--
--

CSOUND

A Manual for the

Audio Processing System

and

Supporting Programs

Barry Vercoe
Media Lab
M.I.T.

--
--

Developed by Barry L. Vercoe at the Experimental Music Studio,
Media Laboratory, M.I.T., Cambridge, Massachusetts,
with partial support from the System Development Foundation
and from National Science Foundation Grant # IRI-8704665.

-- --

CONTENTS

-- --

PREFACE

Realizing music by digital computer involves synthesizing audio signals with discrete points or *samples* that are representative of continuous waveforms. There are several ways of doing this, each affording a different manner of control. *Direct synthesis* generates waveforms by sampling a stored *function* representing a single cycle; *additive synthesis* generates the many partials of a complex tone, each with its own loudness envelope; *subtractive synthesis* begins with a complex tone and filters it. *Non-linear synthesis* uses frequency modulation and wave shaping to give simple signals complex characteristics, while *sampling* and storage of natural sound allows it to be used at will.

Since comprehensive moment-by-moment specification of sound can be tedious, control is gained in two ways: 1) from the *instruments* in an *orchestra*, and 2) from the *events* within a *score*. An orchestra is really a computer program that can produce sound, while a score is a body of data which that program can react to. Whether a rise-time characteristic is a fixed constant in an instrument, or a variable of each note in the score, depends on how the user wants to control it.

The instruments in a **Csound** orchestra are defined in a simple syntax that invokes complex audio processing routines. A score passed to this orchestra contains numerically coded pitch and control information, in *standard numeric score* format. Although many users are content with this format, higher level score processing languages are often convenient. The **Scot** language uses simple alphanumeric encoding of pitch and time, in a fashion that parallels traditional music notation; its translator produces a *standard numeric* score. The **Cscore** program can expand an existing numeric score, according to user-supplied algorithms written in the **C** language. One powerful score strategy, then, is to define a *kernel* score in **Scot**, translate it to *numeric* form, then expand and modify the data using **Cscore** before sending it to a **Csound** orchestra for performance.

The programs making up the **Csound** system have a long history of development, beginning with the *Music 4* program written at Bell Telephone Laboratories in the early 1960's by Max Mathews. That initiated the stored table concept and much of the terminology that has since enabled computer music researchers to communicate. Valuable additions were made at Princeton by the late Godfrey Winham in *Music 4B*; my own *Music 360* (1968) was very indebted to his work. With *Music 11* (1973) I took a different tack: the two distinct networks of *control* and *audio* signal processing stemmed from my intensive involvement in the preceding years in hardware synthesizer concepts and design. This division has been retained in **Csound**.

Because it is written entirely in **C, Csound** is easily installed on any machine running Unix or C. At MIT it runs on VAX/DECstations under Ultrix 4.0, on SUNs under OS 4.1, and on the Macintosh under ThinkC 4.0. With this single language for audio signal processing, users move easily from machine to machine.

The 1991 version has many new features. First, I am indebted to others for the contribution of the phase vocoder and FOF synthesis modules. This release also charts a new direction with the addition of a *spectral data type*, holding much promise for future development. Most importantly, with the advent of a new generation of RISC processors that are an order of magnitude faster than those on which computer music was born, researchers and composers now have access to workstations on which *realtime software synthesis with sensing and control* is now a reality. This is perhaps the single most important development for people working in the field. This new Csound is designed to take maximum advantage of realtime audio processing, and to encourage interactive experiments in this exciting new domain.

B.V.

1. A BEGINNING TUTORIAL

Introduction

The purpose of this section is to expose the reader to the fundamentals of designing and using computer music instruments in **Csound**. Only a small portion of the language will be covered here, sufficient to implement some simple instrument examples. The remaining sections in the text are arranged as a *Reference* manual (not a tutorial), since that is the form the user will eventually find most helpful when inventing instruments. Once the basic concepts are grasped from this tutorial, the reader might let himself into the remainder of the text by locating the information presented here in the Reference entries that follow.

The Orchestra File

Csound runs from two basic files: an *orchestra* file and a *score* file. The orchestra file is a set of *instruments* that tell the computer how to synthesize sound; the score file tells the computer when. An instrument is a collection of modular statements which either *generate* or *modify* a signal; signals are represented by *symbols*, which can be "patched" from one module to another. For example, the following two statements will generate a 440 Hz sine tone and send it to an output channel:

```
asig    oscil   10000, 440, 1
        out     asig
```

The first line sets up an oscillator whose controlling inputs are an amplitude of 10000, a frequency of 440 Hz, and a waveform number, and whose output is the audio signal *asig*. The second line takes the signal *asig* and sends it to an (implicit) output channel. The two may be encased in another pair of statements that identify the instrument as a whole:

```
        instr   1
asig    oscil   10000, 440, 1
        out     asig
        endin
```

In general, an orchestra statement in **Csound** consists of an action symbol followed by a set of input variables and preceded by a result symbol. Its *action* is to process the inputs and deposit the result where told. The meaning of the input variables depends on the action requested. The 10000 above is interpreted as an amplitude value because it occupies the first input slot of an oscil unit; 440 signifies a frequency in Hertz because that is how an oscil unit interprets its second input argument; the waveform number is taken to point indirectly to a stored function table, and before we invoke this instrument in a score we must fill function table #1 with some waveform.

The output of **Csound** computation is not a real audio signal, but a stream of numbers which describe such a signal. When written onto a sound file these can later be converted to sound by an independent program; for now, we will think of variables such as *asig* as tangible audio signals.

Let us now add some extra features to this instrument. First, we will allow the pitch of the tone to be defined as a *parameter* in the score. Score parameters can be represented by orchestra variables which take on their different values on successive notes. These variables are named sequentially: p1, p2, p3, ... The first three have a fixed meaning (see the Score File), while the remainder are assignable by the user. Those of significance here are:

p3 - duration of the current note (always in seconds).
p5 - pitch of the current note (in units agreed upon by score and orchestra).

Thus in

```
asig      oscil     10000, p5, 1
```

the oscillator will take its pitch (presumably in cps) from score parameter 5.

If the score had forwarded pitch values in units other than cycles-per-second (Hertz), then these must first be converted. One convenient score encoding, for instance, combines *pitch class* representation (00 for C, 01 for C#, 02 for D, ... 11 for B) with *octave* representation (8. for middle C, 9. for the C above, etc.) to give pitch values such as 8.00, 9.03, 7.11. The expression

```
          cpspch(8.09)
```

will convert the pitch A (above middle C) to its cps equivalent (440 Hz). Likewise, the expression

```
          cpspch(p5)
```

will first read a value from p5, then convert it from octave.pitch-class units to cps. This expression could be imbedded in our orchestra statement as

```
asig      oscil     10000, cpspch(p5), 1
```

to give the score-controlled frequency we sought.

Next, suppose we want to shape the amplitude of our tone with a linear rise from 0 to 10000. This can be done with a new action statement

```
amp       line      0, p3, 10000
```

Here, *amp* will take on values that move from 0 to 10000 over time p3 (the duration of the note in seconds). The instrument will then become

```
          instr     1
amp       line      0, p3, 10000
asig      oscil     amp, cpspch(p5), 1
          out       asig
          endin
```

The signal *amp* is not something we would expect to listen to directly. It is really a variable whose purpose is to control the amplitude of the audio oscillator. Although audio output requires fine resolution in time for good fidelity, a controlling signal often does not need that much resolution. We could use another kind of signal for this amplitude control

```
kamp      line      0, p3, 10000
```

in which the result is a new kind of signal. Signal names up to this point have always begun with the letter **a** (signifying an *audio* signal); this one begins with **k** (for *control*). Control signals are identical to audio signals, differing only in their resolution in time. A control signal changes its value less often than an audio signal, and is thus faster to generate. Using one of these, our instrument would then become

```
          instr     1
kamp      line      0, p3, 10000
asig      oscil     kamp, cpspch(p5), 1
          out       asig
          endin
```

This would likely be indistinguishable in sound from the first version, but would run a little faster. In general, instruments take constants and parameter values, and use calculations and signal processing to move first towards the generation of control signals, then finally audio signals. Remembering this flow will help you write efficient instruments with faster execution times.

We are now ready to create our first orchestra file. Type in the following orchestra using the system editor, and name it "intro.orc".

```
          sr = 20000                ; audio sampling rate is 20 kHz
          kr = 500                  ; control rate is 500 Hz
          ksmps = 40                ; number of samples in a control period (sr/kr)
```

```
        nchnls = 1                    ; number of channels of audio output

        instr   1
kctrl   line    0, p3, 10000                   ; amplitude envelope
asig    oscil   kctrl, cpspch(p5), 1  ; audio oscillator
        out     asig                           ; send signal to channel 1
        endin
```

It is seen that comments may follow a semi-colon, and extend to the end of a line. There can also be blank lines, or lines with just a comment. Once you have saved your orchestra file on disk, we can next consider the score file that will drive it.

The Score File

The purpose of the score is to tell the instruments when to play and with what parameter values. The score has a different syntax from that of the orchestra, but similarly permits one statement per line and comments after a semicolon. The first character of a score statement is an **opcode**, determining an action request; the remaining data consists of numeric parameter fields (pfields) to be used by that action.

Suppose we want a sine-tone generator to play a pentatonic scale starting at C-sharp above middle-C, with notes of 1/2 second duration. We would create the following score:

```
;    a sine wave function table
f1 0 256 10 1
;    a pentatonic scale
i1   0 .5  0 8.01
i1  .5  .  . 8.03
i1 1.0  .  . 8.06
i1 1.5  .  . 8.08
i1 2.0  .  . 8.10
e
```

The first statement creates a stored sine table. The protocol for generating wave tables is simple but powerful. Lines with opcode **f** interpret their parameter fields as follows:

p1 - function table *number* being created
p2 - *creation time*, or time at which the table becomes readable
p3 - table *size* (number of points), which must be a power of two or one greater
p4 - *generating subroutine*, chosen from a prescribed list.

Here the value 10 in p4 indicates a request for subroutine GEN10 to fill the table. GEN10 mixes harmonic sinusoids in phase, with relative strengths of consecutive partials given by the succeeding parameter fields. Our score requests just a single sinusoid. An alternative statement

```
f1 0 256 10 1 0 3
```

would produce one cycle of a waveform with third harmonic three times as strong as the first.

The *i*-statements, or note statements, will invoke the p1 instrument at time p2, then turn it off after p3 seconds; it will pass all of its p-fields to that instrument. Individual score parameters are separated by any number of spaces or tabs; neat formatting of parameters in columns is nice but unnecessary. The dots in p-fields 3 and 4 of the last four notes invoke a *carry feature*, in which values are simply copied from the immediately preceding note *of the same instrument*. A score normally ends with an *e*-statement.

The unit of time in a **Csound** score is the beat. In the absence of a *Tempo* statement, one beat takes one second. To double the speed of the pentatonic scale in the above score, we could either modify p2 and p3 for all the notes in the score, or simply insert the line

```
t 0 120
```

to specify a tempo of 120 beats per minute from beat 0.

Two more points should be noted. First, neither the *f*-statements nor the *i*-statements need be typed in time order; **Csound** will sort the score automatically before use. Second, it is permissable to play more than one note at a time with a single instrument. To play the same notes as a three-second pentatonic chord we would create the following:

```
;    a sine wave function
f1 0 256 10 1
;    five notes at once
i1   0  3 0 8.01
i1   0  .  . 8.03
i1   0  .  . 8.06
i1   0  .  . 8.08
i1   0  .  . 8.10
e
```

Now go into the editor once more and create your own score file. Name it "intro.sco".

The CSOUND Command

To request your orchestra to perform your score, type the command

```
csound  intro.orc  intro.sco
```

The resulting performance will take place in three phases:
1) sort the score file into chronological order. If score syntax errors are encountered they will be reported on your console.
2) translate and load your orchestra. The console will signal the start of translating each *instr* block, and will report any errors. If the error messages are not immediately meaningful, translate again with the *verbose* flag turned on:

```
csound  -v  intro.orc  intro.sco
```

3) fill the wave tables and perform the score. Information about this performance will be displayed throughout in messages resembling

```
B  4.000 ..  6.000   T 3.000  TT  3.000  M     7929.     7929.
```

A message of this form will appear for every *event* in your score. An event is defined as any change of state (as when a new note begins or an old one ends). The first two numbers refer to beats in your original score, and they delimit the current segment of sound synthesis between successive events (e.g. from beat 4 to beat 6). The second beat value is next restated in real seconds of time, and reflects the *tempo* of the score. That is followed by the Total Time elapsed for all sections of the score so far. The last values on the line show the maximum amplitude of the audio signal, measured over just this segment of time, and reported separately for each channel.

Console messages are printed to assist you in following the orchestra's handling of your score. You should aim at becoming an intelligent reader of your console reports. When you begin working with longer scores and your instruments no longer cause surprises, the above detail may be excessive. You can elect to receive abbreviated messages using the -m option of the **csound** command.

When your performance goes to completion, it will have created a sound file named *test* in your soundfile directory. You can now listen to your sound file by typing

```
play test
```

More about the Orchestra

Suppose we next wished to introduce a small vibrato, whose rate is 1/50 the frequency of the note (i.e. A440 is to have a vibrato rate of 8.8 Hz.). To do this we will generate a control signal using a second oscillator, then add this signal to the basic frequency derived from p5. This might result in the instrument

```
         instr     1
kamp     line      0, p3, 10000
```

```
kvib    oscil   2.75, cpspch(p5)/50, 1
a1      oscil   kamp, cpspch(p5)+kvib, 1
        out     a1
        endin
```

Here there are two control signals, one controlling the amplitude and the other modifiying the basic pitch of the audio oscillator. For small vibratos, this instrument is quite practical; however it does contain a misconception that is worth noting. This scheme has added a sine wave deviation to the cps value of an audio oscillator. The value 2.75 determines the *width* of vibrato in cps, and will cause an A440 to be modified about one-tenth of one semitone in each direction (1/160 of the frequency in cps). In reality, a cps deviation produces a different musical interval above than it does below. To see this, consider an exaggerated deviation of 220 cps, which would extend a perfect 5th above A440 but a whole octave below. To be more correct, we should first convert p5 into a *true decimal octave* (not cps), so that an *interval* deviation above is equivalent to that below. In general, pitch modification is best done in true octave units rather than pitch-class or cps units, and there exists a group of pitch converters to make this task easier. The modified instrument would be

```
        instr   1
ioct    =       octpch(p5)
kamp    line    0, p3, 10000
kvib    oscil   1/120, cpspch(p5)/50, 1
asig    oscil   kamp, cpsoct(ioct+kvib), 1
        out     asig
        endin
```

This instrument is seen to use a third type of orchestra variable, an *i*-variable. The variable *ioct* receives its value at an *initialization* pass through the instrument, and does not change during the lifespan of this note. There may be many such *init time* calculations in an instrument. As each note in a score is encountered, the event space is allocated and the instrument is initialized by a special pre-performance pass. *i-variables* receive their values at this time, and any other expressions involving just constants and *i*-variables are evaluated. At this time also, modules such as **line** set up their target values (such as beginning and end points of the line), and units such as **oscil** do phase setup and other bookkeeping in preparation for performance. A full description of init-time and performance-time activities, however, must be deferred to a general consideration of the orchestra syntax.

2. SYNTAX OF THE ORCHESTRA

An orchestra statement in **Csound** has the format:

label: result opcode argument1, argument2,... ;comments

The label is optional and identifies the basic statement that follows as the potential target of a go-to operation (see Program Control Statements). A label has no effect on the statement per se.

Comments are optional and are for the purpose of letting the user document his orchestra code. Comments always begin with a semicolon (;) and extend to the end of the line.

The remainder (result, opcode, and arguments) form the *basic statement*. This also is optional, i.e. a line may have only a label or comment or be entirely blank. If present, the basic statement must be complete on one line. The opcode determines the operation to be performed; it usually takes some number of input values (arguments); and it usually has a result field variable to which it sends output values at some fixed rate. There are four possible rates:

1) once only, at orchestra setup time (effectively a permanent assignment);
2) once at the beginning of each note (at initialization (init) time: *I-rate*);
3) once every performance-time control loop (perf time control rate, or *K-rate*);
4) once each sound sample of every control loop (perf time audio rate, or *A-rate*).

STATEMENT TYPES

An orchestra program in **Csound** is comprised of *orchestra header statements* which set various global parameters, followed by a number of *instrument blocks* representing different instrument types. An instrument block, in turn, is comprised of *ordinary statements* that set values, control the logical flow, or invoke the various signal processing subroutines that lead to audio output.

An *orchestra header statement* operates once only, at orchestra setup time. It is most commonly an assignment of some value to a *global reserved symbol*, e.g. sr = 20000. All orchestra header statements belong to a pseudo instrument 0, an *init* pass of which is run prior to all other instruments at score time 0. Any *ordinary statement* can serve as an orchestra header statement, eg. gifreq = cpspch(8.09), provided it is an init-time only operation.

An *ordinary statement* runs at either init time or performance time or both. Operations which produce a result formally run at the rate of that result (that is, at init time for I-rate results; at performance time for K- and A-rate results), with the sole exception of the **init** opcode (q.v.). Most **generators** and **modifiers**, however, produce signals that depend not only on the instantaneous value of their arguments but also on some preserved internal state. These performance-time units therefore have an implicit init-time component to set up that state. The run time of an operation which produces no result is apparent in the opcode.

Arguments are values that are sent to an operation. Most arguments will accept arithmetic expressions composed of constants, variables, reserved globals, value converters, arithmetic operations and conditional values; these are described below.

CONSTANTS AND VARIABLES

constants are floating point numbers, such as 1, 3.14159, or -73.45 . They are available continuously and do not change in value.

variables are named cells containing numbers. They are available continuously and may be updated at one of the four update rates (setup only, I-rate, K-rate, or A-rate). I- and K-rate variables are scalars (i.e. they take on only one value at any given time) and are primarily used to store and recall controlling data, that is, data that changes at the note rate (for I-variables) or at the control rate (for K-variables). I- and K-variables are therefore useful for storing note parameter values, pitches, durations, slow-moving frequencies, vibratos, etc. A-variables, on the other hand, are arrays or vectors of information. Though renewed on the same perf-time control pass as K-variables, these array cells represent a finer resolution of time by dividing the control period into sample periods (see *ksmps* below). A-variables are used to store and recall data changing at the audio sampling rate (e.g. output signals of oscillators, filters, etc.).

A further distinction is that between local and global variables. **local** variables are private to a particular instrument, and cannot be read from or written into by any other instrument. Their values are preserved, and they may carry information from pass to pass (e.g. from initialization time to performance time) within a single instrument. Local variable names begin with the letter **p, i, k,** or **a.** The same local variable name may appear in two or more different instrument blocks without conflict.

global variables are cells that are accessible by all instruments. The names are either like local names preceded by the letter **g,** or are special reserved symbols. Global variables are used for broadcasting general values, for communicating between instruments (semaphores), or for sending sound from one instrument to another (e.g. mixing prior to reverberation).

Given these distinctions, there are eight forms of local and global variables:

type	when renewable	**Local**	**Global**
reserved symbols	permanent	--	rsymbol
score parameter fields	I-time	**p**number	--
init variables	I-time	**i**name	**gi**name
control signals	P-time, K-rate	**k**name	**gk**name
audio signals	P-time, A-rate	**a**name	**ga**name

where *rsymbol* is a special reserved symbol (e.g. **sr, kr**), *number* is a positive integer referring to a score statement pfield, and *name* is a string of letters and/or digits with local or global meaning. As might be inferred, score parameters are local I-variables whose values are copied from the invoking score statement just prior to the Init pass through an instrument.

VALUE CONVERTERS:

ftlen(x)	(init-rate args only)
int(x)	(init- or control-rate args only)
frac(x)	" "
dbamp(x)	" "
i(x)	(control-rate args only)
abs(x)	(no rate restriction)
exp(x)	" "
log(x)	" "
sqrt(x)	" "
sin(x)	" "
cos(x)	" "
ampdb(x)	" "

where the argument within the parentheses may be an expression.

Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

ftlen(x)	returns the size (no. of points) of stored function table no. *x*.
int(x)	" " integer part of *x*.
frac(x)	" " fractional part of *x*.
dbamp(x)	" decibel equivalent of the raw amplitude *x*.
i(x)	" an Init-type equivalent of the argument, thus permitting a K-time value to be accessed in at init-time or reinit-time, whenever valid.
abs(x)	" the absolute value of *x*.
exp(x)	" *e* raised to the *x*th power.
log(x)	" the natural log of *x* (*x* positive only).
sqrt(x)	" " square root of *x* (*x* non-negative).
sin(x)	" " sine of *x* (*x* in radians).
cos(x)	" " cosine of *x* (*x* in radians).
ampdb(x)	" amplitude equivalent of the decibel value *x*. Thus 60 db gives 1000, 66 db gives 2000, 72 db gives 4000, 78 db gives 8000, 84 db gives 16000 and 90 db gives 32000.

Note that for **log**, **sqrt**, and **ftlen** the argument value is restricted.

Note also that **ftlen** will always return a power-of-2 value, i.e. the function table guard point (see F statement) is not included.

PITCH CONVERTERS

octpch(pch)	(init- or control-rate args only)
pchoct(oct)	" "
cpspch(pch)	" "
octcps(cps)	" "
cpsoct(oct)	(no rate restriction)

where the argument within the parentheses may be a further expression.

These are really **value converters** with a special function of manipulating pitch data.

Data concerning pitch and frequency can exist in any of the following forms:

name	abbreviation
octave point pitch-class (8ve.pc)	**pch**
octave point decimal	**oct**
cycles per second	**cps**

The first two forms consist of a whole number, representing octave registration, followed by a specially interpreted fractional part. For **pch** the fraction is read as two decimal digits representing the 12 equal-tempered pitch classes from .00 for C to .11 for B. For **oct**, the fraction is interpreted as a true decimal fractional part of an octave. The two fractional forms are thus related by the factor 100/12. In both forms, the fraction is preceded by a whole number octave index such that 8.00 represents Middle C, 9.00 the C above, etc. Thus A440 can be represented alternatively by 440 (**cps**), 8.09 (**pch**), 8.75 (**oct**), or 7.21 (**pch**), etc. Microtonal divisions of the **pch** semitone can be encoded by using more than two decimal places.

The mnemonics of the pitch conversion units are derived from morphemes of the forms involved, the second morpheme describing the source and the first morpheme the object (result). Thus

cpspch(8.09)

will convert the pitch argument 8.09 to its c.p.s. (or hertz) equivalent, giving the value of 440. Since the argument is constant over the duration of the note, this conversion will take place at I-time, before any samples for the current note are produced. By contrast, the conversion

cpsoct(8.75 + K1)

which gives the value of A440 transposed by the octave interval K1, will repeat the calculation every K-period since that is the rate at which K1 varies.

N.B. The conversion from **pch** or **oct** into **cps** is not a linear operation but involves an exponential process which may be time-consuming if executed repeatedly at audio rates. Audio-rate arguments within **cpsoct** are permitted but should be used sparingly.

ARITHMETIC OPERATIONS:

$-a$
$+a$
$a \&\& b$ (logical AND; not audio-rate)
$a || b$ (logical OR; not audio-rate)
$a + b$
$a - b$
$a * b$
a / b

where the arguments *a* and *b* may be further expressions.

Arithmetic operators perform operations of change-sign (negate), don't-change-sign, logical AND, logical OR, add, subtract, multiply and divide. Note that a value or an expression may fall between two of these operators, either of which could take it as its left or right argument, as in

$$a + b * c.$$

In such cases three rules apply:

1) * and / bind to their neighbors more strongly than + and -.
   Thus the above expression is taken as
   $$a + (b * c),$$
   with * taking *b* and *c* and then + taking *a* and $b*c$.
2) + and - bind more strongly than &&, which in turn is stronger than ||:
   $a \&\& b - c || d$ is taken as $(a \&\& (b - c)) || d$
3) When both operators bind equally strongly,
   the operations are done left to right:
   $a - b - c$ is taken as $(a - b) - c$.

Parentheses may be used as above to force particular groupings.

CONDITIONAL VALUES:

($a >$ b ? v1 : v2)
($a <$ b ? v1 : v2)
($a >= b$? v1 : v2)
($a <= b$? v1 : v2)
($a == b$? v1 : v2)
(a != b ? v1 : v2)

where *a*, *b*, v1 and v2 may be expressions, but *a*, *b* not audio-rate.

In the above conditionals, *a* and *b* are first compared. If the indicated relation is true (*a* greater than *b*, *a* less than *b*, *a* greater than or equal to *b*, *a* less than or equal to *b*, *a* equal to *b*, *a* not equal to *b*), then the conditional expression has the value of v1; if the relation is false, the expression has the value of v2. (For convenience, a sole '=' will function as '=='.)
N.B.: If v1 or v2 are expressions, these will be evaluated *before* the conditional is determined.

In terms of binding strength, all conditional operators (i.e., the relational operators (>, <, etc.), and ? and :) are weaker than the arithmetic and logical operators (+ , - , *, /, && and ||).

Example:

(k1 < p5/2 + p6 ? k1 : p7)

binds the terms p5/2 and p6. It will return the value k1 below this threshold, else the value p7.

EXPRESSIONS:

Expressions may be composed to any depth from the components shown above.

Each sub-expression (part of an expression) is evaluated at its own proper rate. For instance, if the terms within a sub-expression all change at the control rate or slower, the sub-expression will be evaluated only at the control rate; that result might then be used in an audio-rate evaluation.

Examples:

```
k1 + abs(p5/2 + sqrt(k2))
int(p5) + frac(p5) * 100/12
```

The above are legal expressions. They may be placed in unit generator argument positions or be part of an assignment statement (q.v.).

STATEMENT TYPES:

There are nine statement types, each of which provides a heading for the descriptive sections that follow in this document:

assignment statements
orchestra header statements
instrument block statements
program control statements
duration control statements
signal generator statements
signal modifier statements
signal display statements
soundfile access statements

INTERPRETIVE NOTE:

Throughout this document, opcodes are indicated in **boldface** and their argument and result mnemonics, when mentioned in the text, are given in *italics*. Argument names are generally mnemonic (*amp*, *phs*), and the result is denoted the letter *r*. Both are preceded by a type qualifier *i, k, a* or *x* (e.g. *kamp, iphs, ar*). The prefix *i* denotes scalar values valid at note Init time; prefixes *k* or *a* denote control (scalar) and audio (vector) values, modified and referenced continuously throughout performance (i.e. at every control period while the instrument is active). Arguments are *used* at the prefix-listed times; results are *created* at their listed times, then remain available for use as inputs elsewhere. The validity of inputs is defined by the following:

arguments with prefix *i* must be valid at Init time;
arguments with prefix *k* can be either control or Init values (which remain valid);
arguments with prefix *a* must be vector inputs;
arguments with prefix *x* may be either vector or scalar (the compiler will distinguish).

All arguments, unless otherwise stated, can be expressions whose results conform to the above.

Most opcodes (such as **linen** and **oscil**) can be used in more than one mode, which one being determined by the prefix of the result symbol.

ASSIGNMENT STATEMENTS

ir	=	iarg	
kr	=	karg	
ar	=	xarg	
kr	**init**	iarg	
ar	**init**	iarg	
ir	**tival**		
ir	**divz**	ia, ib, isubst	(these not yet implemented)
kr	**divz**	ka, kb, ksubst	
ar	**divz**	xa, xb, ksubst	

= (simple assignment) - Put the value of the expression *iarg (karg, xarg)* into the named result. This provides a means of saving an evaluated result for later use.

init - Put the value of the I-time expression *iarg* into a K- or A-variable, i.e., initialize the result. Note that **init** provides the only case of an Init-time statement being permitted to write into a Perf-time (K- or A-rate) result cell; the statement has no effect at Perf-time.

tival - Put the value of the instrument's internal "tie-in" flag into the named I-variable. Assigns 1 if this note has been 'tied' onto a previously held note (see I Statement); assigns 0 if no tie actually took place. (See also **tigoto**.)

divz - Whenever *b* is not zero, set the result to the value of *a*/*b*; when *b* is zero, set it to the value of *subst* instead.

Example:

```
kcps     =     i2/3 + cpsoct(k2 + octpch(p5))
```

ORCHESTRA HEADER STATEMENTS

sr = n1
kr = n2
ksmps = n3
nchnls = n4

These statements are global value *assignments*, made at the beginning of an orchestra, before any instrument block is defined. Their function is to set certain *reserved symbol variables* that are required for performance. Once set, these reserved symbols can be used in expressions anywhere in the orchestra.

sr= (optional) - set sampling rate to *n1* samples per second per channel. The default value is 10000.

kr= (optional) - set control rate to *n2* samples per second. The default value is 1000.

ksmps= (optional) - set the number of samples in a Control Period. **This value must equal sr/kr**. The default value is 10.

nchnls= (optional) - set number of channels of audio output to *n4*. (1 = mono, 2 = stereo, 4 = quadraphonic.) The default value is 1 (mono).

In addition, any **global variable** can be initialized by an *init-time assignment* anywhere before the first **instr** statement.

All of the above assignments are run as instrument 0 (i-pass only) at the start of real performance.

Example of header assignments:

```
            sr = 10000
            kr = 500
            ksmps = 20

gi1         =          sr / 2.
ga1         init       0
gitranspose =          octpch(.01)
```

INSTRUMENT BLOCK STATEMENTS

```
instr   i, j, ...
.
. <body
.    of
.   instrument>
.
endin
```

These statements delimit an instrument block. They must always occur in pairs.

instr - begin an instrument block defining instruments *i, j, ...*

i, j, ... must be numbers, not expressions. Any positive integer is legal, and in any order, but excessively high numbers are best avoided.

endin - end the current instrument block.

Note:

There may be any number of instrument blocks in an orchestra.

Instruments can be defined in any order (but they will always be both initialized and performed in ascending instrument number order).

Instrument blocks cannot be nested (i.e. one block cannot contain another).

PROGRAM CONTROL STATEMENTS

igoto	label
tigoto	label
kgoto	label
goto	label
if	ia R ib **igoto** label
if	ka R kb **kgoto** label
if	ia R ib **goto** label
timout	istrt, idur, label

where *label* is in the same instrument block and is not an expression, and where *R* is one of the Relational operators (>, <, >=, <=, ==, !=) (and = for convenience, see also under Conditional values).

These statements are used to control the order in which statements in an instrument block are to be executed. I-time and P-time passes can be controlled separately as follows:

igoto - During the I-time pass only, unconditionally transfer control to the statement labeled by *label*.

tigoto - similar to **igoto**, but effective only during an I-time pass at which a new note is being 'tied' onto a previously held note (see I Statement); no-op when a tie has not taken place. Allows an instrument to skip initialization of units according to whether a proposed tie was in fact successful (see also **tival**, **delay**).

kgoto - During the P-time passes only, unconditionally transfer control to the statement labeled by *label*.

goto - (combination of **igoto** and **kgoto**) Transfer control to *label* on every pass.

if...igoto - conditional branch at I-time, depending on the truth value of the logical expression "ia R ib". The branch is taken only if the result is true.

if...kgoto - conditional branch during P-time, depending on the truth value of the logical expression "ka R kb". The branch is taken only if the result is true.

if...goto - combination of the above. Condition tested on every pass.

timout - conditional branch during P-time, depending on elapsed note time. *istrt* and *idur* specify time in seconds. The branch to *label* will become effective at time *istrt*, and will remain so for just *idur* seconds. Note that **timout** can be reinitialized for multiple activation within a single note (see example next page).

Example:

```
if    k3 > p5+10 kgoto next
```

reinit	label
rigoto	label
rireturn	

These statements permit an instrument to reinitialize itself during performance.

reinit - whenever this statement is encountered during a P-time pass, performance is temporarily suspended while a special Initialization pass, beginning at *label* and continuing to **rireturn** or **endin**, is executed. Performance will then be resumed from where it left off.

rigoto - similar to **igoto**, but effective only during a **reinit** pass (i.e., No-op at standard I-time). This statement is useful for bypassing units that are not to be reinitialized.

rireturn - terminates a **reinit** pass (i.e., No-op at standard I-time). This statement, or an **endin**, will cause normal performance to be resumed.

Example:

The following statements will generate an exponential control signal whose value moves from 440 to 880 exactly ten times over the duration p3.

```
reset:          timout    0, p3/10, contin      ;after p3/10 seconds,
                reinit    reset                 ;  reinit both timout
contin:   k1    expon     440, p3/10, 880       ;  and expon
                rireturn                        ;  then resume perf
```

DURATION CONTROL STATEMENTS

ihold
turnoff

These statements permit the current note to modify its own duration.

ihold - this I-time statement causes a finite-duration note to become a 'held' note. It thus has the same effect as a negative p3 (see Score I-statement), except that p3 here remains positive and the instrument reclassifies itself to being held indefinitely. The note can be turned off explicitly with **turnoff**, or its space taken over by another note of the same instrument number (i.e. it is tied into that note). Effective at I-time only; no-op during a **reinit** pass.

turnoff - this P-time statement enables an instrument to turn itself off. Whether of finite duration or 'held', the note currently being performed by this instrument is immediately removed from the active note list. No other notes are affected.

Example:

The following statements will cause a note to terminate when a control signal passes a certain threshold (here the Nyquist frequency).

```
        k1      expon   440, p3/10, 880         ;begin gliss and continue
                if      k1 < sr/2 kgoto contin  ;  until Nyquist detected
                turnoff                         ;  then quit
contin: a1      oscil   a1, k1, 1
```

SIGNAL GENERATORS

kr	**line**	ia, idur1, ib
ar	**line**	ia, idur1, ib
kr	**expon**	ia, idur1, ib
ar	**expon**	ia, idur1, ib
kr	**linseg**	ia, idur1, ib[, idur2, ic[...]]
ar	**linseg**	ia, idur1, ib[, idur2, ic[...]]
kr	**expseg**	ia, idur1, ib[, idur2, ic[...]]
ar	**expseg**	ia, idur1, ib[, idur2, ic[...]]

Output values *kr* or *ar* trace a straight line (exponential curve) or a series of line segments (exponential segments) between specified points.

INITIALIZATION

ia - starting value. Zero is illegal for exponentials.

ib, ic, etc. - value after *dur1* seconds, etc. For exponentials, must be non-zero and must agree in sign with *ia*.

idur1 - duration in seconds of first segment. A zero or negative value will cause all initialization to be skipped.

idur2, *idur3*, etc. - duration in seconds of subsequent segments. A zero or negative value will terminate the initialization process with the preceding point, permitting the last-defined line or curve to be continued indefinitely in performance. The default is zero.

PERFORMANCE

These units generate control or audio signals whose values can pass through 2 or more specified points. The sum of *dur* values may or may not equal the instrument's performance time: a shorter performance will truncate the specified pattern, while a longer one will cause the last-defined segment to continue on in the same direction.

Example:

```
k2        expseg     440, p3/2, 880, p3/2, 440
```

This statement creates a control signal which moves exponentially from 440 to 880 and back, over the duration p3.

kr	**phasor**	kcps[, iphs]
ar	**phasor**	xcps[, iphs]

Produce a normalized moving phase value.

INITIALIZATION

iphs (optional) - initial phase, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is zero.

PERFORMANCE

An internal phase is successively accumulated in accordance with the cps frequency to produce a moving phase value, normalized to lie in the range 0.<= phs < 1.

When used as the index to a **table** unit, this phase (multiplied by the desired function table length) will cause it to behave like an oscillator.

Note that **phasor** is a special kind of integrator, accumulating phase increments that represent frequency settings.

Example:

```
k1      phasor   1                    ;cycle once per second
kpch    table    k1*12, 1             ;through 12-note pch table
a1      oscil    p4, cpspch(kpch), 2  ;with continuous sound
```

ir	**table**	indx, ifn[, ixmode][, ixoff][, iwrap]
ir	**tablei**	indx, ifn[, ixmode][, ixoff][, iwrap]
kr	**table**	kndx, ifn[, ixmode][, ixoff][, iwrap]
kr	**tablei**	kndx, ifn[, ixmode][, ixoff][, iwrap]
ar	**table**	andx, ifn[, ixmode][, ixoff][, iwrap]
ar	**tablei**	andx, ifn[, ixmode][, ixoff][, iwrap]
kr	**oscil1**	idel, kamp, idur, ifn
kr	**oscil1i**	idel, kamp, idur, ifn

Table values are accessed by direct indexing or by incremental sampling.

INITIALIZATION

ifn - function table number. **tablei, oscil1i** require the extended guard point.

ixmode (optional) - ndx data mode. 0 = raw ndx, 1 = normalized (0 to 1). The default value is 0.

ixoff (optional) - amount by which ndx is to be offset. For a table with origin at center, use tablesize/2 (raw) or .5 (normalized). The default value is 0.

iwrap (optional) - wraparound ndx flag. 0 = nowrap (ndx<0 treated as ndx=0; ndx>tablesize sticks at ndx=size), 1 = wraparound. The default value is 0.

idel - delay in seconds before **oscil1** incremental sampling begins.

idur - duration in seconds to sample through the **oscil1** table just once. A zero or negative value will cause all initialization to be skipped.

PERFORMANCE

table invokes table lookup on behalf of init, control or audio indices. These indices can be raw entry numbers (0,1,2...siz-1) or scaled values (0 to 1-e). Indices are first modified by the offset value then checked for range before table lookup (see *iwrap*). If ndx is likely to be full scale, or if interpolation is being used, the table should have an extended guard point. **table** indexed by a periodic phasor (see **phsor**) will simulate an oscillator.

oscil1 accesses values by sampling once through the function table at a rate determined by *idur*. For the first *idel* seconds, the point of scan will reside at the first location of the table; it will then begin moving through the table at a constant rate, reaching the end in another *idur* seconds; from that time on (i.e. after *idel* + *idur* seconds) it will remain pointing at the last location. Each value obtained from sampling is then multiplied by an amplitude factor *kamp* before being written into the result.

tablei and **oscil1i** are interpolating units in which the fractional part of ndx is used to interpolate between adjacent table entries. The smoothness gained by interpolation is at some small cost in execution time (see also **oscili**, etc.), but the interpolating and non-interpolating units are otherwise interchangable. Note that when **tablei** uses a periodic index whose modulo *n* is less than the power of 2 table length, the interpolation process requires that there be an (*n*+1)th table value that is a repeat of the 1st (see F statement in Score).

kr	**oscil**	kamp, kcps, ifn[, iphs]
kr	**oscili**	kamp, kcps, ifn[, iphs]
ar	**oscil**	xamp, xcps, ifn[, iphs]
ar	**oscili**	xamp, xcps, ifn[, iphs]
ar	**foscil**	xamp, kcps, kcar, kmod, kndx, ifn[, iphs]
ar	**foscili**	xamp, kcps, kcar, kmod, kndx, ifn[, iphs]

Table *ifn* is incrementally sampled modulo the table length and the value obtained is multiplied by *amp*.

INITIALIZATION

ifn - function table number. Requires a wrap-around guard point.

iphs (optional) - initial phase of sampling, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is 0.

PERFORMANCE

The **oscil** units output periodic control (or audio) signals consisting of the value of *kamp* (xamp) times the value returned from control rate (audio rate) sampling of a stored function table. The internal phase is simultaneously advanced in accordance with the *cps* input value. While the amplitude and frequency inputs to the K-rate **oscil**s are scalar only, the corresponding inputs to the audio-rate **oscil**s may each be either scalar or vector, thus permitting amplitude and frequency modulation at either sub-audio or audio frequencies.

foscil is a composite unit that effectively banks two **oscil**s in the familiar Chowning FM setup, wherein the audio-rate output of one generator is used to modulate the frequency input of another (the "carrier"). Effective carrier frequency = *kcps*kcar,* and modulating frequency = *kcps*kmod.* For integral values of *kcar* and *kmod,* the perceived fundamental will be the minimum positive value of *kcps * (kcar - n*kmod), n = 0,1,2,...* The input *kndx* is the index of modulation (usually time-varying and ranging 0 to 4 or so) which determines the spread of acoustic energy over the partial positions given by n = 0,1,2,..,etc. *ifn* should point to a stored sine wave.

oscili and **foscili** differ from **oscil** and **foscil** respectively in that the standard procedure of using a truncated phase as a sampling index is here replaced by a process that interpolates between two successive lookups. Interpolating generators will produce a noticeably cleaner output signal, but they may take as much as twice as long to run. Adequate accuracy can also be gained without the time cost of interpolation by using large stored function tables of 2K, 4K or 8K points if the space is available.

Example:

```
k1      oscil   10, 5, 1              ;5 cps vibrato
a1      oscil   5000, 440+k1, 1       ;around A440 +-10 cps
```

ar	**buzz**	xamp, xcps, knh, ifn[, iphs]
ar	**gbuzz**	xamp, xcps, knh, klh, kr, ifn[, iphs]

Output is a set of harmonically related cosine partials.

INITIALIZATION

ifn - table number of a stored function containing (for **buzz**) a sine wave, or (for **gbuzz**) a cosine wave. In either case a large table of at least 8192 points is recommended.

iphs (optional) - initial phase of the fundamental frequency, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is zero.

PERFORMANCE

These units generate an additive set of harmonically related cosine partials of fundamental frequency *xcps*, and whose amplitudes are scaled so their summation peak equals *xamp*. The selection and strength of partials is determined by the following control parameters:

knh - total number of harmonics requested. Must be positive.

klh - lowest harmonic present. Can be positive, zero or negative. In **gbuzz** the set of partials can begin at any partial number and proceeds upwards; if *klh* is negative, all partials below zero will reflect as positive partials without phase change (since cosine is an even function), and will add constructively to any positive partials in the set.

kr - specifies the multiplier in the series of amplitude coefficients. This is a power series: if the *klh*th partial has a strength coefficient of A, the (*klh*+n)th partial will have a coefficient of A * (*kr***n), i.e. strength values trace an exponential curve. *kr* may be positive, zero or negative, and is not restricted to integers.

buzz and **gbuzz** are useful as complex sound sources in subtractive synthesis. **buzz** is a special case of the more general **gbuzz** in which *klh* = *kr* = 1; it thus produces a set of *knh* equal-strength harmonic partials, beginning with the fundamental. (This is a band-limited pulse train; if the partials extend to the Nyquist, i.e. *knh* = int(sr/2/fundamental freq.), the result is a real pulse train of amplitude *xamp*.) Although both *knh* and *klh* may be varied during performance, their internal values are necessarily integer and may cause "pops" due to discontinuities in the output; *kr*, however, can be varied during performance to good effect. Both **buzz** and **gbuzz** can be amplitude- and/or frequency-modulated by either control or audio signals.

N.B. These two units have their analogs in GEN11, in which the same set of cosines can be stored in a function table for sampling by an oscillator. Although computationally more efficient, the stored pulse train has a fixed spectral content, not a time-varying one as above.

ar	**adsyn**	kamod, kfmod, ifilno
ar	**pvoc**	ktimpnt, kfmod, ispecwp, ifilno

Output is an additive set of individually controlled sinusoids, using either an oscillator bank or phase vocoder resynthesis.

INITIALIZATION

ispecwp - if non-zero, attempts to preserve the spectral envelope while its frequency content is varied by *kfmod*.

ifilno - control-file suffix (m) of a file named *adsyn.m* or *pvoc.m*, stemming from analysis of an audio signal. **adsyn** control contains breakpoint amplitude- and frequency-envelope values organized for oscillator resynthesis, while **pvoc** control contains similar data organized for fft resynthesis. Note that memory usage depends on the size of the control files involved, which are stored internally during computation.

PERFORMANCE

adsyn synthesizes complex time-varying timbres through the method of additive synthesis. Any number of sinusoids, each individually controlled in frequency and amplitude, can be summed by high-speed arithmetic to produce a high-fidelity result.

Component sinusoids are described by a control file (format described elsewhere) that specifies both frequency and amplitude tracks in breakpoint fashion (to the millisecond). Through interpolation, the instantaneous frequency and amplitude values are used by an internal fixed-point oscillator that adds each active partial into an accumulated output signal. There are no limits on the number of contributing partials, or on their behavior over time. Any sound complex that can be described in terms of the behavior of sinusoids can be synthesized by **adsyn** alone.

In addition, the sound described by the control file can be modified during actual synthesis. The signals *kamod, kfmod,* will modify the amplitude and frequency, respectively, of each contributing partial. Note that these are multiplying factors, with *kfmod* being applied to the cps frequency. Thus the values .7,1.5 will give rise to a softer sound, a perfect fifth higher; the values 1,1 will leave the sound unmodified. Each of these inputs can be a control signal.

pvoc implements signal reconstruction using an fft-based phase vocoder. The control data stems from a precomputed analysis file with a known frame rate. The passage of time through this file is specified by *ktimpnt,* which represents the time in seconds. *ktimpnt* must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file. *kfmod* is a control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and .5 down an octave.

This implementation of **pvoc** was written by Dan Ellis. It is based in part on the system of Mark Dolson, but the pre-analysis concept is new.

ar **fof** xamp, xfund, xforma, xformb, koct, ktex, kband, kdebat, katt,
iolaps, ifna, ifnb, idur[, iphs][, icor]

Audio output is a succession of FOF impulses. With *xfund* above c. 30 Hz these produce a formant (set of harmonically related partials whose spectral envelope can be controlled by k-input parameters). With lower fundamentals this generator provides a special form of granular synthesis.

INITIALIZATION

iolaps - the maximum number of overlapping FOFs in the note event. May be calculated as the maximum value of *xfund * (kdebat + katt)*, rounded up to an integer. Too small a value will result in a warning message; the program will continue to run with possible distortion. An excessively large value will waste computation time.

ifna, ifnb - table numbers of stored functions. Normally both will reference the same sine wave table. As interpolation is not used in the table lookup a table of at least 8192 is recommended.

idur - normally set to "p3" (the note length). A FOF impulse cannot be created unless it can complete its course before time *idur*.

iphs (optional) - initial phase (of the fundamental) expressed as a fraction of a cycle (0 to 1). The default value is 0.

icor (optional) - automatic correction of the spectrum, allowing for *ktex* and *kband*. Normalises the internal amplitude, enabling *xamp* to specify the output amp relative to other formant regions (i.e. FOF unit-generators). Accurate only if all generators concerned have the same fundamental frequency. This is designed to work within the normal range of input values for vocal imitation; its use in other situations may produce strange amplitudes. 1 = on, 0 = off; the default is 1.

PERFORMANCE

xamp - amplitude. This also varies with the number of FOFs overlapping (and with *ktex* and *kband* if *icor* is not on). The experienced user will learn to adjust *xamp* accordingly.

xfund - the fundamental frequency (in Hertz).

xforma, xformb - the formant frequency. Changes to *xforma* only take effect at the start of a new FOF; each FOF impulse has a fixed formant frequency from this input. *xformb* allows continuous change. The actual formant frequency is the sum of these two inputs.

ktex - attack time in seconds of the FOF impulse. The skirtwidth of the formant region (-40dB) varies in inversely to this. A common value for vocal imitation is .003 .

kband - bandwidth of the formant region at -6dB in Hz.

kdebat, katt - the start time (relative to the start of the FOF) and length (in seconds) of the sinusoidal rounding at the end of each FOF impulse. Typical values are .01 and .007 .

The **fof** generator is written by Michael Clarke (Huddersfield Polytechnic, England) based on the CHANT program from IRCAM (Xavier Rodet et al.). Each **fof** generator produces a single formant region, and the output of five or more of these can be summed to produce a rich vocal imitation. FOF synthesis is a special form of granular synthesis and this module has been specifically designed to facilitate transformations between vocal (and other) imitation and granular textures.

ar **pluck** kamp, kcps, icps, ifn, imeth [, iparm1, iparm2]

Audio output is a naturally decaying plucked string or drum sound based on the Karplus-Strong algorithms.

INITIALIZATION

icps - intended pitch value in cps, used to set up a buffer of 1 cycle of audio samples which will be smoothed over time by a chosen decay method. *icps* normally anticipates the value of *kcps*, but may be set artificially high or low to influence the size of the sample buffer.

ifn - table number of a stored function used to initialize the cyclic decay buffer. If *ifn* = 0, a random sequence will be used instead.

imeth - method of natural decay. There are six, some of which use parameters values that follow.

1 - simple averaging. A simple smoothing process, uninfluenced by parameter values.
2 - stretched averaging. As above, with smoothing time stretched by a factor of *iparm1* (>= 1).
3 - simple drum. The range from pitch to noise is controlled by a 'roughness factor' in *iparm1* (0 to 1). Zero gives the plucked string effect, while 1 reverses the polarity of every sample (octave down, odd harmonics). The setting .5 gives an optimum snare drum.
4 - stretched drum. Combines both roughness and stretch factors. *iparm1* is roughness (0 to 1), and *iparm2* the stretch factor (>= 1).
5 - weighted averaging. As method 1, with *iparm1* weighting the current sample (the status quo) and *iparm2* weighting the previous adjacent one. *iparm1* + *iparm2* must be <= 1.
6 - 1st order recursive filter, with coefs .5. Unaffected by parameter values.

iparm1, *iparm2* (optional) - parameter values for use by the smoothing algorithms (above). The default values are both 0.

PERFORMANCE

An internal audio buffer, filled at I-time according to *ifn*, is continually resampled with periodicity *kcps* and the resulting output is multiplied by *kamp*. Parallel with the sampling, the buffer is smoothed to simulate the effect of natural decay.

Plucked strings (1,2,5,6) are best realized by starting with a random noise source, which is rich in initial harmonics. Drum sounds (methods 3, 4) work best with a flat source (wide pulse), which produces a deep noise attack and sharp decay.

The original Karplus-Strong algorithm used a fixed number of samples per cycle, which caused serious quantization of the pitches available and their intonation. This implementation resamples a buffer at the exact pitch given by *kcps*, which can be varied for vibrato and glissando effects. For low values of the orch sampling rate (e.g. *sr* = 10000), high frequencies will store only very few samples (*sr*/*icps*). Since this may cause noticeable noise in the resampling process, the internal buffer has a minimum size of 64 samples. This can be further enlarged by setting *icps* to some artificially lower pitch.

kr	**rand**	xamp[, iseed]
kr	**randh**	kamp, kcps[, iseed]
kr	**randi**	kamp, kcps[, iseed]
ar	**rand**	xamp[, iseed]
ar	**randh**	xamp, xcps[, iseed]
ar	**randi**	xamp, xcps[, iseed]

Output is a controlled random number series between +*amp* and -*amp*.

INITIALIZATION

iseed (optional) - seed value for the recursive psuedo-random formula. A value between 0 and +1 will produce an initial output of *kamp* * *iseed*. A negative value will cause seed re-initialization to be skipped. The default seed value is .5 .

PERFORMANCE

The internal psuedo-random formula produces values which are uniformly distributed over the range *kamp* to -*kamp*. **rand** will thus generate uniform white noise with an R.M.S value of *kamp*/*root 2*.

The remaining units produce band-limited noise: the cps parameters permit the user to specify that new random numbers are to be generated at a rate less than the sampling or control frequencies. **randh** will hold each new number for the period of the specified cycle; **randi** will produce straight-line interpolation between each new number and the next.

Example:

```
i1      =       octpch(p5)              ;center pitch, to be modified
k1      randh   1,10                    ;10 times/sec by random dis-
                                        ;placements up to 1 octave
a1      oscil   5000, cpsoct(i1+k1), 1
```

SIGNAL MODIFIERS

kr	**linen**	kamp, irise, idur, idec
ar	**linen**	xamp, irise, idur, idec
kr	**envlpx**	kamp, irise, idur, idec, ifn, iatss, iatdec[, ixmod]
ar	**envlpx**	xamp, irise, idur, idec, ifn, iatss, iatdec[, ixmod]

linen - apply a straight line rise and decay pattern to an input amp signal.
envlpx - apply an envelope consisting of 3 segments: 1) stored function rise shape, 2) modified exponential "pseudo steady state", 3) true exponential decay

INITIALIZATION

irise - rise time in seconds. A zero or negative value signifies no rise modification.

idur - overall duration in seconds. A zero or negative value will cause all initialization to be skipped.

idec - decay time in seconds. A zero value indicates no decay modification. A value greater than *idur* will cause a truncated decay pattern.

ifn - function table number of stored rise shape with extended guard point.

iatss - attenuation factor, by which the last value of **envlpx** rise pattern will become modified during the note's pseudo "steady state." A factor >1 will cause an exponential growth, and a factor <1 an exponential decay. The value 1 will maintain a true steady state at the last rise value. Note that this attenuation is not by fixed rate (as in a piano), but is sensitive to a note's duration. However, if *iatss* is negative (or if "steady state" < 4 k-periods) a fixed attenuation rate of *abs(iatss)* per second will be used. 0 is illegal.

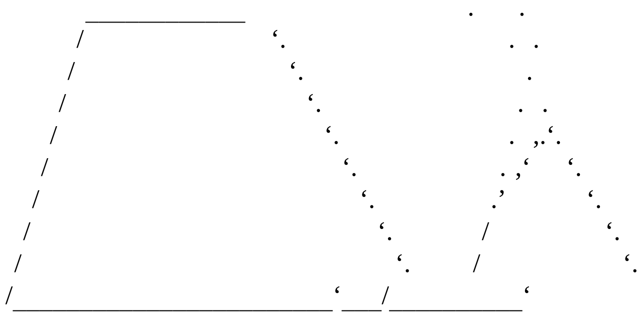
iatdec - attenuation factor by which the closing "steady state" value is to be reduced exponentially over the decay period. This value must be positive and will normally be of the order of .01 . A large or an excessively small value is apt to produce a cutoff which is audible. A zero or neg value is illegal.

ixmod (optional, between +-.9 or so) - exponential curve modifier, influencing the "steepness" of the exponential trajectory during the "steady state." Values less than zero will cause an accelerated growth or decay towards the target (e.g. *subito piano*). Values greater than zero will cause a retarded growth or decay. The default value is zero (unmodified exponential).

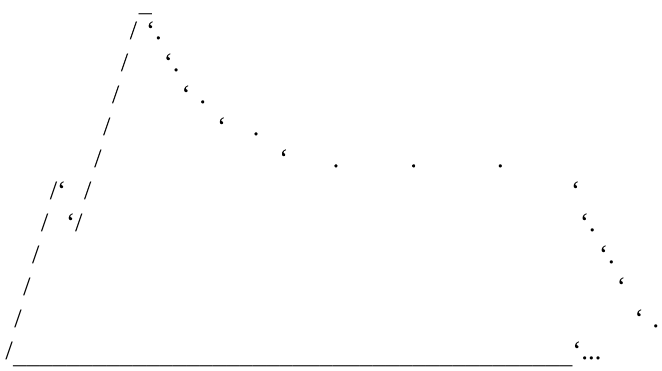
PERFORMANCE

Rise-time modifications are applied for the first *irise* seconds, and decay-time modifications from time *idur - idec*. If these two modification periods are separated in time there will be a real "steady state" period during which *amp* will be unmodified (**linen**) or modified by the first exponential pattern (**envlpx**). For **linen**, if the rise and decay periods overlap then both modifications will be in effect for that time; for **envlpx** an overlap will simply cause a truncated decay pattern. If the overall duration *idur* is exceeded in performance, the final decay pattern will continue on in the same direction, going negative for **linen** but tending asymptotically to zero in the case of **envlpx**.

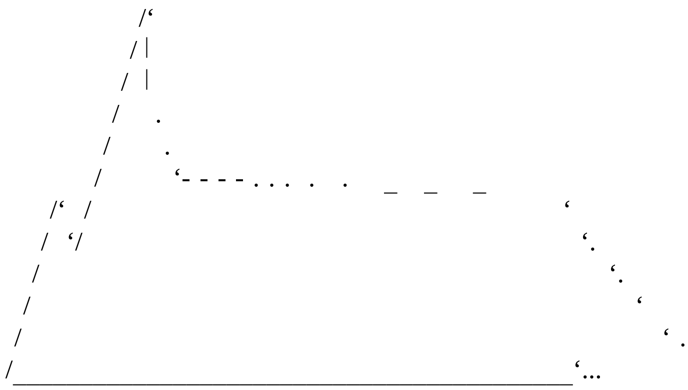
Examples:



Ex 1. **linen** a) normal, b) with overlapping rise and decay



Ex 2. **envlpx** with *iatss* = .5 and *ixmod* = 0



Ex 3. **envlpx** with *iatss* = .5 and *ixmod* = -.9

kr	**port**	ksig, ihtim[, isig]
ar	**tone**	asig, khp[, istor]
ar	**atone**	asig, khp[, istor]
ar	**reson**	asig, kcf, kbw[, iscl, istor]
ar	**areson**	asig, kcf, kbw[, iscl, istor]

A control or audio signal is modified by a low- or band-pass recursive filter with variable frequency response.

INITIALIZATION

isig - initial (i.e. previous) value for internal feedback. The default value is 0.

istor - initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

iscl - coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than *kcf* are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. (This intended equalization of input and ouput power assumes all frequencies are physically present; hence it is most applicable to white noise.) A zero value signifies no scaling of the signal, leaving that to some later adjustment (e.g. see **balance**). The default value is 0.

PERFORMANCE

port applies portamento to a step-valued control signal. At each new step value, *ksig* is low-pass filtered to move towards that value at a rate determined by *ihtim. ihtim* is the "half-time" of the function (in seconds), during which the curve will traverse half the distance towards the new value, then half as much again, etc., theoretically never reaching its asymptote.

tone implements a first-order recursive low-pass filter in which the variable *khp* (in c.p.s.) determines the response curve's half-power point. Half power is defined as peak power / root 2.

reson is a second-order filter in which *kcf* controls the center frequency, or cps position of the peak response, and *kbw* controls its bandwidth (the cps difference between the upper and lower half-power points).

atone, areson are filters whose transfer functions are the complements of **tone** and **reson**. **atone** is thus a form of high-pass filter and **areson** a notch filter whose transfer functions represent the "filtered out" apsects of their complements. Note, however, that power scaling is not normalized in **atone, areson**, but remains the true complement of the corresponding unit. Thus an audio signal, filtered by parallel matching **reson** and **areson** units, would under addition simply reconstruct the original spectrum. This property is particularly useful for controlled mixing of different sources (e.g., see **lpreson**).

Complex response curves such as those with multiple peaks can be obtained by using a bank of suitable filters in series. (The resultant response is the product of the component responses.) In such cases, the combined attenuation may result in a serious loss of signal power, but this can be regained by the use of **balance**.

krmsr,krmso,kerr,kcps	**lpread**	ktimpnt, ifilno[, inpoles][, ifrmrate]
ar	**lpreson**	asig
ar	**lpfreson**	asig, kfrqratio

These units, used as a read/reson pair, use a control file of time-varying filter coefficients to dynamically modify the spectrum of an audio signal.

INITIALIZATION

ifilno - control-file suffix (m) referring to a file named 'lp.m' containing frames of reflection coefficients and four special parameter values derived from n-pole linear predictive spectral analysis of a source file. A negative value will cause file opening and initialization to be skipped.

inpoles, ifrmrate (optional) - number of poles, and frame rate per second in the lpc analysis. These arguments are required only when the control file does not have a header; they are ignored when a header is detected. The default value for both is zero.

PERFORMANCE

lpread accesses a control file of time-ordered information frames, each containing n-pole filter coefficients derived from linear predictive analysis of a source signal at fixed time intervals (e.g. 1/100 of a second), plus four parameter values:

krmsr - root-mean-square (rms) of the residual of analysis,
krmso - rms of the original signal,
kerr - the normalized error signal,
kcps - pitch in cps.

lpread gets its values from the control file according to the input value *ktimpnt* (in seconds). If *ktimpnt* proceeds at the analysis rate, time-normal synthesis will result; proceeding at a faster, slower, or variable rate will result in time-warped synthesis. At each K-period, **lpread** automatically interpolates between adjacent frames to more accurately determine the parameter values (presented as output) and the filter coefficient settings (passed internally to a subsequent **lpreson**).

The error signal *kerr* (between 0 and 1) derived during predictive analysis reflects the deterministic/random nature of the analyzed source. This will emerge low for pitched (periodic) material and higher for noisy material. The transition from voiced to unvoiced speech, for example, produces an error signal value of about .3. During synthesis, the error signal value can be used to determine the nature of the **lpreson** driving function: for example, by arbitrating between pitched and non-pitched input, or even by determining a mix of the two. In normal speech resynthesis, the pitched input to **lpreson** is a wideband periodic signal or pulse train derived from a unit such as **buzz**, and the non-pitched source is usually derived from **rand**. However, any audio signal can be used as the driving function, the only assumption of the analysis being that it has a flat response.

lpfreson is a formant shifted **lpreson**, in which *kfrqratio* is the (cps) ratio of shifted to original formant positions. This permits synthesis in which the source object changes its apparent acoustic size. **lpfreson** with *kfrqratio* = 1 is equivalent to **lpreson**.

Generally, **lpreson** provides a means whereby the time-varying content and spectral shaping of a composite audio signal can be controlled by the dynamic spectral content of another. There can be any number of **lpread/lpreson** (or **lpfreson**) pairs in an instrument or in an orchestra; they can read from the same or different control files independently.

kr	**rms**	asig[, ihp, istor]
ar	**gain**	asig, krms[, ihp, istor]
ar	**balance**	asig, acomp[, ihp, istor]

The **rms** power of *asig* can be interrogated, set, or adjusted to match that of a comparator signal.

INITIALIZATION

ihp (optional) - half-power point (in cps) of a special internal low-pass filter. The default value is 10.

istor (optional) - initial disposition of internal data space (see **reson**). The default value is 0.

PERFORMANCE

rms output values kr will trace the **rms** value of the audio input *asig*. This unit is not a signal modifier, but functions rather as a signal power-guage.

gain provides an amplitude modification of *asig* so that the output *ar* has **rms** power equal to *krms*. **rms** and **gain** used together (and given matching ihp values) will provide the same effect as **balance**.

balance outputs a version of *asig*, amplitude-modified so that its **rms** power is equal to that of a comparator signal *acomp*. Thus a signal that has suffered loss of power (eg., in passing through a filter bank) can be restored by matching it with, for instance, its own source. It should be noted that **gain** and **balance** provide amplitude modification only - output signals are not altered in any other respect.

Example:

```
asrc    buzz     10000, 440, sr/440, 1    ;band-limited pulse train
a1      reson    asrc, 1000, 100          ;sent through
a2      reson    a1, 3000, 500            ;2 filters
afin    balance  a2, asrc                 ;then balanced with source
```

```
kr    downsamp   asig[, iwlen]
ar    upsamp     ksig
ar    interp     ksig[, istor]
kr    integ      ksig[, istor]
ar    integ      asig[, istor]
kr    diff       ksig[, istor]
ar    diff       asig[, istor]
kr    samphold   xsig, kgate[, ival, ivstor]
ar    samphold   asig, xgate[, ival, ivstor]
```

Modify a signal by up- or down-sampling, integration, and differentiation.

INITIALIZATION

iwlen (optional) - window length in samples over which the audio signal is averaged to determine a downsampled value. Maximum length is *ksmps*; 0 and 1 imply no window averaging. The default value is 0.

istor (optional) - initial disposition of internal save space (see **reson**). The default value is 0.

ival, *ivstor* (optional) - controls initial disposition of internal save space. If *ivstor* is zero the internal "hold" value is set to *ival*; else it retains its previous value. Defaults are 0, 0 (i.e. init to zero).

PERFORMANCE

downsamp converts an *audio* signal to a *control* signal by downsampling. It produces one kval for each audio control period. The optional window invokes a simple averaging process to supress foldover.

upsamp, interp convert a *control* signal to an *audio* signal. The first does it by simple repetition of the kval, the second by linear interpolation between successive kvals. **upsamp** is a slightly more efficient form of the assignment 'asig = ksig'.

integ, diff perform *integration* and *differentiation* on an input control signal or audio signal. Each is the converse of the other, and applying both will reconstruct the original signal. Since these units are special cases of low-pass and high-pass filters, they produce a scaled (and phase shifted) output that is frequency-dependent. Thus **diff** of a sine produces a cosine, with amplitude *2*sin(pi*cps/sr)* that of the original (for each component partial); **integ** will inversely affect the magnitudes of its component inputs. With this understanding, these units can provide useful signal modification.

samphold performs a sample-and-hold operation on its input according to the value of *gate*. If *gate > 0*, the input samples are passed to the output; if *gate <= 0*, the last output value is repeated. The controlling *gate* can be a constant, a control signal, or an audio signal.

Example:

```
asrc    buzz      10000, 440, 20, 1    ;band-limited pulse train
adif    diff      asrc                 ;emphasize the highs
anew    balance   adif, asrc           ;   but retain the power
agate   reson     asrc, 0, 440         ;use a lowpass of the original
asamp   samphold  anew, agate          ;   to gate the new audiosig
aout    tone      asamp, 100           ;smooth out the rough edges
```

ar	**delayr**	idlt[, istor]
	delayw	asig
ar	**delay**	asig, idlt[, istor]
ar	**delay1**	asig[, istor]

A signal can be read from or written into a delay path, or it can be automatically delayed by some time interval.

INITIALIZATION

idlt - requested delay time in seconds. This can be as large as available memory will permit. The space required for *n* seconds of delay is 4*n* * **sr** bytes. It is allocated at the time the instrument is first initialized, and returned to the pool at the end of a score section.

istor (optional) - initial disposition of delay-loop data space (see **reson**). The default value is 0.

PERFORMANCE

delayr reads from an automatically established digital delay line, in which the signal retrieved has been resident for *idlt* seconds. This unit must be paired with and precede an accompanying **delayw** unit. Any other **Csound** statements can intervene.

delayw writes *asig* into the delay area established by the preceding **delayr** unit. Viewed as a pair, these two units permit the formation of modified feedback loops, etc. However, there is a lower bound on the value of *idlt*, which must be at least 1 control period (or 1/**kr**).

delay is a composite of the above two units, both reading from and writing into its own storage area. It can thus accomplish signal time-shift, although modified feedback is not possible. There is no minimum delay period.

delay1 is a special form of **delay** that serves to delay the audio signal *asig* by just one sample. It is thus functionally equivalent to "**delay** asig,1/srate" but is more efficient in both time and space. This unit is particularly useful in the fabrication of generalized non-recursive filters.

Example:

```
        tigoto  contin          ;except on a tie,
a2      delay   a1, .05, 0      ;begin 50 msec clean delay of sig
contin:
```

ar	**deltap**	kdlt
ar	**deltapi**	xdlt

Tap a delay line at variable offset times.

PERFORMANCE

These units can tap into a **delayr/delayw** pair, extracting delayed audio from the *idlt* seconds of stored sound. There can be any number of **deltap** and/or **deltapi** units between a read/write pair. Each receives an audio tap with no change of original amplitude.

deltap extracts sound by reading the stored samples directly; **deltapi** extracts sound by *interpolated readout*. By interpolating between adjacent stored samples **deltapi** represents a particular delay time with more accuracy, but it will take about twice as long to run.

The arguments *kdlt, xdlt* specify the tapped delay time in seconds. Each can range from 1 Control Period to the full delay time of the read/write pair; however, since there is no internal check for adherence to this range, the user is wholly responsible. Each argument can be a constant, a variable, or a time-varying signal; the *xdlt* argument in **deltapi** implies that an audio-varying delay is permitted there.

These units can provide multiple delay taps for arbitrary delay path and feedback networks. They can deliver either constant-time or time-varying taps, and are useful for building chorus effects, harmonizers, and doppler shifts. Constant-time delay taps (and some slowly changing ones) do not need interpolated readout; they are well served by **deltap**. Medium-paced or fast varying *dlt's*, however, will need the extra services of **deltapi**.

N.B. K-rate delay times are not internally interpolated, but rather lay down stepped time-shifts of audio samples; this will be found quite adequate for slowly changing tap times. For medium to fast-paced changes, however, one should provide a higher resolution audio-rate timeshift as input.

Example:

```
asource  buzz     1, 440, 20, 1
atime    linseg   1, p3/2, .01, p3/2, 1  ;trace a distance in secs
ampfac   =        1/atime/atime          ; and calc an amp factor
adump    delayr   1                      ;set maximum distance
amove    deltapi  atime                  ;move sound source past
         delayw   asource                ; the listener
         out      amove * ampfac
```

```
ar      comb     asig, krvt, ilpt[, istor]
ar      alpass   asig, krvt, ilpt[, istor]
ar      reverb   asig, krvt[, istor]
```

An input signal is reverberated for *krvt* seconds with "colored" (**comb**), flat (**alpass**), or "natural room" (**reverb**) frequency response.

INITIALIZATION

ilpt - loop time in seconds, which determines the "echo density" of the reverberation. This in turn characterizes the "color" of the **comb** filter whose frequency response curve will contain *ilpt* * *sr/2* peaks spaced evenly between 0 and *sr/2* (the Nyquist frequency). Loop time can be as large as available memory will permit. The space required for an *n* second loop is 4*n* * *sr* bytes. **comb** and **alpass** delay space is allocated and returned as in **delay**.

istor (optional) - initial disposition of delay-loop data space (cf. **reson**). The default value is 0.

PERFORMANCE

These filters reiterate input with an echo density determined by loop time *ilpt*. The attenuation rate is independent and is determined by *krvt*, the reverberation time (defined as the time in seconds for a signal to decay to 1/1000, or 60 db down from its original amplitude). Output from a **comb** filter will appear only after *ilpt* seconds; **alpass** output will begin to appear immediately.

A **reverb** unit is composed of four **comb** filters in parallel followed by two **alpass** units in series. Looptimes are set for optimal "natural room response." Core storage requirements for this unit are proportional only to the sampling rate, each unit requiring approximately 3K words for every 10KC. It is usually expedient to mix several signals together before reverberation. Since **reverb** output will begin to appear only after 1/20 second or so of delay, and often with less than three-fourths of the original power, it is common to output both the source and the reverberated signal together.

Example:

```
a1      oscili   k1, a1, 1          ;create two signals
a2      oscili   k2, a2, 2
a3      reverb   a1+a2, 1.5;mix, then reverberate
        outs     a1+a3, a2+a3       ;send 1 source + both reverbs
                                    ;through each speaker
```

OPERATIONS USING SPECTRAL DATA-TYPES

dsig	**octdown**	asig, ioct s, isamps[, idisprd]
wsig	**noctdft**	dsig, iprd, ifrqs, iq[, ihann, idbout, idsines]
wsig	**specscal**	wsigin, ifscale, ifthresh
wsig	**specaddm**	wsig1, wsig2[, imul2]
wsig	**specdiff**	wsigin
wsig	**specfilt**	wsigin, ifhtim
	specdisp	wsig, iprd[, iwtflg]
ksum	**specsum**	wsig[, interp]

These units generate and process non-standard audio data types, such as down-sampled time-domain audio signals and their frequency-domain (spectral) representations. The new data types (d-, w-) are self-defining, and the contents are not processable by any other Csound units. The unit generators are experimental, and subject to change between releases; they will also be joined by others later. Their inclusion here is to offer the user some initial experience in spectral data processing.

INITIALIZATION

idisprd (optional) - if non-zero, display the output every *idisprd* seconds. The default value is 0 (no display).

ihann, idbout, idsines (optional) - if non-zero, then respectively: apply a hanning window to the input; convert the output magnitudes to dB; display the windowed sinusoids used in DFT filtering. The default values are 0, 0, 0 (rectangular window, magnitude output, no sinusoid display).

imul2 (optional) - if non-zero, scale the *wsig2* magnitudes before adding. The default value is 0.

iwtflg (optional) - wait flag. If non-zero, hold each display until released by the user. The default value is 0 (no wait).

interp (optional) - if non-zero, interpolate the output signal *ksum*. The default value is 0 (repeat the signal value between changes).

PERFORMANCE

octdown - put signal *asig* through *iocts* successive applications of octave decimation and downsampling, and preserve *isamps* down-sampled values in each octave. Optionally display the composite buffer every *idisprd* seconds.

noctdft - generate a constant-Q, exponentially-spaced DFT across all octaves of the multiply-downsampled input *dsig*. Every *iprd* seconds, each octave of *dsig* is optionally windowed (*ihann* non-zero), filtered (using *ifrqs* parallel filters per octave, exponentially spaced, and with frequency/bandwidth Q of *iq*), and the output magnitudes optionally converted to dB (*idbout* non-zero). This unit produces a self-defining spectral datablock *wsig*, whose characteristics are readable by any units that receive it as input, and for which it becomes the template for output.

specscal - scale an input spectral datablock with spectral envelopes. Function tables *ifthresh* and *ifscale* are initially sampled across the (logarithmic) frequency space of the input spectrum; then each time a new input spectrum is sensed the sampled values are used to scale each of its magnitude channels as follows: if *ifthresh* is non-zero, each magnitude is reduced by its corresponding table-value (to not less than zero); then each magnitude is rescaled by the corresponding *ifscale* value, and the resulting spectrum written to *wsig*.

specaddm - do a weighted add of two input spectra. For each channel of the two input spectra, the two magnitudes are combined and written to the output according to: magout = mag1in + mag2in * *imul2*. The operation is performed whenever the input *wsig1* is sensed to be new. This unit will (at Initialization) verify the consistency of the two spectra (equal size, equal period, equal mag types).

specdiff - find the positive difference values between consecutive spectral frames. At each new frame of *wsigin*, each magnitude value is compared with its predecessor, and the positive changes written to the output spectrum. This unit is useful as an energy onset detector.

specfilt - filter each channel of an input spectrum. At each new frame of *wsigin*, each magnitude value is injected into a 1st-order lowpass recursive filter, whose half-time constant has been initially set by sampling the ftable *ifhtim* across the (logarithmic) frequency space of the input spectrum. This unit effectively applies a *persistence* factor to the data occurring in each spectral channel, and is useful for simulating the *energy integration* that occurs during auditory perception.

specdisp - display the magnitude values of spectrum *wsig* every *iprd* seconds (rounded to some integral number of *wsig's* originating *iprd*).

specsum - sum the magnitudes across all channels of the spectrum. At each new frame of *wsig*, the magnitudes are summed and released as a scalar *ksum* signal. Between frames, the output is either repeated or interpolated at the K-rate. This unit produces a k-signal summation of the magnitudes present in the spectral data, and is thereby a running measure of its moment-to-moment overall strength.

Example:

```
asig    in                                    ;get external audio
dsamp   octdown   asig, 6, 180, 0             ;downsample in 6 octaves
wsig1   noctdft   dsamp,.02,12,33,0,1,1       ; & calc 72-point dft (db)
wsig2   specdiff  wsig1                       ;sense onsets
wsig3   specfilt  wsig2, 2                    ; & absorb slowly
        specdisp  wsig1, .1                   ;display all spectra
        specdisp  wsig2, .1
        specdisp  wsig3, .1
kstren  specsum   wsig3, 1                    ;sum final mags, and ksmooth
```

SENSING & CONTROL

```
ktemp    tempest   kin, iprd, imindur, imemdur, ihp, ithresh, ihtim, ixfdbak,
                        istartempo, ifn[, idisprd, itweek]
```

Estimate the tempo of beat patterns in a control signal.

INITIALIZATION

iprd - period between analyses (in seconds). Typically about .02 seconds.

imindur - minimum duration (in seconds) to serve as a unit of tempo. Typically about .2 seconds.

imemdur - duration (in seconds) of the *kin* short-term memory buffer which will be scanned for periodic patterns. Typically about 3 seconds.

ihp - half-power point (in cps) of a low-pass filter used to smooth input *kin* prior to other processing. This will tend to suppress activity that moves much faster. Typically 2 cps.

ithresh - loudness threshold by which the low-passed *kin* is center-clipped before being placed in the short-term buffer as tempo-relevant data. Typically at the noise floor of the incoming data.

ihtim - half-time (in seconds) of an internal forward-masking filter that masks new *kin* data in the presence of recent, louder data. Typically about .005 seconds.

ixfdbak - proportion of this unit's *anticipated value* to be mixed with the incoming *kin* prior to all processing. Typically about .3.

istartempo - initial tempo (in beats per minute). Typically 60.

ifn - table number of a stored function (drawn left-to-right) by which the short-term memory data is attenuated over time.

idisprd (optional) - if non-zero, display the short-term past and future buffers every *idisprd* seconds (normally a multiple of *iprd*). The default value is 0 (no display).

itweek (optional) - fine-tune adjust this unit so that it is stable when analyzing events controlled by its own output. The default value is 1 (no change).

PERFORMANCE

tempest examines *kin* for amplitude periodicity, and estimates a current tempo. The input is first low-pass filtered, then center-clipped, and the residue placed in a short-term memory buffer (attenuated over time) where it is analyzed for periodicity using a form of autocorrelation. The period, expressed as a *tempo* in beats per minute, is output as *ktemp*. The period is also used internally to make predictions about future amplitude patterns, and these are placed in a buffer adjacent to that of the input. The two adjacent buffers can be periodically displayed, and the predicted values optionally mixed with the incoming signal to simulate expectation.

This unit is useful for sensing the metric implications of any k-signal (e.g. the RMS of an audio signal, or the second derivative of a conducting gesture), before sending to a **tempo** statement.

Example:

```
ksum    specsum  wsignal, 1                            ; sum the amps of a spectrum
ktemp   tempest  ksum, .02, .1, 3, 2, 800, .005, 0, 60, 4, .1, .995     ;  & look for beats
```

```
kx, ky   xyin    iprd, ixmin, ixmax, iymin, iymax[, ixinit, iyinit]
         tempo   ktempo, istartempo
```

Sense the cursor position in an input window. Apply tempo control to an uninterpreted score.

INITIALIZATION

iprd - period of cursor sensing (in seconds). Typically .1 seconds.

xmin, xmax, ymin, ymax - edge values for the x-y coordinates of a cursor in the input window.

ixinit, iyinit (optional) - initial x-y coordinates reported; the default values are 0,0. If these values are not within the given min-max range, they will be coerced into that range.

istartempo - initial tempo (in beats per minute). Typically 60.

PERFORMANCE

xyin samples the cursor x-y position in an input window every *iprd* seconds. Output values are repeated (not interpolated) at the K-rate, and remain fixed until a new change is registered in the window. There may be any number of input windows. This unit is useful for Realtime control, but continuous motion should be avoided if *iprd* is unusually small.

tempo allows the performance speed of Csound scored events to be controlled from within an orchestra. If the **csound** command's **-B** (beatmode) flag is set, scored events will be performed from their uninterpreted p2 and p3 (beat) parameters, at a default tempo of 60 beats per minute. When a **tempo** statement is activated in any instrument (with *ktempo* > 0.), the operating tempo will be set to *ktempo* beats per minute. There may be any number of **tempo** statements in an orchestra, but coincident activation is best avoided.

Example:

```
kx,ky   xyin    .05, 30, 0, 120, 0, 75      ; sample the cursor
        tempo   kx, 75                      ; and control the tempo of performance
```

SOUND INPUT & OUTPUT

a1	**in**	
a1, a2	**ins**	
a1, a2, a3, a4	**inq**	
a1	**soundin**	ifilno[, iskptim][, iformat]
a1, a2	**soundin**	ifilno[, iskptim][, iformat]
a1, a2, a3, a4	**soundin**	ifilno[, iskptim][, iformat]
	out	asig
	outs1	asig
	outs2	asig
	outs	asig1, asig2
	outq1	asig
	outq2	asig
	outq3	asig
	outq4	asig
	outq	asig1, asig2, asig3, asig4

These units read/write audio data to/from an external device or stream.

INITIALIZATION

ifilno - integer suffix (n) of a binary file named 'soundin.n', assumed to be in the directory SFDIR (see also GEN01).

iskptim (optional) - time in seconds of input sound to be skipped. The default value is 0.

iformat (optional) - specifies the audio data file format (1 = 8-bit signed char, 2 = 8-bit A-law bytes, 3 = 8-bit U-law bytes, 4 = 16-bit short integers, 5 = 32-bit long integers, 6 = 32-bit floats). If *iformat* = 0 it is taken from the soundfile header, and if no header from the **csound -o** command flag. The default value is 0.

PERFORMANCE

in, ins, inq - copy the current values from the standard audio input buffer. If the command flag **-i** is set, sound is read continuously from the audio input stream (e.g. *stdin* or a soundfile) into an internal buffer. Any number of these units can read freely from this buffer.

soundin is functionally an audio generator that derives its signal from a pre-existing file. The number of channels read in is set by the number of result cells, a1, a2, etc. A **soundin** unit opens this file whenever the host instrument is initialized, then closes it again each time the instrument is turned off. There can be any number of **soundin** units within a single instrument or orchestra; also, two or more of them can read simultaneously from the same external file.

out, outs, outq send audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk. There can be any number of these output units in an instrument. The type (mono, stereo, or quad) must agree with **nchnls**, but units can be chosen to direct sound to any particular channel: **outs1** sends to stereo channel 1, **outq3** to quad channel 3, etc.

a1, a2, a3, a4 **pan** asig, kx, ky, ifn[, imode][, ioffset]

Distribute an audio signal amongst four channels with localization control.

INITIALIZATION

ifn - function table number of a stored pattern describing the amplitude growth in a speaker channel as sound moves towards it from an adjacent speaker. Requires extended guard-point.

imode (optional) - mode of the *kx, ky* position values. 0 signifies raw index mode, 1 means the inputs are normalized (0-1). The default value is 0.

ioffset (optional) - offset indicator for *kx, ky*. 0 infers the origin to be at channel 3 (left rear); 1 requests an axis shift to the quadraphonic center. The default value is 0.

PERFORMANCE

pan takes an input signal *asig* and distributes it amongst four outputs (essentially quad speakers) according to the controls *kx* and *ky*. For normalized input (mode=1) and no offset, the four output locations are in order: left-front at (0,1), right-front at (1,1), left-rear at the origin (0,0), and right-rear at (1,0). In the notation *(kx, ky)*, the coordinates *kx* and *ky*, each ranging 0-1, thus control the 'rightness' and 'forwardness' of a sound location.

Movement between speakers is by amplitude variation, controlled by the stored function table *ifn*. As *kx* goes from 0 to 1, the strength of the right-hand signals will grow from the left-most table value to the right-most, while that of the left-hand signals will progress from the right-most table value to the left-most. For a simple linear pan, the table might contain the linear function 0-1. A more correct pan that maintains constant power would be obtained by storing the first quadrant of a sinusoid. Since **pan** will scale and truncate *kx* and *ky* in simple table lookup, a medium-large table (say 8193) should be used.

kx, ky values are not restricted to 0-1. A circular motion passing through all four speakers (escribed) would have a diameter of root 2, and might be defined by a circle of radius R = root 1/2 with center at (.5,.5). *kx, ky* would then come from Rcos(angle), Rsin(angle), with an implicit origin at (.5,.5) (i.e. ioffset=1). Unscaled raw values operate similarly. Sounds can thus be located anywhere in the polar or cartesian plane; points lying outside the speaker square are projected correctly onto the square's perimeter as for a listener at the center.

Example:

```
                instr   1
        k1      phasor  1/p3                    ;fraction of circle
        k2      tablei  k1, 1, 1                ;sin of angle (sinusoid in f1)
        k3      tablei  k1, 1, 1, .25, 1        ;cos of angle (sin offset 1/4 circle)
        a1      oscili  10000, 440, 1           ;audio signal ..
 a1,a2,a3,a4    pan     a1, k2/2, k3/2, 2, 1, 1 ; sent in a circle (f2=1st quad sin)
                outq    a1, a2, a3, a4
                endin
```

SIGNAL DISPLAY

print	iarg[, iarg, ...]
display	xsig, iprd[, iwtflg]
dispfft	asig, iprd, iwsiz[, iwtyp][, idbout][, iwtflg]

These units will print orchestra Init-values, or produce graphic display of orchestra control signals and audio signals. Uses X11 windows if enabled, else (or if -g flag is set) displays are approximated in ascii characters.

INITIALIZATION

iprd - the period of display in seconds.

iwsiz - size of the input window in samples. A window of *iwsiz* points will produce a Fourier transform of *iwsiz/2* points, spread linearly in frequency from 0 to sr/2. *iwsiz* must be a power of 2. The windows are permitted to overlap.

iwtyp (optional) - window type. 0 = rectangular, 1 = hanning. The default value is 0 (rectangular).

idbout (optional) - units of output for the Fourier coefficients. 0 = magnitude, 1 = decibels. The default is 0 (magnitude).

iwtflg (optional) - wait flag. If non-zero, each display is held until released by the user. The default value is 0 (no wait).

PERFORMANCE

print - print the current value of the I-time arguments (or expressions) *iarg* at every I-pass through the instrument.

display - display the audio or control signal *xsig* every *iprd* seconds, as an amplitude vs time graph.

dispfft - display the Fourier Transform of the audio signal *asig* every *iprd* seconds using the Fast Fourier Transform method.

Example:

```
k1      envlpx   1,.03,p3,.05,1,.5,.01       ;generate a note envelope
        display  k1, p3                      ;and display entire shape
```

3. THE STANDARD NUMERIC SCORE

A score is a data file that provides information to an orchestra about its performance. Like an orchestra file, a score file is made up of statements in a known format. The **Csound** orchestra expects to be handed a score comprised mainly of *ascii numeric characters*. Although most users will prefer a higher level score language such as provided by **Cscore, Scot**, or other score-generating programs, each resulting score must eventually appear in the format expected by the orchestra. A Standard Numeric Score can be created and edited directly by the beginner using standard text editors; indeed, some users continue to prefer it. The purpose of this section is to describe this format in detail.

The basic format of a standard numeric score statement is:

opcode p1 p2 p3 p4 ... ;comments

The *opcode* is a single character, always alphabetic. Legal opcodes are **f, i, a, t, s**, and **e**, the meanings of which are described in the following pages. The opcode is normally the first character of a line; leading spaces or tabs will be ignored. Spaces or tabs between the opcode and *p1* are optional.

p1, p2, p3, etc. are *parameter fields (pfields).* Each contains a floating point number comprised of an optional sign, digits, and an optional decimal point. Expressions are not permitted in Standard Score files. *pfields* are separated from each other by one or more spaces or tabs, all but one space of which will be ignored.

Continuation lines are permitted. If the first printing character of a new scoreline is not an opcode, that line will be regarded as a continuation of the pfields from the previous scoreline.

Comments are optional and are for the purpose of permitting the user to document his score file. Comments always begin with a semicolon (;) and extend to the end of the line. Comments will not affect the pfield continuation feature.

Blank lines or comment-only lines are legal (and will be ignored).

Preprocessing of Standard Scores

A Score (a collection of score statements) is divided into time-ordered *sections* by the **s** statement. Before being read by the orchestra, a score is *preprocessed* one section at a time. Each section is normally processed by 3 routines: Carry, Tempo, and Sort.

1. Carry - within a group of consecutive **i** statements whose p1 whole numbers correspond, any pfield left empty will take its value from the same pfield of the preceding statement. An empty pfield can be denoted by a single point (.) delimited by spaces. No point is required after the last non-empty pfield. The output of Carry preprocessing will show the carried values explicitly. The Carry Feature is not affected by intervening comments or blank lines; it is turned off only by a *non-i* statement or by an **i** statement with unlike p1 whole number.

An additional feature is available for p2 alone. The symbol + in p2 will be given the value of p2+p3 from the preceding **i** statement. This enables note action times to be automatically determined from the sum of preceding durations. The + symbol can itself be carried. The + symbol is legal only in p2. E.g.,

the statements	i1 0 .5 100	will result in	i1 0 .5 100
	i. +		i1 .5 .5 100

i	i1 1 .5 100

The Carry feature should be used liberally. Its use, especially in large scores, can greatly reduce input typing and will simplify later changes.

2. Tempo - this operation time warps a score section according to the information in a **t** statement. The tempo operation converts p2 (and, for **i** statements, p3) from original beats into real seconds, since those are the units required by the orchestra. After time warping, score files will be seen to have orchestra-readable format demonstrated by the following:

i p1 p2beats p2seconds p3beats p3seconds p4 p5 ....

3. Sort - this routine sorts all action-time statements into chronological order by p2 value. It also sorts coincident events into precedence order. Whenever an **f** statement and an **i** statement have the same p2 value, the **f** statement will precede. Whenever two or more **i** statements have the same p2 value, they will be sorted into ascending p1 value order. If they also have the same p1 value, they will be sorted into ascending p3 value order. Score sorting is done section by section (see **s** statement). Automatic sorting implies that score statements may appear in any order within a section.

N.B. The operations Carry, Tempo and Sort are combined in a 3-phase single pass over a score file, to produce a new file in orchestra-readable format (see the Tempo example). Processing can be invoked either explicity by the **scsort** command, or implicity by **csound** which processes the score before calling the orchestra. Source-format files and orchestra-readable files are both in ascii-character form, and may be either perused or further modified by standard text editors. User-written routines can be used to modify score files before or after the above processes, provided the final orchestra-readable statement format is not violated. Sections of different formats can be sequentially batched; and sections of like format can be merged for automatic sorting.

Next-P and Previous-P Symbols

At the close of any of the above operation, three additional score features are interpreted during file writeout: next-p, previous-p, and ramping.

i statement pfields containing the symbols **np***x* or **pp***x* (where x is some integer) will be replaced by the appropriate pfield value found on the next **i** statement (or previous **i** statement) that has the same p1. For example, the symbol np7 will be replaced by the value found in p7 of the next note that is to be played by this instrument. np and pp symbols are recursive and can reference other np and pp symbols which can reference others, etc. References must eventually terminate in a real number or a ramp symbol (see below). Closed loop references should be avoided. np and pp symbols are illegal in p1,p2 and p3 (although they may reference these). np and pp symbols may be Carried. np and pp references cannot cross a Section boundary; any forward or backward reference to a non-existent note-statement will be given the value zero. For example,

the statements		will result in	
	i1 0 1 10 np4 pp5		i1 0 1 10 20 0
	i1 1 1 20		i1 1 1 20 30 20
	i1 2 1 30		i1 2 1 30 0 30

np and pp symbols can provide an instrument with contextual knowledge of the score, enabling it to glissando or crescendo, for instance, toward the pitch or dynamic of some future event (which may or may not be immediately adjacent). Note that while the Carry feature will propagate np and pp through unsorted statements, the operation that interprets these symbols is acting on a time-warped and fully sorted version of the score.

Ramping

i statement pfields containing the symbol < will be replaced by values derived from linear interpolation of a time-based ramp. Ramps are anchored at each end by the first real number found in the same pfield of a preceding and following note played by the same instrument. For example,

the statements		will result in	
	i1 0 1 100		i1 0 1 100
	i1 1 1 <		i1 1 1 200
	i1 2 1 <		i1 2 1 300
	i1 3 1 400		i1 3 1 400

```
i1  4  1   <                  i1  4  1  200
i1  5  1   0                  i1  5  1    0
```

Ramps cannot cross a Section boundary. Ramps cannot be anchored by an np or pp symbol (although they may be referenced by these). Ramp symbols are illegal in p1,p2 and p3. Ramp symbols may be Carried. Note, however, that while the Carry feature will propagate ramp symbols through unsorted statements, the operation that interprets these symbols is acting on a time-warped and fully sorted version of the score. In fact, time-based linear interpolation is based on warped score-time, so that a ramp which spans a group of accelerating notes will remain linear with respect to strict chronological time.

F STATEMENT (or FUNCTION TABLE STATEMENT)

f p1 p2 p3 p4 ...

This causes a GEN subroutine to place values in a stored function table for use by instruments.

PFIELDS

p1	Table number (from 1 to 100) by which the stored function will be known. A negative number requests that the table be destroyed.
p2	Action time of function generation (or destruction) in beats
p3	Size of function table (i.e. number of points). Must be a power of 2, or a power-of-2 plus 1 (see below). Maximum table size is 16777216 (2**24) points.
p4	Number of the GEN routine to be called (see GEN ROUTINES). A negative value will cause rescaling to be omitted.
p5 p6 . . .	Parameters whose meaning is determined by the particular GEN routine.

SPECIAL CONSIDERATIONS

Function tables are arrays of floating-point values. Arrays can be of any length in powers of 2; space allocation always provides for 2**n points plus an additional *guard point*. The guard point value, used during interpolated lookup, can be automatically set to reflect the table's purpose: If *size* is an exact power of 2, the guard point will be a copy of the first point; this is appropriate for *interpolated wrap-around* lookup as in **oscili,** etc., and should even be used for non-interpolating **oscil** for safe consistency. If *size* is set to 2**n + 1, the guard point value automatically extends the contour of table values; this is appropriate for single-scan functions such in **envlpx, oscil1, oscil1i**, etc.

Table space is allocated in primary memory, along with instrument data space. The maximum table number has a soft limit of 100; this can be extended if required.

An existing function table can be removed by an **f** statement containing a negative p1 and an appropriate action time. A function table is also be removed by the generation of another table with the same p1. Functions are not automatically erased at the end of a score section.

p2 action time is treated in the same way as in **i** statements with respect to sorting and modification by **t** statements. If an **f** statement and an **i** statement have the same p2, the sorter gives the **f** statement precedence so that the function table will be available during note initialization.

An **f 0** statement (zero p1, positive p2) may be used to create an action time with no associated action. Such time markers are useful for padding out a score section (see **s** statement).

I STATEMENT (INSTRUMENT or NOTE STATEMENT)

i p1 p2 p3 p4 ...

This statement calls for an instrument to be made active at a specific time and for a certain duration. The parameter field values are passed to that instrument prior to its Initialization, and remain valid throughout its Performance.

PFIELDS

p1	Instrument number, usually a non-negative integer. An optional fractional part can provide an additional tag for specifying ties between particular notes of consecutive clusters. A negative p1 (including tag) can be used to turn off a particular 'held' note.
p2	Starting time in arbitrary units called beats.
p3	Duration time in beats (usually positive). A negative value will initiate a held note (see also **ihold**). A zero value will invoke an initialization pass without performance (see also **instr**).
p4 p5 . .	Parameters whose significance is determined by the instrument.

SPECIAL CONSIDERATIONS

Beats are evaluated as seconds unless there is a **t** statement in this section.

Starting or action times are relative to the beginning of a section (see **s** statement), which is assigned time 0.

Note statements within a section may be placed in any order. Before being sent to an orchestra, unordered score statements must first be processed by Sorter, which will reorder them by ascending p2 value. Notes with the same p2 value will be ordered by ascending p1; if the same p1, then by ascending p3.

Notes may be stacked, i.e., a single instrument can perform any number of notes simultaneously. (The necessary copies of the instrument's data space will be allocated dynamically by the orchestra loader.) Each note will normally turn off when its p3 duration has expired. However, an instrument may modify its own duration by modifying its p3 value during note initialization.

An instrument may be turned on and left to perform indefinitely either by giving it a negative p3 or by including an **ihold** in its I-time code. If a held note is active, an **i** statement *with matching p1* will not cause a new allocation but will take over the data space of the held note. The new pfields (including p3) will now be in effect, and an I-time pass will be executed in which the units can either be newly initialized or allowed to continue as required for a tied note (see **tigoto**). A held note may be succeeded either by another held note or by a note of finite duration. A held note will continue to perform across section endings (see **s** statement). It is halted only by **turnoff** or by an **i** statement with negative matching p1 or by an **e** statement.

A STATEMENT (or ADVANCE STATEMENT)

a p1 p2 p3

This causes score time to be advanced by a specified amount without producing sound samples.

PFIELDS

p1	carries no meaning. Usually zero
p2	Action time, in beats, at which advance is to begin.
p3	Durational span (distance in beats) of time advance.
p4,p5, etc	carry no meaning.

SPECIAL CONSIDERATIONS

This statement allows the beat count within a score section to be advanced without generating intervening sound samples. This can be of use when a score section is incomplete (the beginning or middle is missing) and the user does not wish to generate and listen to a lot of silence.

p2 action time and p3 distance are treated as in **i** statements, with respect to sorting and modification by **t** statements.

An **a** statement will be temporarily inserted in the score by the Score Extract feature when the extracted segment begins later than the start of a Section. The purpose of this is to preserve the beat count and time count of the original score for the benefit of the peak amplitudes messages which are reported on the user console.

Whenever an **a** statement is encountered by a performing orchestra, its presence and effect will be reported on the user's console.

T STATEMENT (TEMPO STATEMENT)

t p1 p2 p3 p4 ..... (unlimited)

This statement sets the tempo and specifies the accelerations and decelerations for the current section. This is done by converting beats into seconds.

PFIELDS

p1	must be zero
p2	initial tempo in beats per minute
p3, p5, p7, ...	times in beats (in non-decreasing order)
p4, p6, p8, ...	tempi for the referenced beat times

SPECIAL CONSIDERATIONS

Time and Tempo-for-that-time are given as ordered couples that define points on a "tempo vs time" graph. (The time-axis here is in beats so is not necessarily linear.) The beat-rate of a Section can be thought of as a movement from point to point on that graph: motion between two points of equal height signifies constant tempo, while motion between two points of unequal height will cause an accelarando or ritardando accordingly. The graph can contain discontinuities: two points given equal times but different tempi will cause an immediate tempo change.

Motion between different tempos over non-zero time is inverse linear. That is, an accelarando between two tempos M1 and M2 proceeds by linear interpolation of the single-beat durations from 60/M1 to 60/M2.

The first tempo given must be for beat 0.

A tempo, once assigned, will remain in effect from that time-point unless influenced by a succeeding tempo, i.e. the last specified tempo will be held to the end of the section.

A **t** statement applies only to the score section in which it appears. Only one **t** statement is meaningful in a section; it can be placed anywhere within that section. If a score section contains no **t** statement, then beats are interpreted as seconds (i.e. with an implicit **t 0 60** statement).

S STATEMENT

s anything

The **s** statement marks the end of a section.

PFIELDS

All pfields are ignored.

SPECIAL CONSIDERATIONS

Sorting of the **i, f** and **a** statements by action time is done section by section.

Time warping for to the **t** statement is done section by section.

All action times within a section are relative to its beginning. A section statement establishes a new relative time of 0, but has no other reinitializing effects (e.g. stored function tables are preserved across section boundaries).

A section is considered complete when all action times and finite durations have been satisfied (i.e., the "length" of a section is determined by the last occurring action or turn-off). A section can be extended by the use of an **f 0** statement.

A section ending automatically invokes a Purge of inactive instrument and data spaces.

N.B. Since score statements are processed section by section, the amount of memory required depends on the maximum number of score statements in a section. Memory allocation is dynamic, and the user will be informed as extra memory blocks are requested during score processing.

For the end of the final section of a score, the **s** statement is optional; the **e** statement may be used instead.

E STATEMENT

e anything

This statement may be used to mark the end of the last section of the score.

PFIELDS

All pfields are ignored

SPECIAL CONSIDERATIONS

The **e** statement is contextually identical to an **s** statement. Additionally, the **e** statement terminates all signal generation (including indefinite performance) and closes all input and output files.

If an **e** statement occurs before the end of a score, all subsequent score lines will be ignored.

The **e** statement is optional in a score file yet to be sorted. If a score file has no **e** statement, then Sort processing will supply one.

4. GEN ROUTINES

The GEN subroutines are function-drawing procedures called by **f** statements to construct stored wavetables. They are available throughout orchestra performance, and can be invoked at any point in the score as given by p2. p1 assigns a table *number*, and p3 the table *size* (see **f** statement). p4 specifies the GEN routine to be called; each GEN routine will assign special meaning to the pfield values that follow.

GEN01, GEN02

Transfer data from a soundfile (GEN01) or from immediate pfields (GEN02) into a function table.

f # time size 1 filno skiptime format
f # time size 2 v1 v2 v3 . . .

size - number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see **f** statement). The maximum tablesize is 16777216 (2**24) points.

filno, skiptime, format - directs GEN01 to read from **soundin.***filno*, beginning at *skiptime* seconds into the file. The parameter *format* specifies the audio data-file format:

1 - 8-bit signed character	4 - 16-bit short integers
2 - 8-bit A-law bytes	5 - 32-bit long integers
3 - 8-bit U-law bytes	6 - 32-bit floats

If *format* = 0 the sample format is taken from the soundfile header, or by default from the **csound -o** command flag. The soundfile is assumed to be in the directory SFDIR (see also **soundin**). Reading stops at end-of-file or when the table is full. Any table locations not filled will contain zeros.

v1, v2, v3, ... - for GEN02 these values will be copied directly into the table space. The number of values is limited by the compile-time variable PMAX, which controls the maximum pfields (currently 150). The values copied may include the table guard point; any table locations not filled will contain zeros.

Note:

If p4 is positive, the table will be post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative p4 will cause rescaling to be skipped.

Example:

f 1 0 16 -2 0 1 2 3 4 5 6 7 8 9 10 11 0

This calls upon GEN02 to place 12 values plus an explicit wrap-around guard value into a table of size next-highest power of 2. Rescaling is inhibited.

GEN03

This subroutine generates a stored function table by evaluating a polynomial in x over a fixed interval and with specified coefficients.

f # time size 3 xval1 xval2 c0 c1 c2 . . . cn

size - number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see **f** statement).

xval1, xval2 - left and right values of the x interval over which the polynomial is defined (*xval1* < *xval2*). These will produce the 1st stored value and the (power-of-2 plus 1)th stored value respectively in the generated function table.

c0, c1, c2, ... cn - coefficients of the nth-order polynomial

$$c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \cdots + c_n x^n$$

Coefficients may be positive or negative real numbers; a zero denotes a missing term in the polynomial. The coefficient list begins in p7, providing an upper limit of 44 terms.

Note:

The defined segment [fn(*xval1*),fn(*xval2*)] is evenly distributed. Thus a 512-point table over the interval [-1,1] will have its origin at location 257 (at the start of the 2nd half). Provided the extended guard point is requested, both fn(-1) and fn(1) will exist in the table.

GEN03 is useful in conjunction with **table** or **tablei** for audio waveshaping (sound modification by non-linear distortion). Coefficients to produce a particular formant from a sinusoidal lookup index of known amplitude can be determined at preprocessing time using algorithms such as Chebyshev formulae. See also GEN13.

Example:

```
f 1 0 1025 3 -1 1 5 4 3 2 1
```

This calls GEN03 to fill a table with a 4th order polynomial function over the x-interval -1 to 1. The origin will be at the offset position 512. The function is post-normalized.

GEN04

This subroutine generates a normalizing function by examining the contents of an existing table.

f # time size 4 source# sourcemode

size - number of points in the table. Should be power-of-2 plus 1. Must not exceed (except by 1) the size of the source table being examined; limited to just half that size if the sourcemode is of type offset (see below).

source# - table number of stored function to be examined.

sourcemode - a coded value, specifying how the source table is to be scanned to obtain the normalizing function. Zero indicates that the source is to be scanned from left to right. Non-zero indicates that the source has a bipolar structure; scanning will begin at the mid-point and progress outwards, looking at pairs of points equidistant from the center.

Note:

The normalizing function derives from the progressive absolute maxima of the source table being scanned. The new table is created left-to-right, with stored values equal to 1/(absolute maximum so far scanned). Stored values will thus begin with 1/(first value scanned), then get progressively smaller as new maxima are encountered. For a source table which is normalized (values <= 1), the derived values will range from 1/(first value scanned) down to 1. If the first value scanned is zero, that inverse will be set to 1.

The normalizing function from GEN04 is not itself normalized.

GEN04 is useful for scaling a table-derived signal so that it has a consistent peak amplitude. A particular application occurs in waveshaping when the carrier (or indexing) signal is less than full amplitude.

Example:

```
f 2 0 512 4 1 1
```

This creates a normalizing function for use in connection with the GEN03 table 1 example. Midpoint bipolar offset is specified.

GEN05, GEN07

These subroutines are used to construct functions from segments of exponential curves (GEN05) or straight lines (GEN07).

> **f** # time size 5 a n1 b n2 c . . .
> **f** # time size 7 a n1 b n2 c . . .

size - number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see **f** statement).

a, b, c, etc. - ordinate values, in odd-numbered pfields p5, p7, p9, ... For GEN05 these must be non-zero and must be alike in sign. No such restrictions exist for GEN07.

n1, n2, etc. - length of segment (no. of storage locations), in even-numbered pfields. Cannot be negative, but a zero is meaningful for specifying discontinuous waveforms (e.g. in the example below). The sum n1 + n2 + ... will normally equal *size* for fully specified functions. If the sum is smaller, the function locations not included will be set to zero; if the sum is greater, only the first *size* locations will be stored.

Note:

If p4 is positive, functions are post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative p4 will cause rescaling to be skipped.

Discrete-point linear interpolation implies an increase or decrease along a segment by equal differences between adjacent locations; exponential interpolation implies that the progression is by equal ratio. In both forms the interpolation from *a* to *b* is such as to assume that the value *b* will be attained in the n+1 th location. For discontinuous functions, and for the segment encompassing the end location, this value will not actually be reached, although it may eventually appear as a result of final scaling.

Example:

f 1 0 256 7 0 128 1 0 -1 128 0

This describes a single-cycle sawtooth whose discontinuity is mid-way in the stored function.

GEN06

This subroutine will generate a function comprised of segments of cubic polynomials, spanning specified points just three at a time.

f # time size 6 a n1 b n2 c n3 d . . .

size - number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see **f** statement).

a, c, e ... - local maxima or minima of successive segments, depending on the relation of these points to adjacent inflexions. May be either positive or negative.

b, d, f ... - ordinate values of points of inflexion at the ends of successive curved segments. May be positive or negative.

n1, n2, n3 ... - number of stored values between specified points. Cannot be negative, but a zero is meaningful for specifying discontinuities. The sum n1 + n2 + ... will normally equal *size* for fully specified functions. (for details, see GEN05).

Note:

GEN06 constructs a stored function from segments of cubic polynomial functions. Segments link ordinate values in groups of 3: point of inflexion, maximum/minimum, point of inflexion. The first complete segment encompasses *b,c,d* and has length *n2+n3*, the next encompasses *d,e,f* and has length *n4+n5*, etc. The first segment (*a,b* with length *n1*) is partial with only one inflexion; the last segment may be partial too. Although the inflexion points *b,d,f*.. each figure in two segments (to the left and right), the slope of the two segments remains independent at that common point (i.e. the 1st derivative will likely be discontinuous). When *a,c,e*... are alternately maximum and minimum, the inflexion joins will be relatively smooth; for successive maxima or successive minima the inflexions will be comb-like.

Example:

f 1 0 65 6 0 16 .5 16 1 16 0 16 -1

This creates a curve running 0 to 1 to -1, with a minimum, maximum and minimum at these values respectively. Inflexions are at .5 and 0, and are relatively smooth.

GEN08

This subroutine will generate a piecewise cubic spline curve, the smoothest possible through all specified points.

f # time size 8 a n1 b n2 c n3 d . . .

size - number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see **f** statement).

a, b, c ... ordinate values of the function.

n1, n2, n3 ... length of each segment measured in stored values. May not be zero, but may be fractional. A particular segment may or may not actually store any values; stored values will be generated at integral points from the beginning of the function. The sum n1 + n2 + ... will normally equal "size" for fully specified functions.

Note:

GEN08 constructs a stored table from segments of cubic polynomial functions. Each segment runs between two specified points but depends as well on their neighbors on each side. Neighboring segments will agree in both value and slope at their common point. (The common slope is that of a parabola through that point and its two neighbors). The slope at the two ends of the function is constrained to be zero (flat).

Hint: to make a discontinuity in slope or value in the function as stored, arrange a series of points in the interval between two stored values; likewise for a non-zero boundary slope.

Examples:

f 1 0 65 8 0 16 0 16 1 16 0 16 0

This example creates a curve with a smooth hump in the middle, going briefly negative outside the hump then flat at its ends.

f 2 0 65 8 0 16 0 .1 0 15.9 1 15.9 0 .1 0 16 0

This example is similar, but does not go negative.

GEN09, GEN10

These subroutines generate composite waveforms made up of weighted sums of simple sinusoids. The specification of each contributing partial requires 3 pfields using GEN09 but just 1 using GEN10.

f # time size 9 pna stra phsa pnb strb phsb . . .
f # time size 10 str1 str2 str3 str4 . . . .

size - number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see **f** statement).

pna, pnb, etc. - partial no. (relative to a fundamental that would occupy *size* locations per cycle) of sinusoid a, sinusoid b, etc. Must be positive, but need not be a whole number, i.e., non-harmonic partials are permitted. Partials may be in any order.

stra, strb, etc. - strength of partials *pna, pnb,* etc. These are relative strengths, since the composite waveform will be rescaled later. Negative values are permitted and imply a 180 degree phase shift.

phsa, phsb, etc. - inital phase of partials *pna, pnb,* etc., expressed in degrees.

str1, str2, str3, etc. - relative strengths of the fixed harmonic partial numbers 1,2,3,etc., beginning in p5. Partials not required should be given a strength of zero.

Note:

Both subroutines generate stored functions as sums of sinusoids of different frequencies. The two major restrictions on GEN10 -- that the partials be harmonic and in phase -- do not apply to GEN09.

In either case the composite wave, once drawn, is then rescaled to unity if p4 was positive. A negative p4 will cause rescaling to be skipped.

Example:

f 1 0 512 9 1 3 0 3 1 0 9 .3333 180

This combines partials 1, 3 and 9 in the relative strengths with which they are present in a square wave, except that partial 9 is "upside down."

GEN11

This subroutine generates an additive set of cosine partials, in the manner of csound generators **buzz** and **gbuzz**.

f # time size 11 nh lh r

size - number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see **f** statement).

nh - number of harmonics requested. Must be positive.

lh (optional) - lowest harmonic partial present. Can be positive, zero or negative. The set of partials can begin at any partial number and proceeds upwards; if *lh* is negative, all partials below zero will reflect in zero to produce positive partials without phase change (since cosine is an even function), and will add constructively to any positive partials in the set. The default value is 1.

r (optional) - multiplier in an amplitude coefficient series. This is a power series: if the *lh*th partial has a strength coefficient of A, the (*lh*+n)th partial will have a coefficient of A * *r***n, i.e. strength values trace an exponential curve. *r* may be positive, zero or negative, and is not restricted to integers. The default value is 1.

Note:

This subroutine is a non-time-varying version of the csound **buzz** and **gbuzz** generators, and is similarly useful as a complex sound source in subtractive synthesis. With *lh* and *r* present it parallels **gbuzz**; with both absent or equal to 1. it reduces to the simpler **buzz** (i.e. *nh* equal-strength harmonic partials beginning with the fundamental).

Sampling the stored waveform with an oscillator is more efficient than using dynamic buzz units. However, the spectral content is invariant, and care is necessary lest the higher partials exceed the Nyquist during sampling to produce foldover.

Examples:

```
f 1 0 2049  11 4
f 2 0 2049  11 4 1 1
f 3 0 2049 -11 7 3 .5
```

The first two tables will contain identical band-limited pulse waves of four equal-strength harmonic partials beginning with the fundamental. The third table will sum seven consective harmonics, beginning with the third, and at progressively weaker strengths (1, .5, .25, .125 ...). It will not be post-normalized.

GEN12

This generates the log of a modified Bessel function of the second kind, order 0, suitable for use in amplitude-modulated FM.

f # time size -12 xint

size - number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see **f** statement). The normal value is power-of-2 plus 1.

xint - specifies the x interval [*0 to +xint*] over which the function is defined.

Note:

This subroutine draws the natural log of a modified Bessel function of the second kind, order 0 (commonly written as I subscript 0), over the x-interval requested. The call should have rescaling inhibited.

The function is useful as an amplitude scaling factor in cycle-synchronous amplitude-modulated FM. (See Palamin & Palamin, *J. Audio Eng. Soc.*, 36/9, Sept. 1988, pp. 671-684.) The algorithm is interesting because it permits the normally symmetric FM spectrum to be made asymmetric around a frequency other than the carrier, and is thereby useful for formant positioning. By using a table lookup index of *I(r - 1/r)*, where *I* is the FM modulation index and *r* is an exponential parameter affecting partial strengths, the Palamin algorithm becomes relatively efficient, requiring only oscil's, table lookups, and a single *exp* call.

Example:

f 1 0 2049 -12 20

This draws an unscaled ln(I0(x)) from 0 to 20.

GEN13, GEN14

These subroutines use Chebyshev coefficients to generate stored polynomial functions which, under waveshaping, can be used to split a sinusoid into harmonic partials having a predefinable spectrum.

f # time size 13 xint xamp h0 h1 h2 . . . hn
f # time size 14 xint xamp h0 h1 h2 . . . hn

size - number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see **f** statement). The normal value is power-of-2 plus 1.

xint - provides the left and right values [*-xint,+xint*] of the *x* interval over which the polynomial is to be drawn. These subroutines both call GEN03 to draw their functions; the p5 value here is therefor expanded to a negative-positive p5,p6 pair before GEN03 is actually called. The normal value is 1.

xamp - amplitude scaling factor of the sinusoid input that is expected to produce the following spectrum.

h0, h1, h2, ... hn - relative strength of partials 0 (DC), 1 (fundamental), 2 ... that will result when a sinusoid of amplitude *xamp * int(size/2)/xint* is waveshaped using this function table. These values thus describe a frequency spectrum associated with a particular factor *xamp* of the input signal.

Note:

GEN13 is the function generator normally employed in standard waveshaping. It stores a polynomial whose coefficients derive from the Chebyshev polynomials of the first kind, so that a driving sinusoid of strength xamp will exhibit the specified spectrum at output. Note that the evolution of this spectrum is generally not linear with varying xamp. However, it is bandlimited (the only partials to appear will be those specified at generation time); and the partials will tend to occur and to develop in ascending order (the lower partials dominating at low xamp, and the spectral richness increasing for higher values of xamp). A negative hn value implies a 180 degree phase shift of that partial; the requested full-amplitude spectrum will not be affected by this shift, although the evolution of several of its component partials may be. The pattern +,+,-,-,+,+, ... for *h0,h1,h2*... will minimize the normalization problem for low *xamp* values (see above), but does not necessarily provide the smoothest pattern of evolution.

GEN14 stores a polynomial whose coefficients derive from Chebyshevs of the second kind.

Example:

f 1 0 1025 13 1 1 0 5 0 3 0 1

This creates a function which, under waveshaping, will split a sinusoid into 3 odd-harmonic partials of relative strength 5:3:1.

GEN15

This subroutine creates two tables of stored polynomial functions, suitable for use in phase quadrature operations.

f # time size 15 xint xamp h0 phs0 h1 phs1 h2 phs2 . .

size - number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see **f** statement). The normal value is power-of-2 plus 1.

xint - provides the left and right values [-*xint,+xint*] of the *x* interval over which the polynomial is to be drawn. This subroutine will eventually call GEN03 to draw both functions; this p5 value is therefor expanded to a negative-positive p5,p6 pair before GEN03 is actually called. The normal value is 1.

xamp - amplitude scaling factor of the sinusoid input that is expected to produce the following spectrum.

h0, h1, h2, ... hn - relative strength of partials 0 (DC), 1 (fundamental), 2 ... that will result when a sinusoid of amplitude *xamp* * *int(size/2)/xint* is waveshaped using this function table. These values thus describe a frequency spectrum associated with a particular factor *xamp* of the input signal.

phs0, phs1, ... - phase in degrees of desired harmonics *h0, h1, ...* when the two functions of GEN15 are used with phase quadrature.

Note:

GEN15 creates two tables of equal size, labelled **f** # and **f** #*+1*. Table # will contain a Chebyshev function of the first kind, drawn using GEN13 with partial strengths *h0cos(phs0), h1cos(phs1), ...* Table #*+1* will contain a Chebyshev function of the 2nd kind by calling GEN14 with partials *h1sin(phs1), h2sin(phs2), ...* (note the harmonic displacement). The two tables can be used in conjunction in a waveshaping network that exploits phase quadrature.

5. CSCORE

Cscore is a program for generating and manipulating numeric score files. It comprises a number of *function* subprograms, called into operation by a user-written *main* program. The *function* programs augment the **C** language library functions; they can optionally read *standard numeric* score files, can massage and expand the data in various ways, then write the data out as a new score file to be read by a **Csound** orchestra.

The user-written *main* program is also in **C**. It is not essential to know the **C** language well to write a main program, since the function calls have a simple syntax, and are powerful enough to do most of the complicated work. Additional power can come from **C** later as the need arises.

Events, Lists, and Operations

An *event* in **Cscore** is equivalent to one statement of a *standard numeric score*. It is either created or read in from an existing score file. An event is comprised of an *opcode* and an array of *pfield* values stored somewhere in memory. Storage is organized by the following structure:

```
struct event {
        char op;                /* opcode */
        char tnum;
        short pcnt;
        float p[PMAX+1];        /* pfields */
};
```

Any function subprogram that creates, reads, or copies an event function will return a *pointer* to the storage structure holding the event data. The event pointer can be used to access any component of the structure, in the form of *e->op* or *e->p[n]*. Each newly stored event will give rise to a new pointer, and a sequence of new events will generate a sequence of distinct pointers that must themselves be stored away. Groups of event pointers are stored in a *list*, which has its own structure:

```
struct evlist {
        int nslots;             /* size of this list */
        struct event *e[1];     /* list of event pointers */
};
```

Any function that creates or modifies a *list* will return a *pointer* to the new list. The list pointer can be used to access any of its component event pointers, in the form of *a->e[n]*. Event pointers and list pointers are thus primary tools for manipulating the data of a score file.

Pointers and *lists of pointers* can be copied and reordered without modifying the data values they refer to. This means that notes and phrases can be copied and manipulated from a high level of control. Alternatively, the data within an *event* or *group of events* can be modified without changing the event or list pointers. **Cscore** provides a library of programming *methods* or *function subprograms* by which scores can be created and manipulated in this way.

In the following summary of **Cscore** function calls, some simple naming conventions are used:

the symbols *e, f* are pointers to events (notes);
the symbols *a, b* are pointers to lists(arrays) of such events;
the letters *ev* at the end of a function name signify operation on an *event*;
the letter *l* at the start of a function name signifies operation on a *list*.

calling syntax   description

```
e = createv(n);             create a blank event with n pfields
e = defev("...");           defines an event as per the character string ...
e = copyev(f);              make a new copy of event f
e = getev();                read the next event in the score input file
putev(e);                   write event e to the score output file
putstr("...");              write the character string ... to score output

a = lcreat(n);              create an empty event list with n slots
a = lappev(a,e);            append event e to list a
n = lcount(a);              count the events now in list a
a = lcopy(b);               copy the list b (but not the events)
a = lcopyev(b);             copy the events of b, making a new list
a = lget();                 read events from score input (to next s or e)
lput(a);                    write the events of list a to score output
a = lsepf(b);               separate the f statements from list b into list a
a = lcat(a,b);              concatenate (append) the list b onto the list a
lsort(a);                   sort the list a into chronological order by p[2]
a = lxins(b,"...");         extract notes of instruments ... (no new events)
a = lxtimev(b,from,to);     extract notes of time-span, creating new events

relev(e);                   release the space of event e
lrel(a);                    release the space of list a (but not events)
lrelev(a);                  release the events of list a, and the list space
```

Writing a Main program.

The general format for a main program is:

```
#include <csound/cscore.h>
main()
{
        /* VARIABLE DECLARATIONS */

        /* PROGRAM BODY */
}
```

The *include* statement will define the event and list structures for the program. The following C program will read from a standard numeric score, up to (but not including) the first **s** or **e** statement, then write that data (unaltered) as output.

```
#include <csound/cscore.h>
main()
{
        struct evlist *a;       /* a is allowed to point to an event list */

        a = lget();             /* read events in, return the list pointer */
        lput(a);                /* write these events out (unchanged) */
        putstr("e");            /* write the string e to output */
}
```

After execution of *lget()*, the variable *a* points to a list of event addresses, each of which points to a stored event. We have used that same pointer to enable another list function (*lput*) to access and write out all of the events that were read. If we now define another symbol *e* to be an *event* pointer, then the statement

```
e = a->e[4];
```

will set it to the *contents* of the 4th slot in the evlist structure. The contents is a pointer to an event, which is itself comprised of an array of parameter field values. Thus the term e->p[5] will mean the value of parameter field 5 of the 4th event in the evlist denoted by *a*. The program below will multiply the value of that pfield by 2 before writing it out.

```
#include <csound/cscore.h>
main()
{
        struct event  *e;      /* a pointer to an event */
        struct evlist *a;

        a = lget();            /* read a score as a list of events        */
        e = a->e[4];                    /* point to event 4 in event list a    */
        e->p[5] *= 2;                   /* find pfield 5 and multiply its value by 2 */
        lput(a);                        /* write out the list of events          */
        putstr("e");                    /* add a "score end" statement           */
}
```

Now consider the following score, in which p[5] contains frequency in cps.

```
f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
e
```

If this score were given to the preceding main program, the resulting output would look like this:

```
f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
i 1 1 3 0 440 10000
i 1 4 3 0 512 10000          ; p[5] has become 512 instead of 256.
i 1 7 3 0 880 10000
e
```

Note that the 4th event is in fact the second note of the score. So far we have not distinguished between notes and function table setup in a numeric score. Both can be classed as events. Also note that our *4th* event has been stored in e[4] of the structure. For compatibility with **Csound** pfield notation, we will ignore p[0] and e[0] of the event and list structures, storing p1 in p[1], event 1 in e[1], etc. The **Cscore** functions all adopt this convention.

As an extension to the above, we could decide to use *a* and *e* to examine each of the events in the list. Note that *e* has not preserved the numeral 4, but the contents of that slot. To inspect p5 of the previous listed event we need only redefine *e* with the assignment

```
e = a->e[3];
```

More generally, if we declare a new variable *f* to be a *pointer to a pointer* to an event, the statement

```
f = &a->e[4];
```

will set *f* to the address of the fourth event in the event list *a*, and **f* will signify the *contents* of the slot, namely the event pointer itself. The expression

```
(*f)->p[5],
```

like e->p[5], signifies the fifth pfield of the selected event. However, we can advance to the next slot in the evlist by advancing the pointer *f*. In C this is denoted by *f++*.

In the following program we will use the same input score. This time we will separate the ftable statements from the note statements. We will next write the three note-events stored in the list *a*, then create a second score section consisting of the original pitch set and a transposed version of itself. This will bring about an octave doubling.

By pointing the variable *f* to the first note-event and incrementing *f* inside a **while** block which iterates n times (the number of events in the list), one statement can be made to act upon the same pfield of each successive event.

```
#include <csound/cscore.h>
main()
{
          struct event *e,**f;              /* declarations. see pp.8-9 in the */
          struct evlist *a,*b;              /* C language programming manual */
          int n;

          a = lget();              /* read score into event list "a" */
          b = lsepf(a);                     /* separate f statements */
          lput(b);                          /* write f statements out to score */
          lrelev(b);                        /* and release the spaces used */
          e = defev("t 0 120");             /* define event for tempo statement */
          putev(e);                         /* write tempo statement to score */
          lput(a);                          /* write the notes */
          putstr("s");                      /* section end */
          putev(e);                         /* write tempo statement again */
          b = lcopyev(a);                   /* make a copy of the notes in "a" */
          n = lcount(b);                    /* and count the number copied */
          f = &a->e[1];
          while (n--)                       /* iterate the following line n times: */
             (*f++)->p[5] *= .5;            /*    transpose pitch down one octave */
          a = lcat(b,a);                    /* now add these notes to original pitches */
          lput(a);
          putstr("e");
}
```

The output of this program is:

```
f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
t 0 120
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
s
t 0 120
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
i 1 1 3 0 220 10000
i 1 4 3 0 128 10000
i 1 7 3 0 440 10000
e
```

Next we extend the above program by using the **while** statement to look at p[5] and p[6]. In the original score p[6] denotes amplitude. To create a diminuendo in the added lower octave, which is independent from the original set of notes, a variable called *dim* will be used.

```
#include <csound/cscore.h>
main()
{
          struct event *e,**f;
          struct evlist *a,*b;
```

```
        int n, dim;                     /* declare new variable as integer */

        a = lget();
        b = lsepf(a);
        lput(b);
        lrelev(b);
        e = defev("t 0 120");
        putev(e);
        lput(a);
        putstr("s");
        putev(e);                       /* write out another tempo statement */
        b = lcopyev(a);
        n = lcount(b);
        dim = 0;                        /* initialize dim to 0 */
        f = &a->e[1];
        while (n--){
            (*f)->p[6] -= dim;          /* subtract current value of dim */
            (*f++)->p[5] *= .5;         /* transpose, move f to next event */
            dim += 2000;                /* increase dim for each note */
        }
        a = lcat(b,a);
        lput(a);
        putstr("e");
}
```

The increment of *f* in the above programs has depended on certain precedence rules of **C.** Although this keeps the code tight, the practice can be dangerous for beginners. Incrementing may alternately be written as a separate statement to make it more clear.

```
        while (n--){
            (*f)->p[6] -= dim;
            (*f)->p[5] *= .5;
            dim += 2000;
            f++;
        }
```

Using the same input score again, the output from this program is:

```
f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
t 0 120
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
s
t 0 120
i 1 1 3 0 440 10000         ; Three original notes at
i 1 4 3 0 256 10000         ; beats 1,4 and 7 with no dim.
i 1 7 3 0 880 10000
i 1 1 3 0 220 10000         ; three notes transposed down one octave
i 1 4 3 0 128 8000                  ; also at beats 1,4 and 7 with dim.
i 1 7 3 0 440 6000
e
```

In the following program the same three-note sequence will be repeated at various time intervals. The starting time of each group is determined by the values of the array *cue*. This time the dim. will occur for each group of notes rather than each note. Note the position of the statement which increments the variable *dim* outside the inner **while** block.

```
#include <csound/cscore.h>

int cue[3]={0,10,17};                    /* declare array of 3 integers */

main()
{
        struct event *e, **f;
        struct evlist *a, *b;
        int n, dim, cuecount, holdn;     /* declare new variables */

        a = lget();
        b = lsepf(a);
        lput(b);
        lrelev(b);
        e = defev("t 0 120");
        putev(e);
        n = lcount(a);
        holdn = n;             /* hold the value of "n" to reset below */
        cuecount = 0;                    /* initilize cuecount to "0" */
        dim = 0;
        while (cuecount <= 2) {          /* count 3 iterations of inner "while" */
           f = &a->e[1];                 /* reset pointer to first event of list "a" */
           n = holdn;                    /* reset value of "n" to original note count */
           while (n--) {
                (*f)->p[6] -= dim;
                (*f)->p[2] += cue[cuecount];    /* add values of cue */
                f++;
           }
           printf("%s %d0, "; diagnostic:  cue=", cue[cuecount]);
           cuecount++;
           dim += 2000;
           lput(a);
        }
        putstr("e");
}
```

Here the inner **while** block looks at the events of list *a* (the notes) and the outer **while** block looks at each repetition of the events of list *a* (the pitch group repetitions). This program also demonstrates a useful trouble-shooting device with the **printf** function. The semi-colon is first in the character string to produce a comment statement in the resulting score file. In this case the value of *cue* is being printed in the output to insure that the program is taking the proper array member at the proper time. When output data is wrong or error messages are encountered, the **printf** function can help to pinpoint the problem.

Using the identical input file, the C program above will generate:

```
f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
t 0 120

; diagnostic:  cue= 0
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
```

```
i 1 7 3 0 880 10000

; diagnostic:  cue= 10
i 1 11 3 0 440 8000
i 1 14 3 0 256 8000
i 1 17 3 0 880 8000

; diagnostic:  cue= 17
i 1 28 3 0 440 4000
i 1 31 3 0 256 4000
i 1 34 3 0 880 4000
e
```

Further development of these scores will lead the composer to techniques of score manipulation which are similar to serial techniques of composition. Pitch sets may be altered with regard to any of the parameter fields. The programing allows for transpositions, time warping, pitch retrograding and dynamic controls, to name a few.

Compiling a Cscore program

A **Cscore** program named *example.c* can be compiled and linked with its library modules by the command:

```
$ cc example.c -lcscore
```

The resulting executable file is called "a.out". It is run by typing:

$ a.out (no input, output printed on the screen)

$ a.out < scorin (input score named *scorin*, output on screen)

$ a.out < scorin > scorout (input as above, output into a file)

6. SCOT: A Score Translator

Scot is a language for describing scores in a fashion that parallels traditional music notation. Scot is also the name of a program which translates scores written in this language into *standard numeric score* format so that the score can be performed by **Csound.** The result of this translation is placed in a file called *score.* A score file written in Scot (named *file.sc,* say) can be sent through the translator by the command

    scot file.sc

The resulting numeric score can then be examined for errors, edited, or performed by typing

    csound file.orc score

Alternatively, the command

    csound file.orc -S file.sc

would combine both processes by informing **Csound** of the initial score format.

Internally, a Scot score has at least three parts: a section to define instrument names, a section to define functions, and one or more actual score sections. It is generally advisable to keep your score sections short to facilitate finding errors. The overall layout of a Scot score has three main sections:

```
orchestra { .... }
functions { .... }
score { .... }
```

The last two sections may be repeated as many times as desired. The functions section is also optional. Throughout this **Scot** document, bear in mind that you are free to break up each of these divisions into as many lines as seem convenient, or to place a carriage return anywhere you are allowed to insert a space, including before and after the curly brackets. Furthermore, you may use as many spaces or tabs as you need to make the score easy to read. Scot imposes no formatting restrictions except that numbers, instrument names, and keywords (for example, *orchestra*) may not be broken with spaces. You may insert comments (such as measure numbers) anywhere in the score by preceding them with a semicolon. A semicolon causes Scot to ignore the rest of a line.

Orchestra Declaration Section

The orchestra section of a **Scot** score serves to designate instrument names for use within the score. This is a matter of convenience, since an orchestra knows instruments only by numbers, not names. If you declare three instruments, such as:

    orchestra { flute=1 cello=2 trumpet=3 }

Csound will neither know nor care what you have named the note lists. However, when you use the name *$flute,* Scot will know you are referring to **instr 1** in the orchestra, *$cello* will refer to **instr 2**, and *$trumpet* will be **instr 3**. You may meaningfully skip numbers or give several instruments the same number. It is up to you to make sure that your orchestra has the correct instruments and that the association between these names and the instruments is what you intend. There is no limit (or a very high one, at least) as to how many instruments you can declare.

Function Declaration Section

The major purpose of this division is to allow you to declare function tables for waveforms, envelopes, etc. These functions are declared exactly as specified for **Csound.** In fact, everything you type between the brackets in this section will be passed directly to the resulting *numeric* score with no modification, so that mistakes will not be caught by the Scot program, but rather by the subsequent performance. You can use this section to write notes for instruments for which traditional pitch-rhythm notation is inappropriate. The most common example would be turning on a reverb instrument. Instruments referenced in this way need not appear in the Scot orchestra declaration. Here is a possible function declaration:

```
functions {
f1 0 256 10 1 0 .5 0 .3
f2 0 256 7 0 64 1 64 .7 64 0
i9 0 -1 3                    ; this turns on instr 9
}
```

Score Section

The Scot statements contained inside the braces of each score statement is translated into a numeric score Section (q.v.). It is wise to keep score sections small, say seven or eight measures of five voices at a time. This avoids overloading the system, and simplifies error checking.

The beginning of the score section is specified by typing:

```
score {
```

Everything which follows this until the braces are closed is within a single section. Within this section you write measures of notes in traditional pitch and rhythm notation for any of the instrument names listed in your orchestra declaration. These notes may carry additional information such as slurs, ties and parameter fields. Let us now consider the format for notes entered in a Scot score.

The first thing to do is name the instrument you want and the desired meter. For example, to write some 4/4 measures for the cello, type:

```
$cello
!ti "4/4"
```

The dollar sign and exclamation point tell Scot that a special declarator follows. The time signature declarator is optional; if present, Scot will check the number of beats in each measure for you.

Pitch and Rhythm

The two basic components of a note statement are the pitch and duration. Pitch is specified using the alphabetic note name, and duration is specified using numeric characters. Duration is indicated at the beginning of the note as a number representing the division of a whole beat. You may always find the number specifying a given duration by thinking of how many times that duration would fit in a 4/4 measure. Also, if the duration is followed by a dot ('.') it is increased by 50%, exactly as in traditional notation. Some sample durations are listed below:

whole note	1
half note	2
double dotted quarter	4..
dotted quarter note	4.
quarter note	4
half note triplet	6
eighth note	8
eighth note triplet	12
sixteenth note	16
thirty-second note	32

Pitch is indicated next by first (optionally) specifying the register and then the note name, followed by an accidental if desired. Normally, the "octave following" feature is in effect. This feature causes any note named to lie within the interval of an augmented fourth of the previous note, unless a new register is chosen. The first note you write will always be within a fourth of middle c unless you choose a different register.

For example, if the first note of an instrument part is notated g flat, the scot program assigns the pitch corresponding to the g flat below middle c. On the other hand, if the first note is f sharp, the pitch assigned will be the f sharp above middle c. Changes of register are indicated by a preceding apostrophe for each octave displacement upward or a preceding comma for each octave displacement downward. Commas and apostrophes always displace the pitch by the desired number of octaves starting from that note which is within an augmented fourth of the previous pitch.

If you ever get lost, prefacing the pitch specification with an '=' returns the reference to middle c. It is usually wise to use the equals sign in your first note statement and whenever you feel uncertain as to what the current registration is. Let us now write two measures for the cello part, the first starting in the octave below middle c and the second repeating but starting in the octave above middle c:

```
$cello
!ti "4/4"
4=g 4e 4d 4c/ 4='g 4e 4d 4c
```

As you can see, a slash indicates a new measure and we have chosen to use the dummy middle c to indicate the new register. A more convenient way of notating these two measures would be to type the following:

```
$cello
!ti "4/4"
4=g e d c/ ''g e d c
```

You may observe in this example that the quarter note duration carries to the following notes when the following durations are left unspecified. Also, two apostrophes indicate an upward pitch displacement of two octaves from two g's below middle c, where the pitch would have fallen without any modification. It is important to remember three things, then, when specifying pitches:

1) Note pitches specified by letter name only (with or without accidental) will always fall within an interval of a fourth from the preceding pitch.

2) These pitches can be octave displaced upward or downward by preceding the note letter with the desired number of apostrophes or commas.

3) If you are unsure of the current register, you may begin the pitch component of the note with an equals sign which acts as a dummy middle c.

The pitch may be modified by an accidental after the note name:

n	natural
#	sharp
- (hyphen)	flat
##	double sharp
-- (double hyphen)	double flat

Accidentals are carried throughout the measure just as in traditional music notation. However, an accidental specified within a measure will hold for that note in all registers, in contrast with traditional notation. Therefore, make sure to specify *n* when you no longer want an accidental applied to that pitch-class.

Pitches entered in the Scot score are translated into the appropriate octave point pitch-class value and appear as parameter p5 in the numeric score output. This means you must design your instruments to accept p5 as pitch.

Rests are notated just like notes but using the letter *r* instead of a pitch name. *4r* therefore indicates a quarter rest and *1r* a whole rest. Durations carry from rest to rest or to following pitches as mentioned above.

The tempo in beats per minute is specified in each section by choosing a single instrument part and using tempo statements (e.g. t90) at the various points in the score as needed. A quarter note is interpreted as a single beat, and tempi are interpolated between the intervening beats (see score **t** statement).

Scot Example I

```
; A BASIC Tune
orchestra    { guitar=1 bass=2 }
functions  {
f1 0 512 10 1 .5 .25 .126
f2 0 256  7 1 120 1 8 0 128 1
}
score {  ;section 1
$guitar
!ti "4/4"
4=c 8d e- f r 4='c/
8.b- 16a a- g  g- f  4e- c/
$bass
2=,,c 'a-/
g  =,c/
}
score {  ;section 2
$guitar
!ti "4/4"
6='c r c 4..c## 16e- /
6f r f 4..f## 16b /
$bass
4=,,c 'g ,c 'g/
2=a-  g /
}
```

The score resulting from this Scot notation is shown at the end of this chapter.

Groupettes

Duration numbers can have any integral value; for instance,

```
!time "4/4"
5cdefg/
```

would encode a measure of *5 in the time of 4* quarter notes. However, specification of arbitrary rhythmic groupings in this way is at best awkward. Instead, arbitrary portions of the score section may be enclosed in *groupette brackets.* The durations of all notes inside groupette brackets will be multiplied by a fraction *n*/*d*, where the musical meaning is *d* in the time of *n.* Assuming *d* and *n* here are integers, groupette brackets may take these several forms:

```
{d:n:    .....    :}    d in the time of n
{d::     .....    :}    n will be the largest power of 2 less than d
{:       .....    :}    d=3, n=2 (normal triplets)
```

It can be seen that the second and third form are abbreviations for the more common kinds of groupettes.

(Observe the punctuation of each form carefully.) Groupettes may be nested to a reasonable depth. Also, groupette factors apply only after the written duration is carried from note to note. Thus, the following example is a correct specification for two measures of 6/8 time:

```
!time "6/8" 8cde {4:3: fgab :} / crc 4.c /
```

The notes inside the groupette are *4 in the space of 3* 8th notes, and the written-8th-note duration carries normally into the next measure. This closely parallels the way groupette brackets and note durations interact in standard notation.

Slurs and Ties

Now that you understand part writing in the Scot language, we can start discussing more elaborate features. Immediately following the pitch specification of each note, one may indicate a slur or a tie into the next note (assuming there is one), but not both simultaneously. The slur is typed as a single underscore ('_') and a tie as a double underscore ('__'). Despite the surface similarity, there is a substantial difference in the effect of these modifiers.

For purposes of Scot, tied notes are notes which, although comprised of several graphic symbols, represent only a single musical event. (Tied notes are necessary in standard music notation for several reasons, the most common being notes which span a measure line and notes with durations not specifiable with a single symbol, such as quarter note tied to a sixteenth.) Notes which are tied together are summed by duration and output by Scot as a single event. This means you cannot, for example, change the parameters of a note in the middle of a tie (see below). Two or more notes may be tied together, as in the following example, which plays an f# for eleven beats:

```
!ti "4/4"
1 f#__ / 1 f#__ / 2. f# 4r /
```

By contrast, slurred notes are treated as distinct notes at the **Csound** level, and may be of arbitrary pitch. The presence of a slur is reflected in parameter p4, but the slur has no other meaning beyond the interpretation of p4 by your instrument. Since instrument design is beyond the scope of this manual, it will suffice for now to explain that the Scot program gives sets p4 to one of four values depending on the existence of a slur before and after the note in question. This means Scot pays attention not only to the slur attached to a given note, but whether the preceding note specified a slur. The four possibilities are as follows, where the p4 values are taken to apply to the note 'd':

4c d	(no slur)	p4 = 0
4c d_	(slur after only)	p4 = 1
4c_ d	(slur before only)	p4 = 2
4c_ d_	(before & after)	p4 = 3

Parameters

The information contained in the Scot score notation we have considered so far is manifested in the output score in parameters p1 through p5 in the following way:

p1: instrument number
p2: initialization time of instrument
p3: duration
p4: slur information
p5: pitch information in octave point pitch-class notation

Any additional parameters you may want to enter are listed in brackets as the last part of a note specification. These parameters start with p6 and are separated from each other with spaces. Any parameters not specified for a particular note have their value carried from the most recently specified value. You may choose to change some parameters and not others, in which case you can type a dot ('.') for parameters whose values don't change, and new values for those that do. Alternatively, the construction *N:*, where *N* is an integer, may be used to indicate that the following parameter specifications apply to successive parameters starting with parameter *N*. For example:

```
4e[15000 3 4 12:100 150] g_ d_[10000 . 5]     c
```

Here, for the first two quarter notes p6, p7, p8, p12, and p13 respectively assume the values 15000, 3, 4, 100, and 150. The values of p9 through p11 are either unchanged, or implicitly zero if they have never been specified in the current section. On the third quarter note, the value of p6 is changed to 10000, and the value of p8 is changed to 5. All others are unchanged.

Normally, parameter values are treated as globals -- that is, a value specification will carry to successive notes if no new value is specified. However, if a parameter specification begins with an apostrophe, the value applies locally to the current note only, and will not carry to successive notes either horizontally or vertically (see *divisi* below).

Pfield Macros

Scot has a macro text substitution facility which operates only on the pfield specification text within brackets. It allows control values to be specified symbolically rather than numerically. Macro definitions appear inside brackets in the orchestra section. A single bracketed list of macro definitions preceding the first instrument declaration defines macros which apply to all instruments. An additional bracketed list of definitions may follow each instrument to specify macros that apply to that particular instrument.

```
orchestra {
        [ pp=2000 p=4000 mp=7000 mf=10000 f=20000 ff=30000
          modi = 11: w = 1 x = 2 y = 3 z = 4
          vib = "10:1 " novib = "10:0 1"
        ]
violin = 1 [ pizz = " 20:1" arco = " 20:0" ]
serpent = 3          [ ff = 25000 sfz = 'f sffz = 'ff]
}
score {
 $violin    =4c[mf modi z.y novib] d e a['f vib3] /
                            8 b[pizz]c 4d[f] 2c[ff arco] /
 $serpent  =,4.c[mp modi y.x] 8b 2c /
                            'g[f ] ,c[ff] /
}
```

As can be seen from this example, a macro definition consists of the macro name, which is a string of alphabetic characters, followed by an equal sign, followed by the macro value. As usual, spaces, tabs, and newlines may be used freely. The macro value may contain arbitrary characters, and may be quoted if spacing characters need to be included.

When a macro name is encountered in bracketed pfield lists in a score section, that name is replaced with the macro text with no additional punctuation supplied. The macro text may itself invoke other macros, although it is a serious error for a macro to contain itself, directly or indirectly. Since macro names are identified as strings of alphabetic characters, and no additional spaces are provided when a macro is expanded, macros may easily perform such concatenations as found in the first serpent note above, where the integer and fractional parts of a single pfield are constructed. Also, a macro may do no more than define a symbolic pfield, as in the definition of modi. The primary intention of macros is in fact not only to reduce the number of characters required, but also to enable symbolic definitions of parameter numbers and parameter values. For instance, a particular instrument's interpretation of the dynamic *ff* can be changed merely by changing a macro value, rather than changing all occurrences of that particular value in the score.

Divisi

Notes may be stacked on top of each other by using a back arrow ('<') between the notes of the divisi. Each time Scot encounters a back arrow, it understands that the following note is to start at the same time as the note to the left of the back arrow. Durations, accidentals and parameters carry from left to right through the divisi. Each time these are given new values, the notes to the right of the back arrows also take on the new values unless they are specified again.

When writing divisi you can stack compound events by enclosing them in parentheses. Also, divisi which occur at the end of the measure must have the proper durations or the scot program will mis-interpret the measure duration length.

Scot Example II

```
orchestra { right=1 left=2 }
functions { f1 0 256 10 1}
score {
$right !key "-b"
; since p5 is pitch, p7 is set to the pitch of next note
!ti "2/4"
!next p5 "p7"      ;since p5 is pitch, p7 refers to pitch of next note
!next p6 "p8"      ;If p6 is vol, say, then p8 refers to vol of next note
t90
8r c[3 np5]<e<='g r c<f<a / t90 r d-<g<b r =c[5]<f<a__ /
!ti "4/4"
t80
4d_<f__<(8a g__) 4c<(8fe)<4g 4.c<f<f 8r/

$left  !key "-b"
!next p5 "p7"
!next p6 "p8"
!ti "2/4"
8=,c[3 np5] r f r/ e r f r/
!ti "4/4"
2b_[5]<(4=,b_c) 4.a<f 8r/
}
```

Notice in this example that the tempo statements occurred in instrument ‘right’ only. Also, all notes had p6=3 until the third measure, at which point p6 took on the value 5 for all notes. The next parameter option used is described under Additional Features. The output score of above is given at the end.

Additional Features

Several options can be included in any of the individual instrument parts within a section. A sample statement follows the description of each option. The keyword which follows the ‘!’ in these statements may be abbreviated to the first two characters.

Key Signatures

Any desired key signature is specified by listing the accidentals as they occur in a key signature statement. Thereafter, all notes of that instrument part are sharpened or flattened accordingly. For example, for the key of D, type

```
!key "#fc"
```

Accidental Following

Accidental following may be turned on or off as needed. When turned off, accidentals no longer carry throughout the measure as in traditional notation. This convention is sometimes used in contemporary scores.

```
!accidentals "off"
```

Octave Following

Turning off octave following indicates that pitches stay in the same absolute octave register until explicitly moved. An absolute octave starts at pitch c and ends at the b above it. The octave middle-c-to-b is indicated with an equals sign (‘=’) and octave displacement is indicated with the appropriate number of commas or apostrophes. These displacements are cummulative. For example,

```
!octaves "off"
4='c g b 'c
```

starts at the c above middle c and ends at two c’s above middle c.

Vertical Following

Turning off vertical following means that durations, register, and parameters only carry horizontally from note to note and not vertically as described in the section on divisi.

```
!vertical "off"
```

Transposition

Any instrument part can be transposed to another key by indicating the intervalic difference between the notated key and the desired key. This difference is always taken with reference to middle c - to transpose a whole step upward, for example, type

```
!transpose "d"
```

This indicates that the part is transposed by the interval difference between middle c and d.

Next-value and Previous-value Parameteres

In order to play a note, it is sometimes necessary for an instrument to know what value one or more parameters will have for the next note. For instance, an instrument might be designed which glisses during the last portion of its performance (perhaps only when a slur is indicated) from its written pitch to the pitch of the next note. This can only be done, of course, if the instrument can know what the pitch of the next note will be. The necessary information can be provided using next-value parameters. A next value parameter might be declared by

```
!next p5 "p6"
```

which is interpreted to mean that for the current instrument, p6 will contain the next note’s p5 value. This holds true globally for all occurences of this instrument until further modifications. If for any reason you wish to override this value, p6 may be filled in explicitly. This is sometimes useful for the last note of a section, for which p6 will otherwise assume the p5 value for the current note. The *next-value* feature is illustrated in the Scot example II.

The necessary information may also be provided using *standard numeric score* next-value parameters. A parameter argument containing the symbol npx (where x is an integer) will substitute parameter number x of the following note for that instrument. Similarly, the value of a parameter occurring during the previous note may be referenced with the symbol ppx (where x is an integer). Details of the next- and previous-value parameter feature may be found in the **Numeric Scores** section.

Pfields containing the symbol < will be replaced by values derived from linear interpolation of a time-based ramp. Ramp endpoints are defined by the first real number found in the same pfield of a preceding and following note played by the same instrument. Details of the ramping feature are likewise found in the **Numeric Scores** section.

f0 Statements

In each score section, Scot automatically produces an f0 statement with a p2 equal to the ending time of the last note or rest in the section. Thus, ‘dead time’ at the end of a section for reverberation decay or whatever purpose may be specified musically by rests in one or more parts. See the eighth rest at the end of Scot example II and its output score shown below.

Output Scores

Output file *score* from Scot Example I.

```
f1 0 512 10 1 .5 .25 .126
f2 0 256  7 1 120 1 8 0 128 1
i1.01 0 1 0 8.00
i1.01 1 0.5 0 8.02
i1.01 1.5 0.5 0 8.03
i1.01 2 0.5 0 8.05
i1.01 3 1 0 9.00
i1.01 4 0.75 0 8.10
i1.01 4.75 0.25 0 8.09
i1.01 5 0.25 0 8.08
i1.01 5.25 0.25 0 8.07
i1.01 5.5 0.25 0 8.06
i1.01 5.75 0.25 0 8.05
i1.01 6 1 0 8.03
i1.01 7 1 0 8.00
i2.01 0 2 0 6.00
i2.01 2 2 0 6.08
i2.01 4 2 0 6.07
i2.01 6 2 0 7.00
t0 60
f0 8
s
i1.01 0 0.6667 0 9.00
i1.01 1.3333 0.6667 0 9.00
i1.01 2 1.75 0 9.02
i1.01 3.75 0.25 0 9.03
i1.01 4 0.6667 0 9.05
i1.01 5.3333 0.6667 0 9.05
i1.01 6 1.75 0 9.07
i1.01 7.75 0.25 0 9.09
i2.01 0 1 0 6.00
i2.01 1 1 0 6.07
i2.01 2 1 0 6.00
i2.01 3 1 0 6.07
i2.01 4 2 0 7.08
i2.01 6 2 0 7.07
t0 60
f0 8
s
```

Output file *score* from Scot Example II.

```
f1 0 256 10 1
c r1 n 7 5
c r1 n 8 6
i1.01 0.5000 0.5000 0 8.00 3 8.00 3
i1.02 0.5000 0.5000 0 8.04 3 8.05 3
i1.03 0.5000 0.5000 0 8.07 3 8.09 3
i1.01 1.5000 0.5000 0 8.00 3 8.01 3
i1.02 1.5000 0.5000 0 8.05 3 8.07 3
i1.03 1.5000 0.5000 0 8.09 3 8.10 3
```

```
i1.01 2.5000 0.5000 0 8.01 3 8.00 5
i1.02 2.5000 0.5000 0 8.07 3 8.05 5
i1.03 2.5000 0.5000 0 8.10 3 8.09 5
i1.01 3.5000 0.5000 0 8.00 5 8.02 5
i1.02 3.5000 0.5000 0 8.05 5 8.05 5
i1.01 4.0000 1.0000 1 8.02 5 8.00 5
i1.03 3.5000 1.0000 0 8.09 5 8.07 5
i1.01 5.0000 1.0000 2 8.00 5 8.00 5
i1.02 4.0000 1.5000 0 8.05 5 8.04 5
i1.02 5.5000 0.5000 0 8.04 5 8.05 5
i1.03 4.5000 1.5000 0 8.07 5 8.05 5
i1.01 6.0000 1.5000 0 8.00 5 8.00 5
i1.02 6.0000 1.5000 0 8.05 5 8.05 5
i1.03 6.0000 1.5000 0 8.05 5 8.05 5
c r2 n 7 5
c r2 n 8 6
i2.01 0.0000 0.5000 0 7.00 3 7.05 3
i2.01 1.0000 0.5000 0 7.05 3 7.04 3
i2.01 2.0000 0.5000 0 7.04 3 7.05 3
i2.01 3.0000 0.5000 0 7.05 3 7.10 5
i2.01 4.0000 2.0000 1 7.10 5 7.09 5
i2.02 4.0000 1.0000 1 6.10 5 7.00 5
i2.02 5.0000 1.0000 2 7.00 5 7.05 5
i2.01 6.0000 1.5000 2 7.09 5 7.09 5
i2.02 6.0000 1.5000 0 7.05 5 7.05 5
t0 60 0.0000 90.0000 2.0000 90.0000 4.0000 80.0000 4.0000 90.0000
f0 8.0000
s
e
```

7. The CSOUND Command

csound is a command for passing an orchestra file and score file to **Csound** to generate a soundfile. The score file can be in one of many different formats, according to user preference. Translation, sorting, and formatting into orchestra-readable numeric text is handled by various preprocessors; all or part of the score is then sent on to the orchestra. Orchestra performance is influenced by command flags, which set the level of displays and console reports, specify I/O filenames and sample formats, and declare the nature of realtime sensing and control.

The format of a command is:

csound [-flags] orchname scorename

where the arguments are of 2 types: *flag* arguments (beginning with a "-"), and *name* arguments (such as filenames). Certain flag arguments take a following name or numeric argument. The available flags are:

-I, -n	sound output inhibitors
-iName, -oName	sound I/O filenames
-bNumb, -h	audio buffer & header control
-c, -a, -u, -s, -l, -f	audio output formats
-v, -mNumb, -d, -g	message & display levels
-S, -xName	score formats
-B, -RName, -MName	realtime event control

Flags may appear anywhere in the command line, either separately or bundled together. A flag taking a Name or Number will find it in that argument, or in the immediately subsequent one. The following are thus equivalent commands:

csound -nm3 orchname -Sxxfilename scorename
csound -n -m 3 orchname -x xfilename -S scorename

All flags and names are optional. The default values are:

csound -s -otest -b iobufsamps -m7 orchname scorename

where *iobufsamps* is a locally defined value (see below),

and *orchname* is a file containing Csound orchestra code,

and *scorename* is a file of score data in standard numeric score format; if omitted,
csound will use the previously processed *score.srt in the current directory.*

Csound reports on the various stages of score and orchestra processing as it goes, doing various syntax and error checks along the way. Once the actual performance has begun, any error messages will derive from either the instrument loader or the unit generators themselves. A **csound** command may include any rational combination of the following flag arguments:

csound -I
I-time only. Allocate and initialize all instruments as per the score, but skip all P-time processing (no k-signals or a-signals, and thus no amplitudes and no sound). Provides a fast validity check of the score pfields and orchestra i-variables.

csound -n
Nosound. Do all processing, but bypass writing of sound to disk. This flag does not change the execution in any

other way.

csound -i isfname
Input soundfile name (interpreted as for output names, below). The name *stdin* will cause audio to be read from standard input.

csound -o osfname
Output soundfile name. Sound is written to the file *osfame* in the user soundfile directory (set by the environment variable SFDIR). If no name is given, the default name will be *test*. If *osfname* is of the form ./*name* or /*path*/*name*, sound will be written into the current or specified directory. If SFREMOTE is enabled at Csound installation, and *osfname* is of the form *machine:*/*path*/*name*, sound will be written to the remote machine. The name *stdout* will cause audio to be written to standard output.

csound -b Numb
Number of audio samples per soundio buffer. Typically 512. A larger number is more efficient, but small numbers reduce the delay in realtime performances.

csound -h
No header on output soundfile. Don't write a file header, just binary samples.

csound {-c, -a, -u, -s, -l, -f}
Audio sample format of the output soundfile. One of:

c	8-bit signed character
a	8-bit a-law
u	8-bit u-law
s	short integer
l	long integer
f	single-precision float (not playable, but can be read by -i, **soundin** and GEN01)

csound -v
Verbose translate and run. Prints details of orch translation and performance, enabling errors to be more clearly located.

csound -m Numb
Message level for standard (terminal) output. Takes the *sum* of 3 print control flags, turned on by the following values: 1 = note amplitude messages, 2 = samples out of range message, 4 = warning messages. The default value is *m7* (all messages on).

csound -d
Suppress all displays.

csound -g
Recast graphic displays into *ascii* characters, suitable for any terminal.

csound -S
Interpret *scorename* as a Scot file and create a standard score file (named "score") from it, then sort and perform that.

csound -x xfile
Extract a portion of the sorted score *score.srt*, according to *xfile* (see extract below).

csound -B
Use the uninterpreted Beats of *score.srt* for this performance. In this mode, the *beat rate* of score performance is controllable from within the orchestra (see the **tempo** unit).

csound -R devname
Read Realtime Line-oriented score events from device *devname.*

csound -M devname
Read MIDI events from device *devname.*

The EXTRACT feature:

This feature will **extract** a segment of a *sorted numeric score* file according to instructions taken from a control file. The control file contains an instrument list and two time points, *from* and *to,* in the form:

instruments 1 2   from 1:27.5   to 2:2

The component labels may be abbreviated as *i, f* and *t.* The time points denote the begininng and end of the extract in terms of:

[section no.] : [beat no.].

Each of the three parts is also optional. The default values for missing *i, f* or *t* are:

all instruments,   begining of score,   end of score.

extract reads an orchestra-readable score file and produces an orchestra-readable result. Comments, tabs and extra spaces are flushed, **w** and **a** statements are added and an **f0** reflecting the extract length is appended to the output. Following an **extract** process, the abbreviated score will contain all function table statements, together with just those note statements that occur in the *from-to* interval specified. Notes lying completely in the interval will be unmodified; notes that lie only partly within will their p3 durations truncated as necessary.

Independent Preprocessing:

Although the result of all score preprocessing is retained in the file *score.srt* after orchestra performance (it exists as soon as score preprocessing has completed), the user may sometimes want to run these phases independently. The command

scot   filename

will process the **Scot** formatted *filename,* and leave a *standard numeric score* result in a file named *score* for perusal or later processing;

scscort < infile   > outfile

will put a numeric score *infile* through Carry, Tempo, and Sort preprocessing, leaving the result in *outfile.*

Likewise **extract,** also normally invoked as part of the **csound** command, can be invoked as a standalone program:

extract xfile < score.sort   > score.extract

This command expects an already sorted score. An unsorted score should first be sent through *scsort* then piped to the extract program:

scsort < scorefile | extract xfile > score.extract

Appendix 1. An Orchestra QUICK REFERENCE

VALUE CONVERTERS

ftlen(x)	(init-rate args only)
int(x)	(init- or control-rate args only)
frac(x)	" "
dbamp(x)	" "
i(x)	(control-rate args only)
abs(x)	(no rate restriction)
exp(x)	" "
log(x)	" "
sqrt(x)	" "
sin(x)	" "
cos(x)	" "
ampdb(x)	" "

PITCH CONVERTERS

octpch(pch)	(init- or control-rate args only)
pchoct(oct)	" "
cpspch(pch)	" "
octcps(cps)	" "
cpsoct(oct)	(no rate restriction)

PROGRAM CONTROL

igoto	label
tigoto	label
kgoto	label
goto	label
if	ia R ib **igoto** label
if	ka R kb **kgoto** label
if	ia R ib **goto** label
timout	istrt, idur, label

SIGNAL GENERATORS

kr	**line**	ia, idur1, ib
ar	**line**	ia, idur1, ib
kr	**expon**	ia, idur1, ib
ar	**expon**	ia, idur1, ib
kr	**linseg**	ia, idur1, ib[, idur2, ic[...]]
ar	**linseg**	ia, idur1, ib[, idur2, ic[...]]
kr	**expseg**	ia, idur1, ib[, idur2, ic[...]]
ar	**expseg**	ia, idur1, ib[, idur2, ic[...]]
kr	**phasor**	kcps[, iphs]
ar	**phasor**	xcps[, iphs]

ir	**table**	indx, ifn[, ixmode][, ixoff][, iwrap]
ir	**tablei**	indx, ifn[, ixmode][, ixoff][, iwrap]
kr	**table**	kndx, ifn[, ixmode][, ixoff][, iwrap]
kr	**tablei**	kndx, ifn[, ixmode][, ixoff][, iwrap]
ar	**table**	andx, ifn[, ixmode][, ixoff][, iwrap]
ar	**tablei**	andx, ifn[, ixmode][, ixoff][, iwrap]
kr	**oscil1**	idel, kamp, idur, ifn
kr	**oscil1i**	idel, kamp, idur, ifn
kr	**oscil**	kamp, kcps, ifn[, iphs]
kr	**oscili**	kamp, kcps, ifn[, iphs]
ar	**oscil**	xamp, xcps, ifn[, iphs]
ar	**oscili**	xamp, xcps, ifn[, iphs]
ar	**foscil**	xamp, kcps, kcar, kmod, kndx, ifn[, iphs]
ar	**foscili**	xamp, kcps, kcar, kmod, kndx, ifn[, iphs]
ar	**buzz**	xamp, xcps, knh, ifn[, iphs]
ar	**gbuzz**	xamp, xcps, knh, klh, kr, ifn[, iphs]
ar	**adsyn**	kamod, kfmod, ifilno
ar	**pvoc**	ktimpnt, kfmod, ispecwp, ifilno
ar	**fof**	xamp, xfund, xforma, xformb, koct, ktex, kdebat, katt, iolaps, ifna, ifnb, idur[, iphs][, icor]
ar	**pluck**	kamp, kcps, icps, ifn, imeth [, iparm1, iparm2]
kr	**rand**	xamp[, iseed]
kr	**randh**	kamp, kcps[, iseed]
kr	**randi**	kamp, kcps[, iseed]
ar	**rand**	xamp[, iseed]
ar	**randh**	xamp, xcps[, iseed]
ar	**randi**	xamp, xcps[, iseed]

SIGNAL MODIFIERS

kr	**linen**	kamp, irise, idur, idec
ar	**linen**	xamp, irise, idur, idec
kr	**envlpx**	kamp, irise, idur, idec, ifn, iatss, iatdec[, ixmod]
ar	**envlpx**	xamp, irise, idur, idec, ifn, iatss, iatdec[, ixmod]
kr	**port**	ksig, ihtim[, isig]
ar	**tone**	asig, khp[, istor]
ar	**atone**	asig, khp[, istor]
ar	**reson**	asig, kcf, kbw[, iscl, istor]
ar	**areson**	asig, kcf, kbw[, iscl, istor]
krmsr,krmso,kerr,kcps	**lpread**	ktimpnt, ifilno[, inpoles][, ifrmrate]
ar	**lpreson**	asig
ar	**lpfreson**	asig, kfrqratio
kr	**rms**	asig[, ihp, istor]
ar	**gain**	asig, krms[, ihp, istor]
ar	**balance**	asig, acomp[, ihp, istor]

kr	**downsamp**	asig[, iwlen]
ar	**upsamp**	ksig
ar	**interp**	ksig[, istor]
kr	**integ**	ksig[, istor]
ar	**integ**	asig[, istor]
kr	**diff**	ksig[, istor]
ar	**diff**	asig[, istor]
kr	**samphold**	xsig, kgate[, ival, ivstor]
ar	**samphold**	asig, xgate[, ival, ivstor]
ar	**delayr**	idlt[, istor]
	delayw	asig
ar	**delay**	asig, idlt[, istor]
ar	**delay1**	asig[, istor]
ar	**deltap**	kdlt
ar	**deltapi**	xdlt
ar	**comb**	asig, krvt, ilpt[, istor]
ar	**alpass**	asig, krvt, ilpt[, istor]
ar	**reverb**	asig, krvt[, istor]

OPERATIONS USING SPECTRAL DATA TYPES

dsig	**octdown**	asig, iocts, isamps[, idisprd]
wsig	**noctdft**	dsig, iprd, ifrqs, iq[, ihann, idbout, idsines]
wsig	**specscal**	wsigin, ifscale, ifthresh
wsig	**specaddm**	wsig1, wsig2[, imul2]
wsig	**specdiff**	wsigin
wsig	**specfilt**	wsigin, ifhtim
	specdisp	wsig, iprd[, iwtflg]
ksum	**specsum**	wsig[, interp]

SENSING & CONTROL

ktemp	**tempest**	kin, iprd, imindur, imemdur, ihp, ithresh, ihtim, ixfdbak, istartempo, ifn[, idisprd, itweek]
kx, ky	**xyin**	iprd, ixmin, ixmax, iymin, iymax[, ixinit, iyinit]
	tempo	ktempo, istartempo

SOUND INPUT & OUTPUT

a1	**in**	
a1, a2	**ins**	
a1, a2, a3, a4	**inq**	
a1	**soundin**	ifilno[, iskptim][, iformat]
a1, a2	**soundin**	ifilno[, iskptim][, iformat]
a1, a2, a3, a4	**soundin**	ifilno[, iskptim][, iformat]
	out	asig
	outs1	asig
	outs2	asig
	outs	asig1, asig2
	outq1	asig
	outq2	asig

	outq3	asig
	outq4	asig
	outq	asig1, asig2, asig3, asig4
a1, a2, a3, a4	**pan**	asig, kx, ky, ifn[, imode][, ioffset]

SIGNAL DISPLAY

	print	iarg[, iarg, ...]
	display	xsig, iprd[, iwtflg]
	dispfft	asig, iprd, iwsiz[, iwtyp][, idbout][, iwtflg]