MAS.110 Fundamentals of Computational Media Design
Fall 2008

PART TWO: THE CSOUND ENGINE

Barry Vercoe

MIT Media Lab

2008

CONTENTS

2. STANDARD NUMERIC SCORE

3. GEN ROUTINES

4. SCOT: A Score Translator

5. Running from MIDI Data

1. 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*).

ORCHESTRA 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 normally begin with the letter **i, k**, **a** or **w**. 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 the above 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).

There are also three special symbols made from **p**, **c**, or **v**, followed by a number (e.g. p3, c10, v5). These are normally local to an instrument, although numbered **gv**'s can also be defined.

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

type	when renewable	**Local**	**Global**
reserved symbols	permanent	--	rsymbol
score parameter fields	I-time	**p**number	--
v-set symbols	I-time	**v**number	**gv**number
init variables	I-time	**i**name	**gi**name
midi controllers	any time	**c**number	--
control signals	P-time, K-rate	**k**name	**gk**name
audio signals	P-time, A-rate	**a**name	**ga**name
spectral data types	K-rate	**w**name	--

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

VALUE CONVERTERS

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

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.

int(x) returns the integer part of *x*.

frac(x) returns the fractional part of *x*.

dbamp(x) returns the decibel equivalent of the raw amplitude *x*.

i(x) returns 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) returns the absolute value of *x*.

exp(x) returns *e* raised to the *x*th power.

log(x) returns the natural log of *x* (*x* positive only).

sqrt(x) returns the square root of *x* (*x* non-negative).

sin(x) returns the sine of *x* (*x* in radians).

cos(x) returns the cosine of *x* (*x* in radians).

ampdb(x) returns the 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.

twopwr(x) returns 2.00 raised to the *x*th power.
Thus 0 gives 1.00, 1.00 gives 2.00, 3.00 gives 8.00,
0.5 gives root 2, -1.00 gives 0.5, and –0.5 gives 1 / (root 2).

rnd(x)	(init- or control-rate args only)
birnd(x)	(init- or control-rate args only)
ftlen(x)	(init-rate args only)
ftlptim(x)*	(init-rate args only)

where the argument within the parentheses may be an expression.

These value converters sample a global random sequence or return information about a stored function table. The result can be a term in a further expression.

rnd(x) returns a random number in the unipolar range 0 to x.

birnd(x) returns a random number in the bipolar range -x to x.
These units obtain values from a global psuedo-random number generator, then scale them into the requested range. The single global generator will thus distribute its sequence to these units throughout the performance, in whatever order the requests arrive.

ftlen(x) returns the size (no. of points, excl. guard point) of stored function table no. *x*.
While most units referencing a stored table will automatically take its size into account (so tables can be of arbitrary length), this function reports the actual size if that is needed.

ftlptim(x) returns the loop segment start-time (in seconds) of stored function table no. *x*.
This reports the duration of the direct recorded attack and decay parts of a sound sample, prior to its looped segment. Returns zero (and a warning message) if the sample does not contain loop points.

PITCH CONVERTERS

octnot(not)*	(init- or control-rate args only)
cpsnot(not)*	(init- or control-rate args only)
octpch(pch)	(init- or control-rate args only)
pchoct(oct)	(init- or control-rate args only)
cpspch(pch)	(init- or control-rate args only)
octcps(cps)	(init- or control-rate args only)
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
MIDI note number (0 - 127)	**not**
octave point pitch-class (8ve.pc)	**pch**
octave point decimal	**oct**
cycles per second	**cps**

The **not** form follows the MIDI convention (Middle C = 60, C# = 61, D = 62), but with fractional parts indicating microtones (62.5 is the next quarter-tone). The **pch** and **oct** 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**), 69 (**not**), or 7.21 (**pch**), etc. Microtonal divisions of the pch semitone can be encoded by using more than two decimal places.

Although negative **cps** values are not meaningful, negative **not**, **pch**, and **oct** are legal, giving small or fractional **cps** values under conversion.

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 cps (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 **not**, **pch** or **oct** into **cps** is not a linear operation but involves an exponential process that could be time-consuming when executed repeatedly (especially at audio rates). **Csound** uses a built-in table lookup to do this efficiently.

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 '= ='.) **NB**.: 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 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. For example, in

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

the 100/12 would be evaluated at orch init, the p5 expressions evaluated at note I-time, and the remainder of the expression evaluated every k-period. The whole might occur in a unit generator argument position, or be part of an assignment statement .

DIRECTORIES and FILES:

Many generators and the **csound** command itself specify *filenames* to be read from or written to. These are optionally *full pathnames*, whose target directory is fully specified. When not fullpath, filenames are sought in several directories in order, depending on their type and on the setting of certain *environment variables*. The latter are optional, but they can serve to partition and organize the directories so that source files can be shared rather than duplicated in several user directories. The environment variables can define directories for soundfiles (SFDIR), sound samples (SSDIR), and sound analysis (SADIR). The search order is:

Soundfiles being written are placed in SFDIR (if it exists), else the current directory.
Soundfiles for reading are sought in the current directory, then SSDIR, then SFDIR.
Analysis control files for reading are sought in the current directory, then SADIR.

NOMENCLATURE:

In **Csound** there are ten statement types, each of which provides a heading for the descriptive sections that follow in this chapter:

assignment statements	midi converter statements
orchestra header statements	signal generator statements
instrument block statements	signal modifier statements
program control statements	signal input and output
duration control statements	signal display statements

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 Perftime (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 *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 to *n3*. **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:

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

gi1     =       sr/2.
ga      init    0
gitranspose =   octpch(.0l)
```

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. The maximum instrument number is currently 200.

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).

GENERAL 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:   expon     440, p3/10,880       ; and expon
          riretum                        ; then resume perf
```

turnon insno[, itime]
ihold
turnoff
maxalloc* imax[, imethod]

Activate an instrument, or cause it to modify its own duration or density.

INITIALIZATION

itime (optional) - time in seconds until actual turnon. The default value is zero.

imax - maximum number of copies that can be allocated.

imethod (optional) - method of handling additional requests. If non-zero the oldest running copy will be taken over; if zero, the request will be ignored. The default value is 0 (ignore).

PERFORMANCE

turnon - activate a particular instrument for an indefinite performance time. This is typically used in the **orchestra header** section (Instrument 0) to activate an instrument required continuously (such as one containing reverb or global effects).

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.

maxalloc - an instrument containing this statement can allocate no more than *imax* instances of itself. Any further requests (from a score, midifile, or keyboard) will cause either the oldest running copy to be taken over or the request ignored, depending on *imethod*. This unit can prevent a computationally expensive instrument from becoming too prevalent.

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
```

STRING SET, PRESET and PROGRAM SET VARIABLES

Many of the following are described as operating at **Orchestra load-time**. This means they make their contribution when the Orchestra is first *translated and loaded*; they have no active role when the instrument in which they occur is run. Their role is one of setup—of creating variables, and defining the relationships between them. Their placement is indicative of scope: those in the Orchestra header (instr 0) generally have global influence, while those in an instrument have local.

strset ndx, string

Put a string into an indexed array for later reference.

This unit operates at **load-time**, putting the string pointer into an internal array at position *ndx*. The *ndx* argument must be a constant, and *string* is any sequence of characters or spaces enclosed in double quotes. The opcode can be repeated for multiple string entries. However, there is a limit of 32 strings that may currently be stored, and the indexes must lie in the range 1 - 32.

Certain opcodes permit *one* of their input arguments to be either a numeric variable or a string variable. (See **adsyn, pvoc, lpread, soundin,** and **GEN01.**) While a quoted string variable is most direct, numeric indexing can offer useful flexibility. These opcodes generally interpret a positive numeric value as an *ndx* into the array of string pointers defined by **strset** commands. While **strset** is itself a load-time action, the numeric value which references it later need only be I-time valid, so can come from generated instrument or score event data.

Example:

```
          strset    1, "pianopiece1"
          strset    2, "/newsamps/piano2.aiff"

          instr     1
a1, a2    soundin   p6          ; get whichever and play it
          outs      a1, a2
          endin
```

This would enable the two named soundfiles to be read and played at times determined by score events. The events may be sequential, overlapping or simultaneous.

pset	con1, con2, con3, ...
vset*	con1, con2, con3, ...
gvset*	con1, con2, con3, ...
vdim*	con1

Give preset values to array variables for later reference or modification.

These units define and initialize numeric arrays at Orchestra **load-time**. Although they are defined within an instrument and adhere to its scope, they are not part of its init-time or perf-time threaded operation. They exist only so their contents can be accessed, modified and re-accessed over time.

Each array belongs to the instrument *template* in which it is defined. This means that all instances of that template (each allocated copy of the instrument) will reference the same physical array. The current values in that array are thus global to all active copies of the instrument.

The variable presets must be *constants*, not expressions or other variables. Variables once defined can be modified by an accessing instrument, or by external operations (such as ontimer and sysex). The three forms available are related, but carefully distinguished.

pset - preset the pfield array (maximum of one statement per instrument). These values are available as I-time defaults. Although an instrument invoked from a numeric score will receive the pfield values of that score line, the same instrument invoked from a MIDI file or MIDI event can receive a *copy* of the template preset values, beginning with the third (positions are ordered: p1, p2, p3, ...). While for a MIDI invoked instrument its p1 and p2 will always receive the instrument number and current activation time, slots p3, p4 ... will receive the actual preset values. These presets are useful when developing a MIDI instrument from a score, and vice versa.

vset - preset a local v-array (maximum of one statement per instrument). This array (not a copy as in pset) is *directly* available to all instances of the host instrument, whether score invoked or midi invoked. Values are accessed as v1, v2, v3 ..., syntactically similar to p1, p2 ... in that they are valid at I-time and can be terms of an expression. They can also be modified, except that (unlike pfields) the modification is seen by all instances, and remains modified until changed again. These values may also be modified externally (by ontimer *vnumset*, or sysex), providing a way for external processes to change the behavior of an active instrument.

gvset - preset a global v-array (maximum one statement per orchestra, header section (instr 0) only). This array is similar to vset, with members gv1, gv2 ... valid anywhere in the orchestra. Modifications will be seen by all referencing instruments from the moment they occur. By further comparison with vset, this enables external processes to modify the behavior of many different instruments at once.

vdim - create a local v-array of dimension *con1* for later loading with data. In the absence of a vset or vprogs, this unit allows v-symbols to be legal in the current instrument definition.

UPDATES AND EXPRESSIONS

Although preset variables are syntactically similar to i-variables, there is a subtle difference. Like i-variables they can be bound at I-time with other i-variables or constants in an expression to form hidden temporary i-variables. But unlike normal i-variables, the vset and gvset members may be modified during performance, and this change will not be reflected in the intermediate

result. When the intent is for a changed *vnum* or *gvnum* to be immediately effective, it should either be used directly in a k-input, or be assigned to a k-variable before becoming part of an expression.

Example:

```
          gvset   12                      ; master volume
          instr   1
          pset    0, 0, 3, 4, 5, 6        ; pfield substitutes
          vset    1, 440
kamp      =       gv1                     ; get current volume
a1        oscil   kamp * 1000,  v2,  p6 ;         and scale it
```

pgminit* pgm.bn, ival1[, ival2, ... ival150]
dpgminit* dpgm.ch, ival1[, ival2, ... ival150]

vprogs* pgmno1[, pgmno2, ... pgmno32]
dvprogs* dpgmno1[, dpgmno2, ... pgmno32]

Construct a *v-array* and initialize it. Associate one or more *v-arrays* with an Instrument block.

INITIALIZATION

pgmno, dpgmno – base program id numbers (1-128) that identify a specific *v-array*. Constants only, valid at Orch **load-time.** These are of two types, standard and auxilliary (*pgmno, dpgmno*). The same id number is valid for different program types, provided each is unique within its type.

pgm.bn, dpgm.ch - like *pgmno*, *dpgmno*, with an *optional* two-digit decimal part denoting a bank number (0-64) or midi drum channel (1-96), e.g. 1.08. A base *pgm* or *dpgm* number can have only one bank or channel associated with it. The default bank is 0, and default drum channel is 10.

ival1, ival2, .. - values assigned to a specific *v-array*. Can be either constants or I-time expressions. The members of a *v-array* are invoked by the symbols v1, v2, ... v150. When used in the body of an instrument these are syntactically equivalent to i-variables.

PERFORMANCE

These units define and reference a *v-array* by a unique program number. **pgminit, dpgminit** can occur only in the Orchestra header (instrument 0); **vprogs** and **dvprogs** (which can be repeated) can be used only in subsequent instruments. The **program** number is the link that associates an initialized *v-array* with a specific instrument definition.

When an instrument contains one or more **vprogs** lists of program numbers, the instrument is invoked with just *one* of these programs active. Any v-symbol in the text (v1,v2,…) will have the value of that member of the currently active program. If the instrument is launched with a different program number, the v-symbols present will take on the initialized values of the new set.

A reference to a program number is thus a reference to a specially parameterized instrument block. During performance this reference can come in the form of a MIDI **program change**, which will cause the communicating MIDI channel to invoke the associated instrument with the v-values set by the **pgminit** for that program number. In MIDI invocation some instruments are denoted as drum sets, with a separate *auxilliary* set of program numbers; in this case, **dpgminit** and **dvprogs** can be used to avoid number clahes with the standard set.

Example:

```
ift1    ftgen       1, 0, 2048, 10, 1     ; create two stored ftables
ift2    ftgen       2, 0, 2048, 10, 3, 2, 1
        pgminit     21, 440, ift1         ;  & define two program sets
        pgminit     22, 660, ift2

        instr       1, 2, 3, 4, 5         ; define an instrument type
        vprogs      21, 22                ;  & assoc both program sets
a1      oscil       10000, v1, v2         ;  so as to run with either
```

```
out             a1
endin
```

dpkeys* dpgm.ch, ikey1, ikey2, …
dpexclus* dpgm.ch, ikey1, ikey2, …
inexclus* chnl, ctrlno, ins1, ins2, ins3, …

Set up a drumset keylist or instrument list, and specify any exclusive members.

LOAD-TIME

These statements are valid only in the Orchestra Header section, and are applied only at load-time. The input arguments must be constants, not variables. The argument *dpgm.ch* specifies a drum program and a two-digit channel number (see **dpgminit**). The default channel is 10.

dpkeys is used to specify which drum-keys will be used within a drumset. The keys should be listed in ascending order. When a program change is received, the system will make an attempt to preload all the sound samples needed by the keylist. A **dpgminit** should precede.

dpexclus is used to specify which drum-keys within a program form an exclusive performing set. When any member of an exclusive set is activated, all other members of the set still active are made mute. There can be up to eight exclusive sets defined within a single drum program, and each exclusive set can have up to 4 members. A **dpgminit** should precede.

inexclus defines a set of instrument numbers which will operate as an exclusive set when turned on by a value from midi controller *ctrlno*. If another member of the set is still active when a new request is received, that instrument is sent a turnoff message and the new instrument turnon will be delayed until the first is fully off. If the active instrument contains a note-off sensing "r" unit such as **linenr**, its life will be extended and the new instrument turnon will be delayed accordingly. There can be any number of exclusive instrument sets, and each set can have up to 16 members.

Example:

```
dpexclus    1.10, 42, 44, 46    ; make closed, pedal, and open hi-hat
                                ; exclusive in channel 10
```

sfprogs* pgmno1[, pgmno2, … pgmno32]
dlsprogs* pgmno1[, pgmno2, … pgmno32]

dzone*
endzone*

Associate specialty-type programs with v-arrays and d-arrays in an Instrument block.

PERFORMANCE

sfprogs and **dlsprogs,** like **vprogs**, serve to associate an Instrument block with specific Midi program changes, and will then pass the program-defining data into the activated instrument during performance. But whereas **vprogs** is a conduit for both explicitly defined programs (**pgminit**) and standard implicit compiled-in ones (**autopgms**), **sfprogs** and **dlsprogs** will pass data from only **auto_sf2** and **auto_dls** sources, respectively. There can be one or more program lists in an instrument, but only one *type* of list per instrument.

dlsprogs will place its articulation data in **v**-array cells, just like **vprogs**. The data arriving from a DLS file is a fixed array of 22 values, pre-processed by the Csound loader from a fixed-point encoded set into a scaled floating-point array for fast and convenient use by the Instrument. The parameters imply a specific instrument design (lfo, 2 envelopes, dB volume control, no filter), which is conveyed by the source file. It uses non-overlapping split data, accessible via v22. Additional source parameters have been folded into the following scaled set.

v1 – lfo frequency, in cps
v2 – lfo delay, in seconds
v3 – lfo attenuation scale
v4 – lfo pitch scale
v5 – lfo modwheel to attenuation
v6 – lfo modwheel to pitch
v7 – vol envlp attack time
v8 – vol envlp decay time
v9 – vol envlp sustain level
v10 – vol envlp release time
v11 – vol envlp velocity to attack
v12 – vol envlp key to decay
v13 – pch envlp attack time
v14 – pch envlp decay time
v15 – pch envlp sustain level
v16 – pch envlp release time
v17 – pch envlp velocity to attack
v18 – pch envlp key to decay
v19 – default pan (0 – 1)
v20 – vol envlp to attenuation
v21 – vol envlp to pitch
v22 – multi-sample split data

sfprogs data implies a more complex instrument. In addition to having a time-varying filter, it has overlapping split zones, defined by both key number and note-on velocity. This structure requires multiple *layers*, either as simultaneous cooperating instruments or as a single multi-dimensioned instrument. Csound enables the latter by the use of *dzones*. The **dzone** statement declares an area in an instrument in which a new **d**-array data type is valid; the area is delimited by an **endzone**. A **d**-value is basically a **v**-value which conveys a different value in different zones. **d**-values are strictly local to the current zone, so that a **d**-symbol (d1, d2, ..) in one zone is quite independent of the same **d**-symbol in another. Any other symbols (ival, ksig, etc.) are still global across all zones. There can be any number of dzones within an instrument. When **sfprogs** passes program data to an instrument, it will pass a unique set of **d**-values to each zone. An instrument must have enough dzones to accommodate this, and when an instrument is first associated with a program number, the soundfont Preset which represents that program will print its maximum required zone depth. An instrument with insufficient zones will render the channel mute. When an instrument has more zones than currently required (e.g. the current key does not lie in overlapped zones), the extra zones are simply skipped during performance.

Articulation data from a Soundfont Preset is a 2-dimensional array of n x 58 values, pre-processed and scaled for efficient use by instruments. Unused slots are available for temporary calculations, but reserved slots must be left intact. The parameters have the following meaning in each zone:

d1 – reserved
d2 – reserved
d3 – reserved
d4 – reserved
d5 – mod lfo to pitch, in octs
d6 – vib lfo to pitch, in octs
d7 – mod env to pitch, in octs
d8 – filter freq, as midi notnum
d9 – filter Q, as dB gain
d10 – mod lfo to filter freq, in nots
d11 – mod env to filter freq, in nots
d12 – reserved
d13 – mod lfo to vol, in dB
d14 – unused
d15 – chor fxsend, as fract
d16 – rvb fxsend, as fract
d17 – pan, as fract left
d18 – ftbl number
d19 – reserved (allkeys flag)
d20 – reserved (allvelocs flag)
d21 – mod lfo delay, in secs
d22 – mod lfo freq, in cps
d23 – vib lfo delay, in secs
d24 – vib lfo freq, in cps
d25 – mod env delay, in secs
d26 – mod env rise, in secs
d27 – mod env hold, in secs
d28 – mod env decay, in secs
d29 – mod env sust, as fract
d30 – mod env rels, in secs
d31 – key to mod env hold, scale per oct
d32 – key to mod env decay, scale per oct
d33 – vol env delay, in secs
d34 – vol env rise, in secs
d35 – vol env hold, in secs
d36 – vol env decay, in secs
d37 – vol env sust, as fract
d38 – vol env rels, in secs
d39 – key to vol env hold, scale per oct
d40 – key to vol env decay, scale per oct
d41 – instrument number
d42 – split hi key, as notnum
d43 – split lo key, as notnum
d44 – split lo veloc, as veloc
d45 – reserved
d46 – forced keynum
d47 – forced veloc
d48 – init attenuation, in dB
d49 – split hi veloc, as veloc
d50 – reserved
d51 – reserved
d52 – reserved
d53 – sample ID
d54 – reserved
d55 – tuning, in nots
d56 – scale tuning, as not mult
d57 – reserved (exclus code)
d58 – rootkey, as notnum

Example:

```
        instr   1, 2, 3                 ; make a simple 2-layer soundfont instr
        sfprogs 5, 27, 35               ; for these 3 progs (Presets)
ioct    octmidi
        dzone                                           ; 1st zone articulators
k1      dahdsr  d33, d34, d35, d36, d37, d38, -96, 0    ; vol envlp as db down
k2      lfo     d24, gisin, d23                         ; vibrato lfo
a1      loscil1 10000 * ampdb(k1), cpsoct(ioct + d6 * k2), d18, cpsnot(d58)
garvb   fxsend  a1, d16
        panouts a1, d17                                 ; send audio to output
        endzone
        dzone                                           ; 2nd zone articulators
k1      dahdsr  d33, d34, d35, d36, d37, d38, -96, 0    ; use as above
k2      lfo     d24, gisin, d23
a1      loscil1 10000 * ampdb(k1), cpsoct(ioct + d6 * k2), d18, cpsnot(d58)
garvb   fxsend  a1, d16
        panouts a1, d17
```

```
endzone
endin
```

autopgms*
auto_sf2*
auto_dls*

Induce automatic loading and activation of program data.

LOAD-TIME

These statement are valid only in the Orchestra Header section. Each will create an **automatic link** between incoming MIDI *program changes*, the *sound samples* and *articulation data* that are available to the system, and the special **vprogs, sfprogs** or **dlsprogs** data cells that pass this information to an activated instrument.

In a Csound orchestra, an instrument needing special data for its performance (sound samples and articulation data) must have these *preloaded* before the first note even begins. This can be done *explicitly* at Orch Init time (instr 0) using **pgminit, dpgminit** and **ftsplit**, and many of our best demonstration orchestras are constructed in just this way. However, such an orchestra can respond to only this set of program changes and articulation data, and no other.

The purpose of the above statements is to induce the Orchestra at Load time to look *elsewhere* for sound samples and articulation data. Each statement can infer a different place, and all three can be present simultaneously. They are characterized as follows:

autopgms will direct the loader to a *compiled-in* list of programs found in the module **progs.c**. This module includes both articulation data and an index into a directory of sound samples that is part of the standard distribution. Any ensuing program-change can be satisfied with sounds and data from this standard set, but none of the sound data is loaded until asked for later.

auto_sf2 will cause the loader to open and peruse a file named by the command-line **–e** flag. The file should be in **SoundFont 2.0** format, and is usually large. It contains all the articulation data needed by various program changes (called Presets), many of which are multi-layered; it also includes all the required sound samples within the file. On perusing the file, Csound will read all the articulation data into Host memory, but will skip over the sound data until asked for later.

auto_dls will cause the loader to open and peruse a file named by the command-line **–j** flag. The file should be in **Downloadable Sounds Level 1** format, and is usually large. It contains a coded version of the signal-processing networks that are the Instruments and programs of this set; it also includes all the required sounds within the file. On perusing this file, Csound will read all the articulation data into Host memory, but will skip over the sound data until asked for later.

When asked to preload the data of a specific program change, Csound will first examine the Instrument definitions to see which of the above program types is required. It will then download the articulation data to the DSP, and read the necessary sound samples from the file.

The program data and associated sound samples can be requested in two ways. On opening a MIDI file, Csound will first scan it for **Csound sysex** data which lists all the expected program changes, and will then load the information as described above. If the Midi stream cannot be pre-searched (e.g. it is being relayed by a Host-based Sequencer), alternative loading is provided by the **Preload** feature **(-P** flag), wherein a Preload list of expected program changes and drumset keys is given by a file. See **Chapter 6, Running from MIDI data.**

In all of these cases, the data that distinguishes a program change and its manner of performance can be preloaded and passed to the respective instruments by the **vprogs**, **sfprogs** and **dlsprogs** operations. An Orchestra of instruments need only be a set of signal-processing templates or methods; the different-sounding instances will be induced by the newly arriving program changes, and the data needed by each program is guaranteed to have been loaded.

These are powerful opcodes, enabling Csound to **emulate** specific audio synthesis systems, such as the Roland Sound Canvas SC88, the General Music WK4 synthesizer, the EMU/Creative SoundFont model, and the DLS general synthesizer. Alternatively, one can *mix and match* different program types within a single orchestra. Whichever the approach, these modules can enable timely loading of the appropriate wavetables and articulation data for any program change that will be invoked.

MIDI CONTROLLERS and CONVERTERS

These units allow instruments to be controlled from a disk-resident MIDI file, from internally generated MIDI commands (see "ontimer"), or from an external MIDI device such as a keyboard. All three sources can be operating simultaneously. Includes automatic scaling for Csound use.

	massign	ichnl, insno
	ctrlinit	ichnl, ictlno1, ival1[, ictlno2, ival2[, ictlno3, ... ival32]]
	pctrlinit*	pgm.bn, ictlno1, ival1[, ictlno2, ival2[, ictlno3, ... ival32]]
	dpctrlinit*	dpgm.ch, ictlno1, ival1[, ictlno2, ival2[, ictlno3, ... ival32]]
	uctrlmap*	ictlno[, ilow, ihigh]
	dsctrlmap*	iparm[, ilow, ihigh]
ival	**midictrl**	ictlno[, ilow, ihigh]
kval	**midictrl**	ictlno[, ilow, ihigh]
ival	**chanctrl**	ichnl, ictlno[, ilow, ihigh]
kval	**chanctrl**	ichnl, ictlno[, ilow, ihigh]
ival	**dsctrl***	iparm

Assign channels, set MIDI controller values, or get values from continuous controllers and convert them to a locally useful format.

INITIALIZATION

pgm.bn, dpgm.ch – base program id numbers (1-128), with bank or channel suffixes and their defaults as in **pgminit**, **dpgminit**. Constants only, so as to be valid at Orchestra **load-time.**

ichnl, ictlno - MIDI channel number (1 - 96) and MIDI controller number (0 - 120).

ilow, ihigh (optional) - I-time range pair onto which an incoming byte value 0-127 will be mapped. The default range for the above units is 0 - 127.

PERFORMANCE

These units manipulate the 121 controller values that are associated with each midi channel. Controllers belong to a channel, and any instrument invoked on a channel has access to all of the channel's controllers.

massign, ctrlinit - these commands will first open a MIDI channel (if not already open) and initialize the channel controllers to a standard set of default values. **massign** will then assign the channel to a specific Csound instrument with the defaults intact. **ctrlinit** will selectively override the defaults by reinitializing them to other values (it presumes the channel has been assigned to an instrument, or to instr 1 by default). These units are **orchestra header** statements, executed as part of instrument 0 before any other instruments are run. The units are repeatable, i.e. there can be any number of these statements in the orchestra header. If both units are applied to the same channel (assign and override), **massign** should precede.

pctrlinit, dpctrlinit - reinitialize one or more controllers following a *program* change (see **vprogs)**. These are **orchestra header** statements, executed as part of instrument 0. In contrast to **ctrlinit**, these values are saved and used later to reinit certain controllers following a *program*

change on the invoking channel. There can be only *one* init list (max of 32 overrides) for each base program id.

uctrlmap - set up a universal mapping function for a specific controller (all channels). This is an **orchestra header** statement, executed at instrument 0 time. Thereafter, all incoming controller values of that number will be mapped as directed, and the mapped value will then serve as the current value of that controller on the invoking channel. This unit provides an efficient means of scaling a standard controller value to the required range immediately on arrival; the scaled value can then be accessed by any instrument on that channel via the symbols c0, c1, c2, ... c120.

dsctrlmap - set up a drumset mapping function for a specific parameter (all drum channels). This is an **orchestra header** statement, executed only at instrument 0 time. Thereafter, all incoming drum instrument parameter settings of this number will be mapped as directed, and the mapped value will then serve as the current value of that parameter. Drum instrument parameters are set via non-registered parameter numbers (NRPN's) arriving on a drum channel. The numbering (normally a MSB value) varies among manufacturers, but this unit enforces a simple numbering for both mapped storage and later retrieval. For GS MIDI the number relations are:

1: MSB 24 (drum instr pitch)
2: MSB 26 (drum instr volume)
3: MSB 28(drum instr pan)
4: MSB 29 (drum instr reverb send)
5: MSB 30 (drum instr chorus send)
6: MSB 31 (drum instr delay send)

The current values can be referenced via the simple numbers (1 - 6).

midictrl, chanctrl - get the current value of a controller and optionally map it onto a specific range. These units are part of some instrument, operating only when that instrument is activated. Their purpose is to grab the current value of a controller, which may have been modified from its initial value by some external process or device. Since each channel has its own set of controllers, these units differ in that the first will access the controller set of the current channel, while the second can access the controllers of some other channel. The optional mapping should be used only if universal mapping (see **uctrlmap**) is not in effect; the option here is to provide a raw controller value with different mappings on different channels.

dsctrl - get the current value of a drumset instrument parameter on the current channel using simplified parameter numbering (see **dsctrlmap**). The values are scaled only on arrival.

Example:

```
uctrlmap    7, 0, 650       ; always map volume
uctrlpan    10, 0, 1        ; and pan
```

ival	**miditran**	idist
ival	**veloc**	[ilow, ihigh]
inot	**notnum**	
inot	**notnumb**	[isens]
knot	**notnumb**	[isens]
icps	**cpsmidi**	
icps	**cpsmidib**	[isens]
kcps	**cpsmidib**	[isens]
ioct	**octmidi**	
ioct	**octmidib**	[isens]
koct	**octmidib**	[isens]
ipch	**pchmidi**	
ipch	**pchmidib**	[isens]
kpch	**pchmidib**	[isens]
iamp	**ampmidi**	iscal[, ifn]
kaft	**polyaft**	[ilow, ihigh]
kaft	**aftouch**	[ilow, ihigh]
ibnd	**pchbend**	[ilow, ihigh]
kbnd	**pchbend**	[ilow, ihigh]

Modify or get a value from the MIDI event that activated this instrument, and convert it to a locally useful format, such as scaled by a non-MIDI value to make it optimum for Csound use.

INITIALIZATION

idist – transposing interval in semitones (up or down).

isens (optional) – pitch bend sensitivity (in semitones) by which the current note value 0-127 can be modified by a pitch bend value (up or down) prior to output in the requested form. If *isens* is 2, a full bend will modify the pitch by two semitones each way. Standard sensitivity is either 2 or 12. If the *isens* value here is zero, the sensitivity is taken from that for the channel (normally 2, but changeable by RPN controller values). The default value is 0 (get from the channel).

ilow, ihigh (optional) - I-time range pair onto which an incoming byte value 0-127 will be mapped. The default range for most units is 0, 127; the default range for **pchbend** is -1, +1.

ifn (optional) - function table number of a normalized *translation* table, by which the incoming value is first interpreted. The default value is 0, denoting no translation.

PERFORMANCE

miditran – transpose the current MIDI note number by *idist* semitones, i.e. make this a transposing instrument. This unit normally occurs at the top of an instrument, so that all subsequent pitch references are with respect to the transposed value.

veloc - get the MIDI byte value (0 - 127) denoting the *velocity* of the current event, and optionally remap it linearly to a different range.

notnum, cpsmidi, octmidi, pchmidi - get the *note number* of the current MIDI event, or express it in **cps, oct,** or **pch** units for local processing.

notnumb, cpsmidib, octmidib, pchmidib - get the *note number* of the current MIDI event, modify it by the current **pitch-bend** value (with *isens* scaling), and express the result in **not**, **cps, oct,** or **pch** units. The output is available as an I-time value or as a continuous *ksig* value.

ampmidi - get the *velocity* of the current MIDI event, optionally pass it through a normalized translation table, and return an amplitude value in the range 0 - *iscal*.

polyaft, aftouch - get the general (across all channels) or current channel *after-touch* value, and map it to the specified range.

pchbend - get the current *pitch-bend* value for this channel, and map it to the specified range. Note that this access to *pitch-bend* data is independent of the MIDI pitch, enabling the value here to be used for any arbitrary purpose.

veloffs*

Make this instrument sensitive to MIDI noteoff velocities.

PERFORMANCE

This unit is seen only at Orchestra translation and **load-time**, but its effect continues throughout Performance. Its purpose is to enable an instrument to respond to the noteoff velocities which are sometimes sent by MIDI files and MIDI performance keyboards.

Many unit generators in the following pages have names ending in "**r**" (**linsegr**, **linenr**, **adsr**). (Some of these also have an *irindep* argument.) These units are capable of extending the duration of a note by an additional "release time", and an instrument containing one of these units will (on reaching the end of its p3 duration or on sensing a MIDI noteoff signal) immediately go into *release* mode and automatically extend its duration by the stated release time. For scored p3 event times, release time is based on I-time data and extension proceeds as directed. But for MIDI events, the noteoff command (which signals the event closing) comes in two flavors: a *noteon* with velocity 0, or a *noteoff* with a release velocity. In the latter case, the presence of a **veloffs** command will cause the **r**-unit release time to vary with the release velocity.

Two rules govern **r**-unit release times. An extended release is necessarily the property of a note, so if two or more **r**-units in an instrument have different release times, and their *rindep* flag is off, each will use the longer time. Secondly, if the instrument contains a **veloffs** command, and if the turnoff data is a MIDI command with noteoff velocity (0 - 127), the **r**-unit release time will be modified. The mapping is exponential, and ranges from **one-tenth** the given release time (for high velocities) to **ten times** the given time (for low velocities). Thus if an **adsr** has a release time of 0.1 seconds, a keyboard performer with good release control could vary the actual time to range from .01 to 1 seconds. A MIDI noteoff velocity of 64 will preserve the original time value.

Example:

```
        instr       1
iamp    veloc       0, 5000
icps    cpsmidi
        veloffs
k1      linenr      iamp, .03, .1
a1      oscil       k1, icps, 1
        out         a1
        endin
```

This instrument will permit its release time of .1 seconds to be actively modified by the performer.

ival **mselect*** iscal, index, ival0, ival1[, ival2, … ival15]
kval **mselect*** kscal, index, kval0, kval1[, kval2, … kval15]

Index into an argument list and scale the selected value.

INITIALIZATION

index – selector index (0-15), used to choose amongst up to 16 input arguments.

PERFORMANCE

These units use a predefined *index* code to select from a set of control values or control signals, and the selected value is then multiplied by a scaling factor prior to output.

These units operate at the I-time rate or at the k-rate, depending on the type of output argument. *index* is always an I-time input, validated in the normal way. I-time **mselect** will choose from amongst a set of constants or I-time variables. For k-rate **mselect**, the k-time inputs (kscal, kval0, kval1, …) may variously contain control signals or constants or i-variables.

mselect is useful for such things as dynamic selection of control signals or modulation sources, especially during performances of an instrument template with behavior-modifying needs, such as those induced by MIDI program-changes and channel controllers.

Example:

```
kfc     mselect     kscal, index, ksig1, ksig2, c1, c7, c11, 0, 1000     ; set filter freq
                                                                         ; from amongst various controllers & control signals
```

id	**ftsplit***	isets, jsize, i1j1, i1j2, ... ,i2j1,i2j2, ... ,i3j1, ... [, isets2, jsize2, …]
i1,i2,i3,...i10	**mtsplit***	[id, ilay]

Store and retrieve multi-sample data from a two-dimensional list, with optional multi-layers.

INITIALIZATION

isets, *jsize* - dimensions of the array of values (*i1j1*, *i1j2*,...) in a layer, where the total number of *injn* values listed must equal *isets* x *jsize* (in that layer).

id (optional) - identifier relating the two opcodes. 0 if **autopgms** in use. The default value is 0.

ilay (optional) - layer level of data to be accessed in the split array. The default value is 1.

PERFORMANCE

These units allow a multi-sample instrument to gain quick acccess to parameters that pertain to its current pitch. The data can also be organized and retrieved in layers.

ftsplit (instr 0 header only) creates a structured list of *isets* groups of *jsize* items for later access via *id*. Each group consist of a MIDI split point, followed by parameter values that pertain to notes at or above the split. The first split is zero, and splits for other groups in a layer must be in ascending order. A new zero denotes the start of a new layer. The number of items in each group (including the split) must equal *jsize*, and the *total* number of *all* groups must equal *isets*. The resulting array is given a unique *id* that may be referenced at any time during performance.

mtsplit is an I-time function that permits an instrument to access information stored by **ftsplit** simply by invoking *id* and *ilay* values. The current midinote is used as an index into the array for a given layer, and the result cells (up to 10) are automatically filled with data from the appropriate split-point group. This enables a multi-sampled and multi-layered instrument to access multiple control values across its entire pitch range. Multiple layers may be referenced in any order.

The unique *id* can be specified directly, or via indirect selection. An example of the latter is seen when two or more *id* symbols are imbedded in alternate midi program data, and the choice is made by a **vprogs** program change. In fact, when **autopgms** is enabled (**ftsplit** not actually present, but a hidden one is implied), this indirect strategy is the one internally implemented.

Example:

```
ip1     ftload      "pno1"
ip2     ftload      "pno2"              ; load samples
ih1     ftload      "hrp1"
ih2     ftload      "hrp2"
ispn    ftsplit     2,2, 0,ip1, 60,ip2  ; set up split data
ishp    ftsplit     2,2, 0,ih1, 58,ih2
        pgminit     1,ispn              ; & store each id in a pgm
        pgminit     7,ishp

        instr       3,4
        vprogs      1,7                 ; on a program change
iamp    ampmidi     1000
icps    cpsmidi
ift     mtsplit     v1                  ; fetch the right split data
```

```
a1      loscil1         iamp, icps, ift         ; & play from that ftable
```

FTABLES

iafno	**ftgen**	ifno, itime, isize, igen, iarga[, iargb, ...iargz]
iafno	**ftload***	ifilnam[, iskiptime, iformat, ichnl, inorm]
iafno	**ftstep***	ix1, ia, ix2, ib[, ix3, ic, ...], ixn
	ftscale*	ifno, iscale

These units allow stored function tables, normally defined in a Csound score (see the **F** statement), to be defined in the orchestra.

INITIALIZATION

iafno - ftable number, either requested or automatically assigned (101, 102, 103 ....). A global (*giafno*) can be referenced from any instrument in the orchestra.

ifno, itime, isize, igen, iarga, iargb , ..- input values as described in the score **F** statement, except for the following differences:

ifno if zero, an ftable number is automatically assigned (101, ...) and copied into *iafno*.
if non-zero, the requested number is copied into *iafno*.

itime ignored, effectively the action time of this instr (e.g. time 0 for header statements).

ifilnam - file name enclosed in quotes of a disk-resident sample file, expected in either the current directory or that defined by SFDIR.

iskiptime, iformat, ichnl (optional) - input values as described in GEN01. All values default to 0.

inorm (optional) – normalization flag (normalize to +,-1, else no rescaling). Default value is 1 (normalize ON).

x1, a, x2 ... - input values as described in GEN17 (x-ordinate plus y-value pairs), except that a final x-ordinate is required to define the size of the function table.

PERFORMANCE

These units bring ftable generation into the orchestra. Table creation time is either at Orchestra Init (header statements, instr 0) or at each action time of an instrument containing one of these units. Each table may be automatically assigned a number, which can be referenced symbolically to reduce the book-keeping that attends score definition. However, automatic numbering means that ftable numbers above 100 should not be explicitly used.

ftgen - invoke any ftable generator, as described in the Gen Routines of Chapter 4. Note that when using **Extended Csound**, the ftable (normally local) can optionally be stored in host memory by appending a fractional part (.3) to *ifno*. The fraction .35 will additionally give the table two other downsampled forms (by 2 and by 4), to speed access by an **loscil** unit.

ftload - a brief way of invoking GEN01 (read a file from disk) when the filename is known (not generated) and the size is automatically deferred. Useful for reading sampled sound into an ftable.

ftstep - a brief way of invoking GEN17 with no rescaling. Creates a step function from x = 0 using some number of x,y coordinate points (x ordinates increasing). Must terminate in a single

x ordinate (power-of-two or one less), to define the overall size of the ftable. Useful for mapping midi note numbers onto control data such as ftable numbers.

ftscale - rescale an existing ftable by the factor *iscale*. When neither the original amplitude nor rescaling to unity is suitable, an existing ftable can be rescaled by any factor. This is useful for balancing a set of recorded samples for consistent voicing. No fidelity is lost in rescaling.

Example:

```
ivln1   ftload      "violin.C4"                         ; get 3 violin tones
ivln2   ftload      "violin.A4"
ivln3   ftload      "violin.F#5
givtbl  ftstep      0, ivln1, 64, ivln2, 73, ivln3, 127 ; fn to map to them from midi

        ftscale     ivln1, 1.2                          ; make the 1st a little louder
        ftscale     ivln3, 0.9                          ; and the 3rd a little softer
```

MACROS

macro* name
endm*

Define a group of Orchestra statements for use later in the text.

Csound statements that are frequently used in fixed combinations may be defined as a macro and invoked anywhere in the orchestra text. Macros must be defined at the top of the Orchestra file; they may be preceded by or contain comments or blank lines, but they cannot be preceded by any real opcodes (such as orchestra header statements). There is currently a limit of ten macro definitions in an orchestra.

Macros will be expanded as literal text repeats; no parameter substitutions are possible. On expansion, the text will be translated as if each line were in the original file. Argument names and labels must conform to the syntax of a normal Csound Orchestra.

On compiling the expanded orchestra, Csound will flag syntactic errors. Line numbers will be those of the *unexpanded* text. Problems with a macro call inside an instrument will be flagged as problems in the original definition.

While the use of macros is an editing convenience, there will be a tendency to include statements not always necessary at every invocation. One should use the editing convenience in pursuit of elegant and efficient instrument designs, especially if realtime performance is the goal.

Example:

```
                        ; this is my first macro
        macro       getmidi
iamp    veloc       0, 100
kcps    cpsmidib    2
kvol    midictrl    7, 0, 20        ;  chk this later  -- maybe too fussy !
kamp    =           iamp * kvol
        endm

        sr = 16000

        instr       1,2,3
        getmidi
a1      oscil       kamp, kcps, 1
        out         a1
        endin
```

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, icI...]]
kr	**linsegr**	ia, idur1, ib[, idur2, ic[...]], irel, iz
ar	**linsegr**	ia, idur1, ib[, idur2, icI...]], irel, iz
kr	**expseg**	ia, idur1, ib[, idur2, ic[...]]
ar	**expseg**	ia, idur1, ib[, idur2, ic[...]]
kr	**expsegr**	ia, idur1, ib[, idur2, ic[...]], irel, iz
ar	**expsegr**	ia, idur1, ib[, idur2, ic[...]], irel, iz

Generate straight line segment(s) or exponential curve segment(s), with optional noteoff sensing.

INITIALIZATION

ia, ib, ic, etc. - value at start, and after *dur1* seconds, etc. Exponentials cannot include zero.

idur1 - duration in seconds of first segment. A negative value will skip all initialization.

idur2, *idur3*, etc. - duration in seconds of subsequent segments. A zero or negative value means move immediately to the next point, permitting discontinuities in the line or curve sequence.

irel, iz - duration in seconds and final value of a note releasing segment.

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 hold the last-defined point (or the one before a release segment).

linsegr, expsegr are amongst the Csound "**r**" units that contain a **note-off sensor** and **release time extender**. When each senses an event termination or MIDI noteoff, it immediately extends the performance time of the current instrument by *irel* seconds, and sets out to reach the value *iz* by the end of that period (no matter which segment the unit is in). "**r**" units can also be modified by MIDI noteoff velocities (see **veloffs**). For two or more extenders in an instrument, extension is by the greatest period.

Example:

```
k2      expsegr   440, .25, 880, .25, 440, .1, 660
```

This statement creates a control signal which moves exponentially from 440 to 880 and back over one half second, then remains there until it senses a noteoff, when it goes to 660 in the extra tenth.

kr **linseg4*** ia, idur1, ib, idur2, ic, idur3, id, idur4, ie [, imode, itmap, ilvlvel, iatvel, iktrk]

kr **linseg4r*** ia, idur1, ib, idur2, ic, idur3, id, idur4, ie, irel, iz[, irind, imode, itmap, ilvlvel, iatvel, iktrk, irelmod]

Generate a four-segment control signal, with optional velocity and key tracking influence.

INITIALIZATION

ia, ib, ic, id, ie – ouput value at start, and after the intervening *dur1, dur2* … seconds.

idur1, *idur2*, etc. – duration in seconds of intervening line segments. A zero or negative value means move immediately to the next point, permitting discontinuities in the line sequence.

irel, iz - duration in seconds and final value of a note releasing segment.

irind (optional) – independence flag. If non-zero, the release time *irel* will not follow the instrument maximum (see **adsr**). The default value is 0 (will follow).

imode (optional) – mode of operation. There are 3 modes available:

0 – normal mode, in which the 4th segment end value *ie* will be sustained until the current note is released, at which time the output will move over *irel* seconds to the value *iz*.

1 – finish mode, in which the sustain portion is skipped, and the release portion happens immediately as a 5th segment, stopping at the value *iz*.

2 – repeat mode. When the 4th segment terminates at *ie*, output will then move over *idur1* seconds to the value *ib*, then over *idur2* to *ic*, etc., repeating this loop until the note is released, when the output will move over *irel* seconds to the value *iz*.

The default mode is 0 (normal mode).

itmap (optional) – invoke time mapping. If non-zero, all time arguments (0 – 99) will be mapped through an internal psuedo-exponential function, represented by the following sample points:

0	0	30	.44	60	3.2	85	18
1	.01	36	.62	65	4.6	90	26
2	.02	43	1.0	70	6.5	95	37
10	.1	50	1.6	75	9.3	99	49
25	.30	55	2.3	80	13		

Time mapping is required for the options below. The default value is 0 (unmapped, no extra opts).

ilvlvel, iatvel, irelmod (optional) – velocity to level, velocity to attack, velocity to release. These are *scaling* factors (0 to 1) that cause the midi-event onset velocity to modify the input levels *ia…iz*, the rise time *idur1,* and the release time *irel*, respectively, each by a proportionate amout. If *ilvlvel* is non-zero, velocities less than 127 will decrease the levels by a proportionate amount. If *iatvel* is non-zero, velocities above zero will decrease the rise time by a proportionate amount. If *irelmod* is non-zero, velocities above zero will decrease the release time by a proportionate amount. The magnitude of this influence is proportional to the scaling factors (0 to 1); negatives will cause the reverse effect. The default values are each 0 (no influence due to velocity).

iktrk (optional) – key-number to timing. This is a scaling factor that causes the midi-event key number to modify the durations of the middle segments. The key influence pivots around key 66: high key numbers will decrease the time values, while low key numbers will increase the values. The magnitude of this influence is proportional to the *iktrk* scaling factor (0 to 1); negatives will cause the reverse effect. The default value is 0 (no influence due to key number).

PERFORMANCE

These are specialty line-segment generators that incorporate several desirable functions that would otherwise be tedious to obtain using normal Csound opcodes. Their inclusion here allows them to be invoked simply and efficiently.

kr	**adsr***	iris, idec, isus, irel, ibas, ipeak[, iconv][, irind][, ioff]
ar	**adsr***	iris, idec, isus, irel, ibas, ipeak[, iconv][, irind][, ioff]
kr	**dahdsr***	idel, iris, ihold, idec, isus, irel, ibas, ipeak[, iconv][, irind][, ioff]

Generate a four- or six-segment control signal or audio signal, with optional convex rise.

INITIALIZATION

idel, iris, ihold, idec, irel - time in seconds for the delay, rise, hold, decay and release segments of the signal. A zero value in any position will move the envelope immediately to the next segment.

ibas, ipeak – floor and ceiling values between which the segments will rise and fall. Can be positive or negative values, with a positive or negative difference (polarity), but one of the pair must be 0 (e.g. 0,1; 1,0; -96,0; etc.). The rise segment will traverse the full range (*ibas* to *ipeak*), after which the remaining segments will trace a return to the base value. In **dahdsr** the delay segment will iterate the initial base value, and the hold will iterate the peak value.

isus – fraction of the peak value to which the decay will fall. This value will then be sustained until the instrument receives a note-off command, when the unit will release towards its base.

iconv (optional) – convex rise flag. If non-zero, will cause the rise segment to be *convex* while other segments remain linear. The curve is such that a –96 to 0 span will, under **ampdb** transformation, revert to a *linear* rise. The default value is 0 (not convex).

irind (optional) - independence flag. If non-zero, the release time (irel) will not follow the instrument maximum (see below). The default value is 0 (will follow).

ioff (optional) – end-of-release turnoff flag. If non-zero, an internal sensor will automatically turn off the instrument upon reaching end-of-release. The default value is 0 (no turnoff).

PERFORMANCE

adsr and **dahdsr** provide efficient and flexible segments suitable for amplitude and pitch control. The second unit is identical to the first, but can prolong the start and end values (base, peak) that surround the rise segment. **dahdsr** currently exists only as a control-signal generator.

These units ordinarily generate straight lines in each segment, which are often then exponentially modified for appropriate amplitude or pitch control. The rise segment, however, can optionally be a convex curve, so that the rise shape will retain a linear character under such modification.

The segment timings (in seconds) are often made to vary with note-on velocity or key number, applied as I-time calculations. Also, as in other "**r**" units, the *release* time may be further modified by a velocity-bearing MIDI note-off (invoked by a **veloffs** anywhere in the instrument).

Example:

```
icps    cpsmidi                          ; notnum in cps form
ioct    octmidi                          ; notnum in oct form
idec    =       .2                       ; decay for middle C
ifact   =       .5                       ; factor per octave up
idec    =       idec * twopwr((8 – ioct) * ifact)   ; decay for this note
kdb     adsr    .1, idec, .9, .3, -96, 0, 1         ; amp envlp as dB down
a1      oscil   10000 * ampdb(kdb), icps, 1         ; now applied to audio sig
```

kr	**dexponr***	ival, idel, idecrat, irel
ar	**dexponr***	ival, idel, idecrat, irel

Trace a delayed exponential decay, with note-off release.

INITIALIZATION

ival, idel - initial value, and time in seconds for which it will be held.

idecrat - decay rate, as a fractional rate per second, that will be applied to *ival* after *idel* seconds.

irel - release time in seconds, following a note-off command.

PERFORMANCE

These units generate control envelopes, by which an independent audio signal can be progressively decayed. They are particularly useful for modifying the amplitude output of a looping oscilator, so that it appears to lose energy during its psuedo steady-state. The initial delay allows the sampled attack portion to remain untouched, and should be set to the first loop point in the sample. The decay rate should also be chosen to suit the sampled sound and the pitch at which it is being played. The release path is linear towards zero over the given time.

dexponr is among the Csound "r" units that contain a **note-off sensor** and **release time extender**. On sensing an event termination or MIDI noteoff, it immediately extends the performance time of the current instrument by *irel* seconds, then moves linearly to the value zero in the extra time allotted. For 2 or more extenders in an instrument, extension is by the greatest period.

Example:

```
kamp   dexponr     1, ftlptim(ifn), .77, .1     ; create a delayed amp decay
a1     loscil1     kamp, 440, ifn, 261          ; and modify the sample
```

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 he 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]
ir	**dtable***	indx, ifn
kr	**oscil1**	idel, kamp, idur, ifn
kr	**oscil1i**	idel, kamp, idur, ifn
ar	**oscil1**	idel, xamp, idur, ifn
ar	**oscil1i**	idel, xamp, idur, ifn
ar	**osciln**	kamp, ifrq, ifn, itimes

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.

ifrq, itimes - rate and number of times through the stored table.

PERFORMANCE

table invokes table lookup for init, control or audio indices—as raw numbers (0,l,2...size-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* can go to full scale or interpolation is being used, the table should have an extended guard point. **table** indexed by a periodic **phasor** simulates an oscillator.

dtable is a very fast table lookup, operating only at instrument I-time. It does no range-checking or wrap around, and presumes the raw index is exactly within range. It does not use interpolation. This unit is often used for extracting values from a GEN17-generated table of split-point data.

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 move 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 is then multiplied by an amplitude factor *kamp* before being written into the result.

osciln will sample several times through the stored table at a rate of *ifrq* times per second, after which it will output zeros. Generates audio signals only, with output values scaled by *kamp*.

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

kr	**lfo***	kcps, ifn[, idel, iphs]
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	**coscil***	xamp, kcps, kcents, ifn[, iphs]

Table *ifn* is incrementally sampled modulo the table size and the value obtained is scaled by *amp*.

INITIALIZATION

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

idel (optional) - delay in seconds before incremental sampling begins. The default value is 0.

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 returned from sampling a stored function table, scaled by *kamp* (*xamp*). The internal phase is then advanced according to *cps*. **lfo** is unique in having an initial delay and no amplitude scaling. While *amp* and *cps* inputs to K-rate **oscils** are scalar only, the inputs to audio-rate **oscils** may be either scalar or vector, thus permitting amplitude and frequency modulation at either sub-audio or audio rates.

foscil is a composite unit that effectively banks two **oscils** 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.

coscil is a chorusing oscillator that banks three **oscils** in detuned additive synthesis. The detuning interval *kcents* (100 for a semitone) causes the second and third oscillators to add sound above and below the central *kcps* frequency to provide an efficient chorusing effect. The three units are internally added and scaled to unity before the amplitude *xamp* is applied. **coscil** is available only with krate frequency controls, and there is no interpolating version.

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      lfo     5, 1                    ; 5 cps vibrato (fn 1 is sinusoid)
```

```
a1      oscil   5000, 440 + 10 * k1, 1          ; of +/- 10 cps around A440
```

ar1 [,ar2] **loscil** xamp, kcps, ifn[, ibas][, imod1,ibeg1,iend1][, imod2,ibeg2,iend2]

Read sampled sound (mono or stereo) from a table, with optional sustain and release looping.

INITIALIZATION

ifn - function table number, typically denoting an AIFF sampled sound segment with prescribed looping points. The source file may be mono or stereo.

ibas (optional) - base frequency in **cps** of the recorded sound. Optionally overrides the frequency given in the AIFF file, but is required if the file did not contain one. The default is 0 (no override).

imod1, mod2 (optional) - play modes for the sustain and release loops. A value of 1 denotes normal looping, 2 denotes forward & backward looping, 0 denotes no looping. The default value (-1) will defer to the mode and the looping points given in the source file.

ibeg1, iend1, ibeg2, iend2 (optional, dependent on *mod1*, *mod2*) - begin and end points of the sustain and release loops. These are measured in **sample frames** from the beginning of the file, so will look the same whether the sound segment is monaural or stereo.

PERFORMANCE

loscil samples the ftable audio at a rate determined by *kcps*, then multiplies the result by *xamp*. The sampling increment for *kcps* is dependent on the table's base-note frequency *ibas,* and is automatically adjusted if the orchestra **sr** value differs from that at which the source was recorded. In this unit, ftable is always sampled with interpolation.

If sampling reaches the *sustain loop* endpoint and looping is in effect, the point of sampling will be modified and **loscil** will continue reading from within that loop segment. Once the instrument has received a *turnoff* signal (from the score or from a MIDI **noteoff** event), the next sustain endpoint encountered will be ignored and sampling will continue towards the *release loop* end-point, or towards the last sample (henceforth to zeros).

loscil is the basic unit for building a sampling synthesizer. Given a sufficient set of recorded piano tones, for example, this unit can resample them to simulate the missing tones. Locating the sound source nearest a desired pitch can be done via table lookup. Once a sampling instrument has begun, its *turnoff* point may be unpredictable and require an external *release* envelope; this is often done by gating the sampled audio with **linenr**, which will extend the duration of a turned-off instrument by a specific period while it implements a decay.

Example:

```
inum    notnum
icps    cpsmidi
iamp    ampmidi     3000, 1
ifno    table       inum, 2                     ;notnum to choose an audio sample
ibas    table       inum, 3
kamp    linenr      iamp, 0, .05                ;at noteoff, extend by 50 millisecs
asig    loscil      kamp, icps, ifno, cpsoct(ibas/12. + 3)
```

ar	**doscil***	kamp, ifno[, iautoff]
ar	**doscilp***	kamp, kcps, ifno[, ibas, iautoff]
ar	**doscilpt***	kamp, kcps, ktempo, ifno, iris, iss[, ibas, iautoff]
ar	**loscil1***	kamp, kcps, ifno[, ibas]
a1,a2	**loscil2***	kamp, kcps, ifno[, ibas]

Read a sampled sound with optional sustain looping and/or pitch and tempo control.

INITIALIZATION

ifno - function table number, typically denoting a WAV file (for **doscil**, **doscilp**, **doscilpt**) or an AIFF sampled file with sustain loop points (for **loscil1, loscil2**).

iris, iss – rise time and steady-state time (in seconds) of an internal cross-fade envelope within the **doscilpt** unit. Both should be small, e.g. .005 and .01, or even smaller if the tempo changes are excessive. *iris* should be at least one k-prd, though *iss* can be as small as 0.

ibas (optional) - base frequency in cps of the recorded sound. Optionally overrides the frequency given in the WAV or AIFF file, but is required if the file did not contain one. The default is 0 (no override).

iautoff (optional) – if non-zero, automatically *turn off* the current instrument when the end of sampled sound is reached (else output zeros). The default value is 1 (automatic turn-off).

PERFORMANCE

doscil is a direct form of **loscil**, with no frequency control and no looping. It simply reads the table from beginning to end, then stops. This is useful for sampled effects such as percussion sounds, where the pitch needs no alteration and loop points are not provided.

doscilp is a variant of the *non-looping* **doscil**, with additional frequency control. It is useful for playing back samples like recorded percussion sounds in which some additional tuning or pitch modification is desired. Note that while instrument sounds can often be varied by as much as an octave up or down, recorded voice moved by just a few semitones will be noticeably distorted since this unit does not preserve speech formants. For pitch-shifted voice, use a **harmon** unit.

doscilpt is a variant of the *non-looping* **doscilp** with independent pitch and tempo control. When ktempo is 1 the sound sample will play at the original tempo; when ktempo is say .5 or 1.5 the sound sample will play at one-half tempo or 1.5 times the original tempo, but with *no pitch shift*. Tempo variation can be quite extreme when iris and iss are carefully set. The unit even allows tempo speed-up with pitch lower, not higher, etc. However, speech formant limits still apply.

NB: doscilp, doscilpt, loscil1 and **loscil2** sample the ftable data at a rate determined by *kcps*. The sampling increment for *kcps* is dependent on the table's base-note frequency *ibas*, and is automatically adjusted if the orchestra **sr** value differs from that at which the source was recorded. The ftables are sampled with two-point interpolation.

Example:

```
kamp   dexponr     1, ftlptim(ifn), .77, .1      ; create a delayed amp decay
a1     loscil1     kamp, 440, ifn, 261           ; and modify the sample
```

ar	**poscil***	kamp, kcps, kfrac[, iphs]
ar	**buzz**	xamp, xcps, knh, ifn[, iphs]
ar	**gbuzz**	xamp, xcps, knh, klh, kr, ifn[, iphs]

Output a pulse train, or set of harmonically related cosine partials.

INITIALIZATION

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.

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.

PERFORMANCE

kfrac - fractional width of the pulse part of a cycle. **poscil** is a k-rate varying pulse-width audio oscillator, with on/off levels set to provide a zero-average DC offset. Although pulse-width oscillation is an unnatural signal (it does not occur in natural instruments), it is a popular source in synthesizers.

The **buzz** 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. The **buzz** and **gbuzz** 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, ksmod, ifilcod
ar	**pvoc**	ktimpnt, kfmod, ifilcod [, ispecwp]

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

INITIALIZATION

ifilcod - integer or character-string denoting a control-file derived from analysis of an audio signal. A positive integer denotes an indexed filename in the Orchestra (see **strset**); a negative integer denotes the suffix of a file *adsyn.m* or *pvoc.m*; a character-string (in double quotes) gives the filename itself. Filenames are optionally fullpath. If not fullpath, the file is sought first in the current directory, then in the one given by the environment variable SADIR (if defined). **adsyn** control contains breakpoint amplitude- and frequency-envelope values organized for oscillator resynthesis, while **pvoc** control contains similar data organized for fft resynthesis. Memory usage depends on the size of the files involved, which are read and held entirely in memory during computation but are shared by multiple calls (see also **lpread**).

ispecwp (optional) - if non-zero, attempts to preserve the spectral envelope while its frequency content is varied by *kfmod.* The default value is zero.

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 describing amplitude and frequency tracks in millisecond breakpoint fashion . Tracks are defined by sequences of 16-bit binary integers:

-1, time, amp, time, amp,...
-2, time, freq, time, freq,...

such as from hetrodyne filter analysis of an audio file. (For details see the Appendix on **hetro**.) The instantaneous amplitude and frequency values are used by an internal fixed-point oscillator that adds each active partial into an accumulated output signal. While there is a practical limit (currently 50) on the number of contributing partials, there is no restriction on their behavior over time. Any sound that can be described in terms of the behavior of sinusoids can be synthesized by **adsyn** alone.

Sound described by an **adsyn** control file can also be modified during re-synthesis. The signals *kamod*, *kfmod*, *ksmod* will modify the amplitude, frequency, and speed of contributing partials. These are multiplying factors, with *kfmod* modifying the cps frequency and *ksmod* modifying the *speed* with which the millisecond bread-point line-segments are traversed. Thus .7, 1.5, and 2 will give rise to a softer sound, a perfect fifth higher, but only half as long. The values 1,1,1 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, xform, koct, kband, kris, kdur, kdec,
iolaps, ifna, ifnb, itotdur[, iphs][, ifmode]

Audio output is a succession of sinusoid bursts initiated at frequency *xfund* with a spectral peak at *xform*. For *xfund* above 25 Hz these bursts produce a speech-like formant with spectral characteristics determined by the k-input parameters. For lower fundamentals this generator provides a special form of granular synthesis.

INITIALIZATION

iolaps - number of preallocated spaces needed to hold overlapping burst data. Overlaps are frequency dependent, and the space required depends on the maximum value of *xfund * kdur.* Can be over-estimated at no computation cost. Uses less than 50 bytes of memory per *iolap*.

ifna, *ifnb* - table numbers of two stored functions. The first is a sine table for sineburst synthesis (size of at least 4096 recommended). The second is a rise shape, used forwards and backwards to shape the sineburst rise and decay; this may be linear (GEN07) or perhaps a sigmoid (GEN19).

itotdur - total time during which this **fof** will be active. Normally set to p3. No new sineburst is created if it cannot complete its *kdur* within the remaining *itotdur*.

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

ifmode (optional) - formant frequency mode. If zero, each sineburst keeps the *xform* frequency it was launched with. If non-zero, each is influenced by *xform* continuously. The default value is 0.

PERFORMANCE

xamp - peak amplitude of each sineburst, observed at the true end of its rise pattern. The rise may exceed this value given a large bandwidth (say, Q < 10) and/or when the bursts are overlapping.

xfund - the fundamental frequency (in Hertz) of the impulses that create new sinebursts.

xform - the formant frequency, i.e. freq of the sinusoid burst induced by each *xfund* impulse. This frequency can be fixed for each burst or can vary continuously (see *ifmode*).

koct - octaviation index, normally zero. If greater than zero, lowers the effective *xfund* frequency by attenuating odd-numbered sinebursts. Whole numbers are full octaves, fractions transitional.

kband - the formant bandwidth (at -6dB), expressed in Hz. The bandwidth determines the rate of exponential decay throughout the sineburst, before the enveloping described below is applied.

kris, kdur, kdec - rise, overall duration, and decay times (in seconds) of the sinusoid burst. These values apply an enveloped duration to each burst, in similar fashion to a Csound **linen** generator but with rise and decay shapes derived from the *ifnb* input. *kris* inversely determines the skirtwidth (at -40 dB) of the induced formant region. *kdur* affects the density of sineburst overlaps, and thus the speed of computation. Typical values for vocal imitation are .003,.02,.007.

Csound's **fof** generator is loosely based on Michael Clarke's C-coding of IRCAM's CHANT program (Xavier Rodet et al.). Each **fof** produces a single formant, and the output of four or more of these can be summed to produce a rich vocal imitation. **fof** synthesis is a special form of granular synthesis, and this implementation aids transformation between vocal imitation and granular textures. Computation speed depends on *kdur, xfund,* and the density of any overlaps.

ar **harmon** asig, kestfrq, kmaxvar, kgenfreq1, kgenfreq2, imode, iminfrq, iprd

Analyse an audio input and generate harmonizing voices in synchrony.

INITIALIZATION

imode - interpeting mode for the generating frequency inputs *kgenfreq1*, *kgenfreq2*.
0: input values are ratios w.r.t. the audio signal analyzed cps.
1: input values are the actual requested frequencies in cps.

iminfrq - the lowest expected frequency (in cps) of the audio input. This parameter determines how much of the input is saved for the running analysis, and sets a lower bound on the internal pitch tracker.

iprd - period of analysis (in seconds). Since the internal pitch analysis can be time-consuming, the input is typically analysed only each 20 to 50 milliseconds.

PERFORMANCE

This unit is a **harmonizer**, able to provide up to two additional voices with the same amplitude and spectrum as the input. The input analysis is assisted by two things: an input estimated frequency *kestfrq* (in cps), and a fractional maximum variance *kmaxvar* about that estimate which serves to limit the size of the search. Once the real input frequency is determined, the most recent pulse shape is used to generate the other voices at their requested frequencies.

The three frequency inputs can be derived in various ways from a score file or MIDI source. The first is the expected pitch, with a variance parameter allowing for inflections or inaccuracies; if the expected pitch is zero the harmonizer will be silent. The second and third pitches control the output frequencies; if either is zero the harmonizer will output only the non-zero request; if both are zero the harmonizer will be silent. When the requested frequency is higher than the input, the process requires additional computation due to overlapped output pulses. This is currently limited for efficiency reasons, with the result that only one voice can be higher than the input at any one time.

This unit is useful for supplying a background chorus effect on demand, or for correcting the pitch of a faulty input vocal. There is essentially no delay between input and output. Output includes only the generated parts, and does not include the input.

Example:

```
asig1   in                                                      ; get the live input
kcps1   cpsmidib                                                ; and its target pitch
asig2   harmon  asig1, kcps1, .3, kcps1, kcps1 * 1.25, 1, 110, .04 ; add maj 3rd
        out     asig2                   ; output just the corrected and added voices
```

ar	**harmon2***	asig, koct, kfrq1, kfrq2, icpsmode, ilowest
ar	**harmon3***	asig, koct, kfrq1, kfrq2, kfrq3, icpsmode, ilowest
ar	**harmon4***	asig, koct, kfrq1, kfrq2, kfrq3, kfrq4, icpsmode, ilowest

Generate harmonizing voices with formants preserved.

INITIALIZATION

icpsmode - interpreting mode for the generating frequency inputs *kfrq1, kfrq2, kfrq3, kfrq4*.
0: input values are ratios w.r.t. the cps equivalent of *koct*.
1: input values are the actual requested frequencies in cps.

ilowest - lowest value of the *koct* input for which harmonizing voices will be generated.

PERFORMANCE

These are high-performance **harmonizers**, able to provide up to four pitch-shifted copies of the input *asig* with spectral formants preserved. The pitch-shifting algorithm requires an accurate running estimate (*koct*, in decimal oct units) of the pitched content of *asig*, normally gained from an independent pitch tracker such as **specptrk**. The algorithm then isolates the most recent full pulse within *asig*, and uses this to generate the other voices at their required pulse rates.

If the frequency (or ratio) presented to *kfrq1, kfrq2, kfrq3* or *kfrq4* is zero, then no signal is generated for that voice. If any of them is non-zero, but the *koct* input is below the value *ilowest*, then that voice will output a direct copy of the input *asig*. As a consequence, the data arriving at the k-rate inputs can variously cause the generated voices to be turned on or off, to pass a direct copy of a non-voiced fricative source, or to harmonize the source according to some constructed algorithm. The transition from one mode to another is cross-faded, giving seemless alternating between voiced (harmonized) and non-voiced fricatives during spoken or sung input.

harmon2, 3, 4 are especially matched to the output of **specptrk**. The latter generates pitch data in decimal octave format; it also emits its base value if no pitch is identified (as in fricative noise) and emits zero if the energy falls below a threshold, so that **harmon2, 3, 4** can be set to pass the direct signal in both cases. Of course, any other form of pitch estimation could also be used. Since pitch trackers usually incur a slight delay for accurate estimation (for **specptrk** the delay is printed by the **spectrum** unit), it is normal to delay the audio signal by the same amount so that **harmon2, 3, 4** can work from a fully concurrent estimate.

Example:

```
a1,a2     ins                                  ; get mic input
w1        spectrum   a1, .02, 7, 24, 12, 1, 3 ; and examine it
koct,kamp specptrk   w1, 1, 6.5, 9.5, 7.5, 10, 7, .7, 0, 3, 1
a3        delay      a1, .065                  ; allow for ptrk delay
a4        harmon2    a3, koct, 1.25, 0.75, 0, 6.9  ; output a fixed 6-4 harmony
          outs       a3, a4                    ; as well as the original
```

ar **grain** xamp, xcps, xdens, kampdev, kcpsdev, kgdur, igfn, iwfn, imaxdur

Generate a granular synthesis texture with variable amplitude, frequency, duration and density characteristics.

INITIALIZATION

igfn - table number of the grain periodic waveform. Normally a sine wave, but could be a sampled sound of any length.

iwfn - table number of the envelope applied to each grain. This can be any shape, and can be conveniently defined using GEN20.

imaxdur - maximum grain duration (in seconds) that can be generated. Limits the range of *kgdur*.

PERFORMANCE

This unit generates a succession of granular sound elements of amplitude *xamp*, frequency *xcps*, and granular density *xdens* grains per second. The amplitude and frequency of successive grains can be randomly varied within the (bipolar) deviations *kampdev* and *kcpsdev*; if these values are 0 there is no random variation in that dimension. The density can also be given a random component (e.g. added noise) via the *xdens* parameter; if this value is constant the result is synchronous granular synthesis, similar to **fof**.

Individual grains are made from cycles of the periodic waveform *igfn*, and each is made to start from a random position within that table. Each grain lasts a duration *kgdur,* and is shaped by the envelope function *igfn*. The k-varying duration is limited by *imaxdur*, and is clipped to that value if *kgdur* exceeds it.

The overall output amplitude of this generator is a dynamic function of the amplitude of each grain, the density of grains, their envelopes, and their durations.

The **grain** generator is based primarily on work and writings of Barry Truax and Curtis Roads. This implementation is by Paris Smaragdis.

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.

ar **pluck2*** ifrq, iamp, ipickup, ipluck, iaw0, iawPi, aamp

Output is a high-quality plucked string simulation using waveguide techniques

INITIALIZATION

PERFORMANCE

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 straightline 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 displacements up to 1 oct.
a1      oscil   5000, cpsoct(i1+ k1), 1
```

xr	**linrand**	krange[, ipol]
xr	**exprand**	krange[, ipol]
xr	**cauchy**	kalpha[, ipol]
xr	**poisson**	klambda
xr	**gauss**	krange
xr	**weibull**	ksigma, ktau
xr	**beta**	krange, kalpha, kbeta

Output is a series of random number with specified distribution over a certain range.

INITIALIZATION

ipol (optional) - polarity attribute. The value 1 denotes unipolar (0 to *krange*), and the value 2 denotes bipolar (-*krange* to +*krange*). The default is 2 (bipolar).

PERFORMANCE

All units are available in three different forms, determined by the output variable **xr** which can be of type **ir**, **kr** or **ar**. Units of type **ir** require **i**-type input, while units of type **kr** and **ar** can take **k**-type input or slower (i.e. can be either **i** or **k**).

linrand generates a linear distribution over the range. The unipolar form has a linear rolloff distribution; the bipolar form has a triangular rolloff distribution (symmetric about 0).

exprand generates an exponential distribution over the range.

cauchy generates a Cauchy distribution, in which *kalpha* controls the *spread* of values away from zero.

poisson generates a Poisson distribution, **positive** numbers only, with *klambda* denoting the mean.

gauss generates a Gaussian distribution, always over a **bipolar** range (-*krange* to +*krange*).

weibull generates a Weibull distribution, **positive** numbers only, in which *ksigma* controls the spread away from zero. If *ktau* = 1, the distribution is exponential; if *ktau* < 1, small numbers are favored; if *ktau* > 1, numbers near *ksigma* are favored.

beta generates a Beta distribution, **positive** numbers only (0 to *krange*). If *kalpha* < 1, numbers near 0 are favored; if *kbeta* < 1, numbers near *krange* are favored. If *kalpha* = 1 and *kbeta* = 1, uniform distribution will result. If both are > 1, a form of Gaussian distribution will result.

SIGNAL MODIFIERS

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

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) exponential decay
linenr, envlpxr - as above, except that the final segment is entered only on sensing a note release, and the note is then extended by the decay time.

INITIALIZATION

irise - rise time in seconds. A zero or neg value signifies no rise portion, and begins from *ifn* end.

idur - overall duration in seconds. A zero or negative value will cause initialization to be skipped; for **envlpx(r)** skipped initialization can also be induced by a zero *ifn* number.

idec - decay time in seconds. Zero means no decay.

irind (optional) - independence flag. If left zero, the release time (*idec*) will influence the extended life of the current note following a note-off. If non-zero, the *idec* time is quite independent of the note extension (see below). The default value is 0.

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

iatss - attenuation factor, by which the last value of the **envlpx** rise is modified during the note's pseudo "steady state." A factor > l causes an exponential growth, and < l an exponential decay. A 1 will maintain a true steady state at the last rise value. NB. 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. **envlpxr** (which has no idur), always uses the fixed per second rate. 0 is illegal.

iatdec - attenuation factor by which the closing "steady state" value is reduced exponentially over the decay period. This value must be positive and is normally of the order of .01. A large or 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 modifications are applied for the first *irise* seconds, and decay from time *idur* - *idec*. If these periods are separated in time there will be a "steady state" during which *amp* will be unmodified (**linen**) or modified by the first exponential pattern (**envlpx**). The decay segments can begin at any point in the pattern; they will go to zero in **linen** and tend asymptotically to zero in **envlpx**.

linenr, **envlpxr** are examples of the special Csound "**r**" units that contain a **note-off sensor** and **release time extender.** Unless made independent by *irind*, when each senses a score event termination or a MIDI noteoff, it will immediately extend the performance time of the current instrument by *idec* seconds, then begin a decay (as described above) from wherever it was at the time. These "**r**" units can also be modified by MIDI noteoff velocities (see **veloffs**). If the *irind* flag is on (non-zero), the overall performance time is unaffected by note-off and veloff data.

Mulitple "r" units. When two or more "r" units occur in the same instrument it is usual to have only one of them influence the overall note duration. This is normally the master amplitude unit. Other units controlling, say, filter motion can still be sensitive to note-off commands while not affecting the duration by making them independent (*irind* non-zero). Depending on their own *idec* (release time) values, independent "r" units may or may not reach their final destinations before the instrument terminates. If they do, they will simply hold their target values until termination. If two or more "r" units are simultaneously master, note extension is by the greatest *idec*.

Example 1:

```
        kamp   linenr   isig, .1, 2          ;fast rise, then slow decay of 2 on note-off (master)
        kvib   linenr   ivib, 1.5, .3, 1     ;slow rise, then quick decay to 0 on note-off
(indep).
```

Example 2:

```
        kenv   envlpxr      ksig,  .1,  .2,  7,  .25,  .01,  -.5
```

This applies an envelope to the signal *ksig*, rising in .1 secs by ftable 7 to its endpoint, then doing a *subito piano* aimed at one fourth that value, and continuing this more gently until sensing a noteoff, when it decays in .2 seconds to one one-hundredth of where it happened to be.

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 output 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 cps) determines the response curve's half -power point. Half power is defined as peak amp / root 2, or peak power / 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" aspects 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 (*iscl* = 1 only, *iscl* = 2 giving twice amplitude). This property is 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**.

ar	**filter1***	asig, kfreq, kdamp, imode[, istor]
ar	**filter2***	asig, kfreq, kdamp, imode[, istor]

Apply a resonance filter to an input signal according to mode control.

INITIALIZATION

imode - mode of filtering, determined by a coded value:

1	low pass resonance filter	4	EQ filter
2	bandpass resonance filter	5	highpass shelving
3	high pass resonance filter	6	lowpass shelving

istor (optional) - initial disposition of internal feedback data space. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

PERFORMANCE

These are multi-purpose filters, whose mode of operation is determined by Init-time data values, and whose performance behavior is determined by two k-rate inputs. The value *kfreq* in cps is both a resonance frequency (bandpass) and a resonance and cutoff frequency (lowpass, highpass). There is no direct bandwidth control as in **reson**; instead, the value *kdamp* is a damping factor that controls the Q of the filter (in an inverse way), with low values giving narrow-band (hi-Q) results and higher values producing broad-band (lo-Q) effects. The damping factor should lie in the range 0.01 to 0.99.

These filters have no automatic gain or rms control, and the overall gain will vary somewhat with *kdamp*. In **filter1** the peak gain (at *kfreq*) will stay constant as *kfreq* moves the resonance about, but may vary with *kdamp*. In **filter2** the DC gain is 1 when the *kdamp* is high (flat response, no resonance), then retreats below 1 as the damping factor is relaxed and a resonance forms at *kfreq*.

The two filters also differ in speed and behavior. **filter1** is fast, but has some non-linear features. It is most useful and reliable when *kdamp* is small (< .2); as *kdamp* increases, the resonance frequency is much less pronounced. Also, the resonance position is not quite linear, and at high frequencies it will be artificially displaced upwards, possibly going unstable. To improve linearity and stability, the resonance frequency is internally limited and remapped. By contrast, **filter2** is both accurate and stable throughout. However, it takes almost twice as long to run. Both filters are slowed slightly by changes in either *kfreq* or *kdamp*, but both are reasonably efficient for the task they perform. **filter1** is generally preferred for most practical applications.

Resonance filters are commonly used in music instrument synthesis because a peak that follows a time-varying cutoff frequency makes filter motion very *audible*. Its use to simulate attenuation of high-frequency energy following an onset is often overdone, but a musically useful resonance can be obtained by setting *kdamp* between .1 and .4. The resonance motion will be quite audible, and any overall gain will usually stay within a factor of 1.5 or 2 (**filter1** is usually louder, while **filter2** is usually softer). For increased gain control, use the rms feature of **balance**.

These units are loosely based on the Chamberlin filter (P. Dutilleux, "Simple to Operate Digital Time Varying Filters", 86th AES Convention, 1989, and J. Dattorro, “Effect Design”, JAES 45/9, Sept 1997), but the versions here are reworked for more stability and similarity.

Example:

```
a1      loscil1     kamp, icps, ifn             ; read sampled data
kfrq    expon       icps * 10, .5, icps * 5     ;   and over the 1st .5 seconds
```

```
a2      filter2         a1, kfrq, .3, 1                 ;   quickly damp the highs
```

ar **filter4*** asig, kfc1, kfc2, imode[, istor]

Apply a 4-stage cascade filter to an input signal according to mode control.

INITIALIZATION

imode - mode of cascade filtering, with cut frequencies *kfc1*, *kfc2* applied as follows:

0	2 low pass, 2 high pass
1	3 low pass, 1 high pass
2	2 low pass, 2 low pass
3	3 low pass, 1 low pass

istor (optional) - initial disposition of internal feedback data space. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

PERFORMANCE

This is a direct cascade of four simple low-pass and high-pass filters, aimed at emulating some early filter-banks found in the music industry. The two cut frequencies *kfc1*, *kfc2* are applied to sub-groups within the cascade. For cut frequencies applied to low-pass sections, the cut-off represents the 3-dB down point in the transfer function compared with unity gain at DC. For cut frequencies applied to high-pass sections, the cut-off represents the 3-dB down point in the transfer function compared with unity gain near the Nyquist (**sr**/2); the high-pass filter has zero transmission at the actual Nyquist. The combination of low-pass and high-pass sections will produce a band-pass effect.

When either low-pass or high-pass sections are themselves in cascade, the normal 6-dB per octave rolloff becomes 12, 18 or 24 dB. While low-pass *kfc* values can range from 0 to **sr**/2, high-pass *kfc* values are capped at (can go no higher than) 1600 Hz. For *kfc* values < **sr**/2, the filter is always partly subtractive, i.e. all component frequencies emerge with a gain <= 1. However, as a special feature, **filter4** is permitted to have low-pass *kfc* values > **sr**/2: specifically, when the low-pass *kfc* is in the range **sr**/2 to **sr**, the subtractive filtering effect is gradually eliminated, until at *kfc* = **sr** the filter has a flat response with unity gain throughout (i.e. the signal passes through unmodified).

Although the cascaded **filter4** is not as flexible as the resonance units like **filter1**, it has many useful and audible features, and is no more expensive to operate.

Example:

```
kfc1    expon   440, 10, 44100                  ; sweep a cutoff above the Nyquist
a1      buzz    1000, 220, 100, gisin
a1      filter4 a1, kfc1, kfc1, 3               ; and watch the spectrum go flat
        dispfft a1, .1, 1024, 1
```

```
ar      butterhp    asig, kcps[, istor]
ar      butterlp    asig, kcps[, istor]
ar      butterbp    asig, kcf, kbw[, istor]
ar      butterbr    asig, kcf, kbw[, istor]
```

Modify an audio signal using a high-pass, low-pass, band-pass or band-reject Butterworth filter.

INITIALIZATION

istor (optional) - initial disposition of the internal data space. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

PERFORMANCE

butterhp, butterlp are high-pass and low-pass filters with a cutoff frequency *kcps*.

butterbp, butterbr are band-pass and band-reject filters with center frequency *kcf* and bandwidth *kbw* (both in cps).

These Butterworth filters are similar to their **reson** counterparts. They are somewhat slower to run, but they exhibit superior characteristics with an almost flat passband and good stopband attenuation.

The implementations are by Paris Smaragdis.

Example:

```
asig    rand        10000           ; generate a white-noise source
alp     butterlp    asig, 2000      ; cut the frequencies above 2000 Hz
abr     butterbr    alp, 800, 200   ;       and those from 700 to 900 Hz
```

krmsr,krmso,kerr,kcps		**lpread**	ktimpnt, ifilcod[, 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

ifilcod - integer or character-string denoting a control-file (reflection coefficients + four parameter values) derived from n-pole linear predictive spectral analysis of a source audio signal. A positive integer denotes an indexed filename in the Orchestra (see **strset**); a negative integer denotes the suffix of a file *lp.m;* a character-string (in double quotes) gives the filename itself. Filenames are optionally fullpath. If not fullpath, the file is sought first in the current directory, then in that of the environment variable SADIR (if defined). Memory usage depends on the size of the file, which is held entirely in memory during computation but shared by multiple calls (see also **adsyn, pvoc**).

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** 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* (0 to 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 nonpitched 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.

ar **cross*** acar, amod, kdepth, kvol, ilocut, ihicut

Filter a carrier signal with the spectrum of a modifier signal.

INITIALIZATION

ilocut, *ihicut* - low frequency and hi-frequency spectral rolloff points (in cps).

PERFORMANCE

This unit performs cross synthesis between two signals by filtering the first with the spectral shape of the second. This is done using Fourier transforms, wherein the spectral coefficients of the carrier are modified by a scaled and normalized spectrum of the modifier.

The degree of modification can be controlled by a spectral depth factor *kdepth* (0 to 1 or above). Zero will give no audible modification, .8 - 1.0 will be normal, and above that will be exaggerated. Modification can also be affected by a loudness factor *kvol* (0 to 1 or above), by which the changing energy of the modifier will influence the loudness of the reconstructed carrier. Both factors can be varied during performance. Typical values are .9 and .75 respectively.

Since this process involves multiplying two scaled amplitude values in each frequency bin, **cross** can produce some high amplitude samples when spectral peaks coincide. To counter this, scale down the *acar* input or its output when pushing *kdepth* or *kvol* beyond normal.

Spectral cross synthesis is most effective when the carrier is broad-band (has good highs) and the modifier has strong and varying spectral peaks. Noisy music and formant resonant speech are good examples; whispered speech is especially effective. The formants of the modifier can be given extra focus by a spectral envelope shaper, whose low-frequency and hi-frequency rolloffs begin at *ilocut* and *ihicut* respectively. Suitable values here are 400 and 4000 Hz, but these will depend on the quality of the modifying signal.

cross is a computationally demanding opcode, currently costing 19 mips at a 16KC sampling rate. The cost and audio quality are somewhat affected by the control rate: the analysis window is less than or equal to 4 kperiods (longest power-of-two samples within), and should hold a full cycle of the lowest frequency present. kr=100 is convenient; greater may tend to boom on the lows, and less will begin to sound reverberant.

Example:

```
a1      soundin     "mytune.aif"
a2      in                                  ; voice input
a3      cross       a1, a2, .75, .5, 400, 4000
```

kr	**rms**	asig[, ihp, istor]
ar	**gain**	asig, krms[, ihp, istor]
ar	**balance**	asig, acomp[, ihp, istor]
ar	**dbgain***	asig, kdb[, iatktim, 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.

iatktim (optional) – flag indicating a zero or positive attack-time. If zero, the internal interpolator will move immediately to the first *kdb* point received; if non-zero, the interpolator will go from zero to that point during the first control period. The default value is 1 (interpolated rise).

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.

dbgain is a signal amplitude modifier whose gain level is specified in *decibels*. The incoming *kdb* is first converted to an amplitude multiplier equivalent (see **ampdb**), and the a-rate *interpolated* value is then used as an amplitude scaler of the incoming audio signal. Thus, k-rate control signals will not cause "zipper noise". The interpolation proceeds identically for each control period except the first, where an additional option *iatktim* allows the first interpolation target to be immediate. *kdb* scaling can be positive (amplifying) or negative (reducing); a zero *kdb* will cause no change to the incoming signal.

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
```

ar **compress*** aasig, acsig, kthresh, kloknee, khiknee, kratio, katt, krel, ilook

Compress, limit, expand, duck or gate an audio signal.

INITIALIZATION

ilook - lookahead time in seconds, by which an internal envelope release can sense what is coming. This induces a delay between input and output, but a small amount of lookahead improves the performance of the envelope detector. Typical value is .05 seconds, sufficient to sense the peaks of the lowest frequency in *acsig*.

PERFORMANCE

This unit functions as an audio compressor, limiter, expander, or noise gate, using either soft-knee or hard-knee mapping, and with dynamically variable performance characteristics. It takes two audio input signals, *aasig* and *acsig*, the first of which is modified by a running analysis of the second. Both signals can be the same, or the first can be modified by a different controlling signal.

compress first examines the controlling *acsig* by performing envelope detection. This is directed by two control values *katt* and *krel*, defining the attack and release time constants (in seconds) of the detector. The detector rides the peaks (not the RMS) of the control signal. Typical values are .01 and .1, the latter usually being similar to *ilook*.

The running envelope is next converted to decibels, then passed through a mapping function to determine what compresser action (if any) should be taken. The mapping function is defined by four decibel control values. These are given as positive values, where 0 db corresponds to an amplitude of 1, and 90 db corresponds to an amplitude of 32768.

kthresh - sets the lowest decibel level that will be allowed through. Normally 0 or less, but if higher the threshold will begin removing low-level signal energy such as background noise.

kloknee, *khiknee* - decibel break-points denoting where compression or expansion will begin. These set the boundaries of a soft-knee curve joining the low-amplitude 1:1 line and the higher-amplitude compression ratio line. Typical values are 48 and 60 db. If the two breakpoints are equal, a hard-knee (angled) map will result.

kratio - ratio of compression when the signal level is above the knee. The value 2 will advance the output just one decibel for every input gain of two; 3 will advance just one in three; 20 just one in twenty, etc. Inverse ratios will cause signal expansion: .5 gives two for one, .25 four for one, etc. The value 1 will result in no change.

The actions of **compress** will depend on the parameter settings given. A hard-knee compressor-limiter, for instance, is obtained from a near-zero attack time, equal-value break-points, and a very high ratio (say 100). A noise-gate plus expander is obtained from some positive threshold, and a fractional ratio above the knee. A voice-activated music compressor (ducker) will result from feeding the music into *aasig* and the speech into *acsig*. A voice de-esser will result from feeding the voice into both, with the *acsig* version being preceded by a band-pass filter that emphasizes the sibilants. Each application will require some experimentation to find the best parameter settings; these have been made k-variable to make this practical.

Example:

```
aout    compress    amus, avoc, 0, 40, 60, 3, .1, .5, .02    ; voice-activated compressor
```

; with low-level sensitivity

ar **distort*** asig, kdist, ifn[, ihp, istor]

Distort an audio signal via waveshaping and optional clipping.

INITIALIZATION

ifn – table number of a waveshaping function with *extended guard point*. The function can be of any shape, but it should pass through 0 with positive slope at the table mid-point. The table size need not be large, since it is read with interpolation.

ihp (optional) – half-power point (in cps) of an 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

This unit distorts an incoming signal using a waveshaping function *ifn* and a distortion index *kdist*. The input signal is first compressed using a running *rms,* then passed through a waveshaping function which may modify its shape and spectrum. Finally it is rescaled to approximately its original power.

The amount of distortion depends on the nature of the shaping function and on the value of *kdist,* which generally ranges from 0 to 1. For low values of *kdist*, we should like the shaping function to pass the signal almost unchanged. This will be the case if, at the mid-point of the table, the shaping function is near-linear and is passing through 0 with positive slope. A line function from -1 to +1 will satisfy this requirement; so too will a sigmoid (sinusoid from 270 to 90 degrees). As *kdist* is increased, the compressed signal is expanded to encounter more and more of the shaping function, and if this becomes non-linear the signal is increasingly "bent" on read-through to cause distortion.

When *kdist* becomes large enough, the read-through process will eventually hit the outer limits of the table. The table is not read with wrap-around, but will "stick" at the end-points as the incoming signal exceeds them; this introduces **clipping**, an additional form of signal distortion. The point at which clipping begins will depend on the complexity (rms-to-peak value) of the input signal. For a pure sinusoid, clipping will begin only as *kdist* exceeds 0.7; for a more complex input, clipping might begin at a *kdist* of 0.5 or much less. *kdist* can exceed the clip point by any amount, and may be greater than 1.

The shaping function can be made arbitrarily complex for extra effect. It should generally be continuous, though this is not a requirement. It should also be well-behaved near the mid-point, and roughly balanced positive-negative overall, else some excessive DC offset may result. The user might experiment with more aggressive functions to suit the purpose. A generally positive slope allows the distorted signal to be mixed with the source without phase cancellation.

distort is useful as an effects process, and is usually combined with reverb and chorusing on effects busses. However, it can alternatively be used to good effect within a single instrument.

Example:

```
gifn    ftgen    0,0, 257, 9, .5,1,270              ; define a sigmoid, or better …
gifn    ftgen    0,0, 257, 9, .5,1,270,1.5,.33,90,2.5,.2,270,3.5,.143,90,4.5,.111,270

kdist   line     0, 10, 1.2                         ; and over 10 seconds
aout    distort  asig, kdist, gifn                  ; gradually increase the distortion
```

kr	**downsamp**	asig[, iwlen]
ar	**upsamp**	ksig
ar	**interp**	ksig[, iatktim, 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.

iatktim (optional) – flag indicating a zero or positive attack-time. If zero, the interpolator will move immediately to the first *kdb* point received; if non-zero, the interpolator will go from zero to that point during the first control period. The default value is 1 (interpolated rise).

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 (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 suppress 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 (with optional control over the first k-period). **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 **octup*** asig, isegtim
ar **octdown*** asig, isegtim

Pitch-shift a signal up or down one octave.

INITIALIZATION

isegtim - time in seconds of each windowed segment.

PERFORMANCE

These units perform an elementary pitch shift using windowed and overlaid sound fragments. Voice formants are *not* preserved (c.f. **harmon, harmon2, 3, 4**). Rather, the octave shifts are due to resampling the input stream prior to windowing and redistribution. The *isegtim* durations depend mostly on the direction of pitch shift, and partly on the audio content. Suggested practical values are .05 for **octup** and .3 for **octdown**, but these should be set by experiment.

ar	**delayr**	idlt[, istor]
	delayw	asig
ar	**delay**	asig, idlt[, istor]
ar	**delay1**	asig[, istor]
ar	**vdelay**	asig, xdlt, imaxdlt[, 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, imaxdlt - requested or maximum delay time in seconds. This can be as large as available memory will permit. The space required for *n* seconds of delay is *4n* * **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/**sr**" but is more efficient in both time and space. This unit is particularly useful in the fabrication of generalized non-recursive filters.

vdelay is a variable-length delay line whose delay time *xdlt* can range from 0 sample-periods to *imaxdlt* seconds. *xdlt* can be either k-rate or a-rate. k-rate *xdlt* is convenient for slow-changing and medium-changing delay times, such as in doppler effects, chorusing and flanging; a-rate *xdlt* is appropriate for faster changing delay times, such as pitch-shift effects. Both versions use interpolated readout. If fast changes still result in audible noise, try **deltapi** as an alternative. Note that all values of *xdlt* should be positive, and should remain within the *imaxdlt* limit. If the boundary is crossed, **vdelay** will still generally produce output but will "click" at each boundary wrap-around.

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 fastpaced 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]
ar	**reverb2**	asig, krvt, khfabs[, istor]

An input signal is reverberated for *krvt* seconds with "colored" (**comb**), flat (**alpass**), or "natural room" (**reverb, reverb2**) frequency response.

INITIALIZATION

ilpt - loop time in seconds, which determines the "echo density" of the reverberation. This sets the "color" of the **comb** filter whose frequency response curve will contain *ilpt* * **sr**/2 peaks spaced evenly between 0 and **sr**/2. Loop time is limited only by memory, an *n* second loop requiring *4n* * **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 the 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 60dB down from its original amplitude). Output from a **comb** filter will appear only after *ilpt* seconds; **alpass** output will begin to appear immediately.

reverb is composed of four **comb** filters in parallel followed by two **alpass** units in series. **reverb2** is composed of six parallel **comb-lopass** filters followed by five **alpass** units in series. While the first is fast and rather bright, the second is slower but also more natural due to a lopass filter in the comb loops that simulates the high-frequency energy absorption of air. The parameter *khfabs* (0 - 1) controls the degree of high-frequency absorption, from none to full.

In both reverb units, internal looptimes are pre-set for optimal "natural room response." Memory requirements are proportional to the sampling rate, the first requiring 3K words per 10KC and the second 5K words per 10KC. Alternative reverberators may be built from similar operators.

Reverberator units are intended to represent only *sound reflections* (from walls and ceilings). Output will begin to appear after about .05 seconds (time of first reflection), and the amplitude is unlikely to grow beyond three-fourths of the original. Consequently it is normal to output both the *source* and the *reverberated* signal. Since the reverberated sound will persist long after the cessation of direct instrument sound, it is normal to put a **reverb** unit in a separate instrument which runs continuously. Sound can then be fed into it via a *global variable*, as shown below:

```
ga1     init    0                       ; init an audio receiver/mixer

        instr   1                       ; instr (there may be any number of these)
a1      oscili  8000, cpspch(p5), 1     ; generate a source signal
        out     a1                      ; output the direct sound
ga1     =       ga1 + a1                ; and add to audio receiver
        endin

        instr   99                      ; (highest instr number executed last)
a3      reverb  ga1, 1.5                ; reverberate whatever is in ga1
        out     a3                      ; and output the result
```

```
ga1     =       0                       ; empty the receiver for the next pass
        endin
```

a1,a2 **lrghall32*** asig
a1,a2 **lrghall44*** asig

Apply large-hall reverberation to a mono input signal to produce stereo independent output.

PERFORMANCE

These are both variants of a *fixed-algorithm* reverberator, designed to produce high-quality stereo output via two uncorrelated output channels. Like other Csound units, these both have an Initialization phase. However, the internal parameters used are *not* accessible to the user, but have been set in advance. The two units are *pretuned* to operate best at two specific sampling rates—32 KHz and 44.1 KHz, respectively. Both will run at other rates without error, but the reverberation will sound best at the rates prescribed.

The algorithms will first pass the monaural input through a low-pass filter, then two allpasses in series, an allpass nested within an allpass, two allpasses nested within an allpass, and finally a lowpass feedback. The uncorrelated ouputs derive from two independent sets of four taps into the combined delay space. There is no tap on the direct input path.

These reverberators may be used in similar fashion to **reverb** and **reverb2**, and are typically used to process data on an effects bus. The direct signal is not included in the output. The algorithms are high-quality but expensive: the tuned parameter settings require 10K and 15K words of delay space respectively, and the two units will cost 13 mips and 20 mips respectively to run at the intended sampling rates.

The algorithms are due to W.G. Gardner, *The Virtual Acoustic Room*, MIT Master's Thesis (1992), and so too is the initial version of the SHARC assembler code.

Example:

```
gar     fxsend      asig, .3            ; write to an effects bus

a1,a2   lrghall32   gar                 ; and reverberate the contents
        outs        a1, a2
gar     =           0
```

a1	**chorus1***	asig, krat1, krat2, krat3, idel1, idel2, idel3, ipred, idpth, ifdbk, ifsin
a1,a2	**chorus2***	asig, krat1, krat2, krat3, krat4, idel1, idel2, idel3, idel4, ipred, idpth, ifdbk, ifsin
a1,a2	**chorus3***	asig, krat1, krat2, krat3, krat4, idel1, idel2, idel3, idel4, ipred, idpth, ifdbk, ifsin

Perform chorus effects processing on an audio signal.

INITIALIZATION

idel1,idel2, idel3,idel4 - delay times in seconds for the mono (3 path) or stereo (2+2 path) signal storage paths. Times should be dissimilar, and normally from .03 to .1 seconds.

ipred - predelay in seconds. An optional delay-line supplement, through which the input *asig* first passes before going to the above delay lines. The value 0 will disable.

idpth - chorusing depth.

ifdbk - feedback factor. This fraction of the chorused output is mixed in with the incoming signal.

ifsin - ftable number of a stored sinwave, used to vary the index of the delay-line readouts. The sine table is read with interpolation, so need not be large.

PERFORMANCE

These units boost an incoming signal with various delayed versions of itself. The delay times are continuously varied, causing small undulating pitch shifts in the added components and a chorus effect in the overall output. The output contains both the source and the delayed signals.

The continuous variation of the delays is a function of the rates *krat1*, *krat2*, .. which cause the readout to traverse the delay lines with sinusoidal oscillatory motion. Rates are expressed in cycles per second, and are normally slow, similar, but not identical; values in the range of .2 to .5 cps are typical. The degree of pitch shift is a function of each delay-line extent and the rate at which its varied readout is being alternately advanced or retarded over this extent. The signal delay paths are cumulative, so the readout pointers will never cross or be coincident; however, the pitch deflections will cross each other as they go from positive to negative.

While **chorus1** gives monaural output, **chorus2** (the 4 controls are applied 1,2 to channel 1 and 3,4 to channel 2) can be set to give stereo uncorrelated output. Both units can be further enhanced by a non-zero feedback factor; in the stereo unit the feedback moves left channel output back into right and right back into left. The user should experiment with various settings to achieve different results. **chorus3** is essentially an assembler version of **chorus2**, taking about 4.8 mips instead of 7.2 mips at 32 KC.

Example:

```
a1      chorus1      asig, .43, .37, .31, .05, .073, .094, 0, 0, .2, 1
```

This sets three dissimilar delay lines, traversed at three dissimilar rates (longest slowest), with feedback but no predelay.

a1	**flange1***	asig, krate, idel, ifdbk, ifn
a1,a2	**flange2***	asig, krate, idel, idiff, ifdbk, ifn

Perform flange effects processing on an audio signal.

INITIALIZATION

idel - delay line length in seconds.

idiff- phase difference between stereo flanges, expressed as a fraction, 0 - .999.

ifdbk - feedback factor. Fraction of flanged audio output to be mixed with the incoming signal.

ifn - ftable number of a stored sinewave, used to vary the index of the delay-line readouts. The sine table is read with interpolation, so need not be large.

PERFORMANCE

These units will augment an audio signal with delayed versions of itself in the form of flange effects. A small time difference between a source input and its delayed addition will create a pitched focus at the frequency 1/timediff, and this pitch can be moved over a wide range as the readout slowly varies its point of scan. The readout is varied with the assistance of a stored periodic function, normally a single sinusoid, and the cyclic change induced by *krate* normally spans several seconds. The resultant flanged audio can optionally be added back into the incoming signal for further enhanced effect.

flange2 differs from **flange1** by maintaining two distinct delay lines, with readouts and pitched effects differing by some initial phase difference (e.g. .25). The flanged output can also optionally be added back into the incoming signal, although unlike **chorus2** the feedback is not interchanged between the two channels.

Example:

```
a1,a2   flange2          asig, 5, .03, .25, .2, 1
```

OPERATIONS USING SPECTRAL DATA-TYPES

These units generate and process a non-standard signal data type *wsig*, which is an exponentially-spaced frequency-domain (spectral) representation of a given control or audio signal. The data type is self-defining, and its contents are not processable by any other Csound units. Its purpose is to enable perceptually-based analysis of audio input. The units below are experimental, and subject to change between releases; they will also be joined by others later.

wsig **spectrum** xsig, iprd, iocts, ifrqs, iq[, ihann, idbout, idisprd, idsines]

This unit will generate a constant-Q, exponentially-spaced DFT across all octaves of a multiply-downsampled control or audio input signal.

INITIALIZATION

ihann (optional) - apply a hamming or hanning window to the input. The default value is 0 (hamming window).

idbout (optional) - coded conversion of the DFT output: 0 = magnitude, 1 = dB, 2 = mag squared, 3 = root magnitude. The default value is 0 (magnitude).

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

idsines (optional) - if non-zero, display the hamming or hanning windowed sinusoids used in DFT filtering. The default value is 0 (no sinusoid display).

PERFORMANCE

This unit first puts signal *asig* or *ksig* through *iocts* of successive octave decimation and downsampling, and preserves a buffer of down-sampled values in each octave (optionally displayed as a composite buffer every *idisprd* seconds). Then at every *iprd* seconds, the preserved samples are passed through a filter bank (*ifrqs* parallel filters per octave, exponentially spaced, with frequency/bandwidth Q of *iq*), and the output magnitudes optionally converted (*idbout*) to produce a band-limited spectrum that can be read by other units.

The stages in this process are computationally intensive, and computation time varies directly with *iocts*, *ifrqs*, *iq*, and inversely with *iprd*. Settings of *ifrqs* = 12, *iq* = 10, *idbout* = 3, and *iprd* = .02 will normally be adequate, but experimentation is encouraged. *ifrqs* currently has a maximum of 120 divisions per octave. For audio input, the frequency bins are tuned to coincide with A440.

This unit produces a self-defining spectral datablock *wsig*, whose characteristics used (*iprd, iocts, ifrqs, idbout*) are passed via the data block itself to all derivative *wsigs*. There can be any number of **spectrum** units in an instrument or orchestra, but all *wsig* names must be unique.

Example:

```
asig    in                                  ; get external audio
```

```
        wsig    spectrum        asig, .02, 6, 12, 33, 0, 1, 1     ; downsample in 6 octs & calc a 72
pt
                                                                  ;   dft (Q 33, dB out) every 20 msecs
```

wsig	**specaddm**	wsig1, wsig2[, imul2]
wsig	**specdiff**	wsigin
wsig	**specscal**	wsigin, ifscale, ifthresh
wsig	**spechist**	wsigin
wsig	**specfilt**	wsigin, ifhtim

INITIALIZATION

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

PERFORMANCE

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.

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*.

spechist - accumulate the values of successive spectral frames. At each new frame of *wsigin*, the accumulations-to-date in each magnitude track are written to the output spectrum. This unit thus provides a running *histogram* of spectral distribution.

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. It may also be used as a time-attenuated running *histogram* of the spectral distribution.

Example:

```
wsig2  specdiff    wsig1          ; sense onsets
wsig3  specfilt    wsig2, 2       ; absorb slowly
       specdisp    wsig2, .1      ; & display both spectra
       specdisp    wsig3, .1
```

koct, kamp **specptrk** wsig, kvar, ilo, ihi, istrt, idbthresh, inptls, irolloff[, iodd, iconfs, interp, ifprd, iwtflg]

Estimate the pitch and amplitude of the most prominent complex tone in the spectrum.

INITIALIZATION

ilo, ihi, istrt - pitch range conditioners (low, high, and starting) expressed in decimal octave form.

idbthresh - energy threshold (in decibels) for pitch tracking to occur. Once begun, tracking will be continuous until the energy falls below one half the threshold (6 db down), whence the *koct* and *kamp* outputs will be zero until the full threshold is again surpassed. *idbthresh* is a guiding value. At initialization it is first converted to the *idbout* mode of the source spectrum (and the 6 db down point becomes .5, .25, or 1/root 2 for modes 0, 2 and 3). The values are also further scaled to allow for the weighted partial summation used during correlation. The actual thresholding is done using the internal weighted and summed *kamp* value that is visible as the second output parameter.

inptls, irolloff - number of harmonic partials used as a matching template in the spectrally-based pitch detection, and an amplitude rolloff for the set expressed as some fraction per octave (linear, so don't roll off to negative). Since the partials and rolloff fraction can affect the pitch following, some experimentation will be useful: try 4 or 5 partials with .6 rolloff as an initial setting; raise to 10 or 12 partials with rolloff .75 for complex timbres like the bassoon (weak fundamental). Computation time is dependent on the number of partials sought; the maximum number is 16.

iodd (optional) - if non-zero, employ only odd partials in the above set (e.g. *inptls* of 4 would employ partials 1,3,5,7). This improves the tracking of some instruments like the clarinet. The default value is 0 (employ all partials).

iconfs (optional) - number of confirmations required for the pitch tracker to jump an octave, pro-rated for fractions of an octave (i.e. the value 12 implies a semitone change needs 1 confirmation (two hits) at the spectrum generating *iprd*). This parameter limits spurious pitch analyses such as octave errors. A value of 0 means no confirmations required; the default value is 10.

interp (optional) - if non-zero, interpolate each output signal *(koct, ksmp)* between incoming *wsig* frames. The default value is 0 (repeat the signal values between frames).

ifprd (optional) - if non-zero, display the internally computed spectrum of candidate fundamentals. The default value is 0 (no display).

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

PERFORMANCE

At note initialization this unit creates a template of *inptls* harmonically related partials (odd partials if *iodd* non-zero) with amplitude rolloff to the fraction *irolloff* per octave. At each new frame of *wsig*, the spectrum is cross-correlated with this template to provide an internal spectrum of candidate fundamentals (optionally displayed). A likely pitch/amp pair (*koct*, *kamp*, in decimal octave and summed *idbout* form) is then estimated. *koct* varies from the previous *koct* by no more than plus or minus *kvar* decimal octave units; it is also guaranteed to lie within the hard limit range *ilo - ihi* (decimal octave low and high pitch). *kvar* can be dynamic, e.g. onset

amp dependent. Pitch resolution uses the originating spectrum *ifrqs* bins/octave, with further parabolic interpolation between adjacent bins. Settings of *root magnitude*, *ifrqs* = 24, *iq* = 15 should capture all the inflections of interest. Between frames, the output is either repeated or interpolated at the K-rate.

Example:

```
a1,a2   ins                                         ; read a stereo clarinet input
krms    rms         a1, 20                          ; find a monaural rms value
kvar    =           0.6 + krms / 8000               ;   & use to gate the pitch variance
wsig    spectrum    a1, .01, 7, 24, 15, 0, 3 ; get a 7-oct spectrum, 24 bins/oct
        specdisp    wsig, .2                        ; display this, and now estimate
koct,ka specptrk    wsig, kvar, 7.0, 10, 9, 20, 4, .7, 1, 5, 1, .2          ; the pch and amp
aosc    oscil       ka * ka * 10, cpsoct(koct), 2   ;   & generate a new tone with these
koct    =           (koct < 7.0 ? 7.0 : koct)       ; replace non pitch with low C
        display     koct - 7.0, .25, 20             ;   & display the pitch track
        display     ka, .25, 20                     ;   plus the summed root mag
        outs        a1, aosc                        ; output 1 original and 1 new track
```

ksum **specsum** wsig[, interp]
specdisp wsig, iprd[, iwtflg]

INITIALIZATION

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

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

PERFORMANCE

specsum - sum the magnitudes across all (iocts * ifrqs) channels of the incoming spectrum. At each new frame of *wsig*, the *idbout* coded magnitudes are arithmetically summed and released as a scalar *ksum* signal. Although such simple summing of dB or root magnitude values is not fully meaningful, these can be taken as very simple substitutes for the simultaneous masking and energy summation that occurs in mammalian hearing, and can thus serve as a moment to moment measure of perceived loudness, especially if temporal energy integration has also been modeled (see **specfilt**). Between frames, the output is either repeated or interpolated at the K-rate.

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

SENSING & CONTROL

midiout* kamp, koct, iampsens, ibendrng, ichan

Create a midi output stream based on amplitude and pitch patterns.

INITIALIZATION

iampsens - sensitivity to amplitude regrowth (0 - 1). When an amp has fallen below this fraction of its maximum, a regain of half the loss will trigger a new event. Typically about .6; a 0 will disable this feature.

ibendrng - range of pitch-bend (in semitones) by which a note may be varied without requiring a new event. Typical values are 2 and 12 semitones (both up and down). 0 will inhibit pitch-bend.

ichan - midi channel number (1 - 16) which will be encoded in each generated midi event.

PERFORMANCE

This unit takes amplitude and decimal-octave pitch data and converts them into a sequence of midi commands whose ultimate destination is set by flag **-G**. The amplitude is conveyed directly as midi velocity, and the oct data is converted to an appropriate midi note number. The two input signals may be derived in any manner, but are most conveniently supplied by **specptrk**.

A midi note-on event requires two things: an amplitude that has reached a local peak, and a non-zero *koct* value. These values then become the velocity and equivalent note-number of a note-on event. If *ibendrng* is non-zero, an accompanying pitch-bend command is also generated.

A note-off will occur when either *kamp* or *koct* becomes zero. It will also occur if pitch-bend is off and the incoming *koct* requires a different note number, in which case the note-off will be followed by a new note-on with revised velocity and pitch.

A note-off is also issued when *kamp* falls below the *iampsens* fraction of its maximum then regains half the loss, thus making way for a new note-on when the gain reaches a local peak. This will track rapid events (like flutter tonguing) with a series of note-off / note-on pairs.

If pitch-bend is enabled, any change in the *koct* value of an active note will induce a new pitch-bend command on each control period. To avoid unnecessary detail, set the krate low, or keep the *koct* value quantized (as in **specptrk** with no k-period interpolation).

Mapping *kamp* values to midi velocity raises a difficult issue, since the inverse is not defined in the midi spec. (Csound simply allows users to define velocity-to-amplitude via tables, see **ampmidi**.) The literal use of *kamp* as velocity in **midiout** implies not only that *kamp* is truncatable (0 - 127), but is already velocity-meaningful. Convenient forms of compressed-range amplitude data are decibels and root magnitude, both of which are optionally provided by **specptrk**.

Example:

```
a1      in                                      ; from audio input
w1      spectrum    a1, .02, 7, 24, 12, 1, 3    ; get spectrum in root-mag,
```

```
koct, kamp    specptrk    w1, 1, 6.3, 8.9, 7.5, 10, 7, .7, 0, 3, 0    ; pitch track &
              midiout     kamp, koct, 0, 2, 1    ; make midi events, with pitch-bend
```

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          ; and 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. It operates only in the presence of the **csound -t** flag. When that flag is set, scored events will be performed from their uninterpreted p2 and p3 (beat) parameters, initially at the given command-line tempo. When a **tempo** statement is activated in any instrument (*ktempo* > 0.), the operating tempo will be adjusted 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
```

```
aramp  iftime*      idgrlvl, label[, idec, iris][, iprop]
acum   timegate*    aramp, anew, acum
```

Sense realtime processor overload and modify the instrument complexity.

INITIALIZATION

idgrlvl - danger level (percentage), above which this unit will emit a rampout function, then begin skipping opcodes up to *label*..

idec, iris (optional) - decay time and rise time (in seconds) of the generated ramping function that precedes and follows skipping. The default time is .05 seconds.

iprop (optional) - proportion by which the skipped code was contributing to the accumulated output. The default proportion is .5.

PERFORMANCE

These units allow a compute-intensive instrument to "lighten up" on sensing that audio synthesis is falling behind realtime. The danger threshold is based on an internal running report of CPU usage. If the CPU starts to fall behind (i.e. the dac buffer is being drained faster than it is being replenished), an internal danger level is posted so that these units can take evasive action.

The two units are used as a pair, the target *label* of the first being the opcode that immediately follows the second. When danger is sensed, **iftime** emits a brief ramp before beginning to jump. During this period **timegate** uses the ramp values to modify the mix of new signal to previously accumulated signal, so that when jumping begins it will be acoustically seamless. On sensing that the danger is over, the two units begin the reverse process of reintroducing the ommited signal.

This technique is a musically practical alternative to "voice stealing". Instead of removing lines from a polyphonic thread on overload, these units enable instruments to be rich when played alone yet progressively thinner (more efficient) as the music itself thickens. It is known that the human ear is not as sensitive to the timbral complexity of every voice in a thick texture, but it can notice if a contrapuntal line has entirely disappeared. These units provide a way of dealing with that.

Example:

```
        acum    =           a1 + a2
        aramp   iftime      95, direct          ; if we have the processor time
        a3      oscil       kamp, icps3, ifn    ;   do one more oscil
        acum    timegate    aramp, a3, acum     ;   and fade it in or out
direct:         out         acum
```

SIGNAL INPUT & OUTPUT

a1	**in**	
a1, a2	**ins**	
a1, a2, a3, a4	**inq**	
a1, a2, a3, a4, a5, a6	**inh***	
a1	**soundin**	ifilcod[, iskptim][, iformat]
a1, a2	**soundin**	ifilcod[, iskptim][, iformat]
a1, a2, a3, a4	**soundin**	ifilcod[, iskptim][, iformat]
	out	asig
	outs1	asig
	outs2	asig
	outs	asig1, asig2
	outq1	asig
	outq2	asig
	outq3	asig
	outq4	asig
	outq	asig1, asig2, asig3, asig4
	outh1*	asig
	outh2*	asig
	outh3*	asig
	outh4*	asig
	outh5*	asig
	outh6*	asig
	outh*	asig1, asig2, asig3, asig4, asig5, asig6
	soundout*	asig1, ifilcod[, iformat]
	soundouts*	asig1, asig2, filcod[, iformat]

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

INITIALIZATION

filcod - integer or character-string denoting the source soundfile name. A positive integer denotes an indexed filename in the Orchestra (see **strset**); a negative integer denotes a file **soundin**.*filcod*; a character-string (double quotes, spaces permitted) gives the filename itself. A filename is optionally a full pathname. If not full path, the file is sought first in the current directory, then in that given by the environment variable SSDIR (if defined) then by 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 (high-order 8 bits of a 16-bit integer), 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, inh** - copy the current values from the standard audio input buffer. If the command-line flag **-i** is set, sound is read continuously from the audio input stream (e.g. adc (microphone) 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 controlled by the number of result cells, a1, a2, etc., which must match that of the input file. 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, outh** 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, quad or hex) 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.

soundout, **soundouts** will write audio output to an arbitrary file. There can be any number of these in an orchestra, but they should not attempt writing to the same file.

a1,a2 **hostin***
hostout* asig, iprd

Receive unsolicited audio from the host, or send same.

INITIALIZATION

iprd - period in seconds of writing sound to the host. Automatic minumum of 64 samples.

PERFORMANCE

hostin is designed to catch an audio stream sent by the host processor at any time, and to incorporate it into a running Csound environment. There can be only one **hostin** opcode in an orchestra, and the instrument containing it will be turned on and off as needed by special commands from the host. The host may send up to eight streams simultaneously, each of which will induce a separate instrument copy and activation pattern.

hostin always produces stereo output at the orchestra sampling rate. The host may send stereo, or mono with a varying pan factor. It can also transmit in any sample format, and at any sample rate, with each concurrent stream potentially different. **hostin** will automatically convert these into audio streams suitable for Csound processing. The opcode is normally encased in a simple instrument armed with appropriate effects-bus sends.

hostout will write the audio stream *asig* to the host.every *iprd* seconds.

Example:

```
a1,a2           hostin                      ; catch any host audio
gach1, garv1    mfxsend     a1, .2, .5      ; send both channels
gach2, garv2    mfxsend     a2, .2, .5      ;    to chorus and reverb
                outs        a1, a2          ;    plus direct out
```

ar	**fxsend***	asig, klevel
a1[,a2,a3,a4]	**mfxsend***	asig, ilvl1[, ilvl2, ilvl3, ilvl4]
	outs12*	asig
	panouts*	asig, kprop

Add a signal to one or more effects busses, or send it to the audio output channels.

INITIALIZATION

ilvl1,ilvl2,ilvl3,ilvl4 - weights for the respective effects-send operations. The number of weights is variable up to 4, but it must agree with the number of output paths.

PERFORMANCE

These units offer an efficient means of sending a signal to special paths or to the output channels. When an instrument needs to transmit data for effects processing it is common to add it into global variables which act as effects send busses, the contents of which are first processed then cleared to zero by the intended receiver. **fxsend** and **mfxsend** provide an efficient way of sending weighted data on up to four effects busses simultaneously.

outs12 and **panouts** provide an efficient means of sending monaural data to stereo output channels. While the first sends the full signal equally to both channels, the second sends just the proportion *kprop* to channel 1 and (1 - *kprop*) to channel 2.

Example:

```
garvb,gacho  mfxsend  asig, v1, v2   ; send to reverb and chorus
             panouts  asig, v3        ; and pan the source as reqd
```

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 (enscribed) 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
```

k1	**kread**	ifilname, iformat, iprd[, interp]
k1, k2	**kread2**	ifilname, iformat, iprd[, interp]
k1, k2, k3	**kread3**	ifilname, iformat, iprd[, interp]
k1, k2, k3, k4	**kread4**	ifilname, iformat, iprd[, interp]
	kdump	ksig1, ifilname, iformat, iprd
	kdump2	ksig1, ksig2, ifilname, iformat, iprd
	kdump3	ksig1, ksig2, ksig3, ifilname, iformat, iprd
	kdump4	ksig1, ksig2, ksig3, ksig4, ifilname, iformat, iprd

Periodically read/write orchestra control-signal values to a named external file in a specific format.

INITIALIZATION

ifilname - character string (in double quotes, spaces permitted) denoting the external file name. May be either a full path name with target directory specified, or a simple filename to be found or created within the current directory.

iformat - specifies the input and output data format: 1 = 8-bit signed character (high-order 8 bits of a 16-bit integer), 4 = 16-bit short integers, 5 = 32-bit long integers, 6 = 32-bit floats, 7 = ASCII long integers, 8 = ASCII floats (2 decimal places). Note that A-law and U-law output are not available, and that all formats except the last two are binary. Note also that these input and output files contain no header information, so the format must be explicit.

iprd - the period of *ksig* output in seconds, rounded to the nearest orchestra control period. A value of 0 implies one control period (the enforced minimum), which will read/write frames to the external file at the orchestra control rate.

interp (optional) - if non-zero, and *iprd* implies more than one control period, interpolate the k-signals between the periodic reads from the external file. The default value is 0 (repeat each signal between frames).

PERFORMANCE

These units allow up to four generated control signal values to be read or saved in a named external file. The file contains no self-defining header information, but is a regularly sampled time series, suitable for later input or analysis. There may be any number of **kread** units in an instrument or orchestra, and they may read from the same or different files. There may be any number of **kdump** units in an instrument or orchestra, but each must write to a different file.

Example:

```
knum    =       knum + 1                                        ; at each k-period
ktemp   tempest krms, .02, .1, 3, 2, 800, .005, 0, 60, 4, .1 .995       ; estimate the tempo
koct,ka specptrk wsig, kvar, 7, 10, 8.5, 20, 6, .8              ;     pitch & amp
        kdump4  knum, ktemp, cpsoct(koct), ka, "what_happened", 8, 0    ; & save them
```

SIGNAL DISPLAY

print	iarg[, iarg,...]
display	xsig, iprd[, inprds][, iwtflg]
dispfft	xsig, 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.

inprds (optional) - number of display periods retained in each display graph. A value of 2 or more will provide a larger perspective of the signal motion. The default value is 1 (each graph completely new).

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, with a minimum of 16 and a maximum of 4096. 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 an audio or control signal (*asig* or *ksig*) every *iprd* seconds using the Fast Fourier Transform method.

Example:

```
k1      envlpx      l, .03, p3, .05, l, .5, .0l     ; generate a note envelope
        display     k1, p3                          ; and display entire shape
```

COSTING

clkon* id
clkoff* id

Start and stop an instrument segment timer.

INITIALIZATION

id - unique identifier (from 1 to 10) that associates each on-off pair.

PERFORMANCE

These units can be placed inside instruments to measure the amount of computation used by the signal processing opcodes. They are placed before and after the opcode(s) to be measured; at the conclusion of the performance Csound will list how many million instructions-per-second (mips) were devoted to the units within each clock pair.

The mips count gives a good estimate of the cost of an instrument segment. The estimate is statistical, based on the computation cycles consumed while that part of the instrument was active, and the rating is independent of how many copies of the instrument were allocated, or how often the instances were in fact active. The mips cost is independent of the speed of the processor, but does give a sense of how those cycles are being spent. At a sampling rate of 32 KHz, simple opcodes like **oscil** and **reson** will be seen to cost about .25 mips, while more complex ones like **loscil** will cost about .4. The heavy users like **harmon**, **chorus** and **reverb** cost anything from .8 to 1.5 mips, while **specptrk** and **harmon2** can cost 3 to 5 mips apiece.

Given a processor rated at 40 mips (e.g. the 21060), a simple instrument that costs .75 mips can play about 50 voices simultaneously. (We leave a few mips for Csound overhead.) The most expensive effects-processing units are best run in a single instrument, getting their data from global effects busses.

clkon and **clkoff** units travel in pairs, with a unique *id* denoting the pairing. This means that the pairs can be nested (one pair inside another) or overlapped (two ons then two offs), provided only that the pairing is clear. Both parts of a pair must reside inside a single instrument block.

Example:

```
        instr   1, 2
        clkon   1               ; measure the loscil cost
a1      loscil  1000, 440, 1, 256
        clkoff  1
        clkon   5               ; and then the filter cost
a2      filter  a1, 10000, .05, 2
        clkoff  5
        out     a2
        endin
```

2. 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 another score-generating program, 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 nonempty 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. It is legal only in p2.

E.g.: the statements

```
i1    0    .5    100
i .   +
i
```

will result in

```
i1    0    .5    100
i1    .5   .5    100
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 explicitly by the **scsort** command, or implicitly 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. Userwritten 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 operations, 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.

E.g.: the statements

```
i1    0    1    10    np4    pp5
i1    1    1    20
i1    1    1    30
```

will result in

```
i1    0    1    10    20    0
i1    1    1    20    30    20
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.

E.g.: the statements

```
i1    0    1    100
i1    1    1    <
i1    2    1    <
i1    3    1    400
i1    4    1    <
i1    5    1    0
```

will result in

```
i1    0    1    100
i1    1    1    200
i1    2    1    300
i1    3    1    400
i1    4    1    200
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 200) 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 200; 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 can 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 (from 1 to 200), 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 score section or a **-t** flag in the command line.

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, or on receipt of a MIDI noteoff signal. An instrument can modify its own duration either by changing its p3 value during note initialization, or by prolonging itself through the action of a **linenr** unit.

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 | These 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 amplitude 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 accelerando 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).

N.B. If the **csound** command includes a **-t** flag, the interpreted tempo of all score **t** statements will be overridden by the command-line tempo.

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 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.

3. 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

This subroutine transfers data from a soundfile into a function table.

f # time size 1 filcod skiptime format channel

size - number of points in the table. Ordinarily a power of 2 or a power-of-2 plus 1 (see **f** statement); the maximum tablesize is 16777216 (2**24) points. If the source soundfile is of type AIFF, allocation of table memory can be *deferred* by setting this parameter to 0; the size allocated is then the number of points in the file (probably not a power-of-2), and the table is not usable by normal oscillators, but it is usable by a **loscil** unit. An AIFF source can also be mono or stereo.

filcod - integer or character-string denoting source soundfile name. A positive integer denotes an indexed filename in the orchestra (see **strset**); a negative integer denotes the file **soundin**.*filcod*; a character-string (double quotes, spaces permitted) gives the filename itself. A filename is optionally a full pathname. If not full path, the file is sought first in the current directory, then in that given by the environment variable SSDIR (if defined) then by SFDIR. See also **soundin**.

skiptime - begin reading at *skiptime* seconds into the file.

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.

channel - read only a specific channel (1,2,3 or 4) from the file. A zero means read all channels.

Note:

Reading stops at end-of-file or when the table is full. Table locations not filled will contain zeros. 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.

Examples:

```
f  1  0  8192  1  23  0  4  0
f  2  0  0  -1  "trumpet  A#5"  0  4  0
```

The tables are filled by reading all channels of 2 files, “soundin.23” and “trumpet A#5”, expected in the current directory, SSDIR or SFDIR. The first table is pre-allocated; the second is allocated dynamically, and its rescaling is inhibited.

GEN02

This subroutine transfers data from immediate pfields into a function table.

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.

v1, v2, v3, ... - values to 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 l)th stored value respectively in the generated function table.

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

$$c_0 + c_1x + c_2x^2 + . \ . \ . \ + c_nx^n$$

Coefficients may be positive or negative real numbers; a zero denotes a missing term in the polynomial. The coefficient list begins in p7, providing a current upper limit of 144 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 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 nonzero 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 + 1th 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 off 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, GEN19

These subroutines generate composite waveforms made up of weighted sums of simple sinusoids. The specification of each contributing partial requires 3 pfields using **GEN09,** 1 using **GEN10,** and 4 using **GEN19.**

```
f  #  time  size  9    pna   stra  phsa  pnb   strb  phsb . . .
f  #  time  size  10   str1  str2  str3  str4 . . . .
f  #  time  size  19   pna   stra  phsa  dcoa  pnb   strb  phsb  dcob . . .
```

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 may be rescaled later. Negative values are permitted and imply a 180 degree phase shift.

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

dcoa, dcob, etc. - DC offset of partials *pna, pnb*, etc. This is applied *after* strength scaling, i.e. a value of 2 will lift a 2-strength sinusoid from range [-2,2] to range [0,4] (before later rescaling).

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:

These 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** or **GEN19**.

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

Examples:

```
f  1  0  1024   9   1  3    0   3  1  0  9  .3333  180
f  2  0  1024  19  .5  1  270  1
```

f 1 combines partials l, 3 and 9 in the relative strengths in which they are found in a square wave, except that partial 9 is upside down. f 2 creates a rising sigmoid [0 - 2]. Both will be rescaled.

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 consecutive 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 +int*] 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 **GEN03** 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.

GEN17

This subroutine creates a step function from given x-y pairs.

f # time size 17 x1 a x2 b x3 c . . .

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.

x1, x2, x3, etc. - x-ordinate values, in ascending order, 0 first.

a, b, c, etc. - y-values at those x-ordinates, held until the next x-ordinate.

This subroutine creates a step function of x-y pairs whose y-values are held to the right. The right-most y-value is then held to the end of the table. The function is useful for mapping one set of data values onto another, such as MIDI note numbers onto sampled sound ftable numbers (see **loscil**).

Example:

```
f 1 0 128 -17 0 1 12 2 24 3 36 4 48 5 60 6 72 7 84 8
```

This describes a step function with 8 successively increasing levels, each 12 locations wide except the last which extends its value to the end of the table. Rescaling is inhibited. Indexing into this table with a MIDI note-number would retrieve a different value every octave up to the eighth, above which the value returned would remain the same.

GEN19

See GEN09, GEN10.

GEN20

This subroutine generates functions of different windows, suitable for spectrum analysis or granular synthesis.

f # time size 20 window max opt

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.

window - window type requested, with the following meanings:

1	Hamming
2	Hanning
3	Bartlett (triangle)
4	Blackman (3 term)
5	Blackman-Harris (4 term)
6	Gaussian
7	Kaiser
8	Rectangle
9	Sinc

max - maximum unscaled value. This is the absolute value at the peak point of the window, provided the ftable is left unscaled (has a negative p4). If p4 is positive, or *max* is omitted, the peak will be rescaled to 1.

opt - optional argument (0 - 10), used only by the Kaiser window to specify how 'open' the window is: 0 produces a rectangular window, while a 10 results in a Hamming-like window.

GEN21

This generates tables of different random distributions, corresponding to noise generators.

f # time size 21 distr range opt1 opt2

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.

distr, range, opt1, opt2 - distribution, range and optional arguments, with the following meanings:

distr	range	opt1	opt2
1 = Uniform			
2 = Linear	range	1=pos, 2=bipolar	
3 = Exponential	range	1=pos, 2=bipolar	
4 = Cauchy	range	1=pos, 2=bipolar	
5 = Poisson	lambda		
6 = Gaussian	range		
7 = Weibull	sigma	tau	
8 = Beta	range	alpha	beta

4. 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 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 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 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

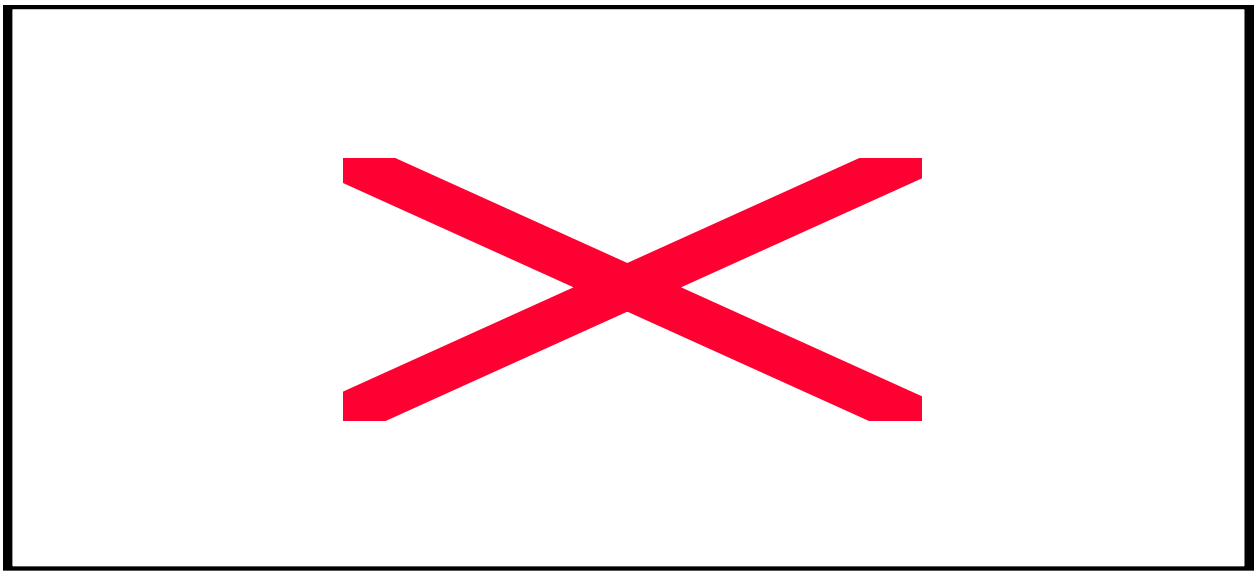
Accidentals are carried throughout the measure just as in traditional music notation. However, an accidental specified within a measure will hold for that note in all registers, in contrast with traditional notation. Therefore, make sure to specify *n* when you no longer want an accidental applied to 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 Scot encoding of this score might appear as follows:

```
; 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 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 operate 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 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 notation *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

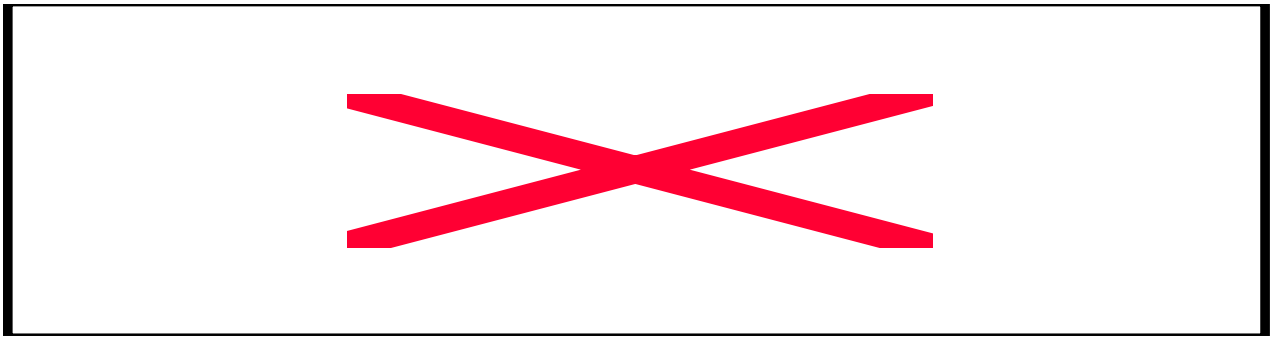
changed merely by changing a macro value, rather than changing all occurrences of that particular value in the score.

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



Scot encoding:

```
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 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 in Additional Features. The output score 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.

Ramping

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
```

5. Running from MIDI Data

A **Csound** orchestra can be driven by MIDI data instead of an ascii score. It can even be driven by both at the same time, provided the Orchestra contains both kinds of instruments. Instruments suitable for MIDI activation are similar to score-driven instruments, and the conversion from one to the other is not difficult.

There are three possible sources of MIDI data. Two of them send data in real-time (to be played immediately) and the other is a file in which time is encoded. These are:

1. Realtime MIDI commands passed directly from a **host program**. This might be a MIDI sequencer like Cakewalk, a computer game sending MIDI data, or a user program on the host that generates MIDI data (see **ontimer**). Set up using the command line flag **–RM**.
2. Realtime MIDI commands coming from some **external device**. This might be a MIDI keyboard or a MIDI controller, allowing live musicians to play Csound like a synthesizer. Set up using the command line flag **–K devname**.
3. A disc-resident **MIDI file**. This is usually created by a sequencer program, then stored as a binary file. The file contains all the information that a realtime source would normally send, but time is encoded as delta-steps between non-simultaneous events. In this format, the real time between events is a function of the current **tempo**, which can be varied. Use **–M filnam**.

Instruments for all three sources share the common task of converting 8-bit integer MIDI data into Csound's floating-point data format. Since MIDI encoding often forces musical data into an unnatural scaling (e.g. amplitude is transmitted as a 7-bit **key velocity**, with no set standard for what that means), the conversion must include scaling or re-mapping by the receiving instrument. The tasks include the following:

MIDI source data form		Computational form needed
key velocity	7-bit non-linear	amplitude for linear mixing
pitch	7-bit note number	cycles per second
pitch bend amount	14-bit controller value	resultant cps value
sound patch needed	7-bit program number	assigned instrument number
attack-decay shaping	presumed in the patch	want something more variable
stereo panning	7-bit parameter value	fraction to each audio channel

The translations are done using Csound's MIDI converters such as **notnum, veloc, cpsmidi**. Most of these opcodes offer an optional linear mapping range, onto which the raw 7-bit MIDI data will be mapped on arrival. Events requiring different sounds are sent on different MIDI channels, and each channel first relays a program number (denoting the desired patch), followed by the events for that patch and any control information specific to those events. It is up to the Csound instrument to trap this data and re-interpret it.

Conventional Csound operation

The relation between a MIDI channel, its patch program, and the Csound instrument that will handle the events is declared by the Orchestra header statement:

massign channel-number, instrument-number

This statement does two things. First, it sets up a MIDI channel of that number, ready to receive any channel-specific MIDI data such as controller values or event commands; second, it essentially *pre-assigns* a program (sound patch) to this channel, by saying that the instrument will produce the sound that will be needed. If more than one sound patch is desired on this

channel, the distinction can be relayed via a *controller* value, which the instrument can use at each note onset to redirect its synthesis. Also, for any given sound patch, the incoming **notnum** can be used for pitch-dependent synthesis, such as reading from pitch-dependent sample tables.

A MIDI instrument with this kind of two-way table sensing might look like:

```
if1     ftgen   1, 0, 0, 1, "lopiano.aif"    ; preload sound samples
if2     ftgen   2, 0, 0, 1, "hipiano.aif"
if3     ftgen   3, 0, 0, 1, "locello.aif"
if4     ftgen   4, 0, 0, 1, "hicello.aif"
        massign 1, 5                        ; send chnl 1 events to instr 5
        ctrlinit 1, 20, 0                   ; init chnl 1 controller 20 to 0

        instr   5
inote   notnum
icps    cpsmidi                             ; remap the midi data
iamp    veloc   0, 2000
ictrl   midictrl 20                         ; use ctrl 20 value
ifn     =       ( ictrl == 0 ? 1 : 3 )      ;   to choose a timbre
ihiflg  =       ( inote < 60 ? 0 : 1 )      ; lo or hi sample ?
kamp    linenr  iamp, 0, .1                 ; envlp with .1 sec release
asig    loscil  kamp, icps, ifn + hiflg     ; read the sound sample
        out     asig
        endin
```

Extended Csound operation

The MIDI resources are considerably broadened for **Extended Csound**. The relation between patch programs and instruments is made via **pgminit** and **vprogs**, and the program change (now sent as a MIDI command) will automatically select the right instrument and invoke an array of program-relevant data. This greatly simplifies and speeds up the run-time instrument body.

```
        ipn1    ftload  "lopiano.aif"               ; preload with auto
numbering
        ipn2    ftload  "hipiano.aif"
        icel1   ftload  "locello.aif"
        icel2   ftload  "hicello.aif"
        ipno    ftsplit 2, 2,  0, ipn1,  60, ipn2   ; set up the keyboard splits
        icel    ftsplit 2, 2,  0, icel1,  60, icel2
                pgminit 1, ipno                     ; and create two programs
                pgminit 43, icel                    ; containing them

                instr   5
                vprogs  1, 43               ; assoc these progs with instr 5
        ifn     mtsplit v1                  ; for each prog, get the split data
        iamp    veloc   0, 2000
        kamp    linenr  iamp, 0, .1
        asig    loscil  kamp, icps, ifn     ;  & read that sound sample
                out     asig
                endin
```

Moreover, if the soundfiles and programs referenced above are known to **Extended Csound** as members of its **General Midi Set**, *the top 8 lines can be replaced* by a single statement:

```
        autopgms
```

Also, with this statement in the Orchestra header, an instrument block is now able to access any of the other programs in the General Midi Set, so this instrument could now be expanded to:

```
          autopgms                        ; access the GM sample set

          instr     5
          vprogs    1, 5, 27, 43, 69      ; assoc ALL these progs with instr 5
ifn       mtsplit   v1                    ; for each prog, get the split data
iamp      veloc     0, 2000
kamp      linenr    iamp, 0, .1
asig      loscil    kamp, icps, ifn       ;  & read that sound sample
          out       asig
          endin
```

Now whenever any one of the above programs is first invoked (usually with a program change) the soundfiles for that timber are immediately loaded, along with additional information that can be accessed by the v-array (v1, v2, …). The General Midi Set is varied in structure. Some members are multi-sampled as above, some are single sounds that do not have split points; still others belong to a **drum set**, which has an independent program numbering scheme. These all require different kinds of instrument profiles, and the reader is referred to the *Example* Orchestras visible through the **Csound Launcher** (see **Appendix 6, Extended Csound on your PC**).

Loading samples from the hard drive, however, takes a non-trivial amount of time, and the requesting entity (host commands, live performer, or midifile) must be willing to wait that long before sending the first **note-on** command. If not, there will be a **delay** while the loading task is completed, and the accumulated first few notes will all appear to be skipped (they will actually happen in zero time). There are two alternatives to this: preloading and Midifile preprocessing.

Preloading by request list

If the set of expected program changes is known, these can be identified to **Extended Csound** while the executable is being launched, and the required sounds will be loaded as part of Orchestra initialization. The expected program numbers are listed in a file 'preprogs.nn', where the suffix *nn* is specified in the **command-line preload** flag **–P nn**.

The preprogs file is ascii, with a simple format (in **pgminit** notation) and a concluding 'e':

pgm1.bn pgm2.bn pgm3.bn ….. dpgm1.ch (drumkey1 drumkey2 …) … e

Example: 1 5 27.08 43 d1.10 (42 45) d9.26 (47 49) e

Preloading will happen before the host MIDI or external MIDI ports are attached, and before any Midifile is opened. The console message 'Orchestra now loaded and running on DSP' is the signal that play can begin. Program changes subsequently requested will happen without the delay of loading more samples.

Preloading by Midifile preprocessing

If the performance is from a **Midifile** to which we have *prior access*, we can pre-process the file and automatically insert the same information as above, tailored to just those programs actually needed. If the Midifile expects to send its MIDI data through **multiple ports** (i.e., it uses more than 16 MIDI channels), we can induce Csound to handle this with *virtual ports*.

midicvt is a standalone program for converting standard midi files from Format 1 to Format 0. Although Format 1 (where tracks are laid end to end) is the preferred format for authoring, this format contains duplicate delta timing data in each of its tracks and depends on a track merging

program to put the events in chronological order during performance. A Format 0 file consists of a single track, with all the Format 1 data now merged into performance order. This is the Format required by Csound.

Format conversion can be very fast, although in requiring multiple pointers into a Format 1 source it is either space hungry or I/O intensive. Also, modern sequencers often demand more than one port, to overcome the 16-channel limit of the midi spec. Csound has a way of simulating multiple ports, and this is applied during format conversion. The Csound utility to do this is invoked by

midicvt filnamein filnameout -pB{trka, trkb, ...} -pC{trkp, trkq, ... } ....

where -pB, -pC ... refer to ports B, C, .., and trka,trkb .. are the associated track numbers.

Port data is not required for a single port (portA) file. However, since port information is not part of the midi spec, nor readily evident in the file, the association of tracks with other ports must be made specific. On conversion to Format 0, the port information is imbedded in the Csound midi file by the use of *virtual port coding*. Csound can respond to up to 6 **virtual ports** (or 96 MIDI channels) within a single Format 0 file. This means that during performance Csound does not open multiple input files to handle multiple ports in the source. Moreover, having virtual port data within a single merged Format 0 file means that the delta-time information is not duplicated. Obviously, this is a custom Format 0 file, playable by other systems only with certain errors.

midicvt also sends to Csound other information critical to smooth and simplified performance. When a *program change* arises during performance, it assumes that the data for that program (sampled sounds and other control information) is already present. It can be, provided the user has developed an orchestra that preloads all the files, keyboard split points, etc. ahead of time. But as described under **autopgms** in the orchestra section, this means the orchestra is tailored to a single midi file, or at least to some generic set of files. A major purpose of the unit **vprogs** is to allow automatic invocation of the correct sampled data and control information at each program change.

Consequently, during format conversion **midicvt** makes an intial *look-ahead pass* over the midifile to extract information that a Csound orchestra should be aware of before starting its performance. The information is encoded in a proprietary **Csound sysex**, placed near the head of the midi file prior to channel controller settings. It includes information such as which drumset keys will be used by which channel, and what program changes can be expected. If **autopgms** is present, the Orchestra will preload all the data needed, and set up an internal pointer system that permits smooth program changes and smooth context switching between relevant control data.

This customized Format 0 file enables a Csound Orchestra to focus on continuous realtime audio processing with minimum interruption. Its only additional concern then is the handling of unscheduled event data, such as might stem from concurrent vocal or keyboard input. This is an ideal performance situation to be in.

Appendix 1: The Soundfile Utility Programs

The Csound Utilities are **soundfile preprocessing** programs that return information on a soundfile or create some analyzed version of it for use by certain Csound generators. Though different in goals, they share a common soundfile access mechanism and are describable as a set. The Soundfile Utility programs can be invoked in two equivalent forms:

csound -U utilname [flags] filenames . . .
utilname [flags] filenames . . .

In the first, the utility is invoked as part of the Csound executable, while in the second it is called as a standalone program. The second is smaller by about 200K, but the two forms are identical in function. The first is convenient in not requiring the maintenance and use of several independent programs—one program does all. When using this form, a **-U** flag detected in the command line will cause all subsequent flags and names to be interpreted as per the named utility; i.e. Csound generation will not occur, and the program will terminate at the end of utility processing.

Directories. Filenames are of two kinds, source soundfiles and resultant analysis files. Each has a hierarchical naming convention, influenced by the directory from which the Utility is invoked. Source soundfiles with a full pathname (begins with dot (.), slash (/), or for ThinkC includes a colon (:)), will be sought only in the directory named. Soundfiles without a path will be sought first in the current directory, then in the directory named by the SSDIR environment variable (if defined), then in the directory named by SFDIR. An unsuccessful search will return a "cannot open" error.

Resultant analysis files are written into the current directory, or to the named directory if a path is included. It is tidy to keep analysis files separate from sound files, usually in a separate directory known to the SADIR variable. Analysis is conveniently run from within the SADIR directory. When an analysis file is later invoked by a Csound generator (adsyn, lpread, pvoc) it is sought first in the current directory, then in the directory defined by SADIR.

Soundfile Formats. Csound can read and write audio files in a variety of formats. Write formats are described by Csound command flags. On reading, the format is automatically determined from the soundfile header, and the data automatically converted to floating-point during internal processing. When Csound is installed on a host with local soundfile conventions (SUN, NeXT, Macintosh) it may conditionally include local packaging code which creates soundfiles not portable to other hosts. However, Csound on any host can *always* generate and read **AIFF** and **WAV** files, which are thus portable formats. Sampled sound libraries are typically AIFF, and the variable SSDIR usually points to a directory of such sounds. If defined, the SSDIR directory is in the search path during soundfile access. Note that some AIFF sampled sounds have an audio looping feature for sustained performance; the analysis programs will traverse any loop segment once only.

For soundfiles without headers, an SR value may be supplied by a command flag (or its default). If both header and flag are present, the flag value will over-ride.

When sound is accessed by the audio Analysis programs (below), only a single channel is read. For stereo or quad files, the default is channel one; alternate channels may be obtained on request.

SNDINFO - get basic information about one or more soundfiles.

csound -U sndinfo soundfilenames . . .
or **sndinfo** soundfilenames . . .

sndinfo will attempt to find each named file, open it for reading, read in the soundfile header, then print a report on the basic information it finds. The order of search across soundfile directories is as described above. If the file is of type AIFF or WAV, some further details are listed first.

Example:

csound -U sndinfo test Bosendorfer/"BOSEN mf A0 st" foo foo2

where the environment variables SFDIR = /u/bv/sound, and SSDIR = /so/bv/Samples, might produce the following:

```
util  SNDINFO:

/u/bv/sound/test:
        srate 22050, monaural, 16 bit shorts, 1.10 seconds
        headersiz 1024, datasiz 48500  (24250 sample frames)

/so/bv/Samples/Bosendorfer/BOSEN mf A0 st:  AIFF, 197586 stereo samples, base Frq
261.6 (midi 60), gain 0 db, sustnLp: mode 1, 121642 to 197454, relesLp: mode 0
        AIFF soundfile, looping with modes 1, 0
        srate 44100, stereo, 16 bit shorts, 4.48 seconds
        headersiz  402, datasiz 790344  (197586 sample frames)

/u/bv/sound/foo:
        no recognizable soundfile header

/u/bv/sound/foo2:
        couldn't find
```

HETRO - hetrodyne filter analysis for the Csound **adsyn** generator.

csound -U hetro [flags] infilename outfilename
or **hetro** [flags] infilename outfilename

hetro takes an input soundfile, decomposes it into component sinusoids, and outputs a description of the components in the form of breakpoint amplitude and frequency tracks. Analysis is conditioned by the control flags below. A space is optional between flag and value.

-s<srate> sampling rate of the audio input file. This will over-ride the srate of the soundfile header, which otherwise applies. If neither is present, the default is 10000. Note that for **adsyn** synthesis the srate of the source file and the generating orchestra need not be the same.

-c<channel> channel number sought. The default is 1.

-b<begin> beginning time (in seconds) of the audio segment to be analyzed. The default is 0.0

-d<duration> duration (in seconds) of the audio segment to be analyzed. The default of 0.0 means to the end of the file. Maximum length is 32.766 seconds.

-f<begfreq> estimated starting frequency of the fundamental, necessary to initialize the filter analysis. The default is 100 (cps).

-h<partials> number of harmonic partials sought in the audio file. Default is 10, maximum 50.

-**M**<maxamp> maximum amplitude summed across all concurrent tracks. The default is 32767.

-m<minamp> amplitude threshold below which a single pair of amplitude/frequency tracks is considered dormant and will not contribute to output summation. Typical values: 128 (48 db down from full scale), 64 (54 db down), 32 (60 db down), 0 (no thresholding). The default threshold is 64 (54 db down).

-n<brkpts> initial number of analysis breakpoints in each amplitude and frequency track, prior to thresholding (**-m**) and linear breakpoint consolidation. The initial points are spread evenly over the duration. The default is 256.

-l<cutfreq> substitute a 3rd order Butterworth low-pass filter with cutoff frequency *cutfreq* (in cps), in place of the default averaging comb filter. The default is 0 (don't use).

Example:

hetro -s44100 -b.5 -d2.5 -h16 -M24000 audiofile.test adsynfile7

This will analyze 2.5 seconds of channel 1 of a file "audiofile.test", recorded at 44.1 KHz, beginning .5 seconds from the start, and place the result in a file "adsynfile7". We request just the first 16 harmonics of the sound, with 256 initial breakpoint values per amplitude or frequency track, and a peak summation amplitude of 24000. The fundamental is estimated to begin at 100 Hz. Amplitude thresholding is at 54 db down. The Butterworth LPF is not enabled.

File format

The output file contains time-sequenced amplitude and frequency values for each partial of an additive complex audio source. The information is in the form of breakpoints (time, value, time, value, ...) using 16-bit integers in the range 0 - 32767. Time is given in milliseconds, and frequency in Hertz (cps). The breakpoint data is exclusively non-negative, and the values -1 and -2 uniquely signify the start of new amplitude and frequency tracks. A track is terminated by the value 32767. Before being written out, each track is data-reduced by amplitude thresholding and linear breakpoint consolidation.

A component partial is defined by two breakpoint sets: an amplitude set, and a frequency set. Within a composite file these sets may appear in any order (amplitude, frequency, amplitude ....; or amplitude, amplitude..., then frequency, frequency,...). During **adsyn** resynthesis the sets are automatically paired (amplitude, frequency) from the order in which they were found. There should be an equal number of each.

A legal **adsyn** control file could have following format:

```
-1  time1  value1  ...   timeK  valueK   32767        ; amplitude breakpoints for partial 1
-2  time1  value1  ...   timeL  valueL   32767        ; frequency breakpoints for partial 1
-1  time1  value1  ...   timeM  valueM  32767        ; amplitude breakpoints for partial 2
-2  time1  value1  ...   timeN   valueN  32767        ; frequency breakpoints for partial 2
-2  time1  value1  .....
-2  time1  value1  .....                              ; pairable tracks for partials 3 and 4
-1  time1  value1  .....
-1  time2  value1  .....
```

LPANAL - linear predictive analysis for the Csound **lp** generators

csound -U lpanal [flags] infilename outfilename
or **lpanal** [flags] infilename outfilename

lpanal performs both lpc and pitch-tracking analysis on a soundfile to produce a time-ordered sequence of *frames* of control information suitable for Csound resynthesis. Analysis is conditioned by the control flags below. A space is optional between the flag and its value.

-s<srate> sampling rate of the audio input file. This will over-ride the srate of the soundfile header, which otherwise applies. If neither is present, the default is 10000**.**

-c<channel> channel number sought. The default is 1.

-b<begin> beginning time (in seconds) of the audio segment to be analyzed. The default is 0.0

-d<duration> duration (in seconds) of the audio segment to be analyzed. The default of 0.0 means to the end of the file.

-p<npoles> number of poles for analysis. The default is 34, the maximum 50.

-h<hopsize> hopsize (in samples) between frames of analysis. This determines the number of frames per second (srate/hopsize) in the output control file. The analysis framesize = hopsize*2 samples. If hopsize is small & npoles large (e.g.hopsize < 5*npoles) the filters produced can be unstable. The default is 200, the maximum 500.

-C<string> text for the comments field of the lpfile header. The default is the null string.

-P<mincps> lowest frequency (in cps) of pitch tracking. -P0 means no pitch tracking.

-Q<maxcps> highest frequency (in cps) of pitch tracking. The narrower the pitch range, the more accurate the pitch estimate. The defaults are -P70, -Q200.

-v<verbosity> level of terminal information during analysis. 0 = none, 1 = verbose, 2 = debug. The default is 0.

Example:

lpanal -p26 -d2.5 -P100 -Q400 audiofile.test lpfil26

will analyze the first 2.5 seconds of file "audiofile.test", producing srate/200 frames per second, each containing 26-pole filter coefficients and a pitch estimate between 100 and 400 Hertz. Output will be placed in "lpfil26" in the current directory.

File format

Output is a file comprised of an identifiable header plus a set of frames of floating point analysis data. Each frame contains four values of pitch and gain information, followed by *npoles* filter coefficients. The file is readable by Csound's **lpread**.

lpanal is an extensive modification of Paul Lanksy's lpc analysis programs.

PVANAL - Fourier analysis for the Csound **pvoc** generator

csound -U pvanal [flags] infilename outfilename
or **pvanal** [flags] infilename outfilename

pvanal converts a soundfile into a series of short-time Fourier transform (STFT) frames at regular timepoints (a frequency-domain representation). The output file can be used by **pvoc** to generate audio fragments based on the original sample, with timescales and pitches arbitrarily and dynamically modified. Analysis is conditioned by the flags below. A space is optional between the flag and its argument.

-s<srate> sampling rate of the audio input file. This will over-ride the srate of the soundfile header, which otherwise applies. If neither is present, the default is 10000**.**

-c<channel> channel number sought. The default is 1.

-b<begin> beginning time (in seconds) of the audio segment to be analyzed. The default is 0.0

-d<duration> duration (in seconds) of the audio segment to be analyzed. The default of 0.0 means to the end of the file.

-n<frmsiz> STFT frame size, the number of samples in each Fourier analysis frame. Must be a power of two, in the range 16 to 16384. For clean results, a frame must be larger than the longest pitch period of the sample. However, very long frames result in temporal “smearing” or reverberation. The bandwidth of each STFT bin is determined by sampling rate / frame size. The default framesize is the smallest power of two that corresponds to more than 20 milliseconds of the source (e.g. 256 points at 10 kHz sampling, giving a 25.6 ms frame).

-w<windfact> Window overlap factor. This controls the number of Fourier transform frames per second. Csound's **pvoc** will interpolate between frames, but too few frames will generate audible distortion; too many frames will result in a huge analysis file. A good compromise for *windfact* is 4, meaning that each input point occurs in 4 output windows, or conversely that the offset between successive STFT frames is framesize/4. The default value is 4. Do not use this flag with **-h**.

-h<hopsize> STFT frame offset. Converse of above, specifying the increment in samples between successive frames of analysis (see also **lpanal**). Do not use with **-w**.

Example:

pvanal asound pvfile

will analyze the soundfile "asound" using the default *frmsiz* and *windfact* to produce the file "pvfile" suitable for use with **pvoc**.

Files

The output file has a special **pvoc** header containing details of the source audio file, the analysis frame rate and overlap. Frames of analysis data are stored as *float,* with the magnitude and 'frequency' (in Hz) for the first N/2 + 1 Fourier bins of each frame in turn. 'Frequency' encodes the phase increment in such a way that for strong harmonics it gives a good indication of the true frequency. For low amplitude or rapidly moving harmonics it is less meaningful. The analysis prints total number of frames, and frames completed on every 20th.

Author: Dan Ellis, dpwe@media.mit.edu

Appendix 2: 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 *control* program, and can be invoked either as a standalone score preprocessor, or as part of the Csound run-time system:

cscore scorefilein > scorefileout
or **csound -C** [otherflags] orchname scorename

The available *function* programs augment the **C** language library functions; they can read either *standard* or *pre-sorted* score files, can massage and expand the data in various ways, then make it available for performance by a **Csound** orchestra.

The user-written *control* program is also in **C**, and is compiled and linked to the function programs (or the entire Csound) by the user. It is not essential to know the **C** language well to write this 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* or *time-warped score* (see any score.srt), stored internally in time-warped format. It is either created in-line, or read in from an existing score file (either format). Its main components are an *opcode* and an array of *pfield* values. It is stored somewhere in memory, organized by a structure that starts as follows:

```
typedef struct {
        CSHDR  h;           /* space-managing header              */
        long   op;          /* opcode—t, w, f, i, a, s or e  */
        long   pcnt;        /* number of pfields p1, p2, p3 ...   */
        long   strlen;      /* length of optional string argument */
        char   *strarg;     /* address of optional string argument */
        float  p2orig;      /* unwarped p2, p3                    */
        float  p3orig;
        float  offtim;      /* storage used during performance    */
        float  p[1];        /* array of pfields p0, p1, p2 ...    */
} EVENT;
```

Any function subprogram that creates, reads, or copies an event 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. Groups of event pointers are stored in an *event list*, which has its own structure:

```
typedef struct {
        CSHDR  h;
        int    nslots;      /* max events in this event list      */
        int    nevents;     /* number of events present           */
        EVENT *e[1];        /* array of event pointers e0, e1, e2.. */
} EVLIST;
```

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. The **Cscore** *function subprograms* enable scores to 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*.
the symbol *fp* is a score input stream file pointer (FILE *);

calling syntax	description
e = createv(n); int 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 string-defined event to score output
a = lcreat(n); int n;	create an empty event list with n slots
a = lappev(a,e);	append event e to list a
a = lappstrev(a,"...");	append a string-defined event to 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 all events from score input, up to next s or e
a = lgetnext(nbeats); float nbeats;	read next *nbeats* beats from score input
a = lgetuntil(beatno); float beatno;	read all events from score input up to beat *beatno*
a = lsepf(b);	separate the f statements from list b into list a
a = lseptwf(b);	separate the t,w & 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); float from, to;	extract notes of time-span, creating new events
lput(a);	write the events of list a to the score output file
lplay(a);	send events of list a to the Csound orchestra for immediate performance (or print events if no orchestra)
relev(e);	release the space of event e
lrel(a);	release the space of list a (but not the events)
lrelev(a);	release the events of list a, and the list space
fp = getcurfp();	get the currently active input scorefile pointer (initially finds the command-line input scorefile pointer)
fp = filopen("filename");	open another input scorefile (maximum of 5)
setcurfp(fp);	make fp the currently active scorefile pointer
filclose(fp);	close the scorefile relating to FILE *fp

Writing a Control program.

The general format for a control program is:

```
#include  "cscore.h"
cscore()
{
        /*  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  "cscore.h"
cscore()
{
        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  "cscore.h"
cscore()
{
        EVENT  *e;          /* a pointer to an event                */
        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, 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  "cscore.h"
cscore()
{
        EVENT *e,**f;               /* declarations. see pp.8-9 in the */
        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 = b->nevents;             /* and get the number present */
        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 "cscore.h"
cscore()
{
        EVENT *e,**f;
        EVLIST *a,*b;
        int n, dim;                 /* declare two integer variables   */
```

```
        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 = b->nevents;
        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  "cscore.h"

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

cscore()
{
        EVENT *e, **f;
        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 = a->nevents;
        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("; diagnostic:  cue = %d\n", 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
```

More advanced examples.

The following program demonstrates reading from two different input files. The idea is to switch between two 2-section scores, and write out the interleaved sections to a single output file.

```
#include "cscore.h"                             /*   CSCORE_SWITCH.C  */

cscore()                            /* callable from either csound or standalone cscore */
{
        EVLIST *a, *b;
        FILE   *fp1, *fp2;              /* declare two scorefile stream pointers */

        fp1 = getcurfp();               /*  this is the command-line score */
        fp2 = filopen("score2.srt");    /*  this is an additional score file */

        a = lget();                     /* read section from score 1 */
        lput(a);                        /* write it out as is */
        putstr("s");
        setcurfp(fp2);
        b = lget();                     /* read section from score 2 */
        lput(b);                                /* write it out as is */
        putstr("s");

        lrelev(a);                      /* optional to reclaim space */
        lrelev(b);

        setcurfp(fp1);
        a = lget();                     /* read next section from score 1 */
        lput(a);                        /* write it out */
        putstr("s");
        setcurfp(fp2);
        b = lget();                     /* read next sect from score 2 */
        lput(b);                        /       * write it out */
        putstr("e");
}
```

Finally, we show how to take a literal, uninterpreted score file and imbue it with some expressive timing changes. The theory of composer-related metric pulses has been investigated at length by Manfred Clynes, and the following is in the spirit of his work. The strategy here is to first create an array of new onset times for every possible sixteenth-note onset, then to index into it so as to adjust the start and duration of each note of the input score to the interpreted time-points. This also shows how a Csound orchestra can be invoked repeatedly from a run-time score generator.

```
#include "cscore.h"                                          /* CSCORE_PULSE.C */
                /* program to apply interpretive durational pulse to      */
                /* an existing score in 3/4 time, first beats on 0, 3, 6 ... */
static float four[4] = { 1.05, 0.97, 1.03, 0.95 };      /* pulse width for 4's*/
static float three[3] = { 1.03, 1.05, .92 };            /* pulse width for 3's*/

cscore()                            /* callable from either csound or standalone cscore  */
{
        EVLIST  *a, *b;
register EVENT  *e, **ep;
        float   pulse16[4*4*4*4*3*4];       /* 16th-note array, 3/4 time, 256 measures */
        float   acc16, acc1,inc1, acc3,inc3, acc12,inc12, acc48,inc48, acc192,inc192;
register float  *p = pulse16;
register int    n16, n1, n3, n12, n48, n192;

                                                    /* fill the array with interpreted ontimes  */
        for (acc192=0.,n192=0; n192<4; acc192+=192.*inc192,n192++)
          for (acc48=acc192,inc192=four[n192],n48=0; n48<4; acc48+=48.*inc48,n48++)
            for (acc12=acc48,inc48=inc192*four[n48],n12=0;n12<4;
                                                    acc12+=12.*inc12,n12++)
              for (acc3=acc12,inc12=inc48*four[n12],n3=0; n3<4; acc3+=3.*inc3,n3++)
                for (acc1=acc3,inc3=inc12*four[n3],n1=0; n1<3; acc1+=inc1,n1++)
                  for (acc16=acc1,inc1=inc3*three[n1],n16=0; n16<4;
                                                        acc16+=.25*inc1*four[n16],n16++)
                    *p++ = acc16;

/*      for (p = pulse16, n1 = 48; n1--; p += 4)            /*      show vals & diffs */
/*        printf("%g %g %g %g %g %g %g %g\n", *p, *(p+1), *(p+2), *(p+3),
/*              *(p+1)-*p, *(p+2)-*(p+1), *(p+3)-*(p+2), *(p+4)-*(p+3));  */

        a = lget();                     /* read sect from tempo-warped score */
        b = lseptwf(a);                 /* separate warp & fn statements      */
        lplay(b);                       /*   and send these to performance    */

        a = lappstrev(a, "s");          /* append a sect statement to note list          */
        lplay(a);                       /* play the note-list without interpretation */

        for (ep = &a->e[1], n1 = a->nevents; n1--; ) { /* now pulse-modifiy it */
            e = *ep++;
            if (e->op == 'i') {
                e->p[2] = pulse16[(int)(4. * e->p2orig)];
                e->p[3] = pulse16[(int)(4. * (e->p2orig + e->p3orig))] - e->p[2];
            }
        }
        lplay(a);                       /* now play modified list */
}
```

As stated above, the input files to Cscore may be in original or time-warped and pre-sorted form; this modality will be preserved (section by section) in reading, processing and writing scores. *Standalone* processing will most often use unwarped sources and create unwarped new files. When running from *within* Csound the input score will arrive already warped and sorted, and can thus be sent directly (normally section by section) to the orchestra.

A list of events can be conveyed to a Csound orchestra using **lplay**. There may be any number of lplay calls in a Cscore program. Each list so conveyed can be either time-warped or not, but each list *must* be in strict p2-chronological order (either from presorting or using **lsort**). If there is **no lplay** in a cscore module run from within Csound, all events written out (via putev, putstr or lput) constitute a **new score**, which will be sent initially to *scsort* then to the Csound orchestra for performance. These can be examined in the files 'cscore.out' and 'cscore.srt'.

A standalone cscore program will normally use the *put* commands to write into its output file. If a standalone cscore program contains lplay, the events thus intended for performance will instead be printed on the console.

A note list sent by **lplay** for performance should be temporally distinct from subsequent note lists. No note-end should extend past the next list's start time, since lplay will complete each list before starting the next (i.e. like a Section marker that doesn't reset local time to zero). This is important when using lgetnext() or lgetuntil() to fetch and process score segments prior to performance.

Compiling a Cscore program

A **Cscore** program can be invoked either as a Standalone program or as part of Csound:

```
        cscore  scorename > outfilename
or      csound -C [otherflags] orchname scorename
```

To create a **standalone** program, write a cscore.c program as shown above and test compile it with 'cc cscore.c'. If the compiler cannot find "cscore.h", try using -I/usr/local/include, or just copy the cscore.h module from the Csound source directory into your own. There will still be unresolved references, so you must now link your program with certain Csound I/O modules. If your Csound installation has created a libcscore.a, you can type

```
cc -o cscore  cscore.c  -lcscore
```

Else set an environment variable to a Csound directory containing the already compiled modules, and invoke them explicitly:

```
setenv  CSOUND  /ti/u/bv/csound
cc -o cscore  cscore.c  $CSOUND/cscoremain.o  $CSOUND/cscorefns.o \
                        $CSOUND/rdscor.o  $CSOUND/memalloc.o
```

The resulting executable can be applied to an input *scorefilein* by typing:

```
cscore  scorefilein  > scorefileout
```

To operate from **csound**, first proceed as above then link your program to a complete set of Csound modules. If your Csound installation has created a libcsound.a, you can do this by typing

```
cc -o mycsound  cscore.c  -lcsound  -lX11 -lm        (X11 if your installation included it)
```

Else copy *.c, *.h and Makefile from the Csound source directory, replace cscore.c by your own, then run 'make csound'. The resulting executable is your own special csound, usable as above. The -**C** flag will invoke your cscore program after the input score is sorted into 'score.srt'. With no lplay, the subsequent stages of processing can be seen in the files 'cscore.out' and 'cscore.srt'.

Appendix 3: An Instrument Design Tutorial

by
Richard Boulanger
Berklee College of Music

Csound instruments are created in an "orchestra" file, and the list of notes to play is written in a separate "score" file. Both are created using a standard word processor. When you run Csound on a specific orchestra and score, the score is sorted and ordered in time, the orchestra is translated and loaded, the wavetables are computed and filled, and then the score is performed. The score drives the orchestra by telling the specific instruments when and for how long to play, and what parameters to use during the course of each note event.

Unlike today's commercial hardware synthesizers, which have a limited set of oscillators, envelope generators, filters, and a fixed number of ways in which these can be interconnected, Csound's power is virtually unlimited. If you want an instrument with hundreds of oscillators, envelope generators, and filters, you just type them in. More important is the freedom to interconnect the modules, and to interrelate the parameters which control them. Like acoustic instruments, Csound instruments can exhibit a sensitivity to the musical context, and display a level of "musical intelligence" to which hardware synthesizers can only aspire.

Since the intent of this tutorial is to familiarize the novice with the syntax of the language, we will design several simple instruments. You will find many instruments of the sophistication described above in various Csound directories, and a study of these will reveal Csound's real power.

The Csound **orchestra** file has two main parts:

1. the "header" section - defining sample rate, control rate, and number of output channels.
2. the "instrument" section - in which the instruments are designed.

The Header Section: A Csound orchestra generates signals at two rates - an audio sample rate and a control sample rate. Each can represent signals with frequencies no higher than half that rate, but the distinction between audio signals and sub-audio control signals is useful since it allows slower moving signals to require less compute time. In the header below, we have specified a sample rate of 16kHz, a control rate of 1kHz, and then calculated the number of samples in each control period using the formula: ksmps = sr / kr.

```
sr     =     16000
kr     =     1000
ksmps  =     16
nchnls =     1
```

In Csound orchestras and scores, spacing is arbitrary. It is important to be consistent in laying out your files, and you can use spaces to help this. In the Tutorial Instruments shown below you will see we have adopted one convention. The reader can choose his or her own.

The Instrument Section: All instruments are numbered and are referenced thus in the score. Csound instruments are similar to patches on a hardware synthesizer. Each instrument consists of a set of "unit generators," or software "modules," which are "patched" together with "**i/o**" blocks — **i**, **k**, or **a** variables. Unlike a hardware module, a software module has a number of variable "arguments" which the user sets to determine its behavior. The four types of variables are:

setup only
i-rate variables, changed at the note rate

k-rate variables, changed at the control signal rate
a-rate variables, changed at the audio signal rate

Orchestra Statements: Each statement occupies a single line and has the same basic format:

result action arguments

To include an oscillator in our orchestra, you might specify it as follows:

a1 oscil 10000, 440, 1

The three "arguments" for this oscillator set its amplitude (10000), its frequency (440Hz), and its waveshape (1). The output is put in i/o block "a1." This output symbol is significant in prescribing the rate at which the oscillator should generate output—here the audio rate. We could have named the result anything (e.g. "asig") as long as it began with the letter "a".

Comments: To include text in the orchestra or score which will not be interpreted by the program, precede it with a semicolon. This allows you to fully comment your code. On each line, any text which follows a semicolon will be ignored by the orchestra and score translators.

Tutorial Instruments

Toot 1: Play One Note

For this and all instrument examples below, there exist **orchestra** and **score** files in the Csound subdirectory **tutorfiles** that the user can run to soundtest each feature introduced. The instrument code shown below is actually preceded by an *orchestra header section* similar to that above. If you run this on a high-speed computer, each example will likely run in realtime. During playback (realtime or otherwise) the audio rate may be modified to suit the local d-a converters.

The first orchestra file, called *toot1.orc,* contains a single instrument which uses an **oscil** unit to play a 440Hz sine wave (defined by f1 in the score) at an amplitude of 10000.

```
instr 1
  a1    oscil   10000, 440, 1
        out     a1
endin
```

Run this with its corresponding score file, *toot1.sco* :

```
f1    0       4096    10      1       ; use "gen1" to compute a sine wave
i1    0       4                       ; run "instr 1" from time 0 for 4 seconds
e                                     ; indicate the "end" of the score
```

Toot 2: “P-Fields”

The first instrument was not interesting because it could play only one note at one amplitude level. We can make things more interesting by allowing the pitch and amplitude to be defined by parameters in the score. Each column in the score constitutes a parameter field (p-field), numbered from the left. The first three parameter fields of the i-statement have a reserved function:

p1 = instrument number
p2 = start time
p3 = duration

All other parameter fields are determined by the way the sound designer defines his instrument. In the instrument below, the oscillator's amplitude argument is replaced by p4 and the frequency argument by p5. Now we can change these values at i-time, i.e. with each note in the score. The orchestra and score files now look like:

```
          instr   2
            a1    oscil   p4, p5, 1                  ; p4=amp
                  out     a1                          ; p5=freq
          endin

     f1   0       4096    10      1                   ; sine wave
; instrument  start   duration   amp(p4)   freq(p5)
     i2   0       1          2000       880
     i2   1.5     1          4000       440
     i2   3       1          8000       220
     i2   4.5     1         16000       110
     i2   6       1         32000       55
      e
```

Toot 3: Envelopes

Although in the second instrument we could control and vary the overall amplitude from note to note, it would be more musical if we could contour the loudness during the course of each note. To do this we'll need to employ an additional unit generator **linen**, which the Csound reference manual defines as follows:

```
     kr      linen   kamp, irise, idur, idec
     ar      linen   xamp, irise, idur, idec
```

linen is a signal modifier, capable of computing its output at either control or audio rates. Since we plan to use it to modify the amplitude envelope of the oscillator, we'll choose the latter version. Three of linen's arguments expect **i**-rate variables. The fourth expects in one instance a **k**-rate variable (or anything slower), and in the other an x-variable (meaning a-rate or anything slower). Our linen we will get its amp from p4.

The output of the **linen** (k1) is patched into the *kamp* argument of an **oscil**. This applies an envelope to the **oscil**. The orchestra and score files now appear as:

```
     instr 3
       k1    linen   p4, p6, p3, p7          ; p4=amp
       a1    oscil   k1, p5, 1               ; p5=freq
             out     a1                      ; p6=attack time
     endin                                   ; p7=release time

  f1  0  4096  10  1                  ; sine wave
; instr  start  duration  amp(p4)  freq(p5)  attack(p6)  release(p7)
  i3     0      1         10000     440        .05        .7
  i3     1.5    1         10000     440        .9         .1
  i3     3      1          5000     880        .02        .99
```

```
   i3     4.5     1          5000      880 .7        .01
   i3     6       2         20000      220           .5        .5
e
```

Toot 4: Chorusing

Next we'll animate the basic sound by mixing it with two slightly detuned copies of itself. We'll employ Csound's "cpspch" value converter which will allow us to specify the pitches by octave and pitch-class rather than by frequency, and we'll use the "ampdb" converter to specify loudness in dB rather than linearly.

Since we are adding the outputs of three oscillators, each with the same amplitude envelope, we'll scale the amplitude before we mix them. Both “iscale” and “inote” are arbitrary names to make the design a bit easier to read. Each is an **i**-rate variable, evaluated when the instrument is initialized.

```
instr 4                                 ; toot4.orc
  iamp = ampdb(p4)                      ; convert decibels to linear amp
  iscale = iamp * .333                  ; scale the amp at initialization
  inote = cpspch(p5)                    ; convert “octave.pitch” to cps
  k1    linen   iscale, p6, p3, p7      ; p4=amp
  a3    oscil   k1, inote*.996, 1       ; p5=freq
  a2    oscil   k1, inote*1.004, 1      ; p6=attack time
  a1    oscil   k1, inote, 1            ; p7=release time
  a1    =       a1 + a2 + a3
        out     a1
endin

  f1 0 4096 10 1                                ; sine wave
; instr   start  duration  amp(p4)  freq(p5)  attack(p6)  release(p7)
  i4      0      1         75       8.04       .1         .7
  i4      1      1         70       8.02       .07        .6
  i4      2      1         75       8.00       .05        .5
  i4      3      1         70       8.02       .05        .4
  i4      4      1         85       8.04       .1         .5
  i4      5      1         80       8.04       .05        .5
  i4      6      2         90       8.04       .03        1
e
```

Toot 5: Vibrato

To add some delayed vibrato to our chorusing instrument we use another oscillator for the vibrato and a line segment generator, **linseg**, as a means of controlling the delay. **linseg** is a **k**-rate or **a**-rate signal generator which traces a series of straight line segments between any number of specified points. The Csound manual describes it as:

```
kr      linseg      ia, idur1, ib[, idur2, ic[...]]
ar      linseg      ia, idur1, ib[, idur2, ic[...]]
```

Since we intend to use this to slowly scale the amount of signal coming from our vibrato oscillator, we'll choose the **k**-rate version. The **i**-rate variables: *ia, ib, ic*, etc., are the values for the points. The i-rate variables: *idur1, idur2, idur3,* etc., set the duration, in seconds, between segments.

```
instr 5                                          ; toot5.orc
  irel = .01                                     ; set vibrato release time
  idel1 = p3 * p10                               ; calculate initial delay (% of dur)
  isus  = p3 - (idel1+ irel)                     ; calculate remaining duration
  iamp = ampdb(p4)
  iscale = iamp * .333                           ; p4=amp
  inote = cpspch(p5)                             ; p5=freq
  k3    linseg  0, idel1, p9, isus, p9, irel, 0  ; p6=attack time
  k2    oscil   k3, p8, 1                        ; p7=release time
  k1    linen   iscale, p6, p3, p7               ; p8=vib rate
  a3    oscil   k1, inote*.995+k2, 1             ; p9=vib depth
  a2    oscil   k1, inote*1.005+k2, 1            ; p10=vib delay (0-1)
  a1    oscil   k1, inote+k2, 1
        out     a1+a2+a3
endin

  f 1  0  4096  10  1
; ins    strt     dur     amp     frq      atk     rel     vibrt   vibdpth  vibdel
  i5     0        3       86      10.00    .1      .7      7       6        .4
  i5     4        3       86      10.02    1       .2      6       6        .4
  i5     8        4       86      10.04    2       1       5       6        .4
e
```

Toot 6: Gens

The first character in a score statement is an **opcode**, determining an action request; the remaining data consists of numeric parameter fields (p-fields) to be used by that action. So far we have been dealing with two different opcodes in our score: **f** and **i**. **I-statements**, or note statements, invoke the p1 instrument at time p2 and turn it off after p3 seconds; all remaining p-fields are passed to the instrument.

F-statements, or lines with an opcode of **f**, invoke function-drawing subroutines called **GENS**. In Csound there are currently seventeen gen routines which fill wavetables in a variety of ways. For example, **GEN01** transfers data from a soundfile; **GEN07** allows you to construct functions from segments of straight lines; and **GEN10**, which we've been using in our scores so far, generates composite waveforms made up of a weighted sum of simple sinusoids. We have named the function "f1," invoked it at time 0, defined it to contain 512 points, and instructed GEN10 to fill that wavetable with a single sinusoid whose amplitude is 1. GEN10 can in fact be used to *approximate* a variety of other waveforms, as illustrated by the following:

```
f1  0  2048  10  1                                          ; Sine
f2  0  2048  10  1  .5  .3  .25  .2  .167  .14  .125  .111  ; Sawtooth
f3  0  2048  10  1  0   .3  0    .2  0     .14  0     .111  ; Square
f4  0  2048  10  1  1   1   1    .7  .5    .3   .1         ; Pulse
```

For the opcode **f**, the first four p-fields are interpreted as follows:

p1 - table number - In the orchestra, you reference this table by its number.
p2 - creation time - The time at which the function is generated.
p3 - table size - Number of points in table - must be a power of 2, or that plus 1.
p4 - generating subroutine - Which of the 17 GENS will you employ.
p5 -> p? - meaning determined by the particular GEN subroutine.

In the instrument and score below, we have added three additional functions to the score, and modified the orchestra so that the instrument can call them via p11.

```
instr 6                                          ; toot6.orc
  ifunc = p11                                    ; select basic waveform
  irel = .01                                     ; set vibrato release
  idel1 = p3 * p10                               ; calculate initial delay
  isus  = p3 - (idel + irel)                     ; calculate remaining dur
  iamp = ampdb(p4)
  iscale = iamp * .333                           ; p4=amp
  inote = cpspch(p5)                             ; p5=freq
  k3    linseg  0, idel1, p9, isus, p9, irel, 0  ; p6=attack time
  k2    oscil   k3, p8, 1                        ; p7=release time
  k1    linen   iscale, p6, p3, p7               ; p8=vib rate
  a3    oscil   k1, inote*.999+k2, ifunc         ; p9=vib depth
  a2    oscil   k1, inote*1.001+k2, ifunc        ; p10=vib delay (0-1)
  a1    oscil   k1, inote+k2, ifunc
        out     a1 + a2 + a3
endin

  f1 0 2048 10 1                                                    ; Sine
  f2 0 2048 10 1 .5 .3 .25 .2 .167 .14 .125 .111                    ; Sawtooth
  f3 0 2048 10 1 0  .3  0   .2  0  .14  0   .111                    ; Square
  f4 0 2048 10 1 1 1 1 .7 .5 .3 .1                                  ; Pulse
; ins  strt  dur  amp    frq     atk     rel     vibrt  vibdpth  vibdel  waveform(f)
  i6    0    2    86     8.00    .03     .7      6      9        .8      1
  i6    3    2    86     8.02    .03     .7      6      9        .8      2
  i6    6    2    86     8.04    .03     .7      6      9        .8      3
  i6    9    3    86     8.05    .03     .7      6      9        .8      4
e
```

Toot 7: Crossfade

Now we will add the ability to do a linear crossfade between any two of our four basic waveforms. We will employ our delayed vibrato scheme to regulate the speed of the crossfade.

```
instr 7                                          ; toot7.orc
  ifunc1 = p11                                   ; initial waveform
  ifunc2 = p12                                   ; crossfade waveform
  ifad1 = p3 * p13                               ; calculate initial fade
  ifad2 = p3 - ifad1                             ; calculate remaining dur
  irel = .01                                     ; set vibrato release
  idel1 = p3 * p10                               ; calculate initial delay
  isus  = p3 - (idel1+ irel)                     ; calculate remaining dur
  iamp = ampdb(p4)
  iscale = iamp * .166                           ; p4=amp
  inote = cpspch(p5)                             ; p5=freq
  k3    linseg  0, idel1, p9, isus, p9, irel, 0  ; p6=attack time
  k2    oscil   k3, p8, 1                        ; p7=release time
  k1    linen   iscale, p6, p3, p7               ; p8=vib rate
  a6    oscil   k1, inote*.998+k2, ifunc2        ; p9=vib depth
  a5    oscil   k1, inote*1.002+k2, ifunc2       ; p10=vib delay (0-1)
  a4    oscil   k1, inote+k2, ifunc2             ; p11=initial wave
```

```
  a3     oscil    k1, inote*.997+k2, ifunc1      ; p12=cross wave
  a2     oscil    k1, inote*1.003+k2, ifunc1     ; p13=fade time
  a1     oscil    k1, inote+k2, ifunc1
  kfade  linseg 1, ifad1, 0, ifad2, 1
  afunc1 = kfade * (a1+a2+a3)
  afunc2 = (1 - kfade) * (a4+a5+a6)
         out      afunc1 + afunc2
endin

f1 0 2048 10 1                                                    ; Sine
f2 0 2048 10 1 .5 .3 .25 .2 .167 .14 .125 .111                    ; Sawtooth
f3 0 2048 10 1 0  .3  0   .2  0  .14  0   .111                    ; Square
f4 0 2048 10 1 1 1 1 .7 .5 .3 .1                                  ; Pulse
; ins  strt  dur  amp  frq    atk rel vibrt vbdpt vibdel startwave  endwave  crosstime
  i7   0     5    96   8.07  .03  .1   5    6     .99     1          2        .1
  i7   6     5    96   8.09  .03  .1   5    6     .99     1          3        .1
  i7  12     8    96   8.07  .03  .1   5    6     .99     1          4        .1
```

Toot 8: Soundin

Now instead of continuing to enhance the same instrument, let us design a totally different one. We'll read a soundfile into the orchestra, apply an amplitude envelope to it, and add some reverb. To do this we will employ Csound's **soundin** and **reverb** generators. The first is described as:

```
a1      soundin      ifilcod[, iskiptime][, iformat]
```

soundin derives its signal from a pre-existing file. *ifilcod* is either the filename in double quotes, or an integer suffix (.n) to the name "soundin". Thus the file "soundin.5" could be referenced either by the quoted name or by the integer 5. To read from 500ms into this file we might say:

```
a1      soundin      "soundin.5",  .5
```

The Csound **reverb** generator is actually composed of four parallel **comb** filters plus two **allpass** filters in series. Although we could design a variant of our own using these same primitives, the preset reverb is convenient, and simulates a natural room response via internal parameter values. Only two arguments are required—the input (*asig*) and the reverb time (*krvt*).

```
ar      reverb       asig, krvt
```

The soundfile instrument with artificial envelope and a reverb (included directly) is as follows:

```
instr 8                                                    ; toot8.orc
  idur        =          p3
  iamp        =          p4
  iskiptime   =          p5
  iattack     =          p6
  irelease    =          p7
  irvbtime    =          p8
  irvbgain    =          p9
  kamp        linen      iamp, iattack, idur, irelease
  asig        soundin    "soundin.aiff", iskiptime
  arampsig    =          kamp * asig
  aeffect     reverb     asig, irvbtime
  arvbreturn  =          aeffect * irvbgain
              out        arampsig + arvbreturn
endin
```

```
; ins  strt      dur     amp   skip     atk     rel    rvbtime  rvbgain
  i8    0       1       .3      0      .03     .1       1.5      .2
  i8    2       1       .3      0      .1      .1       1.3      .2
  i8    3.5     2.25    .3      0      .5      .1       2.1      .2
  i8    4.5     2.25    .3      0      .01     .1       1.1      .2
  i8    5       2.25    .3      .1     .01     .1       1.1      .1
e
```

Toot 9: Global Stereo Reverb

In the previous example you may have noticed the soundin source being "cut off" at ends of notes, because the reverb was **inside** the instrument itself. It is better to create a companion instrument, a global reverb instrument, to which the source signal can be sent. Let's also make this stereo.

Variables are named cells which store numbers. In Csound, they can be either *local* or *global*, are available continuously, and can be updated at one of four rates—setup, **i**-rate, **k**-rate, or **a**-rate**.**

Local Variables (which begin with the letters **p**, **i**, **k**, or **a**) are private to a particular instrument. They cannot be read from, or written to, by any other instrument.

Global Variables are cells which are accessible by all instruments. Three of the same four variable types are supported (i, k, and a), but these letters are preceded by the letter **g** to identify them as "global." Global variables are used for "broadcasting" general values, for communicating between instruments, and for sending sound from one instrument to another.

The reverb instr99 below receives input from instr9 via the global a-rate variable *garvbsig*. Since instr9 *adds into* this global, several copies of instr9 can do this without losing any data. The addition requires *garvbsig* to be cleared before each k-rate pass through any active instruments. This is accomplished first with an **init** statement in the orchestra header, giving the reverb instrument a higher number than any other (instruments are performed in numerical order), and then clearing *garvbsig* within instr99 once its data has been placed into the reverb.

```
sr          =       18900       ; toot9.orc
kr          =       945
ksmps       =       20
nchnls      =       2           ; stereo
garvbsig    init    0           ; make zero at orch init time

instr 9
  idur      =       p3
  iamp      =       p4
  iskiptime =       p5
  iattack   =       p6
  irelease  =       p7
  ibalance  =       p8          ; panning: 1=left, .5=center, 0=right
  irvbgain  =       p9
  kamp      linen             iamp, iattack, idur, irelease
  asig      soundin           "soundin.aiff", iskiptime
  arampsig  =                 kamp * asig
            outs              arampsig * ibalance,  arampsig * (1 - ibalance)
```

```
  garvbsig    =           garvbsig + arampsig * irvbgain
endin
instr 99                                    ; global reverb
  irvbtime    =     p4
  asig        reverb garvbsig,  irvbtime    ; put global signal into reverb
              outs   asig, asig
  garvbsig    =     0                       ; then clear it
endin
```

In the score we turn the global reverb on at time 0 and keep it on until *irvbtime* after the last note.

```
; ins    strt    dur     rvbtime                                 ; toot9.sco
  i99    0       9.85    2.6
; ins    strt    dur     amp   skip    atk     rel     balance(0-1)    rvbsend
  i9     0       1       .5    0       .02     .1      1               .2
  i9     2       2       .5    0       .03     .1      0               .3
  i9     3.5     2.25    .5    0       .9      .1      .5              .1
  i9     4.5     2.25    .5    0       1.2     .1      0               .2
  i9     5       2.25    .5    0       .2      .1      1               .3
e
```

Toot 10: Filtered Noise

The following instrument uses the Csound **rand** unit to produce noise, and a **reson** unit to filter it. The bandwidth of **reson** will be set at **i**-time, but its center frequency will be swept via a **line** unit through a wide range of frequencies during each note. We add reverb as above.

```
nchnls = 2
garvbsig    init    0

instr 10                                    ; toot10.orc
  iattack     =       .01
  irelease    =       .2
  iwhite      =       10000
  idur        =       p3
  iamp        =       p4
  isweepstart =       p5
  isweepend   =       p6
  ibandwidth  =       p7
  ibalance    =       p8                    ; pan: 1 = left, .5 = center, 0 = right
  irvbgain    =       p9
  kamp        linen   iamp, iattack, idur, irelease
  ksweep      line    isweepstart, idur, isweepend
  asig        rand    iwhite
  afilt       reson   asig, ksweep, ibandwidth
  arampsig    =       kamp * afilt
              outs    arampsig * ibalance, arampsig * (1 - ibalance)
  garvbsig    =       garvbsig  +  arampsig * irvbgain
endin

instr 100
  irvbtime    =       p4
  asig        reverb  garvbsig,  irvbtime
              outs    asig, asig
```

```
        garvbsig    =       0
        endin

; ins    strt    dur     rvbtime                         ; toot10.sco
  i100  0       15      1.1
  i100  15      10      5

; ins    strt    dur     amp     stswp   ndswp bndwth  balance(0-1)  rvbsend
  i10   0       2       .05     5000    500     20      .5            .1
  i10   3       1       .05     1500    5000    30      .5            .1
  i10   5       2       .05     850     1100    40      .5            .1
  i10   8       2       .05     1100    8000    50      .5            .1
  i10   8       .5      .05     5000    1000    30      .5            .2
  i10   9       .5      .05     1000    8000    40      .5            .1
  i10   11      .5      .05     500     2100    50      .4            .2
  i10   12      .5      .05     2100    1220    75      .6            .1
  i10   13      .5      .05     1700    3500    100     .5            .2
  i10   15      5       .01     8000    800     60      .5            .15
e
```

Toot 11: Carry, Tempo & Sort

We now use a plucked string instrument to explore some of Csound's score preprocessing capabilities. Since the focus here is on the score, the instrument is presented without explanation.

```
        instr 11
                asig1   pluck   ampdb(p4)/2, p5, p5, 0, 1
                asig2   pluck   ampdb(p4)/2, p5 * 1.003,  p5 * 1.003, 0, 1
                        out     asig1+asig2
        endin
```

The score can be divided into time-ordered sections by the **S statement**. Prior to performance, each section is processed by three routines: Carry, Tempo, and Sort. The score *toot11.sco* has multiple sections containing each of the examples below, in both of the forms listed.

The **Carry** feature allows a dot ("**.**") in a p-field to indicate that the value is the same as above, provided the instrument is the same. Thus the following two examples are identical:

```
; ins   start   dur     amp     freq            |       ; ins   start   dur     amp     freq
  i11   0       1       90      200             |         i11   0       1       90      200
  i11   1       .       .       300             |         i11   1       1       90      300
  i11   2       .       .       400             |         i11   2       1       90      400
```

A special form of the carry feature applies to p2 only. A **"+"** in p2 will be given the value of p2+p3 from the previous **i** statement. The "+" can also be carried with a dot:

```
; ins   start   dur     amp     freq            |       ; ins   start   dur     amp     freq
  i11   0       1       90      200             |         i11   0       1       90      200
  i .   +       .       .       300             |         i11   1       1       90      300
  i .   .       .       .       500             |         i11   2       1       90      500
```

The carrying dot may be omitted when there are no more explicit pfields on that line:

```
; ins   start   dur     amp     freq            |       ; ins   start   dur     amp     freq
```

```
i11   0     1     90    200   |     i11   0     1     90    200
i11   +     2                 |     i11   1     2     90    200
i11                           |     i11   3     2     90    200
```

A variant of the carry feature is **Ramping,** which substitutes a sequence of linearly interpolated values for a ramp symbol (<) spanning any two values of a pfield. Ramps work only on consecutive calls to the same instrument, and they cannot be applied to the first three p-fields.

```
; ins    start  dur   amp   freq  |     ; ins   start  dur   amp   freq
 i11    0      1     90    200   |      i11    0      1     90    200
 i .    +      .     <     <     |      i11    1      1     85    300
 i .    .      .     <     400   |      i11    2      1     80    400
 i .    .      .     <     <     |      i11    3      1     75    300
 i .    .      4     70    200   |      i11    4      4     70    200
```

Tempo. The unit of time in a Csound score is the beat—normally one beat per second. This can be modified by a **Tempo Statement,** which enables the score to be arbitrarily time-warped. Beats are converted to their equivalent in seconds during score pre-processing of each Section. In the absence of a Tempo statement in any Section, the following tempo statement is inserted:

t 0 60

It means that at beat 0 the tempo of the Csound beat is 60 (1 beat per second). To hear the Section at twice the speed, we have two options: 1) cut all p2 and p3 in half and adjust the start times, or 2) insert the statement **t 0 120** within the Section**.**

The Tempo statement can also be used to move between different tempi during the score, thus enabling ritardandi and accelerandi. Changes are linear by beat size (see the Csound manual). The following statement will cause the score to begin at tempo 120, slow to tempo 80 by beat 4, then accelerate to 220 by beat 7:

t 0 120 4 80 7 220

The following will produce identical soundfiles:

```
                                                    t 0 120    ; Double-time via Tempo
; ins   start  dur   amp   freq  |     ; ins   start  dur   amp   freq
 i11   0      .5    90    200   |      i11    0      1     90    200
 i .   +      .     <     <     |      i .    +      .     <     <
 i .   .      .     <     400   |      i .    .      .     <     400
 i .   .      .     <     <     |      i .    .      .     <     <
 i .   .      2     70    200   |      i .    .      4     70    200
```

The following includes an accelerando and ritard. It should be noted, however, that the ramping feature is applied *after* time-warping, and is thus proportional to elapsed chronological time. While this is perfect for amplitude ramps, frequency ramps will not result in harmonically related pitches during tempo changes. The frequencies needed here are thus made explicit.

```
        t   0   60    4     400   8     60    ; Time-warping via Tempo
;       ins     start dur   amp   freq
        i11     0     1     70    200
        i .     +     .     <     500
        i .     .     .     90    800
        i .     .     .     <     500
        i .     .     .     70    200
        i .     .     .     90    1000
        i .     .     .     <     600
```

```
i .     .       .       70      200
i .     .       8       90      100
```

Three additional score features are extremely useful in Csound. The **s statement** was used above to divide a score into Sections for individual pre-processing. Since each **s** statement establishes a new relative time of 0, and all actions within a section are relative to that, it is convenient to develop the score one section at a time, then link the sections into a whole later.

Suppose we wish to combine the six above examples (call them *toot11a - toot11f*) into one score. One way is to start with *toot11a.sco*, calculate its total duration and add that value to every starting time of *toot11b.sco*, then add the composite duration to the start times of *toot11c.sco*, etc. Alternatively, we could insert an **s** statement between each of the sections and run the entire score. The file *toot11.sco*, which contains a sequence of all of the above score examples, did just that.

The **f0 statement**, which creates an "action time" with no associated action, is useful in extending the duration of a section. Two seconds of silence are added to the first two sections below.

```
;       ins     start   dur     amp     freq            ; toot11g.sco
        i11     0       2       90      100
        f 0     4                                       ; The f0 Statement
        s                                               ; The Section Statement
        i11     0       1       90      800
        i .     +       .       .       400
        i .     .       .       .       100
        f 0     5
        s
        i11     0       4       90      50
        e
```

Sort. During preprocessing of a score section, all action-time statements are sorted into chronological order by p2 value. This means that notes can be entered in any order, that you can merge files, or work on instruments as temporarily separate sections, then have them sorted automatically when you run Csound on the file.

The file below contains excerpts from this section of the rehearsal chapter and from instr6 of the tutorial, and combines them as follows:

```
;       ins     start   dur     amp     freq            ; toot11h.sco
        i11     0       1       70      100             ; Score Sorting
        i .     +       .       <       <
        i .     .       .       <       <
        i .     .       .       90      800
        i .     .       .       <       <
        i .     .       .       <       <
        i .     .       .       70      100
        i .     .       .       90      1000
        i .     .       .       <       <
        i .     .       .       <       <
        i .     .       .       <       <
        i .     .       .       70      <
        i .     .       8       90      50
```

```
f1 0 2048 10 1                                          ; Sine
f2 0 2048 10 1 .5 .3 .25 .2 .167 .14 .125 .111          ; Sawtooth
f3 0 2048 10 1 0  .3  0   .2  0 .14  0   .111           ; Square
f4 0 2048 10 1 1 1 1 .7 .5 .3 .1                        ; Pulse

; ins   strt    dur     amp     frq     atk     rel     vibrt   vibdpth  vibdel  waveform
  i6    0       2       86      9.00    .03     .1      6       5        .4      1
  i6    2       2       86      9.02    .03     .1      6       5        .4      2
  i6    4       2       86      9.04    .03     .1      6       5        .4      3
  i6    6       4       86      9.05    .05     .1      6       5        .4      4
```

Toot 12: Tables & Labels

This is by far our most complex instrument. In it we have designed the ability to store pitches in a table and then index them in three different ways: 1) directly, 2) via an lfo, and 3) randomly. As a means of switching between these three methods, we will use Csound's *program control* statements and *logical* and *conditional* operations.

```
                instr 12
                  iseed    =    p8
                  iamp     =    ampdb(p4)
                  kdirect  =    p5
                  imeth    =    p6
                  ilforate =    p7                    ; lfo and random index rate
                  itab     =    2
                  itablesize =  8

                        if (imeth == 1)   igoto direct
                        if (imeth == 2)   kgoto lfo
                        if (imeth == 3)   kgoto random

        direct:   kpitch   table    kdirect, itab               ; index "f2" via p5
                           kgoto    contin

        lfo:      kindex   phasor   ilforate
                  kpitch   table    kindex  *  itablesize, itab
                           kgoto    contin

        random: kindex     randh    int(7), ilforate, iseed
                  kpitch   table    abs(kindex), itab

        contin:   kamp     linseg   0, p3 * .1, iamp, p3 * .9, 0  ; amp envelope
                  asig     oscil    kamp, cpspch(kpitch), 1      ; audio osc
                           out      asig
                endin

  f1 0 2048 10 1                                                  ; Sine
  f2 0 8 -2  8.00 8.02 8.04 8.05 8.07 8.09 8.11 9.00   ; cpspch C major scale

; method 1 - direct index of table values
; ins  start     dur     amp     index  method lforate rndseed
  i12   0       .5      86      7       1       0       0
  i12   .5      .5      86      6       1       0
```

```
  i12   1       .5      86      5       1       0
  i12   1.5     .5      86      4       1       0
  i12   2       .5      86      3       1       0
  i12   2.5     .5      86      2       1       0
  i12   3       .5      86      1       1       0
  i12   3.5     .5      86      0       1       0
  i12   4       .5      86      0       1       0
  i12   4.5     .5      86      2       1       0
  i12   5       .5      86      4       1       0
  i12   5.5     2.5     86      7       1       0
s

; method 2 - lfo index of table values
; ins   start   dur     amp     index   method lforate rndseed
  i12   0       2       86      0       2       1       0
  i12   3       2       86      0       2       2
  i12   6       2       86      0       2       4
  i12   9       2       86      0       2       8
  i12   12      2       86      0       2       16
s

; method 3 - random index of table values
; ins   start   dur     amp     index   method rndrate  rndseed
  i12   0       2       86      0       3       2       .1
  i12   3       2       86      0       3       3       .2
  i12   6       2       86      0       3       4       .3
  i12   9       2       86      0       3       7       .4
  i12   12      2       86      0       3       11      .5
  i12   15      2       86      0       3       18      .6
  i12   18      2       86      0       3       29      .7
  i12   21      2       86      0       3       47      .8
  i12   24      2       86      0       3       76      .9
  i12   27      2       86      0       3       123     .9
  i12   30      5       86      0       3       199     .1
```

Toot 13: Spectral Fusion

For our final instrument, we will employ three unique synthesis methods—Physical Modeling, Formant-Wave Synthesis, and Non-linear Distortion. Three of Csound's most powerful unit generators—**pluck**, **fof**, and **foscil**, make this complex task a fairly simple one. The Reference Manual describes these as follows:

a1 **pluck** kamp, kcps, icps, ifn, imeth [, iparm1, iparm2]

pluck simulates the sound of naturally decaying plucked strings by filling a cyclic decay buffer with noise and then smoothing it over time according to one of several methods. The unit is based on the Karplus-Strong algorithm.

a2 **fof** xamp, xfund, xform, koct, kband, kris, kdur kdec,
iolaps, ifna, ifnb, itotdur[, iphs][, ifmode]

fof simulates the sound of the male voice by producing a set of harmonically related partials (a formant region) whose spectral envelope can be controlled over time. It is a special form of granular synthesis, based on the CHANT program from IRCAM by Xavier Rodet et al.

a1 **foscil** xamp, kcps, kcar, kmod, kndx, ifn [, iphs]

foscil is a composite unit which banks two oscillators in a simple FM configuration, wherein the audio-rate output of one (the "modulator") is used to modulate the frequency input of another (the "carrier.")

The plan for our instrument is to have the plucked string attack dissolve into an FM sustain which transforms into a vocal release. The orchestra and score are as follows:

```
        instr 13                                ; toot13.orc
          iamp          = ampdb(p4) / 2         ; amplitude, scaled for two sources
          ipluckamp     = p6                    ; % of total amp, 1=dB amp as in p4
          ipluckdur     = p7*p3                 ; % of total dur, 1=entire dur of note
          ipluckoff     = p3 - ipluckdur
          ifmamp        =  p8                   ; % of total amp, 1=dB amp as in p4
          ifmrise       = p9*p3                 ; % of total dur, 1=entire dur of note
          ifmdec        = p10*p3                ; % of total duration
          ifmoff        = p3 - (ifmrise + ifmdec)
          index         = p11
          ivibdepth     = p12
          ivibrate      = p13
          iformantamp = p14                     ; % of total amp, 1=dB amp as in p4
          iformantrise = p15*p3                 ; % of total dur, 1=entire dur of note
          iformantdec = p3 - iformantrise

          kpluck        linseg  ipluckamp, ipluckdur, 0, ipluckoff, 0
          apluck1       pluck   iamp, p5, p5, 0, 1
          apluck2       pluck   iamp, p5*1.003, p5*1.003, 0, 1
          apluck        =       kpluck * (apluck1+apluck2)

          kfm           linseg  0, ifmrise, ifmamp, ifmdec, 0, ifmoff, 0
          kndx          =       kfm * index
          afm1          foscil  iamp, p5, 1, 2, kndx, 1
          afm2          foscil  iamp, p5*1.003, 1.003, 2.003, kndx, 1
          afm           =       kfm * (afm1+afm2)

          kfrmnt        linseg  0, iformantrise, iformantamp, iformantdec, 0
          kvib          oscil   ivibdepth,ivibrate,1
          afrmnt1       fof   iamp, p5+kvib, 650, 0, 40, .003, .017, .007, 4, 1, 2, p3
          afrmnt2       fof   iamp, (p5*1.001)+kvib*.009, 650, 0, 40, .003,.017,.007, 10,1,2,p3
          aformnt       =       kfrmnt * (afrmnt1+afrmnt2)

                        out     apluck + afm + aformnt
        endin

f1  0  8192  10  1                      ; sine wave
f2  0  2048   19  .5  1  270  1         ; sigmoid rise

;ins    st   dr   mp     frq     plkmp  plkdr   fmp    fmris   fmdec  indx    vbdp    vbrt    frmp fris
 i13    0    5    80     200     .8     .3      .7     .2      .35    8       1       5       3    .5
 i13    +    8    80     100     .      .4      .7     .35     .35    7       1       6       3    .7
 i13    .   13    80     50      .      .3      .7     .2      .4     6       1       4       3    .6
```

When Things Sound Wrong

When you design your own Csound instruments you may occasionally be surprised by the results. There will be times when you've computed a file for hours and your playback is just silence, while at other times you may get error messages which prevent the score from running, or you may hang the computer and nothing happens at all.

In general, Csound has a comprehensive error-checking facility that reports to your console at various stages of your run: at score sorting, orchestra translation, initializing each call of every instrument, and during performance. However, if your error was syntactically permissable, or it generated only a warning message, Csound could faithfully give you results you don't expect. Here is a list of the things you might check in your score and orchestra files:

1. You typed the letter **l** instead of the number **1**
2. You forgot to precede your comment with a semi-colon
3. You forgot an opcode or a required parameter
4. Your amplitudes are not loud enough or they are too loud
5. Your frequencies are not in the audio range - 20Hz to 20kHz
6. You placed the value of one parameter in the p-field of another
7. You left out some crucial information like a function definition
8. You didn't meet the Gen specifications

Suggestions for Further Study

Csound is such a powerful tool that we have touched on only a few of its many features and uses. You are encouraged to take apart the instruments in this chapter, rebuild them, modify them, and integrate the features of one into the design of another. To understand their capabilities you should compose short etudes with each. You may be surprised to find yourself merging these little studies into the fabric of your first Csound compositions.

The directory 'morefiles' contains examples of the classical designs of Risset and Chowning. Detailed discussions of these instruments can be found in Charles Dodge's and Thomas Jerse's **Computer Music** textbook. This text is the key to getting the most out of these instrumental models and their innovative approaches to signal processing. Also recommended are the designs of Russell Pinkston. They demonstrate techniques for legato phrasing, portamento, random vibrato, and random sequence generation. His instrument representing Dx7 OpCode™ Editor/Librarian patches is a model for bringing many wonderful sounds into your orchestra.

Nothing will increase your understanding more than actually Making Music with Csound. The best way to discover the full capability of these tools is to create your own music with them. As you negotiate the new and uncharted terrain you will make many discoveries. It is my hope that through Csound you discover as much about music as I have, and that this experience brings you great personal satisfaction and joy.

Richard Boulanger - March 1991 - Boston, Massachusetts - USA

Appendix 4: An FOF Synthesis Tutorial

by
J.M.Clarke
University of Huddersfield

The **fof** synthesis generator in Csound has more parameter fields than other modules. To help the user become familiar with these parameters this tutorial will take a simple orchestra file using just one **fof** unit-generator and demonstrate the effect of each parameter in turn. To produce a good vocal imitation, or a sound of similar sophistication, an orchestra containing five or more **fof** generators is required and other refinements (use of random variation of pitch etc.) must be made. The sounds produced in these initial explorations will be much simpler and consequently less interesting but they will help to show clearly the basic elements of **fof** synthesis. This tutorial assumes a basic working knowledge of Csound itself. The specification of the **fof** unit-generator (as found in the main Csound manual) is:

ar fof xamp xfund xform koct kband kris kdur kdec iolaps ifna ifnb itotdur [iphs] [ifmode]

where	xamp, xfund, xform	can receive any rate (constant, control or audio)
	koct, kband, kdris, kdur, kdec	can receive only constants or control rates
	iolaps, ifna, ifnb, itotdur	must be given a fixed value at initialization
	[iphs][ifmode]	are optional, defaulting to 0.

The following orchestra contains a simple instrument we will use for exploring each parameter in turn. On the faster machines (DECstation, SparcStation, SGI Indigo) it will run in real time.

```
        sr = 22050
        kr = 441
        ksmps = 50

        instr   1
a1      fof     15000, 200, 650, 0, 40, .003, .02, .007, 5, 1, 2, p3
        out     a1
        endin
```

It should be run with the following score:

```
f1  0  4096  10  1
f2  0  1024  19  .5  .5  270  .5
i1  0  3
e
```

The result is very basic. This is not surprising since we have created only one formant region (a vocal imitation would need at least five) and have no vibrato or random variation of the parameters. By varying one parameter at a time we will help the reader learn how the unit-generator works. Each of the following "variations" starts from the model. Parameters not specified remain as given.

xamp = amplitude

The first input parameter controls the amplitude of the generator. At present our model uses a constant amplitude, this can be changed so that the amplitude varies according to a line function:

```
a2      linseg  0,  p3*.3,  20000,  p3*.4,  15000,  p3*.3,  0
a1      fof     a2, .....(as before)...
```

The amplitude of a **fof** generator needs care. *xamp* does not necessarily indicate the maximum output, which can also depend on the rise pattern, bandwidth, and the presence of any "overlaps".

xfund = fundamental frequency

This parameter controls the pitch of the fundamental of the unit generator. Starting again from the original model this example demonstrates an exaggerated vibrato:

```
a2      oscil   20, 5, 1
a1      fof     15000, 200+a2, etc.....
```

fof synthesis produces a rapid succession of (normally) overlapping excitations or granules. The fundamental is in fact the speed at which new excitations are formed and if the fundamental is very low these excitations are heard as separate granules. In this case the fundamental is not so much a pitch as a pulse speed. The possibility of moving between pitch and pulse, between timbre and granular texture is one of the most interesting aspects of **fof**. For a simple demonstration try the following variation. It will be especially clear if the score note is lengthened to about 10 seconds.

```
a2      expseg  5, p3*.8, 200, p3*.2, 150
a1      fof     15000, a2 etc.....
```

koct = octaviation coefficient

Skipping a parameter, we come to an unusual means of controlling the fundamental: *octaviation*. This parameter is normally set to 0. For each unit increase in *koct* the fundamental pitch will drop by one octave. The change of pitch is **not** by the normal means of glissando, but by gradually fading out alternate excitations (leaving half the original number). Try the following (again with the longer note duration):

```
k1      linseg  0, p3*.1, 0, p3*.8, 6, p3*.1, 6
a1      fof     15000, 200, 650, k1 etc.....
```

This produces a drop of six octaves; if the note is sufficiently long you should be able to hear the fading out of alternate excitations towards the end.

xform = formant frequency; *ifmode* = formant mode (0 = striated, non-0 = smooth)

The spectral output of a **fof** unit-generator resembles that of an impulse generator filtered by a band pass filter. It is a set of partials above a fundamental *xfund* with a spectral peak at the formant frequency *xform*. Motion of the formant can be implemented in two ways. If *ifmode* = 0, data sent to *xform* has effect only at the start of a new excitation. That is, each excitation gets the current value of this parameter at the time of creation and holds it until the excitation ends. Successive overlapping excitations can have different formant frequencies, creating a richly varied sound. This is the mode of the original CHANT program. If *ifmode* is non-zero, the frequency of each excitation varies continuously with *xform*. This allows glissandi of the formant frequency. To demonstrate these differences we take a very low fundamental so that the granules can be heard separately and the formant frequency is audible not as the center frequency of a "band" but as a pitch in its own right. Compare the following in which only *ifmode* is changed:

```
a2      line    400, p3, 800
a1      fof     15000, 5, a2, 0, 1, .003, .5, .1, 3, 1, 2, p3, 0, 0

a2      line    400, p3, 800
a1      fof     15000, 5, a2, 0, 1, .003, .5, .1, 3, 1, 2, p3, 0, 1
```

In the first case the formant frequency moves by step at the start of each excitation, whereas in the second it changes smoothly. A more subtle difference is perceived with higher fundamental frequencies. (Note that the later **fof** parameters were changed in this example to lengthen the excitations so that their pitch could be heard easily.)

xform also permits frequency modulation of the formant frequency. Applying FM to an already complex sound can lead to strange results, but here is a simple example:

```
acarr   line    400, p3, 800
index   =       2.0
imodfr  =       400
idev    =       index * imodfr
amodsig oscil   idev, imodfr, 1
a1      fof     15000, 5, acarr+amodsig, 0, 1, .003, .5, .1, 3, 1, 2, p3, 0, 1
```

kband = formant bandwidth
kris, kdur, kdec = risetime, duration and decaytime (in seconds) of the excitation envelope

These parameters control the shape and length of **fof** granules. They are shaped in three segments: a rise, a middle decay, and a terminating decay. For very low fundamentals these are perceived as an amplitude envelope, but with higher fundamentals (above 30 Hz) the granules merge together and these parameters effect the timbre of the sound. Note that these four parameters influence a new granule only at the time of its initialization and are fixed for its duration; later changes will affect only subsequent granules. We begin our examination with low frequencies.

```
k1      line    .003, p3, .1                                    ; kris
a1      fof     15000, 2, 300, 0, 0, k1, .5, .1, 2, 1, 2, p3
```

Run this with a note length of 10 seconds. Notice how the attack of the envelope of the granules lengthens. The shape of this attack is determined by the forward shape of *ifnb* (here a sigmoid).

Now try changing kband:

```
k1      linseg  0, p3, 10                                      ; kband
a1      fof     15000, 2, 300, 0, k1, .003, .5, .1, 2, 1, 2, p3
```

Following its rise, an excitation has a built-in exponential decay and *kband* determines its rate. The bigger *kband* the steeper the decay; zero means no decay. In the above example the successive granules had increasingly fast decays.

```
k1      linseg  .3, p3, .003
a1      fof     15000, 2, 300, 0, 0, .003, .4, k1, 2, 1, 2, p3
```

This demonstrates the operation of *kdec*. Because an exponential decay never reaches zero it must be terminated gracefully. *kdur* is the overall duration (in seconds from the start of the excitation), and *kdec* is the length of the terminating decay. In the above example the

terminating decay starts very early in the first granules and then becomes progressively later. Note that *kband* is set to zero so that only the terminating decay is evident.

In the next example the start time of the termination remains constant, but its length gets shorter:

```
k1      expon  .3,  p3,  .003
a1      fof    15000,  2,  300,  0,  0,  .003,  .01 + k1,  k1,  2,  1,  2,  p3
```

It may be surprising to find that for higher fundamentals the local envelope determines the spectral shape of the sound. Electronic and computer music has often shown how features of music we normally consider independent (such as pitch, timbre, rhythm) are in fact different aspects of the same thing. In general, the longer the local envelope segment the narrower the band of partials around that frequency. *kband* determines the bandwidth of the formant region at -6dB, and *kris* controls the skirtwidth at -40dB. Increasing *kband* increases the local envelope's exponential decay rate, thus shortening it and increasing the -6dbB spectral region. Increasing *kris* (the envelope attack time) inversely makes the -40dB spectral region smaller.

The next example changes first the bandwidth then the skirtwidth. You should be able to hear the difference.

```
k1      linseg  100,  p3/4,  0,  p3/4,  100,  p3/2,  100                ; kband
k2      linseg  .003,  p3/2,  .003,  p3/4,  .01,  p3/4,  .003                    ; kris
a1      fof     15000,  100,  440,  0,  k1,  k2,  .02,  .007,  3,  1,  2,  p3
```

[In the first half of the note *kris* remains constant while *kband* broadens then narrows again. In the second half, *kband* is fixed while *kris* lengthens (narrowing the spectrum) then returns again .]

Note that *kdur* and *kdec* don't really shape the spectrum, they simply tidy up the decay so as to prevent unwanted discontinuities which would distort the sound. For vocal imitations these parameters are typically set at .017 and .007 and left unchanged. With high ("soprano") fundamentals it is possible to shorten these values and save computation time (reduce overlaps).

iolaps = number of overlap spaces

Granules are created at the rate of the fundamental frequency, and new granules are often created before earlier ones have finished, resulting in overlaps. The number of overlaps at any one time is given by *xfund* * *kdur*. For a typical bass note the calculation might be 200 * .018 = 3.6, and for a soprano note 660 * .015 = 9.9. **fof** needs at least this number (rounded up) of spaces in which to operate. The number can be over-estimated at no computation cost, and at only a small space cost. If there are insufficient overlap spaces during operation, the note will terminate.

ifna, *ifnb* = stored function tables

Identification numbers of two function tables (see the **fof** entry in the manual proper).

itotdur = total duration within which all granules in a note must be completed

So that incomplete granules are not cut off at the end of a note **fof** will not create new granules if they will not be completed by the time specified. Normally given the value "p3" (the note length), this parameter can be changed for special effect; **fof** will output zero after time *itotdur*.

iphs = initial phase (optional, defaulting to 0).

Specifies the initial phase of the fundamental. Normally zero, but giving different **fof** generators different initial phases can be helpful in avoiding "zeros" in the spectrum.

Appendix 5: A CSOUND QUICK REFERENCE

VALUE CONVERTERS

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

PITCH CONVERTERS

octnot(pch)*	(init- or control-rate arg only)
cpsnot(pch)*	" "
octpch(pch)	" "
pchoct(oct)	" "
cpspch(pch)	" "
octcps(cps)	" "
cpsoct(oct)	(no rate restriction)

GENERAL 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
reinit	label
rigoto	label
rireturn	
turnon	insno[, itime]
ihold	
turnoff	
maxalloc*	imax[, imethod]

STRING SET, PRESET and PROGRAM SET VARIABLES

strset	ndx, string
pset	con1, con2, ...
vset*	con1, con2, ...
gvset*	con1, con2, ...
vdim*	con1
pgminit*	pgm.bn, ival1[, ival2, ...ival150]
dpgminit*	dpgm.ch, ival1[, ival2, ...ival150]
vprogs*	pgmno1[, pgmno2, ...pgmno32]
dvprogs*	dpgmno1[, dpgmno2, ...dpgmno32]
dpkeys*	dpgm.ch, ikey1, ikey2, ...
dpexclus*	dpgm.ch, ikey1, ikey2, ...
inexclus*	ctrlno, ins1, ins2, ins3, ...
autopgms*	
auto_sf2*	
auto_dls*	

MIDI CONTROLLERS and CONVERTERS

	massign	ichnl, insno
	ctrlinit	ichnl, ictlno1, ival1[, ctrlno2, ival2[, ictlno3, ... ival32]]
	pctrlinit*	pgm.bn, ictlno1, ival1[, ictlno2, ival2[, ictlno3, ... ival32]]
	dpctrlinit*	dpgm.ch, ictlno1, ival1[, ictlno2, ival2[, ictlno3, ... ival32]]
	uctrlmap*	ictlno[, ilow, ihigh]
	dsctrlmap*	iparm[, ilow, ihigh]
ival	**midictrl**	ictlno[, ilow, ihigh]
kval	**midictrl**	ictlno[, ilow, ihigh]
ival	**chanctrl**	ichnl, ictlno[, ilow, ihigh]
kval	**chanctrl**	ichnl, ictlno[, ilow, ihigh]
ival	**dsctrl***	iparm
ival	**miditran**	idist
ival	**veloc**	[ilow, ihigh]
inot	**notnum**	
inot	**notnumb**	[isens]
knot	**notnumb**	[isens]
icps	**cpsmidi**	
icps	**cpsmidib**	[isens]
kcps	**cpsmidib**	[isens]
ioct	**octmidi**	
ioct	**octmidib**	[isens]
koct	**octmidib**	[isens]
ipch	**pchmidi**	
ipch	**pchmidib**	[isens]
kpch	**pchmidib**	[isens]
iamp	**ampmidi**	iscal[, ifn]
kaft	**polyaft**	[ilow, ihigh]
kaft	**aftouch**	[ilow, ihigh]
ibnd	**pchbend**	[ilow, ihigh]
kbnd	**pchbend**	[ilow, ihigh]
	veloffs*	

VALUE SELECTORS and SPLITTERS

ival	**mselect***	iscal, index, ival0, ival1[, ival2, … ival15]
kval	**mselect***	kscal, index, kval0, kval1[, kval2, … kval15]
id	**ftsplit***	isets, jsize, i1j1, i1j2, ..,i2j1,i2j2, ..,i3j1, ..[,isets2, jsize2, …]
i1,i2,i3,...i10	**mtsplit***	[id, ilay]

FTABLES

iafno	**ftgen**	ifno, itime, isize, igen, iarga[, iargb, ...iargz]
iafno	**ftload***	ifilnam[, iskiptime, iformat, ichnl, inorm]
iafno	**ftstep***	ix1, ia, ix2, ib[, ix3, ic, ...], ixn
	ftscale*	ifno, iscale

MACROS

macro*	name
endm*	

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	**linsegr**	ia, idur1, ib[, idur2, ic[...]], irel, iz
ar	**linsegr**	ia, idur1, ib[, idur2, icI...]], irel, iz
kr	**linseg4***	ia, idur1, ib, idur2, ic, idur3, id, idur4, ie [, imode, itmap, ilvlvel, iatvel, iktrk]
kr	**linseg4r***	ia, idur1, ib, idur2, ic, idur3, id, idur4, ie, irel, iz[, irind, imode, itmap, ilvlvel, iatvel, iktrk, irelmod]
kr	**expseg**	ia, idur1, ib[, idur2, ic[...]]
ar	**expseg**	ia, idur1, ib[, idur2, ic[...]]
kr	**expsegr**	ia, idur1, ib[, idur2, ic[...]], irel, iz
ar	**expsegr**	ia, idur1, ib[, idur2, icI...]], irel, iz
kr	**dexponr***	ival, idel, idecrat, irel
ar	**dexponr***	ival, idel, idecrat, irel
kr	**adsr***	iris, idec, isus, irel, ibas, ipeak[, iconv][, irind][, ioff]
ar	**adsr***	iris, idec, isus, irel, ibas, ipeak[, iconv][, irind][, ioff]
kr	**dahdsr***	idel, iris, ihold, idec, isus, irel, ibas, ipeak[, iconv] [, irind][, ioff]
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]

ir	**dtable***	indx, ifn
kr	**oscil1**	idel, kamp, idur, ifn
kr	**oscil1i**	idel, kamp, idur, ifn
ar	**oscil1**	idel, xamp, idur, ifn
ar	**oscil1i**	idel, xamp, idur, ifn
ar	**osciln**	kamp, ifrq, ifn, itimes
kr	**lfo***	kcps, ifn[, idel, iphs]
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	**coscil***	xamp, kcps, kcents, ifn[, iphs]
ar1 [,ar2]	**loscil**	xamp, kcps, ifn[, ibas][, imod1,ibeg1,iend1] [, imod2,ibeg2,iend2]
ar	**doscil***	xamp, ifn[, iautoff]
ar	**doscilp***	kamp, kcps, ifno[, ibas, iautoff]
ar	**loscil1***	kamp, kcps, ifno[, ibas]
a1,a2	**loscil2***	kamp, kcps, ifno[, ibas]
ar	**poscil***	kamp, kcps, kfrac[, iphs]
ar	**buzz**	xamp, xcps, knh, ifn[, iphs]
ar	**gbuzz**	xamp, xcps, knh, kih, kr, ifn[, iphs]
ar	**adsyn**	kamod, kfmod, ksmod, ifilcod
ar	**pvoc**	ktimpnt, kfmod, ifilcod[, ispecwp]
ar	**fof**	xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur[, iphs][, ifmode]
ar	**harmon**	asig, kestfrq, kmaxvar, kgenfreq1, kgenfreq2, imode, iminfrq, iprd
ar	**harmon2***	asig, koct, kfrq1, kfrq2, icpsmode, ilowest
ar	**harmon3***	asig, koct, kfrq1, kfrq2, kfrq3, icpsmode, ilowest
ar	**harmon4***	asig, koct, kfrq1, kfrq2, kfrq3, kfrq4, icpsmode, ilowest
ar	**grain**	xamp, xcps, xdens, kampdev, kcpsdev, kgdur, igfn, iwfn, imaxdur
ar	**pluck**	kamp, kcps, icps, ifn, imeth [, iparm1, iparm2]
ar	**pluck2***	ifrq, iamp, ipickup, ipluck, iaw0, iawPi, aamp
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]
xr	**linrand**	krange[, ipol]
xr	**exprand**	krange[, ipol]
xr	**cauchy**	kalpha[, ipol]
xr	**poisson**	klambda
xr	**gauss**	krange
xr	**weibull**	ksigma, ktau
xr	**beta**	krange, kalpha, kbeta

SIGNAL MODIFIERS

kr	**linen**	kamp, irise, idur, idec
ar	**linen**	xamp, irise, idur, idec
kr	**linenr**	kamp, irise, idec[, irind]
ar	**linenr**	xamp, irise, idec[, irind]
kr	**envlpx**	kamp, irise, idur, idec, ifn, iatss, iatdec[, ixmod]
ar	**envlpx**	xamp, irise, idur, idec, ifn, iatss, iatdec[, ixmod]
kr	**envlpxr**	kamp, irise, idec, ifn, iatss, iatdec[, ixmod]
ar	**envlpxr**	xamp, irise, 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]
ar	**filter1***	asig, kfreq, kdamp, imode[, istor]
ar	**filter2***	asig, kfreq, kdamp, imode[, istor]
ar	**filter4***	asig, kfc1, kfc2, imode[, istor]
ar	**butterhp**	asig, kcps[, istor]
ar	**butterlp**	asig, kcps[, istor]
ar	**butterbp**	asig, kcf, kbw[, istor]
ar	**butterbr**	asig, kcf, kbw[, istor]
krmsr,krmso,kerr,kcps	**lpread**	ktimpnt, ifilcod[, inpoles][, ifrmrate**]**
ar	**lpreson**	asig
ar	**lpfreson**	asig, kfrqratio
ar	**cross***	acar, amod, kdepth, kvol, ilocut, ihicut
kr	**rms**	asig[, ihp, istor]
nr	**gain**	asig, krms[, ihp, istor]
ar	**balance**	asig, acomp[, ihp, istor]
ar	**dbgain***	asig, kdb[, iatktim, istor]
ar	**compress***	aasig,acsig,kthresh,khiknee,kloknee,kratio,katt,krel,ilook
ar	**distort***	asig, kdist, ifn[, ihp, istor]
kr	**downsamp**	asig[, iwlen]
ar	**upsamp**	ksig
ar	**interp**	ksig[, iatktim, 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	**octup***	asig, isegtim
ar	**octdown***	asig, isegtim
ar	**delayr**	idlt[, istor]
	delayw	asig
ar	**delay**	asig, idlt[, istor]
ar	**delay1**	asig[, istor]

ar	**vdelay**	asig, xdlt, imaxdlt[, istor]
ar	**deltap**	kdlt
ar	**deltapi**	xdlt
ar	**comb**	asig, krvt, ilpt[, istor]
ar	**alpass**	asig, krvt, ilpt[, istor]
ar	**reverb**	asig, krvt[, istor]
ar	**reverb2**	asig, krvt, khfabs[, istor]
a1,a2	**lrghall32***	asig
a1,a2	**lrghall44***	asig
ar	**chorus1***	asig, krat1,krat2,krat3, idel1,idel2,idel3, ipred, idpth, ifdbk, ifsin
a1,a2	**chorus2***	asig, krat1,krat2,krat3,krat4, idel1,idel2,idel3,idel4, ipred, idpth, ifdbk, ifsin
a1,a2	**chorus3***	asig, krat1,krat2,krat3,krat4, idel1,idel2,idel3,idel4, ipred, idpth, ifdbk, ifsin
a1	**flange1***	asig, krate, idel, ifdbk, ifn
a1,a2	**flange2***	asig, krate, idel, ideff, ifdbk, ifn

OPERATIONS USING SPECTRAL DATA TYPES

wsig	**spectrum**	xsig, iprd, iocts, ifrqs, iq[, ihann, idbout, idisprd, idsines]
wsig	**specaddm**	wsig1, wsig2[, imul2]
wsig	**specdiff**	wsigin
wsig	**specscal**	wsigin, ifscale, ifthresh
wsig	**spechist**	wsigin
wsig	**specfilt**	wsigin, ifhtim
koct, kamp	**specptrk**	wsig, kvar, ilo, ihi, istrt, idbthresh, inptls, irolloff[, iodd, iconfs, interp, ifprd, iwtflg]
ksum	**specsum**	wsig[, interp]
	specdisp	wsig, iprd[, iwtflg]

SENSING & CONTROL

	midiout*	kamp, koct, iampsens, ibendrng, ichan
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
aramp	**iftime***	idgrlvl, label[, idec, iris][, iprop]
acum	**timegate***	aramp, anew, acum

SIGNAL INPUT & OUTPUT

a1	**in**	
a1, a2	**ins**	
a1, a2, a3, a4	**inq**	
a1, a2, a3, a4, a5, a6	**inh***	
a1[, a2][, a3, a4]	**soundin**	ifilcod[, iskptim][, iformat]
	out	asig
	outs1	asig
	outs2	asig
	outs	asig1, asig2
	outq1	asig
	outq2	asig
	outq3	asig
	outq4	asig
	outq	asig1, asig2, asig3, asig4
	outh1*	asig
	outh2*	asig
	outh3*	asig
	outh4*	asig
	outh5*	asig
	outh6*	asig
	outh*	asig1, asig2, asig3, asig4, asig5, asig6
	soundout*	asig1, ifilcod[, iformat]
	soundouts*	asig1, asig2, ifilcod[, iformat]
a1, a2	**hostin***	
	hostout*	asig, iprd
ar	**fxsend***	asig, klevel
a1[, a2, a3, a4]	**mfxsend***	asig, ilvl1[, ilvl2, ilvl3, ilvl4]
	outs12*	asig
	panouts*	asig, kprop
a1, a2, a3, a4	**pan**	asig, kx, ky, ifn[, imode][, ioffset]
k1	**kread**	ifilname, iformat, iprd[, interp]
k1, k2	**kread2**	ifilname, iformat, iprd[, interp]
k1, k2, k3	**kread3**	ifilname, iformat, iprd[, interp]
k1, k2, k3, k4	**kread4**	ifilname, iformat, iprd[, interp]
	kdump	ksig, ifilname, iformat, iprd
	kdump2	ksig1, ksig2, ifilname, iformat, iprd
	kdump3	ksig1, ksig2, ksig3, ifilname, iformat, iprd
	kdump4	ksig1, ksig2, ksig3, ksig4, ifilname, iformat, iprd

SIGNAL DISPLAY

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

COSTING

clkon*	id
clkoff*	id

GEN ROUTINES

f	#	time	size	1	filcod skiptime format channel
f	#	time	size	2	v1 v2 v3 . . .
f	#	time	size	3	xval1 xval2 c0 c1 c2 . . . cn
f	#	time	size	4	source # sourcemode
f	#	time	size	5	a n1 b n2 c . . .
f	#	time	size	7	a n1 b n2 c . . .
f	#	time	size	6	a n1 b n2 c n3 d . . .
f	#	time	size	8	a n1 b n2 c n3 d . . .
f	#	time	size	9	pna stra phsa pnb strb phsb . . .
f	#	time	size	10	str1 str2 str3 str4 . . . .
f	#	time	size	19	pna stra phsa dcoa pnb strb phsb dcob . . .
f	#	time	size	11	nh lh r
f	#	time	size	12	xint
f	#	time	size	13	xint xamp h0 h1 h2 . . . hn
f	#	time	size	14	xint xamp h0 h1 h2 . . . hn
f	#	time	size	15	xint xamp h0 phs0 h1 phs1 h2 phs2 . . .
f	#	time	size	17	x1 a x2 b x3 c . . .
f	#	time	size	20	window max opt
f	#	time	size	21	distr range opt1 opt2