1. Introduction to Sound Design in Csound

Richard Boulanger

Click HERE to download zipped instruments and samples for this chapter.

Csound is an incredibly powerful and versatile software synthesis program. Drawing from a toolkit of over 450 signal processing modules, one can use Csound to model virtually any commercial synthesizer or multi-effects processor. Csound literally transforms a personal computer into a high-end digital audio workstation — an environment in which the worlds of sound-design, acoustic research, digital audio production and computer music composition all join together in the ultimate expressive instrument. However, as with every musical instrument, true virtuosity is literally the product of both talent and dedication. You will soon discover that Csound is the ultimate musical instrument. But you must practice! In return, it will reward your commitment by producing some of the richest textures and uniquely beautiful timbres you have ever heard. In the audio world of Csound, knowledge and experience are the key... and your imagination the only limitation.

The goal of this chapter is to get you started on Csound's road of discovery and artistry. Along the way we'll survey a wide range of synthesis and signal processing techniques and we'll see how they're implemented in Csound. By the end we'll have explored a good number of Csound's many possibilities. I encourage you to render, listen, study and modify each of my simple tutorial instruments. In so doing, you'll acquire a clear understanding and appreciation for the language while laying down a solid foundation upon which to build your own personal library of original and modified instruments. Furthermore, working through the basics covered here will prepare you to better understand, appreciate and apply the more advanced synthesis and signal processing models that are presented by my colleagues and friends in the subsequent chapters of this book.

Now there are thousands of Csound instruments and hundreds of Csound compositions on the CD-ROM, that accompanies this text. Each opens a doorway into one of Csound's many worlds. In fact, it would take a lifetime to fully explore them all. Clearly, one way to go would be to compile all the orchestras on the CD-ROM, select the ones that sound most interesting to you and merely "sample" them for use in your own compositions. This library of "presets" might be just the collection of unique sounds you were searching for and your journey would be over.

However, I believe a better way to go would be to read, render, listen and then study the synthesis and signal processing techniques that fascinate you most by modifying existing Csound orchestras that employ them. Afterward you should express this understanding through your own compositions — your own timbre-based "soundscapes" and "sound collages." Surely through this "active" discovery process, you will begin to develop your own personal Csound library and ultimately your own "voice."

To follow the path I propose, you'll need to understand the structure and syntax of the Csound language. But I am confident that with this knowledge, you'll be able to translate your personal audio and synthesis experience into original and beautiful Csound-based synthetic instruments and some truly unique and vivid sound sculptures.

To that end, we'll begin by learning the structure and syntax of Csound's text-based orchestra and score language. Then we'll move on to explore a variety of synthesis algorithms and Csound programming techniques. Finally we'll advance to some signal processing examples. Along the way, we'll cover some basic digital audio concepts and learn some software synthesis programming tricks. To better understand the algorithms and the signal flow, we'll "block-diagram" most of our Csound "instruments." Also, I'll assign a number of exercises that will help you to fully understand the many ways that you can actually "work" with the program.

Don't skip the exercises. And don't just read them. Do them! They are the keys to developing real fluency with the language. In fact, you may be surprised to discover that these exercises "teach" you more about how to "work" with Csound than any of the descriptions that precede them. In the end, you should have a good strong foundation upon which to build your own library of Csounds and you will have paved the way to a deeper understanding of the chapters that follow.

So, follow the instructions on the CD-ROM; install the Csound program on your computer; render and listen to a few of the test orchestras to make sure everything is working properly; and then let's get started!

What is Csound and How Does it Work?

Csound is a sound renderer. It works by first translating a set of text-based *instruments*, found in the *orchestra file*, into a computer data-structure that is machine-resident. Then, it *performs* these user-defined instruments by interpreting a list of *note* events and

parameter data that the program "reads" from: a text-based *score file,* a sequencer-generated *MIDI file,* a real-time *MIDI controller,* real-time *audio*, or a non-MIDI devices such as the ASCII keyboard and mouse.

Depending on the speed of your computer (and the complexity of the instruments in your orchestra file) the performance of this "score" can either be auditioned in real-time, or *written* directly into a file on your hard disk. This entire process is referred to as "sound rendering" as analogous to the process of "image rendering" in the world of computer graphics.

Once rendered, you will listen to the resulting soundfile by opening it with your favorite sound editor and playing it either through the built-in digital-to-analog converter (DAC) on your motherboard or the DAC on your PC Sound Card.

Thus, in Csound, we basically work with two interdependent and complimentary text files, the orchestra file and the score file. These files can be given any name you would like. Typically, we give the two files the same name and differentiate between them by a unique three letter extension — *.orc* for the orchestra file and *.sco* for the score file. Naming is up to you. In this chapter I have called the files *etude1.orc* and *etude1.sco*, *etude2.orc* and *etude2.sco*, *etude3.orc* and *etude3.sco*, etc. These "etude" orchestras contain six instruments each (*instr 101 — 106*, *instr 107 — 112*, *instr 113 — 118*, etc.). From these multi-instrument orchestras I have also created a set of single instrument orchestras to make it easier for you to isolate and experiment on individual instruments. These are named after the instrument number itself (*101.orc* and *101.sco*, *102.orc* and *102.sco*, *117.orc* and *117.sco*, etc.). Naming the corresponding score file the same as the orchestra file will help you keep your instrument library organized and I highly recommend you do the same. In fact, all the scores and orchestras in The Csound Book and on the accompanying CD-ROM follow this naming convention.

The Orchestra File

The Csound orchestra file consists of two parts: the *header* section and the *instrument* section.

The Header Section

In the header section you define the sample and control rates that the instruments will be rendered and you specify the number of channels in the output. The orchestral header that we will use throughout the text is:

```
sr      =    44100
kr      =    4410
ksmps   =    10
nchnls  =    1
```

Figure 1.1 Csound's default orchestral "header."

The code in this header assigns the sample rate (*sr*) to 44.1 K (44100), the control rate (*kr*) to 4410 and ksmps to 10 (*ksmps = sr/kr*). The header also indicates that this orchestra should render a mono soundfile by setting the number of channels (*nchnls*) to 1. (If we wanted to render a stereo sound file, we would simply set *nchnls* to 2).

The Instrument Section

In Csound, instruments are defined (and thus designed) by interconnecting "modules" or *opcodes* that either generate or modify signals. These signals are represented by *symbols*, *labels* or *variable names* that can be "patched" from one opcode to another. Individual instruments are given a unique instrument number and are delimited by the **instr** and **endin** statements. A single orchestra file can contain virtually any number of instruments. In fact, in Csound "everything" is an instrument — your 8000 voice sampler, your 4000 voice FM synth, your 2000 voice multi-model waveguide synth, your 1000 band EQ, your 500 channel automated mixer, your 250 tap delay-line, Fractal-flanger, convolution-reverb, vetor-spatializer, whatever... To the Csound program each of these very different pieces of synthesis, signal processing and studio gear are merely *instr 1, instr 2, instr 3, instr 4,* etc.

The Orchestra Syntax

In the Csound orchestra file, the syntax of a generic opcode statement is:

```
Output Opcode       Arguments, ... , ...      ; Comments
                                              (optional)
```

In the case of the **oscil** opcode, this translates into the following syntax:

```
Output          Amplitude Frequency F-table #  ; Comment
a1      oscil   10000,    440,      1          ; oscillator
```

Sound Design Etude 1: A Six Instrument Orchestra

etude1.orc etude1.sco

In our first orchestra file *instr 101* uses a table-lookup oscillator opcode, **oscil**, to compute a 440 Hz sine tone with amplitude of 10000. A block diagram of *instr 101* is show in figure 1.2 and the actual Csound orchestra code for this instrument is shown in figure 1.3.

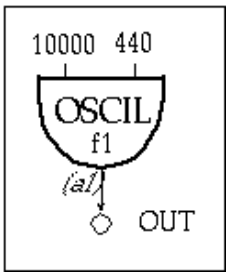


Figure 1.2 Block diagram of *instr 101*, a simple fixed frequency and amplitude table-lookup oscillator instrument.

```
     instr 101       ; SIMPLE OSCIL
a1   oscil 10000, 440, 1
     out   a1
     endin
```

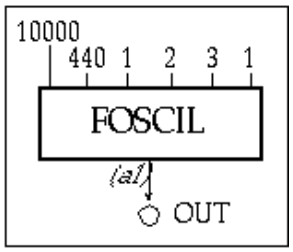
Figure 1.3 Orchestra code for *instr 101*, a fixed frequency and amplitude instrument using Csound's table-lookup oscillator opcode, **oscil**.

The block diagram of *instr 101* clearly shows how the output of the oscillator, labeled *a1*, is "patched" to the input of the **out** opcode that *writes* the signal to the hard-disk.

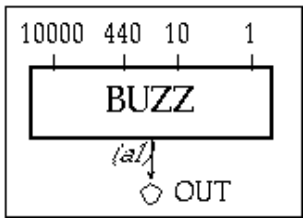
Csound renders instruments line-by-line, from top to bottom. Input arguments are on the right of the opcode name. Outputs are on the left. Words that follow a semi-colon (;) are ignored. They are considered to be comments.

In *instr 101*, as show in figure 1.3, the input arguments to the oscillator are set at 10000 (amplitude), 440 (frequency) and 1 (for the function number of the waveshape template that the oscillator "reads"). The oscillator opcode renders the sound 44100 times a second with these settings and writes the result into the variable named *a1*. The sample values in the local-variable *a1* can then be read as inputs by subsequent opcodes, such as the **out** opcode. In this way, variable names function like "patch cords" on a traditional analog synthesizer. And with these "virtual patch cords" one can route audio and control "signals" anywhere in an instruments using them to: set a parameter to a new value, dynamically control a parameter (like turning a knob), or as an audio input into some processing opcode.

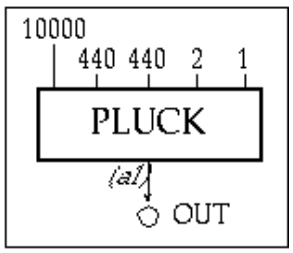
In figure 1.4, you can see that *instr 102 — instr 106* use the same simple instrument design as *instr 101* (one signal generator writing to the hard-disk). We have replaced the **oscil** opcode with more powerful synthesis opcodes such as: **foscil** — a simple 2-oscillator FM synthesizer, **buzz** — an additive set of harmonically-related cosines, **pluck** — a simple waveguide synthesizer based on the Karplus-Strong algorithm, **grain** — an asynchronous granular synthesizer and **loscil** — a sample-based wavetable synthesizer with looping.



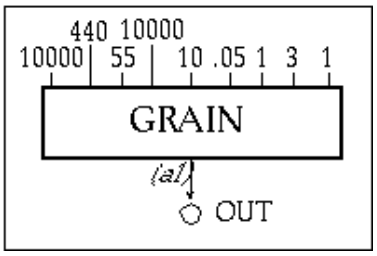
```
    instr 102          ; SIMPLE FM
a1  foscil10000, 440, 1, 2, 3, 1
    out    a1
    endin
```



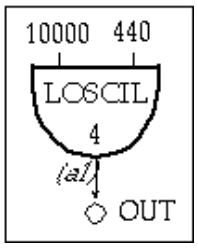

```
    instr 103          ; SIMPLE BUZZ
a1  buzz  10000, 440, 10, 1
    out    a1
    endin
```




```
    instr 104          ; SIMPLE WAVEGUIDE
a1  pluck 10000, 440, 440, 2, 1
    out    a1
    endin
```




```
    instr 105          ; SIMPLE GRANULAR
a1  grain 10000, 440, 55, 10000, 10, .05, 1, 3,
          1
    out    a1
    endin
```




```
    instr 106          ; SIMPLE SAMPLE PLAYBACK
a1  loscil10000, 440, 4
    out    a1
    endin
```

Figure 1.4 Block diagrams and orchestra code for *instr 102 — instr 106,* a collection of fixed frequency and amplitude instruments that use different synthesis methods to produce a single note

with the same amplitude (10000) and frequency (440).

Clearly the "single signal-generator" structure of these instruments is identical. But once you render them you will hear that their sounds are quite unique. Even though they each play with a frequency of 440 Hz and amplitude of 10000, the underlying synthesis algorithm embodied in each opcode is fundamentally different — requiring the specification of a unique set of parameters. In fact, these six signal generating opcodes (**oscil**, **foscil**, **buzz**, **pluck**, **grain** and **loscil**) represent the core synthesis technology behind many of today's most popular commercial synthesizers. One might say that in Csound, a single opcode is an entire synthesizer! Well... maybe not a very exciting or versatile synthesizer, but... in combination with other opcodes, Csound can, and will, take you far beyond any commercial implementation.

The Score File

Now let's look at the Csound score file that "performs" this orchestra of instruments. Like the orchestra file, the score file has two parts: *tables* and *notes*. In the first part, we use Csound's mathematical function-drawing subroutines (**GENS**) to directly "generate" function-tables (*f-tables*) and/or fill them by "reading" in soundfiles from the hard-disk. In the second part, we type in the *note-statements*. These note-events "perform" the instruments and pass them performance parameters such as frequency-settings, amplitude-levels, vibrato-rates and attack-times.

The GEN Routines

Csound's function generating subroutines are called **GENS**. Each of these (more than 20) subroutines is optimized to compute a specific class of functions or wavetables. For example, the **GEN5** and **GEN7** subroutines construct functions from segments of exponential curves or straight lines; the **GEN9** and **GEN10** subroutines generate composite waveforms made up of weighted sums of simple sinusoids; the **GEN20** subroutine generates standard "window" functions such as the Hamming window and the Kaiser window that are typically used for spectrum analysis and grain envelopes; the **GEN21** subroutine computes tables with different random distributions such as Gaussian, Cauchy and Poison; and the **GEN1** subroutine will transfer data from a pre-recorded soundfile into a function-table for processing by one Csound's opcodes such as the looping-oscillator **loscil**.

Which function tables are required, and how the instruments in your orchestra use them, is totally up to you — the sound designer. Sometimes it's a matter of common sense. Other times it's simply a matter of preference or habit. For instance, since *instr 106* used the sample-based looping oscillator, **loscil**, I needed to load a sample into the orchestra. I chose **GEN1** to do it. Whereas in *instr 102,* since I was using the **foscil** opcode, I could have chosen to frequency modulate any two waveforms, but decided on the traditional approach and modulated two sinewaves as defined by **GEN10**.

The Score Syntax

In the score file, the syntax of the Csound function statement (*f-statement*) is:

```
f numberload- table- GEN     parameter1parameter...;
        time  size   Routine                        comment
```

If we wanted to generate a 16 point sinewave, we might write the following f-statement:

```
f 101 0  16 10 1    ; a sinewave
```

As a result, the f-table (*f 101*) would generate the function shown in figure 1.5.

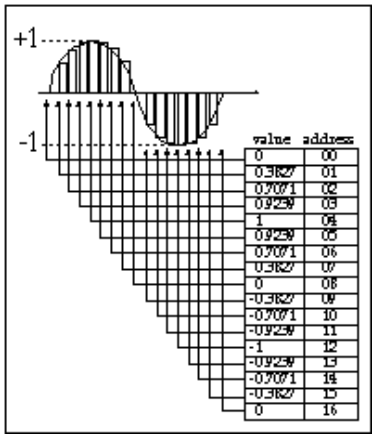


Figure 1.5 A 16 point sine function defined by GEN10 with the arguments: *f 101 0 16 10 1*

As you can see, a sinewave drawn with 16 points of resolution is not particularly smooth. Most functions must be a "power-of-2" in length. For synthesized wavetables, we typically specify function table sizes between 512 (5 K) and 8192 (8 K). In our first score, *etude1.sco,* we define the following functions using **GEN10**, **GEN20** and **GEN1**:

```
f 1 0  4096  10 1
f 2 0  4096  10 1 .5 .333 .25 .2 .166 .142 .125 .111 .1 .09
                .083 .076 .071 .066 .062
f 3 0  4097  20 2 1
f 4 0  0     1  "sing.aif" 0 4 0
```

Figure 1.6 Function tables defined in the score file *etude1.sco*.

All four functions are loaded at time *0*. Both *f 1* and *f 2* use **GEN10** to fill 4 K tables (*4096* values) with: one cycle of a sinewave (*f 1*) and with the first 16 harmonics of a sawtooth wave (*f 2*). **GEN20** is used to fill a 4 K table (*f 3*) with a *Hanning* window for use by the **grain** opcode. Finally, *f 4* uses **GEN1** to fill a table with a 44.1 K mono 16-bit AIFF format soundfile of a male vocalist singing the word "la" at the pitch A440 for 3 seconds. This "sample" is used by the **loscil** opcode. (Note that the length of *f 4* is specified as 0. This tells the **GEN1** subroutine to "read" the actual length of the file from the "header" of the soundfile "sing.aif." In this specific case, then, that length would be 132300 samples — 44100 samples-per-second * 3 seconds.)

The Note List

In the second part of the Csound score file we write the "notes." As in the orchestra file, each *note-statement* in the score file occupies a single line. Note-statements (or *i-statements*) call for an instrument to be "made active" at a specific time and for a specific duration. Further, each note-statement can be used to pass along a virtually unlimited number of unique parameter settings to the instrument and these parameters can be changed on a note-by-note basis.

Like the orchestra, which renders a sound line-by-line, the score file is "read" line-by-line, note-by-note. However, "notes" can have the same *start-times* and thus be "performed" simultaneously. In Csound one must always be aware of the fact that whenever two or more notes are performed simultaneously or whenever they overlap, their amplitudes are added. This can frequently result in "samples-out-of-range" or "clipping." (We will discuss this in detail shortly.)

You may have noticed that in the orchestra, commas separated an opcode's arguments. Here in the score, any number of spaces or tabs separates both the f-table arguments and i-statement parameter fields, (or p-fields). Commas are not used.

In order to keep things organized and clear, sound designers often use tab-stops to separate their p-fields. This practice keeps p-fields aligned in straight columns and facilitates both reading and debugging. This is not required — just highly recommended!

The First Three P-Fields

In all note-statements, the "meaning" of the first three p-fields (or columns) is reserved. They specify the *instrument number*, the *start-time* and the *duration*.

```
;  p1        p2      p3
```

```
i  instrumentstart- duration
   #          time
```

You — the sound designer, determine the function of all other p-fields. Typically, *p4* is reserved for amplitude and *p5* is reserved for frequency. This convention has been adopted in this chapter and in this text. In our first score, *etude1.sco,* a single note with duration of 3 seconds is played consecutively on *instr 101 — instr 106*. Because the start-times of each note are spaced 4 seconds apart, there will be a second of silence between each audio event.

```
; P1         P2         P3
; instrumentstart-    duration
#            time
i 101        0          3
i 102        4          3
i 103        8          3
i 104        12         3
i 105        16         3
i 106        20         3
```

Figure 1.7 Simple score used to play instruments 101 through 106 as shown in figure 1.2 and 1.4.

Exercises for Etude 1

- Render the Csound orchestra and score files: **etude1.orc** & **etude1.sco**.
- Play and listen to the different sound qualities of each instrument.
- Modify the score file and change the duration of each note.
- Make all the notes start at the same time.
- Comment out several of the notes so that they do not "play" at all.
- Cut and Paste multiple copies of the notes, change the start times (*p2*) and duration (*p3*) of the copies to make the same instruments start and end at different times.
- Create a canon at the unison with *instr 106*.
- Look up and read about the opcodes used in *instr 101 — 106* in the *Csound Reference Manual*.

```
aroscil xamp,xcps,  ifn[, iphs]
arfoscilxamp,kcps,  kcar, kmod,   kndx,     ifn[,  iphs]
arbuzz  xamp,xcps,  knh,  ifn[,   iphs]
arpluck kamp,kcps,  icps, ifn,    imeth[,   iparm1,iparm2]
argrain xamp,xpitch,xdens,kampoff,kpitchoff,kgdur, igfn,   iwfn, imgdur
arloscilxamp,kcps,  ifn[, ibas][, imod1,    ibeg1, iend1][,imod2,ibeg2,iend2]
```

- In the orchestra file, modify the frequency and amplitude arguments of each instrument.
- Change the frequency ratios of the carrier and modulator in the **foscil** instrument.
- Change the number of harmonics in the **buzz** instrument.
- Change the initial function for the **pluck** instrument.
- Change the density and duration of the **grain** instrument.
- Make three copies of *f 4* and renumber them *f 5, f 6 and f 7*. Load in your own samples ("yoursound1.aif," "yoursound2.aif," "yoursound3.aif"). Create multiple copies of *instr 106* and number them *instr 66*, *instr 67* and *instr 68*. Edit the instruments so that they each "read" a different soundfile at a different pitch. Play the different samples simultaneously.
- In the file *etude1.orc* duplicate and renumber each duplicated instrument. Set different parameter values for each version of

the duplicated instruments. Play all twelve instruments simultaneously. Adjust the amplitudes so that you have no *samples-out-of-range*.

Sound, Signals and Sampling

To better appreciate what's going on in Csound, it might be best to make sure we understand the acoustic properties of sound and how it is represented in a computer.

The experience of sound results from our eardrum's sympathetic response to the compressions and rarefactions of the air molecules projected outward in all directions from a vibrating source. This vibrating pattern of pressure variations is called a *waveform*. Looking at figure 1.8, we can see that the air molecules would be *compressed* when the waveform is above the x-axis (positive) and *rarefacted* when below (negative). Figure 1.8 shows a single cycle of a square wave in both the time and frequency domains.

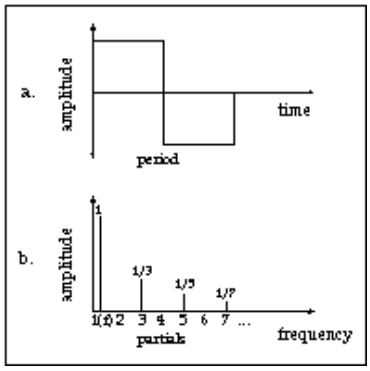


Figure 1.8 A time domain (a) and frequency domain (b) representation of a square wave.

The time domain representation (a) plots time on the x-axis and amplitude on the y-axis a. The frequency domain representation (b) plots frequency on the x-axis and amplitude on the y-axis.

We experience sound in the time-domain as pressure variations, but we perceive sound in the frequency domain as spectral variations. The ear acts as a *transducer*, converting the mechanical motion of the eardrum (through the ossicles: the hammer, anvil and stirrup) to the oval window membrane that causes a traveling wave to propagate in the fluid of the cochlea and stimulate the hair cells on the basilar membrane. These hair cells are like a high resolution frequency analyzer that transmits this complex set of frequency information to the brain through nerves attached to each of the hair cells. With this incredibly sensitive set of sensors, transducers and transmitters we actively and continuously analyze, codify, classify and perceive the complex frequency characteristics of soundwaves as we resonate with the world around us.

Typically we employ a different transducer (a microphone) to convert acoustic waveforms into signals that we can visualize and manipulate in the computer. The process is referred to as *sampling* and is illustrated in figure 1.9.

When sampling a waveform, we use a microphone first to convert an acoustic pressure wave into an *analogous* electrical pressure wave or an *analog* signal. Then we pass this signal through an *anti-aliasing* lowpass filter to remove the frequency components above 1/2 the *sampling-rate*. In fact, a digital system can not accurately represent a signal above 1/2 the sampling rate (this "frequency mirror" is known as the *Nyquist* frequency). So then, after we lowpass filter out the highs, that we can't accurately represent, we proceed to "measure" or "sample" the amplitude of the signal with an analog-to-digital converter (ADC).

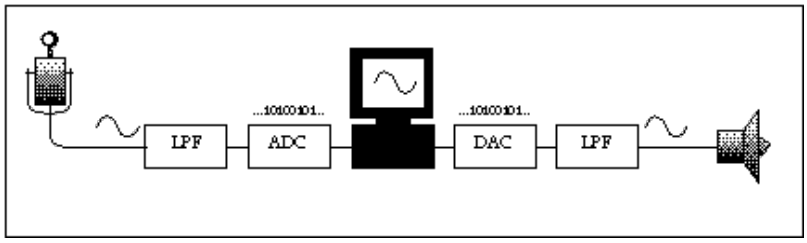


Figure 1.9 Digital recording ("sampling") and playback.

If you have a 16 bit linear system, you would sample the analog waveform with 16 bits of precision (in a range from -32768 to 32767 or 2^{16}) taking a new measurement at the sample-rate (44100 times per second as defined by our default header). In essence we have "quantized" this continuous analog signal into a series of little snapshots (or steps) — literally we are taking thousands of "little samples" from the signal. You can clearly see the "quantization" of the sinewave in figure 1.5 where each address corresponds with the amplitude of the signal at that point in time.

To hear a sound from our computer, we convert the digital signal (this sequence of "samples") back into an analog signal (a continuously varying voltage) using a digital-to-analog converter (DAC) followed by a *smoothing* lowpass filter. Clear? Well, enough of the basics for now. Let's get back to Csound.

Sound Design Etude 2: Parameter Fields

etude2.orc etude2.sco

In our second orchestra, we modify *instr 101 — 106* so that they can be updated and altered from the score file. Rather than setting each of the opcode's arguments to a fixed value in the orchestra, as we did in *etude1.orc*, we set them to "*p*" values that correspond to the p-fields (or column numbers) in the score. Thus each argument can be sent a completely different setting from each note-statement.

In *instr 107*, for example, p-fields are applied to each of the **oscil** arguments: amplitude (*p4*), frequency (*p5*) and wavetable (*p6*) as shown in figure 1.10.

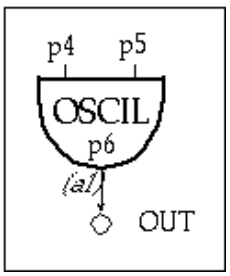


Figure 1.10 Block diagram of *instr 107*, a simple oscillator instrument with p-field substitutions.

```
     instr 107       ; P-Field Oscil
a1   oscil p4, p5, p6
     out   a1
     endin
```

Figure 1.11 Orchestra code for *instr 107*, a simple oscillator instrument with p-field arguments.

Thus from the score file in figure 1.12, we are able to re-use the same instrument to play a sequence of three descending octaves followed by an A major arpeggio.

```
; P1  P2    P3    P4     P5      P6
; inssta rtdur  amp    freq    waveshape

i 1070    1     10000  440     1
i 1071.5  1     20000  220     2
i 1073    3     10000  110     2
i 1073.5  2.5   10000  138.6   2
i 1074    2     5000   329.6   2
i 1074.5  1.5   6000   440     2
```

Figure 1.12 Note-list for *instr 107*, that uses p-fields to "perform" 6 notes (some overlapping) with different frequencies, amplitudes and waveshapes.

In our next p-field example, shown in figures 1.13, 1.14 and 1.15, our rather limited *instr 102* has been transformed into a more musically versatile *instr 108* — an instrument capable of a large array of tone colors.

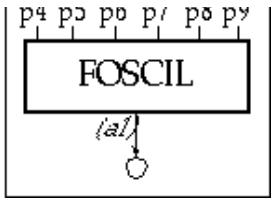


Figure 1.13 Block diagram of *instr 108,* a simple FM instrument with p-fields for each parameter.

```
    instr 108       ; P-Field FM
a1  foscilp4, p5, p6, p7, p8, p9
    out   a1
    endin
```

Figure 1.14 Orchestra code for *instr 108,* a simple FM instrument with p-field substitutions.

```
; p1 p2   p3   p4    p5    p6  p7    p8    p9
;    strt dur  amp   freq  c   m     indexwaveshape
ins

i    7    1    10000440   1   2     3     1
108
i    8.5  1    20000220   1   .5    8     1
108
i    10   3    10000110   1   1     13    1
108
i    10.5 2.5  10000130.81   2.0018     1
108
i    11   2    5000  329.61   3.0035     1
108
i    11.5 1.5  6000  440   1   5.0053     1
108
```

Figure 1.15 Note-list for *instr 108* in which nine p-fields are used to "play" an FM synthesizer with different start-times, durations, amplitudes, frequencies, frequency-ratios, and modulation indices.

In the score excerpt shown in figure 1.15, each of the **foscil** arguments has been assigned to a unique p-field and can thus be altered on a note-by-note basis. In this case *p4* = amplitude, *p5* = frequency, *p6* = carrier ratio, *p7* = modulator ratio, *p8* = modulation index and *p9* = wavetable. Thus, starting 7 seconds into *etude2.sco*, *instr 108* plays six consecutive notes. All six notes use *f 1* (a sine wave in *p9*). The first two notes, for example, are an octave apart (*p5* = 440 and 220) but have different *c:m ratios* (*p7* = 2 and 13) and different modulation indexes (*p8* = 3 and 8) resulting in two very different timbres. Obviously, p-fields in the orchestra allow us to get a wide variety of pitches and timbres from even the simplest of instruments.

Exercises for Etude 2

- Render the Csound orchestra and score: etude2.orc & etude2.sco.
- Play and listen to the different sound qualities of each note and instrument.
- Modify the score file and change the start-times, durations, amplitudes and frequencies of each note.
- Again look up and read about the opcodes used in *instr 107 — 112* in the *Csound Reference Manual* and focus your study and experimentation on one synthesis technique at a time.
- Explore the effect of different C:M rations in *instr 108*.
- Without changing the C:M ratio, explore the effect of a low and a high modulation index.
- Compare the difference in timbre when you modulate with a sine (*f 1*) and a sawtooth (*f 2*).

• Using *instr 109*, compose a 4-part chord progression in which the bass and tenor have more harmonics than the alto and soprano.

• Using *instr 109* and *112* simultaneously, play the same score you composed for *instr 9* by doubling the parts.

• Using *instr 110*, experiment with the various **pluck** methods. (See the *Csound Reference Manual* for additional arguments).

• Using *instr 110*, experiment with different initialization table functions — *f 1* and *f 2*. Also try initializing with noise and compare the timbre.

• Explore the various parameters of the **grain** opcode.

• Create a set of short etudes for each of the instruments alone.

• Create a set of short etudes for several of the instruments in combination. Remember to adjust your amplitude levels so that you do not have any *samples-out-of- range*!

• Lower the sample-rate and the control rate in the header. Recompile some of your modified instruments. Do you notice any difference in sound quality? Do you notice a change in brightness? Do you notice any noise artifacts? Do you notice any "aliasing?" (We will discuss the theory behind these phenomena a little later.)

Amplitudes and Clipping

As stated previously, if you have a 16 bit converter in your computer system (which is typical), then you can express 2^{16} possible raw amplitude values (i.e. 65536 in the range -32768 to +32767). This translates to an amplitude range of over 90 dB (typically you get about 6 dB of range per bit of resolution). But if you have been doing the exercises, then you have probably noticed that note amplitudes in Csound are additive. This means that if an instrument has an amplitude set to 20000 and you simultaneously play two notes on that instrument, you are asking your converter to produce a signal with an amplitude of ± 40000. The problem is that your 16 bit converter can only represent values up to about 32000 and therefore your Csound "job" will report that there are *samples-out-of-range* and the resulting soundfile will literally be "clipped" as show in figure 1.16.

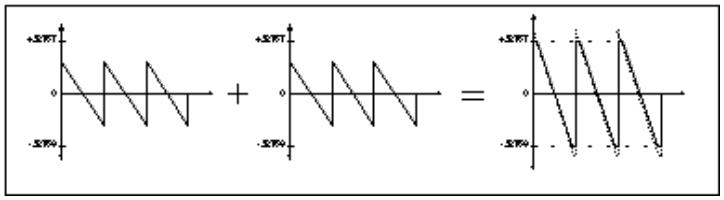


Figure 1.16 Clipping as a result of adding two high-amplitude waveforms together.

Dealing with amplitudes is one of the most problematic aspects of working with Csound. There is no easy answer. The problem lies in the fact that Csound amplitudes are a simple mathematical representation of "the signal." These measurements take no account of the acoustical or perceptual nature of "the sound."

Simply put, two times the linear displacement of amplitude will not necessarily be "perceived" as two times as loud. A good book on Acoustics will help you appreciate the complexity of this problem. In the Csound world, do remember that whenever two or more notes are sounding, their amplitudes are added. If the numbers add up to anything greater than 32000 your signal will be clipped. Now Csound does have some opcodes and tools that will help you "deal" with this "samples-out-of-range" problem but none of the current opcodes or value converters truly solve it. Most of the time you will just have to set the levels lower and render the file again (and again and again) until you get the amplitudes into a range that your system can handle.

Data Rates

As you have seen in the first two etude orchestras, we can set and update parameters (arguments) with "floating-point constants" either directly in the orchestra or remotely through parameter-fields. But the real power of Csound is derived from the fact that one can update parameters with *variables* at any of four update or data-rates: **setup**, **i**-rate, **k**-rate, or **a**-rate, where:

• **i**-rate variables are changed and updated at the note-rate.

• **k**-rate variables are changed and updated at the control-rate (*kr*).

• **a**-rate variables are changed and updated at the audio-rate (*sr*).

Both **i**-rate and **k**-rate variables are "scalars." Essentially, they take on one value at a given time. The **i**-rate variables are primarily used for setting parameter values and note durations. They are "evaluated" at "initialization-time" and remain constant throughout the duration of the note-event.

The **k**-rate variables are primarily used for storing and "updating" envelopes and sub-audio control signals. These variables are recomputed and at the control-rate (4410 times per second) as defined by *kr* in the orchestra header. The **a**-rate variables are "arrays" or "vectors" of information. These variables are used to store and update data such as the output signals of oscillators and filters that change at the audio sampling-rate (44100 times per second) as defined by *sr* in the header.

One can determine and identify the rate that a variable will be "updated" by the "first letter" of the variable name. For example, the only difference between the two oscillators below is that one is computed at the audio-rate and the other at the control-rate. Both use the same opcode, **oscil** and both have the same arguments. What is different, then, is the sample resolution (precision) of the "output" signal.

```
;       opcode  amp,   frq, func ; comment
output

ksig    oscil 10000,1000,1    ; 1000 Hz Sine - f
                              1
asig    oscil 10000,1000,1    ; 1000 Hz Sine - f
                              1
```

Figure 1.17 Two **oscil** opcodes with *asig* versus *ksig* outputs.

Given our default header settings of *sr* = 44100 and *kr* = 4410, *ksig* would be rendered at a sample-rate of 4 K and *asig* will be rendered at a sample-rate of 44.1K. In this case, the resulting output would sound quite similar because both would have enough "sample resolution" to accurately compute the 1000 HZ sinewave. However, if the arguments were different and the waveforms had additional harmonics, such as the sawtooth wave defined by *f 2* in figure 1.18, the **k**-rate setting of 4410 samples-per-second would not accurately represent the waveform and "aliasing" would result. (We will cover this in more detail later.)

```
;       opcode  amp,   frq, func ; comment
output

ksig    oscil 10000,1000,2    ; 1000 Hz Saw - f
                              2
asig    oscil 10000,1000,2    ; 1000 Hz Saw - f
                              2
```

Figure 1.18 An "under-sampled" sawtooth wave (given *kr* = 4410 and a frequency setting of 1000), resulting in an "aliased" *ksig* output.

You should note that it is left up to you, the sound designer, to decide the most appropriate, efficient and effective rate to render your opcodes. For example, you could render all of your low frequency oscillators (LFOs) and envelopes at the audio-rate, but it would take longer to compute the signals and the additional resolution would, in most cases, be imperceptible.

Variable Names

In our instrument designs so far we have been talking about and used *a1, asig,* k1 and *ksig* — in many cases interchangeably! What's with these different names for the same thing? Csound is difficult enough. Why not be consistent?

Well, when it comes to naming variables, Csound only requires that the variable names you use begin with the letter **i**, **k**, or **a**. This is so that the program can determine the rate to render that specific line of code. Anything goes after that.

For instance, you could name the output of the **loscil** opcode below **a**1, **a**sig, **a**sample, or **a**coolsound. Each variable name would be recognized by Csound and would run without error. In fact, given that the lines of code each have the same parameter settings, they would all sound exactly the same when rendered — no matter what name you gave them! Therefore, it is up to you, the sound designer, to decide on a variable naming scheme that is clear, consistent and informative... to you.

```
a1          loscil10000,     ; sample playback of
                 440, 4      f 4 at A440
            out    a1

asig        loscil10000,     ; sample playback of
                 440, 4      f 4 at A440
            out    asig

asample     loscil10000,     ; sample playback of
                 440, 4      f 4 at A440
            out    asample

acoolsoundloscil10000,     ; sample playback of
                 440, 4      f 4 at A440
            out    acoolsound
```

Aliasing and the Sampling Theorem

Let's review a bit more theory before getting into our more advanced instrument designs. As we stated earlier, the "under-sampled sawtooth" (*ksig*) in figure 1.18 is an example of "aliasing" and a proof of the "Sampling Theorem." Simply put, the Sampling Theorem states that, in the digital domain, to accurately reconstruct (plot, draw or reproduce) a waveshape at a given frequency, you need twice as many "samples" as the highest frequency you are trying to render. This "hard" upper limit at 1/2 the sampling rate is known as the *Nyquist frequency*. With an audio-rate of 44100 Hz, you can accurately render tones with frequencies (and partials) up to 22050 Hz — arguably far above the human range of hearing. And with a control-rate set to 4410 Hz, you can accurately render tones up to 2205 Hz. Certainly this would be an incredibly fast LFO and seems a bit high for slowly changing control signals, but you should recognize that certain segments of amplitude envelopes change extremely rapidly. "High-resolution" controllers can reduce the "zipper-noise" sometimes resulting from these rapid transitions.

Figure 1.19 illustrates graphically the phenomena known as "aliasing." Here, because a frequency is "under-sampled" an "alias" or alternate frequency results. In this specific case our original sinewave is at 5 Hz. We are sampling this wave at 4 Hz (remember that the minimum for accurate reproduction would be 10 Hz — 2 time the highest frequency component) and what results is a 1 Hz tone! As you can see from the figure, the values that were returned from the sampling process trace the outline of a 1 Hz sinewave not a 5 Hz one. The actual aliased frequency is the "difference" between the sampling frequency and the frequency of the sample (or its partials too).

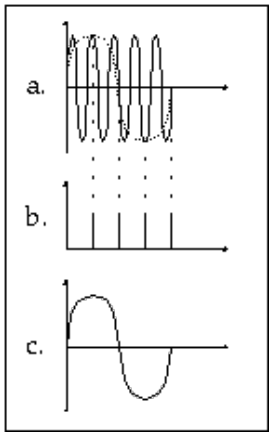


Figure 1.19 Aliasing. A 5 Hz sine (a) is under-sampled at 4 times a second (b) resulting in the incorrect reproduction of a of a 1 Hz sine (c).

To totally understand and experience this phenomena, it would be very informative to go back to the earlier instruments in this chapter and experiment with different rate variables. (I recommend that you duplicate and renumber all the instruments. Then change all the *asig* and *a1* variables to *ksig* and *k1* variables and render again. You'll be surprised, and even pleased, with some of the lo-fi results!) For now lets move on.

Sound Design Etude 3: Four Enveloping Techniques

etude3.orc etude3.sco

It is often stated that a computer is capable of producing "any" sound imaginable. And "mathematically," this is true. But if this is true, why is it that computerized and synthesized sounds are often so "sterile," "monotonous," and "dull?" To my ear, what makes a sound interesting and engaging is the subtle, dynamic and interdependent behavior of its three main parameters — pitch, timbre and loudness. And what makes Csound a truly powerful software synthesis language, is the fact that one can literally patch the output of any opcode into virtually any input argument of another opcode — thereby achieving an unsurpassed degree of dynamic parameter control. By subtly (or dramatically) modifying each of your opcode's input arguments, your "synthetic," "computerized" sounds will spring to life!

Up to this point we have essentially "gated" our Csound instruments — simply turning them on at full volume. I'm not sure that "any" acoustic instrument works like that. Clearly, applying some form of overall envelope control to these instruments would go a long way toward making them more "musical." And by adding other "dynamic parameter controls" we'll render sounds that are ever more enticing.

In *instr 113*, shown in figure 1.20 and 1.21, Csound's **linen** opcode is used to dynamically control the amplitude argument of the oscillator and thus functions as a typical attack-release (AR) envelope generator.

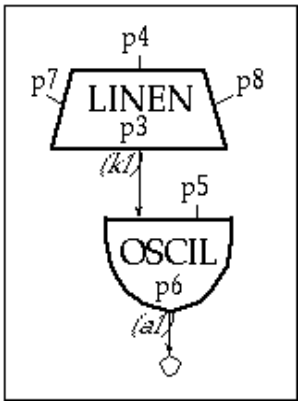


Figure 1.20 Block diagram of *instr 113,* an example one opcode's output controlling the input argument of another. In this case we are realizing dynamic amplitude control by modifying the amplitude argument of the **oscil** opcode with the output of another — the **linen**.

```
      instr   113          ; SIMPLE OSCIL WITH ENVELOPE
k1    linen   p4, p7, p3, p8  ; p3=dur, p4=amp, p7=attack,
                              p8=release
a1    oscil   k1, p5, p6      ; p5=freq, p6=waveshape
      out     a1
      endin
```

Figure 1.21 Orchestra code for *instr 113,* a simple oscillator instrument with amplitude envelope control.

In *instr 115*, shown in figures 1.22 and 1.23, a **linen** is again used to apply a dynamic amplitude envelope. But this time the "enveloping" is done by multiplying the output of the **linen** opcode (*k1*) with the output of the **buzz** opcode (*a1*). In fact, the multiplication is done in the input argument of the **out** opcode (*k1 * a1*). Here we not only see a different way of applying an envelope to a signal (multiplying it by a controller), but we also see that it is possible to perform mathematical operations on variables within an opcode's argument.

In figure 1.22, it can also be seen that an **expon** opcode is used in this instrument to move exponentially from the value in *p10* to the value in *p11* over the duration of the note (*p3*) and thereby sweeping the number of harmonic-cosines that **buzz** produces. The effect is very much like slowly closing a resonant lowpass filter and is another simple means of realizing dynamic timbre control.

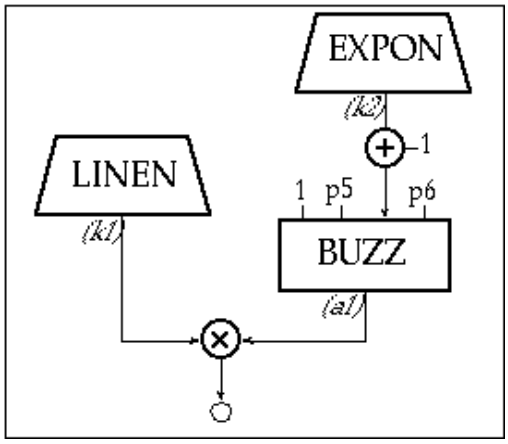


Figure 1.22 Block Diagram of *instr 115* showing amplitude control by multiplying two outputs and dynamic control of an argument.

```
   instr  115        ; Sweeping Buzz with
                     Envelope
k1 linen  p4, p7, p3, p8
k2 expon  p9, p3, p10
a1 buzz   1, p5, k2+1, p6
   out    k1*a1
   endin
```

Figure 1.23 Orchestra code for *instr 115,* an instrument with dynamic amplitude and harmonic control.

If you've been looking through the *Csound Reference Manual* you probably noticed that many opcodes, such as **oscil**, have both **k**-rate and **a**-rate versions. In *instr 117*, shown in figure 1.24, we use an audio-rate **linen** as an envelope generator. To do this, we "patch" the output of the **grain** opcode into the amplitude input of the **linen**, as can be seen in boldface in figure 1.25. Clearly this approach uses the linen to put an "envelope" on the signal coming from the granular synthesizer. In fact, we literally stuff the signal into an "envelope" before sending it out!

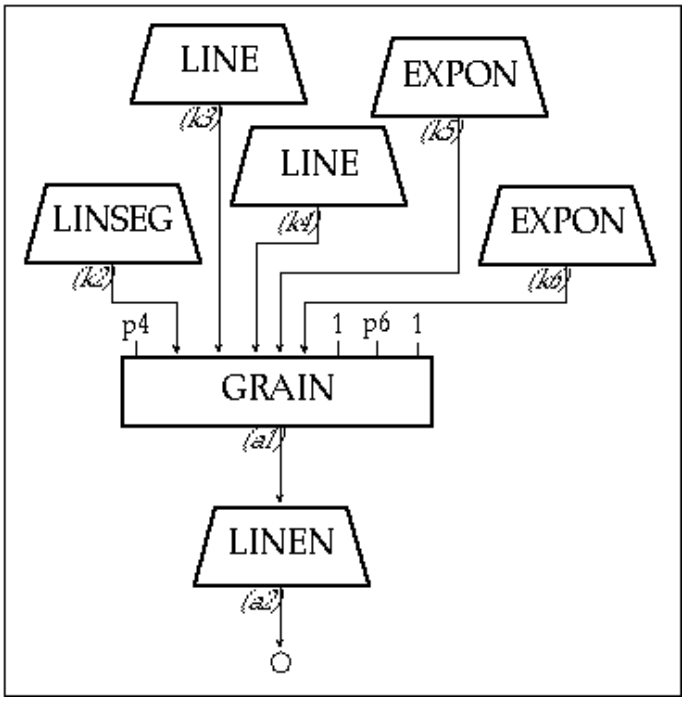


Figure 1.24 Block Diagram of *instr 117* showing amplitude control by passing the signal (*a1*) through an **a**-rate envelope (*a2*).

```
   instr  117        ; Grains through an Envelope
k2 linseg p5, p3/2, p9, p3/2, p5
```

```
k3  line   p10, p3, p11
k4  line   p12, p3, p13
k5  expon  p14, p3, p15
k6  expon  p16, p3, p17
a1  grain  p4, k2, k3, k4, k5, k6, 1, p6, 1
a2  linen  a1, p7, p3, p8
    out    a2
    endin
```

Figure 1.25 Orchestra code for *instr 117*, a granular synthesis instrument with dynamic control of many parameters. Note that the output of **grain** (*a1*) is "patched" into the amplitude argument of an a-rate **linen** to shape the sound with an overall amplitude envelope.

Envelopes

I have to admit that as a young student of electronic music, I was always "confused" by the use of the term "envelope" in audio and synthesis. I thought of "envelopes" as thin paper packages that you could "enclose" a letter to a friend or a check to the phone company and could never quite make the connection. But the algorithm used in *instr 117* makes the metaphor clear to me and hopefully to you. Here we see that the **linen** opcode completely "packages" or "folds" the signal into this "odd shaped" AR container and then sends it to the output. Figure 1.26 is another way to visualize the process. First we see the raw bipolar audio signal. Then we see the unipolar attack-decay-sustain-release (ADSR) amplitude envelope. Next we see the envelope applied to the audio signal. In the final stage we see the bipolar audio signal whose amplitude has been proportionally modified by the "contour" of the ADSR.

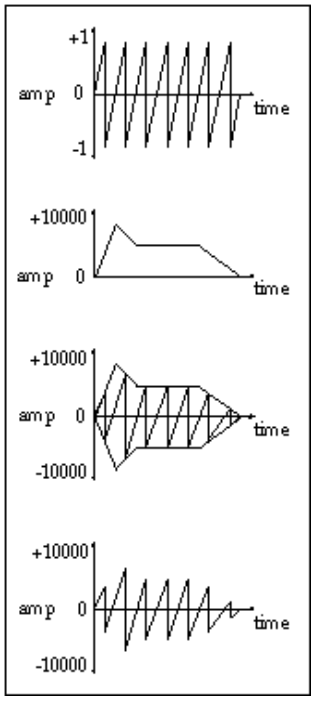


Figure 1.26 "Enveloping" a signal.

Another way of looking at figure 1.26 would be that our bipolar signal is scaled (multiplied) by a unipolar ADSR (Attack-Decay-Sustain-Release) envelope that symmetrically "contours" the unipolar signal. The result is that the unipolar signal is "enveloped" in the ADSR package. Let's apply this new level of understanding in another instrument design.

In *instr 118*, shown in figure 1.27 and 1.28, we illustrate yet another way of applying an envelope to a signal in Csound. In this case, we are using an oscillator whose frequency argument is set to *1/p3*. Let's plug in some numbers and figure out how this simple expression will help us compute the correct sub-audio frequency that will transform our periodic oscillator into an "aperiodic" envelope generator.

For example, if the duration of the note was 10 seconds and the frequency of our oscillator was set to 1/10 Hz it would take 10/10 Hz to completely "read" 1 cycle of the function table found in *p7*. Thus, setting the frequency of an oscillator to 1 divided by the

note-duration, or 1/*p3*, guarantees that this periodic signal generator will compute only 1 period, or read only 1 complete cycle of its f-table during the course of each note event.

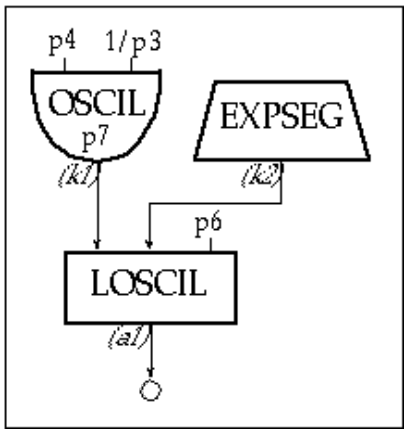


Figure 1.27 Block diagram of *instr 118,* an instrument with an oscillator for an envelope generator.

```
    instr 118               ; Loscil with Oscil
                            Envelope
k1  oscil p4, 1/p3, p7
k2  expseg p5, p3/3, p8, p3/3, p9, p3/3, p5
a1  loscil k1, k2, p6
    out    a1
    endin
```

Figure 1.28 Orchestra code for *instr 118*, an sample-playback instrument with and oscillator-based envelope and dynamic pitch modulation.

In *instr 118* the envelope functions called by *p7* (*f 6, f 7 and f 8*) use **GEN7** and **GEN5** to draw a variety of unipolar linear and exponential contours. It is very important to note that it is *illegal* to use a value of 0 in any exponential function such as those computed by the **GEN5** subroutine or by the **expseg** opcode. You will notice therefore, that *f 8*, that uses **GEN5**, begins and ends with a value of .001 rather than 0.

```
f 6 0  1024 7  0 10 1 1000 1 14 0       ; linear AR
                                        envelope
f 7 0  1024 7  0 128 1 128 .6 512 .6    ; linear ADSR
               256 0                    envelope
f 8 0  1024 5  .001 256 1 192 .5 256 .5 ; exponential ADSR
               64 .001
```

Figure 1.29 Linear and exponential envelope functions using **GEN5** and **GEN7**.

The enveloping technique employed in *instr 118* (using an oscillator as an envelope generator) has several advantages. First, you can create an entire library of preset envelope shapes and change them on a note-by-note basis. Second, since the envelope generator is in fact an oscillator, you can have the envelope "loop" or retrigger during the course of the note event to create interesting "LFO-based amplitude-gating" effects. In *instr 119*, shown in figure 1.30, *p8* determines the number of repetitions that will occur during the course of the note. If *p8* is set to 10 and *p3* is 5 seconds, the instrument will "retrigger" the envelope 2 times per second. Whereas, if the duration of the note was 1 second (*p3* = 1), then the envelope would be re-triggered 10 times per second.

```
     instr  119             ; Retriggering Foscil with
                            Oscil Envelope
k1   oscil  p4, 1/p3 * p8,; P8=retrigger rate per note
            p7              duration
k2   line   p11, p3, p12
a1   foscil k1, p5, p9, p10, k2, p6
     out    a1
     endin
```

Figure 1.30 Orchestra code for *instr 119,* an FM instrument with an oscillator envelope in which *p8* determines the "retriggering" frequency.

Unipolar and Bipolar Functions

Typically we think of an oscillator as something that makes a "sound" by "playing" different waveshapes or samples. However, we have seen that Csound's table-lookup oscillator is capable of reading any unipolar or bipolar function at any rate. Clearly this signal generator can be utilized equally as a control source or an audio source. Unlike commercial synthesizers, in Csound the "function" of an opcode is defined by the use and by the user. So far, we have been using several of Csound's **GEN** routines to compute unipolar and bipolar functions and it is important that we make sure we understand the difference.

Most audio waveforms, such as those created with **GEN10** are bipolar — having a symmetrical excursion above and below zero. On the other hand, most envelope functions, such as those we have created using **GEN5** and **GEN7**, are unipolar — having an excursion in one direction only, typically positive. In Csound, by default, all bipolar functions are normalized in a range of -1 to +1 and all unipolar functions are normalized in a range from 0 to +1 as shown in figure 1.31.

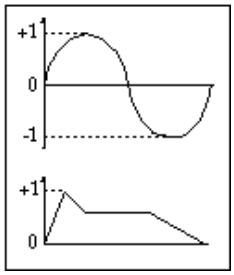


Figure 1.31 A bipolar (-1 to +1) and a unipolar (0 to +1) function.

If you wish to bypass Csound's default normalization process, you must use a minus sign (**-**) before the **GEN** number as shown in *f 3* and *f 4* in figure 1.32.

```
f 1  0  512  10   1                  ; Normalized bipolar Sine
f 2  0  512  7    0 6 1 500 1 6 0    ; Normalized unipolar
                                     envelope
f 3  0  512  -10  .3 .013 .147       ; non-normalized bipolar
                  .026               Sum-of-Sines
f 4  0  512  -7   440 256 220 256    ; non-Normalized unipolar
                  440                envelope
```

Figure 1.32 Two Normalized functions (*f 1* and *f 2*) and two non-normalizing functions (*f 3* and *f 4*).

Exercises for Etude 3

- Render the third Csound orchestra and score: etude3.orc & etude3.sco.
- Play and listen to the different sound qualities and envelope shapes of each note and each instrument.
- Modify the orchestra file and change the variable names to more meaningful ones. For instance, rename all **a**1 variables **a**sig1 and **k**1 variables **k**env1.
- In the *Csound Reference Manual*, look up the new opcodes featured in *instr 113 — 119*:

```
kr   linen    kamp,  irise,  idur,  idec
ar   linen    xamp,  irise,  idur,  idec
kr   line     ia,    idur1,  ib
kr   expon    ia,    idur1,  ib
kr   linseg   ia,    idur1,  ib[,   idur2,   ic[...]]
kr   expseg   ia,    idur1,  ib[,   idur2,   ic[...]]
```

- Modify the attack time (*p7*) and release times (*p8*) of the **linen** opcodes in *instr 113 —117*.
- Add a pitch envelope to *instr 113, 114* and *115* by adding a **linseg** to each instrument and adding it's output to *p5*.
- Experiment with the dynamic controls of the **grain** parameters found in *instr 117*.

• Substitute **oscil**-based envelopes for the **linen**-based envelopes in *instr 113 — 117*.

• Use **GEN5** and **GEN7** to design several additional "envelope" functions. Try to imitate the attack characteristics of a piano — *f 9,* a mandolin — *f 10,* a tuba — *f 11,* a violin — *f 12 and* a male voice singing "la" — *f 13*. Apply these envelopes to your newly designed versions of *instr 113 — 117.*

• Following the examples of the figures you have studied so far, draw "block diagrams" for *instr 112, 113, 114* and *119*.

Sound Design Etude 4: Mix, Chorus, Tremolo and Vibrato

etude4.orc **etude4.sco**

Next we improve the quality of our instruments by first mixing and detuning our oscillators to create a fat "chorused" effect. Then we crossfade opcodes to create a hybrid synthesis algorithm unlike anything offered commercially. Finally we animate our instruments by introducing sub-audio and audio rate amplitude and frequency modulation (AM and FM). We also employ several of Csound's display opcodes to visualize these more complex temporal and spectral envelopes. And we'll learn a little more about the language as we go.

In *instr 120*, shown in figures 1.33 and 1.34, we mix together three detuned oscillators that all use the same **envlpx** opcode for an amplitude envelope. Utilizing the **display** opcode, this envelope is plotted on the screen with a resolution set to trace the envelope shape over the entire duration of the note (*p3*) and thus display the complete contour.

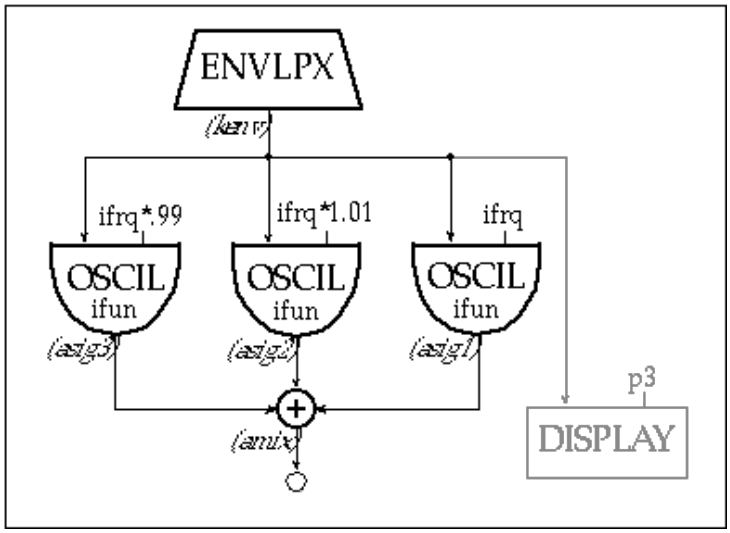


Figure 1.33 Block diagram of *instr 120* illustrating three chorusing oscillators with a common envelope and display.

```
        instr   120                 ; SIMPLE CHORUSING
idur    =       p3                  ; INITIALIZATION BLOCK
iamp    =       ampdb(p4)
ifrq    =       cpspch(p5)
ifun    =       p6
iatk    =       p7
irel    =       p8
iatkfun=        p9
kenv    envlpx  iamp, iatk, idur, irel, iatkfun, .7, .01
asig3   oscil   kenv, ifrq*.99,   ; SYNTHESIS BLOCK
                ifun
asig2   oscil   kenv, ifrq*1.01, ifun
asig1   oscil   kenv, ifrq, ifun
amix    =       asig1+asig2+asig3; MIX
        out     amix
```

```
        displaykenv, idur
        endin
```

Figure 1.34 Orchestra code for *instr 120,* a chorusing instrument in which p-fields are given i-time variable names. Also an **envlpx**, that is displayed, is used as a common envelope.

Although *instr 120* is still rather simple in design, it does serve as a model of the way that more complex instruments are typically "laid-out" and organized in Csound. In figure 1.34 you can see that variables are initialized at the top of the instrument and given names that help us to identify their function (resulting in a "self-commenting" coding style). Clearly you can read that the attack time is assigned to *iatk* from the score value given in *p7* (*iatk = p7*) and that the release time is assigned to *irel* from the score value given in *p9* (*irel = p9*). And most importantly, by looking at where they are "patched" in the **envlpx** opcode you can see and remember what arguments correspond with those particular parameters thereby making the opcode itself easier to "read."

You should note that in Csound the equals sign (=) is the "assignment operator." It is in fact an opcode! Assigning "plain-English" mnemonics and abbreviated names to variables at **i**-time makes an instrument much easier to "read" and is highly recommended.

Spectral Fusion

Next we will look at *instr 122*, as shown in figures 1.35 and 1.36. This instrument uses independent **expon** opcodes to crossfade between a **foscil** and a **buzz** opcode that are both "fused" (morphed/mixed/transfigured) with a **pluck** attack creating a beautiful hybrid timbre. This instrument employs Csound's **dispfft** opcode to compute and display a 512 point Fast Fourier Transform (FFT) of the composite signal updated every 250 milliseconds. Although the **display** and **dispfft** opcodes are a wonderful way to literally "look into" the behavior of your instrument. It is important to note that when you are using your instruments to make music, you should always remember to comment out these **display** and **print** opcodes because they significantly impact the performance of your system. These are informative and educational but really function as debugging tools. You should think of them as such.

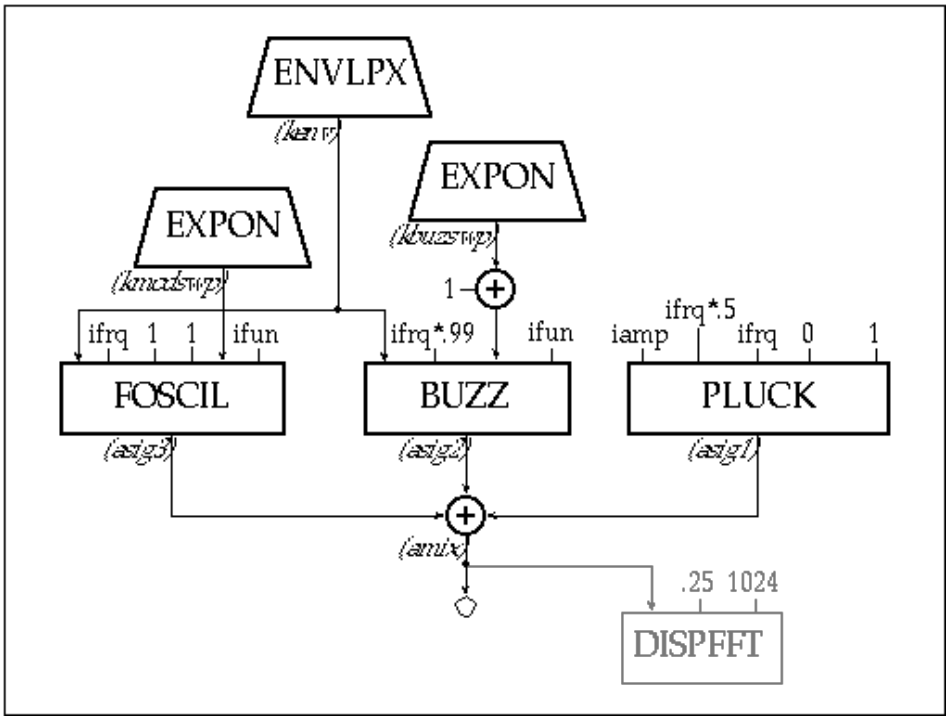


Figure 1.35 Block diagram of *instr 122* illustrating the FFT display of 3 mixed (fused) and crossfaded (morphed) opcodes.

```
        instr   122     ; Simple Spectral Fusion
idur    =       p3
iamp    =       ampdb(p4)
ifrq    =       cpspch(p5)
ifun    =       p6
iatk    =       p7
irel    =       p8
iatkfun=        p9
```

```
index1 =       p10
index2 =       p11
kenv   envlpx  iamp, iatk, idur, irel, iatkfun,
               .7, .01
kmodswpexpon   index1, idur, index2
kbuzswpexpon   20, idur, 1
asig3  foscil  kenv, ifrq, 1, 1, kmodswp, ifun
asig2  buzz    kenv, ifrq*.99, kbuzswp+1, ifun
asig1  pluck   iamp, ifrq*.5, ifrq, 0, 1
amix   =       asig1+asig2+asig3
       out     amix
       dispfftamix, .25, 1024
       endin
```

Figure 1.36 Orchestra code of *instr 122,* an instrument that demonstrates the fusion of three synthesis techniques — **pluck**, **foscil**, and **buzz**.

Rather than simply mixing or crossfading opcodes as we have done in *instr 120* and *instr 122*, another popular approach is to *modulate* one audio opcode with the frequency and amplitude of another. In *instr 124*, as shown in figures 1.37 and 1.38 for example, an **a**-rate **oscil** (*asig*) is amplitude modulated with the output of a dynamically swept **a**-rate **oscil** (*alfo*) whose frequency is dynamically altered with a **line** opcode and whose amplitude is controlled by an **expon**.

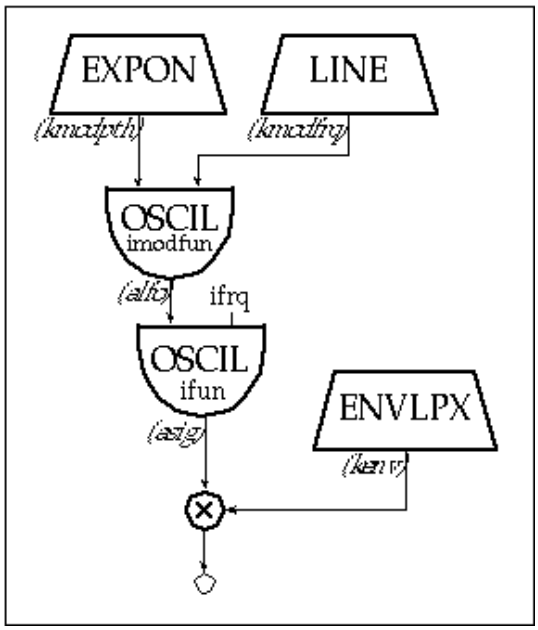


Figure 1.37 Block diagram of *instr 124* a dynamic amplitude modulation instrument.

```
       instr  124   ; Sweeping Amplitude
                    Modulation
idur   =      p3
iamp   =      ampdb(p4)
ifrq   =      cpspch(p5)
ifun   =      p6
iatk   =      p7
irel   =      p8
iatkfun=      p9
imodp1 =      p10
imodp2 =      p11
imodfr1=      p12
imodfr2=      p13
imodfun=      p14
kenv   envlpx iamp, iatk, idur, irel, iatkfun,
```

```
                .7, .01
kmodpthexpon    imodp1, idur, imodp2
kmodfrqline     cpspch(imodfr1), idur,
                cpspch(imodfr2)
alfo    oscil   kmodpth, kmodfrq, imodfun
asig    oscil   alfo, ifrq, ifun
        out     asig*kenv
        endin
```

Figure 1.38 Orchestra code for *instr 124,* an amplitude modulation instrument with independent amplitude envelope and variable LFO.

This simple oscillator combination can produce a wide range of rich harmonic and inharmonic dynamically evolving timbres.

Next in *instr 126*, shown in figures 1.39 and 1.40, we present a simple vibrato instrument that uses a **linseg** opcode to delay the onset of the modulation resulting in a more natural vibrato effect.

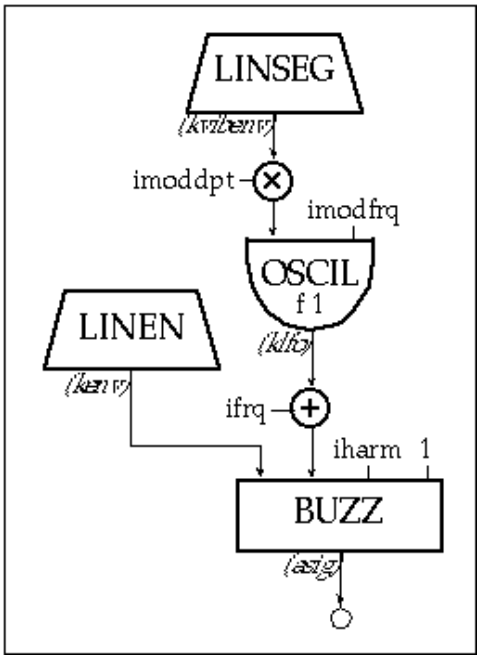


Figure 1.39 Block diagram of *instr 126,* an additive instrument with delayed vibrato.

```
        instr 126    ; Simple Delayed Vibrato
idur    =       p3
iamp    =       ampdb(p4)
ifrq    =       cpspch(p5)
iatk    =       p6
irel    =       p7
ivibdel=        p8
imoddpt=        p9
imodfrq=        p10
iharm   =       p11
kenv    linen   iamp, iatk, idur, irel
kvibenvlinseg0, ivibdel, 1, idur-ivibdel, .3
klfo    oscil   kvibenv*imoddpt, imodfrq, 1
asig    buzz    kenv, ifrq+klfo, iharm, 1
        out     asig
        endin
```

Figure 1.40 Orchestra code for *instr 126,* a **buzz** instrument with delayed vibrato.

Even these relatively simple designs can lend themselves to an incredibly diverse and rich palate of colors. Take time to explore and

modify them.

Value Converters

In the initialization "block" of *instr 120*, shown in figure 1.34 (and in all the instruments in this etude for that matter), you might have noticed that two of Csound's *value converters* **ampdb** and **cpspch** were utilized (*iamp = ampdb(p4)* and *ifrq = cpspch(p5)*). These allow us to express frequency and amplitude data in a more familiar and intuitive format than having to use straight Hz and linear amplitudes we have used thus far.

The **cpspch** value converter will read a number in *octave-point- pitch-class* notation and convert it to Hz (e.g., 8.09 = A4 = 440 Hz). Octave-point-pitch-class is a notation shorthand system in which octaves are represented as whole numbers (8.00 = Middle C or C4, 9.00 = C5, 10.00 = C6, etc.) and the 12 equal-tempered pitch classes are numbered as the two decimal digits that follow the octave (8.01 = C#4, 8.02 = D4, 8.03 = D#4, etc.). The scale shown in figure 1.41 should make the system extremely clear to you.

NOTE #	Hertz (Hz)	CPSPCH	MIDI NOTE NUMBER
C4	261.626	8.00	60
C#4	277.183	8.01	61
D4	293.665	8.02	62
D#4	311.127	8.03	63
E4	329.628	8.04	64
F4	349.228	8.05	65
F#4	369.994	8.06	66
G4	391.955	8.07	67
G#4	415.305	8.08	68
A4	440.000	8.09	69
A#4	466.164	8.10	70
B4	493.883	8.11	71
C5	523.251	9.00	72

Figure 1.41 A chromatic scale beginning at middle C specified by using Csound's **cpspch** value converter.

By adding more decimal places, it is also possible to specify microtones as shown in figure 1.42.

As you can see, **cpspch** converts from the **pch** (octave point pitch-class) representation to a **cps** (cycles-per-second) representation. If you are writing tonal or microtonal music with Csound, you might find this value converter particularly useful.

NOTE #	CPSPCH
C4	8.00
C4+	8.005
C#4	8.01
C#4+	8.015
D4	8.02
D4+	8.025
D#4	8.03
D#4+	8.035
E4	8.04
E4+	8.045
F4	8.05
F4+	8.055
F#4	8.06
F#4+	8.065
G4	8.07
G4+	8.075
G#4	8.08
G#4+	8.085
A4	8.09

```
A4+     8.095
A#4     8.10
A#4+    8.105
B4      8.11
B4+     8.115
C5      9.00
```

Figure 1.42 An octave of equal-tempered quarter-tones specified using the **cpspch** value converter.

Similarly, the **ampdb** value converter will read a decibel value and convert it to a raw amplitude value as shown in figure 1.43.

```
ampdb(42)=   125
ampdb(48)=   250
ampdb(54)=   500
ampdb(60)=   1000
ampdb(66)=   2000
ampdb(72)=   4000
ampdb(78)=   8000
ampdb(84)=   16000
ampdb(90)=   32000
ampdb(96)=   64000  ; WARNING: SAMPLES OUT OF
                    RANGE!!!
```

Figure 1.43 Amplitude conversion using the **ampdb** value converter.

You should note that although the logarithmic dB or decibel scale is linear in perception, Csound doesn't really "use" dB. The **ampdb** converter is a direct conversion with no scaling. Regrettably, you will still have to spend a great deal of time adjusting, normalizing, and scaling your amplitude levels even if you are using Csound's **ampdb** converter because the conversion is done prior to the rendering.

Exercises for Etude 4

- Render the fourth Csound orchestra and score: etude4.orc & etude4.sco.
- Play and listen to the dynamic timbres and modulation effects of each note and each instrument.
- Modify *instr 120* so that you are chorusing three **foscil** opcodes instead of three **oscil** opcodes.
- As shown in *instr 126*, add a delayed vibrato to your three **foscil** version of *instr 120*.
- Block diagram *instr 121* (which was not discussed in this section) and add delayed vibrato plus some of your own samples to this wavetable synthesizer.
- Modify *instr 122* so that you are creating totally different hybrid synthesizers. Perhaps you could add a **grain** of **loscil**?
- Block diagram *instr 123* (which was not discussed in this section) and change the rhythms and pitches. Try audio-rate modulation. Finally, make and use your own set of amplitude modulations functions.
- Modify *instr 124* so that you are not sweeping so radically. Add chorusing and delayed vibrato.
- Block diagram *instr 125* (which was not discussed in this section). Change the modulation frequency and depth using the existing functions. Modulate some of your own samples.
- Modify *instr 126* so that these "synthetic voices" sing microtonal melodies and harmonies.
- Block diagram *instr 127* (which was not discussed in this section). Have fun modifying it with any of the techniques and bits of code you have developed and mastered so far.
- In the *Csound Reference Manual*, look up the new opcodes featured in *instr 120 — 127*:

```
kr      envlpx      kamp, irise, idur, idec, ifn, iatss, iatdec[,
                    ixmod]
        print       iarg[, iarg, ...]
        display     xsig, iprd[, inprds][, iwtflg]
        dispfft     xsig, iprd, iwsiz[, iwtyp][, idbouti][, iwtflg]
```

- Create a new set of attack functions for **envlpx** and use them in all the instruments.

- Add **print**, **display** and **dispfft** opcodes to *instr 123— 127*. (But do remember to comment them out when you are making production runs with your instruments.)

A Little Filter Theory

The next sound design etude is an introductory exploration of a number of Csound's filter opcodes. But before we get too far along it might help if we review some of our filter basics. Four of the most common filter types are: *lowpass, highpass, bandpass* and *bandreject* as illustrated in figure 1.44. In this figure, a signal consisting of 12 harmonic partials of equal strength (a) is first filtered by a one-pole lowpass (b), a one-pole highpass (c), a two-pole bandpass (d), and a two-pole bandreject filter (e). The dotted spikes are in the filter's *stop band* and the solid spikes are in the *pass band.* The cutoff frequency is the -3 dB point in each of the spectral envelope curves outlined by the solid line.

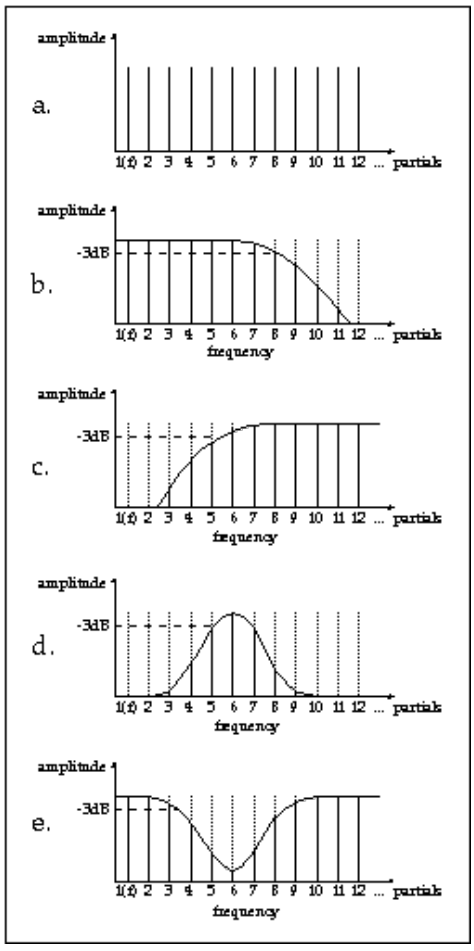


Figure 1.44 A source signal (a) modified by the four basic filters: **tone** — a one-pole lowpass (b), **atone** — a one-pole highpass (c), **reson** — a two-pole bandpass (d) and **areson** — a two-pole bandreject (e).

In Csound these filters would correspond with the **tone** (b), **atone** (c), **reson** (d) and **areson** (e) opcodes. Notice that the dashed line at -3 dB indicates the cutoff frequency of the filter. Why -3 dB? Well, since the slope of a filter is in fact continuous, the cutoff frequency (f_c) of a filter is "somewhere on the curve." and has been defined as the point in this frequency continuum at which the pass band is attenuated by -3 dB.

Sound Design Etude 5: Noise, Filters, Delays and Flangers

etude5.orc **etude5.sco**

The next set of instruments employ a number of Csound signal modifiers in various parallel and serial configurations to shape and transform noise and wavetables. In *instr 128*, shown in figures 1.45 and 1.46, we dynamically filters "white noise" produced by Csound's **rand** opcode. Separate **expon** and **line** opcodes are used to independently modify the cutoff frequency and bandwidth of Csound's two-pole **reson** (bandpass) filter. Also, an **expseg** amplitude envelope is used and displayed.

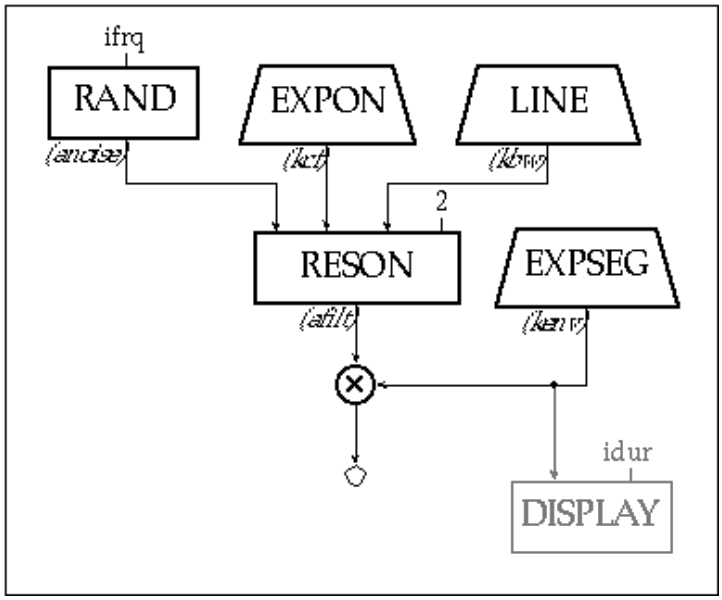


Figure 1.45 Block Diagram of *instr 128,* a bandpass-filtered noise instrument.

```
      instr  128     ; Bandpass-Filtered Noise
idur  =      p3
iamp  =      p4
ifrq  =      p5
iatk  =      p6
irel  =      p7
icf1  =      p8
icf2  =      p9
ibw1  =      p10
ibw2  =      p11
kenv  expseg .01,iatk,iamp,idur/6,iamp*.4,idur(iatk+irel+idur/6),iamp*.6,irel,.01
anoiserand   ifrq
kcf   expon  icf1, idur, icf2
kbw   line   ibw1, idur, ibw2
afilt reson  anoise, kcf, kbw, 2
      out    afilt*kenv
      displaykenv, idur
      endin
```

Figure 1.46 Orchestra code for *instr 128,* a bandpass filtered noise instrument with variable cutoff frequency and bandwidth.

A Cascade Filter Network

In *instr 129* through *132*, shown in figure 1.47, a white noise source (**rand**) is passed through a series of one-pole lowpass filters (**tone**). The significant contribution made by each additional pole should be quite apparent from this set of examples. In fact, each pole increases the "slope" or "steepness" of a filter by 6 dB per octave of "roll-off" at the cutoff frequency. A cascade filter design such as this makes the slope proportionally steeper with each added **tone** thus resulting in a more "effective" filter. Thus in our

"cascade design," *instr 129* would have a slope corresponding to an attenuation of 6 dB per octave; *instr 130* would have a slope of 12 dB per octave; *instr 131* would have a slope of 18 dB per octave; and *instr 132* would have a slope of 24 dB per octave. The **dispfft** opcode should clearly show the progressive effect on the spectrum of the noise in each instrument.

```
      instr  129   ; One-Pole Lowpass
anoiserand   ifrq
afilt tone   anoise, kcut
      dispfftafilt, idur, 4096

      instr  130   ; Two-Pole Lowpass
anoiserand   ifrq
afilt2tone   anoise, kcut
afilt1tone   afilt2, kcut
      dispfftafilt1, idur, 4096

      instr  131   ; Three-Pole Lowpass
anoiserand   ifrq
afilt3tone   anoise, kcut
afilt2tone   afilt3, kcut
afilt1tone   afilt2, kcut
      dispfftafilt1, idur, 4096

      instr  132   ; Four-Pole Lowpass
anoiserand   ifrq
afilt4tone   anoise, kcut
afilt3tone   afilt4, kcut
afilt2tone   afilt3, kcut
afilt1tone   afilt2, kcut
      dispfftafilt1, idur, 4096
```

Figure 1.47 Orchestra code excerpts from *instr 129 — 132*, taht pass white noise through a cascade of one-pole lowpass filters.

Displays

For several examples now, we have been using Csound's **display** and **dispfft** opcodes to look at signals. But what exactly is being displayed? And how are these opcodes different?

As you know, signals can be represented in either the time or the frequency domain. In fact, these are complimentary representations illustrating of how the signal varies in either amplitude or frequency over time. Csound's **display** opcode plots signals in the time domain as an amplitude versus time graph whereas the **dispfft** opcode plots signals in the frequency domain using the Fast Fourier Transform method. Both allow us to specify how often to update the display and thereby provide the means of watching a time or frequency domain signal evolve over the course of a note. Thus we used **display** in *instr 128* to look at the shape of the **expseg** amplitude envelope and see the way that the amplitude varied over the entire duration of the note.

In *instr 129 — 132* we used the **dispfft** opcode to look at the way that the frequencies were "attenuated" by our filter network. In our design, by specifying that the FFT be 4096 we divided the frequency space into 2048 linearly spaced frequency bins of about 21.5 Hz a piece (44100/2048 = 21.533), but we could have divided the spectrum anywhere from 8 bands (each 5512 Hz wide) to 2048 bands (each 21.5 Hz wide). We will continue using these opcodes to "look" into the time domain and frequency domain characteristics of the sounds that our instruments produce. In particular, the **dispfft** opcode will help and better understand the effect that Csound's different filters are having on the signals we put into them.

The **tone** and **reson** filters we have used thus far were some of Csound's first. They are noted for their efficiency (they run fast) and equally noted for their instability (they "blow-up"). In fact it has always been good advice to patch the output of these filters into Csound's **balance** opcode in order to keep the samples-out-of-range under control.

Over the years, however, many new filters have been added to the Csound language. In the early days of analog synthesis, it was the filters that defined the sound of these rare and coveted "classic" instruments. Today in Csound the *Butterworth* family of filters (**butterlp**, **butterhp**, **butterbp** and **butterbr**) are great sounding and becoming more common in virtually all instrument designs. This is due in part to the fact that the Butterworth filters have: more poles (they're steeper and more effective at "filtering," a flatter

frequency response in the pass-band (they are smoother and cleaner sounding) and they are significantly more stable (meaning that you do not have to worry so much about samples-out-of-range). In *instr 133*, shown in figures 1.48 and 1.49 we use a parallel configuration comprised of a 4-pole **butterbp** and a 4-pole **butterlp** filter pair as a way to model the "classic" resonant-lowpass filter commonly found in first generation analog synthesizers.

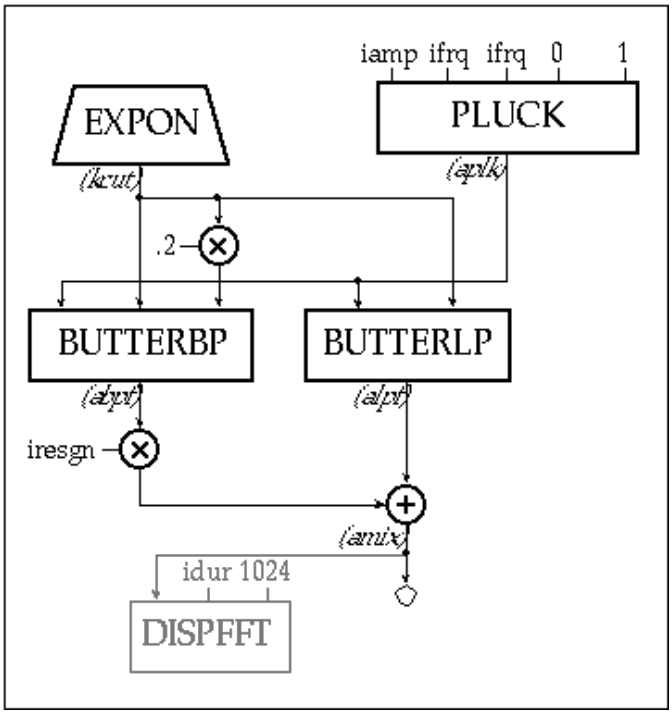


Figure 1.48 Block diagram of *instr 133,* a parallel **butterbp** and **butterlp** configuration resulting in a "classic" resonant-lowpass design.

```
      instr   133    ; Lowpass WITH RESONANCE
idur  =       p3
iamp  =       ampdb(p4)
ifrq  =       p5
icut1 =       p6
icut2 =       p7
iresgn=       p8
kcut  expon   icut1, idur, icut2
aplk  pluck   iamp, ifrq, ifrq, 0, 1
abpf  butterbpaplk, kcut, kcut*.2
alpf  butterlpaplk, kcut
amix  =       alpf+(abpf*iresgn)
      out     amix
      dispfft amix, idur, 1024
      endin
```

Figure 1.49 Orchestra code for *instr 133,* a "classic" resonant-lowpass filter design.

As you can see and hear from the previous set of examples, dynamic parametric control of Csound's filter opcodes, combined in various parallel and serial configurations, opens the door to a wide world of subtractive sound design possibilities.

An Echo-Resonator

Let's shift our focus now to another set of Csound's signal modifiers — **comb** and **vdelay**.

A **comb** filter is essentially a delay line with feedback as illustrated in figure 1.50. As you can see, the signal enters the delay line and is output delayed by the length of the line (25 milliseconds in this case). When it reaches the output, it is fed back to the input after being multiplied by a gain factor.

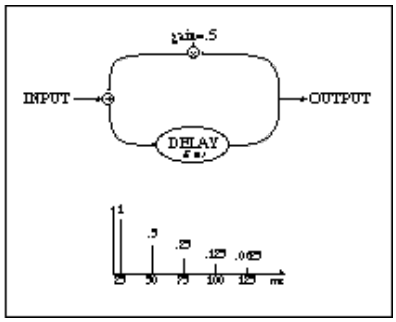


Figure 1.50 Flowchart of a **comb** filter and its impulse response.

The time it takes for the signal to circulate back to the input is called the loop-time. As demonstrated in *instr 135*, shown in figures 1.51, 1.52 and 1.53, a **diskin** opcode is used to play (both forward and reverse) and transpose samples directly from disk into a **comb** filter. When the loop-time is long, we will perceive discrete echoes, but when the loop-time is short, the **comb** filter will function more like a resonator. As shown in figure 1.50, the impulse response of a **comb** filter is a train of impulses spaced equally in time at the interval of the loop-time. In fact, the resonant frequency of this filter is 1/loop-time. In *instr 135* this is specified in milliseconds. In the score comments, you will see where I have converted the period of the loop, specified in milliseconds, into the frequency of the resonator, specified in Hz.

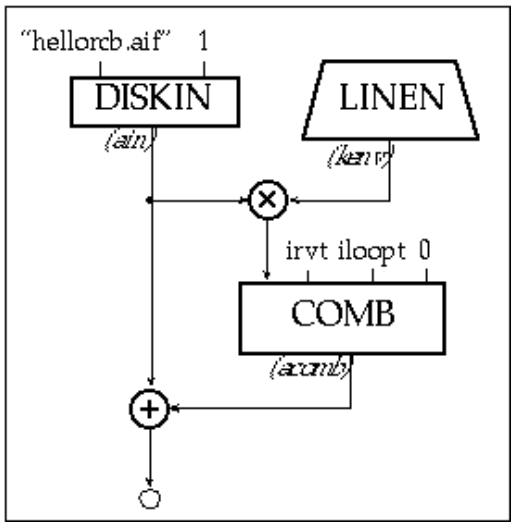


Figure 1.51 Block diagram of *instr 135,* a soundfile delay/resonator instrument using a **diskin** to read directly from disk without the use of an f-table and a **comb** to delay or resonate.

```
      instr 135     ; DISKIN ECHO-RESONATOR
idur  =     p3
iamp  =     p4
irvt  =     p5
iloopt=     p6
kenv  linen iamp, .01, idur, .01
ain   diskin"hellorcb.aif", 1
acomb comb  ain*kenv, irvt, iloopt, 0
      out   ain+acomb
      endin
```

Figure 1.52 Orchestra code for *instr 135,* an echo-resonator instrument using the **comb** opcode.

```
;    st  dur amp gainlooptimeresonant frequency
ins
i    0   5   .4  10   .5       ; 1/.5    = 2 Hz
135
i    5   5   .3  5    .25      ; 1/.25   = 4 Hz
```

```
135
i    10 5   .3  5    .125     ; 1/.125 = 8 Hz
135
i    15 5   .2  2    .0625    ; 1/.0625= 16 Hz
135
i    20 5   .2  2    .03125   ;         = 32 Hz
135                           1/.03125
i    25 5   .2  2    .015625  ;         = 64 Hz
135                           1/.015625
i    30 5   .04 2    .001     ; 1/.001 = 1000 Hz
135
```

Figure 1.53 Score code for *instr 135,* the looptime (*p6*) sets the period and resonant frequency of this recirculating delay line.

Although we can vary the loop time in *instr 135* on a note-by-note basis, by design, the **comb** opcode will not allow you to dynamically vary this parameter during the course of a note event, but the **vdelay** opcode will! And variable delay lines are the key to designing one of the more popular synthesizer effects processors — a "flanger."

In *instr 136*, as shown in figures 1.54 and 1.55, noise cascades through a series of variable delay lines to make a "flanger." By patching the output from one **vdelay** opcode into the input of another, the strength and focus of the characteristic resonance is emphasized (just as in our **tone** example above). Furthermore, this resonant peak is swept across the frequency spectrum under the control of a variable rate LFO whose frequency is dynamically modified by the **line** opcode.

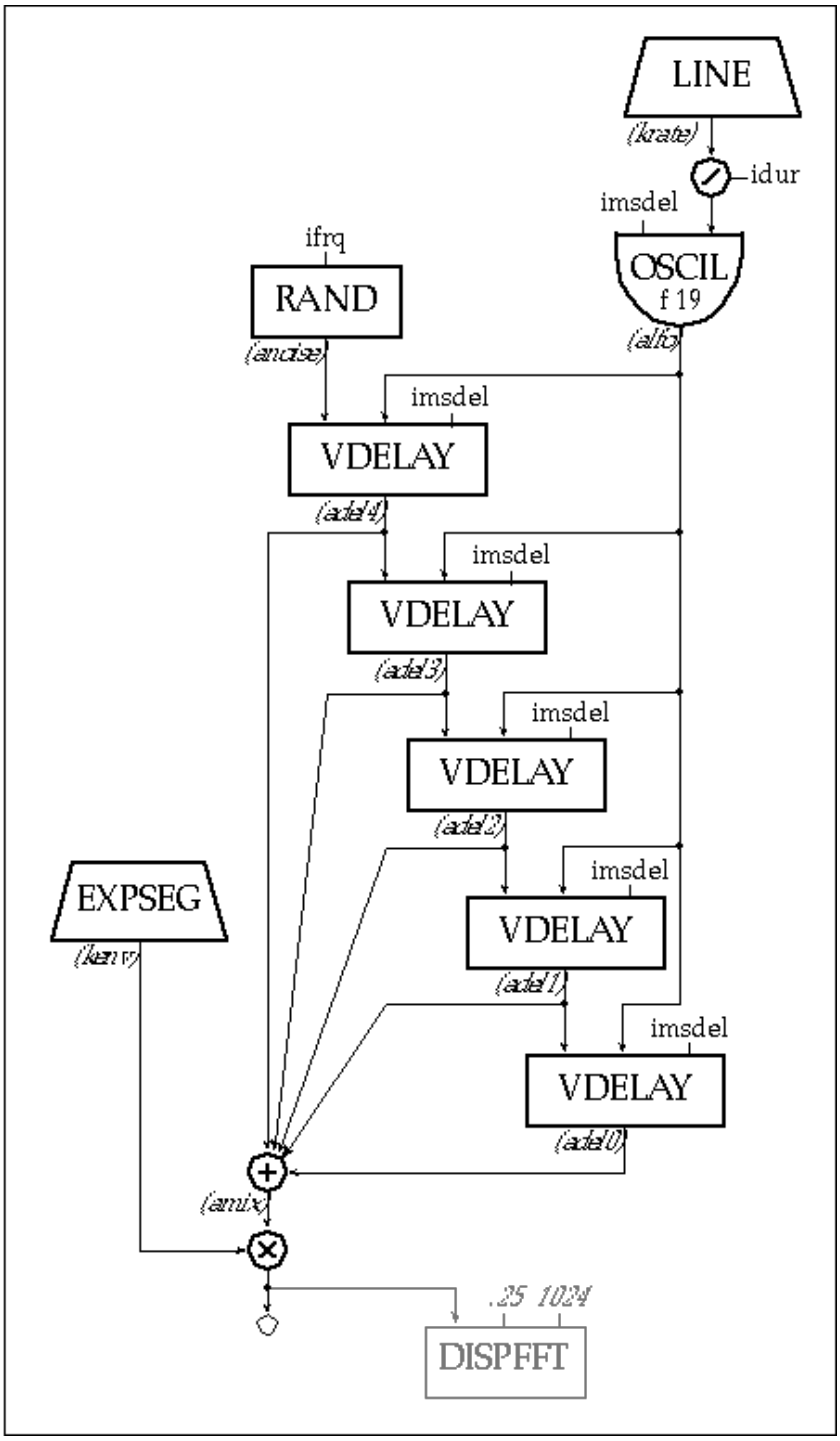


Figure 1.54 Block diagram of *instr 136*, a vdelay "flanger."

```
        instr   136         ; VDELAY FLANGER
idur    =       p3
iamp    =       p4
ifrq    =       p5
iatk    =       p6
irel    =       p7
irat1   =       p8
irat2   =       p9
imsdel  =       p10
kenv    expseg  .001,iatk,iamp,idur/8,iamp*.3,idur-
```

```
                (iatk+irel+idur/8),iamp*.7,irel,.01
krate  line     irat1, idur, irat2
alfo   oscil    imsdel, krate/idur, 19
anoise rand     ifrq
adel4  vdelay   anoise, alfo, imsdel
adel3  vdelay   adel4, alfo, imsdel
adel2  vdelay   adel3, alfo, imsdel
adel1  vdelay   adel2, alfo, imsdel
adel0  vdelay   adel1, alfo, imsdel
amix   =        adel0+adel1+adel2+adel3+adel4
       out      kenv*amix
       dispfft  amix, idur, 1024
       endin
```

Figure 1.55 Orchestra code for *instr 136,* a variable delay line "flanger" instrument.

Score Statements and Note-List Shortcuts

Admittedly, creating and editing text-based note-lists is not fun. Granted, the note-list does offer you the most exacting and direct control over the behavior of your instruments, but it is still one of the most unmusical and tedious aspects of working with Csound.

As stated at the very beginning of this chapter, Csound does read MIDI files and this may prove a more intuitive way of generating notes and playing your Csound instruments. However, Csound instruments must be designed specifically to work with MIDI and you will need to adapt your traditional Csound instruments before they will work with MIDI devices.

Although not covered in the text of The Csound Book, there are a number of chapters on the CD-ROM dedicated to controlling Csound from MIDI keyboards and MIDI files. In fact, I have written a compliment to this chapter entitled *An Introduction to MIDI-based Instrument Design in Csound* for the CD-ROM and I hope it helps you to develop some really great MIDI instruments.

Still, without resorting to MIDI, Csound does feature a collection of **Score Statements and Score Symbols** (text-based shortcuts) that were created to simplify the process of creating and editing note-lists. Like f-statements, these score commands begin with a specific letter and are sometimes followed by a set of arguments. I employ many of them in *etude5.sco*.

The first score statement that I employ in *etude5.sco* is the **Advance Statement** — ***a***, shown in figure 1.56. The advance statement allows the "beat-count" of a score to be advanced without generating any sound samples. Here it is used to skip over the first two notes of the score and begin rendering 10 seconds into the "piece." The advance statement can be particularly useful when you are working on a long and complex composition and you are interested in fine-tuning something about a sound in the middle or at the very end of the piece. Rather than waiting for the entire work to render just to hear the last note, you can advance to the end and only render that section — saving yourself hours and hours. The syntax of the advance statement is shown in the comments in figure 1.56.

```
;       NO        ADVANCE   ADVANCE SKIP
ADVANCEMEANING    START
a       0         0         10

; ins st dur  amp  frq   atk rel  cf1  cf2   bw1  bw2

i 128 1  5    .5   20000.5  2    8000 200   800  30
i 128 6  5    .5   20000.25 1    200  1200010   200

i 128 10 5    .5   20000.5  2    8000 200   800  30
i 128 14 5    .5   20000.25 1    200  1200010   200
i 128 18 3    .5   20000.15 .1   800  300   300  40
```

Figure 1.56 Score excerpt from *etude5.sco* featuring the **advance statement**.

The second score statement that I employ in *etude5.sco* is the **Section Statement** — ***s***, shown in figure 1.57. The section statement has no arguments. It merely divides a score into sections and allow you to begin counting again from time 0. This is particularly useful if you want to repeat a passage. To do so, you would simply insert an **s** at the end of the first section,copy the section, and

paste it after the **s**. Figure 1.57, again from *etude5.sco*, shows exactly this use.

```
; ins st    dur   amp   frq   atk   rel   cut1  cut2

i 129 0     1.5   3     20000 .1    .1    500   500
i 130 2     1.5   3     20000 .1    .1    500   500
i 131 4     1.5   3     20000 .1    .1    500   500
i 132 6     1.5   3     20000 .1    .1    500   500
i 129 8     1.2   1     20000 .01   .01   5000  40
i 130 11    1.2   1     20000 .01   .01   5000  40
i 131 12    1.2   1     20000 .01   .01   5000  40
i 132 13    1.2   1     20000 .01   .01   5000  40
s
; ins st    dur   amp   frq   atk   rel   cut1  cut2

i 129 0     1.5   3     20000 .1    .1    500   500
i 130 2     1.5   3     20000 .1    .1    500   500
i 131 4     1.5   3     20000 .1    .1    500   500
i 132 6     1.5   3     20000 .1    .1    500   500
i 129 8     1.2   1     20000 .01   .01   5000  40
i 130 11    1.2   1     20000 .01   .01   5000  40
i 131 12    1.2   1     20000 .01   .01   5000  40
i 132 13    1.2   1     20000 .01   .01   5000  40
s
```

Figure 1.57 Cut-and-paste repeated score excerpt from *etude5.sco* featuring the **section statement**.

The third score statement that I employ in *etude5.sco* is the **Dummy f-statement** — ***f 0***, shown in figure 1.58. In Csound you can use the score to load f-tables into memory at any time. This would allow you to replace one waveshape or sample with another during the course of a piece and yet refer to them with the same table number in the orchestra. Likewise, you can also load in a dummy f-table (*f 0*) at any time in a score as a means of extending the length of a particular score section or inserting silence between sections. As illustrated in figure 1.58, I am using an *f 0* statement to insert two seconds of silence between two score sections.

```
; ins st     dur  amp  frq      atk  rel   cut1     cut2

i 129 0      1.5  3    20000    .1   .1    500      500
i 130 2      1.5  3    20000    .1   .1    500      500
i 131 4      1.5  3    20000    .1   .1    500      500
i 132 6      1.5  3    20000    .1   .1    500      500
i 129 8      1.2  1    20000    .01  .01   5000     40
i 130 11     1.2  1    20000    .01  .01   5000     40
i 131 12     1.2  1    20000    .01  .01   5000     40
i 132 13     1.2  1    20000    .01  .01   5000     40

s
f 0    2     ; DUMMY F0 STATEMENT - 2 SECONDS OF
             SILENCE BETWEEN SECTIONS
s

; ins st   duramp  frq   atk  rel cf1   cf2    bw1   bw2

i 128 0    5   .5   20000 .5   2   8000  200    800   30
i 128 4    5   .5   20000 .25  1   200   12000  10    200
i 128 8    3   .5   20000 .15  .1  800   300    300   40
i 128 10   11  .5   20000 1    1   40    90     10    40
s
```

Figure 1.58 A score excerpt from *etude5.sco* featuring the **dummy *f0* statement** used to insert 2 seconds of silence between two score sections.

The fourth set of score shortcuts that I employ in *etude5.sco* are the **Carry**, **Ramp** and **+ Symbols**, as shown in figure 1.59. The carry symbol (**.**) copies a p-field value from one note-statement to the next. The ramp symbol (<) linearly interpolates between two p-field values over any number of notes (The number of notes determines the number of points in the interpolation). The + symbol only works for *p2*. There it automatically calculates the start time of the current note by adding together the start-time and duration of the previous note (*p2* + *p3*). Thus the current note will literally be consecutive with the previous one. All three symbols are used in figure 1.59 and the shorthand is "translated" in figure 1.60.

```
; ins st   dur ampdbfrq   fc1   fc2   resongain

i 134 0    .1  90    8.09  8000  80    1
i .   +    .   <     8.095 <     <     <
i .   .    .   .     8.10  .     .     .
i .   .    .   .     8.105 .     .     .
i .   .    .   .     8.11  .     .     .
i .   .    .   .     8.115 .     .     .
i .   .    .   .     9.00  .     .     .
i .   .    .   .     9.005 .     .     .
i .   .    .   .     9.01  .     .     .
i .   .    .   .     9.015 .     .     .
i .   .    .   80    9.02  9000  60    50
```

Figure 1.59 A score excerpt from *etude5.sco* featuring the **carry** (**.**), **p2 increment** (**+**), **and ramp** (**<**)**symbols.**

```
; ins st   dur ampdbfrq   fc1   fc2   resongain

i 134 0    .1  90    8.09  8000  80    1
i 134 .1   .1  89    8.095 8100  78    5
i 134 .2   .1  88    8.10  8200  76    10
i 134 .3   .1  87    8.105 8300  74    15
i 134 .5   .1  86    8.11  8400  72    20
i 134 .5   .1  85    8.115 8500  70    25
i 134 .6   .1  84    9.00  8600  68    30
i 134 .7   .1  83    9.005 8700  66    35
i 134 .8   .1  82    9.01  8800  64    40
i 134 .9   .1  81    9.015 8900  62    45
i 134 1    .1  80    9.02  9000  60    50
```

Figure 1.60 Another view of the *etude5.sco* excerpt shown in figure 1.59 in which the **ramp** (**<**), **carry** (**.**) and **+** symbols are replaced by the actual numerical values they represent.

The final score statement that I employ in *etude5.sco* is the **Tempo Statement** — ***t,*** shown in figure 1.61. The Csound score clock runs at 60 beats per minute. By default Csound inserts a tempo statement of 60 (60 beats-per-minute or 1-beat-per-second) at the beginning of every score (*t 0 60*). Obviously, this means that when you specify a duration of 1 in *p3* the note-event will last for 1 second. Luckily the t-statement lets you change this default value of 60 in both a constant and a variable fashion. Figure 1.61 illustrates both uses.

The statement *t 0 120* will set a constant tempo of 120 beats per minute. Given this setting, the internal "beat-clock" will run twice as fast and therefore all time values in the score file will be cut in half.

The statement *t 0 120 1 30* is used to set a variable tempo. In this case the tempo is set to 120 at time 0 (twice as fast as indicated in the score) and takes 1 second to gradually move to a new tempo of 30 (twice as slow as indicated in the score). It goes without saying that variable tempo can make your scores much less mechanical and much more "musical."

```
; t  0   120          ; FIXED TEMPO STATEMENT: TWICE
                       AS FAST

;    st  dur ampdbfrq   fc1    fc2    resongain
ins

i    0   .1  90    8.09  8000   80     1
```

```
134
i  .  +   .   <    8.095 <      <      <
i  .  .   .   .    8.10  .      .      .
i  .  .   .   .    8.105 .      .      .
i  .  .   .   .    8.11  .      .      .
i  .  .   .   .    8.115 .      .      .
i  .  .   .   .    9.00  .      .      .
i  .  .   .   .    9.005 .      .      .
i  .  .   .   .    9.01  .      .      .
i  .  .   .   .    9.015 .      .      .
i  .  .   .   80   9.02  9000   60     50
s
; t  0   120 1    30 ; VARIABLE TEMPO: TWICE AS FAST
                      TO HALF AS FAST

;    st  dur ampdbfrq   fc1    fc2    resongain
ins

i    0   .1  90   8.09  8000   80     1
134
i  .  +   .   <    8.095 <      <      <
i  .  .   .   .    8.10  .      .      .
i  .  .   .   .    8.105 .      .      .
i  .  .   .   .    8.11  .      .      .
i  .  .   .   .    8.115 .      .      .
i  .  .   .   .    9.00  .      .      .
i  .  .   .   .    9.005 .      .      .
i  .  .   .   .    9.01  .      .      .
i  .  .   .   .    9.015 .      .      .
i  .  .   .   80   9.02  9000   60     50
s
```

Figure 1.61 An excerpt from the end of *etude5.sco* in which the **tempo statement** is used in fixed and variable mode.

Working with Csound's text-based "score language" can be extremely laborious. In fact it has inspired many a student to learn C programming in order to generate their note-lists algorithmically. Real-time and MIDI are one solution. But taking advantage of Csound's score shortcuts can make your work a lot easier and your sound gestures, phrases and textures a lot more expressive.

Exercises for Etude 5

- Render the Csound orchestra and score: etude5.orc & etude5.sco.
- Play and listen to the different sound qualities of the various filters and filter configurations.
- Look up and read about the new opcodes used in *instr 128 — 136* in the *Csound Reference Manual*.

```
ar        rand       xamp[, iseed]
ar        tone       asig, khp[, istor]
ar        butterlp   asig, kfreq[, iskip]
ar        butterbp   asig, kfreq, kband[, iskip]
ar        delayr     idlt[, istor]
ar        comb       asig, krvt, ilpt[, istor]
ar        vdelay     asig, adel, imaxdel [, iskip]
a1[, a2[, diskin     ifilcod, kpitch[, iskiptim][, iwraparound][,
                     iformat]
a3, a4]]
```

- In *instr 128* substitute a **loscil** opcode for **rand** opcode and dynamically filter some of your own samples.
- In *instr 128* substitute a **butterbp** for the **reson** and listen to the difference in quality.

- Substitute **butterlp** filters for the **tone** filters in *instr 129 — 132*. Compare the "effectiveness."
- Transform *instr 133* into a resonant highpass filter instrument.
- Make an instrument that combines the serial filter design of *instr 132* with the parallel filter design of *instr 133*.
- Block diagram *instr 134*, a delay line instrument (not covered in the text).
- By adding more **delay** opcodes transform *instr 134* into a "multi-tap" delay line instrument.
- Modify *instr 135* to make a multi-band resonator.
- Add more **combs** and **vdelays** to *instr 135* and create a "multi-tap" delay with feedback/multi-band-resonator super-flange.
- Using the score statements covered in this section, return to etudes 3 and 4. In them, repeat some section, insert some silences, vary the tempo during sections, advance around a bit and ramp through some parameters to better explore the range of possibilities these instruments have to offer.
- In *instr 136*, substitute a **diskin** opcode for the **rand** opcode and "flange" your samples.
- In *instr 136*, add and explore the dynamic frequency and amplitude modification of the control oscillator.
- Change the waveforms of the control oscillator in *instr 136*. (Try **randh**!)
- Add a resonant-lowpass filter to your modified "flanger" instrument.
- Go for a walk and listen to your world.

Global Variables

Until now, we have used only local **i**-rate, **k**-rate and **a**-rate variables. These have been "local" to the instrument. Local variable are great because you could use the same variable name in separate instruments without ever worrying about the *asig* or *amix* data getting corrupted or signals "bleeding-through" from one instrument to another. In fact, the **instr** and **endin** delimiters truly do isolate the signal processing blocks from one another — even if they have the same exact labels and argument names.

However, there are times when you would like to be able to communicate across instruments. This would make it possible to pass the signal from a synthesis instrument to a reverb instrument, similar to the way one routes the signals on a mixing console to the effects units, using "aux sends" and "aux returns." In Csound this same operation is achieved using *global* variables. Global variables are variables that are accessible by all instruments. And like local variables, global variables are updated at the four basic rates, **setup**, **gi**, **gk** and **ga**, where:

gi-rate variables are changed and updated at the note rate.

gk-rate variables are changed and updated at the control rate.

ga-rate variables are changed and updated at the audio rate.

Because global variable belong both to all instruments and to none, they must be *initialized*. A global variable is typically initialized in **instrument 0** and "filled" from within a "local" instrument. Where is this mysterious "instrument 0?" Well, instrument 0 is actually the lines in the orchestra file immediately following the header section and before the declaration of the first *instr*. Thus, in figure 1.62, immediately after the header, in instrument 0, the *gacmb* and *garvb* variables (our 2 global FX buses) are cleared and initialized to 0.

```
sr     =     44100
kr     =     4410
ksmps  =     10
nchnls=      1
```

```
gacmb init 0
garvb init 0


      instr137
```

Figure 1.62 The two global variables *gacmb* and *garvb* are initialized in **instrument 0** after the header and before the "first" *instr*.

Sound Design Etude 6: Reverb and Panning

etude6.orc etude6.sco

Let's put the global variable to use and add some "external" processing to our instruments.

From within *instr 137*, shown in figures 1.63 and 1.64, the "dry" signal from **loscil** is "added" (mixed) with the "wet" signal on a separate reverb and echo bus.

```
asig  loscilkenv, ifrq, ifun
      out   asig
garvb =     garvb+(asig*irvbsnd)
gacmb =     gacmb+(asig*icmbsnd)
```

Note that the "dry" signal is still sent out directly, using the **out** opcode, just as we have from our very first instrument. But in this case that same signal is also globally passed "out of the instrument" and into two others, in this case *instr 198* (echo) and 199 (reverb) as shown in figures 1.63 and 1.64.

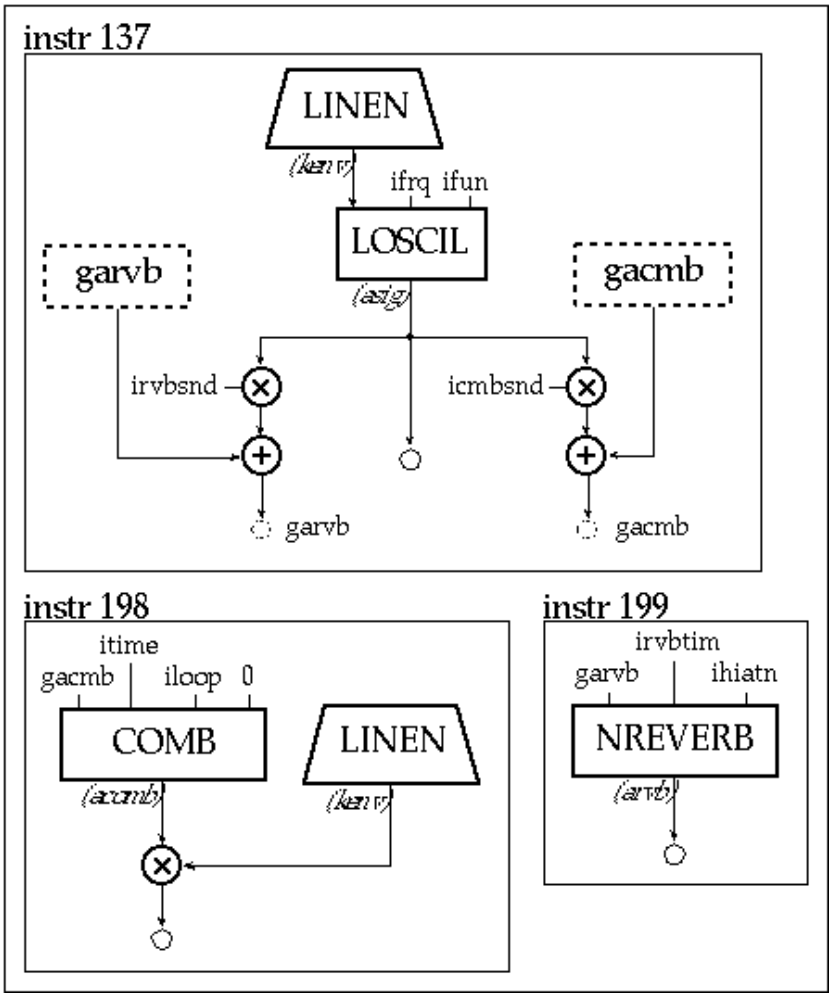


Figure 1.63 Block diagram for *instr 137, 198* and *199,* wavetable synthesis instrument (*instr 137*) and two global effects (*instr 198 and 199*).

```
        instr   137     ; GLOBAL COMB/VERB LOSCIL
idur    =       p3
iamp    =       ampdb(p4)
ifrq    =       cpspch(p5)
ifun    =       p6
iatk    =       p7
irel    =       p8
irvbsnd=        p9
icmbsnd=        p10
kenv    linen   iamp, iatk, idur, irel
asig    loscil  kenv, ifrq, ifun
        out     asig
garvb   =       garvb+(asig*irvbsnd)
gacmb   =       gacmb+(asig*icmbsnd)
        endin

        instr   198     ; GLOBAL ECHO
idur    =       p3
itime   =       p4
iloop   =       p5
kenv    linen   1, .01, idur, .01
acomb   comb    gacmb, itime, iloop, 0
        out     acomb*kenv
gacmb   =       0
        endin

        instr   199     ; GLOBAL REVERB
idur    =       p3
irvbtim=        p4
ihiatn  =       p5
arvb    nreverbgarvb, irvbtim, ihiatn
        out     arvb
garvb   =       0
        endin
```

Figure 1.64 Orchestra code for three instruments that work together to add reverb (*instr 199*) and echo (*instr 198*) to a looping oscillator (*instr 137*).

It is important to note that in the score file (figure 1.65) all three of these instruments must be turned on. In fact, to avoid transient and artifact noises, global instruments are typically left on for the duration of the section and the global variable are always "cleared" when the receiving instrument is turned off (*gacmb = 0* and *garvb = 0*)

```
;   strt dur rvbtimEhfroll
ins
i   0    12  4.6    .8
199

;   strt dur time     loopt
ins
i   0    6   10       .8
198
i   0    6   10       .3
198
i   0    6   10       .5
198

;   strt dur amp  frq1 sampleatk relrvbsnd  cmbsnd
ins
```

```
i   0    2.1 70   8.09 5      .01  .01   .3      .6
137
i   1    2.1 70   8.09 5      .01  .01   .5      .6
137
```

Figure 1.65 Score file for our "global **comb**/**nreverb** **loscil**" instrument. The global **nreverb** instrument, *instr 199* is turned on at the beginning of the score and left on for the duration of the passage. Three copies of our global **comb** instrument *instr 198* are started simultaneously with different looptimes. Finally, two copies of our **loscil** instrument, *instr 137*, start one after another.

Our next instrument, *instr 138*, shown in figures 1.66 and 1.67 is based on an earlier FM design, but now the instrument has been enhanced with the ability to pan the signal.

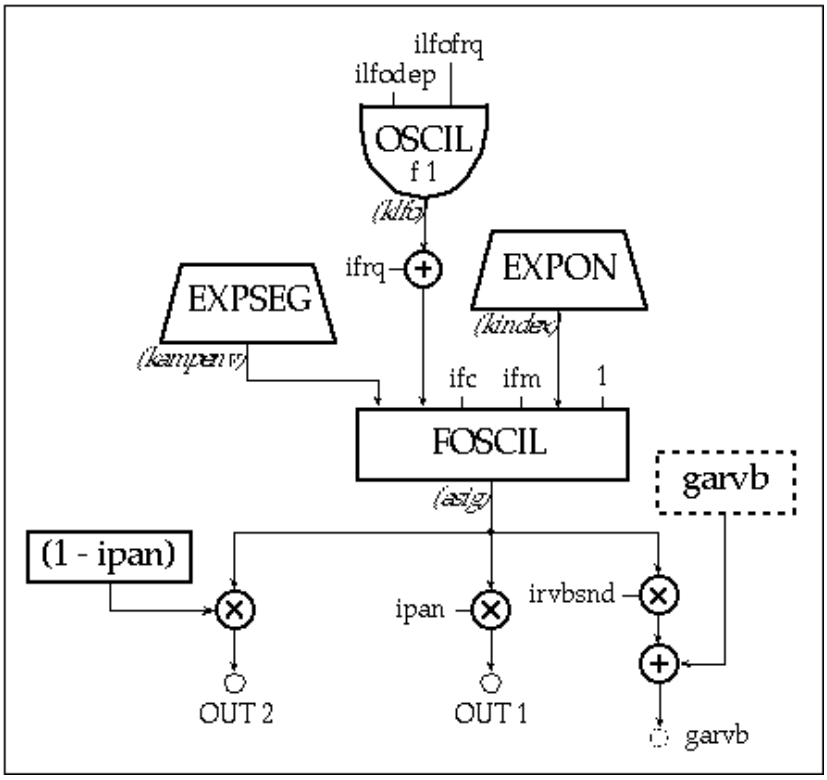


Figure 1.66 Block diagram of *instr 138,* a dynamic FM instrument with panning and global reverb.

```
        instr 138
idur    =     p3
iamp    =     ampdb(p4)
ifrq    =     cpspch(p5)
ifc     =     p6
ifm     =     p7
iatk    =     p8
irel    =     p9
indx1   =     p10
indx2   =     p11
indxtim=      p12
ilfodep=      p13
ilfofrq=      p14
ipan    =     p15
irvbsnd=      p16
kampenvexpseg.01,iatk,iamp,idur/9,iamp*.6,idur/(iatk+irel+idur/9),iamp*.7,irel,.01
klfo    oscil ilfodep, ilfofrq, 1
kindex expon indx1, indxtim, indx2
asig    foscilkampenv, ifrq+klfo, ifc, ifm, kindex, 1
        outs  asig*ipan, asig*(1-ipan)
garvb   =     garvb+(asig*irvbsnd)
```

```
        endin
```

Figure 1.67 Orchestra code for *instr 138,* a dynamic FM instrument with vibrato, discrete panning and global reverb.

You should note that in *instr 138* the panning is realized using a single variable that functions like a panning knob on a traditional mixing console. How is this done?

Well, as you know by now, if we multiply a signal by a scalar in a range of 0 to 1 we effectively control the amplitude of the signal between 0 and 100%. So if we simultaneously multiply the signal by the scalar and its inverse we would have two outputs whose amplitudes would be scaleable between 0 and 100% but would be inversely proportional to each other.

For example, if the scalar is at 1 and that corresponds to 1 times the left output, we would have 100% of our signal from the left and (1 - 1) or 0% signal from the right. If on the other hand the amplitude scalar was set to .2, then we would have .2 times left or 20% of our signal coming from the left and 1 - .2 or .8 time the right signal or 80% coming from the right. This algorithm provides a simple means of using a single value to control the left/right strength of a signal and is used in *instr* 138.orc and instr138.sco and illustrated in figures 1.66 and 1.67.

The file *etude6.orc* contains four additional "globo-spatial" instruments. All are based on those presented in previous etudes. You should recognize the algorithms. But all have been enhanced with panning and global reverb capabilities. You are encouraged to block diagram and study them. Each demonstrates a different panning and reverb approach. You are also encouraged to go back and add global reverb and panning to all of the instruments we have studied so far.

To end the chapter I will present instr 141.orc and instr141.sco, shown in figures 1.68 and 1.69, which adapts an earlier amplitude modulation design and adds both global reverb and LFO-based panning.

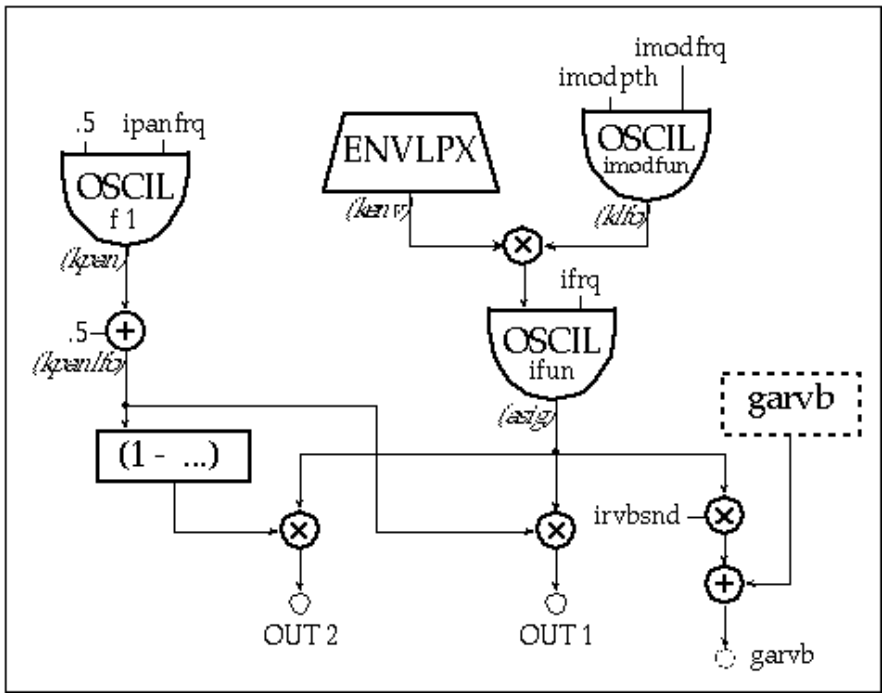


Figure 1.68 Block diagram of *instr 141,* an Amplitude Modulation instrument with an LFO panner and global reverb.

```
        instr 141     ; AM LFO PANNER
idur    =       p3
iamp    =       ampdb(p4)
ifrq    =       cpspch(p5)
ifun    =       p6
iatk    =       p7
irel    =       p8
iatkfun=        p9
imodpth=        p10
```

```
imodfrq=     p11
imodfun=     p12
ipanfrq=     p13
irvbsnd=     p14
kenv   envlpxiamp, iatk, idur, irel, iatkfun,
             .7, .01
kpan   oscil .5, ipanfrq, 1
klfo   oscil imodpth, imodfrq, imodfun
asig   oscil klfo*kenv, ifrq, ifun
kpanlfo=     kpan+.5
       outs  asig*kpanlfo, asig*(1-kpanlfo)
garvb  =     garvb+(asig*irvbsnd)
       endin
```

Figure 1.69 Orchestra code for *instr 141,* an Amplitude Modulation instrument with an LFO panner and global reverb.

Notice here that the amplitude of the panning LFO is set to .5. This means that this bipolar sinewave has a range of -.5 to +.5. Then notice that I "bias" this bipolar signal by adding .5 to it (*kpanlfo = kpan + .5*). This makes the signal unipolar! Now the sinewave goes from 0 to 1 with its center point at .5. And isn't that perfect for our "panning" knob that needs to be in the range of 0 to 1? Panning envelopes, panning oscillators, and random panners can all be realized with this simple solution!

Exercises for Etude 6

etude6.orc etude6.sco

- Write a set of short "musical" etudes using your modified Csound instruments from this chapter and email them to me: **csound@mediaone.net.**

Conclusion

In this introductory chapter I have attempted to introduce the syntax of the Csound language while covering some of the elements of sound design. Given this basic understanding, the subsequent chapters of this text, written by the world's leading educators, sound designers, programmers and composers, should serve to unlock the secret power of Csound and help you find the riches buried therein. Along the way, I sincerely hope that you not only discover some exquisite new sounds, but that your work with Csound brings you both a deeper understanding of your current synthesizer hardware and a deeper appreciation and fuller awareness of the nature and spirit of musical sound itself... the language of the soul.

References

Cage, J. 1976. *Silence*. Middletown, CT: Wesleyan University Press.

Chadabe, J. 1997. *Electric Sound: The Past and Promise of Electronic Music*. New York: Prentice Hall.

De Poli, G., A. Piccialli, and C. Roads. 1991. *Representations of Musical Signals*. Cambridge, MA: M.I.T. Press.

De Poli, G., A. Piccialli, S. T. Pope, Stephen and C. Roads. 1997. *Musical Signal Processing*. The Netherlands: Swets and Zeitlinger.

Dodge, C. and T. Jerse. 1997. *Computer Music*. 2nd rev. New York: Schirmer Books.

Eliot, T.S. 1971. *Four Quartets*. New York: Harcourt Brace & Company.

Mathews, Max V. 1969. *The Technology of Computer Music*. Cambridge, MA: M.I.T. Press.

Mathews, Max V. and J. R. Pierce. 1989. *Current Directions in Computer Music Research*. Cambridge, MA: M.I.T. Press.

Moore, R. F. 1990. *Elements of Computer Music*. New York: Prentice Hall.

Pierce, J. R. 1992. *The Science of Musical Sound*. 2nd rev. edn. New York: W. H. Freeman.

Pohlmann, Ken C. 1995. *Principles of Digital Audio*. 3d edn. New York: McGraw-Hill.

Roads, C. 1989.*The Music Machine*. Cambridge, MA: M.I.T. Press.

Roads, C. 1996. *The Computer Music Tutorial*. Cambridge, MA: M.I.T. Press.

Roads, C. and J. Strawn. 1987. *Foundations of Computer Music*. 3d edn. Cambridge, MA: M.I.T. Press.

Steiglitz, K. 1996. *A Digital Signal Processing Primer*. Reading, MA: Addison-Wesley.