshold */
        f2 = f4 copysign f2;            /* f2 now contains compressed left channel*/

compress_right:
        f4 = abs f3;
        comp(f4,f1);                    /* Is right channel past threshold? */
        if LT jump expansion;           /* if not, return from subroutine */
        f4 = f4 - f1;                   /* signal = signal - threshold */
        f4 = f4 * f0;                   /* signal = signal * ratio */
        f4 = f4 + f1;                   /* signal = signal + threshold */
        f3 = f4 copysign f3;            /* f3 now contains compressed right channel*/

expansion:
        f0 = DM(expan_ratio);           /* f0 = ratio = 1.4 */
        f1 = DM(expan_threshold);       /* f1 = threshold = 0.2 */

expand_left:
        f4 = abs f2;
        comp(f4,f1);                    /* Is left channel past threshold? */
        if LT jump expand_right;        /* If not, check the right channel */
```

```
       f4 = f4 - f1;                /* signal = signal - threshold */
       f4 = f4 * f0;                /* signal = signal * ratio */
       f4 = f4 + f1;                /* signal = signal + threshold */
       f2 = f4 copysign f2;         /* f2 now contains compressed left channel*/

expand_right:
       f4 = abs f3;
       comp(f4,f1);                 /* Is right channel past threshold? */
       if LT jump finish_processing; /* if not, return from subroutine */
       f4 = f4 - f1;                /* signal = signal - threshold */
       f4 = f4 * f0;                /* signal = signal * ratio */
       f4 = f4 + f1;                /* signal = signal + threshold */
       f3 = f4 copysign f3;         /* f3 now contains compressed right channel*/

finish_processing:
       r1 = 31;                     /* scale the result back up to MSBs */
       r2 = fix f2 by r1;           /* convert back to fixed point number */
       r3 = fix f3 by r1;           /* convert back to fixed point number */

       /* send companded results to left and right DAC channels */
       DM(Left_Channel) = r2;
       DM(Right_Channel) = r3;

       rts;
```

3.5 Sound Synthesis Techniques

Sound synthesis is a technique used to create specific waveforms. It is widely used in the audio market in products like sound cards and synthesizers to digitally recreate musical instruments and other sound effects. The most simple forms of sound synthesis such as FM and Additive synthesis use basic harmonic recreation of a sound using the addition and multiplication of sinusoids of varying frequency, amplitude and phase. Sample playback and wavetable synthesis use digital recordings of a waveform played back at varying frequencies to achieve life-like reproductions of the original sound. Subtractive Synthesis and Physical Modeling attempt to simulate the physical model of an acoustic system.

3.5.1 **Additive Synthesis**

Fourier theory dictates that any periodic sound can be constructed of sinusoids of various frequency, amplitude and phase [25]. Additive synthesis is the processes of summing such sinusoids to produce a wide variety of envelopes. By varying the three fundamental properties : frequency, amplitude and phase over time, additive synthesis can accurately reproduce a variety instruments.
In comparison to all other synthesis techniques, additive synthesis can require a significant amount of processing power based on the number of sinusoidal oscillators used. There is a direct relationship between the number of harmonics generated and the number of processor cycles required. Below is the basic formula.

$$y(n) = A_1 \sin(2\Pi f_1 n + f_1) + A_2 \sin(2\Pi f_2 n + f_2) + A_3 \sin(2\Pi f_3 n + f_3)...$$

3.5.2 **FM Synthesis**

FM Synthesis is similar to additive synthesis in that it uses simple sinusoids to create a wide range of sounds. FM synthesis, however, uses one finite formula to create an infinite number harmonics. The FM synthesis equation shown below uses a fundamental sinusoid which is modulated by another sinusoid.

$$y(n) = A(n)\sin\left(2\Pi f_c n + I(n)\sin(2\Pi f_m n)\right)$$

When this equation is expanded, we can see that an infinite number of harmonics are created.

$$y(n) = J_1(n)\sin(2\Pi f_c n)$$

$$+J_1(n)[\sin(2\Pi(f_c + f_m)n) - \sin(2\Pi(f_c - f_m)n)]$$
$$+J_2(n)[\sin(2\Pi(f_c + 2f_m)n) - \sin(2\Pi(f_c - 2f_m)n)]$$
$$+J_3(n)[\sin(2\Pi(f_c + 3f_m)n) - \sin(2\Pi(f_c - 3f_m)n)]\ldots$$ [2]

Because this method is very computationally efficient, it is widely used in the sound card and synthesizer industry.

3.5.3 Wavetable Synthesis

Wavetable synthesis is a popular and efficient technique for synthesizing sounds, especially in sound cards and synthesizers. Using a lookup table of pre-recorded waveforms, the wavetable synthesis engine repeatedly plays the desired waveform or combinations of multiple waveforms to simulate the timbre of an instrument. The looped playback of the sample can also be modulated by an amplitude function which controls its attack, decay, sustain and release to create an even more realistic reconstruction of the original instrument.

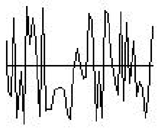


Figure 14:
One waveform period stored in wavetable.

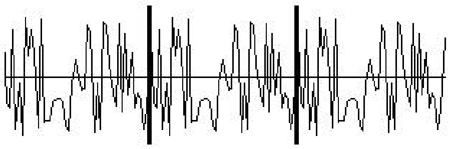


Figure 15:
Waveforms added together and repeated over time

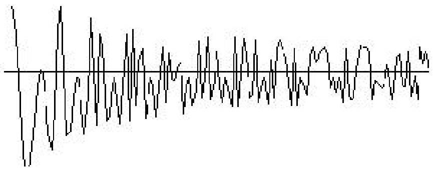


Figure 16 :
Repeated waveform modulated by an amplitude envelop.

This method of synthesis is simple to implement and is computationally efficient. In a DSP, the desired waveforms can be loaded into a circular buffers to allow for zero-overhead looping. The only real computational operations will be adding multiple waveforms, calculating the amplitude envelope and modulating the looping sample with it. The downside of wave-table synthesis is that it is difficult to approximate rapidly changing spectra.[1]

3.5.4 **Sample Playback**

Sample Playback is another computationally efficient synthesis technique that yields extremely high sound quality. An entire sample is stored in memory for each instrument which is played back at a selected pitch. Often times, these sample will have loop points within them which can be used to alter the duration of the sustain thus giving an even more life-like reproduction of the sound.

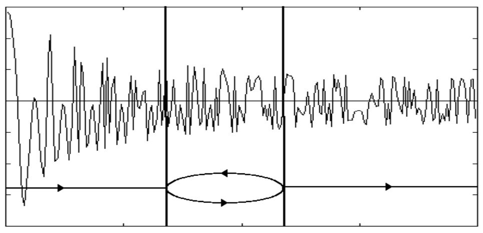


Figure 17

Although this method is capable of producing extremely accurate reproductions of almost any instrument, it requires large amounts of memory to hold the sampled instrument data. For example, to duplicate the sound of a grand piano, the sample stored in memory would have to be about 5 seconds long. If this sample were stereo and sampled at 44.1kHz, this single instrument would require 441,000 Words of memory! To recreate many octaves of a piano, the system would require multiple piano samples because slowing down a sample of a C5 on a piano to the pitch of a C2 will sound nothing like an actual C2. This technique is widely used in high-end keyboards and sound cards. Just like wavetable synthesis, sample playback requires very little computational power. It can be easily implemented in a DSP using circular buffers with simple loop-point detection.

3.5.5 Subtractive Synthesis

Subtractive synthesis begins with a signal containing all of the required harmonics of a signal and selectively attenuating (or boosting) certain frequencies to simulate the desired sound[2]. The amplitude of the signal can be varied using an envelope function as in the other simple synthesis techniques. This technique is effective at recreating instruments that use impulse-like stimulus like a plucked string or a drum.

4. CONCLUSION

We have explored many of the basics in selecting the ADSP-21065L for use in digital audio applications. There are many different DSPs on the market today and chances are there is one that fits your design needs perfectly. Because of this, it is important to fully understand the type of algorithms and the amount of processing power that an application will require before selecting a DSP. This paper has presented a subset of the expanding number of audio applications for DSPs and provided some insight into their functionality and implementation. As DSPs become faster and more powerful, we will undoubtedly witness new creative and ingenious DSP audio applications.

Appendix - References

[1] Steven W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, CA, (1998)

[2] S. J. Orfanidis, *Introduction to Signal Processing*, Chapter 8, Sec 8.2, pp. 355-383, Prentice Hall, Englewood Cliffs, NJ, (1996).

[3] J. G. Proakis & D. G. Manolakis, Introduction To Digital Signal Processing, Macmillan Publishing Company, New York, NY, (1988)

[4] A. V. Oppenheim and R. W. Schafer*, Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, (1989)

[5] P. Lapsley, J. Bier, A. Shoham and E. A. Lee*, DSP Processor Fundamentals: Architectures and Features*, Berkley Design Technology, Inc., Fremont, CA, (1996)

[6] Jon Dattorro, "Effect Design, Part 2: Delay-Line Modulation and Chorus," *J. Audio Engineering Society*, 10, pp. 764-788, October 1997

[7] Scott Lehman, *Harmony Central Effects Explained*, Internet: Http://www.harmony-central.com/Effects/Articcles, (1996)

[8] Jack W. Crenshaw, *Programmer's Toolbox: Putting It Together*, Embedded Systems Programming, pp. 9-24, August 1995

[9] R. Wilson, "Filter Topologies", *J. Audio Engineering Society*, Vol 41, No. 9, September 1993

[10] Udo Zolzer, "Roundoff Error Analysis of Digital Filters", *J. Audio Engineering Society*, Vol42, No. 4, April 1994

[11] J.A Moorer, "About This Reverberation Business," *Computer Music Journal*, **3**, 13 (1979).

[12] M.R. Schroeder, "Natural Sounding Artificial Reverberation," *J. Audio Eng*, Soc., **10**, p. 219, (1962).

[13] M.R. Schroeder, "Digital Simulation of Sound Transmission in Reverberant Spaces", *J. Acoust. Soc. Am.,* **47**, p. 424, (1970).

[14] D. Griesinger, "Practical Processors and Programs for Digital Reverberation," *Audio in Digital Times, Proc. Audio Eng. Soc. 7th Inter. Conf.,* Toronto, Ont., Canada, May 14th-17th 1989, pp. 187-195.

[15] K. L. Kloker, B. L. Lindsley, C.D. Thompson, "VLSI Architectures for Digital Audio Signal Processing," *Audio in Digital Times, Proc. Audio En g. Soc. 7th Inter. Conf.,* Toronto, Ont., Canada, May 14th-17th 1989, pp. 313-325

[16] K. Bogdanowicz & R. Belcher, "Using Multiple Processor for Real-Time Audio Effects", *Audio in Digital Times, Proc. Audio En g. Soc. 7th Inter. Conf.,* Toronto, Ont., Canada, May 14th-17th 1989, pp. 337-342

[17] Gary Davis & Ralph Jones, *Sound Reinforcement Handbook*, 2nd Edition", **Ch. 14**, pp. 259-278, Yamaha Corporation of America, (1989, 1990)

[18] Analog Devices, Inc, ADSP-2106x SHARC User's Manual, Second Edition, Analog Devices, 3 Technology Way, Norwood, MA (1996)

[19] Analog Devices Whitepaper, *ADSP-21065L: Low-Cost 32-bit Processing for High Fidelity Digital Audio*, Analog Devices, 3 Technology Way, Norwood, MA, November 1997

[20] J. Dattorro, "The Implementation of Digital Filters for High Fidelity Audio", *Audio in Digital Times, Proc. Audio En g. Soc. 7th Inter. Conf.,* Toronto, Ont., Canada, May 14th-17th 1989, pp. 165-180.

[21] W. Chen, "Performance of Cascade and Parallel IIR Filters," *Audio in Digital Times, Proc. Audio En g. Soc. 7th Inter. Conf.,* Toronto, Ont., Canada, May 14th-17th 1989, pp. 148-158

[22] V. Pulkki, "Virtual Sound Source Positioning Using Vector Base Amplitude Panning", *J. Audio Engineering Soc.,* Vol. 45, No. 6, pp. 456-466, June 1997

[23] S. J. Orfanidis, "Digital Parametric Equalizer Design with Prescribed Nyquist-Frequency Gain," *J. Audio Engineering Soc.*, Vol. 45, No. 6, pp. 444 - 455, June 1997

[24] D. C. Massie, “An Engineering Study of the Four-Multiply Normalized Ladder Filter,” *J. Audio Engineering Soc*., Vol.41, No. 7/8, pp. 564-582, July/August 1993

[25] Gregory Sandell, “Perceptual Evaluation of Principal-Component-Based Synthesis of Musical Timbres”, *J. Audio Engineering Soc*., Vol.43, No. 12, pp.1013-1027, December 1995

[26] C. Anderton, *Home Recording for Musicians*, Amsco Publications, New York, NY, (1996)

[27] B. Gibson, The AudioPro Home Recording Course, MixBooks, Emeryville, CA, (1996)

[28] D. P. Weiss, “Experiences with the AT&T DSP32 Digital Signal Processor in Digital Audio Applications,” *Audio in Digital Times, Proc. Audio En g. Soc. 7th Inter. Conf.,* Toronto, Ont., Canada, May 14th-17th 1989, pp. 343-351

[29] J. Bier, P. Lapsley, and G. Blalock, “Choosing a DSP Processor,” *Embedded Systems Programming*, pp. 85-97, (October 1996)

[30] Jon Dattorro, “Effect Design, Part 1: Reverberator and Other Filters,” *J. Audio Engineering Society*, 10, pp. 660-684, September 1997

[31] *ME-5 Guitar Multiple Effects Processor*, User Manual, Roland/Boss Corp., 1990.

[32] Dominic Milano, *Multi-Track Recording, A Technical And Creative Guide For The Musician And Home Recorder*, Reprinted from *The Keyboard Magazine*, **Ch. 2**, pp. 37 - 50. Hal Leonard Books, 8112 W. Bluemound Road, Milwaukee, WI (1988)

[33] R. Bristow-Johnson, “A Detailed Analysis of a Time-Domain Formant-Corrected Pitch-Shifting Algorithm”, *J. Audio Engineering Soc.*, Vol. 43, No. 5, May 1995

[34] R. Adams & T. Kwan, “Theory and VLSI Architectures for Asynchronous Sample Rate Converters*”, J. Audio Engineering Soc*., Vol. 41, No. 7/8, pp. 539-555, July/August 1993

[35] R. Bristow-Johnson, “The Equivalence of Various Methods of Computing Biquad Coefficients for Audio Parametric Equalizers”, presented at AES 97th Convention*, J. Audio Engineering Soc. (Abstracts Preprint 3096)*, Vol 42, pp. 1062-1063, (December 1994)

[36] D. J. Shpak, “Analytical Design of Biquadratic Filter Sections for Parametric Filters”, J. Audio Engineering Soc., Vol 40, No 11, pp. 876-885, (November 1992)

[37] NVision*, THE BOOK: An Engineer’s Guide To The Digital Transition - CH2: Designing A Digital Audio System*, Internet: http://www.nvision1.com/thebook/chapter2.htm, NVision, Inc. Nevada City CA

[38] L. D. Fielder, "Human Auditory Capabilities and Their Consequences in Digital-Audio Converter Design", *Audio in Digital Times, Proc. Audio En g. Soc. 7th Inter. Conf*., Toronto, Ont., Canada, May 14th-17th 1989, pp. 45-62.

[39] A. Chrysafis, "Digital Sine-Wave Synthesis Using the DSP56001/2", Application Note, Semiconductor Products Sector, Motorola, Inc., Austin, TX, (1988)